Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems

by

Daniel C. Zilio

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Physical Database Design Decision Algorithms and
Concurrent Reorganization for
Parallel Database Systems

Daniel C. Zilio
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1997

Stringent performance requirements in DB applications have led to the use of parallelism for database processing. To allow the database system to take advantage of the performance of parallel shared-nothing systems, the physical DB design must be appropriate for the DB structure and the workload.

We develop decision algorithms that will select a good physical DB design both when the DB is first loaded into the system (static decision) and while the DB is being used by the workload (dynamic decision). Our decision algorithms take the database structure, workload, and system characteristics as inputs. The static (or initial) physical DB design decision algorithm involves:

- selecting a partitioning attribute for each relation that determines how the relation is fragmented across the nodes (allowing for high I/O bandwidth);
- selecting indexes on the relation attributes to allow faster accesses compared to sequential file scans;
- selecting the attributes by which to cluster a relation in order to take advantage of the prefetching and caching involved in I/O access;
- grouping of relations to allow DB operations (joins) on relation pairs to be executed locally at each node, thus reducing communication costs;
- selecting the number of partitions per relation group (and thus per relation); and
- assigning each partition of each relation to a specific system node.

Our studies show that, among the algorithms we studied, an algorithm based on a branch-and-bound strategy finds designs with the lowest estimated workload average response time and requires an acceptable amount of computation.

The physical DB design reorganization problem, which is the dynamic DB design decision, involves determining how to change the current physical DB design based on new DB structure, workload, and system information. If a new physical DB design is chosen, a strategy to move from the old to the new design must also be identified by the algorithm. A formula is developed to calculate both the benefit resulting after a reorganization is complete and the cost of performing the reorganization. The value from this formula for a specific reorganization is called the *(net) gain* metric, and this metric is used by a decision algorithm to compare reorganizations and to select the reorganization for which the benefit most exceeds its cost. We also develop a method to estimate the costs of executing a reorganization with a workload, and we provide some decision algorithms. The selection of the priority level at which to run the reorganization processes concurrently with the workload is investigated. Our studies indicate that a low priority for the reorganization process compared to the priorities for the workload processes is often but not always best.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel shared-nothing (SN) systems have an interconnection network connecting several independent processing nodes. Each of the nodes has its own processor(s), local memory, and disk drive(s). SN systems are useful because:

- they can be constructed with off-the-shelf hardware;
- they are scalable; and
- they have a low price-performance ratio.

Because of these aspects, several database vendors have implemented database (DB) systems on this type of architecture. Such SN DB management systems (DBMS) utilize the parallel resources by distributing the DB data across the nodes and by assigning a DB management process to each processing node. A node's DB management process handles the node's local resources, manages the query operator executions on the node, and synchronizes with the other managers through messages. Thus, these managers coordinate the parallel system nodes to form a single DBMS system.

On a SN system, good DBMS performance in terms of high query throughputs and low response times comes from:

- selecting a good physical DB layout, also called a *physical DB design*;
- optimizing queries to account for the SN DBMS environment; and
- scheduling query executions to use a portion of the available SN resources (e.g., a portion of the available buffer space) so as not to overload the system and to allow other queries to execute concurrently.

Since optimizing and scheduling depend on selecting a good physical DB design as the basis for all other choices, we will address the problem of constructing and selecting physical DB designs. A physical DB design includes:

- partitioning the DB relations over the nodes and disks of the system to allow for parallelism during query executions;
- providing special access structures to some of the relations (indexes); and
- ordering or clustering the partition of a relation at each node.

Parallelism is achievable in a relational SN DBMS because the data and the query functionality can be partitioned across the SN nodes. Three types of parallelism are possible:

- Inter-query parallelism. If a relation is partitioned across different nodes, then queries accessing different parts of the relation can execute simultaneously at different nodes. Inter-query parallelism also results when queries access different relations that are stored at disjoint sets of nodes.
- Intra-query parallelism. This type of parallelism is possible because each query consists of a set of operators, (e.g., selects and joins), some of which can operate in parallel with others. This parallelism is attainable if different operators access different parts of the same relation or different relations across different nodes.
- Intra-operator parallelism. An operator on a partitioned relation can execute in parallel at the nodes storing the relation's partitions.

In addition to the parallelism benefits that result from partitioning, there are various performance gains possible with indexes. First, the I/O costs are reduced by using an index to directly access the required tuples, thus avoiding a full relation scan. This makes sense when only a small number of tuples is required to answer a query, compared to the number of tuples in a relation. Second, the cost of sorting a relation is reduced because an index can be used to retrieve the relation in a pre-sorted order from disk. This order can differ from the way the relation is stored on disk.

Physically ordering the tuples of a relation's partition on disk also gives a benefit. When a relation is stored in the same order in which it needs to be read, it can be sequentially read from disk, and this reduces I/O costs by allowing for prefetching and disk caching.

Our first goal will be to choose a proper physical DB design (*initial physical DB design*) before a DB is loaded onto a parallel SN system. This choice is derived from the *system environment*, which is specified by the definition of the system, DB, and workload structures of a specific system. A physical DB design is specified by binding the following design decisions:

- how to horizontally partition [RE78] each relation, which includes the decisions of which combination of attributes (partitioning key) in the relation determine a relation's partitioning, how many partitions to create for each relation, and which nodes to use to store each relation's partitions;

- what indexes to create on each relation; and
- how to cluster the tuples in the partitions of each relation.

There are many choices available for each decision, leading to a vast number of possibilities for the complete DB design. Thus, a tool is needed to assist the DB administrator (DBA) in determining a physical DB design that yields good performance since previous algorithms only solved these decisions individually.

In developing a physical DB design tool, we could construct the tool to make its decisions:

- exhaustively (considering every possible design);
- arbitrarily (making decisions expeditiously); or
- heuristically (using guidelines to limit the range of solutions explored).

An exhaustive search has the advantage of always finding the design that yields optimal workload performance with respect to the chosen objective function, but it leads to a time-consuming execution of the design selection algorithm. On the other hand, arbitrary decisions can be made quickly, but do not lead to any guarantees on how well the system will perform. However, heuristic methods reduce the number of designs considered compared to an exhaustive approach, and may or may not provide either optimal or close to optimal solutions. In our work, we seek decision tools with acceptable execution times that result in good performance.

The DBA will use our tool in the following way. He/she will collect and input information about the workload, DB, and system to one of our algorithms. Our algorithms also require the use of the DB optimizer from the DBA's system as well as specification of cost formulas related to operator execution for the DBA's system. When given the input information, our algorithms will output a suggested physical DB design and the estimated workload performance the design will provide.

Because a specific system environment given by the DBA is used to create an initial physical DB design, any changes to this environment while the DB is being used may result in poor performance using the previously chosen design. For example, new query classes could be added to the workload, the system may be given more resources, or more relations and attributes could be added to the DB. Better performance could be achieved by using a different design, and thus *reorganization* to a new physical DB design is desirable. Thus, another goal of our work will be to address the problem of changing the physical DB design in response to changes in the environment.

When choosing a reorganization to apply to an existing physical DB design, a DBA must consider the following issues:

- The *benefit* resulting from the reorganization. This benefit is defined to be the difference between the performance (e.g., workload average response time) of a new design after the reorganization is complete and the performance of the old design.

- The *cost* of the reorganization. This cost is defined as the added delay to the workload transactions' executions while a reorganization executes. These delays are caused by the workload competing with the reorganization to use the SN system resources (*resource contention* or *congestion delays*) and to access the relations being moved (*data contention*). We measure the cost of the reorganization as the increase in the workload average response times when the workload executes with the reorganization compared to the times when no reorganization is executed. This cost is calculated over the time it takes to execute the reorganization.

Minimizing the interference of the reorganization with the workload is desirable in current and future DB systems since these systems require that the DB be available to users continuously (24 hours a day, seven days a week [DG92]). To provide for this availability during a reorganization, reorganizations must be designed to execute concurrently (*on-line*) with the normal DB workload while introducing minimal delays.

A new physical DB design should be chosen such that the cost of the reorganization is outweighed by its benefit. To select a design, the goal could be to either find: (i) the design with the best performance, or (ii) any new design that has better performance than the current design's. Selecting any new design with better performance allows a search heuristic to determine and reorganize to a new design quickly and inexpensively compared to always finding the best design.

The trade-off between the performance gain achieved with an improved physical DB design and the cost of performing the reorganization has not been considered extensively in previous work. Some reorganization tools have been developed that support a specific type of reorganization on a single relation. These tools provide different trade-offs by allowing users to specify whether a reorganization allows the workload to have concurrent access to the DB as its being reorganized. For example, Srinivasan specified index creation methods that each use different methods for how an index is created and how updates during index creation are handled concurrently with the workload [Sri92]. Each method results in different amounts of interference to the concurrently executing workload.

Unfortunately, the DBA is given the responsibility in commercial DB products of determining the trade-off and the tool(s) to apply in a given situation. Since there are so many new designs and associated reorganizations to choose from, a DBA would benefit from the assistance of automatic methods to guide his/her decision. These methods include:

- a method to generate different possible new designs;

- a method to generate different possible reorganization strategies; and
- a method to evaluate the performance of a new design that includes its benefit and the corresponding reorganization's cost.

An algorithm to select the design and reorganization would use the evaluation method to compare the performance resulting from the different choices. We develop such an evaluation method and the associated algorithm.

Also, the procedure to generate a reorganization must specify the priority level at which a reorganization strategy should execute relative to the workload queries' priorities. The priority of reorganization activities may be set to be higher than, equal to, or lower than the priority of the regular query workload. If the reorganization has a higher priority, it will be accomplished quickly, but will cause a higher degree of interference (for a short time). If a lower priority is used, the interference will be small, but it may take a long time to complete the reorganization. Part of our work will be to determine appropriate reorganization priorities under various system environments. We also develop an analytic method to estimate the execution time of a reorganization strategy and response times of transactions under the various priority levels for the reorganization.

1.1 Summary of Contributions

The contributions of this thesis relating to physical DB design on a SN system are in two areas:

- initial physical DB design; and
- concurrent physical DB reorganization.

In the area of initial physical DB design, we present algorithms that, when given the workload, DB, and system information, chooses a good physical DB design based on appropriate partitioning, clustering, and indexing. To compare alternative designs considered by an algorithm, we develop and use a workload cost estimator. The decisions are made based on each design's resulting costs for a workload. The costs for a given physical DB design are determined as the workload average response time that are calculated using the execution plans (output by a query optimizer) of the workload transactions. These costs include the estimation of the contention at the resources between the transactions.

For physical DB reorganization, we create a method to gauge a reorganization's cost and benefit. We also develop an analytic method to estimate the performance of executing a workload concurrently with an ongoing reorganization. A set of experiments is then used to determine the desirable priority levels for the execution of a given set of reorganizations under various system environments. Finally, some example reorganization decision algorithms are developed and demonstrated.

1.2 Outline of the Dissertation

This dissertation is organized as follows. In Chapter 2, we formally define the physical DB design and reorganization problems in detail. We also describe the parameters used to define a system environment including the relevant system, DB, and workload characteristics. This chapter includes an experimental example of the range of response times resulting from the possible designs for a given workload. We also discuss the problem of how to estimate average response times for a workload executing with and without a reorganization. The assumptions used throughout the thesis are defined in this chapter.

In Chapter 3, we provide a survey of the past work on the initial physical DB design problem, the concurrent physical DB reorganization problem, and workload cost estimation.

Algorithms to solve the initial physical DB design problem are presented in Chapter 4. One of the algorithms is based on a branch-and-bound strategy. We also describe all the components required for making the decisions within the algorithms, such as the method to estimate the workload's performance without reorganization, an algorithm to logically and physically group relations, and a method to prune away undesirable candidate designs.

We evaluate these physical DB design algorithms experimentally in Chapter 5, using various workloads. To carry out these experiments, we implemented our algorithms as a prototype tool for a real SN DBMS: IBM DB2 PE [BFG95].

In Chapter 6, we study reorganization. We provide methods to determine a reorganization's benefit and cost. This chapter also describes our method of estimating the workload costs during a reorganization, and presents some possible reorganization algorithms. We identify the appropriate priority of a reorganization in various experimental cases.

We summarize our conclusions and contributions in Chapter 7. This chapter also includes a discussion of future work arising from the thesis.

To complete the dissertation, we include seven appendices. In Appendix A, we outline our analytic cost calculations to estimate query operator costs. These operator costs are used in the estimation of workload and reorganization performance. In this appendix, we also define queuing network model terminology used throughout the dissertation. We describe the simulator we constructed to model a parallel SN DBMS in Appendix B. The simulator was used to validate some of the analytic estimations. Appendix C presents calculations used to rank attributes in a DB for two of the heuristics in Chapter 4. We then describe the DB and workload that is used in some of our experiments in Appendix D. The DB is defined by the popular TPC-D benchmark while the workload is based on the benchmark's queries. Other databases and workloads used in the experiments are defined in Appendix E. In Appendix F, the computational complexity of one of the physical DB design decisions is shown to be NP-complete. Appendix G provides some extra result tables for the experiments in Chapter 6.

Chapter 2

Physical DB Design: Problem Definitions

We define the two main problems addressed in the thesis, namely:

- selecting a physical DB design for a relational DB; and
- reorganizing a physical DB design.

We also provide motivation for the problems' importance and for the solution techniques we choose.

These problems are defined in the context of parallel shared-nothing (distributed-memory) systems. We concentrate on shared-nothing systems because they are consistent with the hardware systems commonly used by current DB vendors as relational database system platforms. For example, vendors such as Teradata [CK92], Tandem [CC93], IBM [BFG95], Informix [Cla93], and Sybase that have introduced DB products implemented for SN systems. Therefore, SN systems are a good system to study, so we concentrated on them in this dissertation.

In parallel shared-nothing systems, an interconnection network connects a set of autonomous processing *nodes*. Communication and synchronization between the nodes can only occur through the interconnection network, so the only shared resource is the network. This network can be of any type, e.g., a bus or an ATM switch. Each of the nodes has its own processor(s), local memory, and disk drive(s). Since shared-memory (SMP) machines provide good performance compared to a single processor machine, a node could be an SMP machine. Because SN systems can be constructed using off-the-shelf hardware, they have low price/performance ratios [DG92].

DB management systems on a shared-nothing system are constructed so that all the nodes of the system work together to provide the image of a single DBMS server. There are DB management processes on each node. Each node's processes handle the portion of the DB stored on its disks. This local portion consists of partitions of the DB relations. The partitions of each relation stored on different nodes are disjoint. To coordinate the execution of queries, each node's DB management processes communicate data and control messages to the other nodes' processes over the interconnection network. The DBMS parallelizes the query execution across the nodes, and strives to localize query operations at the nodes where the associated data resides. This model of execution is called *function shipping.*

When using the function-shipping model, the data layout (physical DB design) across the nodes determines where the DB functions are executed. Therefore, the physical DB design should be chosen so that the functions using the data (possibly across nodes) will execute with good performance. A physical DB design consists of:

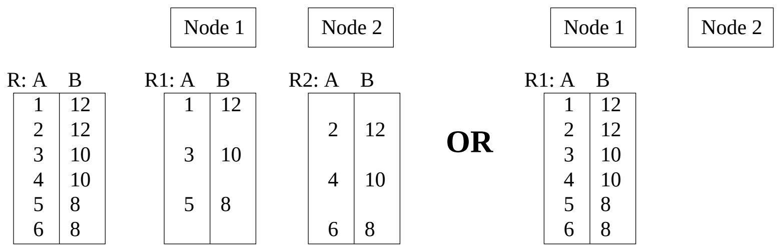
- a partitioning of the relations across the nodes and disks of the parallel system to facilitate parallel query executions;
- an ordering or clustering of records in a partition of a relation at each of the nodes the relation is stored; and
- special access structures (indexes) useful for non-sequential I/O access to some of the relations.

For example, we could store a whole relation R with attributes A and B on a two node SN system. The example relation is given on the left of Figure 2.1, and two examples of storing R are then given. One way we could store the relation is by placing it all at one node, as in the right most figure storing R only at node 1. However, if we had a workload defined by the query class in Figure 2.1 in which each instantiation of a query in the class is given a possibly different value for the variable, all queries of this class would execute at node 1. If we partitioned the relation across both nodes where the same values of the attribute A are stored at a specific node, as seen in the first partitioning of Figure 2.1, then we could direct each query to the appropriate node and reduce the contention at node 1. This would reduce the delays incurred by each query, thus improving their performance.

Related to selecting a physical DB design is the determination of how to evolve (reorganize) to a new design in response to a change in the environment. The problem of selecting both a new design and the method to move between designs differs from the problem of simply selecting an initial physical DB design.

In this thesis, we: (1) indicate how to make each choice for the initial physical DB design, and (2) describe methods to assist in selecting reorganizations and new DB designs.

The structure of this chapter is as follows. We first give the assumed system model and the objective function. These are described in Sections 2.1 and 2.2, respectively. We then



```
QUERY CLASS:  SELECT *
              FROM R
              WHERE R.A = < variable >
```

Figure 2.1: An example partitioning of a relation R across a two node SN system, and example workload class using an execution definable *variable*.

describe the different steps in selecting a physical DB design in detail in Section 2.3. We next describe the problem of moving between designs in Section 2.4. In Section 2.5, we provide an example of selecting a design for a given query workload. The purpose of the example is to show experimentally that there are many possibilities for the partitioning decision with a wide variation in their resulting workload performance. To compare physical DB designs and reorganizations, we require analytic cost estimation methods, so, in Section 2.6, the problem of estimating workload costs is addressed. Finally, in Section 2.7, we collect all of the assumptions we used throughout our thesis.

2.1 System Model

In this section, we describe the system model for our work along with the parameters defining such a model. The model has three components:

- system structure;
- DB structure; and
- workload structure.

These structures will be defined in detail in the following three sections. In Section 2.1.4, we discuss how the parameters for these structures can be obtained for our algorithms.

System Parameter	**Description**	**Example Value**
NumNodes	The number of nodes in the system	16
ReadSize	Num. pages read/written in a sequential disk read/write	4 pages
PageSize	Main memory and disk page size in bytes	4 KB
MemorySize	Amount of main memory per node	64 MB
IOTIME	Average access time per disk page (msecs)	16 msecs
COMMTIME	Latency for communicating a message over the network	0.08 msecs
MsgSize	Maximum size of messages in bytes	1 KB
DBptrsize	Number of bytes used for a pointer	4

Table 2.1: Parameters defining processor, disk drive, and interconnect communication network characteristics.

2.1.1 System Structure

We assume a homogeneous SN system in which each node has identical processor(s) and disk(s) and the same memory size. Our parallel shared-nothing system structure is defined by two sets of parameters:

- parameters defining processor, disk, and interconnection network characteristics; and
- parameters defining the processor instruction path lengths for query operations.

These parameters are shown in Tables 2.1 and 2.2, respectively. Their values can be obtained from system specifications or from system measurement. The parameter values exclude the delays due to contention from competing queries. Resource contention is estimated by our cost model.

We also assume that each node has only one processor. Solving our problems for these types of SN systems is a necessary first step toward solving them for more general SN systems (e.g., SN systems using SMP nodes). Relaxing this assumption is discussed in Chapter 7.

The example values for the path lengths shown in Table 2.2 and the parameters of Table 2.1 were determined from measurements on a 16 node system, where each node was an RS/6000 with a PowerPC 604 133 Mhz processor, 64 MB of RAM, AIX 4.1 operating system, and a 1 GB SCSI-2 Seagate ST15230N disk drive. The nodes are all connected by a 100 Mbps Fast Ethernet.

Instruction path lengths (given in microseconds in Table 2.2) are used to estimate the processor times required by an operator and query. These path lengths exclude the costs

Instruction Parameter	**Description**	**Example (Instruction Timing)**
I_{IO}	I/O overhead per disk page (read/write) of ReadSize pages	215 μs
$I_{STARTCOMM}$	Initiation cost of communication per message	102 μs
I_{COMM}	Communication cost per byte	0.04 μs
$I_{COMPARE}$	Cost to compare two values in memory	0.02 μs
I_{RESULT}	Cost to create/package a result tuple in memory	3.5 μs
I_{INDEX_SEARCH}	Cost to search for a key in an index	5.6 μs
I_{INDEX_INSERT}	Cost to insert a key in an index	5.6 μs
I_{INSERT}	Cost to perform an insert per tuple	3.5 μs
I_{DELETE}	Cost to perform a tuple deletion	3.5 μs
I_{UPDATE}	Cost to update a tuple	3.5 μs
I_{START}	Cost to start a query operator's execution	50 μs
I_{STOP}	Cost to terminate an operator's execution	50 μs
$I_{AGGREGATE}$	Cost to perform an aggregation operation per tuple	0.6 μs

Table 2.2: Instruction path length parameters.

of using the other resources such as disk access time and wire delay for communication. The timings of these path lengths were determined by taking the average over a thousand executions of the isolated operations executing from a C program. We took the times before and after the thousand executions to amortize the overheads of calling the timing routines. Also, we ran a second loop of a thousand iterations that did nothing except the timing so we could remove the cost for the timing and the looping costs from the first loop's cost. In compiling our code, we did not invoke the optimizer so that no statements were ignored in the execution. The parameter I_{IO} was determined by executing a single page read from a file. Our communication parameters ($I_{STARTCOMM}$ and I_{COMM}) were obtained from the measurements made independently by Parsons et al. [PBS97]. A comparison timing ($I_{COMPARE}$) was determined by comparing two floats together. The time to package a result (I_{RESULT}) was calculated using the timing for the assignment of a float that is 4 bytes (0.139 microseconds) and extrapolating this for a 100 byte record size. To calculate the index search timing, we assumed an index page is half filled on average, and half of the information is data (i.e., keys defined by 4 byte floats) while the other half is made up of pointers. This resulted in an average of 256 keys per index page, and we used the $I_{COMPARE}$ measurement multiplied by the number of required comparisons. Inserting into an index page assumed we had to move half of the data and pointers on average, which requires 256 four byte assignments. The cost to insert, delete, or update a tuple in a data page was considered the same as the cost to package a result tuple (I_{RESULT}). The timings for starting (I_{START}) and terminating (I_{STOP}) an operator's execution was measured as half the time to do a fork and kill of a process. Aggregation per tuple was determined by assuming four floats per tuple are used in the aggregation (where this value was higher than what we used as in our queries on average) and using the timing of an assignment of a float.

To model the disks at a node, we use one disk access time to represent one or more real disks per node. Thus, we avoid the problem of how to place a node's data on its local disks. This intra-node placement problem is outside the scope of our work. An aggregate access time for the node's disks is used as the access time, *IOTIME*, for each of our disks. Relaxing this assumption is discussed in Chapter 7. The parameter value for *IOTIME* was derived from the SCSI-2 disk drive's manufacturer specifications, which included the average seek time (9 milliseconds), the average latency (5.54 milliseconds), and the transfer rate (10 MB/sec). We used 16 milliseconds as the disk access time for a 4 KB page, based on the assumption that many requests will require a short or null seek.

The disk parameters include the disk page size, the average access time for a page, and the number of pages that can be read/written together in one disk access for a sequential scan/write. A block of pages read/written sequentially or *sequential read/write block* is a set of disk pages that can be read/written together, and is given by the *ReadSize* parameter. The average I/O cost per page or sequential read block is used for reads and writes.

There are a few parameters that deal with memory sizes. The parameters *PageSize* and *MsgSize* determine the number of disk accesses and messages required to handle a set of fixed length records. For example, if we were reading T tuples each with an average width of t bytes, the number of pages used in main memory and on disk for these tuples is calculated as:

$$\lceil \frac{Tt}{PageSize} \rceil$$

Also, if we are communicating T tuples each of t bytes from one node to another, then the number of messages sent would be calculated as:

$$\lceil \frac{Tt}{MsgSize} \rceil$$

Finally, the time for communicating a message over the network is given by *COMMTIME*. This parameter is derived from the network's nominal bandwidth, *BW* (in bytes per millisecond), and the message size, *MsgSize*, in bytes. The communication time is given by:

$$COMMTIME = \frac{MsgSize}{BW}$$

2.1.2 DB Structure

Our DB structure includes two categories of parameters:

- logical DB parameters; and
- physical DB design parameters.

The physical DB design parameters are described in Section 2.3.

Logical DB parameters include descriptions of each relation and each attribute of each relation. For each relation, the parameters are:

- the set of attributes in the relation (relation scheme);
- the average tuple size (in bytes); and
- an estimated number of tuples (*row cardinality* or just *cardinality*).

For each attribute of each relation, the parameters required are:

- an estimated number of distinct values per attribute (*column cardinality*);
- an indicator to denote whether an attribute's values are unique or not across the tuples;
- the data type of each attribute; and
- the number of bytes required to store an attribute value (which depends on its data type).

```
SELECT *
FROM R1
WHERE R1.A1 = < variable >
```

Figure 2.2: Example of a transaction template of a class using an execution definable *variable*.

Some other logical DB information specifies the relationship among the relations through an entity-relationship (ER) diagram. These relationships help in estimating the cardinalities for joins, and hence, the cost of the queries. For example, one relation's primary key might be used in another relation to link records together through joins. The second relation's attribute would thus be referred to as a *foreign key*. A relationship would result that has arity one-to-many between records in the relations, and constrains how many records with the primary and foreign keys are involved in a join. Throughout the rest of the thesis, we use the term *key* to define an ordered group of attributes in the same relation.

2.1.3 Workload Structure

The workload is composed of a set of transaction classes, which are the same as *query flocks* defined by Ullman [UAC$^+$97]. A transaction class represents a set of queries or updates that are closely related. Each class is described by the following information:

- arrival rate of transactions in the class (transactions per unit time), or a relative frequency of each class in the workload; and
- a transaction template for the class (given as an SQL statement).

A template could include *(host) variables* (e.g., as seen in Figure 2.2) that would take on different values when a transaction of the class is instantiated. These variables allow for the definition of similar queries forming a transaction class.

An SQL statement includes all the necessary information needed to characterize the transaction class, such as:

- relations involved in the transaction;
- attributes accessed in each relation; and
- the operations and the attributes to which they apply.

If any operation involves inserting, deleting, or modifying tuples, then the class is an *update* class; otherwise it is a read-only *query* class. The SQL statements for a class can represent simple on-line transaction processing (OLTP) types of queries or complex decision support system (DSS) types. We assume the workload is given at the time the DB design is being selected.

For an SQL statement, the DB query optimizer can use all the available system model information defined in the previous sections, including the attribute value cardinalities, to allow our algorithms to obtain query execution information. This execution information for each workload transaction class is required by our algorithms to estimate the workload's performance so that each algorithm can evaluate and compare the various DB designs it generates.

In some environments, *ad hoc* queries are supported, and the structures of these are not known at the time the physical DB design is selected. However, a representative set of component operations (such as selections, joins, and sorts) could be derived. We assume that these operations have enough similarities to one another that some could be grouped together to form a transaction class with an estimated frequency or average arrival rate. These representative classes could then be used as part of the workload to derive the DB design.

However, even if the workload is not known or derivable, if the relationships among relations are known through an entity-relationship (ER) diagram, we may assume that each relationship corresponds to a single join. We may also assume that each such join is involved with equal frequency since no frequency information is provided. Such a circumstance is found in data mining and on-line analytic processing (OLAP), where the queries are ad hoc or to be determined, but the DB and the relationships are known.

For simplicity, the workload model excludes subqueries and set operations (such as *UNION* and *INTERSECT*). Even with the exclusions, our model still allows for robust and complex classes. For example, the TPC-D workload is still complex when the exclusions are applied, as seen in Appendix D. We discuss the relaxation of this assumption in Chapter 7.

2.1.4 Obtaining the System Model Parameters

This section indicates how model parameters can be obtained, even for systems being set up for the first time. We will discuss separately obtaining each set of parameters for the system, DB, and workload, respectively.

For the system, some parameters can be obtained from the component (disk drive, interconnect network, and processor) descriptions while the others must be measured or estimated. The instruction path length parameters can all be measured or estimated, and in Section 2.1.1, we described the methods we used to obtain these values. The other parameters can be obtained from the physical structure of the system (e.g., the number of nodes, and page sizes allowed by the database system) or from the component descriptions (e.g., the IOTIME, which is determined using the manufacturer's specification for the average seek, rotational latency, and transfer times).

DB information can be obtained from the user's input and through measurement or estimation. Some parameters of the DB structure are given in the data definition language (DDL). These include the number of relations, the attributes per relation, the data type of each attribute, each attribute in which all its values are unique, and the estimated average number of bytes used to store each value for each attribute, which can be used to calculate the tuple size in bytes for each relation. Cardinalities of the relations and attributes can either be measured or estimated. For example, cardinalities for the TPC-D DB are estimated using information from the TPC-D description [Raa95]. The physical DB design information can either be given by the user (for components that are not to be chosen by a given algorithm) or are generated by the algorithms.

Workloads are obtained from user inputs and by estimations. We must be given the transaction (or query) class templates of the most important transaction classes in the workload along with the estimated relative frequencies or arrival rates of each class. Selectivities of the operations of a transaction class are usually returned by the optimizer within the transaction's execution information. A query optimizer would estimate the selectivities using such information as attribute value cardinalities.

2.2 Objective Function

The objective function used to make physical DB design decisions is the workload *weighted average response time* denoted by $WART$, and is calculated as:

$$WART = \frac{\sum_{i=1}^{Q} w_i f_i R_i}{\sum_{i=1}^{Q} w_i f_i}$$

where Q is the number of different query and update classes in the workload, f_i is the arrival rate of a query class, and R_i is the average response time of query class i. If the response times of some classes are considered more important than those of other classes, then a weight, w_i, is used. If the values of all w_i are equal, the resulting objective function specializes to the workload *average response time* (ART):

$$ART = \frac{\sum_{i=1}^{Q} f_i R_i}{\sum_{i=1}^{Q} f_i}$$

We shall use ART to represent the objective function, but will assume that WART can replace this in any occurrence throughout the thesis.

The average response time for a query (R_i) is derived using the query execution plan and a workload cost estimation model (defined later). For a specific design, workload, and system environment, each query's response time is derived from the query's communication times, I/O times, and processor times. A query's average response time also includes the queuing delay caused by the other queries in the workload.

The reasoning behind our objective function choice now follows. A goal for selecting a good physical DB design is to choose one that leads to low resource contention (i.e., low communication, I/O, and processor times) for the workload. For example, with large DB sizes, one potential bottleneck is in the I/O devices, because the number of I/O accesses will be high and the disk access times are long compared to times at other resources. Disk access times are usually between 5 and 20 milliseconds. To improve the performance, this I/O cost is reduced by determining a partitioning of the DB across the disks to fully exploit the available I/O bandwidth. Partitioning the large relations to as many nodes as necessary causes the access time of the partitions per node to be reduced.

However, inappropriate partitioning of relations may increase the communication costs of operations such as the execution of a join. Excessive communication could reduce the benefits of having fully balanced I/O. For example, in a join, one relation (A) may be partitioned on the join attribute, but the other relation (B) in the join may not be partitioned on the corresponding join attribute. To execute the join, the tuples of relation (B) would need to be communicated to the same nodes where the corresponding tuples of the other relation (A) currently reside so that B is also partitioned based on the join attribute. This adjustment allows the join to be executed locally where each new partition resides. We could thus avoid or reduce this communication by using a partitioning that localizes query operations.

Another important resource cost comes from the processor. Communication or I/O access involves processor costs (e.g., initiating the operation and packaging data). Therefore, by reducing I/O and communication costs (by reducing the number of disk accesses and network operations), we also reduce the associated processor costs. Selecting special data structures (indexes) also lowers the processor cost of some query operations, such as sorts, and this leads to lower execution times for the operations.

Since a good DB design requires low resource usage on, and therefore low contention for, the interconnection network, the processors, and the disks, we require a cost measure that takes all these usages into account, e.g., transaction throughput or average response time. Since response time is important to the user of a DBMS, we choose the minimization of the average workload response time as the objective function. Because query classes are characterized by their arrival rates, any acceptable physical DB design will lead to the throughput of each class being equal to its arrival rate.

2.3 Physical DB Design Problem for Parallel Shared-Nothing Systems

The specification of a physical DB design involves five decisions:

- selecting partitioning keys;
- forming relation groups;
- partitioning the relations;
- choosing the attributes by which to order or cluster each relation; and
- choosing the attributes for which to create indices.

The five decisions are discussed in the following sections.

2.3.1 Selecting Partitioning Keys

The first decision is to select a *partitioning key* for each relation. A partitioning key of a relation R is a subset of the attributes in R whose values in a tuple of R are used to map the tuple into one of several disjoint units of storage, called *fragments*. The attributes that are part of the partitioning key are called the *partitioning attributes*.

Fragments are grouped to form *partitions*, and each partition is associated with one node of the parallel system. Fragments can be grouped to form partitions to attain a balanced distribution of the number of tuples across all the nodes where the relation is stored. All of the tuples of each logical fragment could also be stored together on disk so that queries requiring a subset of the fragments in a partition need only access those fragments it requires. This restricted fragment access will allow for reduced disk accesses for the query. We could also have locks at the level of granularity of a fragment. This could be used to allow greater concurrency than table locking if only a fragment's data is to be accessed or changed. A fragment could also be moved to other nodes without changing the partitioning function. If a DBMS does not use fragments, we could avoid this level by having only one fragment assigned to each partition.

Depending on the partitioning used for the relations, a query may benefit from:

- localizing its execution to a node;
- parallelizing its execution across the nodes; and/or
- reducing the communication needed for its join operations.

Localizing a query's execution to a single partition or to a proper subset of the partitions of a relation causes the query to use less resources. Using less resources benefits the other concurrently executing queries because they incur lower congestion delays. If individual queries of a class each access tuples from different single partitions, then several can execute concurrently without delaying one another.

A query that is parallelized across the nodes accesses records in different partitions and thus on different nodes. Since the partitions are smaller than the full relation size, the access cost for each partition is typically less than the access cost for the full relation if it were on one node. In such a case, each query process at a node executes faster than a single query process executing on the full relation. For some operations, when a query operates on the partitioning key, the query processes can operate on each partition, and thus each node, independently without having to communicate the results to each other. An example of this is when records of a relation are aggregated based on the partitioning key. Sorting each partition independently and applying the aggregation function to each group of records with the same partitioning key means that the whole relation has been aggregated, and thus no communication is required between nodes to aggregate records with the same attribute values.

The choice of partitioning keys also affects communication costs of joins by leading to one of four possible communication patterns during a join execution:

- local join execution;
- directed communication;
- broadcast communication; and
- redistributed communication of both join relations.

A local join execution is the execution of the join locally at each of the nodes where partitions of the relations are located such that only the local partitions at a node are needed. This execution requires no communication. We will discuss this execution method in more detail later when we deal with relation groups.

The second pattern of communication is called *directed* communication, and involves the repartitioning of only one of the join relations and its communication to the nodes where the other relation resides. *Repartitioning*, or *redistribution*, is the communication and movement of a relation so that it is partitioned on a different set of attributes or placed on a different set of nodes. This directed communication occurs when only one of the relations is initially partitioned on the join attribute(s).

A third pattern is *broadcast* communication, in which all tuples of one of the relations are broadcast to all the nodes storing the other relation. Broadcasting for a join is done when neither of the relations being joined is partitioned on the join attribute. After the communication, the join can be done locally. A query optimizer chooses broadcasting when one of the relations is small enough that the cost of broadcasting the small relation is less than that of redistributing both relations.

In the fourth pattern, communication for a join is done by *redistributing* both relations. This is necessary when neither relation is partitioned on the join attribute and broadcasting is too expensive. The relations are each repartitioned on the join attribute and redistributed to the nodes where the join is to be executed. After the redistribution, the join executes locally at each of the join nodes.

The first three patterns provide benefits through reduced communication. However, the benefits from the first two directly result from selecting a relation to be partitioned on a join attribute, although all patterns depend on whether the partitioning keys correspond to the join attributes.

The selection of partitioning keys for the relations of a schema is a computationally complex problem. Exhaustive examination of all subsets of attributes for all relations is impractical for most schemas. Let n_i be the number of attributes in relation i, and k be the number of relations. The total number of possible partitioning attribute sets is $2^{\Sigma_{i=1}^{k} n_i}$. Thus, the problem has complexity that is exponential in the number of relations and attributes. Because of this complexity, our proposed algorithms are based on search heuristics. In Appendix F, we prove that the problem of partitioning key selection is NP-complete.

2.3.2 Forming Relation Groups

The second physical DB design decision is the grouping of relations to form disjoint *relation groups*. All relations in a relation group have compatible partitioning keys, and are partitioned across the same set of nodes.

A reason for setting up a relation group is to allow for localized execution of join operations between relations in the same groups. For example, one possibility is to group relations that will join together on their partitioning keys so that the partitions with the matching join attribute values will be stored at the same nodes. We say the relations are *collocated* or *correlated* under this partitioning. In this type of execution, we reduce:

- communication times for the query;
- processor times incurred because of communication; and
- congestion delays at the processors and interconnect network.

The complexity of the relation grouping decision is exponential in terms of the number of relations because each possible grouping is formed as a disjoint set of subsets using all the relations. For example, if we have k relations in our database, let $N(k)$ denote the number of groups for k relations. We know that $N(1) = 1$. Now if we choose any j relations from the k to be placed in the first group, the remaining $k - j$ would then be placed in other groups given $N(k - j)$ possibilities. Therefore, for $k > 1$, we have:

$$N(k) = \sum_{j=1}^{k-1} \begin{pmatrix} k \\ j \end{pmatrix} N(k-j) = \sum_{j=1}^{k-1} \begin{pmatrix} k \\ k-j \end{pmatrix} N(k-j) = \sum_{i=1}^{k-1} \begin{pmatrix} k \\ i \end{pmatrix} N(i)$$

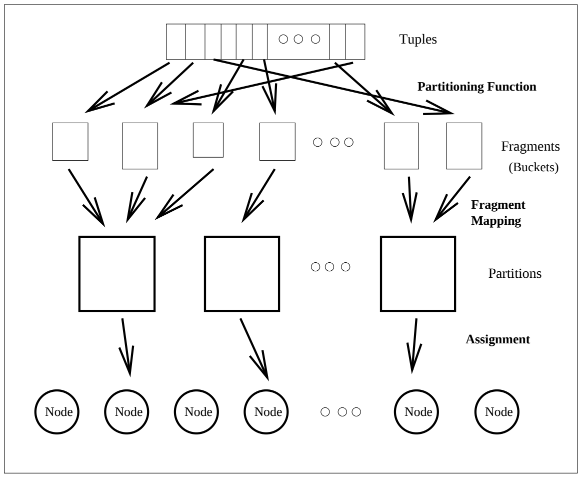


Figure 2.3: Partitioning of a single relation onto a set of nodes.

However, since good relation groups are subsets of sets of relations with compatible partitioning keys, and the partitioning keys are chosen first, the number of relevant relation groupings can be drastically reduced.

We could have developed an algorithm that chose relation groups first and then the corresponding partitioning keys. With this algorithm, the combinatorics of partitioning key selection would be reduced, but we would still require finding the proper partitioning key for each relation within each group since there are still a number of possibilities in how the relations in a group could be joined. We would also need to determine whether a grouping is valid by considering the joins and partitioning keys within the workload. This limits the reduction of the complexity for selecting groups first and then partitioning keys.

2.3.3 Partitioning the Relations: Data Placement

With a given set of partitioning keys and relation groups, the third decision is how to partition each relation group (and thus each relation in the group) across the nodes of a parallel shared-nothing system. The process of partitioning is shown in Figure 2.3, and is termed *data placement* in the literature. Data placement reduces the I/O times for a query at any single node by spreading the I/O activity across nodes. Data placement also spreads the processor costs for I/O across the nodes.

The decision steps for the partitioning decision related to a relation group (and thus each relation in the group) are as follows:

1. Determine the number of fragments allowed for each relation in the group.

2. Choose a partitioning function to map tuples to fragments based on their partitioning key values.

3. Determine the number of partitions to create (i.e., the *degree of declustering*).
4. Choose a fragment mapping function to map fragments to partitions.
5. Assign each partition to a different node (in a one-to-one mapping called the *assignment*).

Some common types of partitioning functions are:

- hashing functions, where tuples have their partitioning key values hashed to give the fragment number in which the tuple should be placed; or
- range partitions, where tuples are split according to ranges of the partitioning key values.

In the literature, the partitioning methods using these functions are called *hash partitioning* and *range partitioning*, respectively.

When the number of partitions equals the number of nodes, we call the partitioning a *full declustering*; otherwise the partitioning is called *partial declustering.* Once the number of partitions is determined, fragments must be mapped to partitions. One simple method we apply that distributes the fragments in a uniform manner is a round-robin mapping. The usefulness of fragments as an extra level of partitioning is described in Section 2.3.1.

The partitioning and placement of one relation group cannot be done independently from the decisions for the other relation groups because partitions of different relations are assigned to the nodes together, and thus interfere with each other through their use by the queries. However, if most relations are fully declustered, then the partitioning problem is simplified because each node is always assigned one partition from each relation.

The complexity of the partitioning problem is exponential in the number of relations and nodes. This decision problem is similar to the bin-packing problem where we are trying to place the fragments into partitions and thus into nodes such that our objective function is minimized. The data placement problem has been shown to be NP-complete by Padmanabhan [Pad92].

2.3.4 Choosing Ordering or Clustering Attributes

The fourth physical DB design decision is the choice of which attributes (if any) are used to order or cluster the tuples in each fragment of a relation. Clustering physically orders a relation's tuples on disk so that tuples close in the ordering will usually be nearly adjacent on disk. We call the set of attributes that determine the ordering or clustering the *ordering* or *clustering key.*

Several relational operations are facilitated by having the tuples of their operand relation(s) ordered or clustered on a specific attribute. For example, if a sort is required on the ordering key of a relation, then an external sort requiring many I/O accesses can be avoided. Each partition of the relation need only be read once, and the sort would reduce to a merge of the partitions to obtain the fully sorted result across the nodes. Also, when an operation, such as a selection, requires only the tuples with a particular value of the clustering key, then the relevant tuples can be read in large sequential reads, which have a much lower cost per tuple than random reads.

Selecting an ordering or clustering key is similar to selecting the partitioning keys because, in both decisions, we are selecting a subset of each relation's attributes as the keys. However, the ordering key decision differs because we must also determine the order of the attributes that compose a multiple attribute key. The complexity would again be exponential in the number of attributes since the total number of clustering keys would be equal to the number of ordered subsets of all the attributes in each relation.

2.3.5 Selecting Indices

The fifth and final physical DB design decision is the selection of the indexes. An *index* on a relation is a data structure used to facilitate the location of a relation's tuples that have a particular value for the indexed attribute(s). Some example structures are a B-tree or a hash table structure. The indexed attributes are called the *indexing attributes* or *indexing key*.

Indexes make it possible to:

- retrieve the relation tuples using an order based on the corresponding indexing key values; and
- obtain all tuples with the same indexing key value without the need for a sequential scan of the whole relation (or relation fragments).

Indexes reduce the number of necessary I/O accesses compared to a sequential scan when only the required tuples for a query operation are accessed through the use of an index, and we use operations (e.g., sorts and predicates of sufficiently low selectivity) that use the indexing key.

For a parallel DB system, another decision that can be made when choosing an index is whether to make the index a *local* or *global* index [Gra93]. A local index is really a group of indexes, where one index is created for each of the relation's partitions and the index referencing a partition is stored at the same node as the partition. A global index is an index that is stored on a parallel system as a single data structure for the whole relation where the index could be partitioned across the nodes independently of the relation partitioning. An example partitioning of a global B-tree index was given by Seeger and Larson [SL91].

There are disadvantages, however, when too many indexes are created. First, the extra storage space for the indexes may exceed the disk space available after the relations are stored. Second, an index may not be used in query execution plans because either an index of the same relation or a sequential scan of the relation allows lower response times. Third, the maintenance costs of updating the set of indexes may outweigh the performance benefit of using the indexes. Indexes require updating when updates on a relation affect the indexing key values or the record locations. A selection algorithm must therefore find the set of indexing keys that provide good performance even after taking into account the maintenance cost. The algorithm for index selection must consider these disadvantages in its decisions.

Although both the indexing and partitioning key decisions select subsets of attributes, the indexing problem requires finding more than one indexing key, and thus more than one subset of attributes can be chosen per relation. In fact, index selection has been shown to be NP-complete [Com78, CFM95]. Capara et al. equated the index selection problem to the knapsack problem to produce this result [CFM95].

2.3.6 Interactions Among the Physical DB Design Decisions

All these five decisions interact and affect each other. One interaction is between the decisions for clustering and indexing keys. In many database systems, a relation can only be clustered on a key if that key is used to create a special index called a *clustering index* or *primary index.* Indexes on keys other than the clustering key are called *nonclustering* or *secondary indexes.* For a clustering index, the key will be indexed and the base relation will be clustered according to the key's values. With such a structure, there only needs to be one pointer to the base relation tuples per key value, as seen in INDEX B of Figure 2.4. This pointer will point to the first tuple having the key value, and all other tuples with the value follow this tuple sequentially on disk. For a nonclustering key and for a specific key value, the pointer for the key value in the leaf page points to a list of pointers, where each pointer in the list points to a tuple in the base relation having the key's value. Thus, an extra level of indirection exists for nonclustering indexes as shown in INDEX A of Figure 2.4.

There are also interactions that exist between the indexing key and partitioning key decisions.

- First, an index on a key reduces the need for partitioning on that key. For example, if the selectivity of a search for the tuples matching an index key value is low enough, we could benefit by parallelizing other operations instead of performing the search by using a partitioning key that is different from the index key. Thus, query operations on either the partitioning key or the indexing keys are assisted.

- Second, in a parallel system, an index could be affected by the underlying partitioning used for the relation. This partitioning effect comes about from the choice of an index as either a local or global index.

- Third, partitioning and indexing together on the same key may reinforce each other when

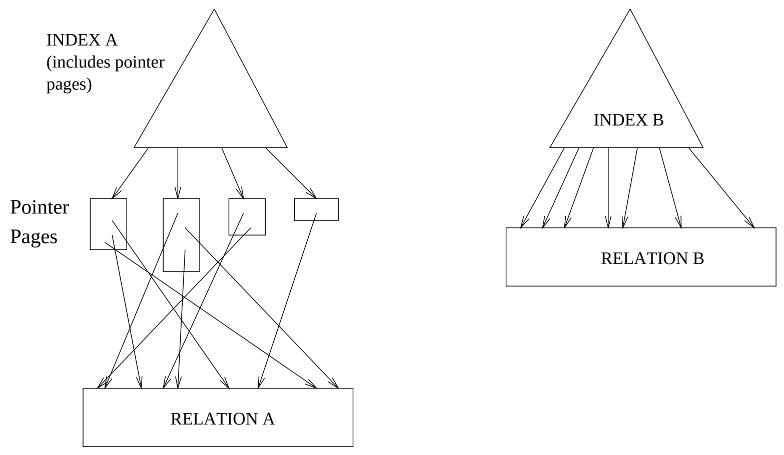


Figure 2.4: Example of index using a nonclustering key (INDEX A) and a clustering (INDEX B) key.

using local indexes. For example, if we want to parallelize a sort based on a key, we benefit by partitioning on that key and performing local sorts at the nodes. We also benefit by using the index at a node to obtain its referenced partition in a pre-sorted order, thereby reducing the I/O and processor costs for each local sort.

A final interaction results from a dependence among the three partitioning decisions when the effects on the workload need to be evaluated for each candidate partitioning. To evaluate a partitioning, partitioning keys, relation groupings, and a data placement need to be chosen. Since a data placement can only be chosen after a set of partitioning keys and a relation grouping is selected, the placement is dependent on these decisions. Also, since the relation grouping requires knowledge of the partitioning keys chosen, grouping is dependent on partitioning key selection. However, to evaluate whether a partitioning key choice is good compared to other candidates, the grouping and placement must be determined for each candidate, and this causes the partitioning key decision to be dependent on grouping and placement. As a result, the three decisions must be considered together.

2.3.7 Overview of the Decision Structure

Because the decisions interact, there is no sequential ordering of the decisions that will consistently lead to the best design. Good designs can be assured only by making all the decisions together. This is the method that we pursue in this thesis.

Our solution to the physical DB design decision problem is implemented as a set of modules shown in Figure 2.5 (where the component connected by dashed arrows is an extra module required for DB design reorganization as will be discussed later). The main module selects the attributes to partition, cluster, and index the relations. For each set of attributes that

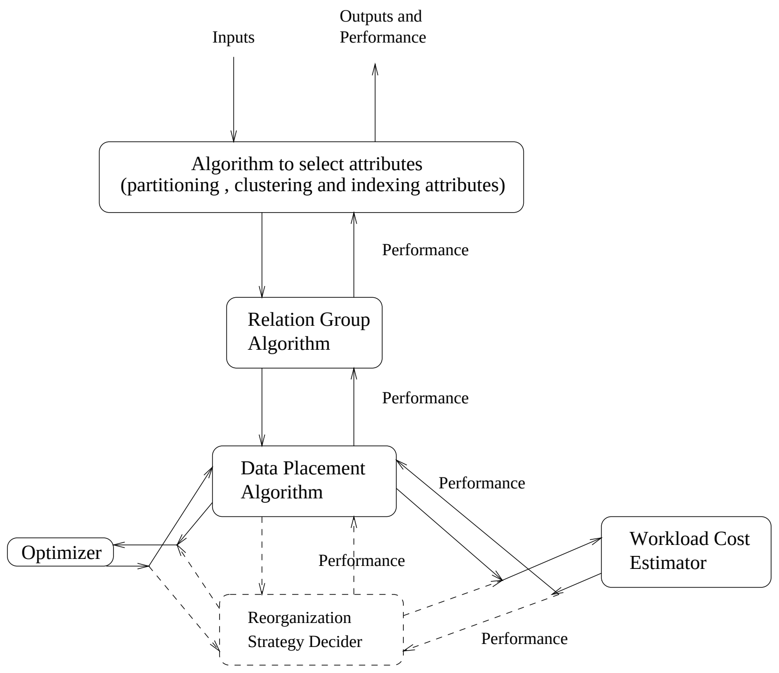


Figure 2.5: Modular diagram of the components needed for an initial physical DB design decision algorithm or a physical DB design reorganization problem.

is considered, relation groups are formed and the data placement is determined by two other modules. Once these three modules have made a choice, the optimizer is called once for each query, and the estimated cost of the workload is obtained. Our algorithm chooses the physical DB design resulting in the lowest estimated performance found during the algorithm's search.

In executing the various decision modules, we can only choose physical DB designs that satisfy a set of constraints:

- A relation group's degree of declustering must be less than or equal to the number of nodes in the system.

- A physical DB design that causes the workload to *saturate* the system is unacceptable. A saturated system is one that is unable to process transactions in some class as fast as they arrive.

- There should only be one partitioning key and at most one clustered index for each relation.

- The space required to store the base relations and the indexes cannot exceed the disk capacity. (In many cases, a design is chosen ignoring capacity constraints on the assumption that additional disk capacity can be added where required.)

2.4 Physical DB Reorganization Problem

Once a DB system is operational, there are inevitably changes in what transactions are executed, in the structure and content of the DB, and in the hardware and software that support the DB. Because the choice of an effective physical DB design depends on all these factors, it is necessary to periodically reassess the physical DB design choice to assure that it continues to provide adequate performance. In this case, the problem is harder because we must be concerned not just with what an ideal design is for the current system, DB, and workload, but also with the method and cost of taking actions to evolve the physical DB design from its current one to one that would be better in light of the changes in the environment.

Database *reorganization* is the process of changing from one physical design to a better one. When we consider two specific designs to move between, we need to find a specific method for moving between the designs. This method is called a *reorganization strategy.*

Because the reorganization strategy and its costs are involved, the resulting decision process differs from the previously defined (initial) physical DB design decision. This new decision problem is called the *physical DB design reorganization problem.* For the initial DB design decision, only the selection of a design matters. However, for the reorganization problem, there are a number of important factors involved in the reorganization decision:

- the selection of the new design;
- the anticipated performance of the workload with the new design;
- the formulation of a reorganization strategy to move to the new design;
- the cost of interference to the workload while moving to a new design;
- the time to move to the new design; and
- the amount of time that the new design is expected to remain appropriate.

Since there are many possible modified designs, and since there are many possible reorganization strategies, a heuristic is required to solve the decision problem. This decision problem involves an objective function that includes all the factors above when selecting a design and an associated reorganization strategy.

To limit the number of candidate designs and reorganization strategies considered, two constraints can be applied. First, ignore designs that do not perform better than the current one. This constraint requires that the reorganization algorithm involve the current design in its operation. Second, the cost of reorganizing must be exceeded by the performance benefit achieved with the new design; otherwise there is no advantage in moving to this design.

Once a candidate design is found that outperforms the current design, a reorganization strategy is constructed. To obtain a suitable strategy, the decision process must determine the cost and benefit of each of many possible strategies. The possible strategies range from ones that do not allow access to the DB by the workload during the reorganization (*off-line execution*) to ones that allow the workload to have full access to the DB during the reorganization (*on-line* or *concurrent execution*). In current systems, it is preferred that the DB be available to the users at all times to prevent the queries from incurring excess delay [DG92]. Given this availability constraint, on-line execution strategies would cause lower delays for the workload queries.

The *priority* at which the reorganization strategy executes relative to the queries' priorities directly affects the concurrency allowed. Different priorities result in different execution times for the reorganization and different degrees of intrusion on the workload. A low reorganization priority will lead to minimal intrusion on the workload but will cause the reorganization to take longer and hence delay the attainment of the new design. A high priority will allow the new design to be reached quickly, but will have the highest level of intrusion on the workload. Choice of the proper priority depends on how quickly the new design should be achieved and how much intrusion is tolerable on the workload. For example, given that the workload is performing poorly on the DB system using the current design, it may be prudent to allow a higher intrusion for a shorter period of time so that the benefits from the new design can be reached as soon as possible. We study the effects of the different reorganization priorities in this thesis.

2.5 Experimental Motivation

We present an example using part of the TPC-D workload and database [Raa95] to provide evidence for the importance of physical DB design selection. We show the number of candidate designs and the range of their resulting average response times. In this example, we concentrate only on choosing the partitioning keys and relation groups. One of the purposes of this section is to illustrate that a wide range of solutions, as well as a wide range of response times for these solutions, exist for the partitioning decisions. Due to the high number of possible solutions, a good search algorithm is required to limit the number of candidate solutions examined explicitly.

To gauge how complex a good heuristic needs to be, we show the resulting average workload response time of a naive algorithm (IR) that considers only one solution. This algorithm uses DB and workload information and bases its decisions on query frequencies and the difference between the simple estimates of the effects on query execution time when a candidate key is chosen or not chosen to be the partitioning key. We thus evaluate a simple naive non-iterative algorithm not involving the query optimizer or workload cost estimator. The difference between the naive solution and an exhaustive algorithm's solution demonstrates the need for an in depth

Query	Relative Frequency
(a)	10
(b)	30
(c)	40
(d)	15

Table 2.3: Relative frequencies for experimental queries.

search.

Our example involves using the IBM DB2 PE optimizer [BFG95] and the calculations in Appendix A to determine response times for a particular workload. We use the example values given for the system parameters in Tables 2.1 and 2.2. The database relation sizes are the ones used for a 4 GB TPC-D benchmark DB (based on a scale factor of four in the benchmark). The workload consists of four queries, shown in Figure 2.6, each based on one of the queries in the TPC-D benchmark [Raa95]. In our example, the relative frequencies of the queries in the workload are given in Table 2.3. The arrival rates were determined by preserving the relative frequencies of the classes while setting the overall rate such that the bottleneck utilization for a naive solution was 60%.

We give a histogram of all the workload response times found through a nearly exhaustive search to show the variation in performance among different possible physical DB designs. The search is called nearly exhaustive because the only partitioning keys allowed in the search are attributes that are used by the queries. The histogram shown in Figure 2.7 was plotted by dividing the difference between the minimum and maximum response times by 100, thus giving 100 intervals. In performing the nearly exhaustive search, we evaluated a total of $69,685$ different designs for relation groups and partitioning keys.

We could have reduced the set of candidate partitioning keys even further by considering only the join attributes of the queries since their selection as partitioning attributes would possibly aid parallel performance. The histogram when only join attributes are used is given in Figure 2.8. Only 37 partitioning key sets were considered. In this case, the best solution found in the limited search was as good as the best one found in the nearly exhaustive search.

In executing our nearly exhaustive search, we obtained a predicted response time of 413 seconds for the best design, but when using the naive algorithm, we only found a design with a predicted response time of 534 seconds. Thus, in this example, the better search algorithm yields a design with performance 23% better than that of the best design chosen by the naive algorithm. In Table 2.4, we show the individual query's average response times along with the workload average times under the physical DB designs from the naive and the nearly exhaustive

```
SELECT N1.N_NAME,YEAR(L_SHIPDATE),L_EXTENDEDPRICE*(1-L_DISCOUNT)
FROM SUPPLIER, LINEITEM, NATION N1
WHERE  S_SUPPKEY = L_SUPPKEY AND S_NATIONKEY = N1.N_NATIONKEY AND
    N1.N_NAME = <hv1> AND L_SHIPDATE BETWEEN DATE(<hv2>) AND
    DATE(<hv3>)
```

Query (a): Based on TPC-D query 7

```
SELECT O_ORDERDATE, MAX(O_SHIPPRIORITY)
FROM CUSTOMER, ORDERS
WHERE C_MKTSEGMENT = <hv1> AND C_CUSTKEY = O_CUSTKEY AND
    O_ORDERDATE  < DATE(<hv2>)
GROUP BY O_ORDERDATE ORDER BY O_ORDERDATE
```

Query (b): Based on TPC-D query 3

```
SELECT COUNT(*)
FROM ORDERS, LINEITEM
WHERE O_ORDERDATE >= DATE(<hv1>) AND
    O_ORDERDATE < DATE(<hv1>) + 3 MONTHS AND
    L_ORDERKEY = O_ORDERKEY AND L_COMMITDATE < L_RECEIPTDATE
```

Query (c): Based on TPC-D query 4

```
SELECT N_NAME, DECIMAL(SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),31,3)
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND O_ORDERKEY = L_ORDERKEY AND
    L_SUPPKEY = S_SUPPKEY AND C_NATIONKEY = S_NATIONKEY AND
    S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
    R_NAME = <hv1> AND O_ORDERDATE >= DATE(<hv2>) AND
    O_ORDERDATE < DATE(<hv3>) + 1 YEAR
GROUP BY N_NAME
ORDER BY 2 DESC
```

Query (d): Based on TPC-D query 5

Figure 2.6: Workload for motivation experiments, where $< hvi >$ represents a variable instantiated per query execution.

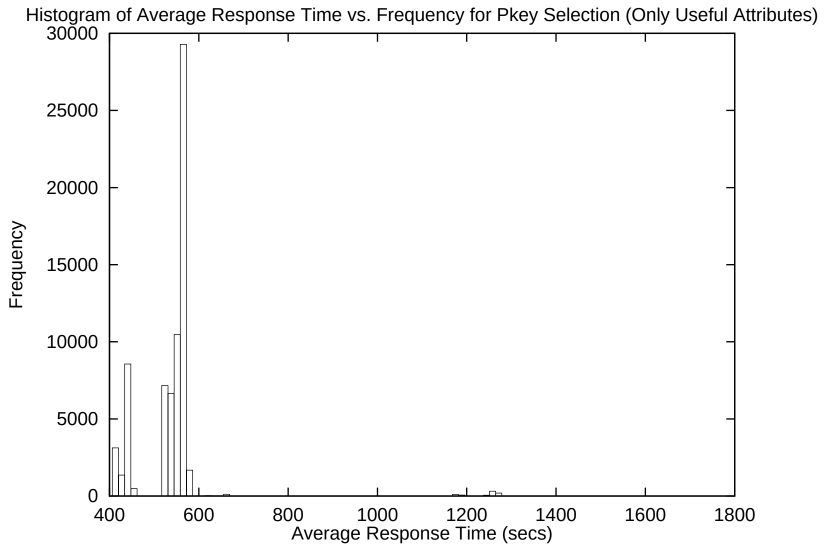


Figure 2.7: Response time histogram of the 69, 685 designs considered by a near exhaustive DB design search.

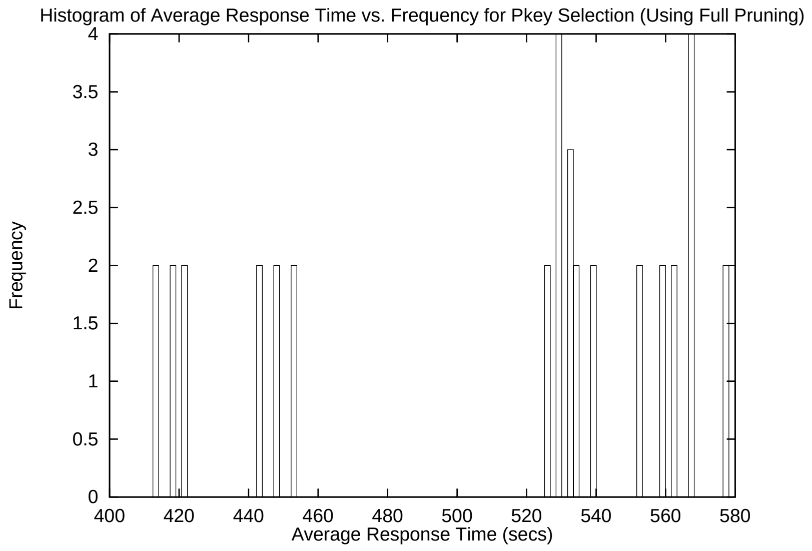


Figure 2.8: Response time histogram of the 37 designs considered by a DB design search using only join attributes.

Query	**Arrival Rates**	**Naive**		**Nearly Exhaustive**	
	(Queries/hour)	**Single Query Response**	**Response with Workload**	**Single Query Response**	**Response with Workload**
(a)	1.116	222	542	222	468
(b)	3.384	44	106	46	93
(c)	4.32	261	631	208	434
(d)	1.692	471	1123	464	962
Workload			534		413

Table 2.4: Response times of queries under the physical DB designs from the naive and nearly exhaustive algorithms. All times given in seconds.

algorithms. We show the single query response times along with the estimated average query class response times when executing in the full workload. The corresponding arrival rates are also given in the table. Also, the maximum utilization was at a disk drive, and it was 60% using the naive design and 54% using the design from the nearly exhaustive search.

To gauge the effects of the algorithms on the resources, we calculate the average workload residence times for the disk drives, the processors, and the interconnection network. For a specific resource k, an average workload residence time (RES_k) is calculated by estimating the average residence time ($RES_{i,k}$) for each query i, and using a weighted sum similar to ART, i.e., we use:

$$RES_k = \frac{\sum_{i=1}^{Q} f_i RES_{i,k}}{\sum_{i=1}^{Q} f_i}$$

where Q is the number of query classes in the workload and f_i is the arrival rate of query i. In Table 2.5, the maximum times among the set of disks and processors along with the average times for the interconnection network are shown. We note that the residence times at the processors and disk drives are improved while the communication residence time is larger in this solution than in the solution of the nearly exhaustive search. One reason for this is that the workload is I/O bound, and improving processor and disk costs are most important. Also, the proper partitioning allows a lower average time spent at the disks because some of the queries can be directed to specific nodes and thus they cause less congestion at the other nodes' resources.

In the histograms, there are also a number of better solutions that have lower response times than the result of the naive algorithm. This motivates us to develop a search algorithm that executes quickly, by explicitly considering only a limited number of designs, yet finds

Resource	**Naive Residence Time**	**Nearly Exhaustive Residence Time**	**% Difference**
	(in seconds)	**(in seconds)**	
Communication	0.18	1.9	-949
Processor	18.1	14.8	18
I/O	203.2	181.7	11

Table 2.5: Average workload residence times of the physical DB designs selected by the naive and nearly exhaustive algorithms.

designs as good as the design from a nearly exhaustive search either with certainty or with high probability.

2.6 Estimating DB Costs on Parallel SN Systems

In this section, we describe a *workload cost estimator* to calculate the average response time of a workload under various DB designs produced by our physical DB design algorithms. A candidate design's workload response time depends on the DB, the parallel SN DB system, and the queries in the workload. Since many choices are considered by a design algorithm, a cost estimator must evaluate each of the choices quickly. Thus, using simulation or experimentation for this purpose is infeasible. The estimators we propose are analytic.

Previous estimation methods focused on determining the costs of a single query's execution, usually for query optimization. However, these methods are insufficient for our purposes because physical DB design decisions are affected by the interaction of all the queries in the workload. For example, some queries are aided by a particular design choice, and some are not. If the unaided queries' costs cause high resource contention for the aided queries, the benefit of the DB design choice could be negated. Therefore, we use an estimator that includes delays caused by contention among the queries at the resources in the workload. This estimator uses a *single query cost estimator* as a submodule to evaluate the resource usages for a query plan of a single SQL query.

For the reorganization problem, besides estimating the workload response time, we must also evaluate the response time of each reorganization strategy and the strategy's effects on the workload. These values for each strategy are formulated into one comparison metric used by a reorganization algorithm to compare all the strategies it considers. The estimator used for reorganization purposes differs from the workload estimator because the reorganization is considered differently from the transactions representing the workload queries. We call this new estimator that considers reorganizations the *reorganization cost estimator*.

Figure 2.9 illustrates how the different estimators interact. The DB decision algorithm obtains performance estimates for the workload by calling either the *workload cost estimation* module or the *reorganization cost estimation* module. Each estimation module must be given the query plans for each query in the workload and the reorganization strategy. These plans are obtained by the decision algorithm by calling the query optimizer under a given physical DB design choice. An estimated average workload response time is then returned to the decision algorithm. We use the following procedure to calculate the average response time for any candidate physical DB design:

1. Pass specifications of the DB, system, and a transaction class structure along with the candidate physical DB design to the optimizer.

2. Obtain the execution plan of each transaction from the optimizer.

3. Use each execution plan along with all the DB structure, system structure, and the physical DB design to estimate the resource usage that each query of the class requires of the system.

4. Use the transaction classes' resource demands to obtain each query's expected average response time including congestion delays from other queries.

5. Calculate the ART associated with the candidate design.

As stated in Section 2.3, the physical DB design parameters include the following:

- the indexing keys;

- the clustering keys;

- the partitioning keys;

- the relation groups; and

- the data placement.

The decision algorithm seen in Figure 2.9 is quite flexible. Since the estimator determines average workload response times, any decision algorithm requiring these times can use the estimator we have created. For example, a DBA could use the estimator to select an appropriate hardware configuration for a given DB and workload. The model remains unchanged, and the decision algorithm changes only the parameters describing an algorithm's decision. For example, if a decision algorithm is used to find the proper number of system nodes, then the *NumNodes* parameter is changed by the algorithm.

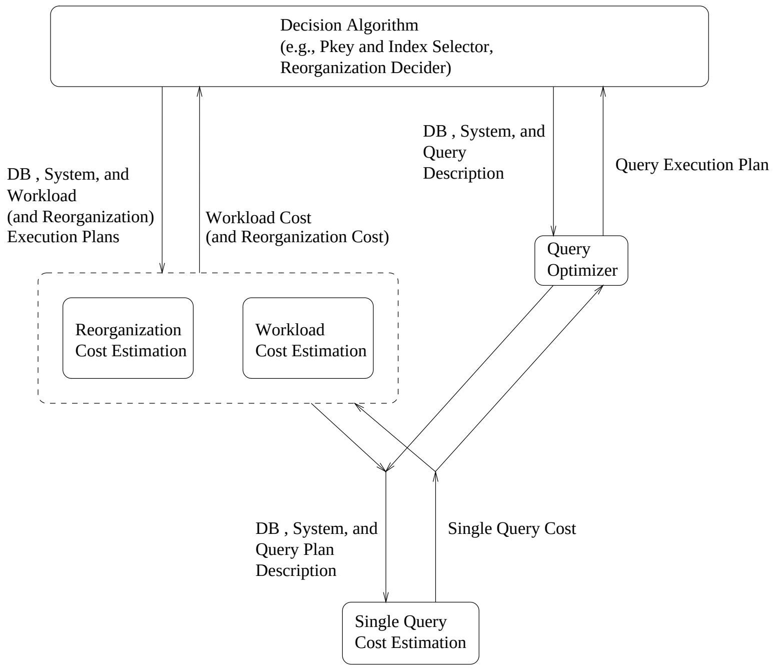

Decision Algorithm
(e.g., Pkey and Index Selector,
Reorganization Decider)
DB , System, and Workload (and Reorganization) Execution Plans
Workload Cost (and Reorganization Cost)
DB , System, and Query Description
Query Execution Plan
Query Optimizer
Reorganization Cost Estimation
Workload Cost Estimation
DB , System, and Query Plan Description
Single Query Cost
Single Query Cost Estimation

Figure 2.9: The structure of the workload cost estimator module.

Workload Structure for Cost Estimation

The SQL structures do not explicitly provide the necessary information to determine a query's actual execution plan or resource requirements. The workload structure for cost estimation requires a query execution plan (obtained from the optimizer) for each SQL statement of a transaction class and the arrival rate of each class.

Each query plan is a detailed description of the execution of an SQL statement. An example of a join query and its associated query plan are given in Figures 2.10 and 2.11, respectively. A plan is basically a tree of operators with only one root operator (or source) that is either an operator executed at a *coordinator* (e.g., the AGG operator in Figure 2.11) or is a non-query operator. A coordinator is the process that starts a query's execution, and always executes at a single node. On each execution of the query, the node executing the root is called the *coordinating node*. Non-query operators are inserts, deletes, and updates, which do not return tuples to the coordinator as the other operators do. The leaves or sink nodes of a query tree access the base relations (e.g., the SEL operators in Figure 2.11), and each query plan executes in a bottom-up fashion by first accessing its base relations. In the tree, the connection between operators denotes passing the output relation from one operator to another. The operator that receives this information is called the parent. For example, in Figure 2.11, the SORT operator is the parent of the JOIN operator. Nodes corresponding to join operators have two children; nodes corresponding to other operators have just one child. We use the term *inner child* for

```
SELECT MANAGER, COUNT(*)
FROM STAFF, ORG
WHERE ID = MANAGER
GROUP BY MANAGER
```

Figure 2.10: Example of join and group by query.

a join to represent the join's right child in the figures. The optimizer chooses the child that will be the inner one. Once the choice is made, we cannot interchange the children. For a nested-loop join, the inner child has the relation we must scan for every record of the *outer* (or left) child's output relation. Some other example query plans are provided in Appendix A.

Each operator in the query plan will have enough information associated with it to determine how the operator will be executed and how costs can be estimated. The information includes the operator type (e.g., a join or a select) and the method to be used in executing the operator. For example, a selection method could be either a file scan or an index scan. There is also quantitative information associated with each operation including the output tuple size, the estimated selectivity of the operation, and the estimated number of tuples in the output relation. For some operations, the plan indicates the indexes that are to be used, any selection conditions used by the operator, and whether the operator pipelines its output to the next operation.

Information denoting where an operator will execute comes in the same form as information about where a relation is placed on the nodes. Along with the placement information, the operator location information also denotes whether the operation is executed at a single node or at all the nodes at which the relevant relations are placed. An operator may use a variable that is instantiated at each execution of a query in the same class, and a specific value for the variable for a query in the class could cause the operator is to be executed at a single node. We thus assume that each query in the class could use any single node where the operator's input table could be located. The costs for the query class are thus determined by assuming that, on average, each query uses one of the possible nodes. For example, a coordinating node could be any node in the system.

Each operator in the plan also has a data placement of the output from the operator. This placement defines a partitioning that includes the estimated number of output tuples per node. The placement of an operator is required as input to the next operator in the tree, and is derived using the base relation placement information and operator selectivities. These placements are propagated upward to the root of the plan. The placement at the root corresponds to the tuples that are either read by the user (at the coordinating node) or used to insert into, update, or delete from a base relation.

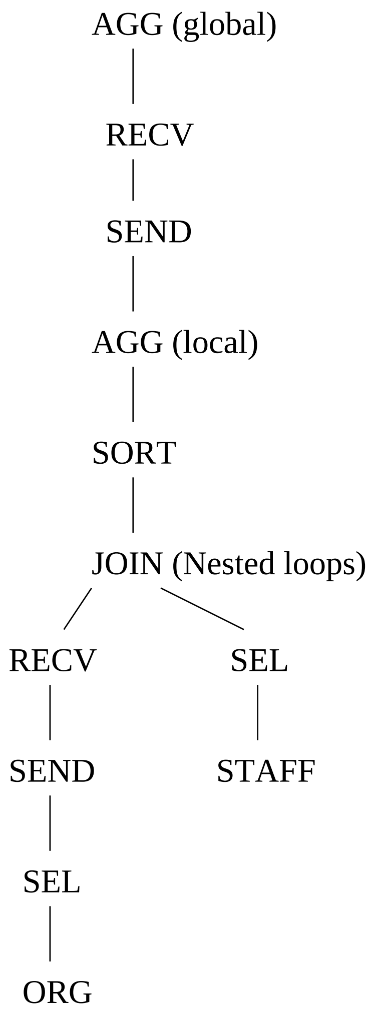


Figure 2.11: Example of join and group by query plan.

2.7 Assumptions

In this section, we give the assumptions made throughout the thesis that influence the (static and dynamic) physical DB design decisions. We group these assumptions into two subsections. In Section 2.7.1, the assumptions relating to the inputs for our physical DB design decisions are stated. Section 2.7.2 gives the assumptions affecting the cost estimations. The future work section of Chapter 7 discusses the relaxation of some of these assumptions. This section also includes descriptions of where uncertainties can lie relating to the inputs and estimations that directly affect the quality of the results from our algorithms choosing a physical DB design.

2.7.1 Assumptions About the Inputs

In this section, we describe the assumptions about the inputs for our physical DB design decisions. We assume that the workload, logical DB, and system inputs are provided by the DBA or some other external source to our algorithms. These input values are made available to our algorithms. In Section 2.1, we described a method by which the DBA could obtain the input values. There are three sets of assumptions:

- assumptions about the DB definition;
- assumptions about the input workload; and
- assumptions about the system configuration.

These assumptions are stated below.

Database Definition Assumptions

If the database information is taken directly from the DDL that defines the DB, then there can be no inaccuracies in the specified quantities. We made the following assumptions on the DB information:

1. Only horizontal partitioning is used and not vertical partitioning. We assume that any normalization of relations takes place before input into our algorithms.
2. No views are materialized.
3. No portions of the DB are replicated across two or more nodes.
4. The only index structures we consider for our systems are B-trees.
5. All indexes are implemented as local indexes.

Workload Assumptions

If the queries are expressed as a template for each query class, with replaceable variables, then the only uncertainties in the workload information are:

- the arrival rates (or relative frequencies) of each query class; and
- the distribution of the values plugged in for the variables in each query class.

Our assumptions for the workload inputs are as follows:

1. Queries are single SQL statements.
2. No subqueries or set operations are allowed in the SQL statements. This reduces the required scope of the operator cost formulas in our cost estimator. However, our algorithms can still address interesting workloads. The structure of the algorithms would not be affected by the use of a more general cost estimator.
3. Variables in a query class take on specific values for each query instantiation. These variables are assumed to be bound to any value in the corresponding attribute's range, and each value is assumed equally likely.

System Configuration Assumptions

The processors, disks, and interconnection network are described at some level of detail. If each node is characterized only by the processing power and I/O capacity and speed, then these rates must be accurately calculated from the detailed node characteristics, which could involve multiple disks or multiple processors in an SMP. The assumptions related to the system configuration are as follows:

1. We use a shared-nothing parallel architecture. We assume one processor and one disk per node, but assume the instruction path lengths and *IOTIME* can be set to account for the faster times when multiple processors or disks are used.
2. The SN system's nodes are homogeneous.
3. Disk capacity constraints are not used in the implementation, but we provide for this input in our algorithms.
4. Main memory pages and disk blocks are of the same size.
5. The maximum number of pages allowed for each operator's execution (*OperatorPages*) is given as input to our algorithms. It is up to the DBA to choose this number of pages relative to the number of pages available per node (*MemorySize*).

6. Decisions affecting locking granularity are made before using our algorithms. Poor locking granularity choices will likely not be overcome by good subsequent design choices.

2.7.2 Assumptions Affecting Estimations

The assumptions affecting the cost estimations for the physical DB design decisions are presented in this section. We present these assumptions in three groups:

- assumptions about the query plans and query optimization;
- assumptions about cost estimation calculations; and
- assumptions about queuing network model (QNM) estimation.

These assumptions are described in the sections that follow.

Query Plan and Optimization Related Assumptions

If the query plans are produced by the actual optimizer for the system being configured (which is our assumption), then no error should be introduced by the selection of the query plan. Our assumptions related to the query plans and optimizer are as follows:

1. The optimizer is assumed to be the optimizer for the DBMS for which we are choosing a physical DB design.
2. Relation grouping is used by the DBMS and the corresponding optimizer.
3. The operators in which we are interested are: join (JOIN), sort (SORT), insert (IUD), update (IUD), delete (IUD), communication send (SEND), communication receive (RECV), selection (SEL), and aggregation (AGG). This set of operations is similar to the set addressed in Padmanabhan's thesis [Pad92].
4. Only placement information for an existing relation group is eligible as placement information for operator executions. One case where an operation does not use a relation group's placement information in determining an execution location is at the coordinating node of a query plan.
5. Since the coordinator could execute at any of the nodes, each of the nodes is equally likely to be involved in a query execution.
6. The placement of an operator's execution is assumed to be determined using the locations where its children executed. For the execution of a selection operator on a base relation, the execution will be at the nodes used by the base relation's placement. There are

two possibilities for the placement of a join's execution since the join has two children. However, a join's placement is assumed to always be at the same nodes as where the inner child executes.

Cost Formula Assumptions

The cost estimates are sensitive to parameters that describe the system and the estimated operator selectivities. This is probably the primary source of errors in the decisions and the easiest place to improve the effectiveness of the physical DB design selection. Our cost calculation assumptions are as follows:

1. We assume that an input tuple size for a relation is the average size for a tuple in that relation in bytes. This assumption will allow us to determine the average number of pages required to store a number of tuples for a relation.

2. Although we are given selectivity estimations and can estimate output placements from operators in a query plan, some information about the DB attribute cardinalities is incomplete. This incomplete information relates to the knowledge of the number of distinct attribute values for a column per node for temporary and base relations. If the attribute has only unique values, we estimate the number of tuples per value per node as 1. When an attribute is a partitioning key, we estimate the number of tuples as: $\lceil \frac{R}{A} \rceil$ where R is the relation cardinality and A is the number of unique attribute values. However, when an attribute is neither a partitioning key nor unique, we assume that it is independent of the partitioning key of its relation. This assumption allows us to estimate the attribute column cardinality per node as being the same as its column cardinality across the whole relation. We also assume that the attributes are independent. If an attribute has A values and its relation has cardinality R, then we assume there are $\lceil \frac{R}{A} \rceil$ tuples with the same attribute value, i.e., the attribute's values are evenly distributed among the tuples. Our TPC-D based workloads fit these assumptions.

3. If an attribute is indexed with a key that does not cluster its relation, then we assume that the leaf index pages will have pointers to pages that have pointers to base relation information, e.g., INDEX A in Figure 2.4. Hence, an extra level of indirection exists relative to indexes using clustering keys. We assume that the cost of accessing a unique attribute value with such a nonclustering index requires an I/O access per pointer in the redirection pages. For an index on a clustering key, e.g., INDEX B in Figure 2.4, a pointer from a leaf points directly to the base relation.

4. We place restrictions on the allowed operator methods. Joins are either nested-loops or merge-joins. Selections are done by either file or index scans. Communication receives

either merge the tuples received from different nodes to obtain a sorted result or do not merge the tuples if sorting is not required. Communication sends can broadcast their tuples to the receivers or direct their tuples to individual receivers.

5. If a single node execution is specified and corresponds to the values of a variable in a query class, each of the nodes defined by the operator's execution placement information is equally likely to be involved in an execution of the query. Using this equal probability assumption, we calculate, on average, what the response time would be on each node over all query executions.

6. We assume that only hash functions are used for partitioning. Hashing is uniform across fragments, and partitions will have an equal number of fragments and thus tuples. However, our model is flexible enough to allow an imbalanced number of tuples across partitions (e.g., as in Chapter 6).

7. Recovery, logging, buffer management, and concurrency control (using locking) are not represented explicitly. We assume that reasonable strategies are used and that these overheads are not significant. For buffer management, we assume each operator in a query execution plan can use at most *OperatorPages* pages during its execution. If more pages are required, then I/O may be required, such as for external sorts.

8. When prefetching is used, we assume the number of pages prefetched is always *ReadSize* pages.

9. If we have T tuples of t bytes each, then these tuples are stored to fit in the smallest possible number of pages, which is equal to

   $$\lceil \frac{Tt}{pgsize} \rceil$$

   where $pgsize$ is the size of the pages.

10. For concurrent reorganizations, we assume *differential files* are used to hold updates to records that are being reorganized. We assume the cost related to using a differential file is negligible.

Queuing Network Model Assumptions

With reasonably accurate input values, QNM's have been shown to provide reasonably accurate performance predictions. The errors due to QNM calculations are probably dominated by other errors in this application. We make the following assumptions:

1. Calculations of performance measures based on queuing network models are acceptably accurate [LZGS84].
2. A query class's arrival process is Poisson, where interarrival times are independent and identically distributed exponential random variables.
3. Scheduling at resources is FCFS unless otherwise stated.

Chapter 3

Related Work

In this chapter, we review the literature in each of the problem areas we are studying. In Section 3.1, we survey the literature on the selection of the initial physical DB design. We then present the research on physical DB design reorganization in Section 3.2. Finally, in Section 3.3, previous research on query and workload cost estimation is discussed.

3.1 Initial Physical DB Design

Since each of the decisions in the initial physical DB design problem has been studied individually, we present the work for each decision in the following sections. There have also been methods that combined some of these decisions into one physical DB design decision algorithm. We review these methods in Section 3.1.5. A survey of the issues relating to selecting a physical DB design, including the importance of using query plans from a DBMS optimizer to determine the effects of the DB design choices, is given by Haberhauer [Hab90].

3.1.1 Clustering, Ordering, and Indexing Key Selection

Although the ordering and clustering key selection is an important problem, only Jakobsson presented a method to select the ordering keys alone [Jak80]. He assumed that the probability of access, or access frequency, of each attribute in a relation was given, and created an ordering key by sorting all the attributes in descending order based on these probabilities. This sorted list becomes the ordering key. However, use of all of the attributes to order a relation leads to a high overhead to maintain this order. Jakobsson did not consider this overhead or a method to select subsets of a relation's attributes for use in ordering.

Solving the clustering key selection problem is usually a side effect of the index selection problem. Some index selection algorithms find the key of a relation on which to create a clustering (primary) or nonclustering (secondary) index. This combination of clustering and

Reference	**Primary Chosen**	**Multiple Attributes Per Key**	**Optimizer**	**Workload**	**Relation Decisions Separated**	**Objective Function**
[ISR83]	NO	NO	NO	SIMPLE	YES	#pages
[BPS90]	NO	NO	NO	SIMPLE	YES	#pages
[FON92]	NO	NO	YES	SQL	YES	CPU or I/O
[CFM95]	NO	NO	NO	SIMPLE	YES	CPU and I/O
[GHRU97]	NO	YES	NO	SQL	YES	#rows
[RS88]	YES	NO	NO	SIMPLE	YES	#pages
[CBC93]	YES	NO	NO	SIMPLE	YES	#pages
[Wha87]	YES	NO	NO	SQL	YES	I/O
[FST88]	YES	NO	YES	SQL	NO	Query times
[CN97]	YES	YES	YES	SQL	NO	Query times

Table 3.1: Summary of index selection work.

indexing selection is usually done because many DBMS systems only allow clustering by way of a clustering index. We thus focus on the index selection problem for the remainder of this section.

Algorithms for index selection are characterized in Table 3.1. All of these algorithms include the selection of both primary and secondary indexes except for the algorithms that only select secondary indexes, namely the algorithms by Ip et al. [ISR83], Barucci et al. [BPS90], Frank et al. [FON92], Capara et al. [CFM95], and Gupta et al. [GHRU97]. The algorithms in the table also make decisions for more than one relation at the same time. Each algorithm is characterized by six features, based on the following questions:

- Are primary indexes selected?
- Are indexing keys with more than one attribute considered?
- Is an optimizer used to obtain query plans and query costs?
- Is the workload composed of simple read/write accesses or actual SQL queries?
- Are decisions about different relations made independently?
- What objective function is used in the search?

Table 3.1 reveals some interesting aspects of the methods. First, some algorithms ignore the choice of the clustering indexing key, and instead select only the nonclustering indexing keys. The number of candidates considered is thus reduced, but the algorithms are constrained by the relation clustering given to them by the user, and thus do not yield as low response times as the full index selection algorithms that find the best full index set. Second, only two methods allow the selection of multi-attribute keys. Third, only three methods use an optimizer to determine the query plans resulting from a candidate design. When a design is actually implemented, the optimizer is used to obtain the query execution plans, which are used to determine how the design affects the execution cost of a workload. Determining the performance for the optimized plans resulting for a given design during the selection algorithm will provide a realistic measure useful for comparing different designs' effects on the given workload. Fourth, in all of the algorithms, none of the objective functions realistically model workload performance, and they do not calculate congestion delays. However, since indexing keys are chosen based on their impact on the workload performance, accurate estimation of the workload costs including congestion delays is warranted.

Fifth, to simplify the selection, most of the methods assume that the indexing decisions can be made independently for the different relations. This assumption is based on the property of separability described by Whang et al. [WWS85]. This property allows the selection algorithm to ignore multiple relation query operations (i.e., joins) when, for each join, the choice of an access structure (index) for one of the join's relations is unaffected by the decisions made for the other relation in the join. Whang et al. state that a design decision is separable when the following three conditions are satisfied:

- The costs of operations relating to a single relation in a query are independent of the access structures for and the operations on the other relations.
- A method to execute an operation on a single relation can be chosen regardless of the access structures and the operations used for other relations.
- No restrictions exist in assigning an access structure for a relation independently of the access structures for the other relations. (This condition is merely an operational condition in which the definition of an access structure for a relation excludes any references to access structures for the other relations.)

Whang et al. concentrated on sort-merge joins, and provided join methods that inherently satisfied these conditions. Unfortunately, a nested-loops join method is not separable because the nesting order of the relations affects whether an index is required. Also, since an optimizer must choose between a sort-merge or a nested-loops, or some other join method, the optimizer's decisions are affected by the access structures of its child relations, and the access structures themselves are chosen based on how well these complex operations can be optimized.

We will now discuss the secondary index selection methods followed by the methods that select both the primary and secondary indexes. Ip et al. solved the index selection problem as a knapsack problem for each relation on a single node [ISR83]. To solve the problem, they also included constraints that make the solution more realistic, such as the disk capacity constraint.

The method by Barucci et al. considered optimization techniques to be computationally expensive, and hence solved the problem using a simple heuristic based on the properties of their objective function (number of pages accessed) [BPS90]. These properties are denoted as follows:

1. If the addition of an index increases the response time cost function, then this index should never be added.

2. If the deletion of an index from a set increases the response time, then the index should not be deleted from consideration.

The method used two heuristic steps that add and drop indexes according to the properties until no more additions or deletions are possible.

Part of the algorithm executes an ADD method that, on each iteration, when given a set S of indexes and set U of all possible indexes, will add an index to the chosen set of indexes (S) whose resulting workload cost is least among all the new indexes to possibly add next. No indexes are added in an iteration when the workload cost is raised by any additional index, and, when no addition is possible, the ADD method terminates. Usually, the ADD method is initially given the empty set, i.e., $S = \emptyset$. However, the algorithm by Barucci et al. determines an initial set of indexes (S) such that each index is guaranteed to be in the optimal set.

Their algorithm proceeds as follows. Initially, they use all the indexes ($S = U$), and determine the indexes in S that cause the response time to increase on their deletion. This set of indexes is used as the core for the set of indexes (S') that will be in the optimal solution, and this set S' is removed from the candidate set ($S \leftarrow S - S'$). Using S', the algorithm determines the indexes that raise response times when added to S', which are the indexes that should not belong to the optimal set. These indexes are removed from the candidate set S. If we find that the candidate set is empty, the algorithm stops with S' as the optimal set. When S is not empty, but no indexes can be added to S' to increase the response time, the ADD algorithm begins by using S' as the initial set, which is guaranteed to be a subset of the optimal set. When neither of these conditions is satisfied, the algorithm again tries to delete and then add indexes until the conditions hold.

The problem with this method is that it uses simplistic cost functions that are linear in the addition or deletion of indexes. Because true response time functions are non-linear and this algorithm does not make use of these functions, the algorithm is not guaranteed to find an optimal solution.

Another heuristic developed by Frank et al. constructs plausible sets of indexes that should be passed to an optimizer so as to determine the effects and resulting response time for a query using this set [FON92]. The set of plausible index sets for a query consist initially of all the indexes that could be used in the query. The remaining sets are constructed as subsets of this initial set using heuristic rules that estimate the effects on performance of a subset in order to eliminate subsets. In their algorithm, the possible index sets and the corresponding query costs they yield are accumulated per query. The algorithm then chooses the indexes that provide a positive savings compared to sequential access. Frank et al. stated that the rules they applied allows the optimal solution to possibly be missed, and they do not provide any guarantees on how far away their answers are from the optimal.

To provide some guarantees on the closeness of the resulting performance to the optimal, Capara et al. equated the secondary index selection problem to a 0-1 integer linear programming problem, and solved it with a branch-and-bound method [CFM95]. The objective function was a linear function based on processor and I/O costs, and they provided many pruning rules based on the linearity of the function. However, cost functions including congestion delays, such as response time, are non-linear.

Gupta et al. provide a method that selects single attribute and multi-attribute indexing keys [GHRU97]. Their algorithm determines secondary indexes for data warehousing views, and determines the views themselves. A greedy scheme is used for each iteration to find a new candidate view or index set that most reduces query costs compared to all other candidates. This method has some deficiencies. First, the choices resulting from a greedy technique are not always optimal. Second, the costs of the queries resulting from all the possible candidates are determined in each iteration, which is computationally expensive.

Because the performance resulting from selecting only secondary indexes is constrained by the chosen clustering, algorithms have been developed to include the selection of primary indexes. Rullo and Sacca select primary and secondary indexes for network databases [RS88]. Their method considers all possible candidate keys along with their effects on the queries' costs, and eliminates indexes that result in greater query costs than the costs resulting from the other indexes. Because this method evaluates the query costs resulting from all candidate keys, it is too expensive to use in realistic settings.

To avoid the expense of selecting primary and secondary indexes together in one algorithm, Choenni et al. suggested a two-phase method [CBC93]. The first phase determines the primary indexes, while the second selects the secondary indexes based on the primary indexes chosen in the first phase and on a performance property called *sub-optimality.* A set of indexes is sub-optimal when it is optimal conditioned on the chosen primary indexes. However, to make the two-phases independent, they determined the cost of the workload using a primary index set by estimating the best possible cost for the workload using any secondary index set along with the primary index set. Because of the indirect association of primary and secondary indexes, the performance resulting from a sub-optimal set of indexes was not guaranteed to be optimal.

Another heuristic, presented by Whang, used the DROP method to select the primary and secondary indexes [Wha87]. Initially, the method uses indexes on all the attributes as the first candidate index set. At each iteration, DROP removes and records the index that would cause the greatest decrease in the response time when it is deleted from the candidate set. The heuristic considers each relation separately, and drops individual indexes first, then pairs of indexes, then triples, and so on. In this way, the method indirectly considers the effects indexes have on each other. DROP terminates when the removal of any set of indexes causes the response time to increase.

All of these previous methods depend on the separability property, and thus their resulting decisions may not be optimal. Finkelstein et al. avoided this problem by eliminating the separability assumption as much as possible [FST88]. This algorithm uses an optimizer to help determine the cost effects of an index on the workload. However, the single query costs (e.g., resource usage or estimated response time) returned by the optimizer are used in a weighted sum to obtain the objective function value, and thus congestion delays and realistic performance measures are not estimated. Also, their algorithm does not choose multi-attribute indexing keys. Their method first removes attributes not used by the queries or used in a way that indexing on them will not assist the queries. The remaining candidate attribute set is further reduced by eliminating indexes resulting in higher query costs than for other indexes, as in Rullo's and Sacca's method [RS88]. Each feasible subset of indexes from the remaining candidate set (*survivor list*) is then considered and the corresponding workload cost is determined. The index set that leads to lowest query costs is chosen.

Another method that avoids the separability property was developed by Chaudhuri and Narasayya for the Microsoft SQL Server [CN97]. The index selection method is the same as the ADD method. However, they added some extra enhancements. First, they included a module to select the important candidate indexes, which are the indexes whose attributes are directly used in the queries. Second, a module is provided to include multi-attribute indexes consisting of two attributes. Third, their objective function is a sum of each query's estimated execution cost returned by the optimizer, which does not include congestion delays. This is similar to the objective function used by Finkelstein et al. [FST88]. One goal of their algorithm was to use a low number of optimizer calls by deriving the cost of an index set from atomic index sets for each query. Reducing the number of calls was deemed necessary since each call is time consuming. Because they use the greedy ADD method, they are not guaranteed on how close their chosen index sets will be to the optimal. Also, their algorithm is manageable because they limit the number of indexes allowed in the chosen set.

3.1.2 Partitioning Key Selection

In the literature, two types of partitioning methods have been identified. There are hash and range partitioning that are based on partitioning keys; and there is the round-robin partitioning that does not depend on partitioning keys. Based on simulations, Ghandiharizadeh concluded that the former methods outperform the latter [Gha90].

Although there have been many data placement methods proposed that depend on partitioning keys [CABK88, Gha90, DGS$^+$90, FM89, MD97, PB92], little attention has been paid to partitioning key selection. All the past data placement work for parallel DB systems assumed that appropriate partitioning keys had already been chosen. Some papers mentioned the importance of selecting partitioning keys [CABK88, Gha90, PB92].

However, some techniques have been created to automate the partitioning key selection. These include:

- a method for distributed DB systems; and
- a tool in a commercial DB system.

Ceri et al.'s method partitions the relations into subranges based on predicates (*minterms*) so as to reduce the amount of communication during query executions and thus localize the executions at the distributed system nodes where the partitions reside [CNP82]. These minterms are constructed from the query predicates, and the attributes in the predicates form the partitioning keys. For example, if the query predicates on relation R are "$R.A < 32$" and "$R.B > 90$", the resulting partitioning key for R would be $\{A, B\}$, and the minterms would be all the combinations of the predicates and their negation, e.g.,:

$R.A < 32$ and $R.B \leq 90$,

$R.A < 32$ and $R.B > 90$,

$R.A \geq 32$ and $R.B \leq 90$,

$R.A \geq 32$ and $R.B > 90$.

The method partitions a set of relations by first categorizing the relations into primary relations, which are relations with no foreign keys, or secondary relations, which have at least one foreign key. Partitioning keys for the primary relations are then selected independently from the other relations based only on the minterms for the relation. A partitioning key for each secondary relation is chosen when all the relations it joins with have had their partitioning keys selected. This secondary relation's partitioning key will be the foreign key in the relation that is involved in the most frequent join with one of these other relations.

However, this is not a good method for choosing partitioning keys for a parallel SN DB because its objective is different and because the partitioning key decision is not separable. Ceri et al.'s objective of partitioning for localizing the query executions is different from the objective of using the partitioning for parallelizing and load balancing query executions across the SN nodes to improve the workload's performance. Also, in the method, decisions for each primary relation were made independently of the decisions for the other relations. In general, there is a dependence between relations when selecting partitioning keys because whether or not we select the join attribute of one relation in a join affects whether the other relation of the join should be partitioned on its join attribute, e.g., for collocation purposes. Thus, a DB design decision is not separable.

As for existing partitioning key selection tools, only the Configurator tool for the Sybase parallel DB system (Navigator) [SD95] is applicable. This tool estimates the workload costs with analytic and simulation modeling. The modeling uses database, system, and workload characteristics to determine data placements including partitioning keys. A DB administrator is allowed to test various scenarios with the tool including partitioning methods and attributes, data placements, other DB design specifications, and hardware configurations. However, there is no indication that the Configurator considers more than one query plan per query class in making its decisions. Also, the methods are confidential and have not been independently evaluated.

3.1.3 Relation Grouping

The concept of selecting relation groups to be physically collocated so as to obtain good workload performance has not been addressed previously. However, the concept of collocating relations is not new. The IBM DB2 PE product introduced *node groups* that are similar to our relation groups, and DB2 PE allows the DBA to create groups, assign relations to groups, and define a data placement for each group [BFG95]. Other vendors do not explicitly support relation groups.

Padmanabhan proposes an assignment heuristic called the Disjoint Assignment by Priority (DAP) algorithm that allows some relations to be collocated [Pad92]. Before assigning a relation R to a set of nodes, the priority of each of the other relations is calculated as the frequency of the joins between R and the other relation. This method constructs a sorted list of relations that interact with relation R in descending order of the priorities. The heuristic then tries to collocate R with the highest priority relation with which it interacts. Collocation is allowed only when R has the same degree of declustering as this other relation and when both relations are partitioned on the join attributes of a join between the two relations. No other collocations were considered.

3.1.4 Data Placement

Placing file partitions across a set of nodes was initially solved as a file assignment problem in which whole files were placed at a node. A survey of this work is given by Dowdy and Foster [DF82]. These problems were formulated as optimization problems with decision variables representing the placement of the files on specific nodes, and were solved by heuristics since the problem was proven to be NP-complete [DF82]. The objective functions were weighted formulas, usually linear, of the resource costs. Resource costs varied, but usually involved I/O and processing times with an equal weighting between them.

To expand on the problem, Ries and Epstein considered horizontally partitioning (*declustering*) relations so that a relation's tuples are distributed across a set of partitions [RE78]. The partitions are placed disjointly on a set of distributed DB system nodes. This declustering spread the accesses to the relations across the nodes, thus spreading the I/O costs for the accesses across the nodes.

To justify the usefulness of declustering, Livny et al. provided simulation studies to compare no declustering to full declustering over multi-disk subsystems [LKB87]. They demonstrated that, under different disk scheduling and query frequencies, full declustering resulted in reduced disk utilizations and greater I/O parallelism.

There has also been work on horizontal and vertical partitioning over a distributed DB system [CNP82, OV91, Ape88]. Vertical partitioning involves the partitioning of the tuples themselves. These methods create fragments so that queries could execute locally at the nodes where the fragment includes all its required attributes. Fragments are placed to reduce the communication costs.

All of these papers dealt with placement methods for single node systems using multi-disk subsystems and for distributed DB systems. However, these methods differ from the data placement method for a SN system. For multi-disk placement, the workload is specified as a set of simple reads and writes, and the interaction between relations is not considered. For distributed DB placement, the goal is to mainly localize the accesses and reduce the amount of required communication as much as possible. The main goal of the SN system placement is to improve the performance of the queries by allowing for their parallel execution. Also, since communication is less expensive in a SN system, the goal is to reduce all resource costs.

To deal with data placement on a SN system, Copeland et al. created a method for MCC's SN DB system called Bubba [CABK88]. Their method assumes that the degree of declustering of each relation is given, and determines where the relation's partitions should be placed. They also assume the user supplies access frequencies (*heat*) for the relations. These heats are used to sort the relations in descending order, and then, one after the other, the hottest relations are placed on the nodes with lowest loads. The load of a node is continually updated during the placement to reflect the sum of the heats for the partitions it stores. Because they assumed the overhead costs (e.g., high latency in communication) from fully declustering a relation is high (actually higher than what exists in present systems), they stated that declustering a relation to fewer than all the nodes (*partial declustering*) was better than full declustering. However, they did not fully justify this claim in their paper.

Rahm and Marek provided simulations for a two relation database to demonstrate that, for simple single join queries with low selectivities, partial declustering allows for lower response times and higher throughputs than full declustering [RM93]. They also assumed that communication latencies are high.

In Padmanabhan's placement work, high latency interconnection networks and overhead costs were used that were also higher than the costs found in present systems [Pad92]. His cost function is a sum of the resource costs (communication, processor, and I/O costs), which inflates the costs by 10% to estimate congestion delays. There was no justification of why the value of 10% was used. Under the particular system environment studied, partial declustering was found to be better than full declustering. This was due to the high overhead used for setting up processes on each node to execute the query and to communicate with all the processes on all the nodes (when full declustering is used). When the number of records per node is low (such as for full declustering), the overhead costs dominate the query execution times. Reducing the declustering across a smaller number of nodes in their environment leads to a reduction in the overheads and the query execution times. He developed combinatorial methods using hill climbing and greedy search strategies to not only obtain the assignment of the partitions onto the nodes of a SN system but also to determine each relation's degree of declustering.

Mehta and DeWitt studied the placement problem using latencies that reflect the current state of the art, i.e., communication latencies that are similar to ones for an ATM switch [MD97]. Simulations were performed using simple and complex queries with various selectivities and frequencies of execution. He concluded that relations should be partitioned across a set of nodes such that each partition had at least $ReadSize$ pages. For medium and large relation sizes, full declustering is thus appropriate. For relations smaller than $NumNodes \times ReadSize$ pages, he applied assignment methods such as the method by Copeland et al. [CABK88], random assignment, or round-robin assignment. Copeland et al.'s method and a round-robin assignment resulted in similar performance, and outperformed the random method.

3.1.5 Physical DB Design Decision Algorithms

The importance of determining all the decisions required for a physical DB design has been surveyed by Graefe [Gra93]. In this survey, Graefe stated the need to investigate indexing, clustering, and partitioning decisions, and noted that some of the physical DB design decisions are interdependent. In this section, we survey methods that were designed to make these interdependent choices.

In commercial SN DB systems, a DBA can specify the indexing, clustering, and partitioning. For most systems, the process of selecting the proper database design, including the partitioning, is left to the DBA. Methods used by these administrators often involve self-built tools based on observations, rules of thumb, and intuition. Some rules of thumb given for IBM DB2 PE [Cor96] are:

- the partitioning key should include the most frequently joined columns;
- the partitioning key should include columns that often participate in a GROUP BY clause; and
- the partitioning key should have enough distinct values to allow a balanced distribution of tuples across the nodes.

However, we shall see that the most frequently joined columns are not always the wisest choice.

Techniques have been proposed in the literature that address all the physical DB design decisions in one algorithm. Unfortunately, all these methods, such as the ones given in Section 3.1.1, were developed for single node DB systems, and thus did not deal with partitioning decisions [MS78, CM83, RS91].

March and Severance [MS78] as well as Carlis and March [CM83] developed methods to select a physical DB design including indexes and record segmentation. Record segmentation is a partitioning of the records of a relation between primary and secondary memory. Their workload consisted of simple single relation selections. The costs resulting from a design were evaluated as the sum of the storage, retrieval, and maintenance costs, and evaluating the costs ignored the optimization of the queries. They provided no guarantee that the resulting performance was optimal.

A more robust method by Rozen and Shasha selects *features* of a physical DB design that includes indexing, ordering, clustering, and vertical partitioning [RS91]. A set of these features that together make a feasible physical DB design was called a *realizable feature set*. The method uses an exhaustive search for each query to determine its realizable feature set, and then constructs the workload's ideal feature set from a subset of the features in the queries' realizable sets. However, for partitioning keys, this feature set may not be optimal. For example, assume we have the following two predicates from two different queries:

- $R.A = x$ and $R.C = y$
- $R.A = z$ and $R.D = w$

In this case, the ideal set for the first predicate might be to partition on attribute C, and the ideal set for the second may be to partition on D, but an ideal partitioning key that benefits the whole workload could be A.

An existing physical DB design decision tool called the Configurator was developed for the Sybase Navigator parallel DB system [SD95]. It includes support for the partitioning problem, where the number of partitions, the locations of the partitions, and the partitioning keys are given as output. This tool determines all the physical DB design decisions, including the choice of clustering and nonclustering indexes, with the help of user interaction. As stated earlier, it was not clear that more than one query plan per query class is considered by the Configurator in determining its decisions. Also, this tool and its cost estimator have not been publicly described or evaluated.

3.2 Physical DB Design Reorganization

The work on physical DB design reorganization can be split into two categories:

- tools or utilities that reorganize one specific aspect of a design; and
- reorganization algorithms.

Reorganization algorithms can be subdivided into:

- off-line algorithms; and
- concurrent reorganization algorithms.

Because of the "24×7" availability requirement, reorganization algorithms should provide for concurrent reorganizations [DG92]. However, the off-line algorithms cannot be ignored as they can be adapted to develop on-line algorithms. The utilities are also of interest because a good reorganization algorithm would use these utilities to construct reorganization strategies. We discuss some existing utilities in Section 3.2.1. In Section 3.2.2, a survey of the off-line algorithms is given, and this is followed by the work on concurrent reorganization in Section 3.2.3. For a detailed survey of related reorganization work, including reorganization of the logical DB design and the issues involved, see the survey by Sockut and Goldberg [SG79] as well as the survey by Sockut and Iyer [SI93].

3.2.1 Reorganization Utilities

The work on utilities assumes that the DBA or a reorganization decision algorithm determines:

- when to activate the utility;

- how much to reorganize; and
- which utility or set of utilities are used to construct a reorganization strategy.

In this section, we survey a few concurrent reorganization utilities. Some utilities for concurrent reorganization exist in commercial products to redistribute or rebalance a parallel data placement (e.g., the RELOAD command and other utilities in the Tandem NonStop-SQL product [CC93, Smi90, Tro96] and the REDISTRIBUTE command in the IBM DB2 PE product [BFG95]).

Omiecinski describes a concurrent method to convert an index structure from a B+tree to a linear hash file when dynamic access is more predominant than sequential access [Omi88]. The method allows the partially created linear hash file to be accessible during a reorganization so that performance benefits from the reorganization can be reaped as early as possible. An interesting aspect of this work is the performance studies. To compare different reorganizations under various scenarios, he defined a *break-even point* as the number of B+tree pages that must be converted to obtain the same workload throughput during a reorganization as the throughput when no reorganization is done. A reorganization decision algorithm could use this measure to select the candidate reorganization strategy with the shortest break-even point.

Omiecinski et al. developed a concurrent reclustering method that made use of *differential files* [OLS92]. These files temporarily hold updates on pages currently being reorganized so that users can execute concurrently with the reorganization. This work also demonstrated the effects of intrusion on the workload throughput during and after a reorganization. When a reorganization was given minimal resources to execute, the throughput of the workload was minimally reduced. However, the reorganization had longer execution times, and the throughput improvement with the reclustered relation was not realized as quickly.

Srinivasan developed concurrent index creation methods, and introduced a performance metric called the *loss metric* to compare the different creation methods [Sri92]. This metric is equal to the reorganization's response time multiplied by the difference between the throughput without the reorganization and with the reorganization. If the reorganization allows more queries to concurrently execute during reorganization, the response time will increase, but there will be a decrease in the loss in throughput during its execution. Thus, this calculation trades-off the change in the reorganization's response time with the change in the allowed throughput during the reorganization's execution. The drawback of this metric is that it can only compare reorganization strategies that lead to the same DB design, and no limit was set for how soon a strategy must achieve its benefit. A better metric is thus required.

3.2.2 Off-line Algorithms

The off-line algorithms proposed previously attempt to answer the following questions:

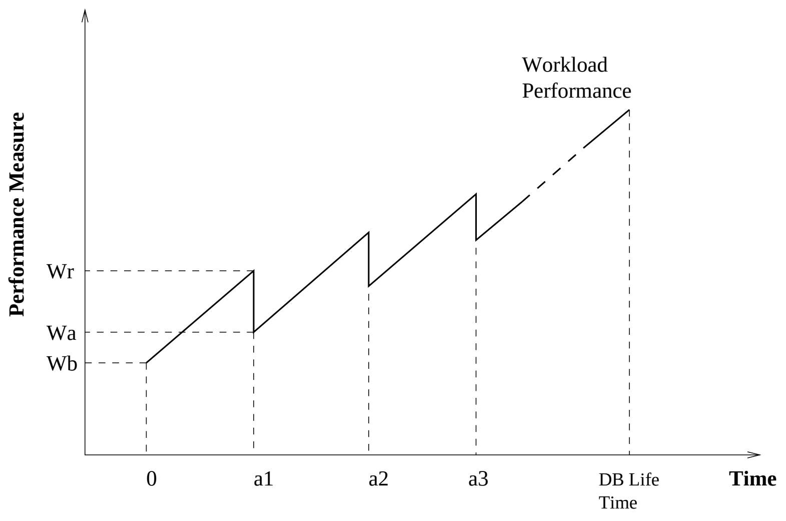


Figure 3.1: The workload performance curve on a serial DB system including reorganizations.

- when to reorganize;
- how to reorganize; and
- what to reorganize.

Some of the work on reclustering on a single node concentrates on the question of when to reorganize [Shn73, YDT76]. Workload response time was assumed to be a linear function of the database size between any two reorganization executions. A reorganization lowers the workload response time by reclustering off-line, and it incurs a cost that is also a linear function of the DB size. With both these response time formulas, the workload performance curve is calculable using the workload response times between reorganization executions along with the effects of the response time after each reorganization is executed. Hence, formulas were developed to determine the times when a reorganization should execute (*reorganization points*).

For example, Figure 3.1 shows a workload's performance including reorganizations. The reorganization points are shown at times $a1$, $a2$, and $a3$. The workload is assumed to have its initial performance at Wb, and when the reorganization begins, its performance measure has increased to Wr. A reorganization is assumed to improve the performance of the workload to Wa. It is assumed that the workload performance curves before and after each reorganization point are linear with given slope. Using the linear functions, the starting performance (Wb), and the database's lifetime allows the area under the performance curve to be calculable. The reorganization points are determined as the points that minimize the area.

Shneiderman determined the reorganization points assuming the times between reorganizations were of equal length [Shn73]. Yao et al. calculated these points heuristically based on the recent past reorganization points [YDT76]. These methods demonstrated that a reorganization decision should take into consideration the reorganization's cost. However, their linear response time functions for the workload reorganizations are not realistic for (parallel) DB systems because:

- response time is typically not linear over different DB sizes and varying workload;
- changes to the system environment can occur at any time, and thus cause reorganizations at any time; and
- adjustments to the changes include more techniques than just reclustering, and using different techniques at different times also causes the response time performance to not be linear over the DB's lifetime.

Rivera-Vega et al. describe a method to answer the question of how to reorganize [RVVN90]. They determine a schedule for redistributing relations across a distributed system such that the reorganization execution time is minimized. Since the redistribution is off-line, the schedule with least intrusion is the one with lowest execution time. They formulated the problem as an integer linear programming problem. This method is useful for constructing an algorithm to select the reorganization strategy for a specific reorganization.

In deciding what to reorganize, heuristics have been suggested. Wolf solved the file re-assignment problem of placing whole files at each disk in a multi-disk single-node system by using a greedy heuristic [Wol89]. The heuristic attempted to move N files to obtain lower I/O costs, and included the reorganization costs as a constraint on N. Besides considering the benefit and cost of a reorganization, the work also demonstrated that the old file allocation influenced the choice of the next allocation.

Ames and Foster developed a heuristic for the file re-assignment problem on a distributed system [AF77]. Their reorganization algorithm first uses the static file allocation algorithm to get the next best allocation. Another decision then determines the reorganization strategy to create this new allocation. The chosen strategy is required to have a cost that is less than the performance benefit from the new allocation relative to the old allocation. Unfortunately, the method does not define how this reorganization cost is obtained.

Copeland et al. presented a dynamic data placement algorithm similar to their static placement algorithm, and this dynamic algorithm determines a reorganization strategy [CABK88]. However, their method only allows this strategy to execute if the estimated work required for the reorganization (its calculation and units are not defined in the paper) is less than the performance improvement provided by the new placement. This performance improvement is measured as the difference between the throughputs associated with the new and old placements. The relations to move are selected as the ones with highest access frequency at the nodes that exceeded pre-defined utilization thresholds at the disks or processors. When re-placing the problem relations, the other relations are assumed to have already been placed. The algorithm places the partitions of the most frequently accessed relations first onto the lowest utilized nodes thereby resulting in a close to balanced load across the system.

A similar rebalancing algorithm to Copeland et al.'s was given by Hua and Lee, where the access frequency is replaced with the size of the partitions [HL90]. The grid cells of a multi-dimensional grid file are assumed to be the fragments that are placed on the nodes. As in Copeland et al.'s method, the reorganization algorithm is initiated when the utilization of a resource at a node exceeds some threshold, thus exemplifying a simple method to determine when to start a reorganization.

3.2.3 Concurrent Reorganization Algorithms

Concurrent reorganization algorithms have been developed for file migration and dynamic file assignment. A survey of the differences between these areas and the issues involved was presented by Gavish and Sheng [GS90]. File migration takes into account only local information at node pairs and moves files dynamically in an ad hoc fashion to adjust some problem in the system (usually reducing the load at highest utilized nodes). However, the dynamic file allocation problem uses global information about the workload, DB, and system to adjust the file placement so as to improve the workload performance. In this thesis, we are more interested in the techniques using global information.

Weikum et al. presented a dynamic file allocation method for disk arrays that is similar to a migration approach because, on a file creation or expansion, the partitions of a file are moved from the highest to the lowest utilized disks [WZS91]. In this and in other migration work, reorganization intrusions on the workload are ignored.

Brunstrom et al. describe a method to dynamically redistribute partitions across a two node system [BLS95]. The redistribution is initiated because a node's utilization threshold is exceeded. Movements are made from the highest to the lowest utilized node, and the cost of the moves are not considered.

In contrast to the work that ignores reorganization costs, Sockut developed a method to model a reclustering that executes with lower priority than the workload queries on a single disk system [Soc78]. This method determines the reorganization cost and whether a reorganization is actually desirable. A disk is modeled as a server with a priority queue, and the user workload and reorganization strategy are assumed to request service from the server. This modeling includes queuing delays in the performance estimates for reorganization. However, only simple disk accesses are modeled, and the reorganization always has lowest priority. Also, the average performance is obtained in a computationally expensive manner by constructing a Markov chain with state transitions.

Vingralek et al.'s method considers reorganizations to redistribute fragments across a network of workstations. The redistribution is usually carried out at low priority. However, the method increases the priority of a reorganization when the workload causes a resource to be 100% utilized without reorganization [VBW95]. The priority level chosen for a reorganization may also be affected by:

- the complexity of the queries;
- a bottleneck resource that is less than 100% utilized; and
- the amount of data to reorganize.

However, Vingralek et al. did not consider these aspects. They developed a storage method called SNOWBALL. Thresholds are used to initiate a reorganization decision, and the problem fragments are moved from the highest to lowest utilized nodes to balance the load across the nodes.

As can be seen, it is assumed in the literature that a lower priority is usually desirable, but not much experimental evidence is given to justify this assumption. In some cases, it may be desirable to have a higher priority level for the strategy because a physical DB design may yield poor enough performance to warrant a quick and drastic change to the design, or the new design is so much better that the delay from the strategy will quickly be overshadowed by the benefit after the reorganization.

3.3 Cost Estimators For DB System Performance

Cost estimators of DB workload performance can be categorized into two groups:

- single query estimators for optimization; and
- workload estimators.

These estimators are further characterized by the features shown in Table 3.2. These features are created to gauge an estimator's applicability for estimating workload performance on parallel relational DB systems. We favor the methods that estimate the parallel execution of workloads including congestion delays and communication costs because they give more realistic estimates. The features we consider are based on the following questions:

1. Does the method estimate congestion delays at the system resources?
2. What type of cost calculation is used, e.g. queuing network models (QNM) or some weighted sum of the resource times (given as *Weighted* in the table)?

Reference	**Congestion Delays**	**Cost Calculation**	**Multiple Queries**	**Comm.**	**Parallel System Estimation**	**Query Plans**
[CLYY92]	NO	Weighted	NO	YES	YES	SQL op. trees
[HM94]	NO	Weighted	NO	YES	YES	SQL op. trees
[GGS96]	NO	Weighted	NO	YES	YES	SQL op. trees
[TO78]	NO	Weighted	YES	NO	NO	Network DB plan
[Pad92]	NO	Weighted	YES	YES	YES	SQL op. trees
[Ser84]	YES	QNM	YES	NO	NO	Hierarchical DB plans
[OPSS89]	YES	QNM	YES	NO	NO	Control flow graphs
[Cas86]	YES	QNM	YES	NO	NO	Network DB plans
[Hys91]	YES	QNM	YES	YES	YES	SQL op. trees

Table 3.2: Summary of query and workload cost estimation work.

3. Are multiple queries estimated together?

4. Are communication costs estimated?

5. Does the method estimate parallel query executions?

6. What type of query plans are used, e.g., SQL operation trees, network DB query plans, hierarchical DB query plans, or a control flow graph annotated with probabilities to denote how often one operation is preceded by another?

We begin the survey by discussing the estimators for optimization. We focus on the estimation methods for parallel optimization. These estimators are similar because they calculate a single query's expected response time as the time to traverse the critical path in the query operation trees (i.e., query plans) when executing on a parallel SN system. Each operator in the path contributes to the estimated time, and its time is calculated as a weighted sum of communication, disk, and processor costs, along with the time to execute the operator's children. If a child pipelines its result to its parent, the parent's time is calculated as the maximum of its resource usage times and its child's estimated time. Chen et al. [CLYY92] as well as Hasan and Motwani [HM94] determine the estimations assuming that each operator fully executes on a single node, so they do not include intra-operator parallelism estimation. Ganguly et al. consider intra-operator parallelism [GGS96].

Because the timing formulas of all these optimization methods are just weighted sums of the estimated resource times, no congestion delays at the resources are considered. We consider the traversal of operation trees to determine a query's cost to be useful, but a different technique to estimate operator timings that includes congestion delays is required.

Instead of predicting the costs of single queries, methods were developed to predict workload costs. Teorey and Oberlander created a DB design evaluator to estimate the performance of physical DB designs for network databases [TO78]. The estimator uses processor and disk times in a sequential system, and determines the execution costs, without congestion delays, as a weighted sum of the I/O and CPU service times for the different operations in a plan.

Padmanabhan estimates the workload response times and throughputs of SQL operation trees for a parallel SN relational DB system [Pad92]. This method also obtains the cost of a query by calculating the time for its tree's critical path. Each query's cost is multiplied by some DBA defined factor (e.g., 10%) to include a rudimentary estimate of locking and resource contention. Workload performance is then calculated as a weighted average of these inflated single query costs.

To better estimate workload costs that include congestion delays, Sevcik [Sev81] constructed a layered framework. The highest layers involved the logical DB design, and the lowest layer modeled the execution of queries on the physical resources with a queuing network model (QNM), as defined by Lazowska et al. [LZGS84]. In Sevcik's method, each resource is modeled as a server with a queue, and a query's execution is broken down into its service times required for each resource. Because queuing is considered, the technique estimates the congestion delays that each query incurs due to the others.

Several applications have been created to apply Sevcik's framework to different DB systems. Orlando et al. developed a layering approach for their Database Analyzer and Predictor CASE tool to support design selection and tuning for single node DB systems [OPSS89]. However, they assume the given query plans are control flow graphs, where operations are connected to other operations by edges, and each edge has a weight corresponding to the probability that the edge is traversed. The number of times an operator is visited is used to calculate a weighted sum of the service times for each operation. This number of visits is converted to a input flow rate into the operator, and its value is determined by equating it to the flow out of the operator. However, this method is not easily used since it is difficult to construct control flow graphs with probabilities from parallel DB queries. Methods directly using query plans, possibly output from an optimizer, are more convenient.

Two methods by Serry [Ser84] and Casas [Cas86] used the query plans for a hierarchical or network DB on a single node. Serry used Sevcik's framework to estimate the workload costs over a hierarchical DB [Ser84], while Casas created an analytic predictor called PROPHET to estimate the workload costs for network and hierarchical DB systems [Cas86]. Their work defined the parameters that each layer required as input from a higher layer or an external user, and provided methods to calculate the outputs from each layer using these parameters. They also described how the service times for a QNM can be derived from the query plans. In their approaches, the processor and disk timings are estimated.

Hyslop expanded upon the techniques by Sevcik and Casas to create a layered estimation method for a relational DB system that was applied to a parallel SN Teradata system [Hys91]. SQL-like operation trees are used to determine the service times of the resources. Communication times are included along with formulas to derive the parallel execution times for a query and its operators. To estimate the parallel execution time of a query, he first uses a QNM that estimates sequential processing to determine the query's residence times, which include the delays due to other queries. Speedup factors are then applied to these times to scale them down so as to approximate the times a query spends at a service center during a parallel execution. These scaled-down resource times are then summed to derive a query's response time. Since the speedup factors are determined for the whole query plan, the different degrees of parallelism involved for and the interactions between the different operators in the plan are not fully accounted for.

None of the previous methods consider how to estimate the effect of priority scheduling of different queries at a resource. Including this estimation would allow us to predict the one-time execution cost of a reorganization strategy having a priority different from the workload query priorities. Bryant et al. developed a mean value analysis (MVA) technique that estimates performance under priority scheduling for a multi-class QNM, including batch and transaction classes [BKL$^+$84]. This method estimates the time a class spends at a resource (*residence time*) by including: (1) the service time for the query at the resource; (2) the services times of the higher and equal priority queries that are already queued for service; and (3) the service times of higher priority queries that arrive while the query is still unfinished. Two scheduling techniques were modeled:

1. preemptive-resume (PR) scheduling, where a higher priority query can preempt the service of a lower priority query so as to quickly gain access to the service center; and

2. head-of-line (HOL) scheduling, where a lower priority query, once started, is allowed to complete its service without preemption from later arriving higher priority transactions.

Because the exact MVA method to model priority scheduling is a computationally expensive technique, Eager and Lipscomb formulated an approximate MVA (aMVA) approach, and described the technique that can be used when the workload is modeled with closed classes [EL88]. The method changes the residence time calculations by including the service times of higher and equal priority queries that obtain service before a given query. They applied the approach to three different MVA priority approximation techniques:

- MVA-shadow approximation;

- Bryant-Krzesinski-Teunissen (BKT) approximation; and

- Chandy-Lakshmi (CL) approximation.

By using various scenarios, such as different loads on the service centers, their results demonstrate that the CL method gives the best results. However, some quantities required for CL are harder to estimate.

In our reorganization chapter, Chapter 6, we will use the BKT approximation with the PR scheduling. The query classes in the workload are assumed to be modeled as transaction classes with given arrival rates. Priorities for processes representing these classes will all be the same. The QNM class for a reorganization will be given a priority that is either lower, higher, or equal to the priorities for the workload classes. Given the classes and priorities, we will estimate (1) the average intrusion the workload classes incur from the reorganization class and the intrusion the reorganization incurs from the workload, (2) the reorganization strategy's average response time, and (3) the average workload response time when the reorganization is executing concurrently with the workload classes.

Chapter 4

Selecting the Initial Physical Database Design

In this chapter, we introduce algorithms to select a physical DB design for a set of relations on a parallel DB system. The algorithms determine the following elements of a physical DB design:

- indexing keys;
- clustering keys;
- partitioning keys;
- relation groups; and
- data placement.

These algorithms are important for the following reasons.

- They consider the partitioning, indexing, and clustering decisions together for a parallel SN system.
- They allow the option of determining any subset of the clustering, indexing, and partitioning decisions.
- The algorithms are applicable to many parallel shared-nothing DB systems.
- Design choices are based on actual DB queries of the workload rather than simple reads and writes as in previous index selection algorithms.

Since there are few existing partitioning key selection algorithms, and since the direct selection of relation groups has not been done before, our algorithms are new to the database literature.

Our algorithms differ from other algorithms because a DB optimizer is used to determine the query plans resulting from a design, and designs are found that minimize the resulting average response time performance of a DB workload.

A DBA would use one of our algorithms in the following way. The DBA must first collect all the necessary inputs for the algorithm. These include the inputs relating to the workload, DB, and system structures as defined in Section 2.1. The method of collection is also defined in that section. Our algorithms require the use of the DB optimizer that is used in the DBA's system. The optimized query plan for each query is produced by the optimizer and provided as input to our cost estimator. A DBA must indicate which decisions are already bound, and which are to be made by an execution of the algorithm. For example, if the indexing and clustering strategy has already been chosen, then that information is provided as input when the algorithm is invoked to do partitioning. Similarly, if the partitioning is already determined, then it is input to the algorithm before the indexing and clustering are determined. When all the decisions remain to be determined, no physical DB design information is included in the input. The output consists of the desired physical DB design decisions made by the algorithm, as well as the resulting estimated ART that will result from using the DB design.

The algorithms we present are:

- the PARtitioning, INdexing, clustering, and relation Groups selection algorithm (PARING);

- the Independent-Relations algorithm (IR); and

- the Combined-Relations algorithm (CR).

Among these algorithms, the PARING algorithm, described in Section 4.1, locates a design that yields the lowest expected workload average response time. Section 4.1 also includes a description of the components of the algorithm, some of which (e.g., relation group selection) are used by the other algorithms. The IR algorithm makes partitioning key decisions independently for each relation, but does not determine indexing or clustering. In contrast, the CR algorithm considers the dependence between relations in making its decisions by investigating sets of partitioning, indexing, and clustering keys from different relations together, and it ranks the attributes (in terms of their use in improving a workload's performance) to consider only the most useful keys. This algorithm is based on an existing index selection algorithm. IR and CR are described in Section 4.2. Note that, for PARING and CR, the problem is tackled by avoiding the use of the separability property [WWS85] as much as possible.

4.1 PARING Algorithm

Our PARtitioning, INdexing, clustering, and relation Groups selection algorithm (PARING) is based on a branch-and-bound strategy. Standard branch-and-bound algorithms are guaranteed to find optimal solutions based on the cost model and assumptions made for the algorithm [LW66, Fle87, PR88, RND77]. A solution is a set of decision variables along with the values that are assigned to them. Two types of solutions exist. A solution that satisfies all constraints and in which all decision variables have been assigned values is called a *complete solution.* A solution is called a *partial solution* if some decision variables without values can be bound such that a complete solution is reached. Thus, a partial solution is a subset of any complete solutions derived from it. The goal of the standard branch-and-bound as well as our PARING algorithm is to find a *complete* solution (S_{opt}) to the problem such that the `cost` for S_{opt} is a minimum. Our method differs from a standard branch-and-bound approach in that it uses strategic pruning based on a tolerance parameter (r) so that branches are not explored if the cost of the partial solution is already too close to the full cost of a best complete solution found so far. We will first describe the standard branch-and-bound approach. This explanation will introduce some of the components included in the PARING algorithm.

A standard branch-and-bound algorithm structure is shown in Figure 4.1. In the standard branch-and-bound algorithm, solutions (S_m) to the problem are found, where a solution is fully defined by specifying values for each of a set of decision variables (V) from an allowable set of values (L).

Solutions are comparatively evaluated using a cost function, $\texttt{cost}(S_c)$. If S_c is a complete solution, then $\texttt{cost}(S_c)$ is an estimate of the quality of the solution, S_c. If S_c is a partial solution, then $\texttt{cost}(S_c)$ must be no greater than the cost of the *best* complete solution that can be obtained by any possible binding of the decisions that are not yet bound in S_c.

The best (lowest cost) known complete solution at any time during the execution of the algorithm is known as $S_{\min}$. Initially, when no complete solution has yet been formed, $\texttt{cost}(S_{\min})$ is considered to be infinity. However, an arbitrary complete solution could also serve as the initial $S_{\min}$. Subsequently, each time a new complete solution is formed, if its cost is less than $\texttt{cost}(S_{\min})$, then $S_{\min}$ and $\texttt{cost}(S_{\min})$ are both updated to reflect the newly found best solution so far. Therefore, the best known $S_{\min}$ found so far has $\texttt{cost}(S_{\min})$ that serves as the most up to date lowest upper bound on the algorithm's best resulting solution cost. The solution ($S_{\min}$) that is left after the algorithm completes the search is the solution (S_{opt}) with the lowest cost found.

A cost function must have the following property to allow the search algorithm to function correctly: If S' and S'' are partial solutions that differ only in that one decision variable that is unbound in S' has been assigned in S'', then $\texttt{cost}(S') < \texttt{cost}(S'')$. That is, as additional

```
algorithm Standard Branch-and-Bound

input: A set $V$ of decision variables and a set $L$ of all possible values
output: A solution $S_{opt} \subseteq V \times L$ such that $S_{opt}$ is a complete solution and
        $cost(S_{opt}) \leq cost(S_{cmpl})$ for any complete solution $S_{cmpl}$.
definitions: $S = \{S_1, S_2, \ldots, S_{|S|}\}$ is a set of some of the partial solutions
        previously examined by the algorithm such that $S_i \subseteq V \times L$, $S_i \neq S_j$ for $i \neq j$.
        $S_i$ is a set of decision variable value pairs
        with variables from $V$ bound to values in $L$.
        $S_{\min}$ is the best complete solution found so far, and
        $S_{opt} = S_{\min}$ on completion of the algorithm.
        $S_{null} = \{\emptyset\}$ is the empty solution.
        cost is a monotonic function on solutions $X \subseteq V \times L$
        such that cost$(X) \leq$ cost$(X \bigcup Y)$ with $Y \subseteq V \times L$.
1. initialize: $S \leftarrow \{S_{null}\}$ and cost$(S_{null}) = \infty$
2. $S_{\min} \leftarrow G$, where $G$ is any complete solution and
        cost$(G)$ is the initial upper bound.
        {Note: if $S_{\min}$ is not initialized, the upper bound is $\infty$ }
3. while $S$ is not empty do
4.      find a solution $S_m$ in $S$ such that cost$(S_m) = \min\{$cost$(S_i)\}$, $i = 1, 2, \ldots, |S|$.
5.      $S \leftarrow S - S_m$
        { Branch from $S_m$ }
6.      Pick a variable $A \subseteq V$ s.t. $A$ is not bound in $S_m$
7.      for each binding $A$ with $A_l \in L$ not contradicting bindings of $S_m$ do
8.          $S_c \leftarrow S_m \bigcup \{(A, A_l)\}$
9.          if cost$(S_c) \geq$ cost$(S_{\min})$ then
                { Prune according to current bounds }
10.             Reject $S_c$
11.         else if other rules (constraints) are violated for $S_c$ then
                { Additional Pruning }
12.             Reject $S_c$
13.         else if $S_c$ is a complete solution then
                { Change best solution found and stop branching from $S_c$ }
14.             $S_{\min} \leftarrow S_c$
15.             for solutions $S_k$ in $S$ such that cost$(S_k) \geq$ cost$(S_{\min})$ do
16.                 $S \leftarrow S - S_k$
17.             end for
18.         else
                { Put $S_c$ in set of partial solutions ($S$) }
19.             $S \leftarrow S \bigcup \{S_c\}$
20.         end if
21.     end for
22. end while
```

Figure 4.1: A Standard Branch-and-Bound Algorithm.

decision variables are bound, the cost function must increase monotonically.

From any partial solution with cost less than $\texttt{cost}(S_{\min})$, one or more decision variables can be bound in all possible ways to form either complete solutions or more complete partial solutions. Any partial solution (S_c) with cost greater than $\texttt{cost}(S_{\min})$ can be discarded since all completions of that partial solution necessarily have cost greater than the cost of a complete solution that is already known. This effectively removes all the solutions that can be derived from S_c since $\texttt{cost}(S_c)$ is a lower bound on all the solutions derivable from S_c. This branching and pruning technique is always a part of branch-and-bound algorithm descriptions [Fle87, PR88]. As a consequence, determining a good lower bound that is as high as possible allows the algorithm to narrow the search space more effectively by allowing quicker progress toward finding a solution that minimizes the objective function.

A branching technique involves determining an additional decision variable (A) that can be bound and added to a partial solution (S_m) to form new partial or complete solutions. Because any unbound decision variable can be bound next and several values for the variable can be chosen, such a technique causes a search tree to be formed where any path between the root and a partial or complete solution is created by a series of additions.

Branching can be done from any of the partial solutions not yet pruned by the algorithm. This set of partial solutions is called S, and each solution in this set has (partial) cost less than $\texttt{cost}(S_{\min})$. In branch-and-bound algorithms, the partial solution (S_m) in this set (S) with lowest bound ($\texttt{cost}(S_m)$) is typically chosen as the next partial solution to extend (giving S_c). If $\texttt{cost}(S_c) \geq \texttt{cost}(S_{\min})$, then S_c can be discarded. Otherwise, if S_c is complete, the $S_{\min}$ is replaced by S_c, and any partial solutions in the set S with cost greater than or equal to the new $\texttt{cost}(S_{\min})$ can be discarded.

We specialized the general algorithm to the physical DB design selection problem, and thus developed our PARING algorithm, which retains all of the main characteristics of the standard branch-and-bound algorithm. In the PARING algorithm, each relation (R_j) has a set (d_j) of decision variables associated with it. This set contains a decision variable for the partitioning key (p_j), the clustering key (c_j), and the set of indexing keys (I_j) for relation R_j, which gives us $d_j = \{p_j, c_j, I_j\}$. We also include the set of unassigned keys n_j as the set of keys to explicitly not include as a partitioning, clustering, and indexing key. Thus, the set of all possible decision variables (V) consists of p_j, c_j, I_j, and n_j for each relation R_j. We denote by L the set of values that the variables can take, where the values consist of the possible keys for the DB relations. For example, assume we have the queries as defined in Figure 4.2 using the given DB schema. The set of all single attribute keys for these queries is the set of all attributes defined for relations $R1$, $R2$, and $R3$. The possible multi-attribute keys are not shown here.

A step of our algorithm is to choose a key to assign to a set of decision variables. The keys used in the search are ones that are directly taken from the query structures in the given

```
---- DB Schema ----
R1(A1,A2,A3,A4,....)
R2(A1,A2,A3,A4,....)
R3(A1,A2,A3,A4,....)

---- Query 1 (Q1) ----
SELECT * FROM R1, R2 WHERE R1.A1 = R2.A1 AND R1.A4 > 3 AND R1.A3 = <variable>

---- Query 2 (Q2) ----
SELECT * FROM R1, R3 WHERE R1.A2 = R3.A1 GROUP BY R3.A3, R3.A1

---- Query 3 (Q3) ----
SELECT * FROM R1, R2, R3 WHERE R1.A1 = R2.A1 AND R1.A2 = R3.A1 AND
                              R2.A2 = R3.A2
```

Set of eligible single attribute keys for the above set of queries:

$T_{Q1} = \{R1.A1,\ R1.A3,\ R2.A1\}$

$T_{Q2} = \{R1.A2,\ R3.A1,\ R3.A3\}$

$T_{Q3} = \{R1.A1,\ R1.A2,\ R2.A1,\ R2.A2,\ R3.A1,\ R3.A2\}$

Figure 4.2: Example queries and eligible keys.

workload. To derive a branch-and-bound style algorithm, we formulated a monotonic cost function similar to the objective function (ART). We use the following property derivable from the ART calculation:

$$\frac{\sum_{s=1}^{|Q'|} \lambda_s R_s}{\sum_{s=1}^{|Q|} \lambda_s} < \frac{\sum_{s=1}^{|Q''|} \lambda_s R_s}{\sum_{s=1}^{|Q|} \lambda_s} < \frac{\sum_{s=1}^{|Q|} \lambda_s R_s}{\sum_{s=1}^{|Q|} \lambda_s}$$

where λ_s is the arrival rate of query class s, Q is the set of query classes in the workload, $|Q|$ denotes the number of classes in the set Q, and $Q' \subset Q'' \subset Q$. Thus, the function

$$\frac{\sum_{s=1}^{|Q'|} \lambda_s R_s}{\sum_{s=1}^{|Q|} \lambda_s}$$

with Q' as the argument is a monotonic cost function over supersets of queries. We therefore made PARING select keys for binding by using the queries to guide the search. Each query specifies the attributes it accesses in its execution along with the operations in which the attributes are involved. The algorithm actually consists of two search components:

1. a search for keys that we can assign to decision variables when investigating a query; and

2. adding new queries to the search, which makes the keys from these queries available for assignment to decision variables.

With this query-driven search technique, a solution consists of a set of decision variable value pairs for a set of relations ($S_m \subseteq V \times L$). Each solution is associated with a set of queries over which the keys have been chosen (U_m), and thus the solution can be thought of as a set S_m and its associated query set U_m. A complete solution is a solution with all decisions bound and with its query set (U_m) containing all the transaction classes (Q) in the workload. A partial solution is defined to be a solution with a subset of decisions bound, and is chosen using a subset of the workload classes ($U_m \subset Q$). When all queries have been considered, a complete solution is constructed from the current partial solution (using a module *CompleteSoln*) by binding each unbound variable for the partitioning, clustering, and/or indexing keys of a relation to either a default key or no key if a default for that decision variable does not exist. Using the queries in Figure 4.2, an example of a partial solution is one where $A1$ is the partitioning key for relation $R1$ and $A1$ is the partitioning key for relation $R2$. An example complete solution (over all queries) is $A2$ as the partitioning key for $R1$, no clustering or indexing keys for $R1$, $A2$ is the clustering and indexing key of $R2$, a default partitioning key for $R2$, $A1$ is the partitioning and indexing key for $R3$, and no clustering key for $R3$.

To avoid finding duplicate solutions by different paths of the search tree, we order the queries, and we partition the key sets among the queries, where a key is only allowed to be chosen by the most important query (defined by the order) in which it is used. From the example in Figure 4.2, if $Q3$ were least important among the queries, the keys it can choose include only $R2.A2$ and $R3.A2$ since the other keys it uses exist in the more important queries. Thus, the most important query first using a key K is used for deciding whether to consider the key or not, and the algorithm either binds the key to an unbound (partitioning, clustering, or indexing) decision or ignores the key when the query is encountered. If the key is ignored, no other query can consider this key in completing the solution. This process restricts the ordering of when keys are added, but does not restrict what keys can be chosen with other keys. Since a single key can be assigned to be a partitioning key, a clustering key, an indexing key, or any subset of these three key types, a branch is created for each assignment. Another branch is also created to denote that the key should not be assigned at all (using the appropriate set n_j). There are restrictions on the assignments:

- A relation can have at most one partitioning key.
- At most one clustering key can be chosen for a relation.

The search for the keys is constrained by selecting the keys that reduce the response times of the most important query first, then the second most important query, and so on. Since the most important queries generally add the most to the average response time calculation, obtaining good response times for these queries will cause the resulting average workload response time to be good as well. Thus, PARING attempts to achieve a solution with a good average workload response time as quickly as possible by binding decisions that assist the queries that affect the average response time the most.

For each iteration of the algorithm, we pick a query and assign one of its keys to one or more of the eligible decision variables. For example, a key could be assigned to be a partitioning key only or a partitioning and indexing key. We could also explicitly assign the key to the set n_j of its relation to signify it should not be assigned. A branch is created for each possible assignment. With a given solution, there is an option of either continuing the search to the next query or adding more keys from the current query to expand the set of keys bound for this query. To distinguish these two possibilities, we create two distinct solutions, where one is delimited by the keyword $SEARCH$ to denote that a new query should be used to search from this solution, and the keyword $EXPAND$ to denote that the solution should have more keys added from the last searched query in the solution. In continuing to the next query, all the keys that are unbound in the last searched query within the solution are implicitly assigned to the set n_j for their corresponding relations to signify that these attributes cannot be used in any assignments of the less important queries.

A cost function (shown in Figure 4.3) then uses the set of chosen partitioning, indexing, and clustering keys along with the queries to determine the relation groups, data placement, and optimized query plans. This function uses the generated groups, placement, and plans to obtain and return a measure ($PART$) similar to the average response time for the queries. We assume the set of relation groups and the data placement are implicit inputs to PART throughout our description of the algorithm. We constructed PART to be a monotonic function so that it and a search driven by the queries and keys are appropriate for a branch-and-bound style algorithm. This PART measure is used to compare the resulting performance from the assigned keys, groups, and placement to the current upper bound, and the algorithm rejects the solution if the bound is exceeded. Note that when a solution causes the workload or any subset of the classes in the workload to saturate the system, the cost will be infinite, and thus the solution will always be pruned. The cost function can be considered as a lower bound, as in the standard branch-and-bound algorithm, which accounts for the response time of the queries already searched to construct a binding of some of the decision variables, and includes the minimum resource cost for the queries yet to be considered. This minimum cost calculation is discussed with the PART function later in this chapter.

Some of the main components of the PARING algorithm include:

- the monotonic cost function (`cost`), outlined in Figure 4.3;
- a best known complete solution ($S_{\min}$ determined using all the transaction classes (Q) in the workload) found so far by the algorithm;
- an upper bound with which to prune solutions away from the search ($\texttt{cost}(Q, S_{\min})$);

```
algorithm Cost Function: cost($U, S_k$)

input: The DB, workload, and system information, the set of queries ($U$)
        to take the cost over, along with a set $S_k$ of bound decision variables
        indicating the partitioning key $p_j$, clustering key $c_j$, and indexing key set $I_j$
        for each relation $R_j$ in the database, and
        the minimum costs ($M_i$) for each query $i$ (implicit inputs), where these
        minimum costs are calculated externally to be the minimum resource
        costs required for query $i$ using any set of keys for that query.
output: A measure of the cost for the queries in $U$
        based on the bound decision variables, i.e., $PART(U, S_k, \{M_i\})$, where PART
        is a function similar to one estimating average workload response time,
        and implicitly includes the decisions made for the relation groups,
        data placement, and query plans as input.
1. Form the relation groups ($G$) using the partitioning keys from $S_k$
2. Determine the data placement for the relation groups $G$
3. Get the query plans for queries in $U$ from the DB optimizer
4. Calculate and return $PART(U, S_k, \{M_i\})$
```

Figure 4.3: The cost function for our algorithms.

- a lower bound (`cost`(U_i, S_i)) on a partial solution (S_i); and
- an initial complete solution (G) and upper bound (`cost`(Q, G)) to the problem.

On completion of the algorithm, the final best known solution ($S_{\min}$) is considered to be the final complete solution (S_{opt}) produced by the algorithm.

The pseudo code for the PARING algorithm is given in Figures 4.4 and 4.5. This code actually defines a *family* of algorithms by using a set of interacting modules. There are a number of possible ways to implement each module, which we will describe in the following sections. However, the implementation of PARING used in the experiments of Chapter 5 uses one specific implementation of each module.

The pseudo code contains the following modules:

- *OrderQueries* is the module used to order the transaction classes in the workload for search ordering purposes.
- *GetNextSoln* is used to determine the next solution ($P_m = \{S_m, U_m, E_m\}$) to expand, which is a solution taken from the partial solution set S. This solution will contain the information in E_m of whether to add keys ($EXPAND$) or move to a new query ($SEARCH$).
- *GetQueryToUse* determines the query (N_m) to use to add new keys to the solution in P_m. If we are to get a new query ($E_m = SEARCH$) for the solution, the next transaction

class not in U_m is chosen from the workload (Q). Otherwise, the last searched query for the solution in U_m is returned as N_m.

- With the query N_m and the solution P_m, we use the module *ExpandSoln* to obtain the next key (x_L) that is not in the solution for which we must assign to a set of decision variables (x_V). This assignment of the key creates new solutions for our algorithm to consider.

- Once an eligible binding is chosen (based on rules we will explain in Section 4.1.1), we create two new solutions: one created by the module *MakeExpandSoln* that will be used to consider additional keys from the current query, and the other created by the module *MakeSearchSoln* that will be used to add another query and its associated keys.

To reduce the number of search cases we consider, we have added pruning rules and a set of pre-scanning modules that are executed before the branching and main body of the algorithm. These pre-scanning modules reduce the number of possible keys and solutions we need to consider in the branching. The main pruning is based on the cost for the new solutions we generate. It should be noted that solutions with $SEARCH$ (that each are used to obtain a new query) are the only ones pruned on their cost, since the monotonic property of the cost function relates to supersets of queries and not to supersets of keys within a query. Solutions with $EXPAND$ are not pruned on their costs since adding new attributes to these solutions for the last query searched by the solution may cause the costs to be lower than the solution's costs. We reduce the number of keys considered for each query by eliminating non-useful keys through the pre-scanning module *FormCandidateKeys*. Only keys involved in partitionable, clusterable, and indexable operations are allowed. To add to this *eligible* key set, the module *FormMultiAttributeKeys* determines the viable multi-attribute keys. There are a number of options for this module, one of which allows an iterative method of adding multi-attribute keys on each iteration of the algorithm. This module would be used in conjunction with the module *ConsiderCompletion*. When no iteration is involved, the multi-attribute keys can be determined as part of the pre-scanning phase.

In the following sections, the modules, their possible implementations, and some other aspects of the algorithm are described. Initially, we expand upon our branching technique and modules in Section 4.1.1. This discussion indicates how solutions are constructed and provides the basis for the cost function. Also included in this section is the description of our query ordering. We present the PART function in Section 4.1.2, along with the properties that demonstrate its monotonicity. The algorithm's pruning method is then described in Section 4.1.3, and includes the idea of pruning away partial solutions for which $\texttt{cost}(U_i, S_i)$ is too close to $\texttt{cost}(Q, S_{\min})$. Section 4.1.4 then discusses the different possibilities of selecting an initial solution for PARING, which is done by the module *GetInitialSoln*. The other components include the relation grouping and data placement algorithms, as described in Sections 4.1.5 and 4.1.6, respectively, as well as a pre-scanning phase component, as detailed in Section 4.1.7. This pre-scanning section includes a description of how default and multi-attribute keys are selected for the relations. We then explain the component for estimating the average response time of a workload in Section 4.1.8. In Section 4.1.9, we present an optional rule that a user can add to the search in order to significantly reduce the number of search cases considered and thus the execution time. However, this rule weakens our claims that our algorithm is nonseparable, and results in a solution having no guaranteed relationship with the best solution found without using this rule. In Chapter 5, we demonstrate the effects of this rule on the number of search cases considered and the resulting ART compared the best solution's ART.

4.1.1 Branching Technique

Each query consists of a set of operations that may or may not benefit from the selection of appropriate partitioning, indexing, and clustering keys (e.g., inequality predicates are not assisted by hash partitioning). To reduce the search space of keys, we concentrate on attributes used in query operations that will benefit from partitioning, indexing, and clustering. We define these attributes as being *eligible* attributes, and the operations are called *partitionable*, *clusterable*, or *indexable* operations. An eligible attribute is basically an attribute used in at least one of the following operations: (1) equi-joins, (2) exact match predicate selection (and inequality selections for indexing), and (3) sorts (used for ORDER BY, GROUP BY, and DISTINCT in SQL). The eligible attributes found for each query q_i form eligible keys that are placed into a set (T_{q_i}) by an initial *pre-scan* phase of the relations and query. An example of eligible keys is given in Figure 4.2.

We now discuss the branching related modules and their possible implementations. The *GetNextSoln* module determines the next partial solution to use in the search based on all solutions in S. There are many possible ways to do this:

1. One method is to use the solution with minimum cost associated with it, i.e., choose the solution $P_m = \{S_m, U_m, E_m\}$ in S such that $\texttt{cost}(U_m, S_m) = \min\{\texttt{cost}(U_i, S_i)\}$ for $i = 1, 2, \ldots, |S|$. We could also order the solutions with $EXPAND$ to be better or not as good as solutions with $SEARCH$. With $EXPAND$ being favored, the search becomes more like a breadth-first; with $SEARCH$ being favored, the search is similar to a depth-first search.

2. Choosing the solution with $SEARCH$ that has minimum cost among the previous solutions we just generated would provide us with a depth-first search.

3. Another method would be to give branches that have considered more queries greater

```
algorithm PARING Algorithm Preamble

input: The set of relations and attributes in the DB along with the query set $Q$
    and a tolerance parameter $r$
output: A complete solution $S_{opt}$ with the keys binding
    its decision variables $V$ with values from $L$, where
    $\texttt{cost}(Q, S_{opt}) \leq (1+r) \times \texttt{cost}(Q, S_{cmpl})$ for any complete solution
    $S_{cmpl}$ considered by the algorithm.
    Any relations not given a key in $S_{opt}$ during the search are
    given (by CompleteSoln) default partitioning, indexing and clustering keys,
    which are determined at the start of the search.
definitions: $L$ is the set of keys (each composed of one or more DB attributes)
    used as candidates in the search. $V$ is the set of decision variables
    such that $V = \{p_j, c_j, I_j, n_j$ for each relation $j = 1, \ldots, R\}$, where
    $p_j$ is the partitioning key, $c_j$ is the clustering key, and $I_j$ is the set of indexing keys,
    along with $n_j$ as the set of keys that are explicitly not bound as
    partitioning, clustering, or indexing keys for relation $R_j$.
    $S = \{P_1, P_2, \ldots, P_{|S|}\}$ is a set of some of the partial solutions and the queries previously
    examined by the algorithm. $P_i = \{S_i, U_i, E_i\}$, where $S_i$ is a partial solution
    and $S_i \subseteq V \times L$ with no decisions from $V$ duplicated in $S_i$.
    $U_i$ is the set of queries used to create $S_i$.
    $E_i \in \{EXPAND, SEARCH\}$ and denotes whether the solution should be
    expanded with new attributes from the last searched query in $U_i$ (using $EXPAND$) or
    used to search on the next query not in $U_i$ (using $SEARCH$).
    $N_i$ is the next query to consider from the partial solution $S_i$.
    We have the properties $S_i \neq S_j$ for $i \neq j$, $U_i \subseteq Q$, $N_i \in Q$, and $N_i \notin U_i$.
    $S_{\min}$ is the best complete solution found so far,
    such that $S_{opt} = S_{\min}$ on completion of the algorithm.
    $S_{null}$ is the empty variable-value pair set, and $P_{null} = \{S_{null}, \emptyset, SEARCH\}$ is the empty solution.
    $T_{q_i}$ is the set of candidate keys for query $q_i$.
    $x_V \subseteq V$ is an eligible set of decision variables for a query $N_m$
    that are bound with a key $x_L \in L$ not in queries $U_m - \{N_m\}$.
    cost is a monotonic function (shown in Figure 4.3) based on the relation group
    and data placement algorithms (in Figures 4.9 and 4.11, respectively)
    along with the optimizer and returns a value for the PART cost function
    that is also the lower bound function for a solution
    such that, for $X \subseteq V \times L$ and $U \subseteq Q$, $\texttt{cost}(U, X) \leq \texttt{cost}(U \bigcup Z, X \bigcup Y)$
    with $Y \subseteq V \times L$ and $Z \subseteq Q$.

Execute the PARING algorithm's body given in Figure 4.5.
```

Figure 4.4: The PARING Algorithm Preamble.

```
algorithm PARING Algorithm

1. initialize: Pre − scan: $T_{q_i} \leftarrow FormCandidateKeys(q_i)$ for each $q_i \in Q$ and set $L = \bigcup_{i=1}^{|Q|} T_{q_i}$.
     $OrderQueries(Q)$ to give a descending order $q_1, \ldots, q_{|Q|}$
     $S \leftarrow \{P_{null}\}$, and cost$(S_{null}, \emptyset) = \infty$

2. $T_{q_i} \leftarrow T_{q_i} \bigcup FormMultiAttributeKeys(q_i, L)$ for each $q_i \in Q$
3. $L = \bigcup_{i=1}^{|Q|} T_{q_i}$
4. $S_{\min} \leftarrow GetInitialSoln$ and cost$(Q, S_{\min})$ is the initial upper bound
5. while $S$ is not empty do
6.   $P_m \leftarrow GetNextSoln(S)$
7.   $S \leftarrow S - P_m$
8.   $N_m \leftarrow GetQueryToUse(Q, P_m)$
9.   $x_L \leftarrow ExpandSoln(P_m, N_m)$
     { Branch from $S_m$ }
10.  for each eligible binding set $x_V \subseteq V$ for $x_L$ do
11.      $P_E \leftarrow MakeExpandSoln(P_m, N_m, (x_V, x_L))$, with $P_E = \{S_e, U_e, EXPAND\}$
12.      if $P_E \neq P_{null}$ and pruning rules violated then GOTO Step 10
13.      if $P_E \neq P_{null}$ then $S \leftarrow S \bigcup P_E$
14.      if $x_L \in$ relation $R_j$ and $x_V = n_j$ then GOTO Step 10
15.      $P_S \leftarrow MakeSearchSoln(P_m, N_m, (x_V, x_L))$, with $P_S = \{S_c, U_c, SEARCH\}$
         {Check for pruning with $P_S$}
16.      if cost$(U_c, S_c) \times (1 + r) \geq$ cost$(Q, S_{\min})$ or
            pruning rules in Section 4.1.3 reveal that $S_c$ violates some constraint then
              { Prune according to current bound }
17.           Reject $S_c$
18.      else if $S_c$ is a complete solution (i.e., $U_c = Q$) then
              { Change best solution found and stop branching from $S_c$ }
19.           $S_{\min} \leftarrow S_c$
20.           for solutions $P_k = \{S_k, U_k, SEARCH\}$ in $S$ such that
                 cost$(U_k, S_k) \times (1 + r) \geq$ cost$(Q, S_{\min})$ do
21.              $S \leftarrow S - P_k$
22.           end for
23.      else
              { Put $S_c$ in the set of partial solutions ($S$) }
24.           $S \leftarrow S \bigcup P_S$
25.      end if
26.  end for
27. end while
28. $ConsiderCompletion$
29. EXIT
```

Figure 4.5: The PARING Algorithm Body.

preference, which would cause the search to be more similar to a depth-first search. For example, we could use the following formula to find the minimum solution $P_i = \{S_i, U_i, R_i\}$:

$$\min_i\{\texttt{cost}(U_i, S_i) - (1 - p^{m_i}) \times [ART^* - \min_j\{\texttt{cost}(U_j, S_j)\}]\}$$

where m_i is the number of steps made to get a partial solution P_i (which we measure as the number of queries searched and placed in U_i), ART^* is the average response time of the best known solution found so far by the search, and p is a parameter that affects how close this search is to a depth-first. We would allow $0 \leq p \leq 1$, where $p = 0$ or 1 would lead to selecting the solution with minimum cost, and $p = 0.5$ pushes the search toward a depth-first.

4. For the implementation of PARING in Chapter 5, this module chooses the solution that needs expanding ($EXPAND$) for the query we previously searched. If no solution with $EXPAND$ exists in the solution set, the module would select the solution with the lowest cost function value. This is similar to a breadth-first search at a query. One drawback of this method is seen when a relatively important query uses many keys. There may be many solutions created for the query. However, we have added pruning and pre-scanning rules to avoid many cases.

Once the partial solution is chosen, *GetQueryToUse* chooses the query used to assign new keys to variables. If the chosen solution has $EXPAND$, the last query examined to obtain the solution will be returned. Otherwise, a new query is selected. If the solution is $P_m = \{S_m, U_m, SEARCH\}$, one method that we use in the implementation of PARING in Chapter 5 is to choose the most important query in the set $Q - U_m$.

Given the solution and query (N_m) to consider, the next step is to determine a key to use for binding. The module *ExpandSoln* performs this function. It returns a valid key, which is one that is assignable to decision variables and that is in T_{N_m}. To restrict the branching, we use two rules to limit which keys are eligible:

- *Rule 1*: Each key chosen for binding must not already be in the solution (S_m).

  This rule allows each key to be chosen only once in any solution. For example, in using S_m from Figure 4.6, this first rule would not allow the bindings with $R1.A1$ or $R3.A1$.

- *Rule 2*: Disallow any added keys by a query N_m to the partial solution S_m that intersect with the eligible attribute sets of the more important queries U_m used to construct S_m, i.e., added keys from query N_m cannot intersect any T_{q_i} for any $q_i \in U_m$.

  The second rule is used to enforce the restriction that the most important query where a key K is first used completely determines whether or not K is used in the solution. For example, in using S_m from Figure 4.6, the second rule would not allow the use of $R2.A1$ since Q1 determines whether or not $R2.A1$ is used in the solution. If the key is added, it can be bound as the partitioning key, clustering key, indexing key, or any subset of these three. The choice of whether or not to use this key does not affect the decisions for other keys, so no search cases are missed.

Using queries in Figure 4.2

Partial Solution S_m:

Queries considered: $U_m = \{Q1, Q2\}$

$A1$ *partitions* $R1$.

$A1$ *partitions* $R3$.

Next query to consider: $N_m = Q3$.

Figure 4.6: Example of a partial solution.

Rule 2 is also useful in reducing the number of calls to the query optimizer. It implies that no decisions based on queries considered later in a search affect queries used earlier. This is because a query considered later in the search cannot alter the use of any key that affects the execution of a query considered earlier. Consequently, once a query has been considered, all decisions that affect it have been bound. So it can be passed to the optimizer to determine its final query query plan. The total number of optimizer calls during an execution of PARING is equal to the total number of key additions that PARING made.

The next key chosen by the module *ExpandSoln* can be determined in a number of ways. One method is to order the keys by some importance measure and choose the next most important key yet to be considered. A possible attribute ordering could be based on the accumulated operator weightings found in Appendix C. In these weight calculations, the importance of the operators and the queries in which the key is involved are all used.

When a key is assigned to a set of decision variables, the next solution with $EXPAND$ needs to be generated. If no more keys can be used for expansion at the current query, the solution returned by the module *MakeExpandSoln* will be the empty set, and this signifies that the solution should not be added back to the set of partial solutions S. Otherwise, we would create a solution $P_E = \{S_m \bigcup S_x, U_m \bigcup N_m, EXPAND\}$ in which S_x is created as the set of pairs with each variable in x_V paired with the key x_L. An important question needs to be resolved: what should the cost associated with P_E be when placing P_E in S? One simple method that we use for our implementation in Chapter 5 is to use $\texttt{cost}(S_e, U_e)$ so that these solutions are given similar costs to the associated $SEARCH$ solutions of the same query. Note that the cost used has implications on the solution chosen by the next call to *GetNextSoln*.

The module *MakeSearchSoln* is used to create the new solution (S_c) and its query set (U_c) that is used to cause our search to consider a new query. A choice in this module is on the creation of the set U_c when using the current query N_m and the previously searched query set (U_m) for the old solution (S_m). The new solution is based on a larger set of queries than used for S_m (namely $U'_c = U_m \bigcup \{N_m\}$). We could assign U'_c to be U_c. However, when S_c is obtained using U'_c, some of the next important queries (U_{na}) in $Q - U'_c$ may not have bindings to add to S_c for two reasons. First, the keys in S_c may include all the eligible attributes in U_{na}. Second, the rest of the keys in U_{na} appear in the more important queries in $U_m \bigcup N_m$ but not in S_c, which means these keys have been chosen to be ignored in this solution and all solutions it generates. If the algorithm could identify these queries, it could add them to U'_c to avoid branching through them in the search and to help obtain a cost ($\texttt{cost}(U_{na} \bigcup U'_c, S_c)$) that is as high as possible, which may cause it to exceed the upper bound thereby allowing S_c to be pruned quickly. We constructed a function ($\texttt{findnoadd}(S_c, Q, U_m \bigcup \{N_m\})$) for our implementation in Chapter 5 to find all the queries in $Q - U'_c$ satisfying the above properties and there is no more important query in $Q - (U_{na} \bigcup U'_c)$ than those in $U_{na} \bigcup U'_c$. The next query to choose in the search from S_m will be the most important query that can bind more decisions. Therefore, the set of queries $U_c = U_m \bigcup N_m \bigcup \texttt{findnoadd}(S_c, Q, U_m \bigcup \{N_m\})$ is considered to be the set that constructed the solution S_c.

A key x_L in relation R_j could be assigned to the set n_j in two ways (when searching a query S_m): explicitly and implicitly. A previous attribute found in the search from the same query (S_m) would create a $SEARCH$ solution to the next query that implicitly assigns x_L to n_j. The algorithm could also explicitly assign x_L to the set n_j when x_L is explicitly considered for query S_m. This explicit assignment could potentially generate a new $SEARCH$ solution that could be the same as the $SEARCH$ solution created by an implicit assignment of x_L. Thus, we do not call *MakeSearchSoln* when a key is explicitly assigned to any set n_i so as to avoid this duplication of solutions. $SEARCH$ solutions are created when a value is explicitly assigned to be a partitioning, indexing, or clustering key.

A query ordering is used by PARING to organize when and how keys are considered. Any order could be used by the *OrderQueries* modules. One possibility is to order the queries on their response times and frequencies. However, for the implementation of PARING in Chapter 5, we extend this ordering because of the existence of update transactions in the workload. For determining the cost of a newly considered query (N_m), we only have to obtain the query plan of this new query. The query plans for the queries considered previously have all been determined. However, on the addition of an indexing or clustering key, any updates (where *updates* also includes inserts and deletes) on the relation containing this key will have their cost increased by the cost of updating the index and the movement of tuples to maintain the clustering. For example, if we use the DB schema as defined in Appendix E, a possible update

($U1$) for this DB is given in Figure 4.7. Now if we were adding an index on $R3.A1$, the cost of the insert in Figure 4.7 would include the cost of inserting tuples into the base relation $R3$ and inserting keys with tuple references into the index on $R3.A1$. This implies that any previously considered updates are affected when an indexing or clustering key is selected on the updated relation. A key of the updated relation may be missing from the query portion of an update. This query portion is used to generate tuples for the update. When the key is missing, this implies the query portion is unaffected by the selection of the key as a clustering or indexing key. In this case, only the added overhead of updating for the key must be calculated. This added overhead will increase the resource usage times determined previously for the update, and thus increase the estimated response time because of added congestion delays.

Update 1 (U1)
INSERT INTO R3 SELECT R1.A1, R1.A2, R1.A3, ..., R1.A21
FROM R1, R2 WHERE R1.A1 = R2.A2 AND R1.A4 > 3 AND R1.A3 = 5

Query 4 (Q4)
SELECT * FROM R2 WHERE R2.A3 = R1.A3

(a) Pruning unused attribute R1.A1
If $S_c = \{p_1 = R1.A1\}$, $U_c = \{\text{U1}\}$, $N_c = \text{Q4}$, and R1.A1 is not used in the query plan of U1, then we can prune away S_c since R1.A1 is not in any of the queries yet to be searched.

(b) Pruning unused attribute R1.A3
If $S_c = \{p_1 = R1.A3\}$, $U_c = \{\text{U1}\}$, $N_c = \text{Q4}$, and R1.A3 is not used in U1 or Q4, then we can only prune away S_c when the attributes that R1.A3 joins with in any of the queries yet to be searched have been considered (namely R2.A3).

Figure 4.7: Examples of pruning with an update.

Because of the additional resource usage for an update on a chosen indexing or clustering key, choosing the key is said to have a *penalty* associated with it. The cost is due to the extra reading and writing necessary to update an index or base relation file structure. Since we chose to consider keys at their first occurrence in the query ordering, an indexing or clustering key's addition benefits only the current query and those yet to be considered. Hence, excluding the current and future queries, the decision to use a key for indexing or clustering can only increase the response times for the set of previously examined updates.

We desire the penalty of adding a key to be included as early as possible since it would present a lower bound on what performance can be achieved with the key. Also, updates always motivate using fewer indexes. The combinatorics of the number of options after considering all updates is limited. Thus, we have ordered the workload to have the updates ordered before the queries. The set of updates and queries are each ordered among themselves using the same importance calculation (frequency times response time). With this ordering, the previously examined queries will always include the updates, and thus all the penalties of having an indexing or clustering key. On selection of an indexing or clustering key by a query, the new penalties for the updates on the associated relation is added to the updates' costs and thus the average response time estimates. Including the penalties as early as possible will give a higher average response time value (PART or WART) that will allow for earlier pruning of some cases.

4.1.2 PART Cost Function

Our special cost function is a *partial workload average response time* (PART) using the set of queries (including updates) examined (U) for a partial solution set (X), and is defined as:

$$PART(U, X, \{M_i\}) = \frac{\sum_{q_i \in U} f_i R_i(X) + \sum_{q_i \notin U} f_i M_i}{\sum_{i=1}^{|Q|} f_i}$$

where Q is the set of queries in the workload, f_i is the relative frequency (or arrival rate) of query class i, $R_i(X)$ is the expected response time of class i using solution X, and M_i is the minimum cost (or time) for class i for any choice of partitioning, indexing, and clustering keys. Because we add in the minimum costs of the queries yet to be considered in the search from solution X, we can consider $PART(U, X, \{M_i\})$ to be a lower bound for the solution that is usable by our branch-and-bound strategy. We use this lower bound to check against the best known solution's ART. If the bound is greater than the best known ART, then this partial solution (X) is pruned. This follows the standard branch-and-bound technique [Fle87, PR88]. The lower bound is also used to choose the next partial solution to consider in the search, as in the standard branch-and-bound technique.

The minimum cost of a query accounts for the minimum I/O and processor times for a query and any choice of partitioning, clustering, and indexing. This minimum cost is calculated using the number of records that the query selects from, updates, inserts into, or deletes from each of the query's base relation. When accessing (or updating) a base relation, the known selectivity will determine how many records are actually read from (or written to) the base relation. If we assume optimistically that these records are evenly distributed across all nodes of the system, the minimum time would be calculated using this average number of records per node. The selectivity of a base relation access gives the number of records that actually match the selection predicates, and this number of records is retrieved whether the relation has or does not have an index. We also consider the records to all be stored together in the minimum number of disk pages, which assumes that the records are clustered in the best possible way for any predicate of the queries. For example, the minimum cost for query Q_3 in Figure 4.2 is calculated from the processor and disk times for reading all the tuples of $R1$, $R2$, and $R3$ after the single relation predicates are applied.

These minimum costs per query are actually derived when determining the initial complete solution by using the selectivities found for the queries in the query plans generated by this solution. Therefore, we avoid recomputing the counts on each computation of PART for any solution. Note that in the pseudo code given in Figures 4.4 and 4.5, these constant minimum costs are implicit inputs to the `cost` function.

The PART function is designed to be similar to the estimated average response time for a given set of queries. It allows the cost function of partial and complete solutions to be directly comparable. In Section 2.2, the average response time of the workload under a complete solution (S_j) is defined as:

$$ART(S_j) = \frac{\sum_{i=1}^{Q} f_i R_i(S_j)}{\sum_{i=1}^{Q} f_i}$$

Since the denominators of ART and PART are the same and since the numerator of PART is a sum of response times of some queries and minimum costs for others, then for partial solution X:

$$PART(U, X, \{M_i\}) \leq PART(U \bigcup Z, X \bigcup Y, \{M_i\})$$

and

$$PART(U, X, \{M_i\}) \leq ART(X \bigcup Y \bigcup W)$$

where Z is a query and Y and W are sets of decision variables and their key bindings such that $U \bigcap Z = \emptyset$, $U \bigcup Z \subseteq Q$, $X \bigcap Y = \emptyset$, $X \bigcup Y \subseteq V \times L$, $(X \bigcap Y) \bigcap W = \emptyset$, $X \bigcup Y \bigcup W \subseteq V \times L$, and $X \bigcup Y \bigcup W$ is a complete solution. Accounting for a good portion of a query's response time by using a good estimate of the minimum cost (M_i) for the query allows $PART(U, X, \{M_i\})$ to be close to $ART(X \bigcup Y)$ for the workload. Thus, partial solutions that would expand to clearly inferior complete solutions are likely to be pruned before expansion since their partial costs would already exceed the cost of a previously identified complete solution. PART defined in this way is monotonically non-decreasing as more queries are considered, and thus can be used as the cost function in a branch-and-bound search.

4.1.3 Pruning away Partial Solutions

The style of pruning described above limits the number of partial solutions as much as possible while still guaranteeing that the best possible solution is found. If the objective is relaxed to require only that the best possible solution has cost at most $r\%$ above the cost of selected solutions, then the computational effort and time required for the branch-and-bound solution can be reduced. Using this relaxation follows the suboptimization alluded to by Lawler and Wood [LW66]. This is accomplished by pruning any partial solution for which $\texttt{cost}(U_c, S_c) \times (1 + r) \geq \texttt{cost}(Q, S_{\min})$. The pruning is possible because our cost function is monotonic. An example of this pruning is given in Figure 4.8 using the queries in Figure 4.2.

Assume $S = \{S_1, ..., S_4\}$ (where only partitioning is considered) and $S_1 = \{p_1 = R1.A1\}$ with $\texttt{cost}(U_1, S_1) = 10$, $S_2 = \{p_1 = R1.A2\}$ with $\texttt{cost}(U_2, S_2) = 15$, $S_3 = \{p_1 = R1.A2, p_3 = R3.A1\}$ with $\texttt{cost}(U_3, S_3) = 39$, $S_4 = \{p_2 = R2.A1\}$ with $\texttt{cost}(U_4, S_4) = 40$, and $S_{\min} = \{p_1 = R1.A1, p_3 = R3.A3, p_2 = R2.A1\}$ with $\texttt{cost}(Q, S_{\min}) = 45$. If we were to expand on S_1 to add $p_3 = R3.A1$, we may obtain the cost of 41 that will cause the partial solution to be pruned since the best known solution has a cost of 45 that is not more than 10% above this new partial solution's cost (for $r = 0.1$).

Figure 4.8: Example of using pruning ratio $r = 0.1$.

Another pruning rule (used in line 12 of the pseudo code in Figure 4.5) only allows a solution (S_E) to be considered if all the keys in the solution are used in at least one query plan. Without this rule, the solution may contain an unnecessary key for the workload. To prune according to this rule, we must guarantee that at least one key is not used in the queries considered so far by examining the query plans directly. We must also guarantee that the key is not in any query (not in T_{q_i}) not yet considered and that additional keys added to the solution will not cause an unused key to become useful to a query plan. The possibility of a key becoming useful on another key's addition is in a join, where the unused key is the join attribute and the other join attribute is the key (K) that could be added. If we have not excluded keys such as K from the solution S_E, we cannot discard S_E. For example, assume we are selecting only partitioning keys, and we have S_m as shown in Figure 4.6 using the queries in Figure 4.2. We then add $R2.A2$ as a partitioning key to obtain a new solution S_E, but this new key is not used in the query plan for $Q3$. Since this attribute is not in any query yet to be examined (because there are no more queries to examine here), and since $R3.A2$ is not allowed to be added to S_m as a partitioning key (since $R3$ has a partitioning key in the solution), then we can safely prune S_E (and implicitly all the solutions it generates) from our search. We also effectively prune all solutions from S_m that will contain $R2.A2$ as the partitioning key because, by adding $R2.A2$ to S_m directly, we have found that no future additions or queries can affect the fact that $R2.A2$ will be unused in all the query plans. Another two examples are given in Figure 4.7 in pruning examples (a) and (b).

From complete solutions that are not pruned, the algorithm can determine a new best solution ($S_{\min}$). A complete solution is determined in the algorithm by having considered all queries (Q) for the solution. When a complete solution changes the upper bound ($\texttt{cost}(S_{\min})$), all partial solutions with costs greater than or equal to this new bound are also pruned at this time. For example, using the queries in Figure 4.2 and the solutions in Figure 4.8, assume we have a complete solution (S_m) derived from S_1 and has a cost of 30. PARING will assign $S_{\min}$ to be S_m, and it will prune away S_3 and S_4 because they now have costs exceeding the newer bound of 30.

4.1.4 Initial Complete Solution

In PARING, the current best solution is $S_{\min}$, and the current upper bound is $\texttt{cost}(Q, S_{\min})$. We require a good initial upper bound and solution so that we can restrict the number of solutions explicitly considered. A low initial upper bound causes a fair number of solutions to be pruned, and thus provides for faster PARING algorithm executions. In reality, any initial complete solution and bound can be used, including no initial solution and infinity as the bound. We could have chosen an arbitrary solution. Our implementation of PARING for Chapter 5 finds an initial solution by examining the queries directly. It does this by determining a ranking for the keys based on how they are used in the queries and by selecting the partitioning keys to be the keys of highest rank for each relation. A description of this method is discussed in a later section, Section 4.2.1.

Indexes are involved in the initial complete solution in two ways depending on whether the DBA is using PARING to determine indexing or not. First, when indexes are to be selected, the initial solution is assumed to contain no indexes so that it is a solution supporting only sequential accesses. PARING will determine the indexes that can be added to improve on the cost of sequential access. Second, if indexes are not to be selected by the algorithm, the indexes are taken from the input DB information. This restriction also applies when partitioning is not to be selected. In this case, the partitioning keys are assumed to be given in the input DB information and no additional partitioning keys are determined by the algorithm.

4.1.5 Forming a Set of Relation Groups

Given the chosen set of partitioning keys, our relation group selection algorithm will group relations that join together if there exists a join between them in which their partitioning keys are the join attributes. The pseudo code for the algorithm is given in Figure 4.9. Note that a relation involved in two different joins, in which all the join attributes are the partitioning keys, will cause a group to be formed containing all three relations.

To demonstrate our grouping, we use the example DB schema and workload in Figure 4.10. In the figure, $<$ *host variable* $>$ represents a variable that could have different values on different invocations of the query. We assume that the partitioning keys of R and S are A and D, respectively, while the partitioning key of Q is G. Relations R and S are joined on their partitioning keys, A and D, so this indicates that they should be grouped together. Since a join with G of relation Q is with the non-partitioning keys of R and S, relation Q is placed in a separate relation group.

The reason for constructing the algorithm in this way is that the optimizer uses partitioning

```
algorithm Relation Grouping Algorithm

input: Relations and their partitioning keys
output: Relation groups and, for each group ($RG_i$), the relations assigned to it.
1. Set $k = 0$ to be the number of currently assigned relation groups.
2. while there exists an unassigned relation $R_j$ do
3.        $k \leftarrow k + 1$
4.        Assign $R_j$ to be in $RG_k$
5.        for each other unassigned relation $R_n$ that is in a join
              with a relation $R_m$ of $RG_k$, where the join attributes are the
              partitioning keys of the corresponding relation do
6.            Assign $R_n$ to be in $RG_k$
7.        end for
8. end while
```

Figure 4.9: Relation grouping pseudo code.

```
---- DB Schema ----
R(A,B,C)
S(D,E,F)
Q(G,H,I)

---- Workload ----
SELECT * FROM R, S, Q
WHERE R.A = S.D AND R.C = Q.G AND S.E = Q.H AND R.B = 5 AND
      Q.I = 5 AND S.F = <host variable (hv)>
GROUP BY R.A, S.D
```

Figure 4.10: Example DB schema and workload involving three relations R, S, and Q.

key information to determine whether a relation grouping can be used to improve performance. When optimizing a join, the optimizer will cause the join to be localized only when both relations are partitioned on their join attributes and are placed in the same relation group. However, the optimizer may not be able to take advantage of the grouping information. In a complex query's plan containing multiple joins, one of the grouped relations may first be joined with another relation in a different group. Since this first grouped relation must be redistributed, the grouping information is no longer useful. Also, a group could be formed to contain relations that do not directly join with each other on the partitioning keys, and the grouping may yield a single group containing all the relations. Since the optimizer will determine whether to redistribute relations not partitioned on their join attributes in order to execute a join, storing the relations in the same group does not change the optimizer's choice. Also, if it is found that the grouping for a specific partitioning key set yields a cost function value that is greater than the cost function values resulting from other groupings and key sets, this grouping is ignored. Thus, we group the relations automatically on the partitioning keys, and let the optimizer decide whether to exploit the relation grouping.

4.1.6 Determining the Data Placement

The data placement component places relations in a relation group together on the same nodes. Thus, the data placement algorithm uses the output from the relation group algorithm. The goal of data placement is to find the number of partitions and the node assignments that lead to the lowest average response time.

Any one of many existing placement algorithms [BAC$^+$90, PB92, GD90, HL90, Wah84, Wol89, CNW83, DF82, MD97] could be used. However, from experience, we have seen that full declustering of relations leads to the best performance for large relations whenever the number of nodes is moderate (e.g., 128 or fewer). We used an algorithm by Mehta and DeWitt as our data placement approach [MD97] because it uses full declustering. The pseudo code for the algorithm is given in Figure 4.11.

In this algorithm, the degree of declustering for each relation R is calculated as:

$$min(N, \lfloor P(R)/ReadSize \rfloor)$$

where N is the number of nodes in the system, $P(R)$ is the number of pages in the relation R, and $ReadSize$ is the number of pages that can be fetched into the disk cache together. Thus, large relations would be fully declustered across all the disks unless N is very large. Relations with declustering degree less than the number of nodes have their partitions placed on the nodes in round-robin order.

In the algorithm, a relation group is placed according to the largest relation. The degree of declustering of a relation group would be the degree of declustering of the largest relation

```
algorithm Data Placement Algorithm

input: Relations, partitioning keys, relation groups, relation cardinalities,
       number of nodes ($N$), and $ReadSize$.
output: Degree of declustering ($D_i$) for each relation group $RG_i$
       (and thus per relation in the group),
       as well as the assignment of each relation group.
definitions: $N_c$ is the current node number at which to place partitions of
       relation groups with $D_i < N$ and where $0 \leq N_c \leq N - 1$
       and the first node has node number 0.
1. for each relation group $RG_i$ do
2.     Set size ($C_{\max}$) of relation group ($RG_i$) to be
           the largest cardinality among the relations of group $RG_i$
3.     Determine number of pages ($P(C_{\max})$) for $C_{\max}$, i.e.,
           $P(C_{\max}) = \lceil C_{\max} t_{\max} / PageSize \rceil$, where
           $t_{\max}$ is the tuple size for the largest relation in $RG_i$
       { Determine degree of declustering }
4.     $D_i = \min(N, \lfloor P(C_{\max}) / ReadSize \rfloor)$
       { Determine the assignment of the group $RG_i$ }
5.     if $D_i = N$ then
6.         Assign one partition of each relation in $RG_i$ to each node
7.     else
           { Assign these groups in a round-robin fashion }
8.         Assign one partition of each relation in $RG_i$ to nodes $N_c$ through
               $N_c + D_i - 1$ modulo $N$
9.         $N_c = (N_c + D_i)$ modulo $N$
10.    end if
11. end for
```

Figure 4.11: Data placement pseudo code.

using the formula in the algorithm by Mehta and DeWitt.

Full declustering often gives a performance benefit because it spreads the I/O across the nodes, which improves the I/O bandwidth, throughput, and response times. In systems where the communication network bandwidths are higher than I/O bandwidths, improving the I/O is most critical.

However, our PARING algorithm also leads to solutions with low communication and processor time for communication depending on the structure of the groups and partitioning key decisions, even under full declustering. For example, in executing a join in which the partitioning keys of the operands do not correspond to the join attributes, communication will result independent of the number of nodes across which the relation is declustered (if this number is greater than one). Our algorithm chooses the partitioning keys that may best serve as the join attributes to reduce communication when this contributes to a low average workload response time.

In this algorithm, we can add the constraint that the disk space available at any node must not be exceeded. If storage space is limited, the algorithm can be extended to ensure that partitions and indexes do not exceed the storage constraints. First, we would calculate the storage space required for the base relations as well as the indexes for a chosen solution per node. The base relation's space in pages is calculated at each node to be the number of records in the partition at the node divided by the number of full records that can fit on a disk page including any free space that should be allowed per page. For a local index on a partition, the space is calculated by summing the number of pointer pages and the number of pages needed at each level of the index structure. For the leaf-level of the structure, the required number of pages is calculated as the number of unique indexing key values divided by the number of key values allowed to be in an index page. To obtain the number of pages for each other level in the index, we would take the number of pages of the next lower level and again divide by the number of key values per page. If the index is a non-clustering one, then the number of records (used to represent the number of base relation pointers) is divided by the number of pointers allowed in a page to obtain the number of indirect pointer pages.

Second, we determine the maximum storage space required for the relations and indexes at any of the nodes, and compare it to the given storage constraint. If the constraint were exceeded, we would reject the chosen solution. Since the base relation size would not change, and since adding indexes to a solution would increase the storage space, the solutions generated from a rejected solution would also cause the space constraint to be exceeded, so these solutions can also be rejected. Thus, with such a storage constraint, our algorithm would find a solution with the best response time that also satisfies the constraint.

4.1.7 Pre-scanning the Queries for Important Candidate Keys

To reduce the number of search cases, the PARING algorithm pre-scans the list of keys (L) to remove any keys that would not assist performance. The module *FormCandidateKeys* performs this function. This pre-scanning includes:

- eliminating attributes with low column cardinality;
- removing attributes not used in partitionable, clusterable, or indexable operations (as described below); and
- determining the key for each relation among the keys not involved in any join of any query that would best serve as the (default) partitioning, indexing, and clustering keys so that the search can concentrate on the join attributes only to reduce the number of search cases.

The default keys are used to complete a partial solution in the *CompleteSoln* module so that we can obtain a solution to check against and possibly become the best known solution so far ($S_{\min}$).

We will now give the rules used for each of these. Later in the section, we will discuss and justify these rules. First, the algorithm eliminates attributes with low column cardinalities when they satisfy the following empirical rules:

- an attribute's cardinality is less than $10N$, where N is the number of nodes; or
- some value of the attribute occurs in more than $1/(2N)$ fraction of the tuples.

Second, *partitionable*, *clusterable*, and *indexable* query operations are operations whose costs may be reduced by the proper selection of partitioning, indexing, or clustering keys:

- joins;
- sorts;
- tuple grouping actions (because of the GROUP BY clause);
- duplicate tuple removal; and
- selections.

Any attribute not used in any of these operations for any of the queries is removed from the set of candidate keys. If all attributes of a relation are eliminated, the partitioning defaults to the attribute with the highest column cardinality in order to partition the relation as evenly as possible across the nodes. Any relation not involved in any join will be assigned to its own relation group.

Third, in order to limit the number of key combinations considered in the PARING search, we determine *default* partitioning, clustering, and indexing keys for each relation. A default partitioning key is a key that:

1. is not involved in a join in any transaction class of the workload; and
2. when chosen as the partitioning key, results in the lowest workload ART among all the other candidate partitioning keys of the relation that are not involved in any joins.

We then use this default key to be the partitioning key of the relation if a join key is not chosen as the relation's partitioning key. A similar rule is used to determine the default indexing and clustering keys for the attributes not involved in any join. The method to determine the default indexing and clustering is to first take the cost of the queries using a sequential scan. Then one by one, determine the cost of adding a non-join indexing or clustering key. If the key's addition lowers the workload response time, it is added to the default set. Otherwise, the key is removed. This default set of keys for indexing includes the empty set, which signifies the default is to use sequential scans for all operations on the relation.

We now discuss and justify our rules. The elimination of low column cardinality attributes is done because attributes with low column cardinality may cause skew when declustered across more nodes than the column cardinality. This would leave some nodes with no records at all, and others with a larger number of records. For example, an attribute describing a person's gender would only have two possible distinct values, and thus would lead to at most two partitions. The imbalance of tuples across the nodes (data skew) resulting from the partitioning of the relation using the low column cardinality attributes would cause load imbalance problems in choosing the physical DB design in a number of ways. First, load imbalance results when a query accesses all the partitions of a relation and the partition sizes are unequal because the partitioning key values have an imbalanced distribution across the nodes. Second, we could also have a load imbalance when a class of queries each accesses only one of the relation's partitions at any one time. For example, a query class performing an exact match on a partitioning key value will most heavily load the node with the most tuples. Since more queries of this exact match class use one node more than the others, this node's resource will limit performance, and cause longer delays for queries.

Low cardinality attributes are not good choices for indexing. Accessing records using an index on a key with low column cardinality results in selections with low selectivities that retrieve a good proportion of the base relations. However, indexes are beneficial when selectivities are high causing only a small number of tuples to be retrieved. In the low selectivity case, an optimizer may find that the cost is lower using a full relation scan over this index scan. In this case, the index is not useful, and so including this indexing key in the search is not warranted.

We now discuss the partitionable and indexable query operations and how selecting their attributes as partitioning, indexing, or clustering keys affects each operation's performance. Joins are facilitated in terms of partitioning if (1) the join attributes are selected as the partitioning keys, and (2) the relations are placed in the same relation group. To illustrate, consider the two relation schemas and the SQL query (Query 1) shown in Figure 4.12. Assume that `Customers` and `Orders` are fully declustered across a set of nodes. Four cases arise:

- C_CUSTKEY and O_CUSTKEY are the partitioning keys and `Customers` and `Orders` are in the same group – then we could compute the join without repartitioning any relation (*localized join*).
- C_CUSTKEY and O_CUSTKEY are the partitioning keys but `Customers` and `Orders` are in different groups – then we would need to *direct* each tuple of one relation to the correct node storing tuples with the same join attribute value in the other relation.
- C_CUSTKEY and O_ORDERKEY are the partitioning keys – then we could just direct each tuple of `Orders` to the corresponding node containing the same join attribute value in `Customers`, thus repartitioning only one relation.
- C_NATION and O_ORDERKEY are the partitioning keys – then we would need to either repartition both relations, or broadcast one to the other.

For a single join, it is best if both relations are partitioned on their join attributes and if they both belong to the same group.

The partitioning key decision is affected by multiple joins in a query and how the joins are executed together in the query plan. An example of this join interaction can be seen if we use Query 2 from Figure 4.12. The query differs from Query 1 because it also requires a join between `Lineitem` and `Orders` using the clause: L_ORDERKEY = O_ORDERKEY. The query would thus be a multi-join query. If the join between `Customers` and `Orders` were to be executed in the query plan first, the join of this result with `Lineitem` could still be optimized by using the partitioning information for `Lineitem`. If `Lineitem` were partitioned on L_ORDERKEY, the result of the first join could be directed to the nodes storing `Lineitem` so that the second join could be done at these nodes without communicating `Lineitem`. The PARING algorithm makes use of this multiple join information because we optimize each query to obtain workload costs where the optimization and hence join order is affected by the partitioning keys selected.

Joins are also facilitated by having indexes (or a clustering) on the join attributes. If the join is carried out by a merge-join, one relation of the join could be read in pre-sorted order from disk using an index (or by clustering as well) on the corresponding join attribute. If the join is a nested-loops join and the selectivity of the predicates on a child relation is high, the relation could be read in using an index on an attribute in a predicate to reduce the disk and

```
---- DB schema ----

Customers(C_CUSTKEY,C_NATION,....)
Orders(O_ORDERKEY,O_CUSTKEY,...)
Lineitem(L_ORDERKEY,...)


---- Query 1 ----

SELECT O_ORDERKEY, ...
FROM Customers, Orders
WHERE ...
AND C_CUSTKEY = O_CUSTKEY
AND ...
GROUP BY O_ORDERKEY, ...


---- Query 2 ----

SELECT O_ORDERKEY, ...
FROM Customers, Orders, Lineitem
WHERE ...
AND C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND ...
GROUP BY O_ORDERKEY, ...
```

Figure 4.12: Simplified DB Schema and queries from the TPC-D benchmark specification.

processor costs for accessing the relation. It should be noted that using the index to read a relation in pre-sorted order also assists the sorting operations in SQL (ORDER BY, GROUP BY, and DISTINCT). These sorting operations and selections in a query can exploit indexes on predicate attributes for a predicate yielding a low number of tuples. Clustering also assists sorting as it can be used to physically order the relation in the sorted order we require.

Sorts, grouping actions, and duplicate removal are all facilitated in a similar manner as for joins when the attributes on which they are based are the partitioning keys of the relation. If the sort key is also the partitioning key, the sort can be done by local sorts on each node. For range partitioning, these local sorts would be equivalent to a global sort. For hash partitioning, when aggregation is involved with the sort, the aggregation of tuples using the same attribute values can be all at one node where those tuples reside, which allows the aggregation to be localized at each of the relation's nodes. Also, after the local sorts for the hash partitioned case, the tuples with the same values are all on the same nodes, so a redistribution to achieve this is not required. This state of local sorts at each node is beneficial for merge-joins.

For selection predicates and partitioning, when the attribute in the predicate is the partitioning key, the selection can be performed at the nodes that are known to satisfy the predicate. For example, if we have an equality predicate on a partitioning key, all qualifying tuples can be found at one node, so the selection can be performed at this node only. To illustrate, assume that a query has a clause: C_NATION = < **host variable** >. If we partition `Customers` on C_NATION, then we will be able to evaluate the query by involving only one node. This is good for the throughput since different queries of this type that use different values for < **host variable** > will likely execute on different nodes, thus balancing the load across the nodes. This single node restriction could reduce throughput if one of the nodes becomes a bottleneck and causes load imbalance because of data skew. An index or clustering could also assist in retrieving tuples satisfying inequality selections, e.g., C_NATION < < **host variable** >.

For default keys, we choose the keys that are not involved in a join and that are the best keys in improving performance. PARING uses keys that are join attributes during its search. When a join key is not chosen for a relation, the default keys are used for that relation, since they would be the best keys among the keys not involved in joins. To allow this in PARING's search, PARING allows the choice of a join's key for a relation or no join key. In the case of using no join key, the key used for the relation must be determined as one among the keys not in a join. PARING would need to choose one of these keys that improves the single relation access. The pre-scanning rule determines the key that improves the single relation access, and assigns it to the default keys. Hence, when a join key is not chosen for accessing a relation, the default key would be the next best choice. The amount of searching in the PARING algorithm is reduced because it uses only the default key in the rest of the search algorithm rather than considering all the keys not involved in any join. Using the default key rule does not affect the non-separability property of the problem because the only keys affected by the non-separability property are the join attributes.

Since we use the join attributes to guide our search for a good partitioning, the result of the PARING algorithm that determines only partitioning is the same as an algorithm where the relation groups are chosen first and then the partitioning keys are chosen. Because of our relation grouping, one grouping is chosen for one possible partitioning key set choice. If a group of relations linked by joins is chosen first, then the join pairs determine what partitioning key sets are worth considering. This specific set of partitioning keys is exactly the way we select our partitioning key sets. We use the join attributes to guide our partitioning key selection and prune all other cases away. However, our pruning has the ability to prune some of these specific partitioning key sets away, and thus our algorithm will execute with less than or equal cost to the algorithm choosing relation groups first. Also, we benefit by choosing the keys first because we can combine the partitioning, indexing, and clustering decisions together.

Along with single attribute keys, we include candidate keys that have more than one attribute (multi-attribute keys). Since we consider partitioning, clustering, and indexing methods that could use single or multi-attribute keys, we identify candidate multi-attribute keys that will be beneficial to query performance.

The multi-attribute keys of interest are keys that are constructed from sets of attributes from the same relation. A set of attributes is also only allowed when all of its attributes are used in a query together with certain types of operations, where the operations are either only on the relation itself or involve only one other relation. The attributes must be used in the following ways:

- the attributes are all used to perform an exact match (or inequalities for indexing consideration) on the relation;

- the attributes are used in a group as join attributes when joining with some other relation; or

- they are used together in a GROUP BY or ORDER BY clause.

For constructing multi-attribute candidate keys, attributes that can be grouped in one of these keys must all be used in the same query. One reason for this is because a query using all attributes in the multi-attribute key as the partitioning key can be directed to a specific partition in the multi-attribute partitioning and thus to a specific node. However, partial matches (matches on fewer than the total number of partitioning keys for a relation) must use more than one node, and possibly all nodes. This would not benefit queries that perform matches on only a proper subset of the attributes of the multi-attribute keys. These partial matches effects apply to the different types of partitioning that use multi-attribute partitioning, for example:

- a grid-file approach, where each attribute corresponds to a dimension in the grid [Gha90, NHS84]; and
- a partitioning approach, where the values for the attributes in the multi-attribute key are combined in a function to return one logical partition number, such as hash partitioning.

Partitioning a relation on the attributes of a multi-attribute partitioning key means that one of these methods is used with the attributes.

Different attributes of a relation used in different queries or each used in an operation of the same query with a different relation (e.g., two joins with a different relation for each join) do not provide the localization benefits that partitioning on a subset of these attributes would. For example, assume we have relations R with attributes A and B, S with attribute C, and T with attribute D. One possible predicate for a query is:

$R.A = S.C \textit{ and } R.B = T.D$

If we partition R on a multi-attribute key made up of A and B, we would only aid the access of the relations by searching on attribute A or B but not both since we execute each pair-wise join one at a time. Now assume we have the predicate:

$R.A = < \textbf{host variable} > \textit{ and } R.B = T.D$

where $< \textit{host variable} >$ is a variable used in a query that would take on a different value in each invocation of the query. This query would be a case where a multi-attribute key is useful. The match on attribute A for a query would limit the number of tuples to examine, and when we match D with B, we can narrow the match to a specific partition of R together with the predicate on A. Hence using A and B as the multi-attribute key for partitioning is a good choice.

Similarly, multi-attribute indexing or clustering can reduce the cost of more than one operation at a time. For example, in the same query, there may be predicates on more than one attribute that are all part of an index key. Using the index for the predicates' execution leads to finding the tuples that satisfy all the predicates at the same time, and thus reduces the disk and processor costs for the predicates' executions. Also, if all the attributes in an ORDER BY clause are used in a clustering key, the relation could be retrieved in a pre-sorted order, and we would avoid the processor and possibly disk costs for sorting the relation if it were not clustered in this way. If an index was on a predicate attribute and a join attribute, we could reduce the cost by using the index to perform the selection and the retrieval for a join at the same time. For an equi-join, the index could quickly find the tuples that match a corresponding join attribute's value.

Limiting the keys to include only these multi-attribute keys enables us to reduce the number of candidates considered by the algorithm. If there was no limit, then approximately 2^n multi-attribute partitioning keys would be considered for an n attribute relation.

The multi-attribute keys can be constructed in a number of ways. One method would be to use the module *FormMultiAttributeKeys* in conjunction with the module *ConsiderCompletion* to form an iterative method of creating the possible multi-attribute keys. This is similar to the method by Chaudhuri and Narasayya, except that multi-attribute keys of more than two attributes are allowed [CN97]. In the first pass through *FormMultiAttributeKeys*, no multi-attributes are created. In the second pass, attribute pairs are formed as the first multi-attribute keys. In the third pass, multi-attribute keys of three attributes are formed, and so on to the k-th pass. One method for creating multi-attribute keys of k attributes on the k-th pass is to use the multi-attribute keys of size $k-1$ in the solution chosen for the $k-1$-st pass and add a new attribute such that all the attributes in the key of size k attributes satisfies the restrictions on the allowable multi-attributes given above, i.e., the attributes must be used together in at least one query in certain operations. For the second pass, single attribute keys are used to form the pairs of attributes.

The module *ConsiderCompletion* would be constructed to allow these multiple passes. On the k-th pass through the algorithm ($k \geq 1$), the module would determine if multi-attribute keys of $k+1$ attributes can be formed. If they cannot, then the module causes the algorithm to proceed to the next step and exit. The best solution on this last pass would be considered to be the best solution found by the algorithm. Otherwise, the module would cause the algorithm to loop back to the point where the *FormMultiAttributeKeys* is invoked again. Thus, another pass of the algorithm can begin. There are a few methods to limit the number of passes (and thus avoid creating $k+1$ attribute keys):

1. We could explicitly accept a user defined limit on the number of passes allowed.

2. Another method would be linked to the module *FormMultiAttributeKeys*, which forms multi-attribute keys of size k from ones of size $k-1$. If no attributes can be added to keys of size $k-1$, then the *ConsiderCompletion* module causes the algorithm to stop.

In our implementation of PARING in Chapter 5, *ConsiderCompletion* causes the execution to proceed to the next step, which does not cause a loop. Also, *FormMultiAttributeKeys* pre-scans the queries to form all the eligible multi-attribute keys only once before the branching. This effectively avoids multiple iterations through the algorithm.

4.1.8 Workload Cost Estimation Method

In this section, we present the procedure used to determine the costs of a workload, individual transactions, and operators within a transaction. The estimations are all based on an open queuing network model (QNM). The QNM terminology used in this section is defined in Appendix A. Also, the detailed operator cost estimations can be found in Appendix A.

To derive the estimated average response time of a workload, we follow the three steps shown in Figure 4.13 using the variables defined in Table 4.1. In the first step, we estimate each transaction's resource visit counts ($V_{s,k}$ and $V_{s,k}^{O_s}$ as calculated in Appendix A) and utilization on each of the resources ($U_{s,k}$). The calculations for the visit counts for an operator involve the following inputs:

- the estimated output and input placement of the operator, which includes the cardinality per node of each input relation;
- the selectivity of the operator;
- the operator type;
- the algorithm required to execute the operator type;
- the input tuple sizes; and
- the nodes on which the operator executes.

The total resource utilizations for the workload (U_k) are then determined, and we check whether a resource is saturated in the second step.

In the third step, these total utilizations are used to estimate each resource's residence times for each transaction ($R_{s,k}$) by using the delays due to other transactions. Actually, it is the operator's residence times ($R_{s,k}^{O_s}$) that are calculated. This is done by using the utilizations to inflate the resource usage of a transaction's operators ($R_{s,k}^{O_s} = \frac{V_{s,k}^{O_s} S_{s,k}}{1-U_k}$). This inflation method is used for modeling transaction class workloads in an open QNM [LZGS84]. The reason for using this formula is because a transaction class's average response time (R_s) is calculated as the cost of the critical path in the query plan tree, which is calculated using the residence times of the operators ($R_s^{O_s}$) in its query plan. Therefore, the inflation must be applied to each operator to account for the average congestion delays or interference each operator (and thus each transaction) experiences from other transactions. Response and residence times (R_s and $R_s^{O_s}$) are also calculated using a transaction's operator placement. We use the following assumptions:

- data from an operator's child is pipelined to it unless explicitly stated otherwise in the query plan (e.g., pipelining does not occur from external sorts); and

```
algorithm: Workload Response Time Estimator

{ Step 1: Determine the visit counts and utilizations of the resources }
for each transaction class $s = 1, \ldots, Q$ and its operators $O_s$, and for all $k$ do
        Calculate $V_{s,k}$ and $V_{s,k}^{O_s}$ for resource $k$
        $U_{s,k} = \lambda_s V_{s,k} S_{s,k}$
end for
{ Step 2: Use the resource utilizations to determine saturation }
for each resource $k$ do
        $U_k = \sum_{s=1}^{Q} U_{s,k}$
        if $U_k \geq 1$ then
            { Saturated system }
            Return $ART = \infty$
        end if
end for
{ Step 3: Calculate response times including congestion delays }
for each transaction $s$ do
        { Calculate response time of the transaction's operators }
        for each operator $O_s$ encountered by traversing
             the query plan for $s$ in a bottom-up fashion do
            for each resource $k$ do
                { Calculate residence times including congestion delays }
                $R_{s,k}^{O_s} = \frac{V_{s,k}^{O_s} S_{s,k}}{1-U_k}$
            end for
            { Calculate execution times of the operator alone }
            $T_{O_s} = R_{s,COMM}^{O_s} + \max_k(R_{s,IO_k}^{O_s} + R_{s,CPU_k}^{O_s})$
            { Calculate the response time of operator $O_s$ }
            $R_s^{O_s} = \begin{cases} T_{O_s} + \max_i(R_{\text{child } i \text{ of } O_s}) & \text{if the children of } O_s \text{ do not execute at the same nodes as each other,} \\ T_{O_s} + \sum R_{\text{children of } O_s} & \text{otherwise} \end{cases}$
        end for
        { Calculate response time for transaction $s$ (critical path of query plan) }
        $R_s = R_s^{O_s}$ where $O_s$ is the root of the query plan
end for
{ Calculate the average workload response time }
Return $ART = \frac{\sum_{s=1}^{Q} \lambda_s R_s}{\sum_{s=1}^{Q} \lambda_s}$
```

Figure 4.13: Algorithm to calculate average workload response time.

Variable	**Description**
O_s	An operator of transaction class s
$children\ of\ O_s$	Children operators of operator O_s in the query plan tree of transaction class s
$S_{s,k}$	Avg. service time of transaction class s on resource k
$V_{s,k}$	Avg. visit count of transaction class s to resource k
$V_{s,k}^{O_s}$	Avg. visit count of operator O_s in transaction class s to resource k
$U_{s,k}$	Avg. utilization of transaction class s on resource k
U_k	Total average utilization on resource k
$R_{s,k}$	Average residence time of class s on resource k
$R_{s,k}^{O_s}$	Avg. residence time of operator O_s of class s on resource k
R_s	Average response time of class s
$R_s^{O_s}$	Average response time for executing the query plan tree of class s rooted at operator O_s
T_{O_s}	Average time to execute only operator O_s of class s without the children of O_s
λ_s	Average arrival rate for class s
$COMM$	Communication resource
IO_k	I/O resource for node k
CPU_k	CPU resource for node k

Table 4.1: Queuing network model variables.

- each operator executes concurrently on the different nodes it uses.

In the response time formulas, the operator cost is first calculated as:

$$T_{O_s} = R^{O_s}_{s,COMM} + \max_k (R^{O_s}_{s,IO_k} + R^{O_s}_{s,CPU_k})$$

and this is used along with the response times of the children of O_s to obtain the cost of the critical path for the subtree rooted at O_s in the query plan tree of transaction class s. Our calculation for T_{O_s} reflects the time to perform communication along with the time taken at the slowest node (the node with the highest I/O and processor residence times for the operator). Determining this slowest node requires that we use information about the nodes at which an operator executes. The summation of the residence times in T_{O_s} uses the same technique as in the QNM estimations [LZGS84], but accounts for the parallel execution among the nodes. These calculations are followed by the workload's average response time (ART) calculation.

Before determining the costs of an operator, the operator's children are investigated. The estimation of the operators is therefore done in the same order they would execute: in a bottom-up fashion from the query plan tree. The costs of the leaves of the query plan are derived from the base table information.

Once the operator's stand-alone time (T_{O_s}) and its children's response times have been estimated, the operator response times are determined as the time to execute the critical path of the subplan. This estimation is done by using the relevant query plan information at each operator that includes the estimated placements from the children. If the children do not execute at the same nodes, we assume they execute in parallel, and hence we only need to calculate the time spent waiting for the children as the maximum response time between the children, i.e., $R^{O_s}_s = T_{O_s} + \max_i(R_{\text{child } i \text{ of } O_s})$. Otherwise, the time waiting for the children is the same as waiting for all the children at the nodes in which they both execute, i.e., $R^{O_s}_s = T_{O_s} + \sum R_{\text{children of } O_s}$.

If the execution of a query involves only a single node of the node set, we estimate the operator response time that would result at each of the nodes (e.g., when a host variable is used in the query class definition), and average the costs over all the nodes in the set. The resource usages are calculated as if the operation took the same time at each of the nodes. However, if the execution of an operator is across all the nodes where the operator can execute (i.e., the nodes receiving the temporary input relations used to execute the operation), the subplan execution cost is estimated as the maximum time across all these nodes.

In some circumstances, when communication is involved between operations, the costs depend on the operation (sender or receiver) with the longest response time. We make the assumption that, if the sender and receiver execute at different nodes, then the response time of the receiver is estimated as the maximum time to execute the receiver operator along with the receiving of data and the sender subplan time along with the sending time. However, execution at the sender and receiver nodes may overlap. Thus, both the send and receive costs would be incurred at the overlapping nodes, so a sum of the send and receive costs at these nodes is used.

Our cost estimation method as shown in Figure 4.13 differs from the approximate mean-value analysis (aMVA) method used in QNM literature [LZGS84]. In the aMVA method for an open QNM, the visit counts are calculated, the residence time for a transaction at each resource is the sum of all the operator service demands for the resource along with congestion delays incurred at the resource, and the response time of a transaction is a sum of all the residence times of all the resources. However, this does not allow for the concurrent execution between the resources, such as the independent execution between two different nodes. Since we are modeling a parallel shared-nothing system, we estimate a transaction's response time by constructing the execution cost from the plan as it would execute on such a system.

Validation Through Simulation

We now show the results of some simulation experiments, which are used to validate some of our analytic results. The simulator we developed is described in Appendix B. For the TPCD workload, we used a 100MB DB on 16 nodes.

We performed three experiments. In the first two experiments, we varied the DB sizes as a fraction of the full DB size, while in the third experiment, we varied COMMTIME. In the first DB size experiment, we determined the partitioning only, and the results can be seen in Figure 4.14. The second DB size experiment's results, in which only indexing and clustering are chosen, are shown in Figure 4.15. Our COMMTIME experiment results, in which only partitioning is chosen, are given in Figure 4.16. We note that the relative agreement between the analytic and simulated results is good. Simulations are labeled with SIM in the legends. The agreement carries through for different parameter values. Note that analytic estimates are higher than the simulated ones mainly because of the increased parallelism allowed between the resources during the simulation across the nodes and within the nodes for the different operations of the same query. The analytic formulas sum up the resource times for the processor and disks at each node.

4.1.9 Shortcuts to Reduce Algorithm Execution Times

This section describes an optional *shortcut* that we allow in the algorithm's search. We use the term shortcut because it reduces the number of search cases and thus the execution time but at some risk of weakening our algorithm's claim of non-separability. The shortcut may cause the algorithm to find a solution that results in a higher ART than would have resulted without the shortcut. Because of this, the decision to take the shortcut is left to the discretion of the DBA.

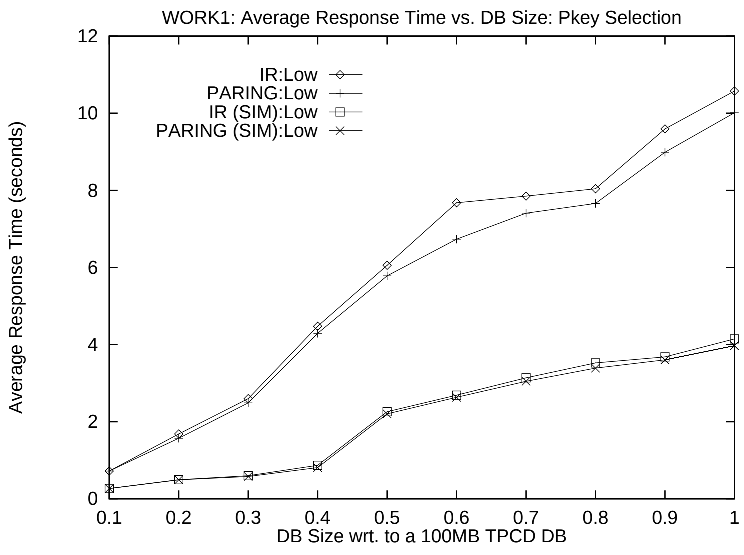


Figure 4.14: Analytic and simulated average response time versus the DB size when selecting partitioning keys for workload WORK1.

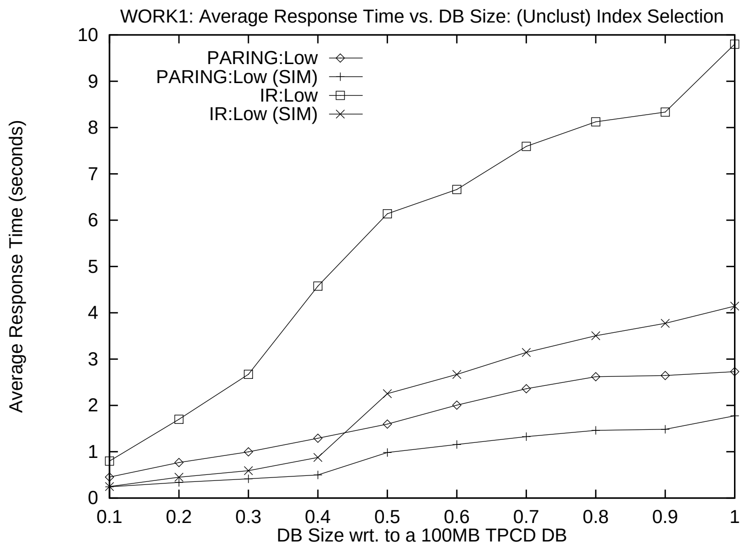


Figure 4.15: Analytic and simulated average response time versus the DB size when selecting indexing keys for workload WORK1.

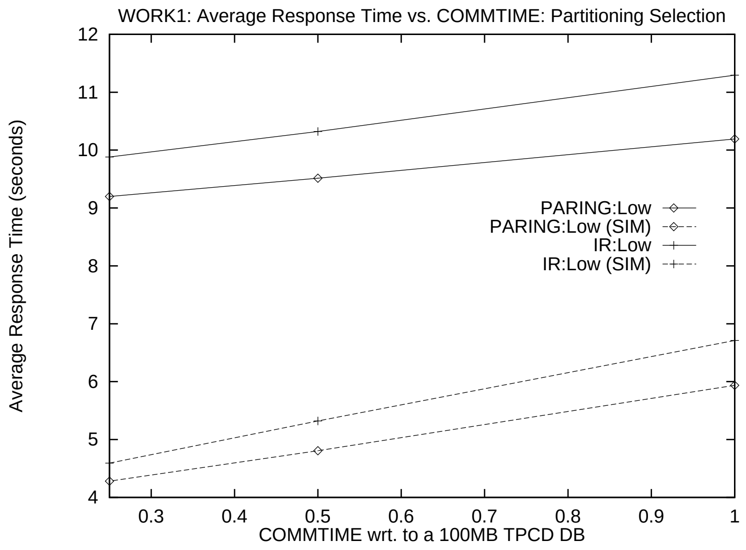


Figure 4.16: Analytic and simulated average response time versus the communication latency when selecting partitioning keys for workload WORK1.

Our shortcut involves an extra pre-scanning rule where we remove nonclustering indexing keys from the search. This rule is taken from Finkelstein et al.'s index selection algorithm [FST88], and affects the selection of indexing keys that do not cluster a relation but that are involved in a join when the indexing and clustering decisions are tied together. The rule requires that, if using a nonclustering indexing key results in a workload response time that is higher than the response time of selecting any other index, we can remove the corresponding nonclustering indexing key from the set of candidates.

For this rule, the indexing keys that can be eliminated are involved in joins, and the search using the elimination is affected by the non-separability of the joins. When we remove the non-clustering indexing key without determining the other relations joining with this key's relation, we may miss a possible candidate in the searching and branching. We shall present experiments in Chapter 5 that compare the quality of the solutions along with how long it takes to find them when using and not using the rule.

4.2 Other Algorithms

In addition to the PARING algorithm, we designed two other partitioning key and relation group selection algorithms. These algorithms are:

- the *Independent-Relations* (IR) algorithm; and
- the *Combined-Relations* (CR) algorithm.

The purpose of developing these algorithms was to:

- present algorithms that are related to ones in the literature;
- provide algorithms that can be compared with our PARING algorithm; and
- develop a good initial complete solution (determined by IR) for PARING.

We first develop the simple but fast IR algorithm in Section 4.2.1 that determines only partitioning decisions, and then proceed to describe the search-oriented CR algorithm in Section 4.2.2 to determine partitioning, indexing, and clustering decisions.

4.2.1 IR Algorithm

The IR algorithm is designed to be fast. It makes its partitioning key choices independently for each relation. Thus, we reduce the complexity from being proportional to the product of the number of attributes in each relation over all relations to being proportional to the sum of the number of attributes over all relations.

The structure of the algorithm was designed to be similar to algorithms presented previously in the literature [Ape88, CNP82, OV91, Jak80]. In these algorithms, the key of most importance in terms of query performance is chosen as the partitioning key for a relation. This method is also used in the DBA's rules of thumbs defined in Chapter 3. The pseudo code for the IR algorithm[1] is presented in Figure 4.17.

The main idea is to rank the keys according to their significance in the queries, as seen in line 2.1 of the algorithm. This ranking of a key is formulated from *weight* calculations for all the partitionable operators (shown in Table 4.2) in which the key is involved within the workload. These operator weights are the savings of executing the operator when the key is chosen as a partitioning key as opposed to it not being a partitioning key. If the execution cost of the operator is reduced by its key being chosen, then the weight will be a positive number; otherwise it is negative. For example, a negative weight results when the load imbalance of executing a query on a single partition, and thus on a single node, causes greater contention than executing it at all the nodes of a relation. Appendix C defines the operator weight calculations. Operator weights are used in an *aggregate weight* calculation to give the ranking of the key. The aggregate weight of attribute A_j of relation R_i is calculated as:

$$W_{i,j} = \sum_{k=1}^{|Q|} \lambda_k \sum_{h=1}^{O_k} W_{R_i,A_j,k,m_h}$$

where $|Q|$ is the total number of queries in the workload, O_k is the number of partitionable operators in transaction class k, λ_k is the arrival rate of transaction class k, m_h is the h-th operator in the transaction class k, and W_{R_i,A_j,k,m_h} is the the operator weight for attribute A_j

[1]This work was done jointly with Anant Jhingran and Sriram Padmanabhan, and its initial description can be found in Zilio et al. [ZJP94].

```
Independent Relations Algorithm
-------------------------------
Input: DB, system, and workload information.
Output: Complete partitioning key solution and its cost B

1. Pre-scan attributes to get the possible candidate keys (single
      and multi-attributes) for partitioning. This phase is the
      same as the pre-scan phase used in the PARING algorithm.

2. Partition Key Selection:
2.1.  Calculate each key's ''relative importance (or weight)''
      over all the queries in the workload to rank their
      usefulness in being used as the partitioning key.
2.2.  Choose the highest weighted key of each relation to be the
      partitioning key for the relation. Break ties among the
      keys based on higher cardinalities (if possible). Otherwise,
      break ties by choosing the key arbitrarily.

3. Group relations using the same method as in the PARING algorithm.
4. Select the data placement as in the PARING algorithm.
5. Obtain the cost using the chosen partitioning key set,
      relation grouping, and data placement, and assign it to B.
```

Figure 4.17: The Independent Relations Algorithm

($W_{R_i,A_j,k,m_h} = 0$ for the operators not using A_j). Thus, the ranking of a key is based on two components:

- the performance benefit of selecting the key for each operator in which the key is used; and
- the frequency (arrival rate) of the query to which each operator belongs.

A key with higher ranking will thus be one that, on its selection as the partitioning key, will result in greater performance benefits.

After the ranking, the highest weighted key for a relation is chosen as the partitioning key, as seen in line 2.2 of Figure 4.17. If there is more than one key in the relation with this weighting, then the key to be chosen should be the one with highest cardinality, which generally results in the least data skew. If there are still keys that tie based on the highest cardinality, we choose one arbitrarily.

IR involves other aspects besides the weighting. The pre-scanning, grouping, data placement, and cost estimation components are the same as the ones used for the PARING algorithm, and are described in Sections 4.1.7, 4.1.5, 4.1.6, and 4.1.8, respectively.

The IR algorithm is fast, and it works well in simple cases where the choice is straightforward. However, in more complex cases, the IR algorithm often fails to make the best choice. For

Operator Name
Join
Group By
Duplicate Removal
Order By (Sort)
Selection (constant)
Selection (host variable)

Table 4.2: Partitionable and indexable database operations

example, if a relation has two or more keys with the same or nearly the same weight, the IR algorithm could possibly make a bad choice. Another problem results when the ER-design and/or the queries are complex in terms of the interactions, through joins, between relations. In this case, there are many possible partitioning keys per relation. Because the IR algorithm does not use the optimizer, and thus does not consider the different possible query plans, the interaction of relations in the query plans is not considered, and the weighting calculations are only crude estimates for selectivities. There is also no feedback from the optimizer and cost estimator to the algorithm for it to refine it's decisions as in the PARING and CR algorithms.

An example of the consequences of not considering one simple parameter, selectivity, is as follows. Assume we have three relations R, S, and Q. Also, assume the sizes of the relations after single relation selection predicates are performed are 100,000 records for R and S and 10,000 records for Q. Assume the workload is given in Figure 4.18, where edges represent joins. In the workload, there are two joins $R.A = S.A$ and $Q.B = R.B$ of equal frequency. The two possible choices are:

- group R and S together and partition them on attributes A; or
- group R and Q together and partition them on attributes B.

The first choice would cause the join between R and S to be localized, whereas the second choice causes the join between R and Q to be localized. The difference between the two plans is in the amount of communication involved for the whole workload. The first plan requires lower communication costs because the joins between relations R, S, T, and Q can all be localized, so it should be chosen. This example illustrates the benefit of using the optimizer with a good cost model to better predict the consequences of various choices of partitioning keys.

Consider Figure 4.18 again and assume that $R.B$ has a slightly higher weight than $R.A$ and that S, T, and Q are partitioned on A. In that case, IR will choose $R.B$ as the partitioning key, resulting in two relation groups – one containing S, T, and Q, and the other containing R. Thus, joins on $R.A$ and $R.B$ are not localized. However, we could have chosen $R.A$ as the

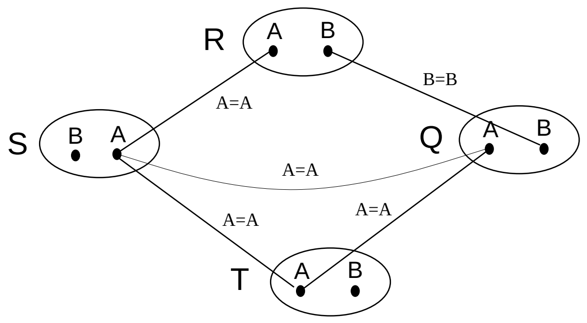


Figure 4.18: Example of a workload on a four relation DB (R, S, Q, T).

partitioning key, and obtained one relation group where only the join on $R.B$ is not localized. In contrast to IR, the CR and PARING algorithms will test different groupings for different combinations of partitioning keys and choose the grouping and partitioning key set yielding the best performance.

4.2.2 CR Algorithm

For our *combined relations* (CR) algorithm, the search for the indexing keys, clustering keys, partitioning keys, and relation groups is driven by the keys only rather than the keys and the queries as in PARING. The workload ART resulting from each candidate physical DB design is used for comparing designs as in the PARING algorithm. This CR algorithm is adapted from an index selection algorithm by Finkelstein et al. [FST88], which constructs a list of keys that are good candidates for indexing, and selects subsets of keys from this *candidate key* list. CR differs from Finkelstein et al.'s algorithm because it includes:

- more and different pruning and pre-scanning rules (thus reducing the number of iterations compared to Finkelstein et al.'s algorithm);
- the partitioning decisions; and
- a workload response time estimator that includes resource contention.

The pseudo code for the CR algorithm is given in Figure 4.19. The main idea of CR is to use the attribute aggregate weighting calculations to rank the keys. Weighting was not used in Finkelstein's algorithm, so the number of possible keys considered by that algorithm is greater than CR's [FST88]. We use the same weighting as in the IR algorithm, which used it for partitioning key selection, but supplement it with weighting for indexable keys as described in Appendix C. This weighting is also useful for indexing and clustering key selection because it gives a relative difference between using versus not using a key for execution of an operator.

CR overcomes some deficiencies of IR by allowing more possibilities to be evaluated. Our CR algorithm goes beyond just an aggregate weighting by:

```
algorithm CR Algorithm

input: Query set $Q$, DB information, system information,
        $\alpha$ pruning factor ($0 \leq \alpha \leq 1$)
output: Complete solution $S_B$ having best cost: cost($Q, S_B$)
objective: Find a solution that minimizes average workload response time
definitions: $L$ is the set of keys (each composed of one of more DB attributes)
        used as candidates in the search, and is the set considered after
        filtering using $\alpha$. $V$ is the set of decision variables
        such that $V = \{p_j, c_j, I_j$ for each relation $j = 1, \ldots, R\}$, where
        $p_j$ is the partitioning key, $c_j$ is the clustering key, and $I_j$ is the indexing keys decisions for $R_j$.
        $S = \{S_1, S_2, \ldots, S_{|S|}\}$ is the set of solutions to still search.
        $S_i \subseteq V \times L$ with no decisions from $V$ are duplicated in $S_i$, and
        $S_i$ is a complete solution (with default keys added)
        as well as a partial one when keys can still be bound to decisions in $S_i$.
        cost is the same as for PARING. $S_{null}$ is the empty solution.
1.1. initialize: Pre − scan: Form the candidate key list $T$ (for all queries together)
        and remove unwanted keys from the search (as in PARING's pre-scanning).
1.2. $T \leftarrow T \bigcup FormMultiAttributeKeys(q_i, L)$ for each $q_i \in Q$
1.3. Calculate each key's relative importance (weight) over all the
        queries in the workload in terms of parameters bound in each
        execution of the algorithm (as in the IR algorithm).
1.4. Remove from $T$ all keys of a relation not within $1 - \alpha$ from
        the highest weighted key for the relation to give
        survivor list $L$
1.5. Get the IR algorithm's solution $S_B$ (unless a previous $S_B$ exists)
{ Root of search is empty Pkey list }
1.6. $S \leftarrow \{S_{null}\}$
{ Start CR search using survivor list $L$ and search set $S$ }
2. while $S$ is not empty do
3.      Remove an element $S_i$ from $S$: $S \leftarrow S - S_i$
        { Only add to $S$ when there are keys to bind to decisions in $S_i$ }
4.      for each eligible key $P'$ in $L$ that can be bound to some
            added decisions in $S_i$ do
5.          $S' \leftarrow S_i$ with added decisions bound by $P'$
6.          $S \leftarrow S \bigcup S'$
            { Complete solution ($S''$) using $S'$ and
                assigning default keys for some relations }
7.          $S'' \leftarrow CompleteSoln(S')$, $C \leftarrow$ cost($Q, S''$)
8.          if $C <$ cost($Q, S_B$) then
9.              $S_B \leftarrow S''$
10.         end if
11.     end for
12. end while
13. ConsiderCompletion (potentially go to Step 1.2)
14. EXIT
```

Figure 4.19: The CR algorithm

- searching through all key sets that have a weight within a specified factor of the highest weight;
- determining the relation groupings and data placement for each choice of partitioning keys;
- using the query optimizer to generate the plans and estimated query costs based on the given indexing keys, clustering keys, partitioning keys, relation groupings, and data placement; and
- choosing the particular partitioning, indexing, and clustering key set that minimizes the average response time of the workload.

Before the search is performed, the candidate keys are determined and placed into a list (L) called the *survivor list*[2]. The purpose of the survivor list is to reduce the set of possible candidate keys so that the number of partitioning, indexing, and clustering key sets considered in the search is also reduced. There are two components used in determining the survivor list:

- pre-scanning; and
- aggregate attribute weighting.

The pre-scanning is the same pre-scanning done for the PARING algorithm, as described in Section 4.1.7, and the weighting is the same as defined for the IR algorithm in Section 4.2.1. This pre-scanning includes the choice for the best possible keys not involved in any joins as the default partitioning, indexing, and clustering keys for a relation. CR starts with each relation having its default partitioning, indexing, and clustering key. The algorithm then considers improving the selected set of partitioning, indexing, and clustering keys by using join keys instead for some or all relations. Because default keys could be chosen for each solution, each solution can be considered a complete solution. A solution that can have join keys added to it as partitioning, indexing, and clustering keys is also considered a partial solution.

As in the PARING algorithm, CR uses the same modules *FormMultiAttributeKeys* and *ConsiderCompletion* to formulate the multi-attribute keys. However, for our implementation of CR used in the experiments for Chapter 5, we use the one pass pre-scanning method.

Unfortunately, the set of possible candidate keys after the pre-scanning would still result in many possible partitioning key sets. For this reason, we take the output from the pre-scanning and remove any keys with weight not within a factor $(1-\alpha)$ of the highest weighted key for each relation where $0 \leq \alpha \leq 1$. This α factor allows for some flexibility in how large the survivor list is and thus how many partitioning key sets are considered. The two extremes for α are:

[2]The term *survivor list* was taken from Finkelstein et al.'s algorithm [FST88].

- $\alpha = 0$, which causes the highest weighted keys to be the only keys on the survivor list, so the search will only break ties among the highest weighted keys; and
- $\alpha = 1$, which causes all keys with non-negative weights to be considered.

Since aggregate weight calculations are a rough estimate of the importance of a key, and since the calculations do not involve the optimized query plans and the selectivities on the relation in the plans, the weight calculations may not fully reflect the resulting performance for selecting a key as the partitioning, indexing, and clustering key. We therefore allow the use of the α factor to include keys close to the highest weighted key. These keys provide more eligible candidates when assessing query plans and costs directly. Also, we allow a search to break the ties of the highest weighted keys so that an arbitrary choice no longer exists, as in the IR algorithm.

Once the survivor list is created, a tree expansion is then performed to enumerate the *eligible subsets*, or *solutions*, of partitioning, indexing, and clustering keys from this survivor list. The set of possible solutions to evaluate and expand at any time in the algorithm are placed in a set S. A solution would have the following properties:

- each relation has at most one partitioning key;
- each relation has at most one clustering key; and
- default keys are used for the partitioning, clustering, or indexing keys of a relation when the corresponding decision is not bound by the solution.

The tree of possible search cases is built as follows. We first create the root of the search to be the case of not selecting any key, i.e., we initialize the set S to contain only the empty set. The search then removes a set from S until it is empty, i.e., no solutions remain. For each solution (S_i) removed from S, all the single key additions (P') are made to this set. Each addition will cause a new solution to be inserted into the set S unless no additions to the solution S_i can be made. The eligible keys that can be added to a solution (S_i) are the keys in the survivor list (L) that are not already used in the solution (S_i).

A vertex in the resulting search tree will either be the empty set (the root of the tree) or a bound candidate key. Solutions can be read from the tree by starting at the root and following a path from the root to any level of the tree. The vertices traversed along the path from the root to any node will constitute an eligible solution.

When we create a new solution (S'), the solution is transformed into a complete solution (S'') using the module *CompleteSoln* in which the default partitioning, indexing, and clustering keys are added (as in PARING). The cost of the workload using this key set is compared to the cost of the best known solution ($\texttt{cost}(Q, S_B)$). A new lower cost (C) will cause the solution

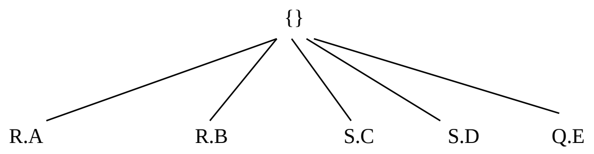


Figure 4.20: Example of the search tree expanded to one level from the root, where only partitioning is to be chosen and all keys shown are assigned to the appropriate relation's partitioning key decision variable.

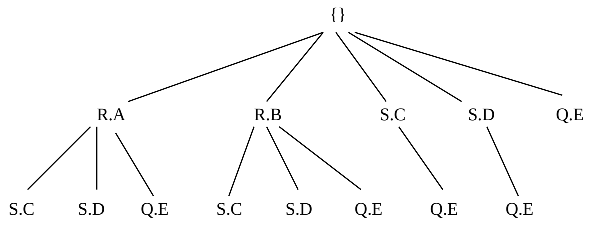


Figure 4.21: Example of the search tree expanded to two levels from the root, where only partitioning is to be chosen and all keys shown are assigned to the appropriate relation's partitioning key decision variable.

(S'') to become the new best known solution. In order to make these comparisons, all the queries in the workload must be optimized and their costs estimated. This will cause the total number of calls to the optimizer by the CR algorithm to be roughly equal to the number of solutions evaluated times the number of queries.

For example, if we had the following survivor list: $\{R.A, R.B, S.C, S.D, Q.E\}$, the first expansion from the root when only partitioning keys are selected would be as in Figure 4.20. Expanding this tree down one level gives the tree in Figure 4.21. Note that $R.B$ is not considered in the subtrees under the node for $R.A$, and no duplicate subsets are considered with this procedure. The final tree would be as in Figure 4.22. Some eligible subsets from this tree are $\{\}$, $\{p_R = R.A\}$, $\{p_R = R.B, p_S = S.C\}$.

The CR algorithm uses some of the same components as in the PARING algorithm. These components include:

- the method to group relations;
- the data placement technique; and
- the optimizer and workload cost estimator.

We also added the pruning of solutions with unused keys, as in the PARING algorithm. This rule causes our CR algorithm to differ even more from Finkelstein et al.'s [FST88] since they did not consider this extra pruning.

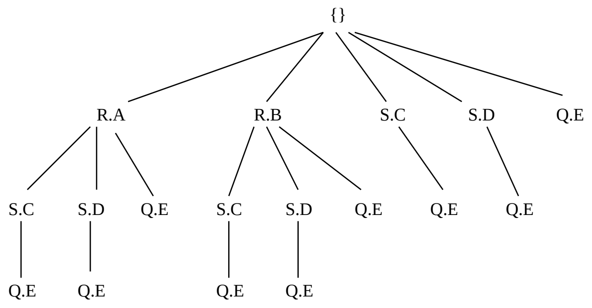


Figure 4.22: Example of the full search tree expanded from the root, where only partitioning is to be chosen and all keys shown are assigned to the appropriate relation's partitioning key decision variable.

Compared to the PARING algorithm, there is an advantage and some disadvantages to using the CR algorithm. The advantage for CR results when there are fewer candidate key possibilities than there are queries. Since PARING uses the queries to drive the search, the CR algorithm should result in fewer search cases. However, there are a few disadvantages with the CR algorithm. First, even for low values of α, the algorithm may still be expensive for big workloads and ER-designs. Second, CR's resulting solution is not guaranteed to have an ART that is as low as the solution's ART from PARING. Pruning with α may actually discard this good PARING solution, so α trades-off the speed of executing the algorithm with the resulting quality of solution. Even when $\alpha = 1$, CR may not return as good a solution as PARING, e.g., if an attribute is given positive weight w for one query and negative weight $-w$ for another query, the aggregate for that attribute could be zero, but choosing it may still be required as it assists the first query significantly.

Chapter 5

Physical DB Design: Experiments

In this chapter, we carry out experiments to accomplish two goals. First, we demonstrate how well the algorithms we presented in Chapter 4 (i.e., IR, PARING, and CR) perform. The algorithms are compared analytically. A set of figures are given to show the following performance measures: workload average response times, resource (or residence) times (on average), and the number of optimizer calls made by the algorithms. The number of optimizer calls is used as a way to determine the *cost* of executing an algorithm, since we found that, when an algorithm was implemented as an application tool for our real DB system (IBM DB2 PE) and using the DB system's optimizer, most of the algorithm execution time is spent in optimizing a query. A good algorithm is one with a relatively low number of optimizer calls that produces a physical DB design with a low average response time. Note that the naive IR algorithm requires only one optimizer call for each query in the workload and CR requires optimizing all queries for each solution it finds.

A performance measure that relates to workload average response times is *response time percent reduction* of an algorithm (A) over algorithm IR. This measure is calculated by comparing the average response time of each query class i ($ART_{A,i}$) determined by its output designs in a given environment with the response time ($ART_{IR,i}$) determined by the design chosen by the IR algorithm. The following calculation provides the response time percent reduction between an algorithm (A) and IR weighted over all classes in the workload:

$$\frac{\sum_{i=1}^{Q} \lambda_i \frac{ART_{IR,i} - ART_{A,i}}{ART_{IR,i}}}{\sum_{i=1}^{Q} \lambda_i}$$

where λ_i is the arrival rate of transaction class i and Q is the number of transaction classes in the workload. An algorithm with the highest response time percent reduction over IR is best.

Second, we show how well the best algorithm performs under various scenarios, such as varying the instruction path lengths and communication latency. In these experiments, we are interested in determining how the algorithm goes about choosing a good physical DB design, and how its cost (number of optimizer calls) differs under the various scenarios.

In these experiments, the algorithms are implemented as applications on top of IBM DB2 PE (parallel SN) system. The algorithms use the DB2 PE optimizer to optimize the queries for each of the candidate designs considered. We modeled a 16 node SN system. The parameters used to define the IBM DB2 PE system for our analysis experiments are given in Tables 2.1 and 2.2. Since DB2 only allows clustering a relation if a clustering indexing key exists on it, we restricted the clustering and indexing decisions so that a clustering key cannot be selected without an index on the key. This restriction causes the number of search cases considered by an algorithm to be reduced.

We used two types of workloads: a simple workload (defined in Appendix D and based on the TPC-D benchmark queries) and complex workloads (defined in Appendix E). Each complex workload was created to have similar structures to the TPC-D DB structure with more joins between relations. Since there are a limited number of possibilities for good keys in the TPC-D benchmark (usually the first attribute of each relation), we created the other workloads to have many more keys for our algorithms to consider. In our experiments, the workload based on TPC-D is called WORK1, and the complex workloads are named WORK2, WORK3, and WORK4.

Throughout this chapter, we make use of some terms relating to the saturation of the system by a workload. When using a DB design, a workload may saturate the system, and we say that the design *fails to support* a particular workload. However, if the workload does not saturate the system (i.e., transactions of each class can be completed at the same rate at which they arrive), we say the design *supports* the workload.

For most of the experiments, the initial arrival rates of the queries in the workload are all equal (except for WORK4 in which the second query's arrival rate is twice the rate of the other queries). We set the arrival rates to low, medium, and high rates such that, for a given workload using the design from the IR algorithm, the utilization of the bottleneck resource of the system is equal to 40%, 60%, and 80%, respectively. In the legends of the graphs for the experiments, we label these rates as *Low*, *Med*, and *High* arrival rates, respectively. These different arrival rates are used to show the impact of the system's load on the performance of the physical DB design created by the algorithms. Because the bottleneck is not utilized beyond its capacity to service requests, the design from IR supports the workload. Since PARING and CR either find the same design as IR or a better design in terms of the resulting ART, these algorithms will also result in designs that support the workload.

We now describe the results in the sections that follow. In Section 5.1, the IR, CR, and PARING algorithms are compared analytically. The best resulting algorithm (which turns out to be PARING) is then investigated further in Section 5.2 to demonstrate how it performs under various scenarios. This section also contains an experiment to demonstrate that our analytic cost model ART results are relatively similar to times measured on an IBM DB2 PE system.

A summary of the results and claims is given at the end of each section.

5.1 Comparing the Physical DB Design Algorithms

The algorithms IR, CR, and PARING are compared using several different types of experiments. Our goals in performing these experimental studies are as follows:

1. Determine how well the combinatorial algorithms (PARING and CR) perform compared to the naive IR algorithm.

2. Compare the performance of PARING and CR to determine whether there is a difference in the output designs and the corresponding response time of the workloads, as well as the cost of executing the algorithms (which in our implementation is measured by the number of optimizer calls).

3. Compare the algorithms to determine their performance under cases where various input parameters are varied. We are really interested in how the algorithms compare under system environment input parameters for fast and slow systems.

4. Demonstrate how the algorithms compare when selecting either partitioning keys or indexing keys. There are two types of experiments performed: (1) ones selecting only the partitioning keys; and (2) ones selecting only the clustering and indexing keys. Selecting partitioning, clustering, and indexing keys together is studied later in Section 5.2.

In performing our experiments, we wanted to determine performance results over a range of workload intensities. Hence our graphs of performance measures will typically have (relative) arrival rates indicated in the legends of the graphs.

To achieve these goals, we have performed the following experiments:

1. We vary the DB sizes to demonstrate how the algorithms are affected by the size of the data used by the workloads. Since we have added pre-scanning rules to ignore attributes and relations of low cardinalities, we were interested in determining how the pre-scanning affects the number of cases considered in PARING and CR. Since PARING and CR both use IR, we wanted to also determine whether PARING and CR find good designs even when a design from IR (used to initiate PARING and CR) fails to support the workload. We discuss these experiments in Section 5.1.1.

2. Since PARING and CR were defined using tolerance parameters r and α, respectively, we study how varying these parameters affects the algorithms' performance, and how they compare to each other. We wanted to determine good values for the tolerance parameters for each algorithm, and study how these values affect the cost of executing the algorithms

(measured in our work by the number of optimizer calls) and the performance of the workload using the designs they yield. These results are shown in Section 5.1.2.

3. Some of our experiments determine how the algorithms compare for systems that differ in the speed of the processors or communication system. One goal was to determine if moving between systems such that a new system becomes communication- or CPU-bound will affect how the algorithms perform with respect to each other. We show these results in Section 5.1.3.

4. We provide experiments that vary the arrival rates directly to allow us to determine what happens to the algorithms' performance for arrival rates that cause the workload to incur light to heavy system loads. Section 5.1.4 gives a discussion of these results.

We summarize the results in Section 5.1.5.

5.1.1 Varying DB Size

The purpose of these experiments is to determine how DB size affects the performance of the algorithms. We varied the DB sizes given in Appendices D and E by multiplying the relation cardinalities and column cardinalities by a factor. This factor was varied from 0.1 to 1 to provide DB sizes that were one tenth to the full size of the DB sizes as defined in the appendices. For the TPC-D database, we use DB sizes between 400 MB and 4 GB.

We used zero as the ratio (r) for PARING, and we used 0.1 as the factor (α) for CR. Setting r to zero means that PARING must find its best possible solution, and assigning α to 0.1 implies that CR will use the keys in a relation that are in the top 10% of the weightings from the highest weighted attribute for that relation. The study of appropriate ratios is given later in Section 5.1.2.

In this section, we describe experiments that vary the DB sizes and arrival rates so as to focus on the following:

1. Determine how the ART results from PARING and CR differ from ART results from IR. This includes investigating the effects on ART as DB sizes and arrival rates increase to their highest values or lowest values in the experiments.

2. Demonstrate the improvement PARING and CR provide in their resulting resource times compared to IR.

3. Show that PARING yields better ART results than CR. Also, show how the number of optimizer calls differs between CR and PARING. One goal is to determine how the difference between the number of optimizer calls from PARING and CR changes depending on

increasing the arrival rates. We also determine the effects of saturation on the resulting number of optimizer calls.

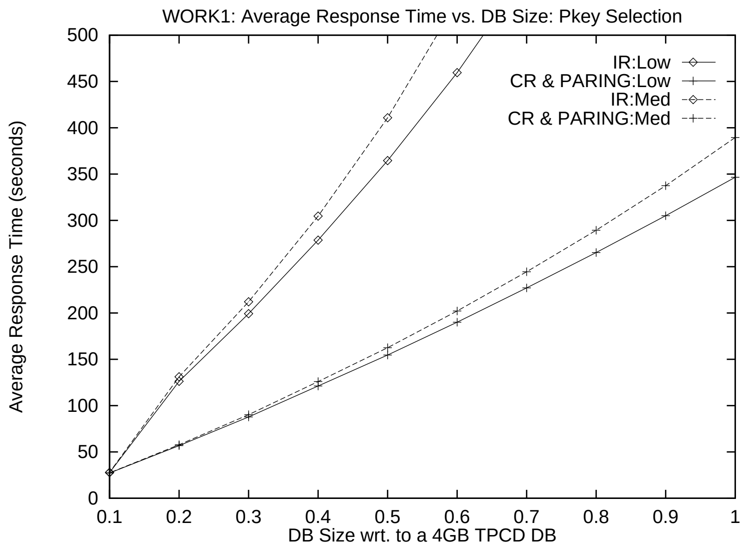
For the different DB sizes, the search algorithms (PARING and CR) find designs that allow lower workload average response times compared to the times produced using the designs from IR. This is true for the three different arrival rates (*Low*, *Med*, and *High*). Figure 5.1 for workload WORK1 provides an example when only partitioning keys are to be chosen. We only show the *Low* and *Med* rates so as to not clutter the figure. When indexing and clustering keys are chosen, the same results apply, as shown in Figure 5.2.

As the arrival rates increase for a given DB size, we notice that the differences between PARING's and CR's resulting response times and IR's resulting response times increased. For workload WORK1, these results are shown in Figure 5.3. When using the complex workload WORK4, similar results (in Figure 5.4) were obtained for the increase in the response time percent reduction. However, in Figure 5.4, the difference between CR and IR is zero since they return the same physical DB design. These increased ART differences between PARING and CR relative to IR occur when the DB size increases as well as when the arrival rates increase.

For smaller DB sizes, the percent reductions of the response times relative to the times from IR are low. The main reason for this is due to the pre-scanning phase. There are more low cardinality attributes, keys, and relations that are eliminated in pre-scanning and thus ignored in the search. This causes the number of keys considered by all the algorithms to be low, which causes the algorithms to have an increased chance of finding similar designs leading to similar response times. This reduction in the number of keys reduces the cost of the algorithms (which in our case is the number of optimizer calls) as well. For example, in Figure 5.1 for workload WORK1, the reduction is evident at DB size factor of 0.1. Note that, in this figure and all others plotting the number of optimizer calls, the calls for the IR algorithm are not shown because the number of calls for IR is always equal to the number of queries.

Also, for some higher arrival rates, the algorithms could not identify a design that avoided saturation of the system for large DB sizes. The bottleneck utilizations for all designs at these arrival rates were always equal to one. However, there were cases when the design found by IR fails to support the workload while the designs from PARING and CR do support the workload. Two such cases are shown in Table 5.1, where the TPC-D workload, WORK1, is used with different DB sizes to select partitioning keys.

The residence times for all the resources (communication, processor, and disk times) are improved for the designs found by PARING and CR over IR. For example, Figures 5.5, 5.6, and 5.7 show the residence times for workload WORK1 when partitioning keys are chosen. These residence times are calculated in our experiments as the average times the queries in the workload spend at a resource.



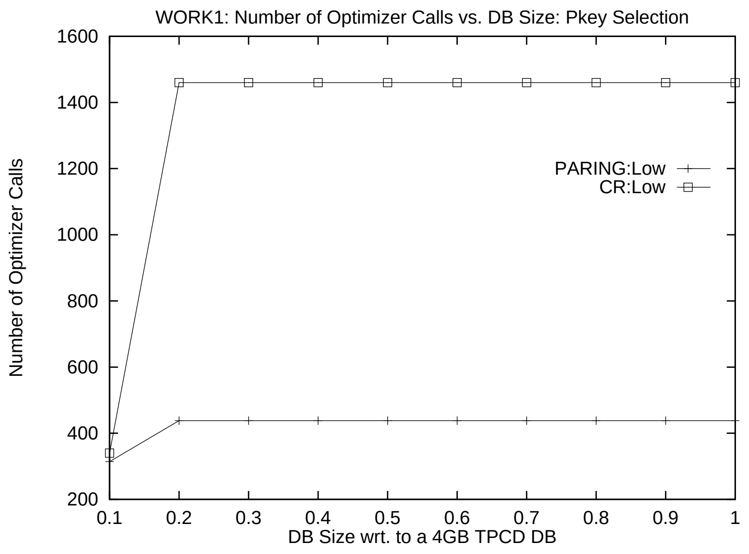
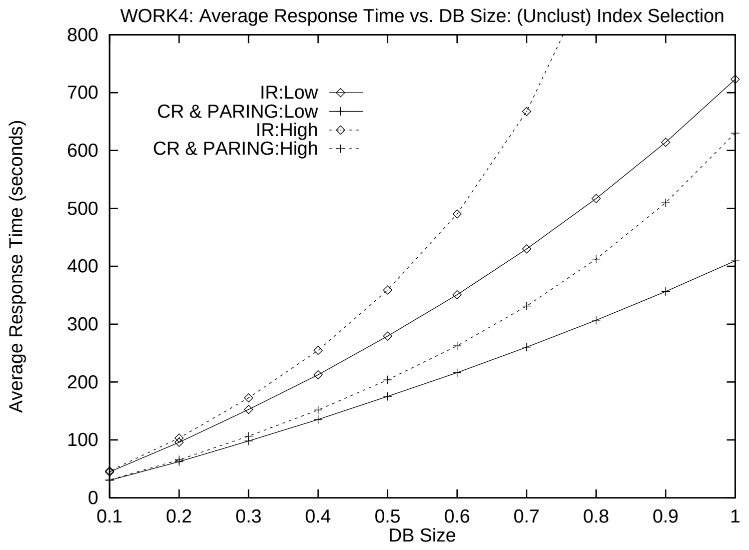


Figure 5.1: Average response time and number of optimizer calls for each algorithm versus the DB size factor when selecting partitioning keys for workload WORK1.

TPCD DB Size	**Arrival Rate for Each Query (queries/hour)**	**PARING's and CR's ART (seconds)**
1 GB	2.772	126.7
4 GB	0.684	525.7

Table 5.1: Average response times for PARING and CR that were found when the design from IR fails to support the workload for partitioning key selection and workload WORK1.



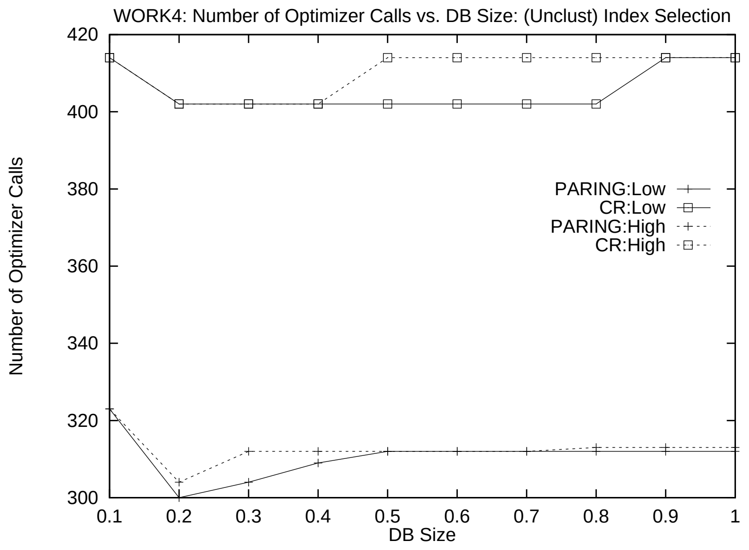


Figure 5.2: Average response time and number of optimizer calls for each algorithm versus the DB size factor when selecting indexing keys for workload WORK4.

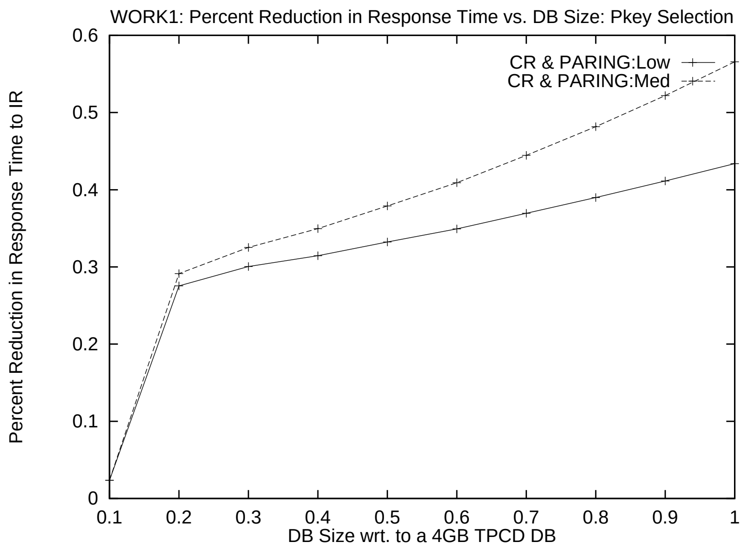


Figure 5.3: Response time percent reduction of our algorithms over IR versus the DB size factor when selecting partitioning keys for workload WORK1.

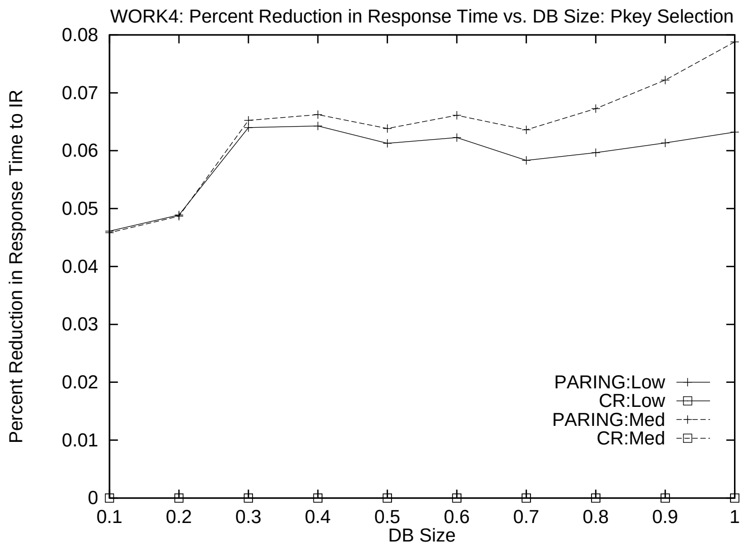


Figure 5.4: Response time percent reduction of our algorithms over IR versus the DB size factor when selecting partitioning keys for workload WORK4.

As can be seen in the previous ART figures, there are some anomalies that exist. For example, for workload WORK4 and DB size factor 0.3 in Figure 5.4, the response time percent reduction increases to a percent reduction that is higher than the 0.4 factor. In these cases, between different DB sizes, the chosen designs were the same but the query plans differed. Different query plans were chosen by the optimizer because of the different DB sizes, and thus, the percent reduction is not monotone with the DB size.

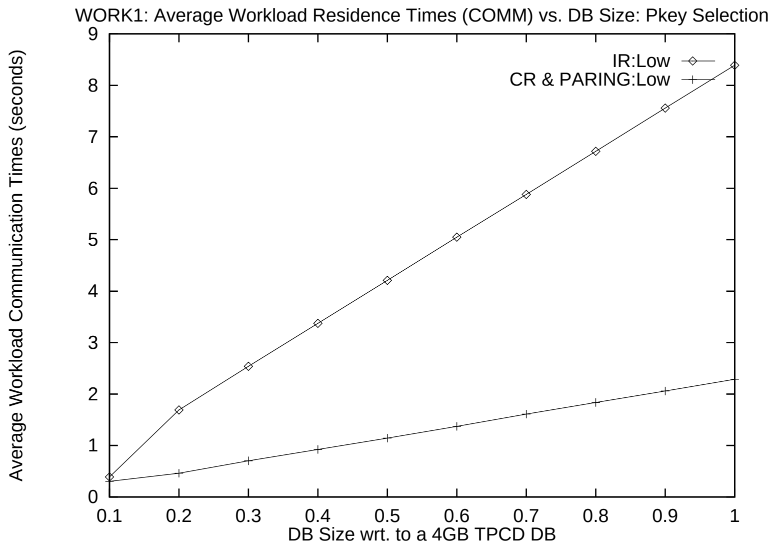


Figure 5.5: Maximum of the average workload interconnect network communication time versus the DB size factor when selecting partitioning keys for workload WORK1.

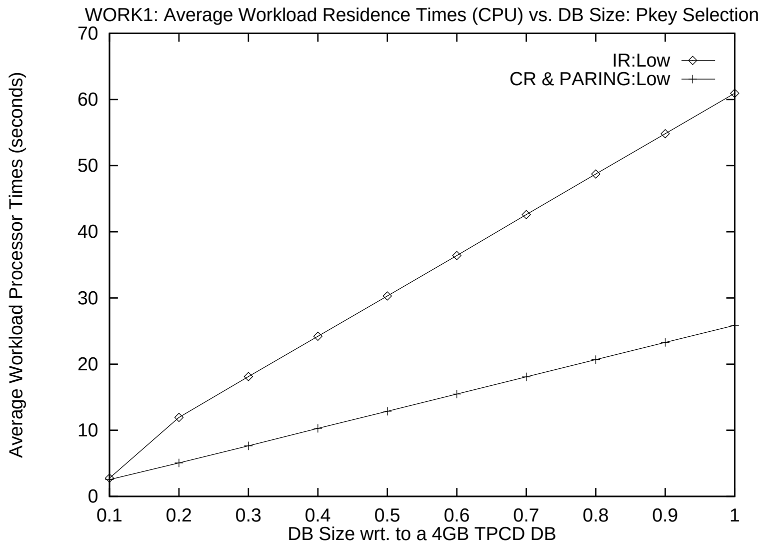


Figure 5.6: Maximum of the average workload processor time versus the DB size factor when selecting partitioning keys for workload WORK1.

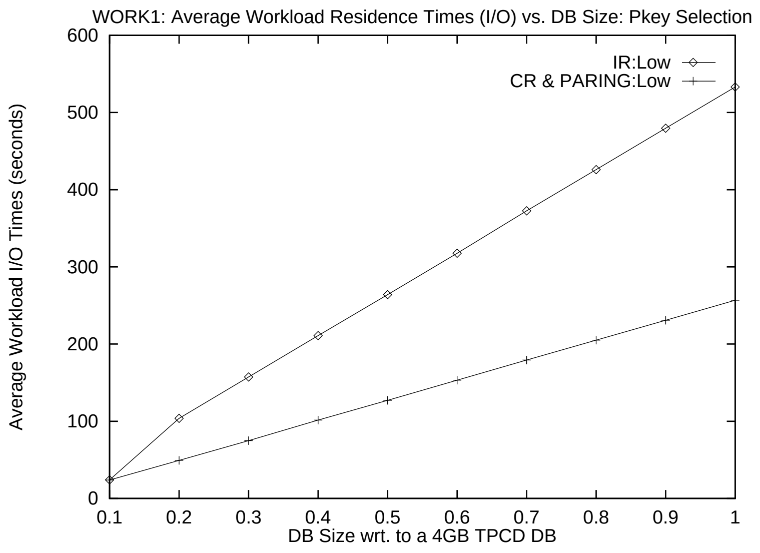
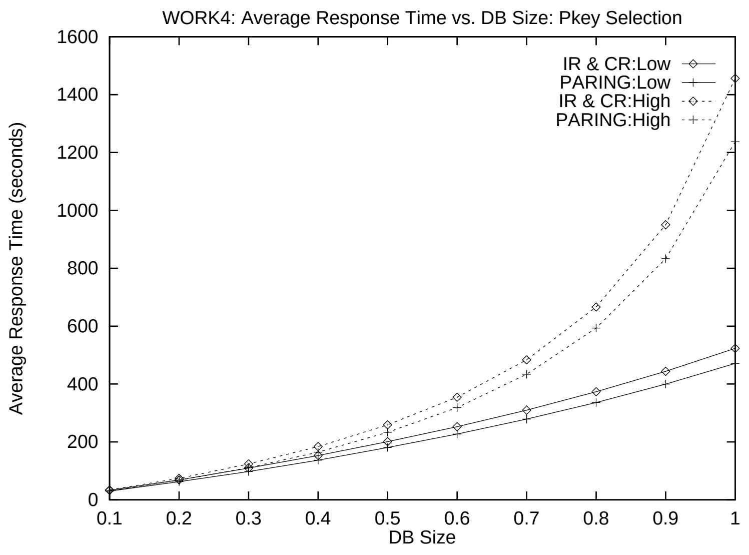


Figure 5.7: Maximum of the average workload I/O time versus the DB size factor when selecting partitioning keys for workload WORK1.

Along with PARING's improvement over IR, PARING also results in equal or better average workload response times than CR. For example, PARING's ART results are lower as shown in Figure 5.8 for workload WORK4.

However, the number of optimizer calls made by a PARING execution compared to the number of calls required by CR can be either higher or lower depending on the circumstances. When the workload is more complex, the number of calls is increased for PARING and sometimes exceeds the calls for CR. From Figure 5.8 for workload WORK4, the number of optimizer calls for PARING is higher than the number of calls for CR. However, for index selection, we have seen that the improved pruning and pre-scanning causes the number of calls for PARING to be lower than CR (as seen in Figure 5.2 for workload WORK4) while the resulting response times are still similar (as seen in Figure 5.2). Note that the comparisons of the number of optimizer calls between CR and PARING are sensitive to the values chosen for α and r. However, we have chosen a value for r that results in a pessimistic number of calls for PARING and a value for α that results in a favorable number of optimizer calls for CR. Thus, if PARING results in a lower number of calls than CR, we can deduce that this will apply to other values of r and α.

Also, varying the arrival rates affects the number of calls. For example, the number of optimizer calls increases when the arrival rates increase (from *Low* to *High* rates), e.g., for WORK4 in Figure 5.8. This phenomenon is a result of the difference between PART and ART values being increased for higher arrival rates, and this difference is directly related to the formula used for PART. All the arrival rates are used in the denominator and in multiplying



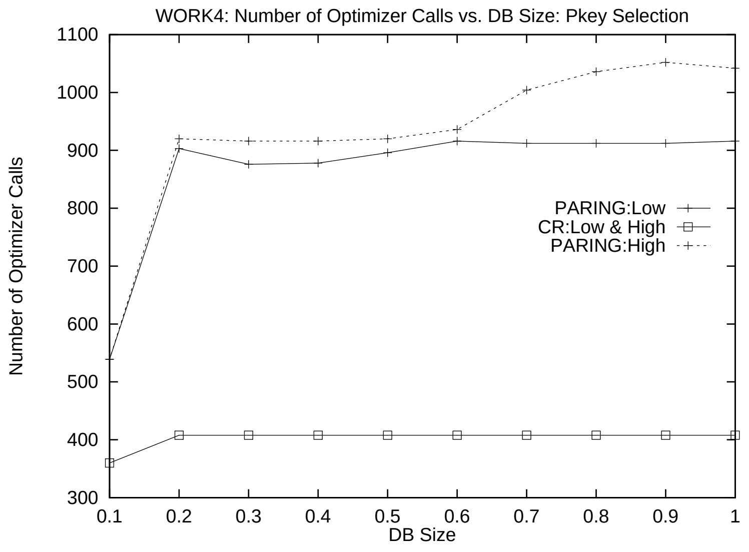


Figure 5.8: Average response time and number of optimizer calls for each algorithm versus the DB size factor when selecting partitioning keys for workload WORK4.

each query class's average response time in the numerator. When arrival rates increase, for a subset of query classes, the partial summation of the subset of queries in PART is further from a full ART for all the query classes using lower arrival rates. If we prune using the ratio r to get an ART that is at most $r\%$ above the best ART (found using $r = 0$), more designs will have PART values for a subset of queries that are beneath the best known upper bound and that cannot be pruned using the ratio and bound. This reduction in pruning causes the number of solutions examined to increase for higher arrival rates, which causes the number of optimizer calls to increase.

The number of optimizer calls for PARING and CR becomes smaller for high arrival rates. For example, when analyzing workload WORK1 as shown in Figure 5.9, the number of calls is lower for both PARING and CR using *High* rates as opposed to *Low* rates. In this case, the number of optimizer calls for CR is reduced along with the number for PARING because the pre-scanning phase finds more designs that fail to support the workload. Hence, PARING avoids examining more of these cases and the cases they generate later in the search.

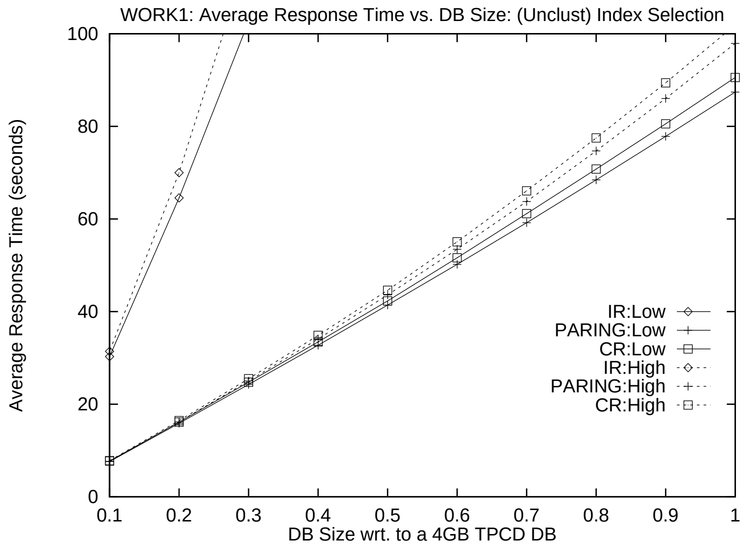
Saturation also affects the number of calls for different DB sizes. For larger DB sizes, the number of calls is smaller for PARING due to PARING finding many more designs that fail to support the workload. When a design fails to support the workload, it and the designs it generates are pruned by PARING. For example, Figure 5.10 demonstrates that, for DB size factor 1.0, the number of calls is smaller than for factor 0.9 in performing index selection for WORK3.

5.1.2 Comparing Ratios r for PARING and α for CR

In this section, we compare how CR and PARING perform when the ratios required for each algorithm (α and r, respectively) are varied. We are interested in evaluating the trade-off between how the resulting ART from each algorithm differs from the best found ART (determined by PARING with $r = 0$) and how the number of optimizer calls changes. To demonstrate the effects of the ratio (r) used in PARING, we ran some experiments that varied r from 0 to 0.45 in increments of 0.05. To show the results of changing the α factor, we executed some experiments that varied α from 0 to 0.9. Remember that, for higher values of r, we are moving further away from the best design found by PARING, but, for higher values of α, we consider more designs so that we are more likely to get answers closer to this best design.

In varying r for PARING, the ART result is at most $r\%$ higher than the best possible ART, which is found when $r = 0$. For example, we see this result in Figure 5.11 for WORK3 when choosing partitioning keys only. As expected, the increase in r causes a decrease in the number of optimizer calls, e.g., for WORK3 as shown in Figure 5.11.

For CR, when α increases, the ART moves closer to the best design, but we can provide no guarantee of how quickly or how close this will approach this best design for a given value of α.



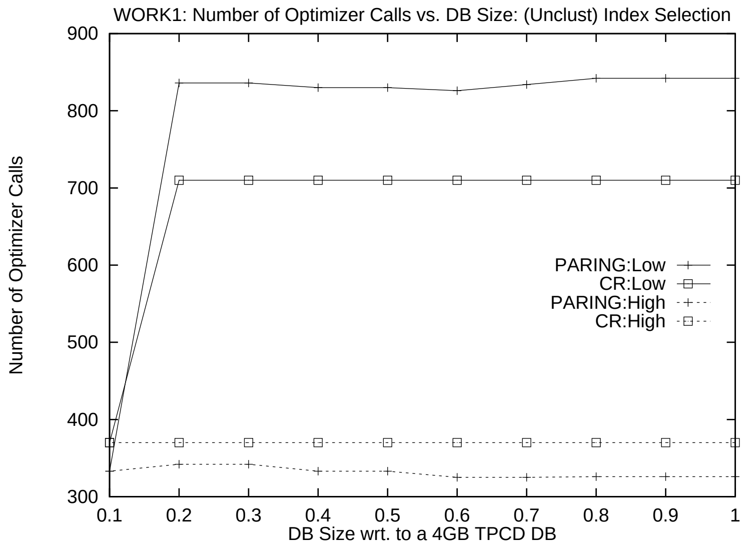
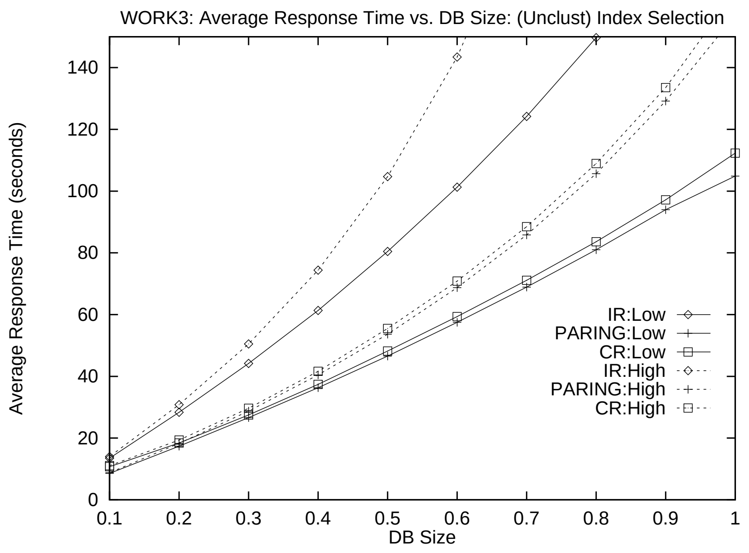


Figure 5.9: Average response time and number of optimizer calls for each algorithm versus the DB size factor when selecting indexing keys for workload WORK1.



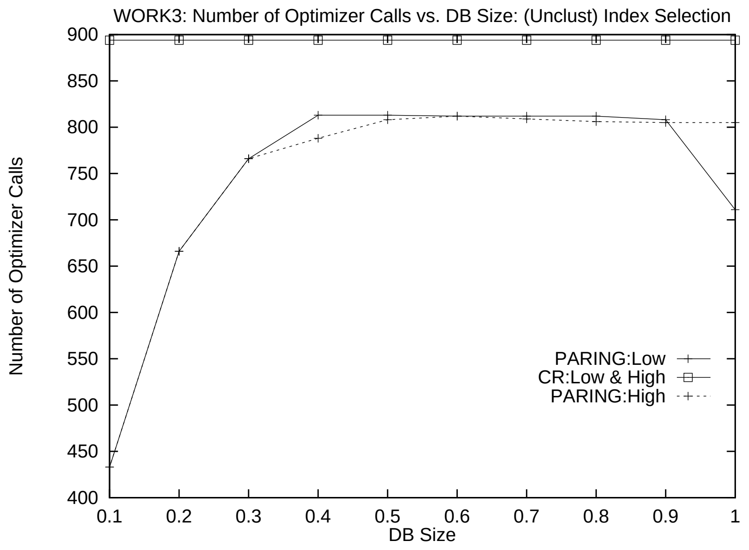
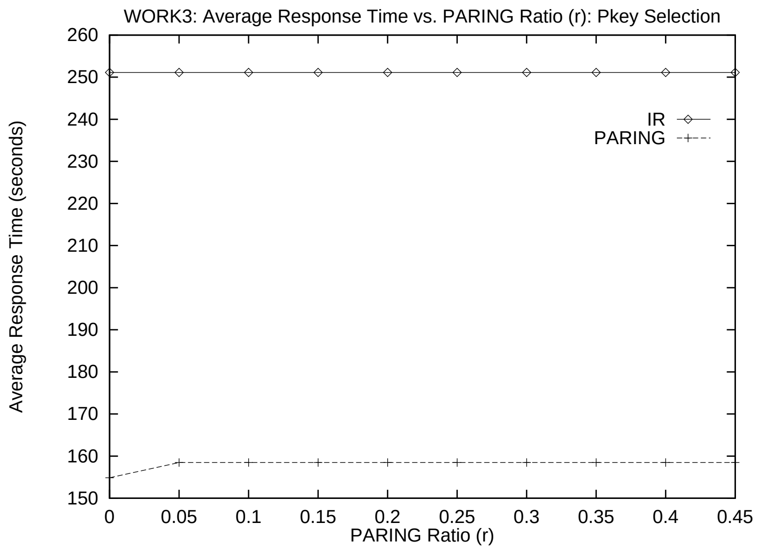


Figure 5.10: Average response time and number of optimizer calls for each algorithm versus the DB size factor when selecting indexing keys for workload WORK3.

However, CR does return ART values closer to the best ART for α values closer to one. Some ART results for CR can be seen in Figure 5.12 for workload WORK3.

As α increases, the number of optimizer calls made by CR also increases, as seen in Figure 5.12. This increase occurs because higher α values allow for less pruning of the keys. The number of optimizer calls increases to become more than what was required for the PARING algorithm at $r = 0$.



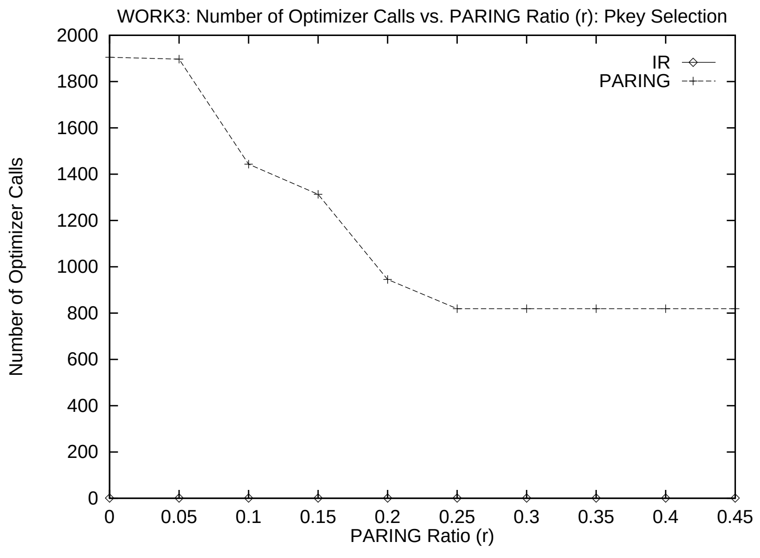
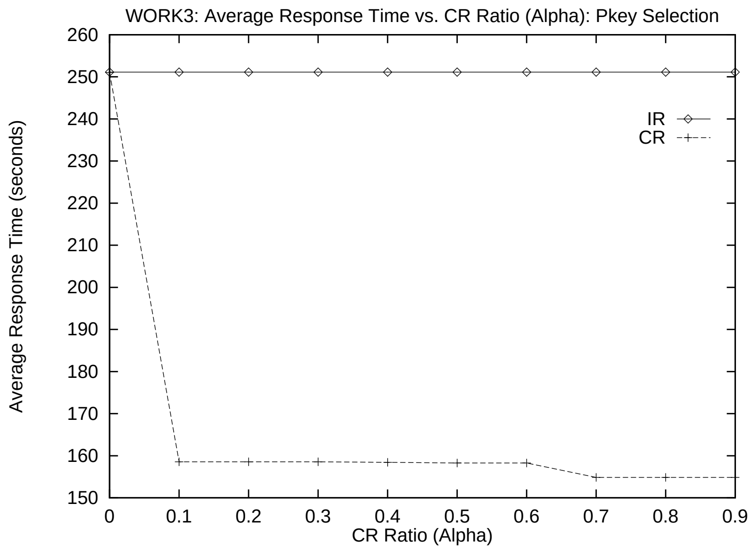


Figure 5.11: Average response time and number of optimizer calls for each algorithm versus the PARING ratio r when selecting partitioning keys for workload WORK3.

From our experiments, we conclude that PARING should be used when we must guarantee the resulting ART is at worst a specified percentage worse than the best ART at $r = 0$. In our other experiments, we chose $r = 0$ because it results in the worst case for the number of



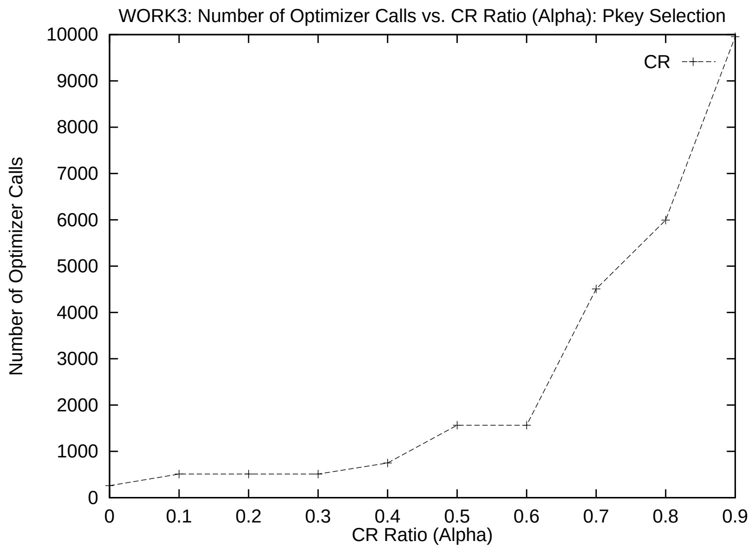


Figure 5.12: Average response time and number of optimizer calls for CR versus the CR factor α when selecting partitioning keys for workload WORK3.

optimizer calls required. We also chose $\alpha = 0.1$ since it requires a low number of optimizer calls, but results in good ART results. This causes us to use a worst case assumption for PARING, and a favorable case assumption for CR. Thus, if PARING is better than CR under this situation, we can confidently claim that PARING is better than CR under other cases.

5.1.3 Varying Resource Service Times

We investigated the effects on the algorithms of varying resource service times (such as communication latency and instruction path lengths). The goal was to assess the algorithms with a variety of combinations of different system characteristics. We vary one resource's service time across the system, and determine the ART results and number of optimizer calls over the different service times. The goal was to determine under which service times the ART differences between PARING and both CR and IR increase. We also determine the service times that cause the number of optimizer calls for PARING to be lower than the number of calls for CR.

A single resource's service times was varied in the following way. The first service times varied were the instruction path lengths. The path lengths found in Table 2.2 were varied by multiplying each instruction path length by a factor. This factor was given values between one and five, where larger factors result in longer processing times. Another parameter we varied was the service time of the interconnection network. We varied the communication latency (COMMTIME) from 4 microseconds to 2 milliseconds in order to simulate from slow to fast interconnection networks.

In selecting partitioning or indexing keys, the PARING algorithm results in better average response time results than IR or CR. This can be seen for workload WORK3 in Figures 5.13 and 5.14 when the instruction path lengths are varied. The results for IR at the Med rates exceed the y-axis values shown. We also saw this improvement of PARING using other workloads over the different communication timings, such as WORK2 shown in Figure 5.15 and Figure 5.16 for partitioning and indexing (with clustering) key selection, respectively.

For longer resource times, the difference in resulting average response times between IR and the other algorithms increases, and PARING results in fewer optimizer calls. For example, the number of optimizer calls for WORK3 under index selection for various instruction path lengths is shown in Figure 5.14. Even with an α factor of only 0.1, CR causes about 20% as many optimizer calls as PARING for WORK3. We also show the number of calls for workload WORK2 under different communication times for partitioning key selection in Figure 5.15. This increase results from reducing processor or communication costs that become more important to the overall response time when the workload moves from being I/O-bound to CPU- or communication-bound, and PARING can find designs that improve the CPU and communication costs more than CR. Also, for the longer resource service times, more designs

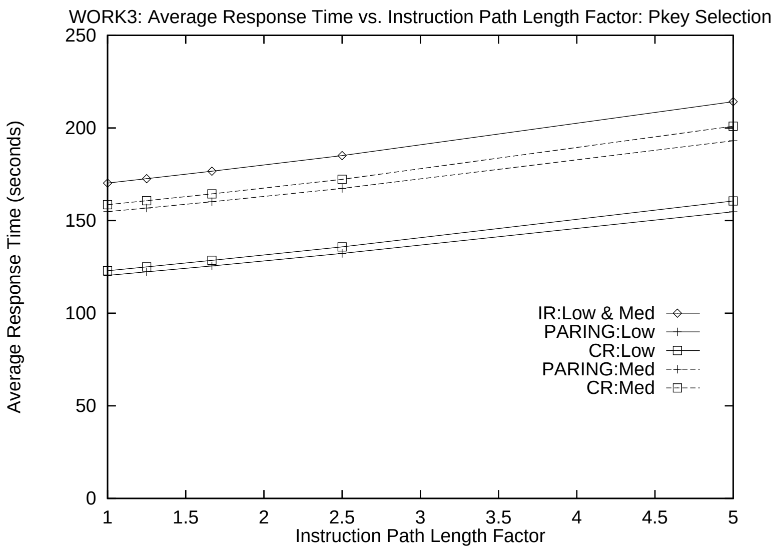


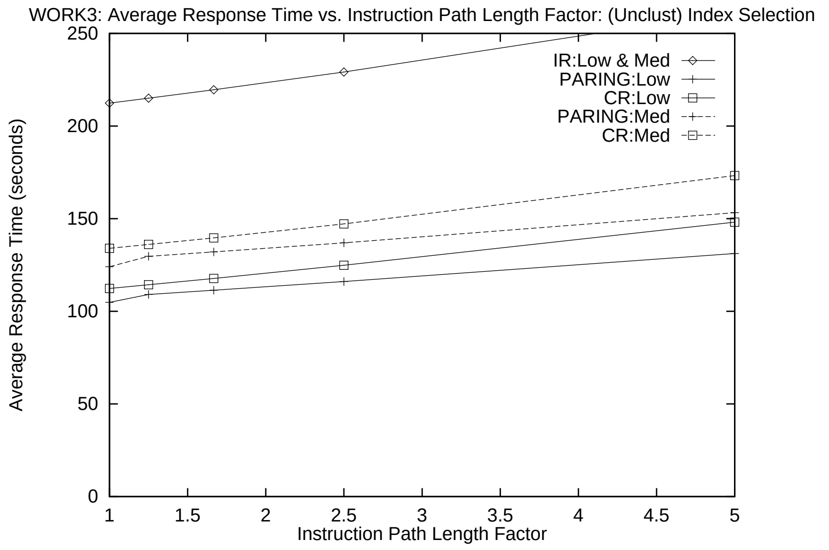
Figure 5.13: Average response time versus the factor multiplying the initial instruction path length when selecting partitioning keys for workload WORK3.

that fail to support the workload are found. Thus, PARING will prune more of these cases, which leads to fewer optimizer calls.

5.1.4 Assigning Non-Equal Arrival Rates

We assumed in all of the previous experiments that the arrival rates for all the queries in a workload are the same (except for query two of WORK4, which had twice the arrival rate of the other queries). In this section, we describe a set of experiments that vary the arrival rates so that they are not equal for all transaction classes. This was done to determine the arrival rates that improve the ART results and number of optimizer calls for PARING over those for CR. We did this by varying the arrival rate (relative frequency) of the most important query (the one with the highest value for its arrival rate times its response time using a design from the IR algorithm). The arrival rate was varied by multiplying it by a factor greater than or equal to one.

We observe that, when the relative frequency factor increases, the number of optimizer calls decreases. This should apply to all workloads since the increased relative frequency factor affects the most important query and thus the ART more, which will permit more extensive pruning. For workload WORK2, Figures 5.17 and 5.18 show these results for partitioning and indexing key selection, respectively. Since the increase of the arrival rate using a factor from 1 to 3 does not introduce many more designs that fail to support the workload, the main reason for the decrease in the calls for factor 3 was due to the frequency change. We can see that this is true by noting that the number of optimizer calls for CR also decreases. This decrease is



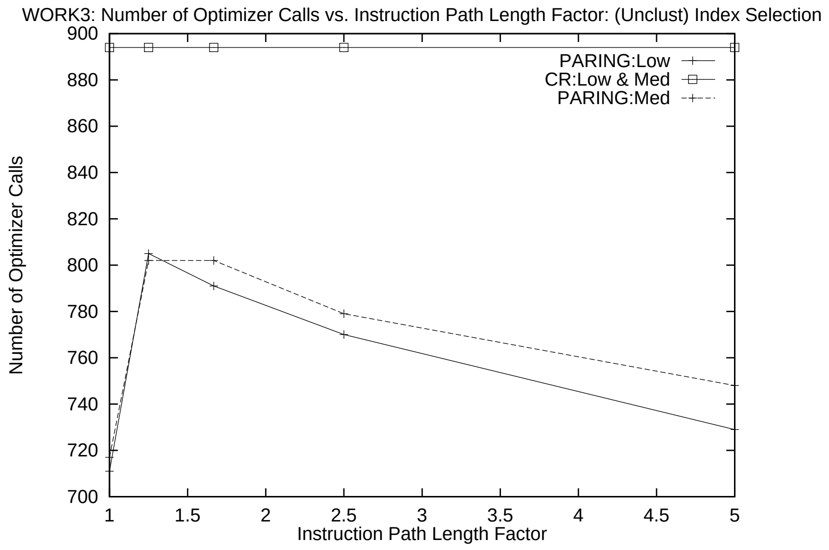
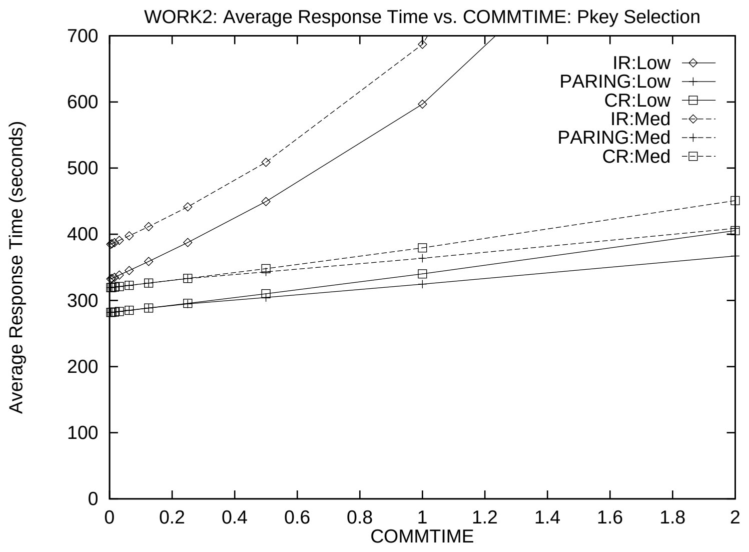


Figure 5.14: Average response time and number of optimizer calls for each algorithm versus the factor multiplying the initial instruction path length when selecting indexing keys for workload WORK3.



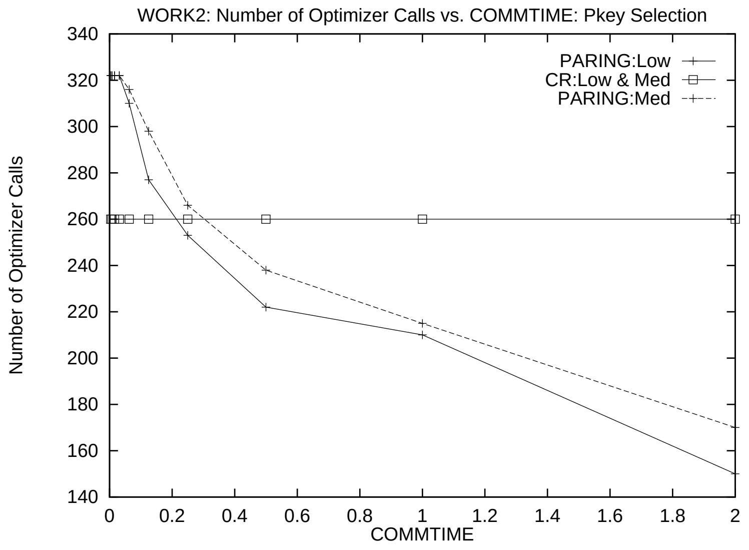


Figure 5.15: Average response time and number of optimizer calls for each algorithm versus the communication latency (in milliseconds) when selecting partitioning keys for workload WORK2.

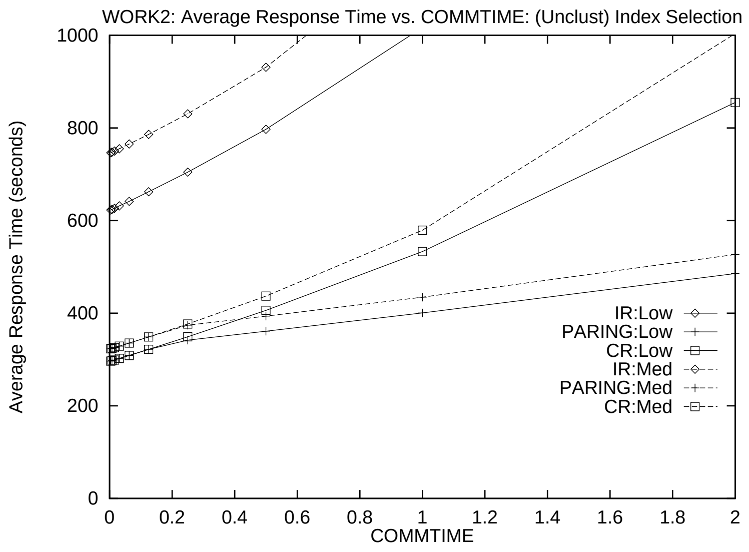


Figure 5.16: Average response time versus the communication latency (in milliseconds) when selecting indexing keys for workload WORK2.

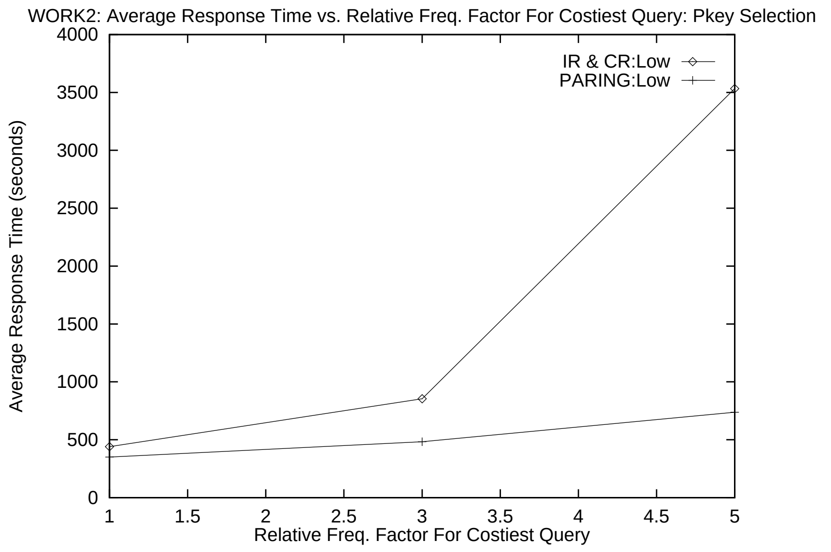
related to the fact that the pruning for CR is dependent on the weightings, which are directly dependent on the values for the arrival rates. Higher arrival rates for a subset of queries leads to greater differences of the weights of the keys for a relation, and thus causes more pruning of the keys not within a factor $1 - \alpha$ from the highest weighted key. The combinatorial search of CR and PARING are still important as IR's design does not result in the best possible ART (e.g., as seen in Figure 5.17).

The number of optimizer calls for PARING is smaller than the number required for CR when the arrival rate factor increases. PARING allows more pruning because the most important query adds more to any PART value and thus the PART values are closer to the ART values for the workload. This closeness allows for more pruning of partial solutions compared to the current best known bound. It should be noted that the number of optimizer calls is higher when the arrival rates are higher because this basically decreases the PART values compared to the ART values.

5.1.5 Summary of Algorithm Comparisons

We have made a number of findings in our comparisons among the different physical DB design algorithms.

1. We found that PARING and CR are always good and always resulted in average workload response times that were better than IR's resulting times. In some cases, although a design from IR fails to support the workload, PARING and CR were able to still find a design that supports the workload.



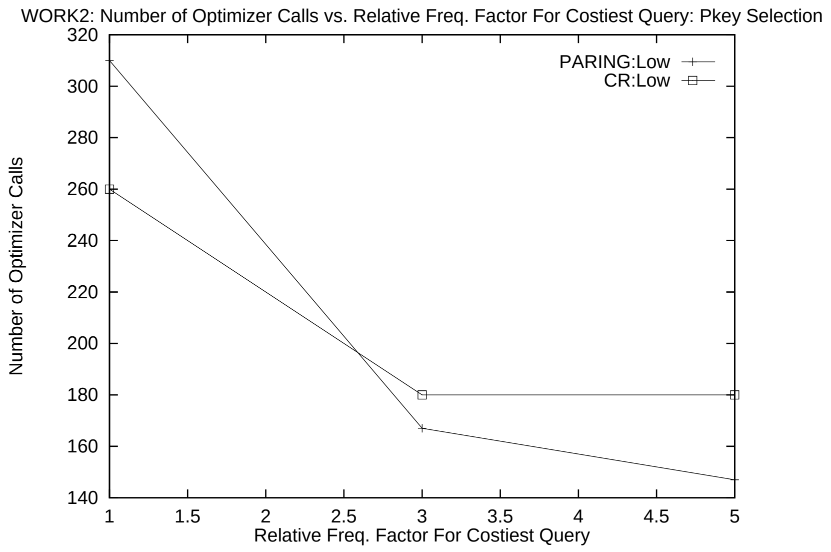
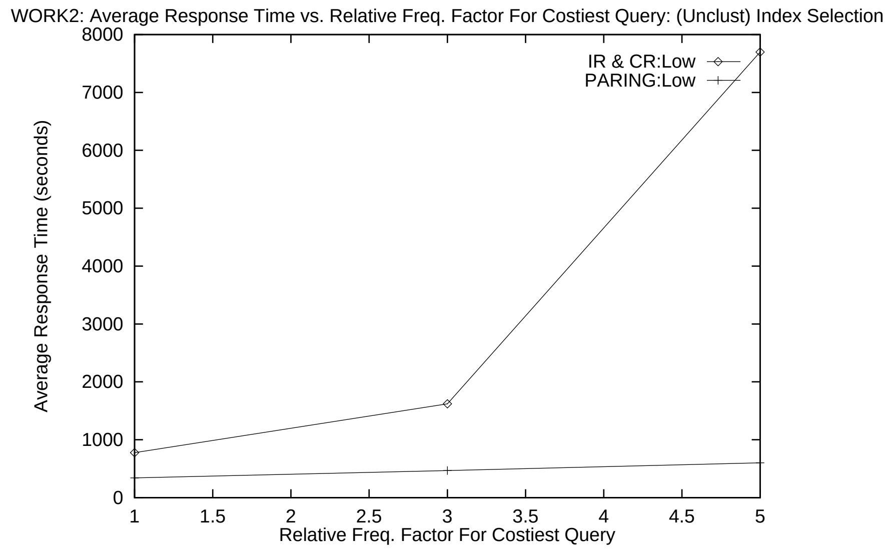


Figure 5.17: Average response time and number of optimizer calls for each algorithm versus the arrival rate (relative frequency) factor for the most important query when selecting partitioning keys for workload WORK2.



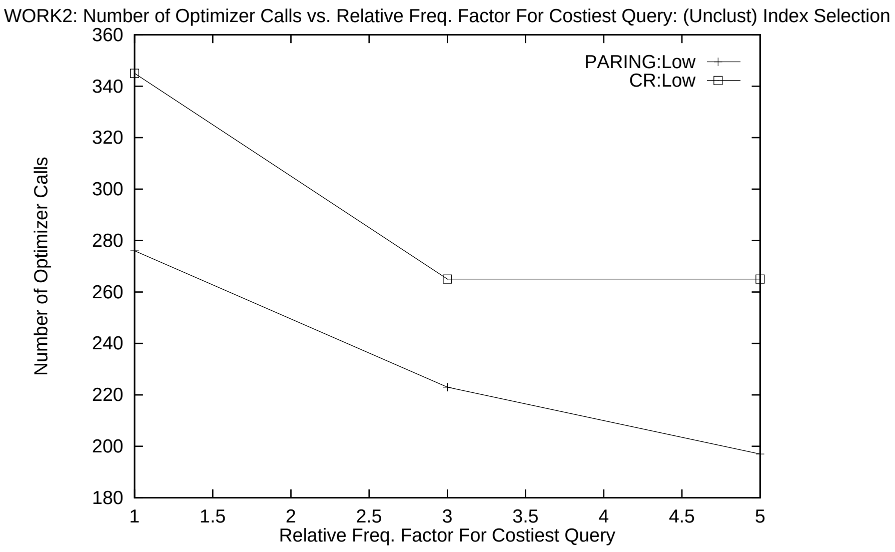


Figure 5.18: Average response time and number of optimizer calls for each algorithm versus the arrival rate (relative frequency) factor for the most important query when selecting indexing keys for workload WORK2.

2. The proper choice for a physical DB design is dependent on the input values for the DB size, query characteristics (such as arrival rates), and system parameters.

3. Pre-scanning was found to assist in reducing the number of search cases for both PARING and CR.

4. PARING resulted in either the same or better ART as CR. Also, the number of optimizer calls (or the cost of the algorithms' executions) required to determine the design for PARING was sometimes lower than the number of calls required for CR.

5. Some designs result in high utilizations for a workload, and high utilizations cause greater delays (resource contention) at the resources, which in turn increase the workload response times. These increased response times cause the PART values for a subset of queries to be close to the ART values for their corresponding designs and close to the current best upper bound in a PARING execution. Thus, a design resulting in higher utilizations than other designs leads to pruning this design.

6. Having arrival rates in a workload that are greater for the more important queries results in a reduction of the number of optimizer calls for PARING and CR compared to when the more important queries have lower rates. Arrival rates for these queries that are higher than those of less important queries also result in the number of calls for PARING to be lower than for CR.

7. Performing the combinatorial searching of CR and PARING leads to designs in which all resources' residence times (processor, disk, and communication times) are smaller than those of IR for both partitioning key and indexing (with clustering) key selection.

8. Larger DB sizes cause the response time percent reductions between the ART's from the algorithms to increase, where the increase favors PARING. Also, the difference between the response times of the algorithms was larger for slower interconnection networks and processors as compared to faster ones. There is also an increase in the difference when the arrival rates of the queries in the workload are high rather than low.

9. The number of optimizer calls decreases as the PARING ratio (r) increases while the number of calls increases as the CR filter factor (α) increases. As α increases, the ART from CR moves toward the best ART (found by PARING using $r = 0$), but there are no guarantees on how close the ART from CR is to this best ART when $\alpha \leq 1$. Varying the ratio r for PARING provides guarantees on the closeness to the best ART.

From our findings, we conclude that the PARING algorithm should be chosen over the other algorithms when a design leading to a good response time is desired at a moderate cost of executing the search algorithm.

5.2 Evaluating the PARING Algorithm

To show how the PARING algorithm performs under various scenarios, we present a few experiments that vary some of the input parameter values. We have included the following experiments:

- In Section 5.2.1, we describe experiments in which we varied the DB sizes to see how PARING is affected by different sizes relative to the same system characteristics.
- We investigate the effects of using different values for the ratio r in executing PARING. These results are given in Section 5.2.2.
- Another experiment described in Section 5.2.3 varies the initial DB design that PARING would use. One of these designs is output by the IR algorithm. We want to determine whether using the design from IR to initiate PARING results in a low number of optimizer calls relative to other possible initial designs.
- Resource services times of the interconnection network, processors, and disk drives were varied in experiments described in Section 5.2.4. The purpose of these experiments is to study how PARING is affected by different system characteristics.
- The arrival rates of the classes in a workload are modified for one set of experiments described in Section 5.2.5. We determine whether the load on the system affects the ART and the number of optimizer calls for PARING.
- Section 5.2.6 discusses the importance of updates in a workload. Our goal is to investigate whether adding more updates affects the indexes chosen by PARING.
- Since we allow the elimination of nonclustering indexes in the pre-scanning phase, and since this elimination violates the property of separability [WWS85], we carried out experiments for Section 5.2.7 to compare the ART and number of optimizer calls from PARING when nonclustering indexes are eliminated or not in the pre-scan.
- For Section 5.2.8, we perform experiments to show whether it is better to select the partitioning, indexing, and clustering keys in one algorithm execution or with separate algorithm executions.
- In Section 5.2.9, we provide an experiment to validate the relative values of a workload's estimated ART with measured times from an IBM DB2 PE system.

A summary of the important results found from the experiments in this section is given in Section 5.2.10.

5.2.1 Varying the DB Sizes

When we presented the comparison among the algorithms in Section 5.1, we included experiments that varied the DB sizes used by the PARING algorithm. From these experiments, we noticed that there are two possibilities for how the number of optimizer calls changes for different arrival rates. It was seen that the higher arrival rates resulted in a lowering of the number of optimizer calls due to an increased number of designs encountered that fail to support the workload, thus allowing greater pruning by PARING. This can be seen in Figure 5.9. We also see this reduction in the number of calls when the DB size increases, e.g., Figure 5.10 shows a reduction in calls between DB size factor 0.9 and 1. However, under some circumstances, the number of optimizer calls for the higher arrival rates increases because the pruning using the best known bound is less severe when the arrival rates cause the PART values for a subset of queries to be lower than the ART values for the workload (e.g., see Figure 5.8).

We can provide an intuitive explanation of this phenomenon relating to the number of optimizer calls required over different parameters values, e.g., arrival rates.

- When the load on the system is low, there is little queuing, so the query classes' response times do not fluctuate except for the class helped by the change in the chosen design, e.g., an added index. Because of the small variation, when PARING finds a good best known solution, it can be used to prune numerous solutions with similar performance. Pruning more partial solutions results in a reduction in the number of optimizer calls.
- As the load becomes heavier, one or more resources comes close to saturation, so small changes in utilization can cause large changes or even saturation in the ART values for the solutions. Thus, lots of partial designs have many extensions that have to be considered because there are more of these designs that have their PART values further from the ART values of the best known designs.
- At still heavier loads, saturation is reached earlier in the tree expansion (after only a few query classes are considered) while the tree is not yet very broad, and this causes a reduction in the total number of designs to consider.

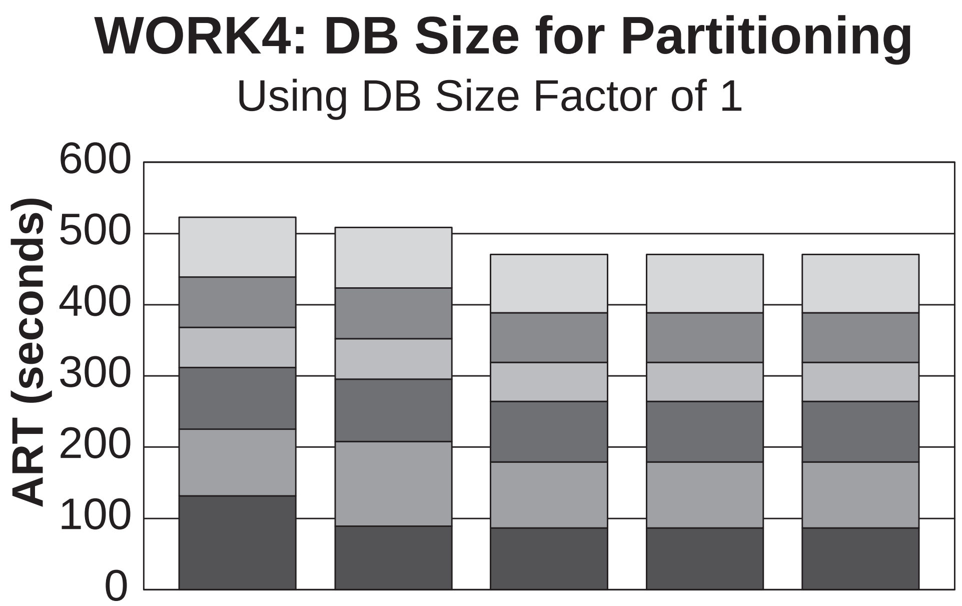
When saturation is involved, PARING is affected in a number of ways. In order for PARING to perform pruning based on the ART of the best known complete solution, it must find a complete solution that supports the workload. If the design from IR fails to support the workload, then PARING must find a design during its search. However, for some database sizes and arrival rates, it is possible that no design exists that supports the workload. Since PARING prunes away solutions that fail to support the workload, the number of optimizer calls will be controlled. There is more pruning in this case when the minimum costs of the queries together lead to a saturated system or when a small number of queries in the full workload that are the first queries considered by PARING saturate the system. It may be the case that all designs fail to support the workload. However, saturation may not be detectable until the costs of most of the queries in the workload are considered. Since all designs fail to support the workload, the upper bound will never be set by PARING, so no pruning will be done using a best known bound. These factors would cause our algorithm to result in a high number of optimizer calls.

To illustrate how the PARING algorithm makes its selection of the best solutions, we have produced bar graphs showing all the changes to a best current solution as the algorithm executes once. We refer to the set of changes as the *history* of the algorithm's execution, and the points when a change is made are each referenced by a *history number*. The height of each bar is the average workload response time for the corresponding design chosen at that point. The first bar is always the IR's ART since all designs from IR in our examples support the corresponding workload. The last bar is always the best ART found by the algorithm. We also show the contribution each query makes to an average response time by dividing the area within a bar into areas for each query. For example, if a query has response time R with arrival rate f and the sum of the arrival rates for the workload is F, the response time contribution from this query is $\frac{fR}{F}$. Note that, in these bar graphs, query 1 is always located at the bottom of each bar and the last query is the area at the top of each bar.

Along with the bar graph, we will give the corresponding design for each *history number* in a table. This table will include the queries that are directly affected by the selections for each *history number*. Each list of keys in these tables will contain only the change in the keys from the previous *history number*.

In Figure 5.19, the *history* is shown when PARING selects the partitioning keys for WORK4 for a database with size factor of 1.0. The corresponding designs for the histories and the queries affected by the selection are given in the lower table of Figure 5.19. From the second bar, we see that the first change made by PARING improved on the response time of query 1. Note that, although some of the bars do seem equivalent, each new change brings out a better ART value, although some of the changes are too small to be seen with the number of significant digits shown in the tables. These slight changes (e.g., between *history numbers* 3, 4, and 5) are due to the PARING ratio (r) being set to zero so that every improvement, however small, is recorded. Increasing the value of r would allow for small changes to be avoided. For example, when we used $r = 1\%$, we obtain a history with three bars that are the same as the first three in Figure 5.19.

We also show PARING's behavior during index selection. In Figure 5.20, we show the index selection for WORK3 with a database of size factor 1.0. The successive reductions in query costs along with the ART are also exhibited by this example. The indexes selected are shown in the lower table of Figure 5.20 along with the queries directly affected and the ART's for each



History Number		1	2	3	4	5
Query 1	■	131.7	89.2	86.7	86.7	86.7
Query 2	■	93.6	119.1	92.6	92.6	92.6
Query 3	■	86.9	87.7	85.2	85.2	85.2
Query 4	■	56.3	56.8	55.2	55.2	55.2
Query 5	■	70.9	71.4	69.5	69.5	69.5
Query 6	■	83.8	84.9	82.1	82.1	82.1

History Number	**ART (secs)**	**Keys Selected**	**Queries Directly Affected**
1	523.1	All keys are the first attribute (A1) except for R2.A4 and R10.R2	All queries
2	509.1	R1.A2, R2.A5, R6.A3, and R11.A3	Queries 1, 5, and 6
3	471.3	R2.A4	Queries 2, 3, 4, and 6
4	471.2	R1.A3	Query 6
5	471.2	R1.A1	Queries 4 and 6

Figure 5.19: History of changes to the best solution by PARING for workload WORK4 and selecting partitioning for DB size factor 1.0. The lower table shows the partitioning keys selected and queries affected by PARING's change of the best solution.

history number.

5.2.2 Comparing Ratios (r) for the PARING Algorithm

To further demonstrate how PARING performs using different values of r, we allowed r to vary between 0 and 0.45, and we plotted the resulting ART values and the number of optimizer calls. As expected, the resulting response times from PARING increase when the ratio r increases. However, the number of optimizer calls required by PARING decreases when r is increased due to the pruning of solutions that are more than $r\%$ above the best ART (found with $r = 0$). We show these results for WORK1. For partitioning selection, the average workload response time from IR is $1,400$ seconds while the ART from PARING is given in Table 5.2. For index selection using WORK1, the response time for IR is 724 seconds while the ART from PARING is given in Table 5.3. The number of optimizer calls for these respective cases are also shown in Tables 5.2 and 5.3.

r	0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45
ART	389	394	394	394	394	394	394	394	394	394
#Calls	438	438	428	428	428	428	428	428	428	428

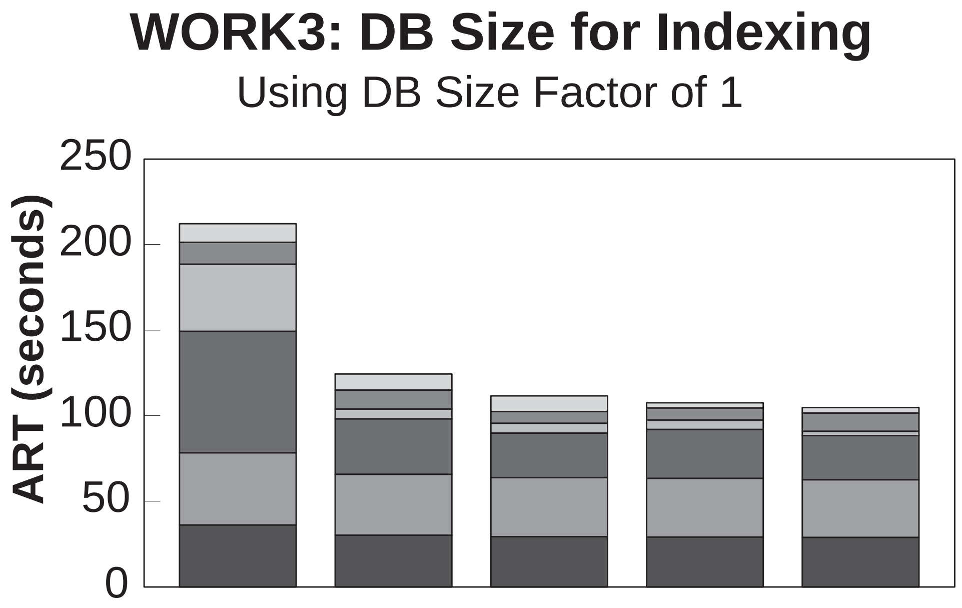
Table 5.2: Average response time (in seconds) and number of optimizer calls for PARING versus the PARING ratio r when selecting partitioning keys for workload WORK1.

r	0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45
ART	92.3	95.8	95.8	95.8	95.8	95.8	95.8	95.8	95.8	95.8
#Calls	842	840	806	802	780	756	740	735	735	735

Table 5.3: Average response time (in seconds) and number of optimizer calls for PARING versus the PARING ratio r when selecting indexing keys for workload WORK1.

We show the differences between the PARING ratio (r) values by providing *history* bar graphs for WORK1 when selecting keys using different ratios. Other workloads exhibited the same trends for partitioning and indexing key selection. In Figure 5.21, we see the *history* for partitioning key selection when the ratio for PARING is set to zero. Figure 5.21 also has a lower table that shows the ART for each *history number*. When the ratio is set to 0.3, as shown in Figure 5.22, the number of times the best solution is changed is reduced, and ratio $r = 0$ provides a better answer than the higher ratio. This is also true for index selection as shown for WORK1 between ratios 0 (seen in Figure 5.23) and 0.1 (as seen in Figure 5.24).

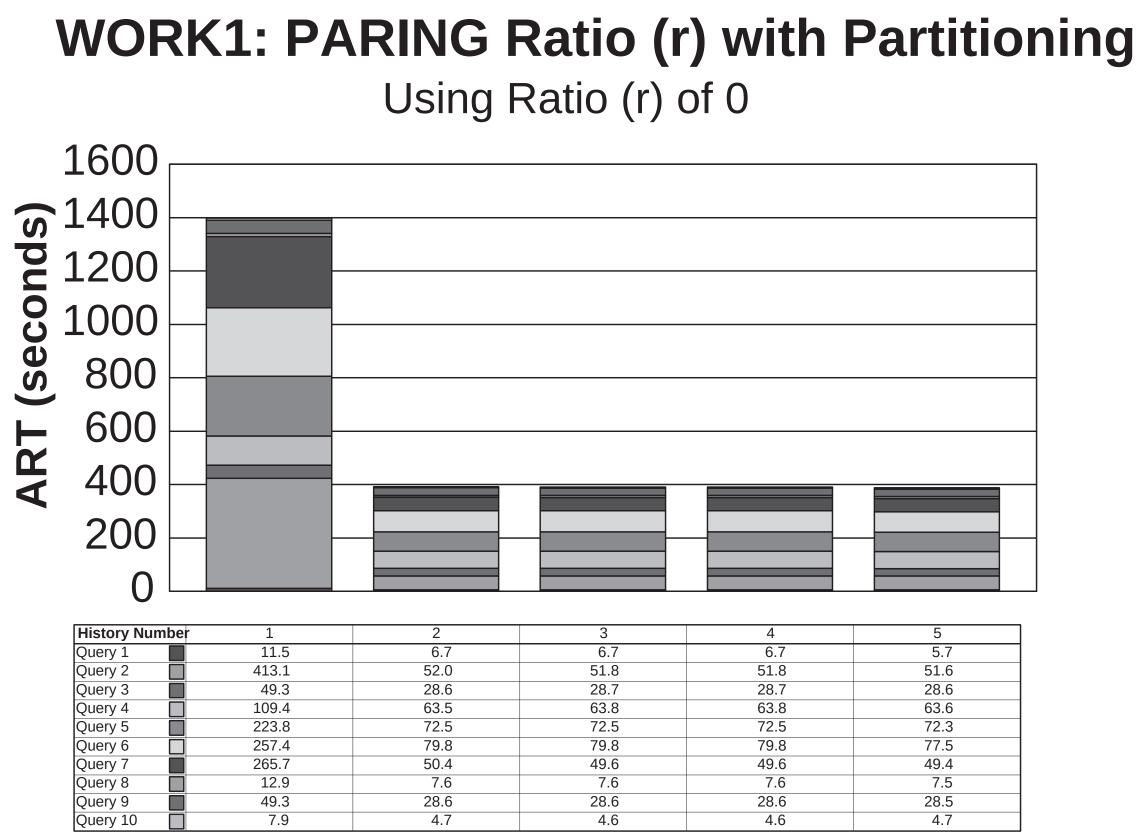
The value of r actually affects a history by suppressing any history step that is not at least an $r\%$ improvement in its ART value. This suppression causes the designs found by PARING



History Number		1	2	3	4	5
Query 1	■	36.1	30.2	29.3	29.1	29.0
Query 2	□	42.4	35.5	34.5	34.3	33.6
Query 3	■	71.0	32.5	26.2	28.7	25.9
Query 4	□	39.1	5.9	5.6	5.6	2.6
Query 5	■	12.8	11.1	7.0	6.9	10.7
Query 6	□	10.9	9.4	9.2	3.0	3.0

History Number	**ART (secs)**	**Keys Selected**	**Queries Directly Affected**
1	212.4	No indexes	All queries
2	124.6	R1.A1 (nonclustered)	Queries 2, 3, and 4
3	111.8	R1.A1, R3.A2, R4.A1, R5.A2 R6.A6, and R7.A2 (all clustered)	Queries 1 to 6
4	107.7	R4.A6 (clustered)	Queries 3 and 6
5	104.8	R4.A1, R5.A5, R6.A6 (all clustered) and R7 has no indexes	Queries 1, 3, 5, and 6

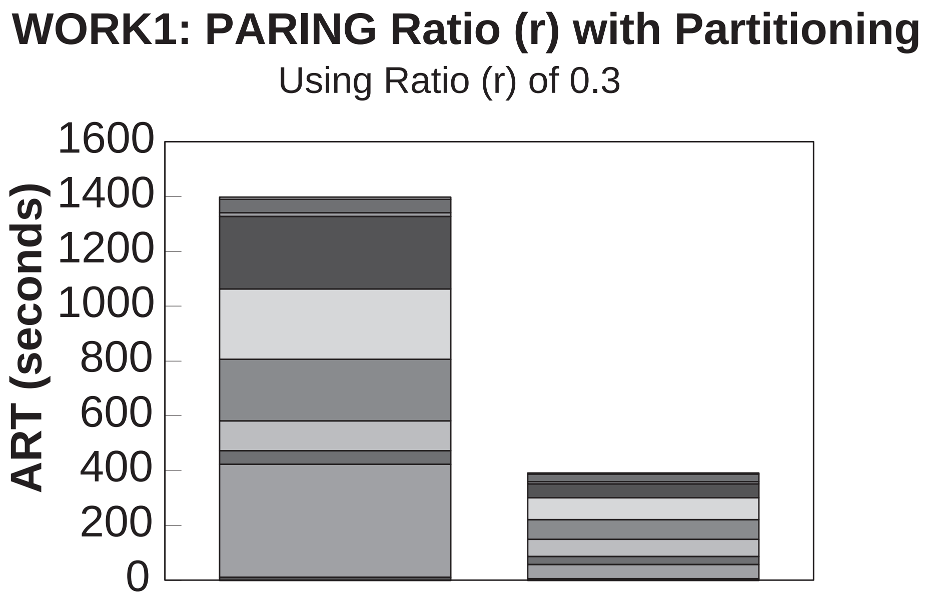
Figure 5.20: History of changes to the best solution by PARING for workload WORK3 and selecting indexes for DB size factor 1.0. The lower table shows the indexing keys selected and queries affected by PARING's change of the best solution.



History Number	1	2	3	4	5
Query 1 ■	11.5	6.7	6.7	6.7	5.7
Query 2 ■	413.1	52.0	51.8	51.8	51.6
Query 3 ■	49.3	28.6	28.7	28.7	28.6
Query 4 ■	109.4	63.5	63.8	63.8	63.6
Query 5 ■	223.8	72.5	72.5	72.5	72.3
Query 6 ■	257.4	79.8	79.8	79.8	77.5
Query 7 ■	265.7	50.4	49.6	49.6	49.4
Query 8 ■	12.9	7.6	7.6	7.6	7.5
Query 9 ■	49.3	28.6	28.6	28.6	28.5
Query 10 ■	7.9	4.7	4.6	4.6	4.7

History Number	**ART (secs)**	**Keys Selected**	**Queries Directly Affected**
1	1400.2	LINEITEM.L_SUPPKEY and all other relations have first attribute	All queries
2	394.4	LINEITEM.L_ORDERKEY and CUSTOMER.C_PHONE	Queries 2 to 7
3	393.6	CUSTOMER.C_CUSTKEY, SUPPLIER.S_ACCTBAL, and NATION.N_REGIONKEY	Queries 1,2,4,5,6,7 and 8
4	393.5	SUPPLIER.S_SUPPKEY	Queries 1,4,5,6, and 8
5	389.5	PARTSUPP.PS_PARTKEY and SUPPLIER.S_ACCTBAL	Queries 1,4,5,6,8, and 10

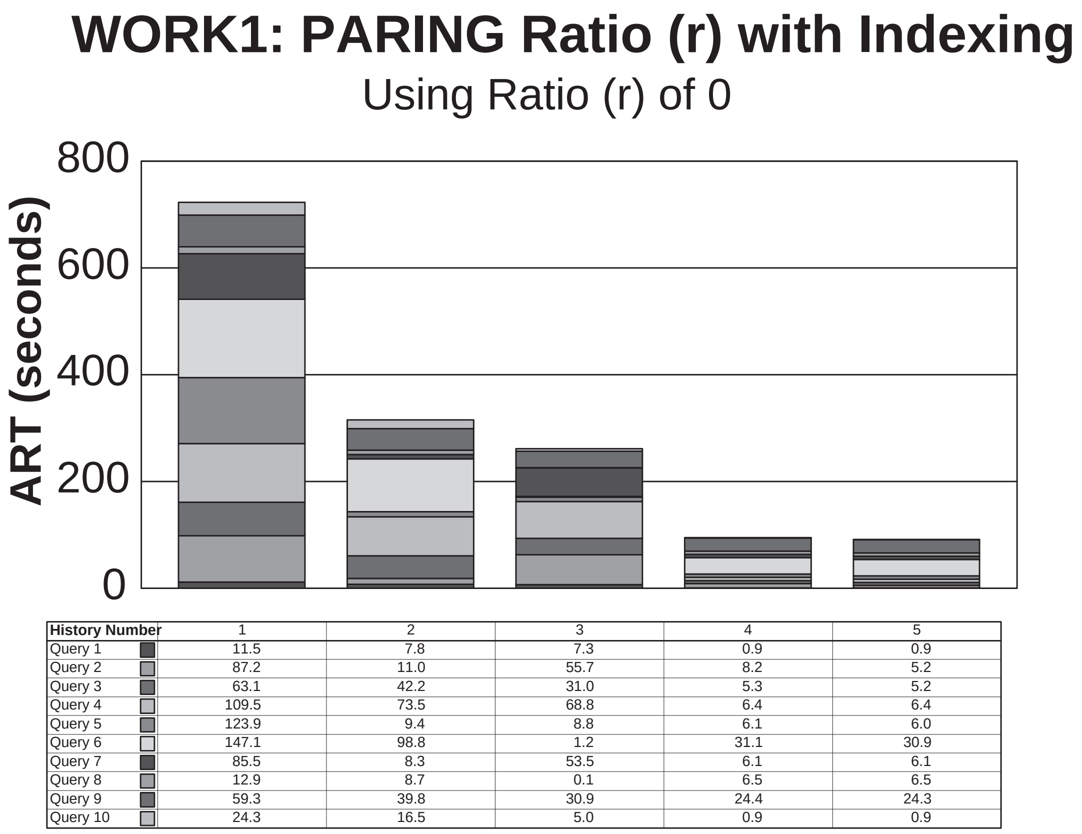
Figure 5.21: History of changes to the best solution by PARING for workload WORK1 and selecting partitioning for ratio (r) of zero. The lower table shows the partitioning keys selected and queries affected by PARING's change of the best solution.



History Number		1	2
Query 1	■	11.5	6.7
Query 2	■	413.1	52.0
Query 3	■	49.3	28.6
Query 4	■	109.4	63.5
Query 5	■	223.8	72.5
Query 6	■	257.4	79.8
Query 7	■	265.7	50.4
Query 8	■	12.9	7.6
Query 9	■	49.3	28.6
Query 10	■	7.9	4.7

History Number	ART (secs)	Keys Selected	Queries Directly Affected
1	1400.2	LINEITEM.L_SUPPKEY and all other relations have first attribute	All queries
2	394.4	LINEITEM.L_ORDERKEY and CUSTOMER.C_PHONE	Queries 2 to 7

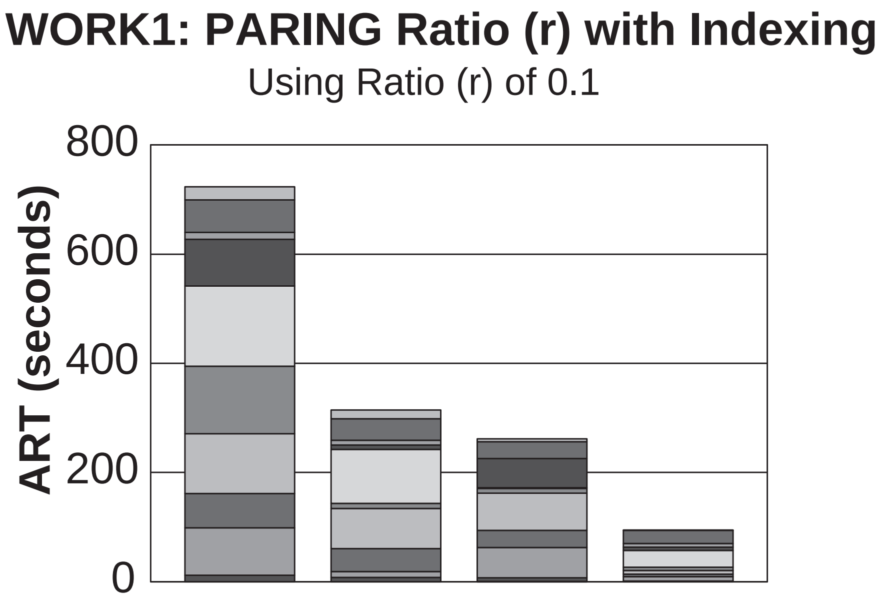
Figure 5.22: History of changes to the best solution by PARING for workload WORK1 and selecting partitioning for ratio (r) of 0.3. The lower table shows the partitioning keys selected and queries affected by PARING's change of the best solution.



History Number	1	2	3	4	5
Query 1 ■	11.5	7.8	7.3	0.9	0.9
Query 2 ■	87.2	11.0	55.7	8.2	5.2
Query 3 ■	63.1	42.2	31.0	5.3	5.2
Query 4 □	109.5	73.5	68.8	6.4	6.4
Query 5 ■	123.9	9.4	8.8	6.1	6.0
Query 6 □	147.1	98.8	1.2	31.1	30.9
Query 7 ■	85.5	8.3	53.5	6.1	6.1
Query 8 ■	12.9	8.7	0.1	6.5	6.5
Query 9 ■	59.3	39.8	30.9	24.4	24.3
Query 10 □	24.3	16.5	5.0	0.9	0.9

History Number	**ART (secs)**	**Keys Selected**	**Queries Directly Affected**
1	724.2	No indexes	All queries
2	316	LINEITEM.L_ORDERKEY (nonclustered)	Queries 2 to 7
3	262.3	LINEITEM.L_PARTKEY, ORDERS.O_ORDERKEY, PARTSUPP.PS_SUPPKEY, and PART.P_PARTKEY (all clustered)	Queries 1 to 10
4	95.8	LINEITEM has no index and PARTSUPP.PS_PARTKEY (clustered)	Queries 1,2,3,4,5,6,7,8 and 10
5	92.3	ORDERS.O_CUSTKEY (clustered)	Queries 2,4,5, and 7

Figure 5.23: History of changes to the best solution by PARING for workload WORK1 and selecting clustering and indexing for ratio (r) of zero. The lower table shows the indexing keys selected and queries affected by PARING's change of the best solution.



History Number		1	2	3	4
Query 1	▨	11.5	7.8	7.3	0.9
Query 2	▨	87.2	11.0	55.7	8.2
Query 3	▨	63.1	42.2	31.0	5.3
Query 4	▨	109.5	73.5	68.8	6.4
Query 5	▨	123.9	9.4	8.8	6.1
Query 6	▨	147.1	98.8	1.2	31.1
Query 7	▨	85.5	8.3	53.5	6.1
Query 8	▨	12.9	8.7	0.1	6.5
Query 9	▨	59.3	39.8	30.9	24.4
Query 10	▨	24.3	16.5	5.0	0.9

History Number	ART (secs)	Keys Selected	Queries Directly Affected
1	724.2	No indexes	All queries
2	316	LINEITEM.L_ORDERKEY (nonclustered)	Queries 2 to 7
3	262.3	LINEITEM.L_PARTKEY, ORDERS.O_ORDERKEY, PARTSUPP.PS_SUPPKEY, and PART.P_PARTKEY (all clustered)	Queries 1 to 10
4	95.8	LINEITEM has no index and PARTSUPP.PS_PARTKEY (clustered)	Queries 1,2,3,4,5,6,7,8 and 10

Figure 5.24: History of changes to the best solution by PARING for workload WORK1 and selecting clustering and indexing for ratio (r) of 0.1. The lower table shows the indexing keys selected and queries affected by PARING's change of the best solution.

using the corresponding r value to result in different ART values between a set of ranges of r values. For example, from Figure 5.21, we can deduce that, for various ranges of r, only certain history steps remain:

$0 \leq r \leq 0.0003$	results in history bars 1 2 3 4 5,
$0.0003 \leq r \leq 0.002$	results in history bars 1 2 3 5,
$0.002 \leq r \leq 0.0125$	results in history bars 1 2 5,
$0.0125 \leq r \leq 0.71$	results in history bars 1 2, and
$0.71 \leq r \leq 1$	results in history bar 1 only.

So, $r < 1.25\%$ results in PARING finding its best solution, $1.25\% \leq r < 71\%$ results in a solution with ART 394.4 seconds, and $r \geq 71\%$ results in a solution with ART equal to 1400.2 seconds.

5.2.3 Varying the Initial Design Used By PARING

In the description of PARING from Chapter 4, we stated that a good initial solution for the algorithm would be the solution from the naive IR algorithm. To determine how useful this is, we experimented with different initial solutions. From experiments described in Section 5.2.2, we saw that raising the ratio r causes the number of optimizer calls to be reduced. Thus, we decided to use the output design from the PARING algorithm using a given ratio $r > 0$ (actually $r \geq 0.2$) as the initial solution for PARING using a ratio of zero. We shall call the initial ratio ($r > 0$) the *start ratio*.

We will use WORK1 to demonstrate the results we saw for all the workloads and for either partitioning or indexing key selection. For partitioning key selection with WORK1, the average workload response times were the same across the different *start ratio* values ($1,400$ seconds for IR and 389 seconds for PARING). The average response times for index selection for workload WORK1 were 724 seconds for IR and 92 seconds for PARING. Note that IR has different resulting response times for partitioning than for indexing key selection. This difference exists because, for index selection, the default partitioning keys are unchanged by IR, and IR assumes no indexes exist, whereas for partitioning key selection, IR selects the partitioning keys but uses the indexing keys input with the DB information.

The graphs plotting the number of optimizer calls required for both partitioning key selection and index selection are given in Figures 5.25 and 5.26, respectively. For a *start ratio* value of zero, the first PARING execution would use the initial design from IR, and then find its best design. Giving this best design as the input to PARING would not change the result, so, for a *start ratio* of zero, we did not call PARING a second time. The execution using *start ratio* zero is equivalent to executing PARING with $r = 0$. The number of calls for $r = 0$ is used as the baseline for comparison among the non-zero ratios. If the number of calls for the non-zero *start ratios* is lower than the number of calls for $r = 0$, then we have found a better way to get the initial solution for PARING. We noticed that the number of calls using a *start ratio* greater than zero requires the lowest number of optimizer calls for a ratio of one. However, *start ratio* should be less than one to make sure a decent initial solution is found for the second execution of PARING. For our experiments, we see that using IR's design as the initial guess for PARING resulted in the lowest number of optimizer calls.

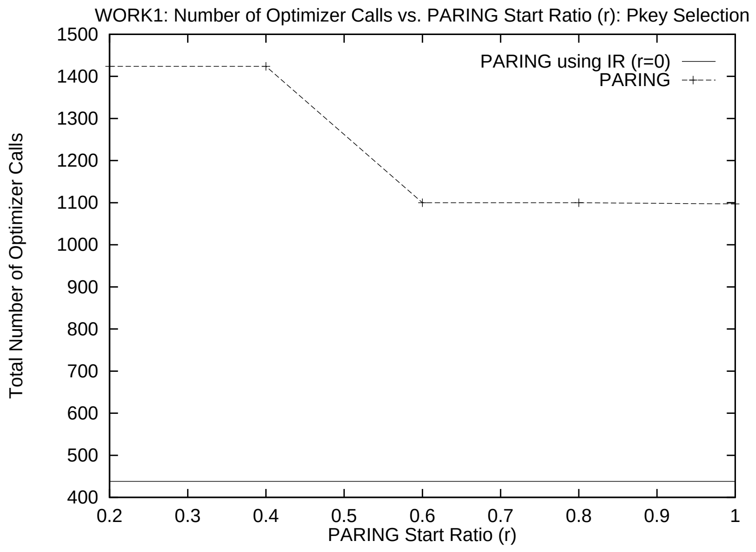


Figure 5.25: Total number of optimizer calls (determining the initial design and using PARING to obtain the final design) for PARING with *start ratio* r when selecting partitioning keys for workload WORK1.

5.2.4 Varying the Service Times of the Resources

We varied either the instruction path lengths, the disk access time, or the communication latency. As in Section 5.1, we investigate the effects of changing the instruction path lengths by multiplying the initial path lengths in Table 2.2 by a factor between one and five. Changes to the processor speeds can also be represented in this same way. We also varied the parameter *COMMTIME* to have latency values for fast and slow interconnect communication networks. To vary the disk access time (*IOTIME*) in order to emulate fast and slow disk access times, we used times between 1 and 17 milliseconds.

In all of these experiments, the slower resource service times resulted in higher ART times returned by PARING because of the increased load on the system. For example, for partitioning key selection using WORK1, the average response times of the algorithms increase for higher instruction path lengths as seen in Figure 5.27.

All of the experiments resulted in similar trends for the number of optimizer calls. This trend follows the ones outlined in Section 5.2.1. For short service times, the load on the system

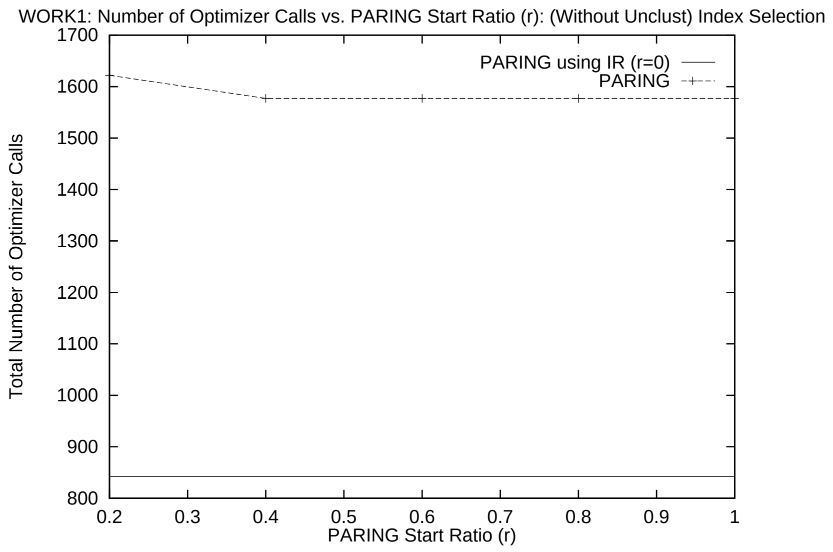


Figure 5.26: Total number of optimizer calls (determining the initial design and using PARING to obtain the final design) for PARING with *start ratio r* when selecting indexing keys for workload WORK1.

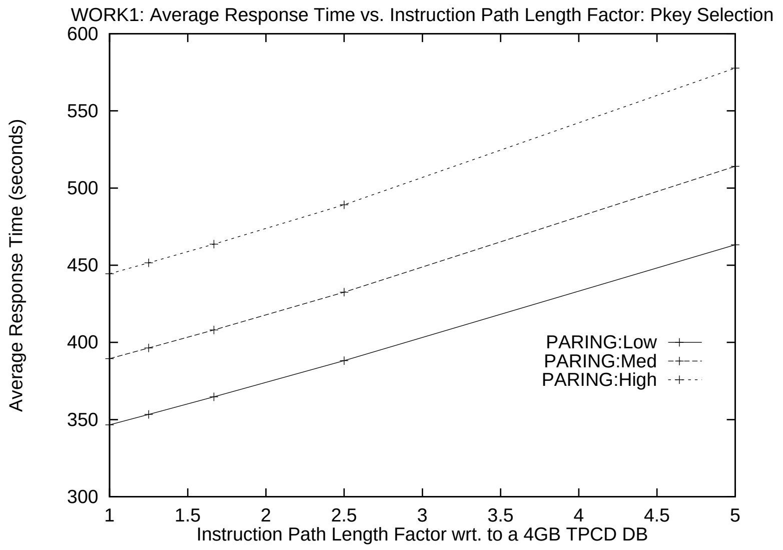


Figure 5.27: Average response time versus factor multiplying the initial instruction path length when selecting partitioning keys for workload WORK1.

is low, and these low loads result in low number of optimizer calls compared to the other service times. Heaviest load is incurred when we use long service times for a resource, which resulted in more saturation and a reduction in the number of optimizer calls. For loads in between, the number of optimizer calls is higher.

For example, in Figure 5.28, we see this trend when instruction path lengths are varied for index selection using workload WORK2. Also, the number of calls across different arrival rates (as seen in Figure 5.28) remains the same or increases for higher rates. As the path lengths increase, the number of designs that fail to support the workload increases for the higher arrival rates, and this causes the number of calls for the higher rates to be lower than for the lower arrival rates (e.g., at instruction path length factor of five in Figure 5.28).

We also notice that as the communication latency increases for a given set of arrival rates for the queries in a workload, then ART increases, but also the number of optimizer calls decreases. For the higher latencies, the higher communication costs result in more designs that fail to support the workload (as seen for workload WORK2 in Figure 5.15). For higher arrival rates, we also notice that PARING requires more optimizer calls.

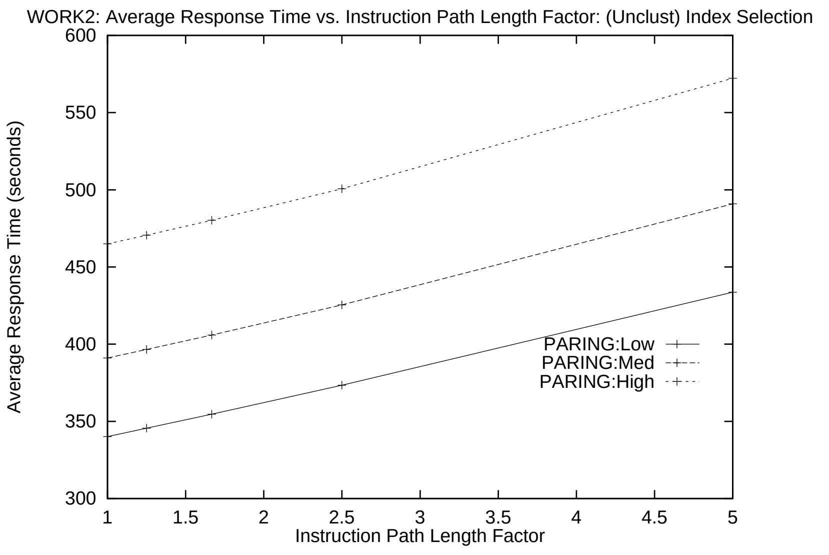
When varying the disk access times, the number of optimizer calls increase when the disk access times increase (e.g., see Figure 5.29 for index selection using workload WORK2). The workload moves from being CPU-bound to I/O-bound as the disk access times increase.

5.2.5 Varying the Bottleneck Utilization

In this set of experiments, we varied the arrival rates of a workload to change the bottleneck utilization for the workload using the design given by IR. This allows us to study how the load on a system affects PARING's execution. We assumed that the arrival rates across the queries were the same (except for the second query class in WORK4, which has twice the rate as the other queries). For a given utilization, we executed PARING using the corresponding arrival rates, and recorded the results. The increase in the bottleneck utilization (and thus arrival rates) causes an increase in the number of optimizer calls. For example, we see this in Figure 5.30 for partitioning key selection using workload WORK4. Similar results were seen for the other workloads when selecting either partitioning keys or indexes. As was stated previously, the reason for this increase is because the increase in arrival rates causes the PART values for a subset of queries to have even lower values compared to the ART values for the workload. Thus, there are fewer partial solutions that are pruned because the current best known ART is less than $r\%$ more than the ART of the partial solution.

5.2.6 Varying Update Rates

To show how our index selection adapts to updates, we have introduced updates to each of the relations in each of the workloads. These updates are specified in Appendices D and E. The



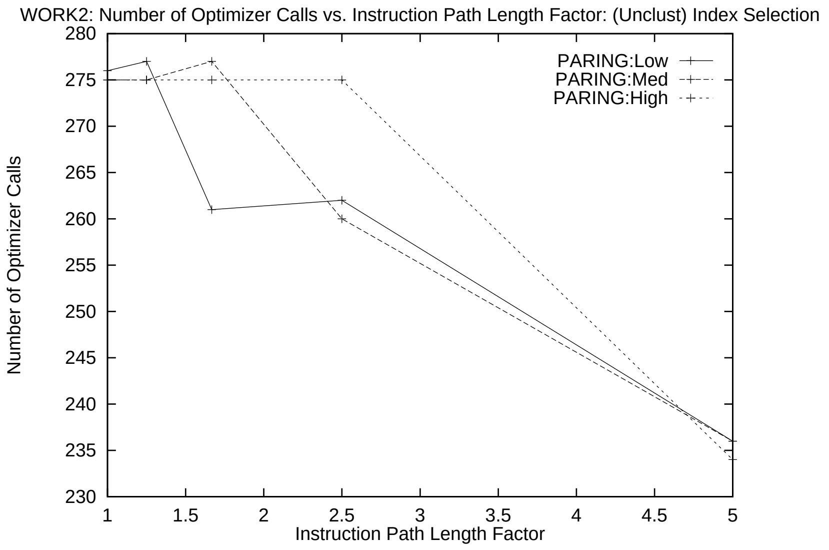
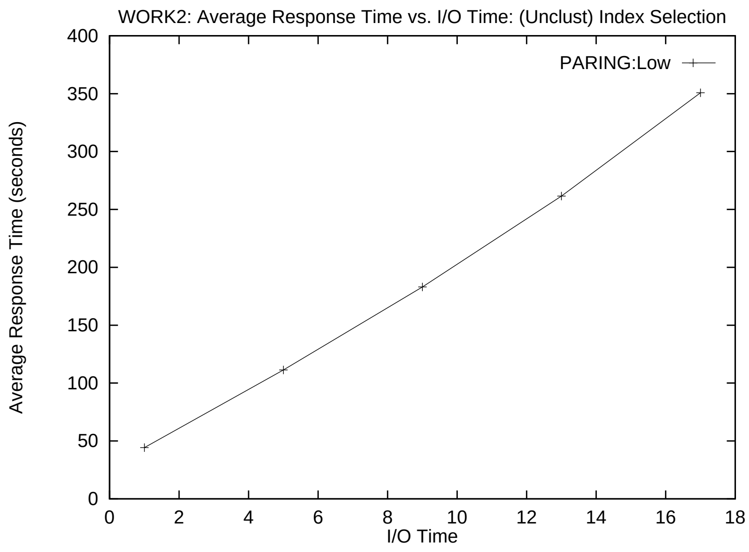


Figure 5.28: Average response time and number of optimizer calls for PARING versus the factor multiplying the initial instruction path length when selecting indexing keys for workload WORK2.



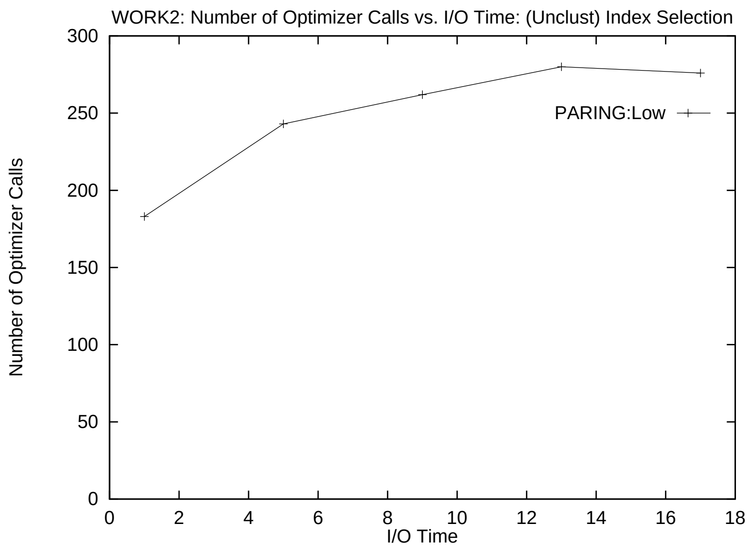
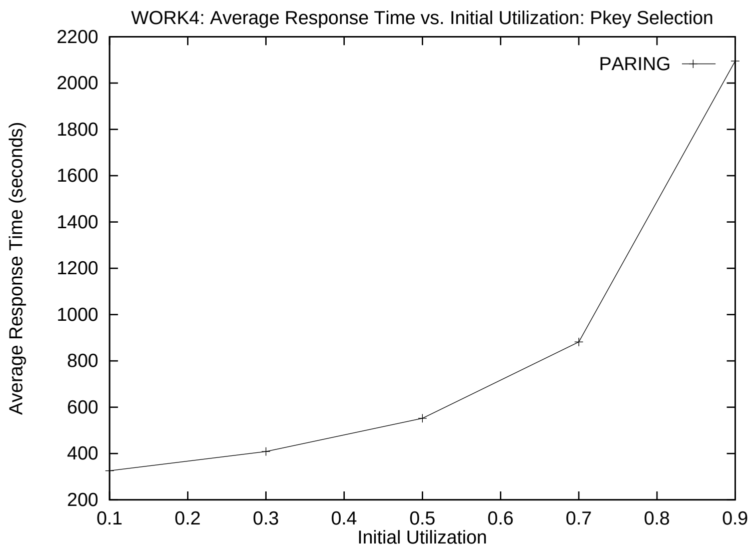


Figure 5.29: Average response time and number of optimizer calls for PARING versus the disk access times (in milliseconds) when selecting indexing keys for workload WORK2.



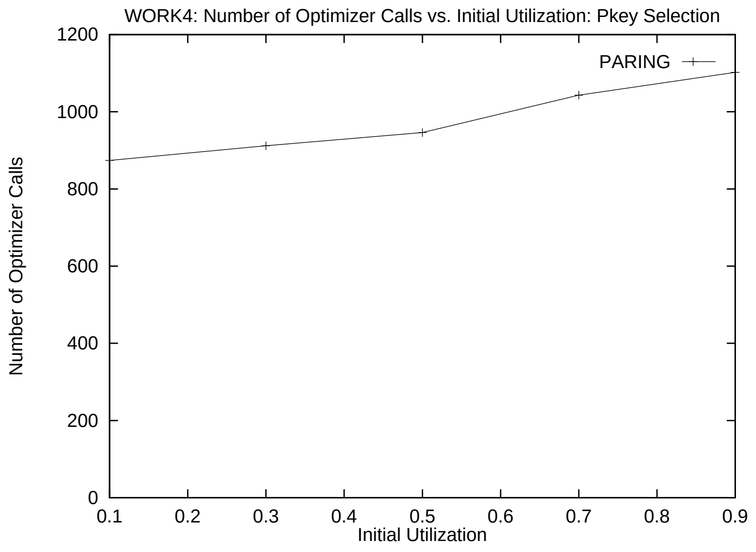


Figure 5.30: Average response time and number of optimizer calls for PARING versus the bottleneck utilization for IR when selecting partitioning keys for workload WORK4.

arrival rate for each of the update classes is calculated using a factor times the arrival rates of the queries in the workloads, where each query class has the same arrival rate as the other query classes. This factor is called the *update rate*, and is used as the x-axis in the graphs for this section.

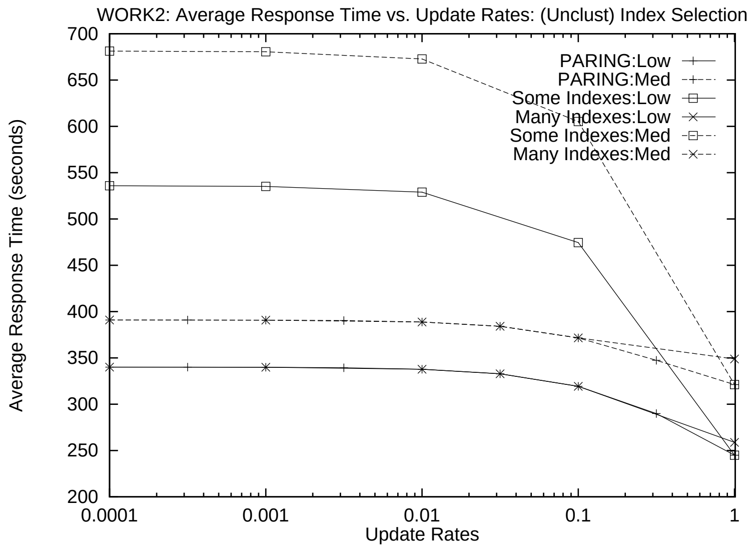
We see that the average response time of the workload including the updates decreases when the update rates increase. For example, we see this decrease when choosing indexes for workload WORK2 in Figure 5.31. This decrease exists because the average response times are calculated as an average of the response times of the queries and updates that are all weighted by their arrival rates. When the update rates increase, the average moves towards the lower response times of the updates.

When there are more updates, the indexes chosen by PARING are different and fewer because of the penalty involved in updating the indexes. For WORK2 at an update rate of 1, the indexes selected by the PARING algorithm changes. For the update rate of 1, it selects the indexes shown in Table 5.4. The indexes chosen for lower updates rates are also shown in this table. In this case, the update costs do affect the index selection. To illustrate this point further, in Figure 5.31, we included the ART curves for the index sets shown in Table 5.4, where the index set for the update rate of 1 is labeled by *some indexes*, and the other index set is labeled as *many indexes*. There is a cross-over point when the index set found at an updates rate of 1 gives an improved ART result than for the other index set. We also determined the ART values for the empty index sets, and found the ART curve had a similar downward trend. However, the ART values using the design with no indexes were always greater than the ART values resulting for the other index sets. These values are higher than the ones shown in Figure 5.31.

Update Rates	**Indexes Chosen**
≥ 0.0001 and < 1	A1 of relation R2 (clustering)
	A1 of relation R12 (nonclustering)
≥ 1	A1 of relation R1 (clustering)
	A1 of relation R4 (clustering)
	A1 of relation R5 (clustering)
	A1 of relation R12 (clustering)
	A2 of relation R14 (clustering)

Table 5.4: Indexing key sets chosen by PARING for the corresponding update rates when selecting indexing keys for workload WORK2.

The number of optimizer calls for the algorithms at higher update rates decreases. For example, Figure 5.31 shows such a decrease for workload WORK2. The number of calls for PARING decreases for the update rate of 1. This result is similar to previous cases when the



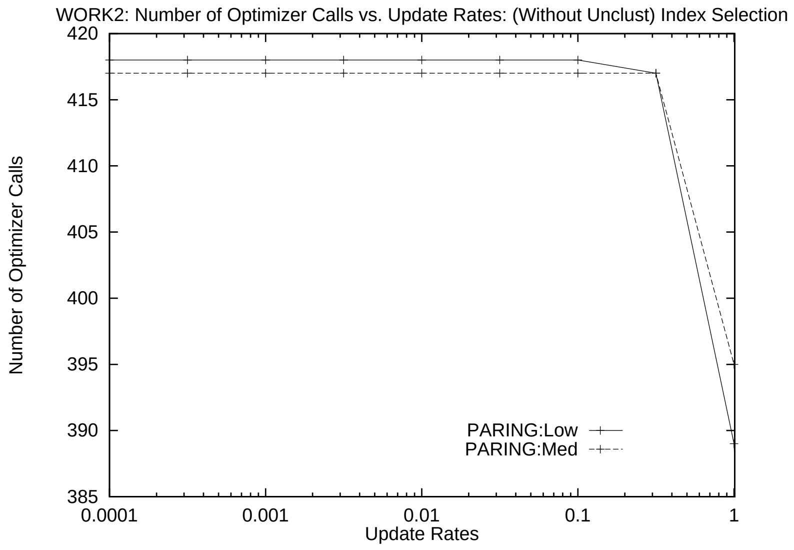


Figure 5.31: Average response time and the number of optimizer calls for PARING versus the update rates when selecting indexing keys for workload WORK2.

load on the system is heavy. More cases are pruned away because of the increase in the number of designs that fail to support the workload.

5.2.7 Comparing Index Selection when Avoiding or not Avoiding Nonclustered Indexes

In our clustering key and index selection algorithms, the pre-scanning phase prunes the nonclustering indexing keys that lead to higher response times than the rest of the keys for its relation. To demonstrate how well PARING performs using this pruning, we show experiments for index selection when nonclustering indexing keys are or are not pruned. We used the same experimental environment as in Section 5.2.6, where the update rates vary.

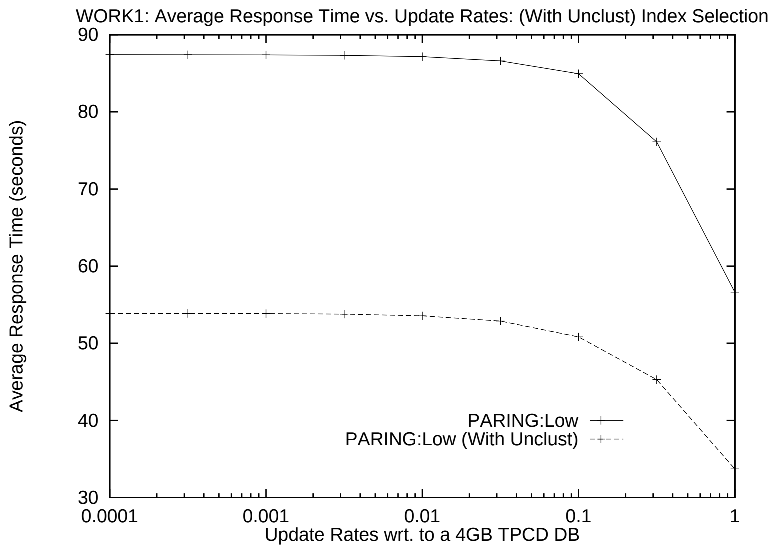
The average response time resulting from the design selected by PARING is better when the nonclustering keys are not all pruned away, as seen in Figure 5.32. This shows that the pruning eliminates the best design found when this rule is not applied. However, by leaving the nonclustering indexes in the selection, the number of optimizer calls is substantially higher than with the pruning as seen in Figure 5.32. In fact, the ART values are roughly halved by using ten times more effort in the selection of the design when nonclustering indexing keys are not eliminated by pre-scanning. However, if some guarantee is required that the resulting ART be within a certain distance from the best ART (found by not using the rule), then the nonclustering indexes cannot be ignored. To lower the number of optimizer calls, we could raise the ratio r to be greater than zero, which still provides us with some guarantees on the distance of the ART from PARING relative to the ART of the best design.

5.2.8 Selecting Partitioning, Clustering, and Indexing Keys Together

We now show how the PARING algorithm behaves when partitioning, clustering, and indexing keys are to be selected together. We used three possible ways of determining the keys:

- in two separate PARING executions, selecting partitioning keys first and then clustering and indexing keys;
- in two separate PARING executions, selecting clustering and indexing keys first and then partitioning keys; or
- selecting partitioning, clustering, and indexing keys in a single PARING execution.

When partitioning keys are selected first, the partitioning in the chosen design from PARING constrains the next execution of PARING that finds the clustering and indexing keys only. When indexing and clustering is determined first, the indexing and clustering chosen by PARING constrain the next PARING execution that chooses only the partitioning.



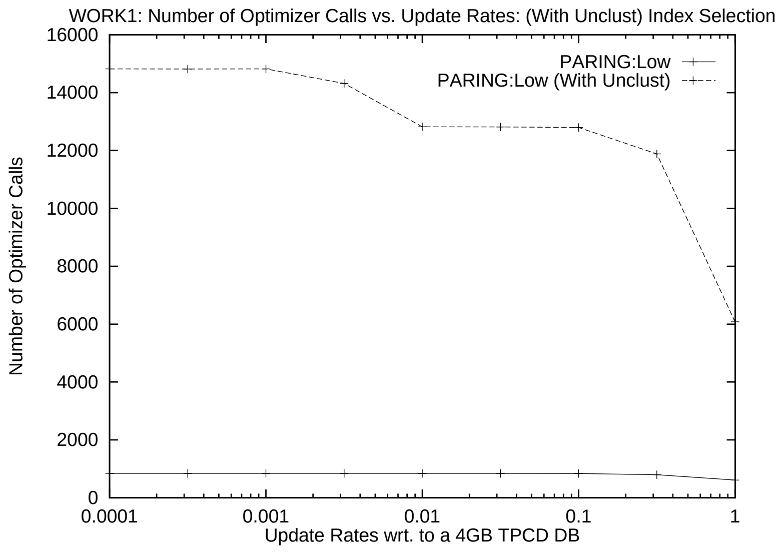


Figure 5.32: Average response time and number of optimizer calls for PARING versus the update rates when selecting indexing keys for workload WORK1 avoiding or not avoiding the nonclustering indexes.

Figure 5.33 shows an example of the resulting average response time for the bottleneck utilization experiment using the PARING algorithm for the three possible ways of selecting the partitioning, clustering, and indexing keys together. The selection of the keys separately gave response times that were close to those obtained by selecting all the keys together, especially when the indexes were chosen first. The number of optimizer calls is much lower when the keys are chosen separately (e.g., see Figure 5.33). We observed similar results for the response times and number of optimizer calls when using other workloads (e.g., WORK2's results are shown in Figure 5.34, WORK3's results are shown in Figure 5.35, and WORK4's results are shown in Figure 5.36). Based on these results, we suggest that the best method for selecting the keys is to pick the indexes (and clustering keys) first and the partitioning keys second.

5.2.9 Validating Our Cost Estimates

The goal of this section is to demonstrate that, although our cost model does not provide accurate response times compared to measured times, it does provide times that are relatively similar to measured times. To demonstrate this fact, we used a workload, and compared the analytic and measured ART results for two designs that vary in their partitioning only:

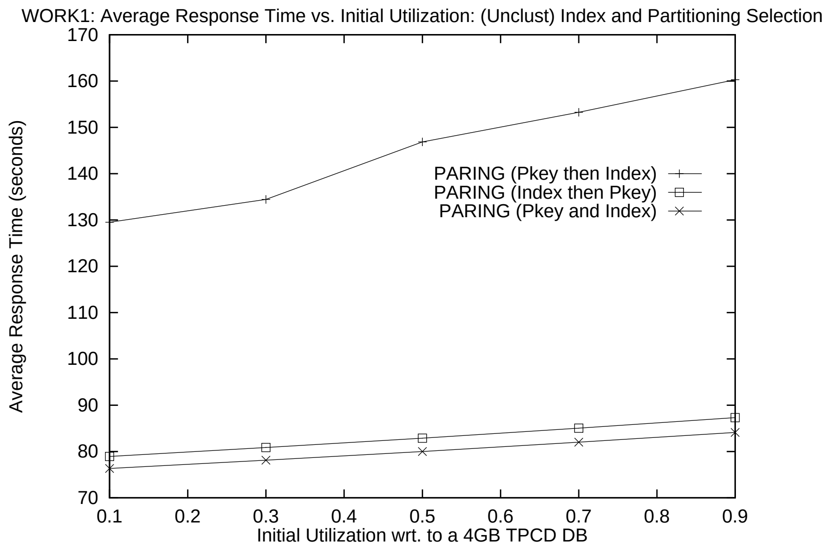
- The first design is chosen using a rule of thumb defined in Chapter 3. Namely, the partitioning keys were chosen as the keys that were most frequently used as join attributes and in GROUP BY clauses.

- The second design is the design chosen by our PARING algorithm.

The workload is given in Figure 5.37 in which the queries use the TPC-D DB defined in Appendix D. Each query class has the same arrival rate of 5.7 queries per hour. The partitioning keys for the designs are shown in Table 5.5. The database relation sizes are the ones used for a 1 GB TPC-D benchmark DB (based on a scale factor of one in the benchmark).

We used a sixteen node shared-nothing system running IBM DB2 PE. Each node was an IBM RS6000 with a PowerPC 133 MHz processor, 800 MB on hard disk, and 64 MB of main memory. The nodes were connected using a 100 Mbps Fast Ethernet. The system parameters used in the experiment are given in Table 2.1. Also, the instruction path lengths for all the operations are given in Table 2.2.

To generate the queries on IBM DB2 PE, we created a process for each class that would generate queries for the corresponding class. The queries are generated using a pseudo-random number generator for exponentially distributed numbers with the mean interarrival time based on the arrival rate for the class. Each process was executed simultaneously with the other processes on different nodes of the 16 node system. The average response time of each query class was obtained by averaging the execution times of 20 queries from that class. Table 5.6



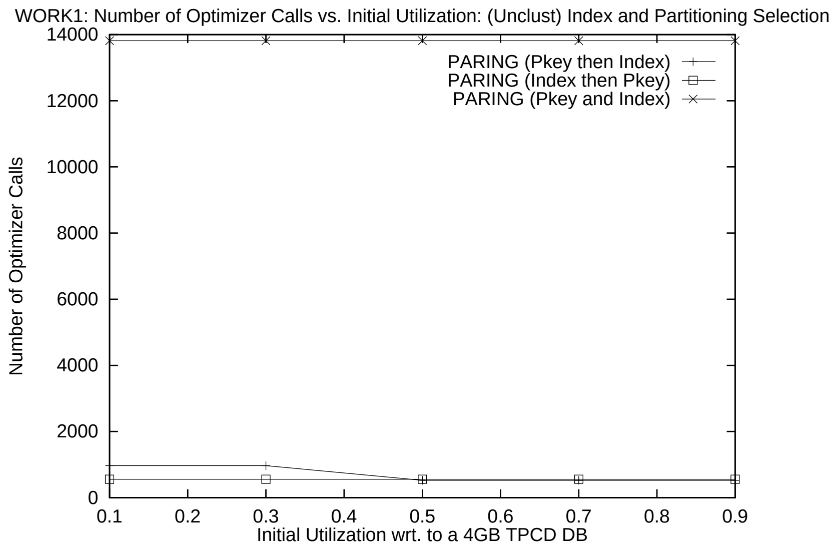
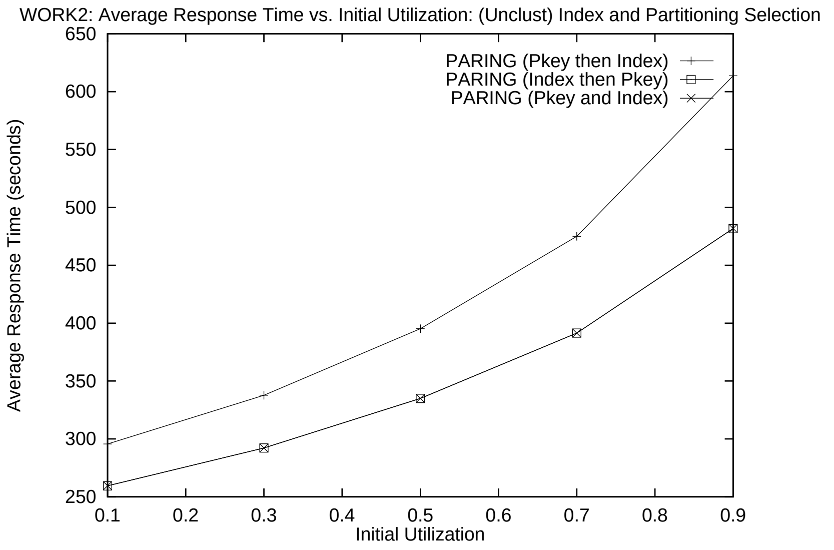


Figure 5.33: Average response time and number of optimizer calls for PARING versus the bottleneck utilization for IR when selecting all keys for workload WORK1.



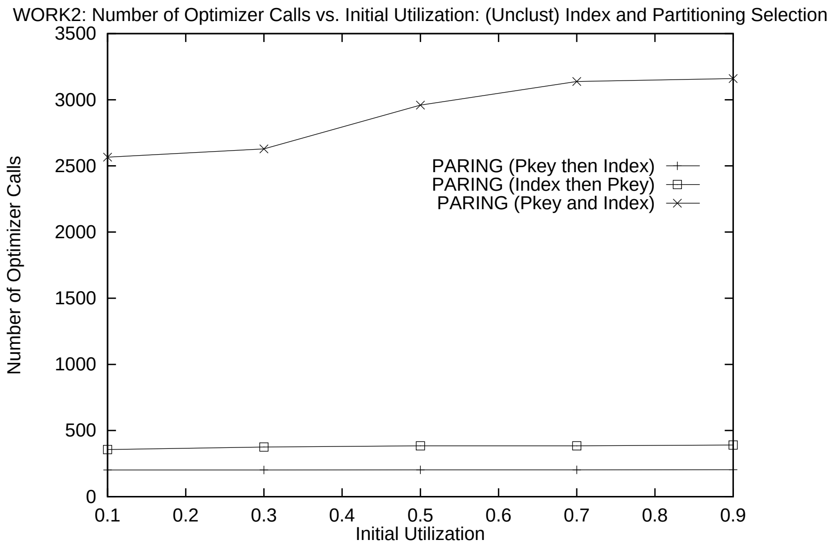
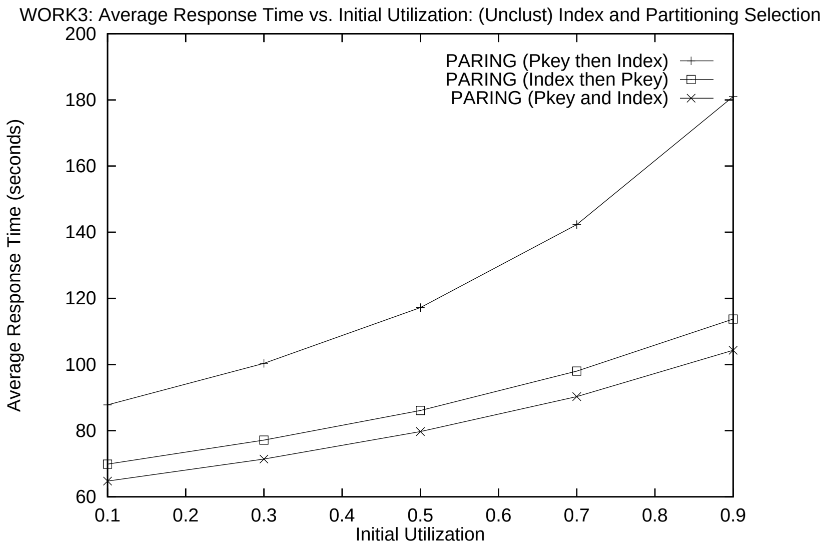


Figure 5.34: Average response time and number of optimizer calls for PARING versus the bottleneck utilization for IR when selecting all keys for workload WORK2.



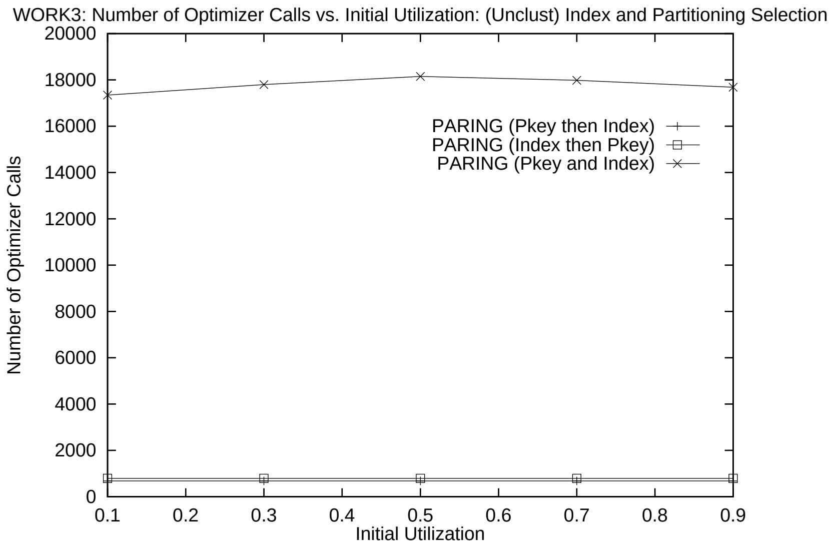
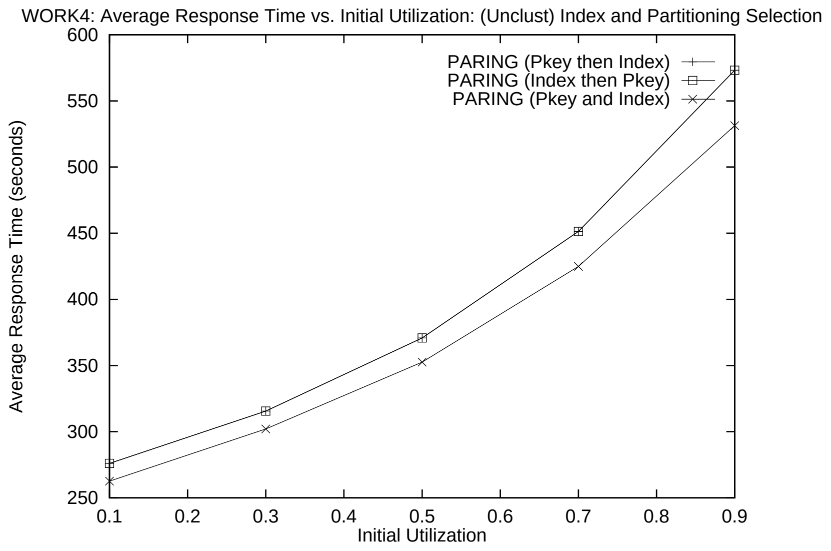


Figure 5.35: Average response time and number of optimizer calls for PARING versus the bottleneck utilization for IR when selecting all keys for workload WORK3.



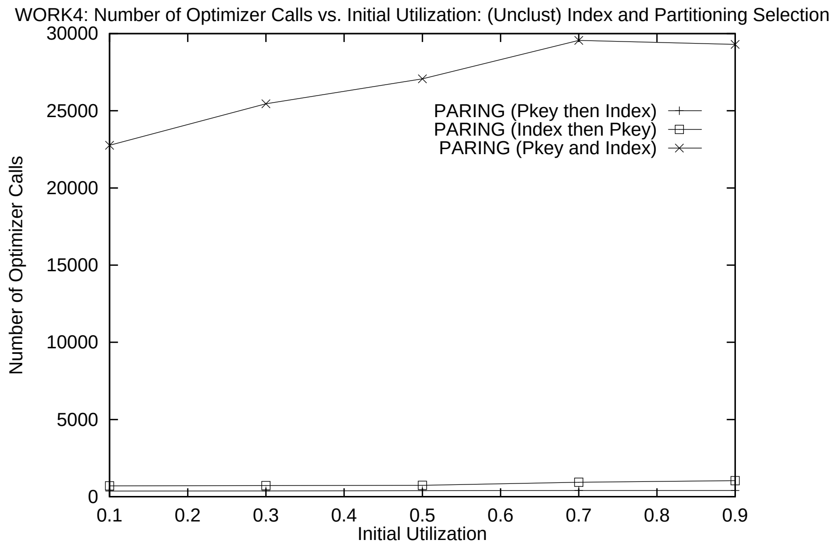


Figure 5.36: Average response time and number of optimizer calls for PARING versus the bottleneck utilization for IR when selecting all keys for workload WORK4.

```
SELECT N_NAME, DECIMAL(SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),31,3)
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND O_ORDERKEY = L_ORDERKEY AND
      L_SUPPKEY = S_SUPPKEY AND C_NATIONKEY = S_NATIONKEY AND
      S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
      R_NAME = <hv1> AND O_ORDERDATE >= DATE(<hv2>) AND
      O_ORDERDATE < DATE(<hv2>) + 1 YEAR
GROUP BY N_NAME
ORDER BY 2 DESC
```

Query (a)

```
SELECT YEAR(O_ORDERDATE), L_EXTENDEDPRICE*(1-L_DISCOUNT), N2.N_NAME, R_NAME
FROM PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION N1, NATION N2, REGION
WHERE P_PARTKEY = L_PARTKEY AND S_SUPPKEY = L_SUPPKEY AND
      L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
      C_NATIONKEY = N1.N_NATIONKEY AND N1.N_REGIONKEY = R_REGIONKEY AND
      R_NAME = <hv1> AND S_NATIONKEY = N2.N_NATIONKEY AND
      O_ORDERDATE BETWEEN DATE(<hv2>) AND DATE(<hv3>) AND
      P_TYPE = <hv4>
```

Query (b)

```
SELECT L_SUPPKEY, MAX(DECIMAL(L_EXTENDEDPRICE*(1-L_DISCOUNT),31,3))
FROM LINEITEM
WHERE L_TAX > <hv1> and L_TAX < <hv2>
GROUP BY L_SUPPKEY
ORDER BY L_SUPPKEY
```

Query (c)

```
SELECT O_ORDERDATE, MAX(O_SHIPPRIORITY)
FROM CUSTOMER, ORDERS
WHERE C_MKTSEGMENT = <hv1> AND C_CUSTKEY = O_CUSTKEY AND
      O_ORDERDATE < DATE(<hv2>)
GROUP BY O_ORDERDATE
ORDER BY O_ORDERDATE
```

Query (d)

Figure 5.37: Workload for validation, where $< hvi >$ represents a variable instantiated per query execution.

Relation	**First Design's Partitioning Keys (from Rules of Thumb)**	**Second Design's Partitioning Keys (from PARING)**
LINEITEM	L_SUPPKEY	L_ORDERKEY
ORDERS	O_CUSTKEY	O_CUSTKEY
PARTSUPP	PS_PARTKEY	PS_PARTKEY
PART	P_PARTKEY	P_PARTKEY
SUPPLIER	S_SUPPKEY	S_SUPPKEY
CUSTOMER	C_CUSTKEY	C_CUSTKEY
NATION	N_NATIONKEY	N_NATIONKEY
REGION	R_REGIONKEY	R_REGIONKEY

Table 5.5: Partitioning keys for the designs using either the rules of thumb or PARING.

shows the individual query class response times as well as the analytic and measured ART times. In this table, we also show the percent reduction between the measured (and also the estimated) response times from using the first design to the second. If $R_{first,j}$ is the measured (or estimated) response time using the first design and $R_{second,j}$ is the measured (or estimated) response time using the second design for query class j, then the percent reduction is calculated as:

$$\frac{R_{first,j} - R_{second,j}}{R_{first,j}}$$

The percent reduction of the measured (or estimated) ART results is calculated similarly as:

$$\frac{ART_{first} - ART_{second}}{ART_{first}}$$

We used a timing routine on DB2 PE in which the optimization of each query from a class is done on the fly in parallel with other query executions. The time to optimize a query is not counted as part of its execution time by this routine. However, the delays incurred by optimizing a query while other queries are executed are implicitly included in the timings of the executing queries, and this accounts for some of the differences between the analytic and measured times. Also, IBM DB2 PE has many overheads that we did not account for in our cost estimator. These include such tasks as buffer management, concurrency control, and logging. The omission of these overheads and optimization costs in our estimates, which stay relatively the same across the measured results for the different designs, accounts for our optimistic estimated values compared to the measured times. Table 5.6 demonstrates that the percent reductions between the estimated timings from the two designs are higher relative to the

Query	**First Design**		**Second Design**			
	Measured Response Times	**Estimated Response Times**	**Measured Response Times**	**Estimated Response Times**	**Measured Percent Reduction**	**Estimated Percent Reduction**
(a)	602	273	514	199	0.15	0.27
(b)	654	562	394	223	0.40	0.6
(c)	256	129	215	95	0.16	0.26
(d)	166	27	139	19	0.16	0.26
ART	420	248	316	134	0.25	0.46

Table 5.6: ART and percent reduction of workload under the first and second physical DB designs. All times given in seconds.

percent reductions between the measured timings for the two designs. However, the differences between the measured and estimated percent reductions are similar.

5.2.10 Summarizing the Results from PARING

Throughout this section, there were a number of results we found to be interesting with regards to our PARING algorithm's execution.

1. Partial designs that fail to support the workload assist in reducing the number of optimizer calls for PARING.

2. Pruning the nonclustering indexing keys in the pre-scan phase may cause the best design (found when the keys are not pruned) to be missed, but it does reduce the number of optimizer calls as well. However, there is no guarantee on how close the ART values are to the best ART values when nonclustering indexing keys are eliminated in pre-scanning. When some guarantee on the ART result is required, it is warranted to not prune with this rule and use higher values for r to lower the number of optimizer calls.

3. Selecting the partitioning keys separately from the indexing and clustering keys reduces the number of optimizer calls compared to selecting all design components together in the same execution of PARING. The ART's from the designs found by the separate determination (indexing and clustering chosen before partitioning) were found to be close and sometimes equal to the ART's from the designs of the single algorithm execution in the experiments we carried out.

4. The number of optimizer calls for PARING is affected by the load of the system from the workload. When the load is low (e.g., for low arrival rates), the number of calls is low. The number of calls is higher for higher loads. Finally for even higher loads, the number of optimizer calls is low again. This trend was seen when any of the DB sizes, service times, update rates, or arrival rates were varied.

5. As the PARING ratio (r) increases, the number of optimizer calls decreases. However, increasing the ratio to be greater than zero sometimes leads to PARING finding solutions with ART values that are further from the best ART (found using $r = 0$), although never more than $r\%$ worse than the best ART.

6. Using a ratio (r) of zero, PARING requires the lowest number of optimizer calls when the initial design is determined by the IR algorithm as opposed to by another execution of the PARING algorithm with the ratios $r > 0$ that we tested, i.e., $r \geq 0.2$.

7. High update arrival rates compared to the queries' arrival rates result in different and fewer indexes being selected relative to indexes chosen when using lower update arrival rates.

Chapter 6

Reorganizing a Physical DB Design

Once a physical DB design has been chosen, the DB is loaded, and the DB is then used by the workload. During this time, the workload could change or updates in the workload could cause the number of tuples on a subset of the nodes to be greater than on the other nodes. In such cases, the physical DB design should be reorganized. If these changes happen periodically, the database administrator (DBA) should execute an automatic decision algorithm periodically to determine if a new physical DB design is needed and if the reorganization to reach this design is not overly costly. We address the reorganization problem by defining the significant aspects that are involved in all reorganization decisions and by studying the performance of reorganizations that move tuples between nodes in a parallel DB system. The aspects of the physical DB reorganization problem include:

- the metric to compare reorganizations;
- a method to estimate the response times for a workload and a reorganization strategy that execute together; and
- the priority of a reorganization strategy compared to the workload.

The reorganizations investigated in this chapter assume that there are more tuples of the DB stored at one or more nodes than on the other nodes of a parallel DB system resulting in *data imbalance* or *data skew*. Moving the extra tuples from the nodes with more tuples to the nodes with fewer will balance the data and load across the nodes. This type of reorganization includes the case where new nodes initially containing no data are added to the parallel DB system. To determine the proper reorganization to use, we assume that the number of tuples of each relation on each node is known. The system environment (workload, DB, and system characteristics) are all used as inputs to determine the priority for a specific reorganization and the best reorganization to execute for the given workload.

In reality, data imbalance (or data skew) could be a result of any of several causes:

- inserts, deletes, and updates of tuples in an imbalanced way across the nodes;

- workload changes resulting in the access of tuples with specific attribute values to be more predominant on one of the nodes than the others; and

- the addition of new nodes, which initially would contain no tuples.

In studying our specific rebalancing strategies, we capture the characteristics of most rebalancing strategies resulting from these causes. We also capture characteristics (such as benefit versus cost) for other types of reorganizations.

Our approach is new and important because:

- it allows for comparisons between different reorganization strategies possibly yielding different physical DB designs;

- it accounts for the interaction between the workload and a reorganization strategy; and

- some aspects, such as the comparison metric, can be applied to any type of DB system, not just a SN system.

We consider both concurrent (on-line) reorganizations of different priorities and *off-line* reorganizations. An off-line reorganization is one that prevents the workload from using the DB while it executes. We included off-line reorganizations to validate our estimates with reorganizations from a real DB system (IBM DB2 PE) and to compare response times of a concurrent reorganization to those when no concurrency is allowed. The cost of a concurrent reorganization is calculated using the delay it imposes on the workload. For an off-line reorganization, the penalty or cost of executing it is derived by assuming all queries during an off-line reorganization must be delayed by the time it takes to execute the off-line reorganization. We discuss the implications of estimating an off-line reorganization's costs and benefits in Section 6.2.

We have structured the rest of this chapter as follows. In Section 6.1, we first present some general structures for reorganization decision algorithms. These algorithms determine the new physical DB design to move to along with the corresponding reorganization to use. One purpose of this section is to present the components required to make a reorganization decision, such as (1) the cost estimation of a workload and reorganization executing together, (2) a metric for integrating the costs and benefits from a reorganization, (3) a method to choose a new physical DB design, and (4) a method to choose a reorganization strategy. We next describe the reorganization and workload cost estimation in Section 6.2. This estimation is analytical, and is based on a queuing network model (QNM) approach. In Section 6.3, we describe our performance metric used to compare reorganization strategies and their effects on the workload. We then study the priority of a reorganization by experimenting with some reorganization strategies in Section 6.4. Finally, in Section 6.5, reorganization algorithms are presented along with some results from executing these algorithms.

6.1 Possible Reorganization Algorithm Structures

The purpose of this section is to show two possible reorganization algorithm structures. They suggest how to adapt or include the physical DB design decision algorithms of Chapter 4 in a reorganization algorithm.

To determine how to best adjust to the changes in a system environment, a general dynamic physical DB design decision is required. This decision can be stated as an optimization problem, as seen in Equation 6.1. The goal of the optimization is to maximize the benefit after a reorganization relative to the cost of the reorganization (defined by our *(net) gain metric* in Section 6.3). Since we require a reorganization to have a benefit greater than its cost, the objective value should be greater than zero for the reorganization we choose. To eliminate other possible reorganizations, we must ensure the performance resulting from the new design (ART_{new}) is better than that from the old (ART_{old}), otherwise the reorganization would not improve the physical DB design. However, even when this constraint holds, the cost of performing the reorganization with the workload may exceed the performance improvement of the workload after the reorganization, which implies the reorganization should not be chosen. We also require that the reorganization completes with response time (R) within a time interval (T). This time interval constrains the amount of time in the future over which we allow the costs and benefits to be calculated. This constraint is required since new changes to the system environment may occur, causing our estimates of the future performance to be in error. Time intervals are further discussed in Section 6.3. Other DB system constraints could also be added, such as a constraint that the storage required at a node cannot exceed the node's storage space. Another constraint would be that the new DB design cannot fail to support the workload (i.e., the workload cannot saturate the system).

$$\begin{aligned}
\max \quad & Benefit - Cost\ for\ a\ reorganization \\
s.t. \quad & ART_{old} < ART_{new} \\
& Benefit - Cost > 0 \\
& R < T
\end{aligned} \tag{6.1}$$

This optimization problem can be solved using a search algorithm. A search algorithm compares different reorganization strategies in order to find the best on-line (or even off-line) strategy. We will describe the general structure for two such search algorithms.

The first reorganization algorithm is given in Figure 6.1. This algorithm's structure is a straightforward extension from the physical DB design algorithms from Chapter 4. In this algorithm, a new (*ideal*) physical DB design is determined using the changed system environment, which includes the changed system, DB, and workload information. Physical DB design algorithms from Chapter 4 can be used directly for this purpose. To improve on the performance of the DB design decision, we could use PARING with a large value for its ratio r ($r \leq 0.5$). The new DB design is then used as input into a *reorganization selector*. This selector finds a reorganization strategy (along with its priority level) to move from the current physical DB design to the ideal design for which the cost of reorganization can be more than justified by the improved performance once the reorganization is done. A heuristic could be developed to execute this selector, where the workload and reorganization costs are evaluated for a reorganization strategy by using a *reorganization cost estimator*. Other inputs to the selector are the old design and the time interval over which to measure the gain metric. If initially the workload performance (ART) resulting with the new DB design is not as good as the performance from the old design, then no reorganization is chosen, and the selector will terminate. This workload ART when the workload executes without a reorganization can be estimated with the reorganization cost estimator by ignoring the reorganization strategy in the estimator. If the selector cannot find a strategy that yields a gain metric value greater than zero, then no strategy is output from the selector.

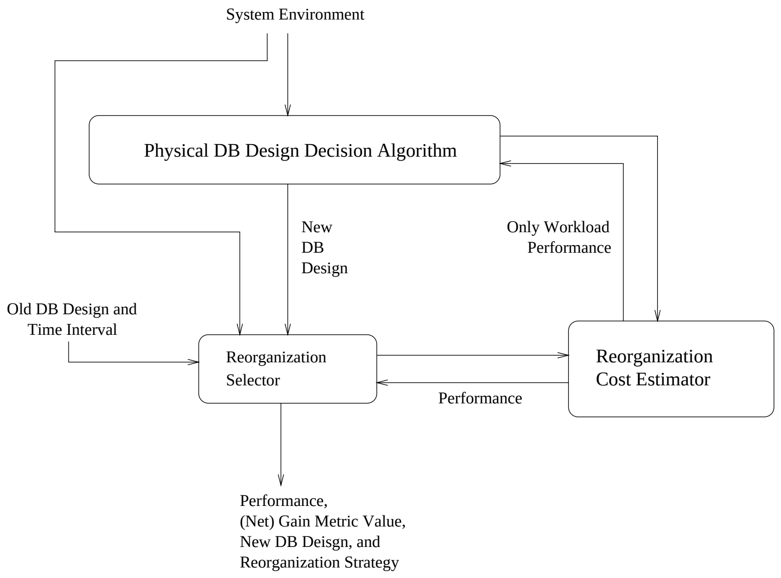
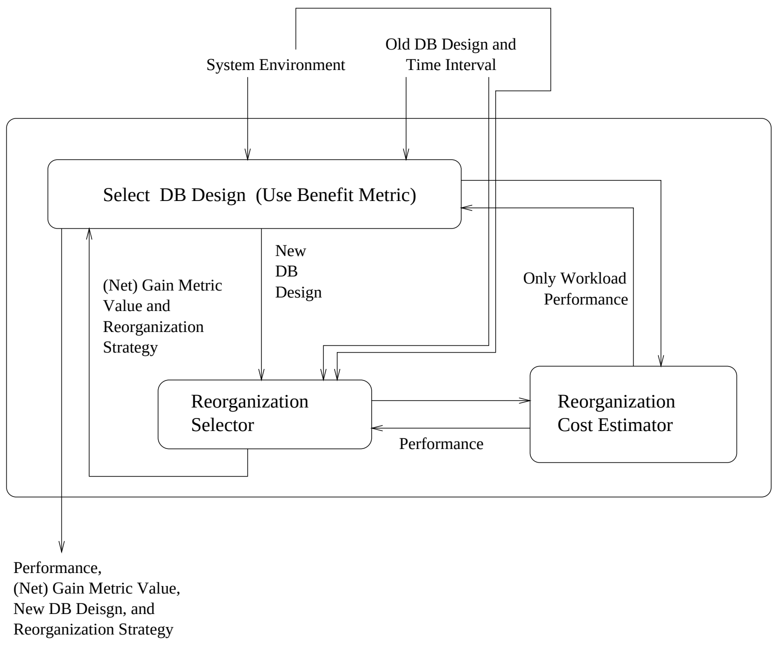

System Environment
Physical DB Design Decision Algorithm
New DB Design
Only Workload Performance
Old DB Design and Time Interval
Reorganization Selector
Reorganization Cost Estimator
Performance
Performance, (Net) Gain Metric Value, New DB Deisgn, and Reorganization Strategy

Figure 6.1: Example reorganization decision algorithm (Algorithm A).

Another reorganization algorithm is given in Figure 6.2. This algorithm involves iterating between the DB design selector and the reorganization selector. The main difference between this algorithm and the algorithm in Figure 6.1 is that there is more than one possible new DB design used in the algorithm of Figure 6.2, and thus the gain metric is more crucial since the performance (and thus benefit) after a reorganization will vary depending on the new DB design chosen. Another difference is that the module selecting the DB design in Figure 6.2 only selects one design at a time to pass to the reorganization selector and it bases its decision on the gain metric passed back from the selector module. Also, a DB design is not passed to the selector when the workload performance (ART) of the new design is not better than the performance resulting from the old design. In reality, the selection module calls the reorganization selector as a subroutine to determine a reorganization strategy (if any is possible with gain greater than zero) and its corresponding performance result (the gain metric value).



System Environment
Old DB Design and Time Interval
Select DB Design (Use Benefit Metric)
New DB Design
(Net) Gain Metric Value and Reorganization Strategy
Only Workload Performance
Reorganization Selector
Reorganization Cost Estimator
Performance
Performance, (Net) Gain Metric Value, New DB Deisgn, and Reorganization Strategy

Figure 6.2: Example reorganization decision algorithm (Algorithm B).

One possible reorganization algorithm that fits into the structure defined in Figure 6.2 is an algorithm that uses information from the old design and the new system environment to determine where the problems are in the system and how to fix them. The algorithm can continue until either it finds some reorganization strategy and new design with a gain metric value greater than zero or until it finds the strategy and design leading to the highest gain metric value.

The types of reorganizations that could be included in both of the previous algorithms are:

- reorganizations that move data (and load) from the bottleneck disks to the least utilized disks, where these least utilized disks could include ones that have just been added to the system;
- ones that create or delete indexes;
- ones that change the partitioning of one or more relations including their partitioning keys, relation groups, and data placements; and
- reorganizations that modify how a relation is clustered.

6.2 Reorganization and Workload Cost Estimation

To estimate the workload and reorganization response times, we use a method based on queuing network model (QNM) estimations. QNM terminology used in this chapter is defined in Appendix A, and the variables are defined in Tables 4.1 and 6.1. We require query plans as input so that we can determine the resource requirements and average response times of a query class.

Variable	**Description**
s	Respresents a query or reorganization class
X_s	Avg. throughput of the query or reorganization class s
N	Number of customers
$A^{(s)}_{r,k}(N)$	Avg. number of customers of class r at resource k when a class s arrives
$\lambda^{(s)}_{r,k}(N)$	Avg. number of customers of class r at resource k when a class s is in service
$H_{s,k}$	Classes that have higher priority than class s at resource k
$E_{s,k}$	Classes that have equal priority to class s at resource k
$Q_{s,k}$	Avg. queue length of class s for resource k
$Q_{s,k}(N)$	Avg. queue length of class s for resource k when N customer are in the system

Table 6.1: Queuing network model variables for the reorganization cost estimator.

We model each resource as an M/M/1 server [LZGS84]. The scheduling discipline at the resources were chosen in two ways. In the first way, only the processors use preemptive (PR) priority scheduling [LZGS84, EL88, BKL$^+$84], and the other resources (disks and interconnect network) use FCFS scheduling. In the figures, the label *CPU PR* denotes this case. In the second way, all resources use preemptive priority scheduling. The label *ALL PR* denotes this second case in the figures. The first approach is more realistic for most systems in common use today, but the second allows us to determine the implications of having priority-based scheduling at the other resources and to investigate the effects of more than just processor scheduling on a reorganization's performance. We expect the results from the second approach to show clearer differences between reorganizations at higher or lower priority compared to the workload processes.

In our model, the queries in a workload are each represented as a transaction class with an arrival rate in the QNM, and the reorganization is represented as a batch class with one customer. A single customer batch class provides us with a way to model a reorganization that executes once with the workload. Performance estimates from this QNM will provide us with estimates of average performance measures of the reorganization executing with the workload. For example, the estimated average response time of a reorganization would include the congestion delays from the workload, and a query's average response time would include the congestion delays from the reorganization as well as from other queries. Note that reorganization intrusion on the workload is modeled by the inclusion of these congestion delays.

This QNM model can be used for a number of purposes. As was stated, the model can be used to estimate the costs related to a reorganization executing with a workload using different priorities. These costs during a reorganization are calculated using the DB design before the reorganization begins since we assume the reorganization does not make its new design available until it is complete. The model also allows us to estimate the performance of the workload before and after the reorganization (using the old and new designs, respectively) by excluding the reorganization class from the model.

We use an approximate mean value analysis (aMVA) method (as seen in Figure 6.3) to estimate costs with priorities and mixed-class workloads [EL88, BKL$^+$84]. The main formula representing the preemptive (PR) priority scheduling at a resource is given by:

$$R_{s,k}(N) = \frac{V_{s,k}[S_{s,k} + \sum_{r \in H_{s,k} \cup E_{s,k}} S_{r,k} A^{(s)}_{r,k}(N)]}{1 - \sum_{r \in H_{s,k}} S_{r,k} \lambda^{(s)}_{r,k}(N)}$$

where $R_{s,k}(N)$ is the residence time of class s at resource k when N customers are in the system, $A^{(s)}_{r,k}(N)$ is the number of customers at resource k for a class r when a customer of class s arrives, and $\lambda^{(s)}_{r,k}(N)$ is the number of customers of class r that arrive while a customer of class s is in service. Note that $A^{(s)}_{r,k}(N)$ and $\lambda^{(s)}_{r,k}(N)$ are calculated using performance estimates from the other classes and the priority levels between classes. The formula basically states that the residence time for a customer of class s is equal to its service time plus the added waiting times due to higher and equal priority customers already in the server's queue ($A^{(s)}_{r,k}(N)$) and high priority customers arriving while this customer is queued ($\lambda^{(s)}_{r,k}(N)$). Estimates for $A^{(s)}_{r,k}(N)$ and $\lambda^{(s)}_{r,k}(N)$ were calculated using the BKT approximation method [EL88, BKT83]. In this method, $A^{(s)}_{r,k}(N) = Q_{r,k}(N - e_s)$ in which e_s is the unit vector with one at the s-th position and $Q_{r,k}(N - e_s)$ is the queue length at server k of customer type r. Also, $\lambda^{(s)}_{r,k}(N) = X_{r,k}(N)$ in which $X_{r,k}(N)$ is the throughput at resource k for customer class r. The formula for queue length is simplified because there is only one customer in the batch class and removing it yields the steady state queue lengths for the transaction classes when no other classes are in the system. We can use this formula for residence times to incorporate both of our modeling methods. When only processors are scheduled using PR priority scheduling, FCFS scheduling at the disks and interconnect network can be included in the estimation by having each $H_{s,k}$ at the resources being empty, i.e., the reorganization and workload processes are all scheduled with equal priority.

Using this residence time estimate has implications at the resources using PR priority scheduling when the reorganization process is of lower priority than the processes executing the workload classes. The residence times of the reorganization would include the delays caused by the higher priority query classes. However, since the appropriate residence time formulas for the query classes would exclude the lower priority reorganization, these query classes' residence times would be the same as if the reorganization were not executing in the system. In the experiment sections that follow, we will see this reflected by the workload average response times during a reorganization being equivalent to the response times when no reorganization is executed (which are evaluated as the times taken before a reorganization executes) using the old DB design when all resources are scheduled with preemptive priority.

In a real DB system, for a highly intrusive high priority reorganization, the queries arriving during a reorganization would have their response times increased by at most the time it takes to complete the reorganization. However, in our cost estimations, if the high priority reorganization causes the utilization of the bottleneck resource to approach 100%, the queries could be delayed so much that the system becomes saturated. In this situation, our model estimates become overly pessimistic. We shall call the workload average response times where our model becomes completely unreliable as the *model-limit*.

We assume that the response times before, during, and after a reorganization can be estimated using steady-state response times as shown in Figure 6.4 by B, D, and A, respectively. In reality, when a reorganization begins executing on the system, it introduces its delays over time, and also when the reorganization completes, the queries backed-up in the system because of the reorganization's intrusion will themselves impact on the newly arriving queries by introducing more delays. This extra delay is reduced as the backed-up queries are serviced and removed from the system. An example of these delays are shown in Figure 6.4, where R is the reorganization's response time, R'' is the time at which the initial delay from the reorganization reaches the steady-state response time during the reorganization (D), and $R' - R$ represents the time it takes to remove the backed-up transactions. The gradual increase and dissipation of delay at the start and end, respectively, of a reorganization strategy's execution could be modeled with transient (rather than steady-state) queuing network models. Accurate estimates of the workload performance during these transient times would allow for a more realistic estimate of a strategy's intrusion on the workload, and thus would allow for a better estimate of the reorganization strategy's benefit. This inaccuracy problem is most severe in the case of off-line reorganization, and the problem has to be addressed before the off-line strategy can be

```
algorithm: Reorganization Cost Estimator

{ Step 1: Use the arrival rates and visit counts to determine utilizations }
for each class $s$ (transactions $1, \ldots, Q$ and batch class) and its operators $O_s$ do
    Calculate $V_{s,k}$ and $V_{s,k}^{O_s}$ for each resource $k$ (as in Appendix A)
    $X_s = \begin{cases} f_s & \text{if } s \text{ is a transaction class} \\ 1/R_s & \text{if } s \text{ is the batch class and } R_s \text{ is the initial response time} \end{cases}$
    $U_{s,k} = X_s V_{s,k} S_{s,k}$ for each resource $k$
end for
{ Step 2: Use the resource utilizations to determine saturation }
for each resource $k$ do
    $U_k = \sum_{s=1}^{Q} U_{s,k}$ (exclude batch class if priority is not higher than the transactions)
    if $U_k \geq 1$ then Return $ART = \infty$ { Saturated system }
end for
{ Step 3: Calculate response times including congestion delays by iterating}
for each transaction and batch class $s$ do
    { Reset queue lengths and batch class throughput }
    if $s$ is the batch class then $X_s = 0$
    for each resource $k$ do $Q_{s,k} = 0$
end for
loop
    $Stopflag = TRUE$
    for each class $s$ do
        { Calculate response time of the class's operators }
        for each operator $O_s$ in query plan for $s$ (using bottom-up order) do
            for each resource $k$ do
                { Calculate residence times including congestion delays }
                $R_{s,k}^{O_s}(N) = \frac{V_{s,k}[S_{s,k} + \sum_{r \in H_{s,k} \cup E_{s,k}} S_{r,k} A_{r,k}^{(s)}(N)]}{1 - \sum_{r \in H_{s,k}} S_{r,k} \lambda_{r,k}^{(s)}(N)}$
            end for
            { Calculate execution times of the operator alone }
            $T_{O_s} = R_{s,COMM}^{O_s} + \max_k(R_{s,IO_k}^{O_s} + R_{s,CPU_k}^{O_s})$
            { Calculate the response time of operator $O_s$ }
            $R_s^{O_s} = \begin{cases} T_{O_s} + \max_i(R_{\text{child } i \text{ of } O_s}) & \text{if the children of } O_s \text{ do not execute at the same nodes as each other,} \\ T_{O_s} + \sum R_{\text{children of } O_s} & \text{otherwise} \end{cases}$
        end for
        { Calculate response time for class $s$ (critical path of query plan) }
        $R_s = R_s^{O_s}$ where $O_s$ is the root of the query plan
        if $s$ is the batch class then $X_s = 1/R_s$
        { Check if we should continue to loop, i.e., the queue lengths are not yet converging }
        if ( $|Q_{s,k} - X_s R_{s,k}| > \epsilon$ for any resource $k$ ) then $Stopflag = FALSE$
        for each resource $k$ do $Q_{s,k} = X_s R_{s,k}$
    end for
    if $Stopflag = TRUE$ then *Exit the loop*
end loop
{ Calculate the average workload response time for transaction classes $s$ }
Return $ART = \frac{\sum_{s=1}^{Q} f_s R_s}{\sum_{s=1}^{Q} f_s}$
```

Figure 6.3: Algorithm to estimate reorganization and workload average response times.

compared with the on-line strategies. Because of the limitations on off-line reorganizations, we do not compare off-line with on-line reorganizations except in the time it takes to complete a reorganization.

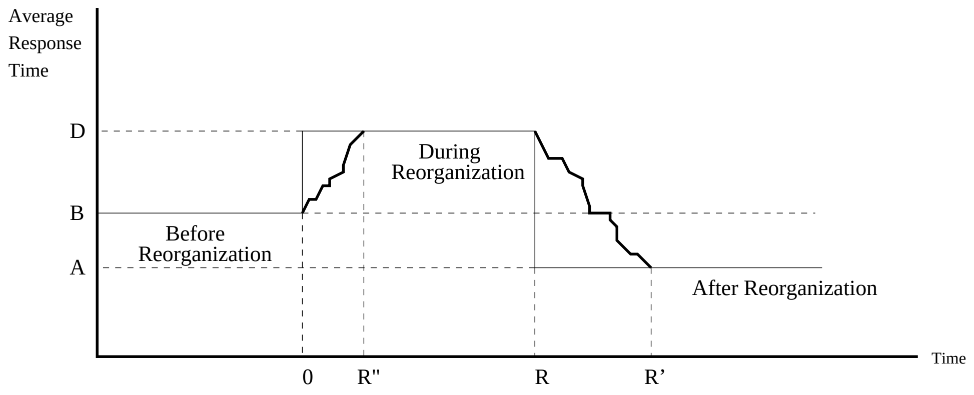


Figure 6.4: Example graph of a reorganization strategy's cost and benefit on a workload including transient times.

To allow reorganizations to be specified using the same structure as the queries, we added a new operator (SEQ) with which we can describe the sequential execution of two or more queries using one query tree. An example of this operator is shown in Figure 6.5, where a deletion is executed sequentially after an insertion to a relation X. We assume that this operator does not contribute to the visit counts, and only affects the response time, which is calculated for the SEQ operator as:

$$R_{s,k}^{SEQ} = R_{s,k}^{left\ child\ of\ SEQ} + R_{s,k}^{right\ child\ of\ SEQ}$$

The estimation method is executed as follows. For each query and reorganization, we estimate the number of times each resource (processor, disk, or interconnection network) is visited. Some formulas are given in Section A.3 of Appendix A. Once the visit counts are calculated, they are input into a QNM model to calculate the times each query and reorganization spends at a resource (*residence times*) including the delays caused by waiting for other queries to use the resources. Our QNM model includes the priority of the reorganization compared to the workload at all the resources. Once the residence times are determined, the response time of a query or reorganization is calculated similarly to the method in Figure 4.13. However, the formulas that include congestion delays change, and the estimation method becomes an iterative process. The average workload response time is again calculated as:

$$ART = \frac{\sum_{i=1}^{Q} f_i R_i}{\sum_{i=1}^{Q} f_i}$$

where Q is the number of different query classes in the workload, f_i is the arrival rate of a query class i, and R_i is the average response time for queries in class i.

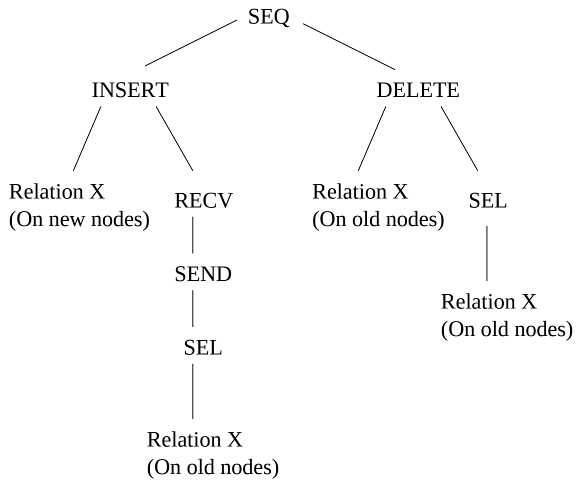


Figure 6.5: Rebalance strategy for a single relation X from a set of old nodes to a set of new nodes.

Validating a Reorganization's Response Time

To validate some of our analytic estimations for reorganizations, we used the *PART* relation from the TPC-D DB, and distributed it across an IBM DB2 PE system [BFG95] with four nodes. This relation was specified to have no indexes in one case and a nonclustering index on P_SIZE in another case. *PART* was also partitioned on $P\_PARTKEY$. The relation was imbalanced across the four nodes so that the first node was given a fraction q ($1/4 < q < 1$) of the relation's tuples. The other tuples were evenly spread across the remaining three nodes.

Our reorganization strategy is one that moves the extra tuples from the first node and evenly distributes these tuples to the other three nodes. The number of tuples moved by the reorganization is calculated as the total number of tuples times $q - 1/4$ so that each node will store $1/4$ of the relation after the reorganization.

Since the reorganization of a relation in the IBM DB2 PE system locks the whole relation so that no queries can access it during the reorganization, we compared our off-line reorganization estimates to the IBM DB2 PE reorganization timings. In the comparison, the *PART* relation had $200,000$ tuples (as in a 1 GB DB), and we varied the value of q to be between 0.3 and 0.45. The system parameters used to make the estimates are defined in Tables 2.1 and 2.2, but the number of nodes was changed to four. We show the measured and analytic reorganizations' response times in Table 6.2 for the cases when *PART* has no indexes and when it has an index on P_SIZE. Response times for the IBM DB2 PE system were obtained by averaging 20 executions of the rebalancing for each factor q. The analytic results are pessimistic compared to the measured values because our estimates are obtained using operators and timings similar to those used for normal query executions, but IBM DB2 PE has optimized these operators for reorganizations. A few examples of the optimizations are:

1. blocks of records are logged together when they are moved as for the reorganization instead of separately as for the queries; and
2. deletions are executed with updates instead of after.

However, the trends of the measured and estimated times are similar. For decision making, it is the relative differences between the timings for different reorganizations that is most important, and these relative differences are not affected substantially by our estimation method. For our experiments, we provide comparisons based on the analytic estimations.

q	**No Index**		**Index on P_SIZE**	
	Measured	**Estimated**	**Measured**	**Estimated**
0.3	13	36	20	77
0.35	23	42	32	89
0.4	33	48	54	102
0.45	52	54	61	115

Table 6.2: Response times (in seconds) for an off-line rebalancing strategy across four nodes that are analytically estimated and measured on an IBM DB2 PE system.

6.3 Defining a Reorganization Comparison Metric

To compare different reorganizations, we designed a metric that includes the cost of performing the reorganization, the benefit a workload experiences after a reorganization is complete, and the time needed to develop (or switch to) the new physical DB design. This metric will be used by a dynamic decision algorithm to allow comparisons among various candidate reorganizations that yield the same or different DB designs. We avoided using only the resulting throughput or the response time of the workload caused by the reorganized DB design because they only present the benefit after a reorganization.

Our metric is calculated using the expected response times before, during, and after a reorganization, as shown in Figure 6.6. We estimate the response times of the workload and the reorganization analytically. On the x-axis, R is the reorganization's response time, T is the time interval; and on the y-axis, B, D, and A are the average workload response times before, during, and after the reorganization, respectively. Note that the *before* response time is the response time when not performing the reorganization and using the old DB design. The new DB design is not made available to the workload until a reorganization commits. During a reorganization's execution, the reorganization adds extra cost to the workload processes as seen by the area between response times D and B and times 0 and R, while after the reorganization completes at time R, the DB state should allow for a performance improvement by the workload, as seen in the area between response times B and A and times R and T. Because of the lack of future information, we assume our average workload response time estimates are steady throughout each time period. Our calculation thus determines the areas of rectangles.

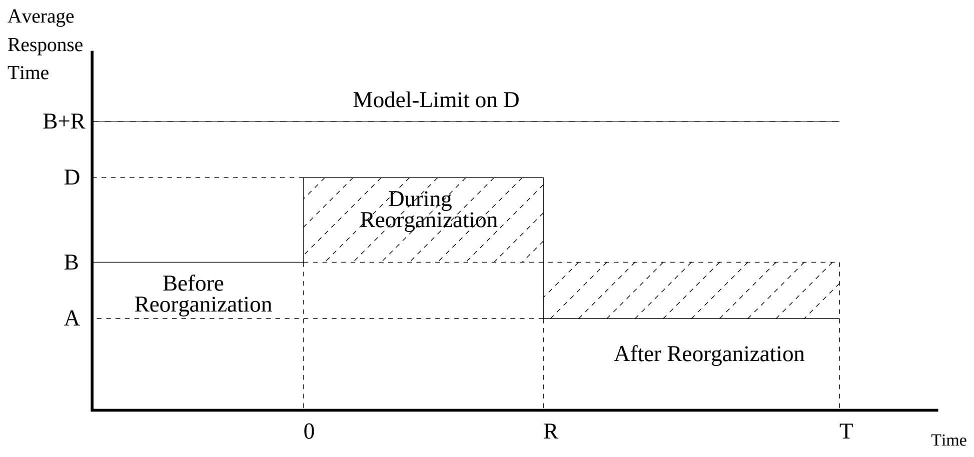


Figure 6.6: Example graph of a reorganization strategy's cost and benefit on a workload.

The metric is calculated as a net gain between the performance when executing a reorganization (calculated from the area under D between times 0 and R and under A between times R and T) compared to not executing a reorganization (calculated from the area under B between times 0 and T). A reorganization strategy is a good candidate in a decision algorithm if its net gain is greater than zero, which means executing the reorganization leads to an overall performance gain compared to not performing the reorganization. The *net gain* (or simply the *gain*) metric is defined as follows:

$$\begin{aligned}
\textit{(Net) Gain} \quad &= \textit{Area under workload ART curve without a reorganization} - \\
&\quad \textit{Area under ART curve with a reorganization} \\
&= BT - (DR + A(T - R)) \\
&= (B - A)T - (D - A)R
\end{aligned}$$

As can be seen, for a given workload and reorganization strategy, the gain metric would be linear with respect to the time interval. This formula only makes sense when $R \leq T$; otherwise, a negative gain results (assuming $A < B$).

As stated in Section 6.2, there is a *model-limit* on the response times of the queries. At the point where a high priority reorganization causes a query's response time to be delayed by more than the time to execute the reorganization, then we know the inaccuracies in our estimations are high. We calculate the model-limit on ART as $B + R$. This model-limit is shown in Figure 6.6. The value for the response times of the queries executing during a high priority reorganization is not recorded when the estimated response time exceeds this bound.

We use response times in our formula to include the effects on all resources (in the form of added delays) when the reorganization and other queries are in the system. The use of throughputs was also considered for the formulas instead of using response times. The system must be able to service the queries at their arrival rates, to avoid saturation [LZGS84]. Thus, the throughput for a query class should be equal to the class's arrival rate. However, the input arrival rate for a query class is assumed to be constant before, during, and after a reorganization. With no saturation, the throughputs do not change before, during, and after a reorganization, so using them in a metric to compare reorganizations is not helpful.

In our gain metric, we assumed that the DB design before a reorganization supports the workload. When the workload saturates the system using an old DB design, any reorganization resulting in a new design that supports the workload is a good candidate for a decision algorithm to consider.

To complete the explanation of the gain metric, we need to discuss the reason for limiting the time interval to some time T. A reorganization is only worthwhile if its cost can be (more than) amortized away before the system environment changes again. Once the system environment does change during a reorganization, our estimates for a reorganization's costs and benefits would be obsolete, and our chosen reorganization may not yield the results we expect. The DBA or some automatic algorithm would thus need to limit how long the reorganization is allowed to have an impact on the system, so that the reorganization estimates can be sufficiently accurate. This time would be system dependent. For this reason, a time limit or *time interval* (T) is included in our metric to limit the amount of time in the future over which the metric is calculated.

Choosing a suitable length for the time interval requires some thought. Too long an interval is inappropriate because further system environment changes will occur. Too short an interval leaves insufficient time to reap a benefit, thus preventing any reorganization from being chosen. We assume this time is determined and supplied by the DBA or some other source.

In reality, choosing this time interval may be difficult. One way to alleviate this problem is to determine a good lower bound for the time interval of a specific reorganization based on the response time estimates. This time interval would be the point in time when the benefit resulting after a reorganization equals the cost incurred during the reorganization, which is the time when the gain metric value equals to zero. After this time, the benefit from the reorganization will exceed its cost. We call this time the *break-even* time interval (T_{be}) for the gain metric, and it is calculated as follows:

$$Gain\ Metric = 0 \quad = \quad (B - A)T_{be} - (D - A)R$$

$$T_{be} = \frac{D-A}{B-A}R$$

assuming $B > A$. A reorganization with lowest T_{be} would be a good reorganization. However, in some experimental cases, the reorganization with the lowest T_{be} time resulted in a benefit after a reorganization being quite low. It would be preferable to only consider reorganizations that result in a high enough benefit to warrant their selection. To avoid choosing such low benefit cases, we can constrain the decision algorithm so that it chooses reorganizations only if their after reorganization ART (A) is at least $d\%$ lower than the before reorganization ART (B). In our experiments, we used $d = 5$. In the figures and tables of the following experiment sections, we use this 5% constraint unless otherwise stated.

Our model of calculating the areas under the response times curves to obtain the costs and benefits of a reorganization is flexible. The concept can be applied to *incremental reorganizations*. In an incremental reorganization, the strategy reorganizes the physical DB design using a number of steps, and when a portion of the design has been reorganized, it is made available to the DB system. For example, if a relation R were imbalanced across a set of N nodes such that the first node contains a fraction q of all the tuples (t) of R, the previous reorganization strategy would move all the excess tuples from the first node (calculated using $t(q - 1/N)$) to the other nodes. An incremental reorganization could move these excess tuples in a number of phases (h), where each phase moves a portion of these excess tuples calculated with a factor w (i.e., move $wt(q - 1/N)$ tuples in which we could have $w = 1/h$). The reason for using phases is so that the benefit from the rebalanced portion can be reaped as soon as possible by the workload.

An example of the response time curves during an incremental reorganization of h phases is shown in Figure 6.7. In these reorganizations, we assume phase i will reorganize from the DB design resulting after the reorganization phase $i - 1$, and the intrusion on the workload's response time is lowered in phase i from phase $i-1$ (i.e., lowered to D_i from D_{i-1} in the figure). This reduction between phases could cause the workload response time to become lower than the response time before the reorganization (B), as seen in Figure 6.7 after phase two. The response time curve for the workload during the reorganization would be a step function. Because we would know the DB design of each phase, we could again use the reorganization cost estimator to determine average response times for the workload and reorganization during the phase. Each phase i is assumed to end at time R_i, and the reorganization's response time is equal to R, where $R = R_h$. For such a reorganization, the gain metric would be as follows:

(Net) Gain = *Area under workload ART curve without a reorganization −*
Area under ART curve with a reorganization
$= BT - (\sum_{i=1}^{h}[D_i(R_i - R_{i-1})] + A(T - R))$

where $R_0 = 0$ and $R_h = R$. Note that the model-limit for each phase i would be calculated as $D_{i-1} + R_i$.

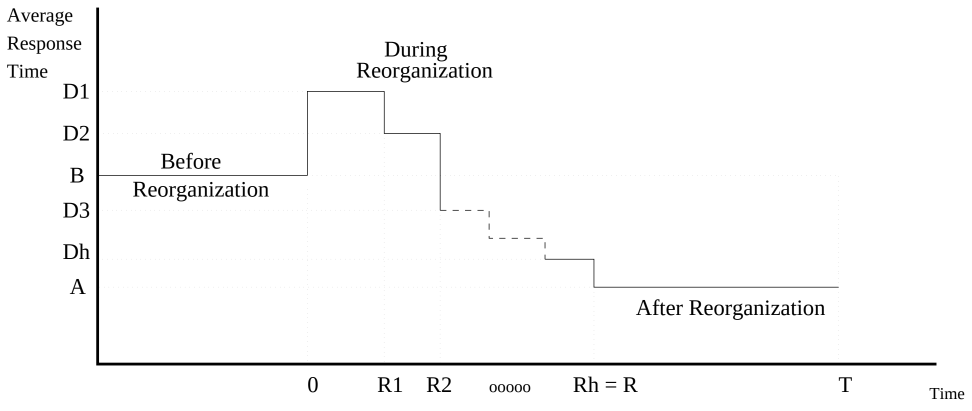


Figure 6.7: Example graph of an incremental reorganization strategy's costs and benefits on a workload.

6.4 Determining a Reorganization Strategy's Priority Level

This section focuses on the priority of the process that performs the on-line reorganization relative to the processes executing the workload. The priority of a reorganization strategy affects both the impact on the workload and the reorganization's response time. In our work, we consider three reorganization priority levels: higher, lower, or equal to the workload's level. We also consider off-line reorganization strategies only to compare their reorganization response times to the on-line ones. Each workload process is assumed to have the same priority as the other workload processes.

One common type of reorganization on which we concentrate is the rebalancing of a relation across the nodes of a parallel shared-nothing DB system. Rebalancing is the movement of a relation's tuples between nodes, with the goal of balancing the number of tuples across the nodes.

The relative priority for a reorganization is an important factor for a decision algorithm to determine. We show that the best choice of reorganization priority level with respect to the workload is affected by the arrival rate of the workload and the amount of work required from a reorganization, which, in our studies, relates to the degree of imbalance before a reorganization.

In Section 6.4.1, we first describe the reorganization strategies, the assumed data placement for the input and output DB designs, and the system structure required for the experimentation. We then show, in Section 6.4.2, some results from our priority experiments. A summary of the results then follows in Section 6.4.3.

6.4.1 Rebalancing Strategy

Our reorganizations were special types of rebalancing across a set of SN nodes. Before describing the strategies, we note that the partitioning information for the initial DB design before each rebalance was determined using the IR algorithm. The indexes were assumed to be part of the DB information input by the DBA. This initial DB design for each rebalance was used as input to the optimizer to determine the execution plans for each query in the workload. These plans were unchanged when evaluating the average workload response times before, during, and after a reorganization.

We considered two types of rebalancing scenarios, where the strategies for each of these scenarios are executed in one phase rather than incrementally. The first scenario results when the first node of an N node system contains more data than the other nodes. In this case, the strategy would be to spread some of the excess tuples at the first node evenly across the other nodes. We call this scenario: *OLDNODE*.

For our OLDNODE rebalancing, we assume that a fraction q ($1/N < q < 1$) of all the tuples are stored on the first node. The rest of the tuples for a relation are assumed to be evenly spread among the other nodes. This applies for the relations that can be spread across more than one node. In our experiments, all these relations are fully declustered. Thus, the OLDNODE strategy for a relation R executes as follows (assuming $|R|$ is the number of tuples in R and a fraction q of these tuples are on the first node):

- Read in the $(q - 1/N)|R|$ tuples from the first node that are to move to the $N - 1$ other nodes (requires a selection predicate);
- Send them (a block at a time in pipelined fashion) to the appropriate $N - 1$ nodes so that the rebalanced relation will have $|R|/N$ tuples at each node;
- As tuples are received, insert the relocated tuples at the $N - 1$ nodes; and
- Delete the original copy of the moved tuples on the first node.

In our second scenario, the system before a reorganization consists of N nodes, and a DB is placed on these N nodes. The DBA then adds N more nodes to the system, and requires the DB to be placed across the $2N$ nodes because it is deemed that the workload performance across the N node system is not good (since either the system is heavily utilized or the workload performs much better across $2N$ nodes). When all the relations are placed initially on the first N nodes, the new N nodes are not utilized, and there exists a data (and load) imbalance across the nodes. Because of the imbalance, the reorganization must rebalance the tuples (and load) across the nodes of the $2N$ system. We call this scenario: *NEWNODE*.

Our NEWNODE rebalancing can actually be considered to be the movement of the excess records of one or more relations stored across the first N nodes to the N new nodes. Thus, we constructed the strategy to move half of the records of a fully declustered relation to the new nodes. The NEWNODE strategy for a relation R (as shown in Figure 6.5) is executed as follows:

- Read in the half of R that is to move to the new N nodes (requires a selection predicate).
- Send them (a block at a time in pipelined fashion) to the appropriate new nodes so that the fully declustered relations are now fully declustered over $2N$ nodes.
- As tuples are received, insert them on the new nodes.
- Finally, delete the original copy of the moved records on the original N nodes.

We assume that, as the reorganization is executing, the tuples being relocated are still accessible at their original locations. In reorganization strategies defined in the literature, this is usually accomplished with a differential file for each relation on the original nodes to capture the update operations on the relocated tuples by the workload. These updates must be applied to the tuples at their new locations before the original tuples are deleted and the new copies are allowed to be accessible. Thus, relocated tuples would only be accessible on a reorganization's completion. However, for our experiments, we assume the costs related with using a differential file to be negligible.

6.4.2 Experiments

Before presenting the results of our experiments, we describe the set of inputs used to determine the analytic costs of the reorganization and workload. Tables 2.1 and 2.2 give the system-related inputs, while Appendixes D and E describe the inputs for the DB and workloads (WORK1, WORK2, WORK3, and WORK4). We use a 4 GB TPC-D database size as our default DB size for WORK1. For WORK2, WORK3, and WORK4, we use the database cardinalities defined in Appendix E. The number of nodes N for NEWNODE is eight, and thus we analyze the rebalancing from an eight to a sixteen node SN system. However, for OLDNODE, we use sixteen for N, and for most experiments (except when the degree of imbalance, q, is varied), the first node stores 70% ($q = 0.7$) of the fully declustered relations' tuples. All relations in the databases (except for the NATION and REGION relations) are considered to be fully declustered across the initial nodes of the system.

In our experiments, we use three performance measurements. These are used for the y-axes in the graphs. The first is the average response time of the workload executing during a reorganization, and the second is the response time of the reorganization. These measurements are used to show how the response times are influenced under various conditions relating to the execution strategy and priority for a reorganization. Our third measurement is the break-even time interval (T_{be}). The reorganization and priority with the lowest break-even time interval for a given system environment is considered to be best. This lowest break-even time can be considered as the lower bound for any time interval relating to these types of reorganizations. It should be noted that, by using our gain metric calculation, the winning reorganization would always be the reorganization with the lowest break-even time interval. The reason for this is because, in our NEWNODE or OLDNODE experiments, each reorganization results in the same physical DB design and thus in the same benefit after the reorganizations were complete. Thus, it is important to reach this benefit as soon as possible with a low relative cost of performing the reorganization.

Varying the DB Size

In our first set of experiments, we vary the DB sizes to demonstrate whether the different sizes result in different effects upon the chosen priority for a reorganization. We varied the DB sizes by multiplying the full DB size cardinalities (defined by the 4 GB DB for TPC-D and by the cardinalities in Appendix E for the complex workloads) by a factor between 0.1 and 1. Experiment results were obtained for both NEWNODE and OLDNODE reorganization strategies. The results for the *Low* arrival rates are shown, where the rates are determined to be the rates yielding a bottleneck utilization of 40% for DB size factor 1.

From our investigations of the priority level effects on the reorganization and workload response times, we noticed that in many cases, the high priority reorganization causes the workload ART during the reorganization to reach its model-limit, and thus we do not plot the high priority reorganization results for these cases. For example, Figure 6.8 shows the average response time of the NEWNODE reorganization and workload under the HIGH, EQUAL, and LOW priorities when PR priority scheduling is only done at the processors. The off-line reorganization times are also shown. From our studies, we concluded that the off-line reorganization's response times are the lowest among all reorganizations, but the workload's delays (shown as response times in the figures) incurred while a high priority reorganization executes are the highest among the times for a workload when it executes with the other reorganizations. As was expected, both the reorganization and workload times increase when the DB size increases, as seen in Figures 6.8 and 6.9. This result holds for all reorganizations. The increase is a result of the greater amount of data that must be moved for larger DB sizes, which leads to longer reorganizations with greater intrusions on the workload queries.

Similar results were obtained when we used PR priority scheduling at all the resources. An example of executing the OLDNODE decision algorithm with workload WORK1 is shown in Figure 6.10, while the results when executing the NEWNODE decision algorithm are shown in Figure 6.11. However, the model-limit on ART during high priority reorganizations is reached

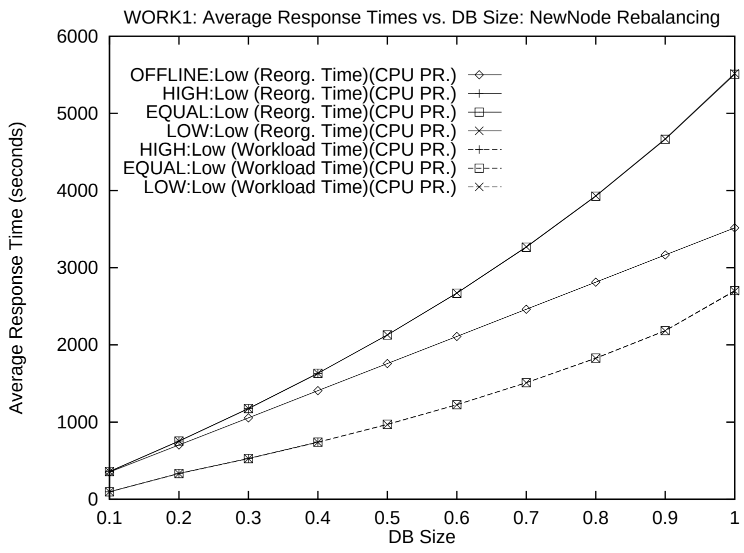


Figure 6.8: DB Size factor versus the average response time for the NEWNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is only done at the processors (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

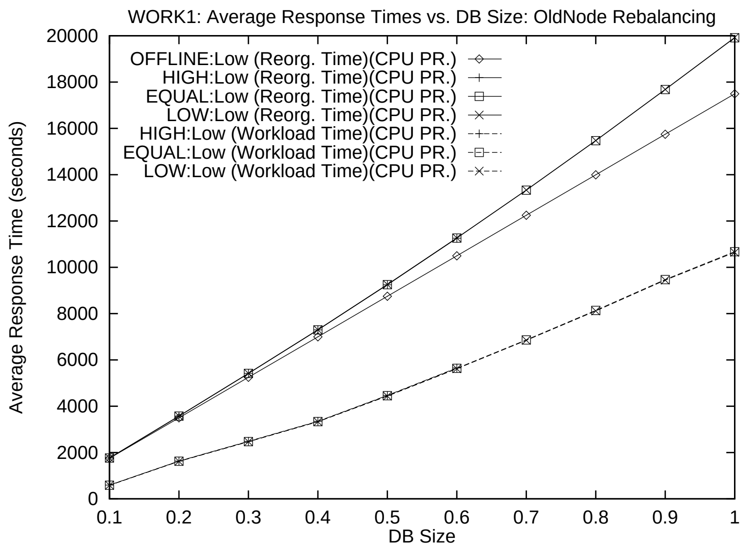


Figure 6.9: DB Size factor versus the average response time for the OLDNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is only done at the processors (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

for all DB sizes, and the difference between the low and equal priority is more pronounced. This greater difference is a result of the delay on the reorganization incurred at the disks from the workload at low priority.

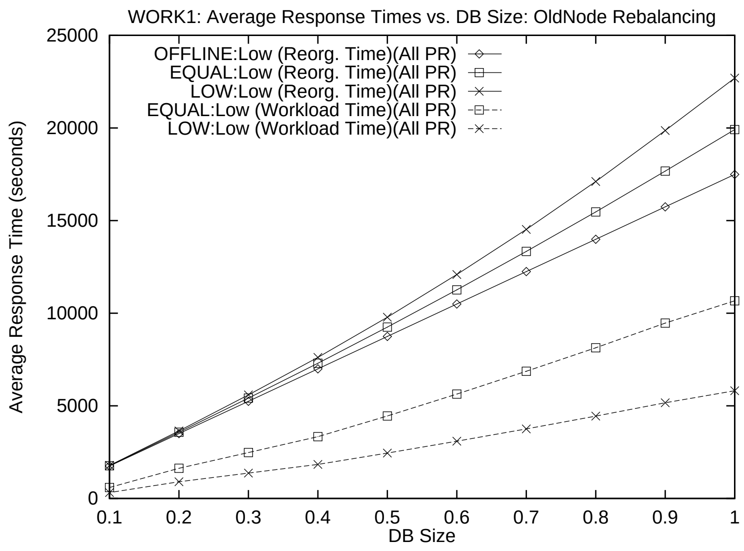


Figure 6.10: DB Size factor versus the average response time for the OLDNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is done at all the resources.

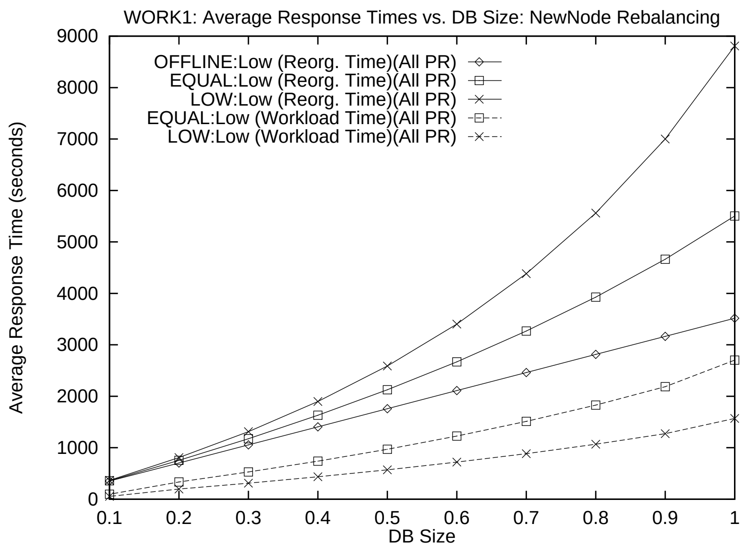


Figure 6.11: DB Size factor versus the average response time for the NEWNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is done at all the resources.

We wished to determine the priority with the shortest break-even time. We also wanted to determine how increasing the arrival rates affects the priority that yields the lowest break-even time interval. Using PR priority scheduling at the processors only results in low priority reorganizations having marginally better break-even time intervals than the times for equal priority reorganizations. The reason for this is that the low, equal, and high priority reorganization times are close to each other, but low priority reorganizations cause marginally better workload ART during its execution. For example, these results are seen for workload WORK1 and the NEWNODE reorganization in Figure 6.12. Some break-even time intervals are shown in Table 6.3. These results are also seen for the other workloads, e.g. for workload WORK2 using the NEWNODE reorganization as seen in Figure 6.13 and Table 6.4.

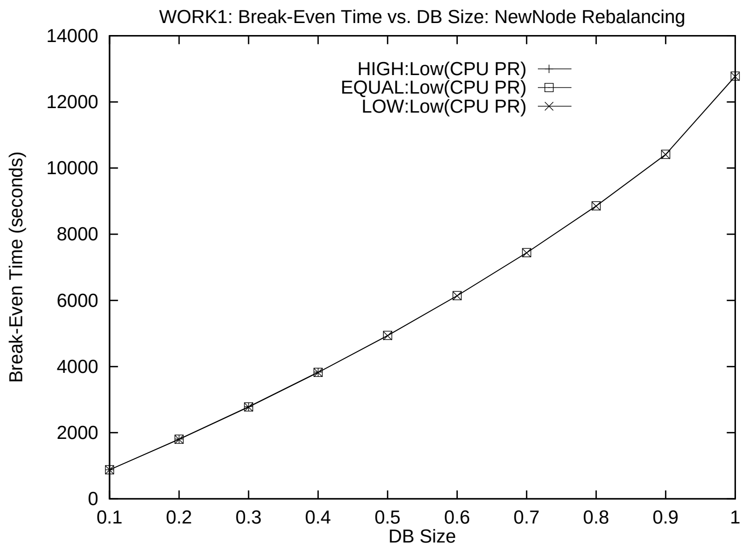


Figure 6.12: DB Size factor versus the break-even time interval for the NEWNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is only done at the processors (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

When using PR priority scheduling at all the resources, the results are similar to having PR priority scheduling at the processors only, but are more pronounced. For example, Figure 6.14 shows the results when executing the NEWNODE rebalancing decision algorithm with workload WORK2. The break-even time intervals for low priority reorganizations are much better than the times for the other reorganizations. Also, for most DB sizes and arrival rates, the break-even times for the high priority reorganizations are quite high. This is due to the fact that, for WORK2, WORK3, and WORK4, all the fifteen relations in the database are reorganized, which leads to high reorganization times. More importantly, the reorganization causes very high delays to the workload when the transaction classes are delayed by the higher priority reorganizations. In the formula for break-even time intervals, the delay is in the numerator,

DB Size Factor	T_{be} **for Low Priority**	T_{be} **for Equal Priority**	T_{be} **for High Priority**
0.1	874.6	877.6	880.8
1	12767.3	12782	N/A

Table 6.3: Break-even time intervals (in seconds) for reorganizations of different priorities versus the DB size factor for the NEWNODE reorganization using workload WORK1 when PR priority scheduling is only done at the processors.

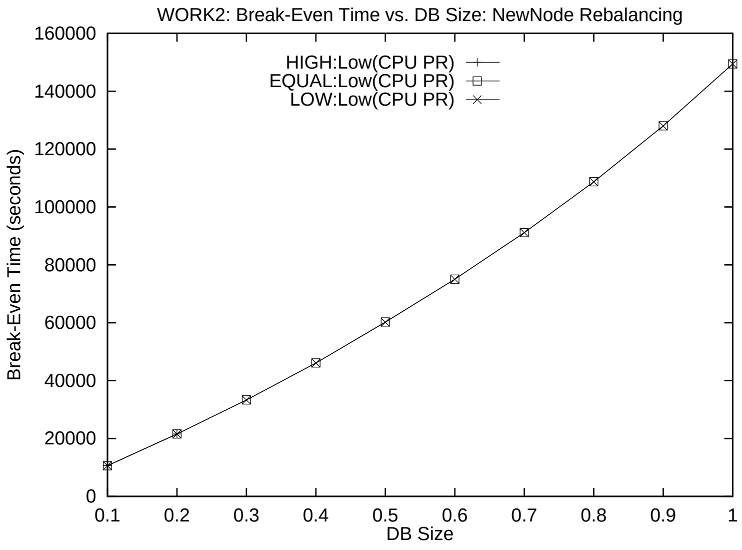


Figure 6.13: DB Size factor versus the break-even time interval for the NEWNODE reorganization at different priority levels and the workload WORK2 executing during a reorganization when PR priority scheduling is only done at the processors (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

DB Size Factor	T_{be} **for Low Priority**	T_{be} **for Equal Priority**	T_{be} **for High Priority**
0.1	10605.3	10623.9	10643.5
1	149394	149451	N/A

Table 6.4: Break-even time intervals (in seconds) for reorganizations of different priorities versus the DB size factor for the NEWNODE reorganization using workload WORK2 when PR priority scheduling is only done at the processors.

and for large delays, the break-even times are high.

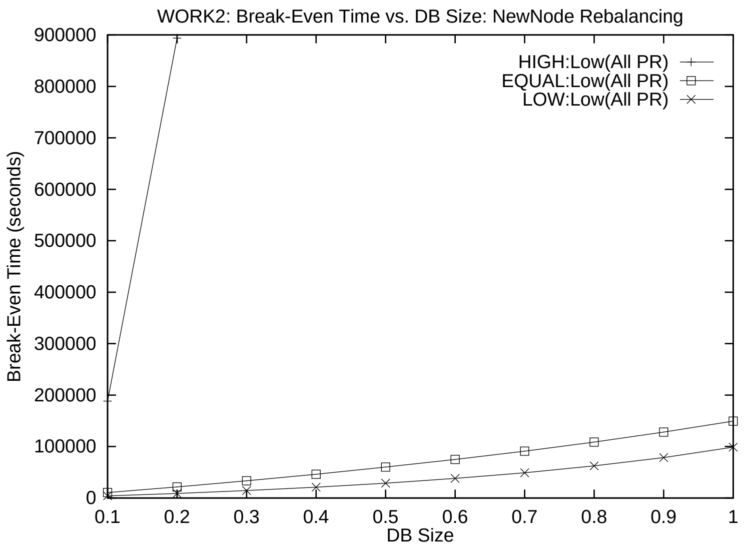


Figure 6.14: DB Size factor versus the break-even time interval for the NEWNODE reorganization at different priority levels and the workload WORK2 executing during a reorganization when PR priority scheduling is done at all the resources.

For higher arrival rates and PR priority scheduling at all the resources, the reorganization having the lowest break-even time interval changes to one with higher priority compared to the priority used at lower rates for larger DB sizes. For example, from the results shown in Figure 6.15 when executing the OLDNODE decision algorithm using workload WORK4, we noticed that, for *Med* arrival rates compared to *Low*, a change occurs from the low priority to the equal priority reorganization for DB size factors greater than 0.9. This increase in the priority level is caused by the higher priorities allowing reorganizations to execute with the workload, which reduces the reorganization time and thus the break-even times. The change from low to equal priority reorganizations for higher arrival rates and larger DB sizes was also seen when using PR priority scheduling at the processors only, but the change was less pronounced.

Varying the Arrival Rates (Load on the System)

In this experiment, the (*Low*) arrival rates were varied so that the resulting bottleneck utilization for the workload before the reorganization varies between 0.1 and 0.9. We found that the priority with the lowest break-even time changes depending on the bottleneck utilization. For example, for NEWNODE rebalancing, this result is shown for workload WORK1 in Figure 6.16 when PR priority scheduling is used at all the resources. When the utilization is low, the winning priority is the low priority for the reorganization process. However, as the utilization increases (and thus the load on the system increases), the winning priority moves from low

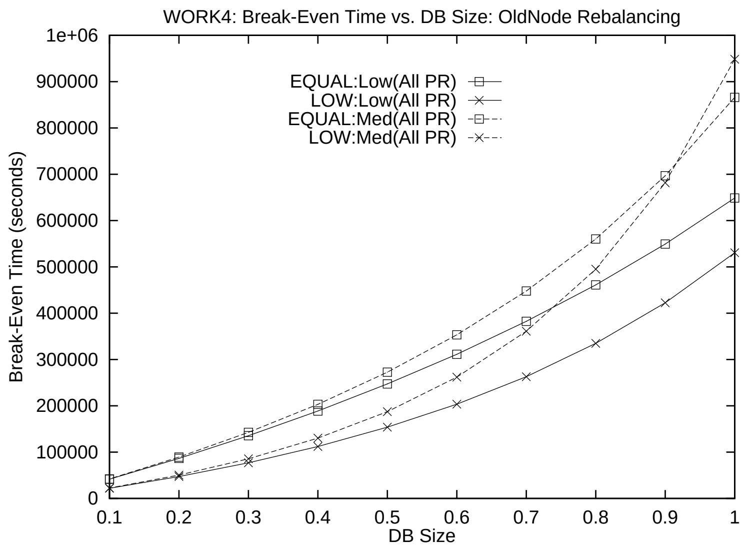


Figure 6.15: DB Size factor versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK4 executing during a reorganization when PR priority scheduling is done at all the resources.

to equal priority. The reason for this change of priorities is because, at higher utilizations, the response times for the low priority reorganization becomes much higher than for the times of equal priority reorganizations. This can be seen in the response time results for workload WORK1 in Figure 6.17. These results were also seen for the other workloads (e.g., WORK2 has results shown in Figure 6.18) and for the OLDNODE rebalancing (e.g., the results for WORK4 are given in Figure 6.19). Therefore, as the load on a system increases, faster reorganizations that achieve their goals as quickly as possible are best.

Similar results were seen when PR priority scheduling is used only at the processors. However, the large differences between low and equal priority reorganization results are reduced because PR priority scheduling at the processors only is not as intrusive. This suggests that the processors are not the critical resources for our reorganizations. For example, Figure 6.20 shows the results when executing the OLDNODE decision algorithm for workload WORK1. Table 6.5 gives the results for the different priorities for two bottleneck utilizations. Results for WORK3 are shown in Figure 6.21 and Table 6.6. For both modeling approaches for PR priority scheduling, high priority reorganizations always caused the model-limit on the workload's ART during the reorganization to be exceeded except at low utilizations.

Varying the Degree of Imbalance

For OLDNODE rebalancing, the first node is always the node storing more data than the other nodes. This amount of data is q times the total data for each fully declustered relation

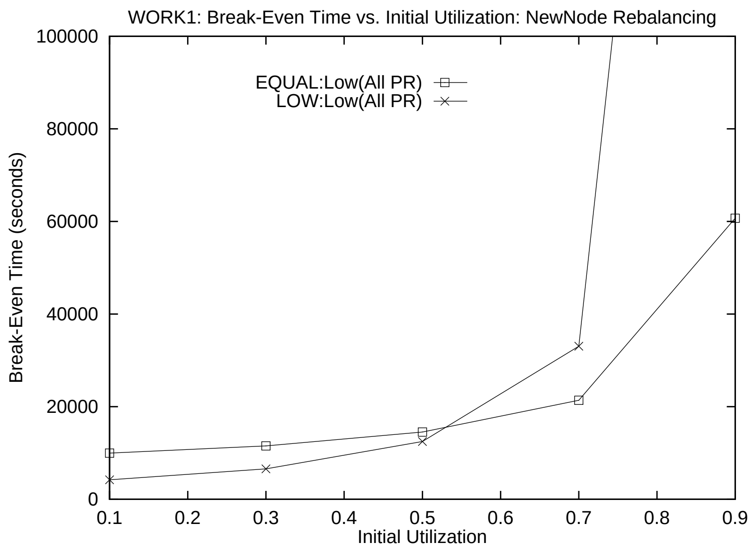


Figure 6.16: Bottleneck utilization versus the break-even time interval for the NEWNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is done at all the resources.

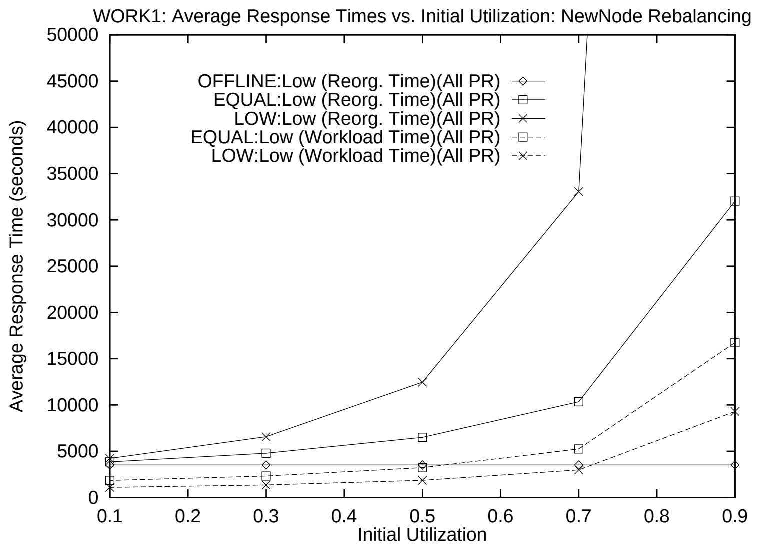


Figure 6.17: Bottleneck utilization versus the average response time for the NEWNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is done at all the resources.

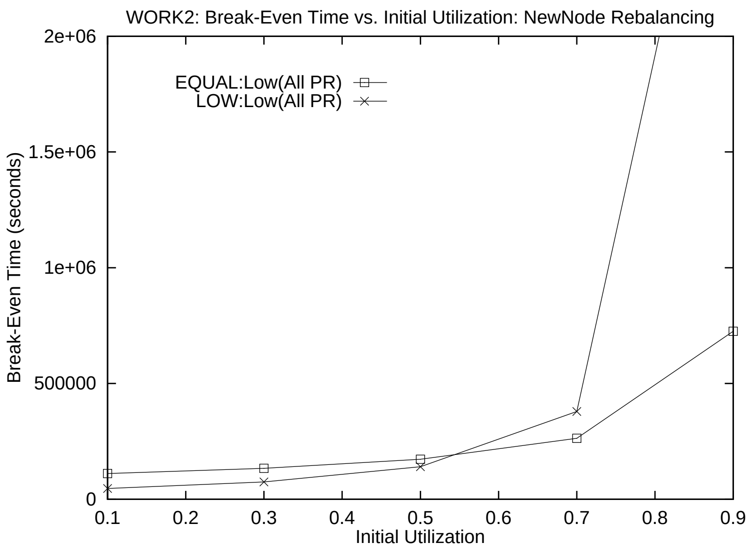


Figure 6.18: Bottleneck utilization versus the break-even time interval for the NEWNODE reorganization at different priority levels and the workload WORK2 executing during a reorganization when PR priority scheduling is done at all the resources.

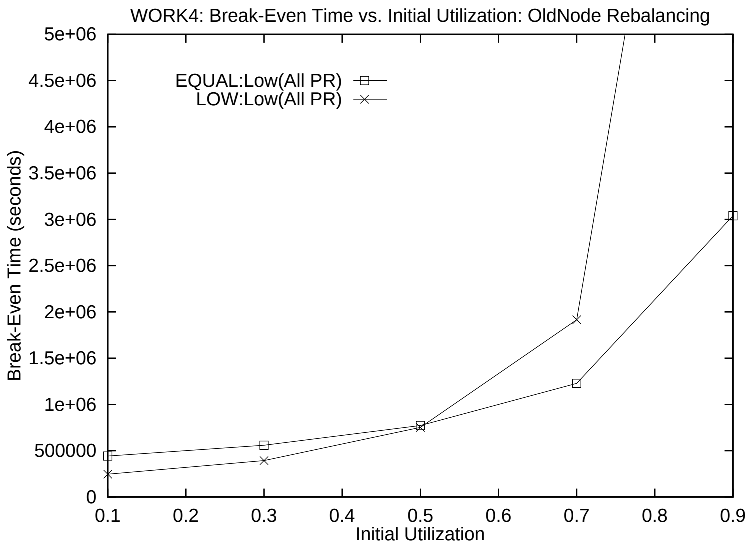


Figure 6.19: Bottleneck utilization versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK4 executing during a reorganization when PR priority scheduling is done at all the resources.

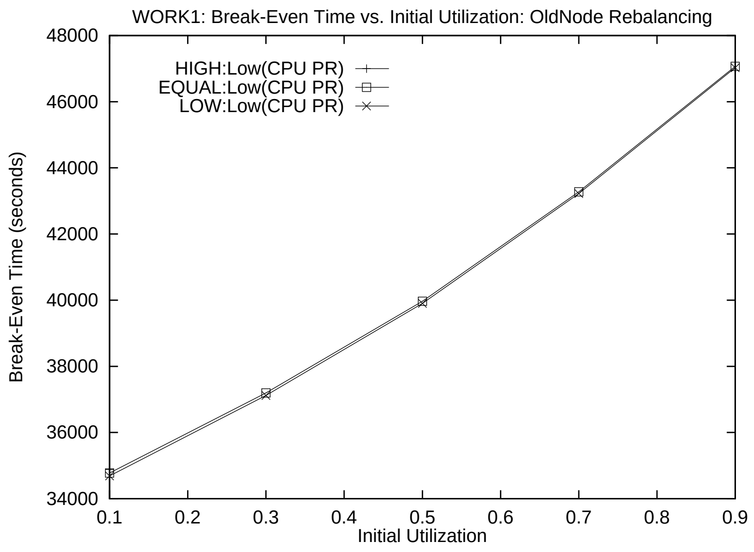


Figure 6.20: Bottleneck utilization versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK1 executing during a reorganization when PR priority scheduling is done at the processors only (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

Bottleneck Utilization	T_{be} **for Low Priority**	T_{be} **for Equal Priority**	T_{be} **for High Priority**
0.1	34685	34775.8	34878.4
0.9	47027.3	47068.7	N/A

Table 6.5: Break-even time intervals (in seconds) for reorganizations of different priorities versus the bottleneck utilization for the OLDNODE reorganization using workload WORK1 when PR priority scheduling is only done at the processors.

Bottleneck Utilization	T_{be} **for Low Priority**	T_{be} **for Equal Priority**	T_{be} **for High Priority**
0.1	73970.6	74027.6	N/A
0.9	798704	798637	N/A

Table 6.6: Break-even time intervals (in seconds) for reorganizations of different priorities versus the bottleneck utilization for the OLDNODE reorganization using workload WORK3 when PR priority scheduling is only done at the processors.

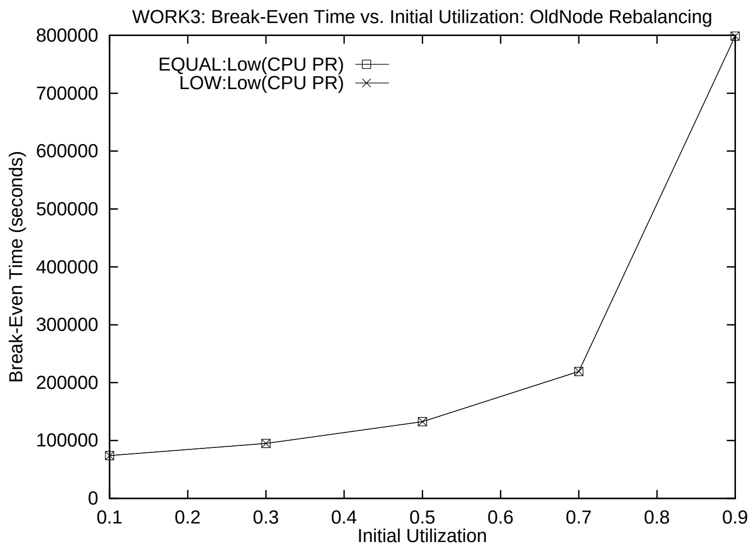


Figure 6.21: Bottleneck utilization versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK3 executing during a reorganization when PR priority scheduling is done at the processors only (where the times for EQUAL and LOW priority reorganizations are not identical but are close to each other).

($1/N < q < 1$). For these experiments, we used the full DB size required for each workload, but varied the imbalance at the first node so that q is between 0.1 and 0.9. A reorganization for each imbalanced DB using a specific value for q is one that results in each node storing $1/N$-th of the tuples in the DB. The arrival rates were determined using $q = 0.9$, as opposed to $q = 0.7$ for the other experiments.

The break-even time intervals for the high priority reorganization are limited by the model-limit on the workload response times for both modeling methods of including PR priority scheduling. For example, this can be seen in Figure 6.22 for workload WORK3 when PR priority scheduling is used only at the processors. For degrees of imbalance greater than 0.2, the high priority reorganizations cause the model-limit to be exceeded. Thus, the low and equal priority reorganizations have the lowest break-even times. Some of the break-even time intervals for the different priority levels and arrival rates are given in Table 6.7.

For higher arrival rates using PR priority scheduling at all the resources, equal priority reorganizations have the lowest break-even times at high degree of imbalance compared to lower priority reorganizations. For example, in Figure 6.23, for Med rates, the equal priority reorganization has lowest break-even times starting at a degree of imbalance of 0.8. One reason for this change in priorities is that the Low, Med, and $High$ arrival rates are determined using the degree of imbalance of 0.9, and the other degrees of imbalance result in less work and lower utilizations. For workloads causing low utilizations, a low priority reorganization is preferred.

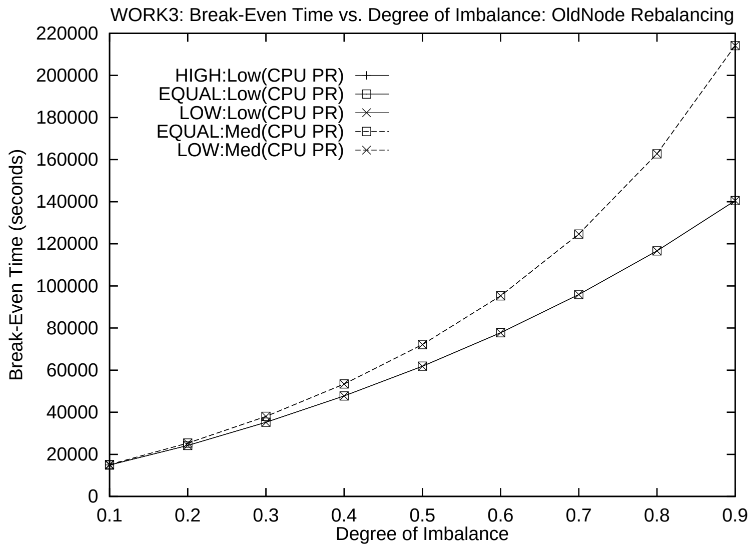


Figure 6.22: Degree of imbalance at the first node versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK3 executing during a reorganization when PR priority scheduling is done only at the processors (where the times for HIGH, EQUAL, and LOW priority reorganizations are not identical but are close to each other).

Degree of Imbalance	Arrival Rate	T_{be} **for Low Priority**	T_{be} **for Equal Priority**	T_{be} **for High Priority**
0.1	Low	14907.6	14921.6	14936.1
0.9	Low	140513	140532	N/A
0.1	Med	15132.6	15146	N/A
0.9	Med	214161	214136	N/A

Table 6.7: Break-even time intervals (in seconds) for reorganizations of different priorities versus the degree of imbalance for the OLDNODE reorganization using workload WORK3 when PR priority scheduling is only done at the processors.

In the other experiments, such as when the DB sizes were varied, the default imbalance is $q = 0.7$, and the higher arrival rates for this imbalance increase the priority level of the winning reorganization (as seen in Figure 6.15 for workload WORK4).

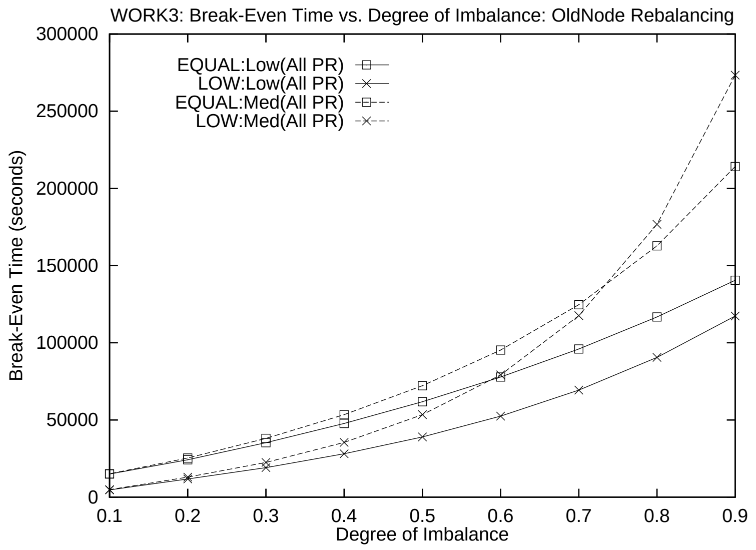


Figure 6.23: Degree of imbalance at the first node versus the break-even time interval for the OLDNODE reorganization at different priority levels and the workload WORK3 executing during a reorganization when PR priority scheduling is done at all of the resources.

Modeling with PR priority scheduling at the processors only causes the low priority reorganization to execute on the disks and interconnect network more than when PR priority scheduling is used at all the resources. This causes break-even time intervals for low priority reorganizations to be close to the intervals found for equal priority reorganizations when PR priority scheduling is used only at the processors. For example, this result can be seen in Figure 6.22 and Table 6.7 when executing the OLDNODE rebalancing decision for workload WORK3. At low arrival rates, low priority reorganizations had better break-even times, but at higher rates and high degree of imbalance, equal priority reorganization had better times.

6.4.3 Summary

While studying several reorganizations that produce the same physical DB design, we observed the following:

1. The off-line reorganization has the lowest response times followed by the high, equal, and low priority reorganizations. However, when the workload executes with the reorganization, the workload response times are highest for low priority reorganizations, and the workload response times during equal and higher priority reorganizations are next highest and lowest, respectively.

2. When the reorganizations all result in the same physical DB design after a reorganization, the best reorganization (and priority) is one that yields the lowest break-even time interval.

3. Increasing either the database size, the arrival rates, or the degree of imbalance of the DB across the nodes causes the reorganization and workload response times to increase. For the high priority reorganizations, the workload's average response times reach the model-limit on response times for the increased parameter values.

4. With PR priority scheduling at all the resources, the high priority reorganizations reached the model-limit faster than when PR priority scheduling was allowed at only the processors.

5. The increase in either the database size, the arrival rates, or the degree of imbalance of the DB across the nodes causes the reorganization with the lowest break-even time interval to change between a low priority and an equal priority, in that order. This is caused by the greater need to reach the benefits as quickly as possible. However, the change between priorities is less severe when PR priority scheduling is only used at the processors.

6.5 Specific Reorganization Decision Algorithms

In this section, we discuss and evaluate reorganization decision algorithms that have structures similar to Algorithm B of Figure 6.2. The goals we wish to achieve through the studies in this section are as follows:

1. We want to gain some insight about general reorganization decision algorithms by investigating two example decision algorithms.

2. We demonstrate how the components required for a decision algorithm (e.g., the comparison metric) are used in a real algorithm.

3. We provide insights when comparisons are made between reorganizations yielding different physical DB designs.

4. In Section 6.4, the only useful time interval was the break-even time interval since all reorganizations resulted in the same physical DB design and thus the same performance after the reorganization. However, when we allow the resulting physical DB designs to differ between reorganizations, the performance after a reorganization differs between the reorganizations. This will have an effect on the gain metric and break-even time interval calculations. Also, we conjecture that varying the time interval will result in the choice of a different reorganization for each time interval because the gain metric values will vary

depending on the given time interval. Thus, a goal of our work is to investigate the effects of varying the time interval on the chosen reorganization's gain metric values.

5. Also in Section 6.4, each reorganization moved the same amount of data. In this section, we investigate the effects of comparing reorganizations using different priorities and moving different amounts of data. These studies involve various system environments, e.g., environments that vary the total DB size or the degree of data skew (imbalance) across the nodes, as well as environments with different workloads and arrival rates.

In our example decision algorithms, we determine a new (rebalanced) physical DB design to attain, and then determine the reorganization strategy by which to attain this design. The decision algorithm considers only two specific types of reorganizations, each of which determines whether to rebalance all relation groups or just a single relation group. The result is two different decision algorithms that differ in the reorganization type they consider. At each decision point, the choice is made to rebalance the group with the lowest break-even time interval or highest gain metric value. Note that, in Appendix G, we provide tables, corresponding to some tables in this section, labeled with *other results* to show the T_{be} values for some other relation groups and priorities that were not the lowest. The search is performed exhaustively. It also determines at which priority to reorganize the chosen group. We present the pseudo code for the algorithm that chooses the reorganization with the lowest break-even time interval in Figure 6.24. Although we use two specific reorganization types (rebalancing) to implement the algorithm in Figure 6.24, the pseudo code for the algorithm is general enough to include any reorganization type.

The first algorithm only considers reorganization strategies that rebalance the relation group(s) across $2N$ nodes as opposed to N nodes. The resulting strategy from executing this algorithm will denote the best group (or all the groups) to reorganize. We call this decision algorithm the *NEWNODE* rebalancing decision algorithm.

For the second algorithm, we assume that the first node is the node with the most amount of data. The reorganization strategies considered are ones that move the excess data of the relations of a single group or all the groups from the first node to all the other nodes. We call this decision algorithm the *OLDNODE* rebalancing decision algorithm.

Our experiments include varying the DB sizes, the degree of imbalance at the first node, the time interval, and the arrival rates of each query class in the workload. These are varied to demonstrate their effects on which group is chosen and either the resulting break-even time interval or the best gain metric value. The results are given in the following sections. In the tables that follow, the relation groups will be given, where *ALL* represents the choice to rebalance all relation groups. As in the priority experiments, the partitioning for the initial DB design for each decision was determined by the IR algorithm while the indexing and clustering was given

```
algorithm Reorganization Decision Algorithm

input: The current physical DB design (including groups),
        along with current DB, workload, and system information
output: A best break-even time interval $T_{be}^{\min}$,
        the best reorganization strategy $Y_{\min}$ (which, if it is
        a rebalancing, contains the best set of groups $G^{\min}$),
        and the best priority for the strategy $p^{\min}$
definitions: $T_{be}$ is the break-even time interval for a specific reorganization
        strategy with a given priority ($p$)
1. initialize: None.
2. for each possible reorganization strategy $Y$ do
    {If the strategy is to be a rebalancing of one or more relation groups,
     the set of groups ($G$) must also be determined as part of the strategy $Y$, where
     all the groups (if chosen to be reorganized together) is given a special
     group number ($ALL$).}

3.      for each priority $p$ of the reorganization process relative
                to the workload processes do
4.          Determine the break-even time $T_{be}$ for strategy $Y$ with priority $p$
5.          if $T_{be} < T_{be}^{\min}$ or $T_{be}^{\min}$ is not set then
6.              $T_{be}^{\min} \leftarrow T_{be}$
7.              $Y^{\min} \leftarrow Y$
8.              $p^{\min} \leftarrow p$
9.          end if
10.     end for
11. end for
```

Figure 6.24: An exhaustive reorganization decision algorithm that uses the break-even time interval as the measure for comparison.

with the input DB information. Also, the query plans for each decision were determined using the initial DB design, and the plans remained unchanged throughout the decision in order to avoid including the cost of re-optimizing as part of a reorganization's cost.

Through all of our experiments, a high priority reorganization was never chosen. There were two reasons for this. In some cases, the reorganization was worse than the other reorganizations when comparing them with the break-even time or the gain metric values. In all other cases, the reorganization caused the workload ART to surpass its model-limit, and thus did not allow the ART to be estimated by our model.

6.5.1 Varying the DB Sizes

We varied the DB sizes to determine how the reorganization decisions are affected. In these experiments, the size varies from a factor of one tenth to a factor of one relative to the full DB size. We also varied the arrival rates in these experiments from *Low* rates (determined as the rates required to set the bottleneck utilization to 40% for the workload using the full DB size and the initial DB design) to Med and $High$ rates (determined as the rates causing the bottleneck utilizations to be 60% and 80%, respectively).

We observe that, as the DB size increases, the group with the best break-even time interval (T_{be}) changes from a group with more and larger relations to one with fewer and smaller relations. For example, these results can be seen for the NEWNODE rebalancing decision algorithm in Table 6.8 for workload WORK1 using the *Low* arrival rates when PR priority scheduling is used at all the resources. From Table 6.8, the winning priority was the low priority for the reorganization. However, for larger DB sizes, the group to reorganize is reduced from relation group 1 (which contains relations *LINEITEM* and *SUPPLIER*) to the smaller group 3 (which contains relations *PARTSUPP* and *PART*). Notice that the response times of the workload before and during a reorganization tend to be the same since the low priority reorganization does not hinder the workload during its execution. Because of this, the resulting break-even time interval will be equal to the reorganization's response time. When PR priority scheduling is done only at the processors, more relations can be reorganized at the large DB sizes, and the break-even time intervals increase compared to times when PR priority scheduling is modeled at all the resources for the same DB factor. An example of this was seen for the results from the NEWNODE decision algorithm for WORK1, where Table 6.8 shows the results when PR priority scheduling is at all the resources and Table 6.9 gives the results when PR priority scheduling is at the processors only. This is because more intrusion on the workload from low priority reorganizations exist when PR priority scheduling is used only at the processors.

As the arrival rates increase, the priority of the reorganization with the lowest break-even time interval also increases and moves eventually to being an equal priority reorganization. For example, Table 6.10 shows the results of the OLDNODE rebalancing decision algorithm for

	Low Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	909	909	1113	1113	1048	3	LOW
0.9	750	750	904	904	852	3	LOW
0.8	616	616	772	772	730	3	LOW
0.7	2106	2106	649	649	383	1	LOW
0.6	1685	1685	535	535	321	1	LOW
0.5	1315	1315	430	430	262	1	LOW
0.4	989	989	332	332	205	1	LOW
0.3	698	698	240	240	150	1	LOW
0.2	440	440	154	154	97	1	LOW
0.1	301	301	55	55	28	1	LOW

Table 6.8: The best break-even times when varying DB sizes for the NEWNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done at all the resources.

	Low Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	7172	2733	1113	1872	645	1	LOW
0.9	6039	2355	904	1514	515	1	LOW
0.8	5236	2018	772	1289	447	1	LOW
0.7	4476	1705	649	1081	383	1	LOW
0.6	3752	1413	535	889	321	1	LOW
0.5	3061	1140	430	713	262	1	LOW
0.4	2400	885	332	549	205	1	LOW
0.3	1765	644	240	397	150	1	LOW
0.2	1144	417	154	254	97	1	LOW
0.1	755	295	55	96	28	1	LOW

Table 6.9: The best break-even times when varying DB sizes for the NEWNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done at the processors only.

WORK4 using the *High* arrival rates with PR priority scheduling allowed at all the resources. For the *Low* and *Med* rates, the winning priority was always the low priority. When we increased the rates to the *High* rates, the reorganization with the lowest break-even time interval changed from a low priority to an equal priority reorganization. Also, at higher DB sizes, the benefit after a reorganization and the shorter reorganization time offsets the higher intrusion to the workload from the equal priority reorganization. When PR priority scheduling is only at the processors, similar results were seen. For example, Table 6.11 shows the results of the OLDNODE decision algorithm for WORK4.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	149078	48271.3	12021.9	23623.6	6466.4	10	EQUAL
0.9	98959.1	98959.1	8565.6	8565.6	7413.1	1	LOW
0.8	59961.3	59961.3	6313.4	6313.4	5600.3	1	LOW
0.7	38083.7	38083.7	4724.8	4724.8	4262.0	1	LOW
0.6	24805.5	24805.5	3543.2	3543.2	3237.6	1	LOW
0.5	16257.8	16257.8	2627.1	2627.1	1920.7	1	LOW
0.4	10510.5	10510.5	1894.9	1894.9	1762.8	1	LOW
0.3	6510.7	6510.7	1294.9	1294.9	1212.5	1	LOW
0.2	3649.6	3649.6	793.1	793.1	746.6	1	LOW
0.1	1555.8	1555.8	363.1	363.1	313.5	1	LOW

Table 6.10: The best break-even times when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK4 when PR priority scheduling is done at all the resources.

For databases with relations of different sizes (which result in relation groups of different sizes), fewer and smaller relations are reorganized when PR priority scheduling is done at all resources as opposed to just the processors. For example, Table 6.12 gives the results for the OLDNODE decision algorithm using workload WORK1 when PR priority scheduling is only at the processors. Table 6.13 shows the results when PR priority scheduling is at all resources. The database for workload WORK1 has relation group 1 containing the *LINEITEM* and *SUPPLIER* relations, relation group 3 contains the *PARTSUPP* and *PART* relations, and group 2 contains the *ORDERS* relation. These results indicate that relation group 1 (the larger group) should be reorganized for all DB sizes when PR priority scheduling is only at the processors. This group should also be reorganized with equal priority for DB factors greater than 0.7. When

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	149078	48271.3	12021.9	23623.6	6466.4	10	EQUAL
0.9	115648	34336.5	8565.6	16748	5110.3	10	LOW
0.8	92001.2	25252.7	6313.4	12288.9	4052.7	10	LOW
0.7	73775.7	18859.7	4724.8	9156.3	3202.9	10	LOW
0.6	58887.5	14113	3543.2	6837.0	2505.0	10	LOW
0.5	46230.9	10442.8	2627.1	5047.9	1920.7	10	LOW
0.4	35138.1	7517.1	1894.9	3626.0	1423.8	10	LOW
0.3	25163.5	5127.2	1294.9	2467.9	994.7	10	LOW
0.2	16053.6	3135.6	793.1	1505.6	620.2	10	LOW
0.1	7682.8	1448.0	363.1	687.0	287.9	10	LOW

Table 6.11: The best break-even times when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK4 when PR priority scheduling is done only at the processors.

PR priority scheduling is at all the resources, relation group 3 is reorganized at low priority for DB size factors greater than 0.1.

6.5.2 Varying the Query Arrival Rates

We varied the arrival rates so that the resulting bottleneck utilization for each workload using the initial DB design changes from 10% to 90%. We observed the same trends as for the experiments when the DB sizes varied. For example, in Table 6.14, the results from executing the NEWNODE rebalancing decision are shown for workload WORK3 using PR priority scheduling at the processors only when the only constraint on A and B is that $A < B$. When we constrain the algorithm to choose reorganizations that result in A being at least 5% as low as B, we obtain Table 6.15. As the arrival rate increases, a choice is made to reorganize a relation group with fewer and smaller relations possibly at a higher priority. For the database used with WORK3, relation group 1 has relation $R1$ with 10^7 tuples and $R5$ with 10^6 tuples, and relation group 5 has relation $R6$ with 10^6 tuples.. For utilizations greater than 30%, group 5 is chosen, and group 5 is also reorganized at an equal priority when the utilization is 90%

The example in Table 6.16 also demonstrates that, as the arrival rates increase, not only does the selected relation group change, but also the reorganization priority changes from low to equal. Table 6.16 shows the results when executing the OLDNODE decision algorithm for

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	58103.2	29002.7	15378.1	29622.9	1181.3	1	EQUAL
0.9	43003.7	21459.9	11394.3	21780.1	1049	1	EQUAL
0.8	31905.7	15904	8310.5	15746.3	920.1	1	EQUAL
0.7	24069.2	12006.7	6143.3	11517.1	794.3	1	LOW
0.6	18182.4	9054.6	4515.4	8391.4	670.4	1	LOW
0.5	13598.5	6751.4	3248.4	5983.6	551.4	1	LOW
0.4	9925.5	4906.8	2251.9	4111.1	434.1	1	LOW
0.3	6946.6	3432.9	1577.9	2864.2	321.2	1	LOW
0.2	4333.5	2144.9	983.6	1775.8	207.2	1	LOW
0.1	3007.4	1498.6	331.1	609.4	54.6	1	LOW

Table 6.12: The best break-even times when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done only at the processors.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	22656	22656	15378.1	15378.1	13565.6	3	LOW
0.9	13635.2	13635.2	11394.3	11394.3	10274.9	3	LOW
0.8	8333.5	8333.5	8310.4	8310.4	7613.7	3	LOW
0.7	5354.7	5354.7	6143.2	6143.2	5688.6	3	LOW
0.6	3524.4	3524.4	4515.3	4515.3	4212.3	3	LOW
0.5	2335.4	2335.4	3248.3	3248.3	3045.1	3	LOW
0.4	1533.8	1533.8	2251.8	2251.8	2117.6	3	LOW
0.3	997.6	997.6	1577.9	1577.9	1489.2	3	LOW
0.2	582.6	582.6	983.5	983.5	934.3	3	LOW
0.1	249.9	249.9	331	331	305.3	2	LOW

Table 6.13: The best break-even times when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done at all the resources.

Util.	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	12626.5	2485.5	2105	4169.4	1599	5	EQUAL
0.7	9123.2	798.1	671.7	1303.6	611.1	5	LOW
0.5	8304.4	488.1	408.3	777.9	385.3	5	LOW
0.3	7780.4	355.1	295.4	552.92	283	5	LOW
0.1	7343.8	282.1	233.4	429.82	225.5	5	LOW

Table 6.14: The best break-even times when varying the bottleneck utilization for the NEWNODE rebalancing decision algorithm for workload WORK3 using PR priority scheduling at the processors only. We only constrain the after reorganization ART to be less than the before reorganization ART.

Util.	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	12626.5	2485.5	2105	4169.4	1599	5	EQUAL
0.7	9123.2	798.1	671.7	1303.6	611.1	5	LOW
0.5	8304.4	488.1	408.3	777.9	385.3	5	LOW
0.3	12556.6	3905.1	295.4	552.93	179.1	1	LOW
0.1	10866.4	3102.8	233.4	429.83	154.9	1	LOW

Table 6.15: The best break-even times when varying the bottleneck utilization for the NEWNODE rebalancing decision algorithm for workload WORK3 using PR priority scheduling at the processors only. We constrain the after reorganization ART to be at least 5% lower than the before reorganization ART.

workload WORK2 with PR priority scheduling at all the resources. We see that the group changes from relation group 1 to 14 as well as the chosen priority changes for higher arrival rates.

Util.	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	161325	65623.7	13740.6	27119.2	4566.8	14	EQUAL
0.7	93000.5	93000.5	7392.4	7392.4	4805.8	1	LOW
0.5	43268.1	43268.1	5092.5	5092.5	3732.4	1	LOW
0.3	24619.4	24619.4	3873.7	3873.7	3042.6	1	LOW
0.1	16153.6	16153.6	3158.0	3158.0	2588.2	1	LOW

Table 6.16: The best break-even times when varying the bottleneck utilization for the OLDNODE rebalancing decision algorithm for workload WORK2 using priority scheduling at all resources.

6.5.3 Varying the Degree of Imbalance

In the OLDNODE rebalancing, the first node initially contains a fraction q of each fully declustered relation's data ($1/N < q < 1$). We varied the value for q to determine the effects of varying the amount of data moved by (and thus the cost of) a reorganization. Arrival rates were derived using $q = 0.9$ instead of $q = 0.7$ as for the other experiments. The results are similar to those in the experiments when the arrival rates and DB sizes were varied. This can be seen in Table 6.17 when PR priority scheduling is only done at the processors using workload WORK4 and Table 6.18 for workload WORK2 when PR priority scheduling is used at all the resources. Only the results for the $High$ arrival rates are shown since the Low and Med rates all result in reorganizing the same groups at a low priority. As the degree of imbalance at the first node increases, and thus the cost of reorganization increases, the chosen reorganization changes from a low priority to an equal priority one.

Also, increasing the degree of imbalance causes the chosen reorganization to rebalance a relation group with fewer and smaller relations. For example, Table 6.19 gives the results when executing the OLDNODE decision algorithm for workload WORK3 using PR priority scheduling at all the resources. At degrees of imbalance higher than fraction 0.2, the choice is to rebalance relation group 3 (which has relation *R3*) as opposed to relation group 2 (which has the larger *R2* relation).

	High Rates						
q	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	191294	62775.6	15622.9	30692.2	8262.2	10	EQUAL
0.8	144519	42902.2	10698.9	20906.3	6389.4	10	LOW
0.7	112821	30509.1	7631.7	14841.6	4959.3	10	LOW
0.6	89087.9	22055.1	5535.1	10711.8	3831.8	10	LOW
0.5	70104.7	15929.2	4006.1	7708.0	2917.6	10	LOW
0.4	54252.3	11279.6	2840.0	5426.3	2161.1	10	LOW
0.3	40684.6	7630.5	1918.6	3630.2	1523.4	10	LOW
0.2	29203.8	4660.8	1170.4	2184.8	977.8	10	LOW
0.1	21314.8	3697.0	550.3	1020	451.8	10	LOW

Table 6.17: Varying the degree of imbalance (q) at the first node for the OLDNODE rebalancing decision algorithm for workload WORK4 when PR priority scheduling is only done at the processors.

	High Rates						
q	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	157302	60082.7	12596.7	24720.9	5103.9	14	EQUAL
0.8	114451	114451	8756.9	8756.9	5306.7	1	LOW
0.7	67158	67158	6306.9	6306.9	4331.6	1	LOW
0.6	41325.7	41325.7	4603.4	4603.4	3342.9	1	LOW
0.5	25982.3	25982.3	3347.6	3347.6	2542.5	1	LOW
0.4	16291.3	16291.3	2381.2	2381.2	1880.1	1	LOW
0.3	9891.6	9891.6	1612.6	1612.6	1322.0	1	LOW
0.2	5488.4	5488.4	985.5	985.5	844.7	1	LOW
0.1	2001.5	2001.5	463.0	463.0	431.1	1	LOW

Table 6.18: Varying the degree of imbalance (q) at the first node for the OLDNODE rebalancing decision algorithm for workload WORK2 when PR priority scheduling is used at all the resources.

	Low Rates						
q	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	4713.4	4713.4	2399.7	2399.7	2187.6	3	LOW
0.8	3634.2	3634.2	1990.6	1990.6	1827.3	3	LOW
0.7	2784.2	2784.2	1632.5	1632.5	1508.1	3	LOW
0.6	2107.7	2107.7	1316.4	1316.4	1223.8	3	LOW
0.5	1567	1567	1037.6	1037.6	970	3	LOW
0.4	1128	1128	788	788	741	3	LOW
0.3	769	769	562.8	562.8	532.9	3	LOW
0.2	4704.3	4704.3	359.3	359.3	307.9	2	LOW
0.1	1860.2	1860.2	173.6	173.6	160.7	2	LOW

Table 6.19: Varying the degree of imbalance (q) at the first node for the OLDNODE rebalancing decision algorithm for workload WORK3 using PR priority scheduling at all resources.

6.5.4 Varying the Time Interval

Since rebalancing each group results in a different DB design after the corresponding reorganization, the ART of the workload after a reorganization changes. Therefore, the gain metric is not linear across the different time intervals for the various possible groups that can be reorganized. We varied the time interval for the decision algorithms to determine how the gain metric values for the chosen reorganizations change.

If the time interval is long enough, more data should be reorganized at a higher priority. For example, in Table 6.20, the results from the NEWNODE rebalancing decisions are given for workload WORK2 for the *Low* and *Med* arrival rates when PR priority scheduling is at all the resources. As the time interval increases, the chosen reorganization changes from one that moves relation group 14 to one that moves all relation groups. At the *Med* arrival rates, the reorganization's priority also changes from low to equal. The reason for this increase in the reorganization's priority and the amount of data that is rebalanced is because the break-even time of the equal priority reorganization is longer than the time for the low priority reorganization and, for time intervals longer than the equal priority rebalancing's break-even time, the gain metric value is greater than the low priority reorganization's. This result applies to executing the OLDNODE decision algorithm as well, as shown in Tables 6.21 and 6.22 when PR priority scheduling is only at the processors or at all the resources, respectively. However, when PR priority scheduling is only at the processors, the move to rebalancing more relations is done at a longer time interval. An example of the reorganization priority increasing as well as the amount of data rebalanced is given in Table 6.23 when executing the OLDNODE decision algorithm for workload WORK3 using PR priority scheduling at all the resources. The priority increases from low to equal priority.

	Low Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
8000	273629	6582.7	865.3	865.3	672.2	14	LOW
16000	1.81821e+06	6582.7	865.3	865.3	672.2	14	LOW
32000	4.90738e+06	6582.7	865.3	865.3	672.2	14	LOW
64000	1.10857e+07	6582.7	865.3	865.3	672.2	14	LOW
128000	2.34424e+07	6582.7	865.3	865.3	672.2	14	LOW
256000	8.30196e+07	98741.5	865.3	865.3	337.4	ALL	LOW
512000	2.18167e+08	98741.5	865.3	865.3	337.4	ALL	LOW
	Med Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
16000	606612	14443.8	1266.4	1266.4	876.6	14	LOW
32000	6.84336e+06	14443.8	1266.4	1266.4	876.6	14	LOW
64000	1.93169e+07	14443.8	1266.4	1266.4	876.6	14	LOW
128000	4.42638e+07	14443.8	1266.4	1266.4	876.6	14	LOW
256000	9.41578e+07	14443.8	1266.4	1266.4	876.6	14	LOW
512000	2.70634e+08	89666.8	1266.4	2426.6	379.3	ALL	EQUAL

Table 6.20: Time interval (seconds) vs. the gain metric values (seconds squared) for the NEWNODE rebalancing decision algorithm for workload WORK2 when PR priority scheduling is at all the resources.

The time interval at which the priority and the relation group changes is affected by the arrival rates in the workload. For higher rates, the time interval at which a change occurs increases compared to the times at the lower rates. This can be seen in Table 6.20 between the *Med* and *Low* arrival rates.

As the arrival rates increase from *Low* to *Med* to *High* rates for the same time interval, the chosen reorganization changes from one at low to equal priority. For example, in Table 6.24, we show the results for the gain metric value chosen by the NEWNODE rebalancing decision algorithm for WORK3, where each reorganization rebalances all relation groups.

	High Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
32000	3.81263e+07	12625.9	6740.5	12389.3	1091.4	1	LOW
64000	2.189e+08	12625.9	6740.5	12389.3	1091.4	1	LOW
128000	5.80446e+08	12625.9	6740.5	12389.3	1091.4	1	LOW
256000	1.30354e+09	12625.9	6740.5	12389.3	1091.4	1	LOW
512000	2.85877e+09	23222	6740.5	12483	618.9	ALL	LOW

Table 6.21: Time interval (seconds) vs. the gain metric values (seconds squared) for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is only at the processors.

	High Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
8000	1.406e+06	4069.4	6740.5	6740.5	6382.8	3	LOW
16000	4.26768e+06	4069.4	6740.5	6740.5	6382.8	3	LOW
32000	8.6253e+07	16731.7	6740.5	6740.5	1091.4	1	LOW
64000	2.67026e+08	16731.7	6740.5	6740.5	1091.4	1	LOW
128000	6.28573e+08	16731.7	6740.5	6740.5	1091.4	1	LOW
256000	1.37799e+09	30898.6	6740.5	6740.5	618.9	ALL	LOW
512000	2.94513e+09	30898.6	6740.5	6740.5	618.9	ALL	LOW

Table 6.22: Time interval (seconds) vs. the gain metric values (seconds squared) for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is at all the resources.

	Med Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
16000	2.67078e+06	8604.27	2839.21	2839.21	2478.09	3	LOW
32000	8.44877e+06	8604.27	2839.21	2839.21	2478.09	3	LOW
64000	2.00048e+07	8604.27	2839.21	2839.21	2478.09	3	LOW
128000	1.00329e+08	37146.2	2839.21	5555.98	624.156	1	EQUAL
256000	3.83856e+08	37146.2	2839.21	5555.98	624.156	1	EQUAL
512000	9.5091e+08	37146.2	2839.21	5555.98	624.156	1	EQUAL

Table 6.23: Time interval (seconds) vs. the gain metric values (seconds squared) for the OLDNODE rebalancing decision algorithm for workload WORK3 when PR priority scheduling is at all the resources.

	Low Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
256000	5.09028e+07	16634.8	342.8	342.8	130.2	ALL	LOW
	Med Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
256000	8.0354e+07	15139.1	510.7	982.4	147.4	ALL	EQUAL
	High Rates						
Time Interval	**Gain Metric**	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
256000	1.60528e+08	29625.1	1005.1	1970.3	169.6	ALL	EQUAL

Table 6.24: Time interval (seconds) vs. the gain metric values (seconds squared) for the NEWNODE rebalancing decision algorithm for workload WORK3 when using PR priority scheduling at all the resources.

6.5.5 Summary

We now summarize the conclusions we derived from our experiments with the rebalancing decision algorithms.

1. The chosen reorganization priority and the amount of data to reorganize are influenced by the arrival rates, the degree of imbalance across the nodes, the DB sizes, and the length of the time interval. This is true when modeling with PR priority scheduling either at all of the resources or just at the processors. As the DB size, degree of imbalance, or arrival rates increases, the decision algorithms choose reorganizations that reap a benefit sooner, which translates to reorganizations of either higher priority or ones that reorganize less data. This is true because the benefit after the higher priority reorganizations and the lower response times for these reorganizations offset the higher intrusion incurred during the reorganizations.

2. As DB sizes, arrival rates, degree of imbalance or time intervals increase, the chosen priority level of a reorganization increases to an equal priority reorganization.

3. High priority reorganizations were always found to have worse break-even times than equal or low priority ones, or their resulting workload ART during the reorganization exceeded the model-limit.

4. When using the gain metric in a decision instead of the break-even time intervals, a longer time interval causes the decision algorithm to choose a reorganization that moves more data at a higher priority. However, the change between relation groups and priorities is done at a longer time interval when PR priority scheduling is used at the processors only as opposed to all the resources.

5. The break-even time interval for a possible reorganization at low priority is higher when PR priority scheduling is only at the processors compared to the break-even times when PR priority scheduling is used at all the resources.

Chapter 7

Conclusions

The proper selection of a physical DB design for a SN DB system leads to low average workload response times. A good design causes the queries to spend less time at the disks, processors, and interconnection network. Our designs involve partitioning, indexing, and clustering. In this dissertation, we studied two related problems: (1) determining an (initial) physical DB design, and (2) changing a physical DB design. To determine an (initial) physical DB design, we developed new and efficient algorithms that base their decisions on the average workload response times resulting from the candidate designs considered. We also created components that assist in solving the problem of reorganizing between designs. By using these components, an automatic physical DB reorganization algorithm can be constructed. Our results and contributions in solving these two problems are described in Section 7.1. Future directions for our work are then discussed in Section 7.2.

7.1 Summary of Results and Contributions

7.1.1 Initial Physical DB Design Decision

For the initial physical DB design decision, the main contribution made in this dissertation is the specification of algorithms to automatically select a design when given information about the DB, workload, and system. The algorithms we created allow for the selection of any of the partitioning, indexing, clustering, or all of these together. We created three new algorithms; one was created to select only the partitioning (IR), and two were created to make all design decisions (CR and PARING). These algorithms contain two new partitioning decisions. One decision determines the selection of partitioning keys. The other decision forms the relation groups. The method used for the IR algorithm basically involves a frequency count of each attribute's usage, and the partitioning decisions are made independently for different relations. To improve on this naive algorithm, we developed an algorithm based on a branch-and-bound strategy (PARING) that was guided by the attributes used in the queries along with an ordering of the queries based on their frequencies of use and their initial estimated response times. We also created an attribute-driven heuristic based on an index selection algorithm (CR). In the PARING and CR heuristics, we use a DB optimizer in order to determine the effects of the design choices on the workload.

In developing the heuristics, we introduced two sets of rules that were used to reduce the number of candidate designs considered by the heuristics: pre-scanning rules and pruning rules. These candidate designs include the specification of both partitioning and indexing keys. Pre-scanning rules are applied to the attributes before the algorithms execute to eliminate keys that are unlikely to result in designs with low average response times. The attributes we eliminated were:

- attributes that were not used at all by any of the queries;
- attributes with low column cardinality (that lead to high data and load skew);
- attributes used in the queries, but not involved in any join; and
- candidate attributes for nonclustering indexes that, when we calculated their cost effects to the workload, do not result in improved performance.

The pruning rules vary with the algorithm being used. For PARING, when it finds a feasible design (D_1), it evaluates the estimated response time (R_1) of the workload using this design. If the current best solution found by the algorithm is R_{min}, and if $R_1 \geq R_{min}$, then PARING can eliminate D_1 and all its related designs. However, to reduce the cost of the algorithm, we have an added pruning rule such that, if $(R_1 \times (1+r)) \geq R_{min}$, we also eliminate D_1 and all its related designs, where r is a factor supplied by the user and $0 \leq r \leq 1$. Since R_1 is the lower bound on response time for D_1 and all its related designs, this equation effectively guarantees that the algorithm finds an R_{min} guaranteed to be no more than $r\%$ from the best response time that could be found by PARING (using $r = 0$), which is the lowest time among all the algorithms. However, CR prunes away candidates based on pre-calculated weighting factors. Since the factors do not consider actual query execution plans and their resulting performance, this pruning does not provide any optimality guarantees, and thus CR generally leads to higher workload average response times than PARING.

To compare different designs, we created a workload cost estimator that evaluates the average workload response time resulting from each design. This estimator uses a queuing network model (QNM) method that includes congestion delays, and makes its estimates by using query plans, the SN architecture structure, the DB characteristics, and the design itself. We verified in Chapter 4 that the workload performance estimates were relatively similar to the results returned from a simulator using a TPC-D DB.

We provided experimental results for the algorithms in Chapter 5. To perform these experiments, we implemented our algorithms as a tool on top of the IBM DB2 PE system. Our experiments were performed using various workloads. For the combinatorial algorithms (PARING and CR) and under various scenarios, such as selecting partitioning alone, clustering and indexing alone, or everything together, we found that communication, I/O, and processing times are reduced compared to the IR algorithm's resulting workload performance. These benefits translate to reductions in response times compared to the IR algorithm. The reductions were seen when varying workload arrival rates, DB sizes, communication latencies, instruction path lengths, and update arrival rates. We also performed experiments with PARING to determine partitioning, clustering, and indexing together. Low response times at low algorithm execution costs (measured by the number of optimizer calls made during an execution) are obtained when partitioning was chosen in a separate execution after indexing is chosen. The separate executions resulted in response times that were similar to times when partitioning and indexing were selected together in the same execution of the algorithm. However, the separate executions had a much lower cost. In our experiments, PARING resulted in designs leading to average response times that were either the same or lower than the times from CR. PARING also results in a number of optimizer calls that were sometimes much lower than CR's because there is more pruning done by PARING than for CR. We also found that increasing the tolerance ratio r reduces the number of optimizer calls for PARING. From our investigations, we conclude that the PARING algorithm should be the method of choice since it leads to lower workload average response times with a moderate cost of executing the algorithms.

From our experiments with PARING, we noticed that PARING's cost can be reduced for a number of reasons when changing some system environment parameter. First, PARING may find designs that cause the workload to saturate the system. For some parameters, such as arrival rates or resource latencies, increasing these values cause the system to be more heavily utilized, which leads to more designs that cause the workload to saturate the system. Second, when the partial cost for a subset of queries is close to the workload's average response time, and thus the bound in PARING's branch-and-bound strategy, many designs and the designs they generate are pruned away. We concluded from the experiments with PARING that it is a flexible algorithm and it results in good performance under many different system environments.

7.1.2 Concurrent Physical DB Reorganization

To solve the reorganization problem, we defined three components required to compare different reorganization strategies and their resulting designs: (1) the reorganization cost estimator to estimate the performance of reorganizations and workloads together; (2) a measure of a reorganization's performance effects on a workload (called the *(net) gain metric*); and (3) the determination of a strategy's priority level with respect to the workload transactions' priorities. The gain metric is used to combine a strategy's benefit (the resulting performance improvement after a strategy completes) and cost (the intrusion on workload performance during the strategy's execution) into one value that is useful for comparisons between strategies. We also investigated two types of reorganizations based on our components.

We estimated the performance of executing a concurrent reorganization strategy with a workload by using another QNM model. This model represented the reorganization activity as a batch class executing with the workload transactions, which were modeled as QNM transaction classes. Reorganization intrusion on the workload was included in the form of congestion delays, where the effects of scheduling the reorganization strategy and workload with priorities were also included. To solve the QNM, we developed a technique in Chapter 6 based on an aMVA priority approximation method.

Output from this estimator allowed us to determine values for our reorganization *(net) gain* metric. This metric compares the average response time performance with a specific reorganization strategy to the performance without any reorganization. Thus, we can compare different reorganization strategies that possibly result in different physical DB designs.

The calculation of the gain is based on a given time interval. As was seen in the priority experiments of Chapter 6, the time at which a reorganization's benefit starts to outweigh its cost is considered to be the best (break-even) time interval for the reorganization. We derived and used this best (lowest) time to compare the reorganizations. Also, when the time interval is fixed, we chose the best reorganization to be one that has the highest value for the gain metric.

We developed a decision algorithm to determine the best new design and reorganization to choose with the constraint that reorganizations considered by the algorithm only rebalance a data placement.

From our reorganization priority studies in Chapter 6, we have challenged the assumption in the literature that a low priority for the reorganization is always best. A reorganization should be given an equal priority compared to the workload's when the system is heavily utilized or when there is a large amount of data to reorganize. The reason for this is that the performance of the workload with the old design is so much worse than with the new design that a quick reorganization benefits the system greatly.

Some other conclusions were made from the experimental studies. We saw that the differences between reorganizations with different priorities were more pronounced when preemptive scheduling was used at all the resources as opposed to just the processors. We also saw that increasing the time interval causes a decision algorithm to choose a reorganization of higher priority that reorganizes more data as opposed to choices for shorter time intervals.

7.2 Future Directions

We have provided methods to select and adjust physical DB designs. There are several ways we could expand upon our work. The next few sections will suggest future directions for this research.

Enhance the Physical DB Design Algorithms

In constructing our algorithms, we made various assumptions. However, we could relax some of these assumptions to make the algorithms more robust.

1. We could allow global indexes to be selected. Since the placement of the index pages would not directly correspond to the placement of the relation to which they refer, the placement of the global index must be taken into account by our algorithms.

2. Index types besides B-trees could be considered, and this would affect the number of indexing cases determined by an algorithm and the cost estimations required for these types.

3. Other partitioning functions besides hash partitioning can be considered. For example, with range partitioning, the cost estimations may change because of skew resulting from the ranges constructed. Also, the skew that occurs in hash partitioning could be handled more carefully in the cost estimations.

4. Different data placement algorithms may be useful, e.g., when the nodes are heterogeneous.

5. Replication of relations can be added. However, previous work such as Hsaio's chained declustering techniques that include replication with data placement have demonstrated that the replication decisions can be applied after the data placement is chosen [Hsa90].

The first four additions would present more cases for an algorithm to consider, and hence, a selection algorithm with intelligent pruning, such as the ones based on a branch-and-bound strategy, would be even more beneficial.

Build upon our Reorganization Components and Studies

Extensions to our reorganization work include the following:

1. We defined some important components required for the reorganization decision. More robust reorganization decision algorithms could be constructed to use these components. Such algorithms would consider more types of reorganization strategies than we did in our studies of Chapter 6. We have shown some viable algorithms to determine reorganizations that involve the previous physical DB design algorithms to select the new physical DB design. However, it would be necessary to implement them.

2. Include the costs related to using a differential file while a reorganization executes. Because a low priority reorganization executes for longer times than when using the other priority (and off-line) reorganizations, the number of updates in a differential file may be greater, and thus the costs related to these files will cause the reorganization's costs to increase at a higher rate than for the other reorganizations.

3. Construct and study incremental reorganizations.

Improve Analytic Estimations

As seen in Chapter 4, the cost estimators give workload response times that are relatively similar to those times from a simulator. However, we could improve the estimations in a number of ways.

1. We could include the SQL operations that we ignored in this thesis. For example, we need to include the estimation of the execution for subqueries and set operations (such as $UNION$ and $INTERSECT$). We could also include *compound SQL* statements. A compound SQL statement's average response time could be calculated by summing the component SQL statement response and residence times.

2. We made simplifying assumptions that each node can be modeled using a single processor and disk drive. We can expand upon this to allow for SMP nodes and multiple independent disks or a disk array at each node. With multiple disks, we may need to add another decision to the physical DB design decision related to placing all the partitions at the same node across its disks. Since we abstract our model so that the interconnection network is represented only by its communication speed and each node can be represented only by its processing power and I/O capacity and speed, these rates may be estimated from the detailed node and network characteristics, which may include information about multiple disks and multiple processors in an SMP or information about an ATM switch. This allows our work to be applicable to systems in which each node is more complex than one with a single processor and disk, and to systems in which the interconnection network is more complex than a bus.

3. Since we assumed the DB system was a homogeneous SN system, we could change our model to estimate the performance for the queries on a heterogeneous SN system. Our model does allow for such calculations already since we assumed each resource of each node was a separate server. Each server could be given its own latencies and other characteristics such as scheduling policies.

4. In our estimations, we used simple estimates for buffering and skew, and we ignored the delays resulting from concurrency control, data skew, load skew, recovery, logging, and

buffer management. More accurate calculations of these (such as the ones derived by Hyslop [Hys91]) would provide results closer to real query execution times. We included only congestion delays. This inclusion allows us to prune away designs along with any of their related designs that may result in the workload saturating the system. Estimating these extra delays (e.g., data contention delays due to concurrency control) would result in some designs leading to high response times in our search that may also be prunable or infeasible.

5. We could improve on the calculation of the reorganization gain metric by including the estimation of the gradual build-up of delay to the workload as the reorganization begins and the extra delay incurred as the backed-up transactions during a reorganization are processed after a reorganization completes. This can be done by using transient (rather than steady-state) queuing network models. A good estimation method for these delays will allow us to obtain accurate response times for off-line and high priority reorganizations.

Address Other Performance-Related Decisions

There are other decisions besides the ones for physical DB design that the DBA must make. The decisions of interest are ones required when a DBMS is installed on a SN system and ones that affect performance. We assumed these decisions are made before our algorithms are used. However, the decisions can use our cost estimator to guage the performance results of each of the choices. We could also have the algorithms determining these decisions use our branch-and-bound style algorithm as a submodule to broaden the scope of our decisions. The possible decisions include:

1. Which views should be materialized? (A materialized view, if input to our algorithms, could be treated as a special relation that could have its partitioning, indexing, and clustering chosen by our algorithms.)

2. How should the DB relation be normalized? (This relates to how the relations should be vertically partitioned.)

3. What is the maximum buffer space that should be allocated to each operator executing on the SN system?

4. What values should be used for the lock granularity parameters? (For example, one parameter could be the maximum number of row-level locks that can be used on a relation before the DB engine will escalate the locks to a single table lock.)

Bibliography

[AF77] James E. Ames and Derrell Foster. Dynamic file assignment in a star network. In *Proc. of the Computing Networks Symp.*, pages 36–39, Gaithersburg, Maryland, December 1977.

[Ape88] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Trans. on Database Systems*, 13(3):263–304, September 1988.

[BAC$^+$90] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[BFG95] Chaitanya K. Baru, Gilles Fecteau, and Ambuj Goyal. DB2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322, February 1995.

[BKL$^+$84] Raymond M. Bryant, Anthony E. Krzesinski, M. Seetha Lakshmi, , and K. Mani Chandy. The MVA priority approximation. *ACM Trans. on Computer Systems*, 2(4):335–359, November 1984.

[BKT83] R. M. Bryant, A. E. Krzesinski, and P. Teunissen. The MVA pre-empt resume priority approximation. In *Proc. of the ACM Int. Conf. on Measurement and Modeling of Computer Systems, ACM SIGMETRICS*, pages 12–27, Minneapolis, MN, 1983.

[BLS95] Anna Brunstrom, Scott T. Leutenegger, and Rahul Simha. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. Technical Report NASA Contractor Report 195024, ICASE Report No. 95-2, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, January 1995.

[BPS90] Elena Barucci, Renzo Pinzani, and Renzo Sprugnoli. Optimal selection of secondary indexes. *IEEE Trans. on Software Engineering*, 16(1):32–38, January 1990.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in Bubba. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 99–108, Chicago, June 1988.

[Cas86] Ignacio R. Casas. Prohpet: A layered analytical model for performance prediction of database systems. Technical Report CSRI-180, PhD Thesis, Dept. of Computer Science, University of Toronto, April 1986.

[CBC93] Sunil Choenni, Henk M. Blanken, and Thiel Chang. On the selection of secondary indices in relational databases. *Data & Knowledge Engineering*, 11(3):207–233, 1993.

[CC93] Liz Chambers and Dave Cracknell. Parallel features of NonStop SQL. In *Int. Conf. on Parallel and Distributed Information Systems*, pages 69–70, San Diego, January 1993.

[CFM95] Alberto Capara, Matteo Fischetti, and Dario Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):955–967, December 1995.

[CK92] Felipe Carino Jr. and Pekka Kostamaa. Exegesis of DBC/1012 and P-90 - industrial supercomputer database machines. In *Int. PARLE'92 Conf. Parallel Architectures and Languages Europe*, Paris, June 1992. Springer-Verlag.

[Cla93] D. Clay. Informix Parallel Data Query. In *Int. Conf. on Parallel and Distributed Information Systems*, pages 71–72, San Diego, January 1993.

[CLYY92] Ming-Syan Chen, Mingling Lo, Philip S. Yu, and Honesty C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proc. of the 18th Int. Conf. on VLDB*, pages 15–26, Vancouver, August 1992.

[CM83] John V. Carlis and Salvatore T. March. Computer-aided physical database design methodology. *Computer Performance*, 4(4):198–214, December 1983.

[CN97] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL Server. In *Proc. of the 23th Int. Conf. on VLDB*, pages 146–155, Athens, August 1997.

[CNP82] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 128–136, Orlando, FL, June 1982.

[CNW83] Stefano Ceri, Shamkant Navathe, and Gio Wiederhold. Distribution design of logical database schemas. *IEEE Trans. on Software Engineering*, 9(4):487–504, July 1983.

[Com78] Douglas Comer. The difficulty of optimum index selection. *ACM Trans. on Database Systems*, 3(4):440–445, December 1978.

[Cor96] IBM Corporation. DB2 Parallel Edition for AIX: Administration guide and reference, 1996.

[DF82] Lawrence W. Dowdy and Derrell V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

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

[DGS$^+$90] David J. DeWitt, Shahram Ghandeharizadeh, D. A. Schneider, A. Bricker, Hui-I Hsaio, and R. Rasmussen. The GAMMA database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[EL88] Derek L. Eager and John N. Lipscomb. The AMVA priority approximation. *Performance Evaluation*, 8:173–193, 1988.

[Fle87] R. Fletcher. *Practical Methods of Optimization: 2nd Edition.* John Wiley and Sons, 1987.

[FM89] Christos Faloutsos and Dimitrios Metaxas. Declustering using error correcting codes. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 253–258, Philadelphia, March 1989.

[FON92] Martin R. Frank, Edward R. Omiecinski, and Shamkant B. Navathe. Adaptive and automated index selection in RDBMS. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 277–292, Vienna, Austria, March 1992.

[FST88] S. Finkelstein, Mario Schkolnick, and Paulo Tiberio. Physical database design for relational databases. *ACM Trans. on Database Systems*, 13(1):91–128, March 1988.

[GD90] Shahram Ghandeharizadeh and David DeWitt. Hybrid-range partitioning: A new declustering strategy for multiprocessor database machines. In *Proc. of the 16th Int. Conf. on VLDB*, pages 481–492, Brisbane, Australia, August 1990.

[GGS96] Sumit Ganguly, Ashkay Goel, and Avi Silberschatz. Efficient and accurate cost models for parallel query optimization. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 172–181, Montreal, June 1996.

[Gha90] Shahram Ghandeharizadeh. Physical database design in multiprocessor database systems. Technical Report #964, PhD Thesis, Computer Sciences Dept., Univ. of Wisconsin-Madison, September 1990.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proc. of the Int. Conf. on Data Engineering*, pages 208–219, Birmingham, U.K., April 1997. IEEE Computer Society Press.

[GJ78] Michael R. Garey and David S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W.H. Freeman and Co., 1978.

[Gra93] Goetz Graefe. Options in physical database design. *ACM SIGMOD Record*, 22(3):76–83, September 1993.

[GS90] Bezalel Gavish and Olivia R. Liu Sheng. Dynamic file migration in distributed computer systems. *Comm. of the ACM*, 33(2):177–189, February 1990.

[Hab90] Franz Haberhauer. Physical database design aspects of relational DBMS implementations. *Information Systems*, 15(3):375–389, 1990.

[HCL$^+$90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. Starburst Mid-Flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HL90] Kien A. Hua and Chiang Lee. An adaptive data placement scheme for parallel database computer systems. In *Proc. of the 16th Int. Conf. on VLDB*, pages 493–506, Brisbane, Australia, August 1990.

[HM94] Waqar Hasan and Rajeev Motwani. Optimization algorithms for exploiting the parallelism–communication tradeoff in pipelined parallelism. In *Proc. of the 20th Int. Conf. on VLDB*, pages 36–47, Santiago, Chile, September 1994.

[Hsa90] Hui-I Hsaio. Performance and availability in database machines with replicated data. Technical Report #963, PhD Thesis, Computer Sciences Dept., University of Wisconsin-Madison, August 1990.

[Hys91] William F. Hyslop. Performance prediction of relational database management systems. Technical Report CSRI-254, PhD Thesis, Dept. of Computer Science, University of Toronto, September 1991.

[ISR83] Maggie Y. L. Ip, L. V. Saxton, and Vijay V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. on Software Engineering*, 9(2):135–143, March 1983.

[Jak80] Matti Jakobsson. Reducing block accesses in inverted files by partial clustering. *Information Systems*, 5(1):1–5, 1980.

[LKB87] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. In *Proc. of the ACM Int. Conf. on Measurement and Modeling of Computer Systems, ACM SIGMETRICS*, pages 69–77, May 1987.

[LW66] Eugene L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, pages 699–719, July 1966.

[LZGS84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance.* Prentice-Hall, 1984.

[MD97] Manish Mehta and David J. Dewitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, January 1997.

[MS78] Salvatore T. March and Dennis G. Severance. A mathematical modeling approach to the automatic selection of database designs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 52–65, Austin, TX, May 1978.

[NHS84] Jurg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.

[OLS92] Edward Omiecinski, Liehuey Lee, and Peter Scheuermann. Concurrent file reorganization for record clustering: A performance study. In *Proc. of the Int. Conf. on Data Engineering*, pages 265–272, Tempe, Arizona, February 1992. IEEE Computer Society Press.

[Omi88] Edward Omiecinski. Concurrent storage structure conversion from B+tree to linear hash file. In *Proc. of the Int. Conf. on Data Engineering*, pages 589–596, Los Angeles, February 1988. IEEE Computer Society Press.

[OPSS89] Salvatore Orlando, V. Perri, S. Scrivano, and W. Staniszkis. Database analyzer and predictor - an overview. In *Proc. of the Int. Conf. on Data Engineering*, pages 625–634, Los Angeles, February 1989. IEEE Computer Society Press.

[OV91] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[Pad92] Sriram Padmanabhan. *Data Placement in Shared-Nothing Parallel Database Systems*. PhD thesis, EECS Dept., Univ. of Michigan-Ann Arbor, 1992.

[PB92] Sriram Padmanabhan and Chaitanya Baru. Data placement in shared-nothing parallel database systems. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 345–352, Baltimore, 1992.

[PBS97] Eric W. Parsons, Mats Brorsson, and Kenneth C. Sevcik. Predicting the performance of distributed virtual shared memory applications. Technical Report CSRI-353, Dept. of Computer Science, University of Toronto, January 1997.

[PR88] R. Gary Parker and Ronald L. Rardin. *Discrete Optimization*. Academic Press, 1988.

[Raa95] Francois Raab (editor). TPC benchmark D (decision support) working draft 9.0. Technical report, Transaction Processing Council (TPC), January 1995.

[RE78] Daniel Ries and Robert Epstein. Evaluation of distributed criteria for distributed database systems. Technical Report UCB/ERL M78/22, University of Berkeley, May 1978.

[RM93] Erhard Rahm and Robert Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *Proc. of the 19th Int. Conf. on VLDB*, pages 182–193, Dublin, Ireland, 1993.

[RND77] Edward Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977.

[RS88] Pasquale Rullo and Domenico Sacca. An automatic physical designer for network model databases. *IEEE Trans. on Software Engineering*, 14(9):1293–1306, September 1988.

[RS91] Steve Rozen and Dennis Shasha. A framework for automating physical database design. In *Proc. of the 17th Int. Conf. on VLDB*, pages 401–411, Barcelona, Spain, September 1991.

[RVVN90] Pedro I. Rivera-Vega, Ravi Varadarjan, and Shamkant B. Navathe. Scheduling data redistribution in distributed databases. In *Proc. of the Int. Conf. on Data Engineering*, pages 166–173, Los Angeles, February 1990. IEEE Computer Society Press.

[Sch94] Herbert Schwetman. *CSIM17 Users' Guide*. Mesquite Software, Inc., 1994.

[SD95] Rick Stellwagen and David DeWitt. Configurator for the SYBASE parallel DB system, January 1995. (Information obtained through correspondance with the Configurator implementers and maintainers: Rick Stellwagen of NCR in San Diego and David DeWitt of the Univ. of Wisconsin at Madison).

[Ser84] Alan Eldin Ismail Serry. An analytical approach to modeling IMS systems. Technical Report CSRI-161, PhD Thesis, Dept. of Computer Science, University of Toronto, July 1984.

[Sev81] Kenneth C. Sevcik. Data base system performance prediction using an analytic model. In *Proc. of the 6th Int. Conf. on VLDB*, pages 182–198, 1981.

[SG79] Gary H. Sockut and Robert P. Goldberg. Database reorganization – principles and practice. *ACM Computing Surveys*, 11(4):371–395, December 1979.

[Shn73] Ben Shneiderman. Optimum data base reorganization points. *Comm. of the ACM*, 16(6):362–365, June 1973.

[SI93] Gary H. Sockut and Balakrishna R. Iyer. Reorganizing databases concurrently with usage: A survey. Technical Report TR 03.488, IBM Santa Teresa Laboratory, San Jose, CA, June 1993.

[SL91] Bernhard Seeger and Per-Ake Larson. Multi-disk B-trees*. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 436–445, Denver, May 1991.

[Smi90] Gary S. Smith. Online reorganization of key-sequenced tables and files. Technical Report Tandem Systems Review, Tandem Computers Inc., October 1990.

[Soc78] Gary H. Sockut. A performance model for computer data-base reorganization performed concurrently with usage. *Operations Research*, 26(5):789–804, September 1978.

[Sri92] Venkatachary Srinivasan. *On-Line Processing in Large-Scale Transaction Systems.* PhD thesis, Dept. of Comp. Science, Univ. of Wisconsin-Madison, Madison, WI, 1992.

[TO78] Toby J. Teorey and Lewis B. Oberlander. Network database evaluation using analytical modeling. In *National Computer Conference and Exposition: AFIPS Conf. Proc.*, pages 833–842, Anaheim, CA, June 1978.

[Tro96] Jim Troisi. NonStop SQL/MP availability and database configuration operations. *IEEE Bulletin of the Technical Committee on Data Engineering*, 19(2):12–18, June 1996.

[UAC$^+$97] Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Dick Tsur. A system for managing query flocks, May 1997. (This information can be obtained at: http://www-db.stanford.edu/ ullman/pub/flocks.html).

[VBW95] Radek Vingralek, Yuri Breitbart, and Gerhard Weikum. SNOWBALL: Scalable storage an networks of workstations with balanced load. Technical Report Tech. Report 260-95, Dept. of Comp. Science, University of Kentucky, 1995.

[Wah84] Benjamin Wah. File placement in distributed computer systems. *IEEE Computer*, 17(1):23–33, January 1984.

[Wha87] Kyu-Young Whang. Index selection in relational databases, 1987. From book: Foundations of Data Organization, edited by S. P. Ghosh, Y. Kambayashi, and K. Tanaka.

[Wol89] Joel L. Wolf. The placement optimization program: A practical solution to the disk file assignment problem. In *Proc. of the ACM Int. Conf. on Measurement and Modeling of Computer Systems, ACM SIGMETRICS*, pages 1–10, May 1989.

[WWS85] Kyu-Young Whang, Gio Wiederhold, and Daniel Sagalowicz. The property of separability and its application to physical database design, 1985. From the book: Query Processing in Database Systems, edited by: Wom Kim, David S. Reiner, and Don S. Batory.

[WZS91] Gerhard Weikum, Peter Zabback, and Peter Scheuermann. Dynamic file allocation in disk arrays. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 406–415, Denver, May 1991.

[YDT76] S. Bing Yao, K. Sundar Das, and Toby J. Teorey. A dynamic database reorganization algorithm. *ACM Trans. on Database Systems*, 1(2):159–174, June 1976.

[ZJP94] Daniel C. Zilio, Anant Jhingran, and Sriram Padmanabhan. Partitioning key selection for a shared-nothing parallel database system. Technical Report RC 19820 (87739) 11/10/94, IBM T. J. Watson Research Center, 1994.

Appendix A

Database Operator Cost Estimation

This appendix describes the operator cost calculations and assumptions required for the workload and reorganization cost estimation. The calculations derive the visit counts for the operators in a query plan, and these counts are used to calculate the average resource usages and response times for the workload queries.

Section A.1 defines QNM terminology that is required in the definition of the workload and reorganization cost models. Section A.2 provides some examples of SQL queries and their associated query plans that are input into our cost estimator. In Section A.3, the cost calculations are specified for each operation. Some of our formulas are similar to ones given in the thesis by Padmanabhan [Pad92]. We went beyond these calculations to include such costs as the per node resource usage, the index calculations, the join executions, the column cardinality estimations, and the workload cost formulation.

A.1 Queuing Network Model Terminology

In this section, we describe some queuing network model (QNM) terminology. This terminology is used in the descriptions of the workload cost estimation methods in Chapter 4 and in Chapter 6, since the estimators are based on QNM's. Queuing network models and the associated general estimation methods are described in the literature [LZGS84].

In our model, we have a DB system that is utilized by SQL classes (queries and updates). An *SQL class* is basically a set of queries or updates all having a similar structure, and the costs for the class can be taken as an average over the class. In QNM terminology, each of these classes must be represented by a QNM class. There are two types of QNM classes of interest to this thesis: transaction and batch classes. A transaction class is a set of similar transactions having an arrival rate specified as transactions per unit time. Each transaction in the class enters the system, executes, and then leaves the system after it has completed. We model each of the SQL classes as a QNM transaction class. However, a QNM batch class has a fixed number of active jobs, and, once a job has completed its service, another (statistically identical) job immediately begins being serviced by the system so that the number of active jobs in the class remains constant over time. The average response time of the class will include the average congestion delays from the other classes in the system. This on-average congestion is important for estimating the response times of our reorganization strategies as well as the response times of workloads executing during a reorganization. We will model reorganization processes with a batch class in which just a single customer is active.

Associated with a transaction class is its arrival rate (f_i). The times between arrivals are assumed to be exponentially distributed. When calculating the utilizations of an operator in a transaction class, we assume the operator's arrival rate is the same as that of the class's.

In QNM's, the interconnection network, processors, and disks of a parallel shared-nothing system are each called a *resource* of the QNM. The *service time* ($S_{i,r}$) for a transaction class i at a resource r is the time the resource is busy serving each transaction on each use of the resource. The average number of requests for a resource r from transaction i is called the transaction's *visit count* ($V_{i,r}$). When we multiply a transaction's service time with its visit count for a resource, we obtain the transaction's *service demand* ($D_{i,r} = V_{i,r}S_{i,r}$). If the queuing network has certain properties (such as homogeneous service times and homogeneous routing, which are approximately satisfied in real systems), the steady state behavior of the network is characterized by a *product form* solution [LZGS84], which can be exploited to efficiently obtain performance measures for the model.

Transactions of class i utilize resource r for a percentage of time called the *utilization* ($U_{i,r}$). The total utilization of a resource (U_r) is the sum of the utilizations from each class. These utilization values indicate how heavily loaded the resources are and how much congestion delay different classes present to each other. If any resource r is utilized for 100% of the time, the system is said to be overloaded or *saturated*. Otherwise, the system is said to be *unsaturated*. If a resource is saturated, it cannot keep up with the demands placed upon it, and so the number of requests queued waiting for service at the resource will grow indefinitely.

When the congestion delays for a transaction i are added to the service demand at a resource r, the resulting time is termed the *residence time* ($R_{i,r}$) for the transaction. The residence times are used to form the total time a transaction spends in the system, and we call this total time the transaction's response time (R_i). One estimation method for calculating the utilizations, residence times, queuing, and response times for classes on a system is called the approximate MVA method (aMVA) [LZGS84]. We modify this aMVA method for our workload estimation purposes.

Congestion delays are dependent on the scheduling used at each of the resources and the priority of the different classes. A first-come-first-serve (FCFS) scheduling implies that priorities between classes do not affect residence times. However, if the scheduling is based on priorities and is *pre-emptive*, classes with higher priority will cause lower priority transactions to be placed

```
ORG(DEPTNUMB,LOCATION,MANAGER,COMPANY)

STAFF(ID,NAME,DEPT,ADDRESS)
```

Figure A.1: Example DB Schema.

back in a resource's queue while this high priority class executes. Thus, estimated residence times for a higher priority class do not include the congestion from lower priority classes, but the times for lower priority classes include delays from the execution of higher priority classes. If the priority scheduling is based on priorities but is *non-preemptive*, newly arriving high priority transactions must wait for the current transaction being serviced to finish, whether it is of high or low priority. Hence, both high and low priority transactions can incur delay from each other.

A.2 Example SQL Queries and Their Associated Query Plans

In this section, we give some example SQL queries and their associated query execution plans. We use the DB schema given in Figure A.1. The first relation is ORG and has four attributes: DEPTNUMB, LOCATION, MANAGER, and COMPANY. The second relation, STAFF, also has four attributes. We do not describe any specific placement information, but we do assume that the partitioning keys of ORG and STAFF are DEPTNUMB and ID, respectively.

The first example is an insert given in Figure A.2. In this query we see that the tuples of STAFF are used to insert tuples into ORG. Specifically, the tuples with values for ID equal to 20 are used. Since ID is the partitioning attribute of STAFF, the execution of this query is done at only one of the nodes storing STAFF tuples. We also note that the value for DEPT is used as the value for ORG's DEPTNUMB on insert. This means that the insert will be on the node that has the partition with this DEPTNUMB value. Since the node (M) for this DEPTNUMB value does not correspond to the node where the tuples are generated, the tuples must be communicated to the relevant node (M) where the insert into ORG must occur. The query plan is given in Figure A.3, where the insert operator has two children. The left child is the base relation that is modified while the right child represents the query producing the tuples to be inserted. Tuples are selected from the STAFF relation by using the SEL operator. The SEL operator in this query plan specifies single node execution as does the INSERT operation. Tuples are moved to the place where they are to be inserted by the communication operations SEND and RECV.

Another type of query involves a join, sort, and aggregation as seen in Figure 2.10. In this query, the join attribute for STAFF is its partitioning attribute. This information is made use of in the query plan in Figure 2.11 to cause the execution of the join to be at the nodes where

```
INSERT INTO ORG
SELECT DEPT, "TORONTO", ID, "IBM"
FROM STAFF
WHERE ID = 20
```

Figure A.2: Example of insert query.

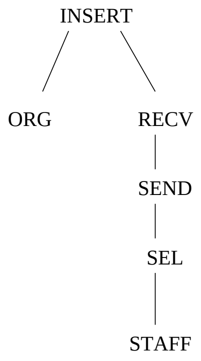


Figure A.3: Example of insert query plan.

the inner relation, STAFF, resides. Also, the ORG tuples are directed to the nodes containing partitions of STAFF with the corresponding join attributes. This query plan requires only a single relation to be communicated. We also note that the query contains an aggregation. In the optimized plan, the aggregation is done locally on the sorted result, and the result is sent to the coordinating node where a global aggregation is done. The output of the join operation is sorted locally and an aggregation is done locally. Localized aggregated results from each node executing the aggregation are then sent to the coordinator and aggregation is done for all the inputs.

An example of a simple selection is given in Figure A.4. In this figure, we notice that the predicate selecting tuples with MANAGER equal to 10 must be done on all nodes storing ORG because MANAGER is not the partitioning attribute. If we assume that MANAGER has an index, then an index scan on ORG is possible. The query plan for the index scan is given in Figure A.5. In the plan, the result after the selection on ORG is communicated to the coordinating node.

```
SELECT *
FROM ORG
WHERE MANAGER = 10
```

Figure A.4: Example of an index scan.

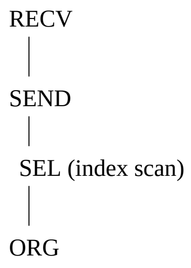


Figure A.5: Example of an index scan plan.

Variable	**Description**
$\{X\}$	Cardinality of X (function)
$P(X)$	Number of data pages used for X (function)
$tup(X)$	Tuple size of X (function)
$\|X\|$	Number of bytes for X (function)
$P_m(X)$	Number of message pages used for X
R_i	Partition of relation R at node i
R	Input relation to an operator
S	Input relation to an insert, delete, or update operator
Q	Output relation from an operator
$ColCard_j$	Column cardinality of attribute j at all nodes
$ColCard_{i,j}$	Column cardinality of attribute j at node i

Table A.1: DB related cost variables.

A.3 Detailed Operator Cost Calculations

Before discussing the detailed cost calculations of each operator, we must define some variables that will be used throughout this section. The variables used for DB information, that apply to temporary relations also, are shown in Table A.1. These variables include defining the cardinality of a base or temporary relation at a specific node. This cardinality at a node represents a portion of the placement information for the relation, and is used as input and output for an operator calculation. The variables representing calculated values are defined in Table A.2. We also use the parameters defined in Tables 2.1 and 2.2. Another parameter is *OperatorPages* that defines the maximum amount of memory any query operation can use. We use 256 pages as the default. For certain operations, such as sorting, this memory limit could force an external sort to be executed rather than an in-memory sort. The external sort requires more disk accesses and leads to an increase in a query's response time.

In our operator cost calculations, we have some calculations that are repeated per operator

Variable	**Description**
V^x_{s,IO_k}	Visit count for disk at node IO_k by query s and operator x Represents the number of I/O accesses
V^x_{s,CPU_k}	Amount of CPU time at node CPU_k by query s and operator x
$V^x_{s,COMM}$	Number of network messages by query s and operator x
$KeysPerPage$	Max. number of unique index key values
$Leafs_{i,j}$	Number of leaf pages for index on attribute j at node i
$Height_{i,j}$	Height of index on attribute j at node i
$NumSearches$	Num. of searches from root to leaf pages for an index scan
$NumLeafs$	Num. of index leaf pages accessed for index scan
$NumLeafKeys$	Num. of indexing keys causing access to base relation
$TupsPerValue$	Avg. tuples per attribute value of an attribute
$NumSimplePreds$	Number of selection predicates on a base relation
$NumPtrPages$	Num. of pointer pages accessed when index on a nonclustering key
$NumPtrPagesPerValue$	Num. of pointer pages per attribute value in index
$NumPtrPerPage$	Num. of base relation pointers in a pointer page
$NumRecvs$	Number of receivers for a RECV operation
$NumSends$	Number of senders for a SEND operation
$NumInnerScans$	Number of scans on the inner (right) join child
$NumJoinPreds$	Num. of join predicates to compare tuples from each join child
$NumSortTuples$	Num. of tuples that can be sorted in memory
$NumRuns$	Num. of runs required for external sort
$keySize$	Size in bytes of the key used to sort
$numKeys$	Num. of attrs used as sort key
$KeysInMem$	Num. of pages of a run and its keys fitting in memory
$NumMerges$	Num. of iterations in merge phase of sort
$NumAggPreds$	Num. of aggregation operations
$NumSplits$	Num. of page splits and merges of index on insert/update/delete

Table A.2: Cost variables.

description. The repeated costs are incurred by all operators except an insert, delete, or update. One of these costs is the cost required to package an operator's result tuples in memory. This cost is added to V^x_{s,CPU_k} by multiplying the number of output tuples from the operator at each node by the instruction path length for result creation, I_{RESULT}. Another cost of interest is the cost to initiate and terminate the operator execution at each node. This initiation and termination is also added to V^x_{s,CPU_k} using the instruction path lengths I_{START} and I_{STOP}, respectively. We will leave these out of the formulas below.

Our formulas have simplified the CPU costs by usually leaving out the costs due to I/O. We use the following calculations for the CPU required in performing I/O:

$$V^x_{s,CPU_k} = \begin{cases} V^x_{s,IO_k} \times \mathit{ReadSize} \times I_{IO} & \text{if sequential read/write} \\ V^x_{s,IO_k} \times I_{IO} & \text{otherwise} \end{cases} \tag{A.1}$$

where x is an operation. *ReadSize* is used here because our sequential I/O is done per sequentially read block of disk pages, and this block is stored together in the disk cache. However, on each access of a disk page, a processor cost is incurred whether the page was cached or not. We also leave out communication counts since they are assumed to be zero when not specified.

A.3.1 Selections (Scans)

A selection on a relation is always done by scanning in a relation stored on disk by the query plan operator SEL. Scans execute using two different techniques: file and index scans. The file scan can be modeled as a sequential scan through all the pages containing tuples of a relation. The relation in a file scan can be either a base or temporary relation. An index scan can only be performed using an existing index on a base relation.

Before giving the formulas for selection, we give the formulas to write out temporary relations to disk. Temporary relations are used to store an operator's output when the output cannot be pipelined to its parent operator. A sequential relation scan is used to read the temporary relations back. The creator of the relation is the child of the SEL operator. In our cost estimations, this child operator will have these costs assigned to it. These costs depend on the number of tuples and pages output per node and the tuple size of the temporary relation. We assume that each written page incurs one I/O access per *ReadSize* block of pages for the number of pages to write that exceed *OperatorPages*. The cost formula for writing the temporary relation at node i is given in the calculations below. We assume that the output relation from an operator is Q and its partition at node i is Q_i.

$$V^{temp\ write}_{s,IO_i} = \lceil \max(0, P(Q_i) - \mathit{OperatorPages})/\mathit{ReadSize} \rceil \tag{A.2}$$

In calculating the cost of a file scan for a relation, we assume that the relation is read sequentially using blocks with *ReadSize* pages. In performing a file scan, only the I/O and processor costs are incurred. The calculations are given below. On any scan, a set of predicates would be performed on the relation's tuples. We assume that the number of simple predicates (i.e., non-subquery predicates) in a scan is given by $NumSimplePreds$.

$$V_{s,IO_i}^{file\ scan} = \lceil P(R_i)/ReadSize \rceil \tag{A.3}$$

$$V_{s,CPU_i}^{file\ scan} = \{R_i\} \times NumSimplePreds \times I_{COMPARE} \tag{A.4}$$

The other type of scan is the index scan. To estimate the costs of such a scan, we must first estimate the number of distinct attribute values (i.e., column cardinalities as defined in Section 2.7) at a node for the indexed attribute. The number of values determines the estimated height and number of leaves of the B-tree index.

As part of the description of the indexed attribute j, the column width ($|A_j|$) in bytes should be given or estimated. This width will determine the number of bytes a key occupies in an index page. Each key in a B-tree index page has a pointer to an index subtree or to tuple storage. We represent this pointer's size in bytes by $DBptrsize$ in the formulas. The column width and pointer size are required to calculate the maximum number of key values that can be stored in an index page. This calculation is given below.

$$KeysPerPage = \lfloor PageSize/(|A_j| + DBptrsize) \rfloor \tag{A.5}$$

This number of keys per index page is used in the calculation of the number of leaves and height of the index. The number of leaves is the smallest number of index pages that needs to be used to store all the distinct attribute values. Since each index page has a maximum fanout equal to $KeysPerPage$, the height is calculated to be the log of the number of distinct attribute values using a base equal to the fanout. These calculations are given below. We assume we are dealing with an index on column j at node i.

$$Leafs_{i,j} = \lceil ColCard_{i,j}/KeysPerPage \rceil \tag{A.6}$$

$$Height_{i,j} = \log_{KeysPerPage}(ColCard_{i,j}) \tag{A.7}$$

The cost of the index scan depends on an estimation of the number of base relation pages and index pointer pages for nonclustering indexes that a scan will have to access from disk. This number of pages is dependent on the selectivity of the Boolean predicates used to scan an index. In our work, the predicates' selectivities are determined by the query plan derived by the optimizer. The selectivities in the query plan define the number of index leaf pages ($NumLeafs$) accessed on average.

For an index scan, there can exist three possible predicates. The first two predicates are used to define the range of key values searched for by the index. The third predicate is applied after the first two on the resulting leaf pages, and is a predicate choosing specific key values in the searched range of values to further restrict the keys and the pointers used to access the base relation.

In finding the range of key values for the index search, we must determine the upper and lower bounds of this range. The first Boolean predicate defines the key value to search for as the lower bound in a range search. The second Boolean predicate defines the key for the upper bound. If the first predicate is not explicitly stated in the query plan, the lower bound begins at the lowest key value in the leaves. If the second predicate is not in the query plan, the upper bound is implicitly the highest key value in the leaves. If both bounds are absent, then all the leaf pages are to be searched in order. Searching the leaves in order implies that the output result at a node will be in sorted order for the indexed attribute. If the upper and lower bounds are the same, then the index scan is assumed to be for a single attribute value and requires only one scan down the height of the index tree to the leaf pages. Otherwise, a traversal from root to leaf pages for a SEL operator is done for each of the upper and lower bound specified. No specification of the bound implies one search down the height of the tree is still necessary. The number of searches down an index tree is called $NumSearches$.

The cost formulas for searching down the index pages is given below. One page is subtracted in the equation for $V_{s,IO_i}^{index\ page\ access}$ because the height includes the leaf page level of the index. A comparison for a key at each index page requires on average $\log_2$ of half the keys that could be in an index page. We use half the keys as an estimate of the average occupancy of an index page. It should be noted that the Booleans on the leaf pages, $NumLeafs$, include the comparison costs for the third Boolean predicate.

$$V_{s,IO_i}^{index\ page\ access} = NumSearches \times Height_{i,j} + NumLeafs - 1.0 \tag{A.8}$$

$$V_{s,CPU_i}^{index\ page\ access} = \log_2(V_{s,IO_i}^{index\ page\ access} \times KeysPerPage/2) \times I_{COMPARE} \tag{A.9}$$

The cost of accessing the base relation pages after index pages are accessed depends on whether the indexing key is used for clustering or not. If the key clusters the relation, then the pointers at the leaf pages are used to access the base relation pages directly and sequentially. Otherwise, pointer pages to base relation records will need to be accessed. The formulas for clustered access are given below. In the following equations, the number of keys used to access the base relation information is given by $NumLeafKeys$. This can be determined by using the selectivity on the leaves after the third Boolean predicate is applied to them. We also require the average number of tuples per attribute value called $TupsPerValue$. Because we did not include skew in the cost estimations, this parameter is estimated using $\{R\}/ColCard_j$, unless the attribute is unique causing this value to be one.

$$V_{s,IO_i}^{clustered\ access} = \left\lceil \frac{NumLeafKeys \times TupsPerValue \times tup(R_i)}{PageSize \times ReadSize} \right\rceil \tag{A.10}$$

$$V_{s,CPU_i}^{clustered\ access} = NumLeafKeys \times TupsPerValue \times NumSimplePreds \times I_{COMPARE} \tag{A.11}$$

For a nonclustering index key, the costs are as follows. Each leaf pointer in the index points to a set of pages with pointers to base relation data. We represent this number of pointer pages by $NumPtrPagesPerValue$. Each pointer in these pages would cause an I/O access to a base relation page. In these formulas, the parameter $NumPtrPerPage$ also denotes the number of pointers each key value has in its pointer pages, and determines the number of I/O accesses.

$$NumPtrPages = NumLeafKeys \times NumPtrPagesPerValue \tag{A.12}$$

$$V_{s,IO_i}^{nonclustered\ access} = \lceil NumPtrPages + \min(P(R_i), NumPtrPages \times NumPtrPerPage) \rceil \tag{A.13}$$

$$V_{s,CPU_i}^{nonclustered\ access} = NumPtrPages \times NumSimplePreds \times I_{COMPARE} \tag{A.14}$$

A.3.2 Communication Operator Costs

There are two operators in the query plans used to communicate data between a set of source and destination nodes. These operators are SEND (to send tuples) and RECV (to receive tuples). We calculate the cost of the subplan rooted at RECV to be the sum of the network communication time and the maximum time between executing the RECV operator and the subplan rooted at the SEND operation.

The network time is dependent on the total number of messages that are sent between the source and destination nodes. The network time per message is calculated as:

$$COMMTIME = \frac{MsgSize}{\text{Network Bandwidth}} \tag{A.15}$$

where the network bandwidth depends on the network type (e.g., Ethernet has 10 Mbps, Fast Ethernet has 100 Mbps, and ATM switches have approximately 155 Mbps). The number of messages sent across the network is dependent on whether the communication is broadcast or not. If the communication is broadcast, the number of messages we used is equal to

$$\sum_{i=1}^{NumSends} P_m(R_i) \times NumRecvs$$

where $P_m(R_i)$ is the number of messages sent from the i-th sender. If no broadcast is required, the number of messages over the network would be $\sum_{i=1}^{NumSends} P_m(R_i)$. For some broadcast protocols, lots of receivers are contacted with only a single message being sent, so the number of messages for a broadcast for these protocols would be $\sum_{i=1}^{NumSends} P_m(R_i)$. The number of network messages is assigned to $V_{s,COMM}^{SEND/RECV}$.

The cost formulas for a receiver are given below. In these formulas, we use R_i to denote the relation received at node i. The formulas for sending data are similar except R_i is the relation at node i to send.

$$V_{s,CPU_i}^{receiver} = P_m(R_i) \times I_{STARTCOMM} + \{R_i\} \times tup(R_i) \times I_{COMM} \tag{A.16}$$

In receiving data, a receiver may be required to merge the inputs so as to yield a sorted result. This merging would be a way to perform a global sort on a temporary result after each of the source nodes performed a local sort of the relation. In estimating the merge costs, we only need to calculate the processor cost. We use $NumSortAttributes$ to denote the number of attributes that need to be compared to each other for the sort.

$$V_{s,CPU_i}^{receiver\ merge} = \{R_i\} \times NumSortAttributes \times I_{COMPARE} \tag{A.17}$$

A.3.3 Join Costs

To estimate the costs of a join operator and the subplan rooted at the join operator, we must calculate the cost of executing the join itself, and then calculate the costs of the join using the information of how the inner and outer children executed. An inner or outer child produce an inner or outer output relation, respectively. The information that is of use is whether the inner or outer relation were communicated and the locations of where the inner and outer relation originated.

There are two possible join algorithms used in the query plans leading to two sets of cost formulas for the join operator. These algorithms are nested-loops and merge-join. For the cost calculations of a join, we assume that only $OperatorPages$ can be used for in-memory execution of the join. For a nested-loops join's cost calculation, we assume that the outer relation will be placed in $OperatorPages - 1$ pages of memory at a time. For each of the memory page blocks, one page of the inner relation is read from disk. The number of blocks that the outer relation can fit into will determine the total number of scans of the inner relation. The inner child is always a SEL operator used to scan a base or temporary relation multiple times. We call the number of scans of the inner relation $NumInnerScans$. Since the costs of one scan is already accounted for by the SEL cost estimation, we need to reduce cost calculations for the nested-loops by one relation scan. The cost calculations are given below. Note that in the join cost calculations, we also assume that the outer child accounts for any I/O required in creating the outer relation. Therefore, the join's I/O cost only includes the cost to read the inner child's relation on multiple scans. The number of I/O accesses per scan was defined in Section A.3.1.

Part of the costs of a join includes the execution of the join predicates that are associated with a join. These predicates require the costs of comparison between the tuples from the inner and outer relations to be included in the processor costs. We assume that the number of join predicates ($NumJoinPreds$) is given in the query plan.

$$V_{s,IO_i}^{nested\ join} = V_{s,IO_i}^{SEL(index/file\ scan)} \times (NumInnerScans - 1) \tag{A.18}$$

$$V_{s,CPU_i}^{nested\ join} = \{R_{i,inner}\} \times \{R_{i,outer}\} \times NumJoinPreds \times I_{COMPARE} \tag{A.19}$$

The other join algorithm is the merge-join. This join assumes that the inner and outer relations are pre-sorted, so the cost of the join only involves the cost of merging the tuples to match them for the join. Because of our exclusion of skew, we assume that the two inputs of the join are not skewed so that no temporary I/O of the children's tuples is required and only the processor cost of merging records is determined. The cost calculations are:

$$V_{s,CPU_i}^{merge\ join} = (\{R_{i,inner}\} + \{R_{i,outer}\}) \times NumJoinPreds \times I_{COMPARE} \tag{A.20}$$

As stated earlier, the information from the children of a join is needed in estimating the cost of the subplan rooted at the join. In executing the children, if communication is involved and the children created their outputs on nodes separate from the other child, then we take the subplan cost to be the sum of the join operator execution and the maximum cost of the children's execution time estimates. When the nodes at which the children executed overlap, a sum of the join cost and both the children's costs determine the subplan cost.

A.3.4 Sort Costs

In determining the cost of sorting, we must decide whether a sort will be done internally using main memory only or externally using memory and temporary files. The reason for having to execute a sort externally is because the tuples to sort and the sorting structure cannot all fit in memory at the same time. The sort structure is a heap or tournament tree. Memory limits for executing a sort are given as $OperatorPages$. In calculating the number of input tuples that can fit in memory at any time for an in-memory sort, we use the equation below, where $NumSortTuples$ represents the number of tuples that can be sorted in memory. An internal sort is used when the number of input tuples is less than $NumSortTuples$. The number of tuples is calculated using the estimated memory required to store each tuple along with its relevant sort information in main memory. Each tuple occupies space in main memory for itself and for the necessary space in the heap. Heap space for a tuple is defined to be the space required to hold the tuple's key used for the sort and two pointers to other vertices in the heap. In the formulas, we use $keySize$ as the size in bytes of the key used to sort the input.

$$NumSortTuples = \frac{(OperatorPages - ReadSize)PageSize}{tup(R_i) + keySize + 2 \times DBptrsize} \tag{A.21}$$

The costs incurred for an internal sort are given below. In an internal sort, only processor costs are incurred since we assume that the input relation has either been read from disk by a child or was created as the output of the sort's child operation. In the sort equations, we assume the number of keys used to sort is given by $numKeys$.

$$V_{s,CPU_i}^{internal\ sort} = \{R_i\} \log_2(NumSortTuples) \times numKeys \times I_{COMPARE} \tag{A.22}$$

For an external sort, the sort executes in two phases called the run creation and merging phases. The run creation phase sorts the input relation into memory-sized blocks of $NumSortTuples$ tuples, where each block is called a *run*. Each run is sorted and put in a temporary file. When this phase is complete, we have a number of runs that are to be merged in the merging phase a pair at a time to create longer sorted runs. The longer sorted runs are then sorted a pair at a time. This process is repeated until one sorted run is left, where the final run is the sorted output of the operator. The I/O costs for the run creation phase are given below. CPU costs for this phase is the same as $V_{s,CPU_i}^{internal\ sort}$.

$$V_{s,IO_i}^{run\ creation} = \lceil P(R_i)/ReadSize \rceil \tag{A.23}$$

The costs of the merging phase are given below. Note that $KeysInMem$ tells us how many pages of the runs and their keys (two per page using the high and low key value for each page of a run) can fit in memory at any time assuming the merging uses a heap sort. The number of I/O's represents the cost of writing and reading all runs for the number of merge repetitions ($NumMerges$) needed for the sort. Each merge repetition merges a list of runs that are stored in temporary files.

$$NumRuns = \lceil \{R_i\}/(2 \times NumSortTuples) \rceil \tag{A.24}$$

$$KeysInMem = \frac{(OperatorPages - ReadSize)PageSize}{PageSize + 2 \times (keySize + 2 \times DBptrsize)} \tag{A.25}$$

$$NumMerges = \lceil log_{KeysInMem}(NumRuns) \rceil \tag{A.26}$$

$$V_{s,IO_i}^{merge} = (2 \times P(R_i)/ReadSize) * NumMerges \tag{A.27}$$

$$V_{s,CPU_i}^{merge} = \{R_i\} \log_2(NumRuns) \times I_{COMPARE} \tag{A.28}$$

A.3.5 Aggregation Cost

In calculating the costs of aggregation, we assume that the child of an aggregation has calculated the cost required for any necessary sorting. All that is left for the aggregation operator is to perform aggregation operations on each input tuple. The cost is given below. We assume that all the aggregation operations required can be executed in one aggregation operator and that the number of operations is given as part of the query plan as $NumAggPreds$.

$$V_{s,CPU_i}^{AGG} = \{R_i\} \times NumAggPreds \times I_{AGGREGATE} \tag{A.29}$$

A.3.6 Insert, Update, and Delete Costs

In calculating the insert, update, or delete costs to a base relation, the cost estimator must know about the base relation that is modified by these operations. This information includes whether the relation contains indexes and whether there is an index whose key clusters the relation. The tuples to insert, update, or delete are in relation S. If no indexes exist, the inserted tuples are placed at the end of a relation, and the tuples to remove for deletion and updates are found using a file scan. The I/O costs are:

$$V_{s,IO_i}^{no\ index\ insert} = \lceil P(S_i)/ReadSize \rceil \tag{A.30}$$

$$V_{s,IO_i}^{no\ index\ X} = \lceil (P(S_i) + \min(P(S_i), P(R_i)))/ReadSize \rceil \tag{A.31}$$

where X can be $UPDATE$ or $DELETE$. The CPU costs use I_Y as the time to perform operator Y, where Y can be $INSERT$, $UPDATE$, or $DELETE$. The following formula applies to inserts, deletes, and updates on a base relation when indexes may or may not exist.

$$V_{s,CPU_i}^{no\ index\ Y} = \{S_i\} \times I_Y \tag{A.32}$$

When indexes exist, the cost must include the cost in changing the indexes as well as the changes to the base relation. Therefore, every index j on the base relation is affected. The index cost on index j is given below. Note that $NumSplits$ is the average number of page splits and merges per index. It is also assumed that each tuple to insert, delete, or update incurs an I/O access to a leaf page.

$$V_{s,IO_i}^{index\ j\ for\ Y} = Height_{i,j} + NumSplits + \min(Leafs_{i,j}, \{S_i\}) \tag{A.33}$$

$$\begin{aligned} V_{s,CPU_i}^{index\ j\ for\ Y} \ &= \ Height_{i,j} \times I_{INDEX\_SEARCH} + \\ &\quad \{S_i\} \times (1 + NumSplits) \times I_{INDEX\_Y} \end{aligned} \tag{A.34}$$

All I/O visit counts for insert, update, and delete include the cost to insert and delete all indexes. The update however must incur this cost twice for an insertion and deletion for every updated tuple. The additional I/O costs would be:

$$V_{s,IO_i}^{all\ index\ for\ Y} = \begin{cases} \sum_j 2 \times V_{s,IO_i}^{index\ j\ for\ Y} & \text{if Y=UPDATE} \\ \sum_j V_{s,IO_i}^{index\ j\ for\ Y} & \text{otherwise} \end{cases} \tag{A.35}$$

Note that these I/O counts are used to add to the CPU counts as well.

The I/O for a base relation depends on whether there exists a clustering index. These costs are:

$$V_{s,IO_i}^{relation\ for\ INSERT} = \begin{cases} \min(P(R_i), \{S_i\}) + & \\ \min(\max(P(R_i), P(S_i)), \{S_i\}) & \text{if clustered index} \\ & \\ \lceil (1 + P(S_i))/ReadSize \rceil & \text{otherwise} \end{cases} \tag{A.36}$$

$$V_{s,IO_i}^{relation\ for\ X} = 2 \times \min(P(R_i), \{S_i\}) \tag{A.37}$$

where X is $UPDATE$ or $DELETE$.

Appendix B

Simulator Description

We created a simulator of a parallel shared-nothing DB system in order to validate our analytical estimates. The simulation uses a query workload that has been optimized by an external parallel DB optimizer. We designed this simulator to provide the various execution modes, operators, and interconnections that were necessary for our work.

The simulator was written using the event-driven simulation language of CSIM/C++ [Sch94]. We based the simulator on the PDB simulator developed at the University of Wisconsin, as described by Mehta and DeWitt [MD97].[1]

B.1 Process Model

The process model for the simulator is shown in Figure B.1. Our model is made up of FCFS queuing servers (CSIM facilities) for each resource (processors, disks, and interconnect communication network) as seen in Figure B.1. To model the execution, simulation was done at the level of blocks of tuples in the database, where blocks of tuples of specified sizes are passed around among the various simulated servers. Thus, no actual DB data is used.

Each server requires information from the queries. A processor's server requires information about the operation it would execute next (in the queue). This information includes the instruction path lengths required for the operator's execution on a processor. The service time at a processor is determined by the instruction path lenths for the operator being executed.

The communication network consists of one communication server. This server has a latency per byte associated with it. When a message of certain length is sent, the communication time is the latency of the network times the message length.

Each disk is also simulated with a facility, where the time for a disk access is *IOTIME*. Sequential reads and writes of size *ReadSize* are supported, with the cost per disk page being lower than if each page were read or written separately.

[1]We thank Kurt Brown from the University of Wisconsin-Madison for his assistance with the parallel DB simulator.

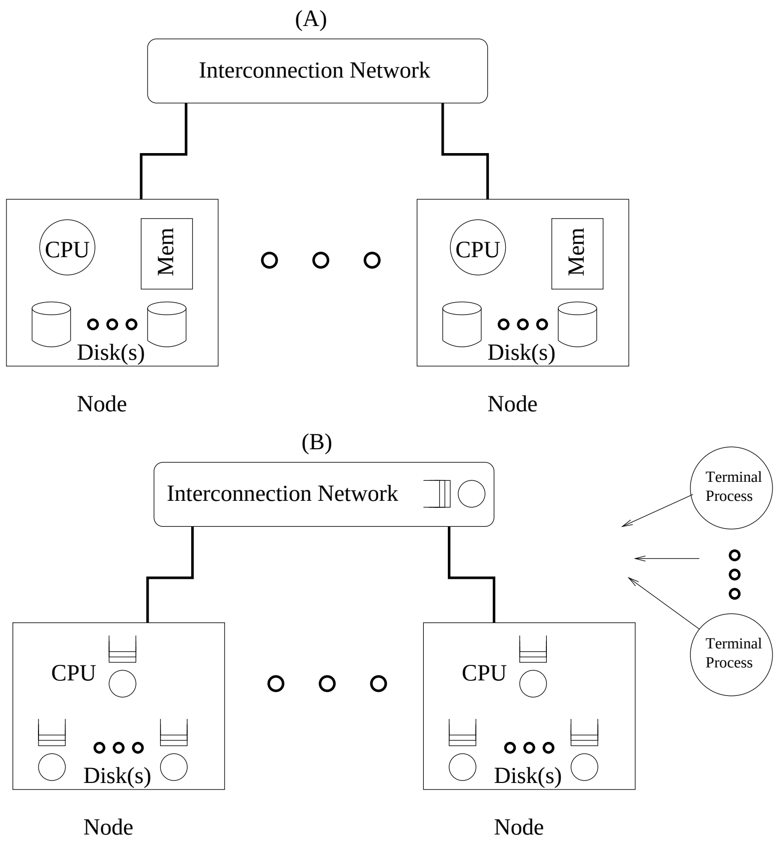


Figure B.1: System structure simulated in (A) and queuing server model in (B)

The model assumes that queries are generated with a certain arrival rate. Each query class has its queries generated through a terminal process, as shown in Figure B.1.

The simulator uses the LRU memory management policy extended with *love/hate* hints [HCL$^+$90]. In this technique, access to a memory page is augmented by a love or hate hint. A hate hint would be used in the case where a page once read and processed, such as for a sequential read, can be discarded from an operator's buffer allowing it to be reallocated by the same query or other queries. Loved pages are ones that would be best kept in memory by the operator until explicitly removed. These pages are usually index pages. In order to access any data, an operator must always access a memory location, which may lead to a page fault and hence a disk page access. Paging is done independently for each node.

Once a query has been initiated by a terminal, it is received at a central host node of the simulated parallel DB system. Upon receiving a query, the host node checks if enough memory is available or whether the specified maximum programming level (MPL) for all the queries has been met. If the MPL is exceeded for the query's class, the host node places the query on a waiting queue until it can enter the system with its resource requirements. The queries in this queue are executed in a FCFS order. Before executing a query, the host node estimates the memory requirements of a query and its operators, where predefined formulas are used to determine the amount of memory to allocate to each query operator given the amount of available memory in the system's nodes. These operator requirements are summed to determine the query requirements. If the requirements exceed the available memory size, the query is queued until it can obtain the memory it needs. The user can supply maximum or minimum memory limits for each operator in a query. We use *OperatorPages* as the maximum memory for an operator.

B.2 Inputs

The inputs to the simulator are the same as the ones required for our cost estimators. First of all the simulator requires information about the system itself. This includes the number of nodes, the number of processors and disks per node (for our work, always one processor and one large disk), the instruction path lengths of some operations, the speed of access for the disk, the main memory size per node, the communication network characteristics, and instruction costs for various operations.

A database is input as a list of relations. For each relation, the nodes used to store the relation are given (i.e., its assignment) along with the number of tuples, the tuple size in bytes, and the indexes (as well as clustering information). This information gives the physical DB design. However, partitioning keys are not used since we only required the node locations for the partitions and the number of partitions. We assume that each partition is the same size across these nodes for a single relation. Partitioning keys and relation groupings are used indirectly by representing their effects on the required communication and operations for the queries (e.g., localized or redistributed join execution).

Queries are input to the simulator in form of optimized query execution plans. This requires each query to be optimized before using the simulator. The basic operators include: selections, joins, aggregations, and sorts. Each operator contains information about the indexes to be used, selectivities, data inputs (base relations or other operations), communication required (localized, directed or full redistribution), and operator types (e.g., nested-loops or merge join). These query plans are input as trees. For example, a possible join between relations R, S, and T is represented as:

$$plan\ join2 = termsend(nestedjoin(fscan(R), nestedjoin(fscan(S), iscan(T, I_{T,1}))))$$

where *termsend* denotes communicating the final result to the coordinating node, *nestedjoin* refers to a nested-loops join, *iscan* refers to an indexed scan in which the first argument is the relation to scan and the second is the index to use, and $fscan$ refers to a sequential scan of the relation in brackets.

B.3 Query Execution

The simulator is used to execute all the query classes together. We input all the query classes with their arrival rates to the simulator, which continues executing until each query class's average response time has the 95% confidence interval within 10% of its average. The purpose of such a simulation is to allow it to validate the analytic results from a cost estimation for a workload.

Measurements are collected per query class and the averages, minimums, and maximums are given in a final report at the end of the simulation. These measurements include the average execution time for the class, the resource demands, and the amount of time spent waiting in the queue for each resource.

The scheduling of a query involves the determination of which nodes will be used for the query's execution, which files are affected, the order of the operations, and the communication involved between operator processes. Once a query is scheduled and is allowed to execute on the system, a child of the scheduler process becomes the coordinator of the operator processes, and determines when they are to begin, which nodes to send them to, and what happens when they complete.

The operator tree of each query class determines the structure of a query's schedule along with the nodes where each operator will execute. When a single node must be chosen randomly to execute an operator, as for the coordinating operator or an operation that used a host variable causing execution at a single node, each query from the query class causes each of these operations to be executed at a randomly chosen node among the possible nodes in which it could execute. The random selection of a node for an operation is part of the operator input information in the query plan. This information gives the possible nodes and specifies that one should be chosen at random on each query instantiation. A query's schedule will determine how the processes that execute the operators communicate with each other. We used CSIM processes and messages for this purpose, where the processes can be used to simulate concurrent execution between operations. Data flows between the processes are modeled with CSIM mailboxes.

All of the operators in Appendix A are simulated so that query plans for SQL queries are used in the simulator. Query plans are interpreted by the scheduler to determine how queries are to be parallelized. The scheduler dispatches operation processes to execute in parallel across different nodes. These nodes then execute the processes in parallel, synchronizing with each other wherever necessary.

To model the communication between operations, we include redistribution commands. These commands are used in communications of the relations in a join as well. Our work involves the cases of redistributing none, one, or all of an operator's inputs thereby allowing local operator execution, directed communication, or full redistribution.

Appendix C

Operator Weight Calculations

In this appendix, we describe the calculations used to rank (or weight) an attribute in terms of its use in a query operator (as discussed in Chapter 4). Operators' weights for an attribute are summed for the IR and CR algorithms in Chapter 4 so as to provide a rough estimate of the contribution an attribute makes to the query execution costs. We can make estimates for an attribute's contribution to performance only by investigating the individual query operators, so we required the weighting to be at the operator level. The summed weights are used to rank the attributes from the same relation in order to select the attribute that most positively affects the performance of the workload when used as the partitioning (or indexing) key.

Weights are dependent on a number of factors. One factor is the operator itself. Associated with the operators are the cardinalities of the relations and attribute columns involved in the operation. These cardinalities are used in the simple selectivity and cost estimations below. The simple estimations do not require the use of an optimizer or optimized query plan.

Simple query frequency counts of the attributes could serve as a weight calculation. However, the weights should reflect the operator and query importance. We want the weight of an operator to serve as a quick estimation of the query plan's cost, which would be a rough estimate of the ART for the operation executed in a query plan. This cost would be the rough ART savings between executing the operator if the attribute was chosen compared to its not being chosen as the partitioning (or indexing) key. Therefore, our weight calculations expand upon just a simple frequency count of the attributes. Note that a large positive weight for an attribute is considered good since it would denote a high estimated ART savings. A physical DB design decision algorithm could choose the attribute with the highest positive accumulated weight over all the operators in the workload to be a partitioning (or indexing) key. The range of allowable weights is dependent of the cardinality of the relations and their attributes, the operation, the estimated selectivities of some operations, the order of operator execution, and the degree of declustering.

The important SQL operations benefiting from compatible partitioning (or indexing), where the partitioning (or indexing) is on the operator's attributes, are: equi-joins, Group By operations, duplicate elimination, Order By (sort) operations, and selections. Some operations are dependent on others (such as selections), and so their weights are calculated using a predetermined ordering of execution of the operations (which will be defined later).

A *selection* could be given different weightings depending on its type of predicate, which either gives good or not so good performance results. An example of a selection that is not good for performance is an exact match on a constant value or set of constants, e.g., $A = 5$ or $A \in \{1, 2, 3\}$. An exact match of an attribute to a constant requires only one partition's data if that attribute is the partitioning key. Thus, at least part of the query's execution would always be localized to one specific node resulting in load imbalance. If the match is against a constant, then the savings of not choosing this attribute as the partitioning key would be from spreading the query over the nodes. The throughput would optimistically improve by the degree of declustering, and thus the response time reduces by a factor equal to the degree of declustering. We assume that the maximum degree of declustering is the minimum of:

- the total number of nodes; and
- the number of sequential read blocks required to store the relation.

The calculation for a bad selection (which is one that is not good for performance) is thus:

$$W_{bad\ select} = -D$$

where D is the degree of declustering. Note that a weight for an operator i (W_i) has the property that W_i is a real number. A larger degree of declustering makes the weight lower because of the added performance penalty executing on one node provides.

If the attribute is compared to a host variable, then this is considered a useful selection in terms of obtaining good performance because the variable could take on any attribute value. Choosing this attribute as the partitioning key would cause each invocation of this query to possibly be executed at a different node, in order to select the correct attribute value. Thus, multiple queries of the same form can execute in parallel (e.g., for transaction processing). Optimistically, the system could handle D times as many queries when the selection requires the partitioning key. Therefore, we made the weight calculation equal to:

$$W_{good\ select} = 1/D$$

The response times of the two different ways this selection could execute would be similar if we consider each partition to have the same size and the disk time to be directly proportional to this size. When the degree of declustering is larger, the relation partition sizes diminish in size, so the cost of executing at all the nodes is reduced.

Exact matches, e.g., $A = 5$, are the only selections of interest for parallelism under hash partitioning. Range selections of the form:

$$x \leq R.A \leq y$$

are harder to balance across the partitions, but if range partitioning or indexing is used, these predicates need to be considered. We give these predicates similar weighting as the exact matches above only if the range can fit in one partition. For indexing, we estimate the selectivities of these inequalities by comparing the x and y ranges across the full range of attribute values for $R.A$. We calculate the weight as the total number of tuples in R minus the estimated number of tuples found by an index in this range since the index would allow us to avoid reading the whole relation. For range partitioning, this requires that we find the proper data placement before a cost can be estimated. Thus, the degree of declustering and assignment decisions must be combined with range partitioning key decisions to determine the best balancing of the ranges that denote each partition.

In calculating the operator weightings, we use an assumed ordering of the operator executions in any query plan. This order would execute single relation predicates first followed by joins, then grouping and aggregation, and finally sorting and duplicate removal. The weighting of an operation later in the ordering depends on operations earlier in the order. If an operator causes poor execution performance due to poor localization at a node, e.g. execution is localized to a single node for an exact match on a constant, then the operators that follow will also be delayed. Hence, all these operators on this attribute are given negative weightings.

The ordering also becomes important when determining what cardinality is actually treated by an operator. For example, if we have a selection followed by a join, the join will actually operate on the portion of the relation that matches the selection predicate. Hence, the selectivity of the relation after the predicate is an important quantity for the operations that follow in the query order.

An equi-join would benefit greatly if one or more of its join attributes were chosen as the partitioning attributes. We assign this type of join a weight of:

$$W_{join} = R \times sel$$

where R is the relation cardinality of the relation whose join attribute we are weighting and sel is the estimated selectivity of all of the single relation predicates on R in the same query as the join. This weight calculation represents the savings in not having to redistribute or communicate the relation for the join if the partitioning attribute were the join attribute. This weighting is also useful for indexing because it represents the amount of data retrieved by the index for the join.

The selectivity (sel) is either given a default of one if no predicates exist or is estimated using the predicates and the predicates' attributes column cardinalities. For example, if we have a predicate of $A = 5$, the estimated selectivity would be:

$$sel = 1/C_A$$

where C_A is the input column cardinality of A.

Non-equi-joins only influence partitioning attribute selection when using range partitioning on bands that fall within the ranges of the partitions. For example, for the query predicate:

$$R.A \leq S.B < R.A + 40,$$

choosing to range partition R on attribute A would be helpful. However, the selection for the proper ranges creates added difficulty in the approach. Inequalities do not in general influence partitioning attribute selection, especially for hash partitioning.

For grouping in a *GROUP BY*, lower communication and merging costs are encountered if the *GROUP BY* attributes are chosen for partitioning in order to localize the aggregation. The reduction is proportional to the number of groups, and thus the weight calculation is:

$$W_{group\ by} = sel \times R/G$$

where sel is the selectivity of all the predicates on the relation (which is the sum of all the individual predicate selectivities), R is the relation cardinality, and G is the estimated average number of records in each group formed in the relation. We are assuming that each group would have an equal number of records. This calculation, R/G, gives the expected number of groups, where this number would be the estimated number of records returned by the grouping operation. A default group size of 10 is used. However, if the column cardinality of the grouping attribute is given, the group size would be calculated as:

$$G = R/C_A$$

where C_A is again the column cardinality of the attribute. This weighting is also useful for indexing because it represents the amount of data to be sorted and grouped by the index.

We treat both duplicate removal and ordering in the same way since both of them require sorting. The selection of the partitioning key as the key on which to sort could cause the execution of the sort to be localized, thus saving in the merge and communication costs of the locally sorted partitions. If the attribute(s) are partitioning keys, the duplicate removal can be localized, i.e., the sorting and removal of duplicates can be local to each node. The *DISTINCT* and *ORDER BY* operations do not aggregate afterwards, hence these operations are less expensive than *GROUP BY* in general. The weight calculation is:

$$W_{sort} = sel \times (R/G)/C$$

where C is the cost to process each element in a group, and a group in this sense is the set of records with the same attribute value. Since the operation is cheaper than grouping because of no aggregation, we divide $W_{group\ by}$ by the cost to aggregate per tuple in each group. This cost C would be proportional to $1/G$. The weighting is useful for indexing because it involves only data to be sorted by the index.

C.1 Example of Weighting

We now present an example of using such weight calculations with the schema and workload shown in Figure 4.10. In the figure, $<$ *host variable* $>$ represents a variable that could have different values on different invocations of the query. We concentrate on just the join between S and Q and the GROUP BY on attribute D of relation S. Assume a group size of ten tuples as an estimate here.

If D is not chosen as the partitioning key, the GROUP BY would be executed by first performing local sorts where S resides followed by a local grouping and aggregation. The grouped data must then be redistributed and merged at the receiving nodes to obtain a final whole relation GROUP BY result. Because the input to the nodes performing the merge is already sorted, the communication and merging can happen in parallel. If D is the partitioning key, the GROUP BY would be executed locally with about the same cost for the sorting and grouping above. Therefore, the difference in the costs is attributed to the communication after the local grouping, which is about a tenth of the size of S, using the default group size of ten tuples in a group.

Now for the join, if E is chosen as the partitioning key for S, where E is also the join attribute in the join with Q, we could either save on the communication of S by directing Q to the nodes with S or also on the communication of Q if S and Q were both in the same relation group. However, if E is not the partitioning key, S needs to be communicated. The savings in communication using the join attribute as the partitioning key would be related to either the size of S or the sum of the sizes of S and Q.

This illustrates why we chose the join and GROUP BY weightings. The saving of using the join attribute over the GROUP BY attribute for partitioning would yield a factor of ten in savings. However, the selectivities and relation sizes would have an effect on the actual amount of data communicated. The weighting factors are merely a quick way to rank the attributes with one pass of the workload.

We illustrate the use of *attribute weighting* for selecting partitioning keys with the example in Figure 4.10. Assume that relations R, S, and Q have cardinalities as given in Table C.1. Also, assume the selectivities on predicates using attributes F, I, or B are all 0.01. The degree of declustering of these relations would be assumed to be equal to the number of nodes, which

Relation	**Cardinality**	**Attribute**	**Weight**
R	10000	A	110
		B	-16
		C	100
S	5000	D	55
		E	50
		F	0.0625
Q	2000	G	20
		H	20
		I	-16

Table C.1: Example aggregate weight of attributes for relations R, S, Q.

in the example would be 16. The result of the weight assignments is given in Table C.1.

The partitioning keys are chosen to be the ones with the highest weight in each relation. In the case of ties, the key(s) with the higher cardinality or lower skew are selected. If this extra cardinality information is not available, the choice among tied attributes is arbitrary. Thus from Table C.1, the chosen partitioning keys will be $R.A$, $S.D$ and one of $Q.G$ or $Q.H$.

Appendix D

TPC-D Benchmark

In this appendix, we present the TPC-D benchmark used in our experiments. We used the DB schema (in Figure D.1), the ER-design (in Figure D.2), and cardinalities (with respect to a scale factor) as defined in the TPC-D benchmark [Raa95]. In the ER-design, there are a number of relationships between the relations. These relationships are all between the primary key of a relation and a foreign key of another relation. The attributes that represent these relationships have the following postfixes:

- ORDERKEY;
- PARTKEY;
- SUPPKEY;
- CUSTKEY;
- NATIONKEY; and
- REGIONKEY.

For example, L_PARTKEY of LINEITEM is a foreign pointer to the relation PART using its primary key P_PARTKEY of PART.

In our experiments, we assume the default partitioning key of a relation is the first attribute of that relation. Also, there is a default clustering and indexing key. Each first attribute of ORDERS, PART, SUPPLIER, CUSTOMER, NATION, and REGION is assumed to have a clustering index and each of these attributes also have unique values for every tuple in the corresponding relation.

We changed the queries to remove the subqueries. This was done because we do not estimate subquery execution as part of our cost estimation in Appendix A. In the TPC-D benchmark there are seventeen queries. We reduced this number to ten, as shown in in Figures D.3 to D.12, because of the repetitiveness among the TPC-D queries. These queries use variables ($< hv_i >$)

```
LINEITEM(L_ORDERKEY,L_PARTKEY,L_SUPPKEY,L_LINENUMBER,L_QUANTITY,
     L_EXTENDEDPRICE,L_DISCOUNT,L_TAX,L_RETURNFLAG,L_LINESTATUS,
     L_SHIPDATE,L_COMMITDATE,L_RECEIPTDATE,L_SHIPINSTRUCT,
     L_SHIPMODE,L_COMMENT)

ORDERS(O_ORDERKEY,O_CUSTKEY,O_ORDERSTATUS,O_TOTALPRICE,
     O_ORDERDATE,O_ORDERPRIORITY,O_CLERK,O_SHIPPRIORITY,O_COMMENT)

PARTSUPP(PS_PARTKEY,PS_SUPPKEY,PS_AVAILQTY,PS_SUPPLYCOST,
     PS_COMMENT)

PART(P_PARTKEY,P_NAME,P_MFGR,P_BRAND,P_TYPE,P_SIZE,
     P_CONTAINER,P_RETAILPRICE,P_COMMENT)

SUPPLIER(S_SUPPKEY,S_NAME,S_ADDRESS,S_NATIONKEY,S_PHONE,
     S_ACCTBAL,S_COMMENT)

CUSTOMER(C_CUSTKEY,C_NAME,C_ADDRESS,C_NATIONKEY,C_PHONE,
     C_ACCTBAL,C_MKTSEGMENT,C_COMMENT)

NATION(N_NATIONKEY,N_NAME,N_REGIONKEY,N_COMMENT)

REGION(R_REGIONKEY,R_NAME,R_COMMENT )
```

Figure D.1: DB Schema for the TPC-D benchmark.

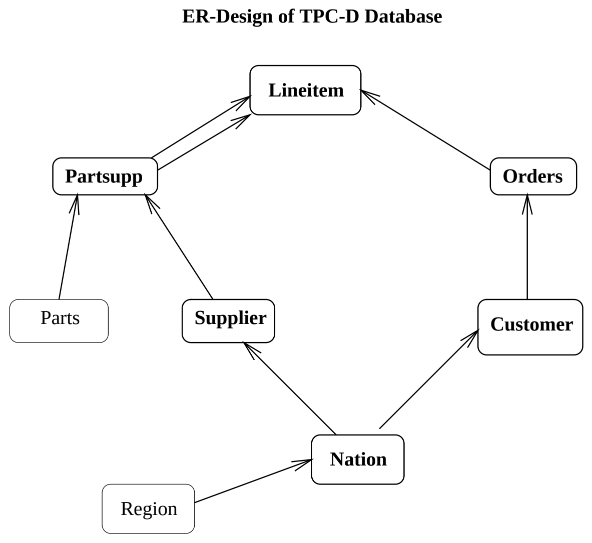


Figure D.2: The ER diagram for TPC-D.

```
SELECT DECIMAL(S_ACCTBAL,31,3), S_NAME,N_NAME,P_PARTKEY,
    P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM PART, SUPPLIER, PARTSUPP, NATION, REGION
WHERE P_PARTKEY = PS_PARTKEY AND S_SUPPKEY = PS_SUPPKEY AND
    P_SIZE = <hv1> AND P_TYPE LIKE <hv2> AND
    S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
    R_NAME = <hv3>
ORDER BY 1 DESC, N_NAME,S_NAME,P_PARTKEY
```

Figure D.3: Query 1 of WORK1 based on TPC-D query 2

instead of constant values as in the TPC-D queries. On each instantiation of a query, different values for the variables could be used. We call this revised workload WORK1.

In some of the experiments in Chapter 5, updates were required to determine how the physical DB design algorithms manage them. For these experiments, we have defined the updates in Figures D.13 to D.18 for the TPC-D relations.

```
SELECT L_ORDERKEY, DECIMAL(SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),31,3),
    O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = <hv1> AND C_CUSTKEY = O_CUSTKEY AND
    L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE < <hv2> AND
    L_SHIPDATE > <hv3>
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY 2 DESC, O_ORDERDATE
```

Figure D.4: Query 2 of WORK1 based on TPC-D query 3

```
SELECT O_ORDERPRIORITY, COUNT(*)
FROM ORDERS, LINEITEM
WHERE O_ORDERDATE >= <hv1> AND
    O_ORDERDATE < <hv1> + 3 MONTHS AND
    L_ORDERKEY = O_ORDERKEY AND L_COMMITDATE <  L_RECEIPTDATE
GROUP BY O_ORDERPRIORITY
ORDER BY O_ORDERPRIORITY
```

Figure D.5: Query 3 of WORK1 based on TPC-D query 4

```
SELECT N_NAME, DECIMAL(SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),31,3)
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND O_ORDERKEY = L_ORDERKEY AND
    L_SUPPKEY = S_SUPPKEY AND C_NATIONKEY = S_NATIONKEY AND
    S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
    R_NAME = <hv1> AND O_ORDERDATE >= <hv2> AND
    O_ORDERDATE < <hv2> + 1 YEAR
GROUP BY N_NAME
ORDER BY 2 DESC
```

Figure D.6: Query 4 of WORK1 based on TPC-D query 5

```
SELECT YEAR(O_ORDERDATE), L_EXTENDEDPRICE * (1-L_DISCOUNT),
    N2.N_NAME, R_NAME
FROM PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION N1,
    NATION N2, REGION
WHERE P_PARTKEY = L_PARTKEY AND S_SUPPKEY = L_SUPPKEY AND
    L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
    C_NATIONKEY = N1.N_NATIONKEY AND N1.N_REGIONKEY = R_REGIONKEY AND
    R_NAME = <hv1> AND  S_NATIONKEY = N2.N_NATIONKEY AND
    O_ORDERDATE BETWEEN <hv2> AND <hv3> AND
    P_TYPE = <hv4>
```

Figure D.7: Query 5 of WORK1 based on TPC-D query 8

```
SELECT N_NAME, YEAR(O_ORDERDATE),
    DECIMAL(L_EXTENDEDPRICE*(1-L_DISCOUNT)-PS_SUPPLYCOST*L_QUANTITY,31,3)
FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
WHERE S_SUPPKEY = L_SUPPKEY AND PS_SUPPKEY = L_SUPPKEY AND
    PS_PARTKEY = L_PARTKEY AND P_PARTKEY = L_PARTKEY AND
    O_ORDERKEY = L_ORDERKEY AND S_NATIONKEY = N_NATIONKEY AND
    P_NAME LIKE <hv1>
```

Figure D.8: Query 6 of WORK1 based on TPC-D query 9

```
SELECT C_CUSTKEY, C_NAME,
    DECIMAL(SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),31,3),
    DECIMAL(C_ACCTBAL,31,3), N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM CUSTOMER, ORDERS, LINEITEM, NATION
WHERE  C_CUSTKEY = O_CUSTKEY AND  L_ORDERKEY = O_ORDERKEY AND
    O_ORDERDATE >= <hv1> AND
    O_ORDERDATE <  <hv1> + 3 MONTHS AND
    L_RETURNFLAG = <hv2> AND  C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL, C_PHONE, N_NAME,
    C_ADDRESS, C_COMMENT
ORDER BY 3 DESC
```

Figure D.9: Query 7 of WORK1 based on TPC-D query 10

```
SELECT PS_PARTKEY, DECIMAL(SUM(PS_SUPPLYCOST*PS_AVAILQTY),31,3)
FROM PARTSUPP, SUPPLIER, NATION
WHERE PS_SUPPKEY = S_SUPPKEY AND S_NATIONKEY = N_NATIONKEY AND
    N_NAME = <hv1>
GROUP BY PS_PARTKEY
ORDER BY 2 DESC
```

Figure D.10: Query 8 of WORK1 based on TPC-D query 11

```
SELECT P_TYPE, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
FROM LINEITEM, PART
WHERE L_PARTKEY = P_PARTKEY AND  L_SHIPDATE >= <hv1> AND
    L_SHIPDATE < <hv1> + 1 MONTH
GROUP BY P_TYPE
```

Figure D.11: Query 9 of WORK1 based on TPC-D query 14

```
SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT PS_SUPPKEY)
FROM PARTSUPP, PART
WHERE P_PARTKEY = PS_PARTKEY AND P_BRAND <> <hv1> AND
    P_TYPE NOT LIKE <hv2> AND
    P_SIZE IN (<hv3>,...,<hv6>)
GROUP BY P_BRAND, P_TYPE, P_SIZE
ORDER BY 4 DESC, P_BRAND, P_TYPE, P_SIZE
```

Figure D.12: Query 10 of WORK1 based on TPC-D query 16

```
INSERT INTO LINEITEM
SELECT O_ORDERKEY,1,1,1,1,0.1,0.1,0.1,'L','L','1995-12-25',
       '1995-12-25','1995-12-25','L','L','L'
FROM ORDERS
WHERE O_ORDERKEY = <hv1>
```

Figure D.13: Update 1 on LINEITEM

```
INSERT INTO ORDERS
SELECT L_ORDERKEY,1,'L',0.1,'1995-12-25','L','L',1,'L'
FROM LINEITEM
WHERE L_ORDERKEY = <hv1>
```

Figure D.14: Update 2 on ORDERS

```
INSERT INTO PARTSUPP
SELECT O_ORDERKEY,1,1,0.1,'L'
FROM ORDERS
WHERE O_ORDERKEY = <hv1>
```

Figure D.15: Update 3 on PARTSUPP

```
INSERT INTO PART
SELECT O_ORDERKEY,'L','L','L','L',1,'L',0.1,'L'
FROM ORDERS
WHERE O_ORDERKEY = <hv1>
```

Figure D.16: Update 4 on PART

```
INSERT INTO CUSTOMER
SELECT O_ORDERKEY,'L','L',1,'L',0.1,'L','L'
FROM ORDERS
WHERE O_ORDERKEY = <hv1>
```

Figure D.17: Update 5 on CUSTOMER

```
INSERT INTO SUPPLIER
SELECT O_ORDERKEY,'L','L',1,'L',0.1,'L'
FROM ORDERS
WHERE O_ORDERKEY = <hv1>
```

Figure D.18: Update 6 on SUPPLIER

Appendix E

Complex Workload Description

We now present the DB and queries of the non-TPC-D workloads we used in the experiments of this thesis. These workloads were created to have similar but more complex ER diagrams than for the TPC-D workload in Appendix D. There are more joins in the workloads, and there are more foreign keys in each relation. Three workloads were used to carry out the experiments: WORK2, WORK3, and WORK4, described in the three sections that follow. Note that the queries make use of variables ($< hv_i >$) for predicate selection. On each instantiation of a query, different values for the variables could be used. The ER diagrams used in each workload are also given in the sections below. In the ER diagrams, there are a number of relationships between the relations. These relationships are all between the primary key of one relation and a foreign key of another relation.

The DB schema is the same for each of the three complex workloads we used, and is shown in Figure E.1. The cardinalities of each of the relations when using WORK2 and WORK4 is the same, where the column cardinalities and other attribute information for WORK2 and WORK4 is defined in Table E.1. Each relation's cardinality is ten million tuples, where each tuple is 180 bytes wide. As can be seen in Table E.1, attributes A1 to A4 have unique attribute values, A1 is the default partitioning key (when only indexes are selected), and A1 has a default clustering index (when only partitioning attributes are selected).

The DB used for WORK3 has the same tuple sizes, attribute widths, default partitioning and indexes, and the same unique attributes as in the DB used for WORK2 and WORK4. However, the cardinalities for WORK3 differ to the ones shown in Table E.1. The cardinalities of the relations are as follows:

- R1 and R2 have the same cardinalities as in Table E.1.
- R3 to R6 have cardinalities that are one tenth of the cardinalities in Table E.1.
- R7 to R15 have cardinalities that are one hundredth of the cardinalities in Table E.1.

```
R1(A1,A2,...,A20,A21)
R2(A1,A2,...,A20,A21)
R3(A1,A2,...,A20,A21)
R4(A1,A2,...,A20,A21)
R5(A1,A2,...,A20,A21)
R6(A1,A2,...,A20,A21)
R7(A1,A2,...,A20,A21)
R8(A1,A2,...,A20,A21)
R9(A1,A2,...,A20,A21)
R10(A1,A2,...,A20,A21)
R11(A1,A2,...,A20,A21)
R12(A1,A2,...,A20,A21)
R13(A1,A2,...,A20,A21)
R14(A1,A2,...,A20,A21)
R15(A1,A2,...,A20,A21)
```

Figure E.1: DB Schema for the complex workloads.

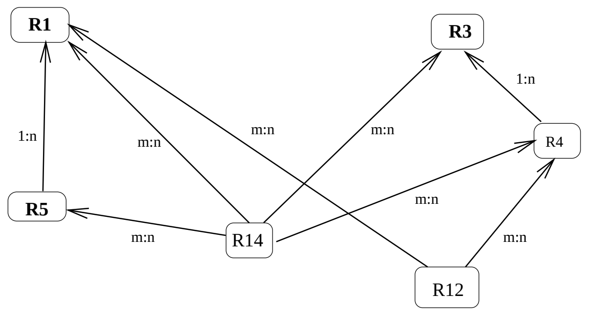


Figure E.2: The ER diagram for WORK2.

E.1 Complex Workload: WORK2

The first complex workload has five queries, each with the same relative frequency. Its ER diagram is in Figure E.2. These queries are defined in Figure E.3.

E.2 Complex Workload: WORK3

The second complex workload has six queries each with the same relative frequency. Its ER diagram is in Figure E.4. These queries are defined in Figure E.5.

E.3 Complex Workload: WORK4

The third complex workload has six queries, each with the same relative frequency (except for the second query, which has double the frequency from each of the others). Its ER diagram is in Figure E.6. These queries are defined in Figure E.7.

```
------ QUERY 1 FOR WORK2 ------
SELECT R14.A16, R14.A3, R14.A14, R1.A1, R1.A19, R1.A2, R5.A1, R1.A3,
      R5.A8, R14.A1, R12.A1
FROM R1, R14, R5, R12
WHERE R1.A1 = R14.A3 AND R1.A19 = <hv1> AND R14.A14 < <hv2> AND
      R1.A2 = R5.A1 AND R5.A8 < <hv3>  AND R1.A3 = R12.A1
ORDER BY R14.A16, R5.A8, R14.A1, R12.A1

------ QUERY 2 FOR WORK2 ------
SELECT R5.A2, R5.A10, R14.A14, R14.A1, R14.A2
FROM R14, R5
WHERE R5.A2 = R14.A2 AND R5.A10 = <hv1> AND R14.A14 < <hv2>
ORDER BY R14.A1, R14.A2

------ QUERY 3 FOR WORK2 ------
SELECT R1.A3, R1.A9, R12.A9, R1.A2, R5.A1, R12.A1, R5.A8
FROM R1, R12, R5
WHERE R1.A3 = R12.A1 AND R1.A9 = <hv1> AND R12.A9 < <hv2> AND
      R1.A2 = R5.A1
ORDER BY R12.A1, R5.A8

------ QUERY 4 FOR WORK2 ------
SELECT R4.A2, R4.A8, R14.A18, R4.A3, R14.A4, R12.A2, R12.A1
FROM R4, R14, R12
WHERE R4.A2 = R14.A4 AND R4.A8 = <hv1> AND R14.A18 < <hv2> AND
      R4.A3 = R12.A2
ORDER BY R14.A4, R12.A1

------ QUERY 5 FOR WORK2 ------
SELECT R1.A1, R14.A3, R1.A19, R14.A14, R3.A1, R3.A8, R3.A2,
       R3.A19, R4.A15, R14.A16, R14.A1, R4.A1, R3.A16, R1.A13
FROM R1, R14, R3, R4
WHERE R1.A1 = R14.A3 AND R1.A19 = <hv1> AND R14.A14 < <hv2> AND
      R3.A1 = R14.A1 AND R3.A8 = <hv3> AND R14.A1 < <hv4> AND
      R3.A2 = R4.A1 AND R3.A19 = <hv5> AND R4.A15 < <hv6>
ORDER BY R14.A16, R14.A1, R4.A1
```

Figure E.3: Workload WORK2

Attribute Name	**Column Cardinality**	**Width (in bytes)**	**Partition Key**	**Index Key**	**Unique**
A1	10000000	4	Yes	Yes	Yes
A2	10000000	4	No	No	Yes
A3	10000000	4	No	No	Yes
A4	10000000	4	No	No	Yes
A5	1000000	4	No	No	No
A6	1000000	4	No	No	No
A7	1000000	4	No	No	No
A8	1000000	4	No	No	No
A9	1000000	4	No	No	No
A10	1000000	4	No	No	No
A11	1000000	4	No	No	No
A12	1000000	4	No	No	No
A13	100000	4	No	No	No
A14	100000	4	No	No	No
A15	100000	4	No	No	No
A16	100000	4	No	No	No
A17	100000	4	No	No	No
A18	100000	4	No	No	No
A19	100000	4	No	No	No
A20	100000	4	No	No	No
A21	10000	100	No	No	No

Table E.1: Cardinalities and other attribute information for WORK2 and WORK4.

E.4 Updates on the Complex DB

For some of the experiments in Chapter E, we required the definition of updates on each of the relation. For the DB defined in Figure E.1, we defined the updates in Figure E.8.

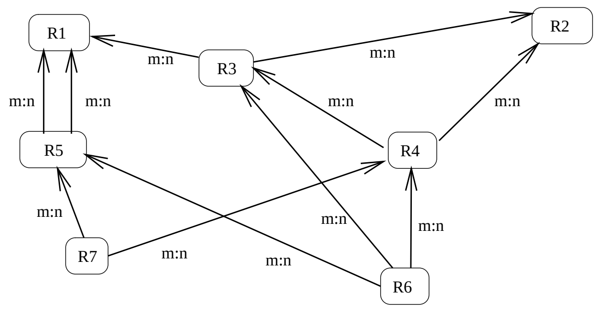


Figure E.4: The ER diagram for WORK3.

```
------ QUERY 1 FOR WORK3 ------
SELECT R1.A8, R5.A20, R5.A3, R5.A1, R7.A2, R1.A2
FROM R1, R5, R7
WHERE R1.A6 = R5.A5 AND R1.A2 = R5.A1 AND R1.A8 = <hv1> AND
      R5.A20 < <hv2> AND R5.A6 = R7.A2
ORDER BY R5.A1, R7.A2

------ QUERY 2 FOR WORK3 ------
SELECT R3.A3, R3.A13, R4.A18, R1.A4, R3.A11, R4.A4, R4.A7, R4.A2,
      R3.A1, R6.A1, R1.A1
FROM R3, R4, R1, R6
WHERE R3.A6 = R4.A2 AND R4.A18 < <hv1> AND R1.A1 = R3.A1 AND
      R1.A7 = <hv2> AND R3.A11 < <hv3> AND R4.A5 = R6.A1 AND
      R6.A1 < <hv4>
ORDER BY R4.A2, R3.A1, R6.A1

------ QUERY 3 FOR WORK3 ------
SELECT R1.A15, R3.A9, R2.A4, R3.A16, R2.A2, R3.A1, R3.A2,
      R4.A1, R1.A1, R2.A1
FROM R1, R2, R3, R4
WHERE R1.A1 = R3.A1 AND R1.A15 = <hv1> AND R3.A9 < <hv2> AND
      R2.A1 = R3.A2 AND R2.A7 = <hv3> AND R3.A16 < <hv4> AND
      R2.A2 = R4.A1
ORDER BY R3.A1, R3.A2, R4.A1

------ QUERY 4 FOR WORK3 ------
SELECT R3.A3, R3.A14, R1.A1, R1.A13, R3.A10, R3.A1, R5.A1, R1.A2
FROM R3, R1, R5
WHERE R1.A6 = R5.A5 AND R3.A14 = <hv1> AND R1.A1 = R3.A1 AND
      R1.A13 = <hv2> AND R3.A10 < <hv3> AND R1.A2 = R5.A1
ORDER BY R3.A1, R5.A1

------ QUERY 5 FOR WORK3 ------
SELECT R5.A20, R6.A14, R6.A2, R6.A6, R5.A2, R3.A4
FROM R5, R6, R3
WHERE R5.A2 = R6.A2 AND R6.A14 < <hv1> AND R3.A5 = R6.A6
ORDER BY R6.A2, R6.A6

------ QUERY 6 FOR WORK3 ------
SELECT R5.A16, R7.A6, R7.A2, R7.A1, R5.A3, R4.A3
FROM R5, R7, R4
WHERE R5.A6 = R7.A2 AND R5.A16 = <hv1> AND R7.A6 < <hv2> AND
      R4.A6 = R7.A1
ORDER BY R7.A2, R7.A1
```

Figure E.5: Workload WORK3

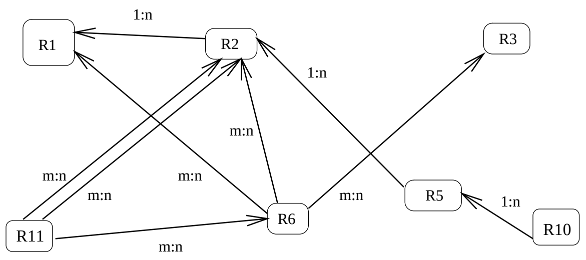


Figure E.6: The ER diagram for WORK4.

```
------ QUERY 1 FOR WORK4 ------
SELECT R3.A1, R3.A18, R6.A14, R3.A3, R3.A4, R6.A4, R6.A3, R11.A3
FROM R3, R6, R11
WHERE R3.A1 = R6.A3 AND R3.A18 = <hv1> AND R6.A14 < <hv2> AND
      R6.A4 = R11.A3 AND R11.A17< <hv3>
ORDER BY R6.A3, R11.A3

------ QUERY 2 FOR WORK4 ------
SELECT R5.A10, R2.A2, R2.A4, R5.A1, R11.A1
FROM R2, R5, R11
WHERE R2.A4 = R5.A1 AND R2.A11 = <hv1> AND R5.A10 < <hv2> AND
      R2.A2 = R11.A1
ORDER BY R2.A4, R5.A1, R11.A1

------ QUERY 3 FOR WORK4 ------
SELECT R5.A2, R5.A12, R10.A4, R2.A4, R2.A15, R5.A13, R10.A2,
      R5.A1, R10.A7
FROM R5, R10, R2
WHERE R5.A2 = R10.A2 AND R5.A12 = <hv1> AND R10.A4 < <hv2> AND
      R2.A4 = R5.A1 AND R2.A15 = <hv3> AND R5.A13 < <hv4>
ORDER BY R10.A2, R5.A1, R10.A7

------ QUERY 4 FOR WORK4 ------
SELECT R1.A1, R2.A1, R1.A10, R2.A19, R2.A3, R2.A9, R11.A9, R2.A4,
      R2.A4, R11.A2, R5.A1
FROM R2, R1, R11, R5
WHERE R1.A1 = R2.A1 AND R1.A10 = <hv1> AND R2.A19 < <hv2> AND
      R2.A3 = R11.A2 AND R2.A9 = <hv3> AND R11.A9 < <hv4> AND
      R2.A4 = R5.A1
ORDER BY R2.A4, R11.A2, R5.A1

------ QUERY 5 FOR WORK4 ------
SELECT R11.A13, R2.A3, R2.A10, R11.A17, R11.A3, R11.A2
FROM R11, R2
WHERE R11.A13 < <hv1> AND R2.A3 = R11.A2 AND R2.A10 = <hv2>
ORDER BY R11.A3, R11.A2

------ QUERY 6 FOR WORK4 ------
SELECT R1.A13, R6.A12, R2.A7, R6.A1, R6.A2, R1.A2, R2.A5
FROM R6, R1, R2
WHERE R1.A2 = R6.A1 AND R1.A13 = <hv1> AND R6.A12 < <hv2> AND
      R2.A5 = R6.A2 AND R2.A7 = <hv3> AND R6.A7 < <hv4>
ORDER BY R6.A1, R6.A2
```

Figure E.7: Workload WORK4

```
INSERT INTO R1 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R2 SELECT * FROM R1 WHERE R1.A1 = <hv1>

INSERT INTO R3 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R4 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R5 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R6 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R7 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R8 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R9 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R10 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R11 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R12 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R13 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R14 SELECT * FROM R2 WHERE R2.A1 = <hv1>

INSERT INTO R15 SELECT * FROM R2 WHERE R2.A1 = <hv1>
```

Figure E.8: Updates in the complex workloads.

Appendix F

Complexity of the Partitioning Key Selection Problem

The complexity of the partitioning key selection problem is NP-complete. We use 3SAT for this proof [GJ78]. Because the problem is NP-complete, practical solutions for the problem with realistic problem sizes must be done using a heuristic.

Theorem 1 *The partitioning key selection problem is NP-complete.*

Proof: Let each clause of the input to the 3SAT satisfiability problem be of the form:

$$(X \vee Y \vee Z)$$

where X, Y, and Z are three distinct literals including negation. For example, we could have: $(a \vee b \vee \neg c)$. The goal of 3SAT is to report whether there exists a truth assignment for the literals in a set of clauses which makes the set satisfiable (each clause would be true).

Before we show how to convert a 3SAT instance to an instance for the partitioning key selection problem, we define a variation of the partitioning key selection problem. First, we change the cost function for a set of queries to be the maximum communication cost among the queries. Therefore, the objective will be to minimize the maximum communication cost for any of the queries. Second, we reduce the communication cost to simply be a cost of 1 when a relation has to be redistributed, and zero otherwise. Third, our variant problem requires an optimizer that will choose to redistribute a relation when the join attribute of a relation in a query is not the partitioning key.

To transform a 3SAT instance to a partitioning key selection instance, we first transform the literals into a DB schema. For each literal x, we make a relation with the name R_x that has two attributes x and $\neg x$. We then transform a clause in the 3SAT instance to a query that joins the corresponding relations of the literals with the join attributes representing the literal's corresponding attribute name. For example, if the clause is $(a \vee \neg b \vee \neg c)$, the query would be:

SELECT * FROM R_a, R_b, R_c
WHERE $a = \neg b$ AND $\neg b = \neg c$ AND $a = \neg c$

For this query, if relation R_a were partitioned on attribute a, the relation would not need to be redistributed, and thus incurs no communication cost. This corresponds to setting the literal a to be true in 3SAT. Partitioning on $\neg a$ corresponds to setting the literal to false.

Since a clause is a disjunction, the clause would be true if any of the corresponding attributes for the literals is selected as a partitioning key. It is only when a query has a communication cost of three, corresponding to redistributing all three relations, that the clause would have a false value and hence the set of clauses would be unsatisfiable. Therefore, a solution for the partitioning key selection problem for these relations would be a solution to the 3SAT problem. □

Appendix G

Other Reorganization Experiment Results

This appendix presents reorganization experiment results for the reorganization decision algorithms in Chapter 6. These results show the reorganization and workload response times for reorganizing relation groups that did not lead to the best reorganization. These non-best results provide the reader with the costs of some of the other reorganization possibilities considered by our reorganization decision algorithm in Section 6.5.

These tables include the following:

- Table G.1 provides some other results related to ones in Table 6.10.
- In Table G.2, some of the other results are shown that are related to ones in Table 6.11.
- Table G.3 shows some other results related to ones in Table 6.12.
- In Table G.4, some other results are given that are related to ones in Table 6.13.
- Table G.5 provides some other results related to ones in Table 6.16.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	177083	177083	12021.9	12021.9	10034.2	1	LOW
0.9	115652	34314.7	8565.5	16755.9	5110.2	10	EQUAL
0.8	92015.9	25235.7	6313.3	12295.8	4052.6	10	EQUAL
0.7	73798.5	18846.8	4724.7	9162.2	3202.8	10	EQUAL
0.6	58916.1	14103.7	3543.1	6841.9	2504.9	10	EQUAL
0.5	46262.9	16257.8	2627	5051.9	1920.6	10	EQUAL
0.4	35170.4	7513	1894.8	3629.2	1423.7	10	EQUAL
0.3	25193	5124.9	1294.8	2470.2	994.7	10	EQUAL
0.2	16076.8	3134.6	793.1	1507.1	620.1	10	EQUAL
0.1	7695.9	1447.7	363.1	687.7	287.9	10	EQUAL

Table G.1: Other results when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK4 when PR priority scheduling is done at all the resources.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	149084	48298.4	12021.9	23614.7	6466.4	10	LOW
0.9	115652	34314.7	8565.5	16755.9	5110.2	10	EQUAL
0.8	92015.9	25235.7	6313.3	12295.8	4052.6	10	EQUAL
0.7	73798.5	18846.8	4724.7	9162.2	3202.8	10	EQUAL
0.6	58916.1	14103.7	3543.1	6841.9	2504.9	10	EQUAL
0.5	46262.9	16257.8	2627	5051.9	1920.6	10	EQUAL
0.4	35170.4	7513	1894.8	3629.2	1423.7	10	EQUAL
0.3	25193	5124.9	1294.8	2470.2	994.7	10	EQUAL
0.2	16076.8	3134.6	793.1	1507.1	620.1	10	EQUAL
0.1	7695.9	1447.7	363.1	687.7	287.9	10	EQUAL

Table G.2: Other results when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK4 when PR priority scheduling is done only at the processors.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	58132.3	29055.8	15378.1	29585.1	1181.3	1	LOW
0.9	43019.9	21502.3	11394.3	21747	1049	1	LOW
0.8	31910.1	15936.7	8310.4	15717.9	920	1	LOW
0.7	24074	11982.3	6143.2	11541.1	794.3	1	EQUAL
0.6	18193.9	9037.2	4515.3	8411.1	670.3	1	EQUAL
0.5	13614.4	6739.7	3248.3	5999.3	551.4	1	EQUAL
0.4	9943.3	4899.5	2251.8	4123.1	434	1	EQUAL
0.3	6962.7	3428.8	1577.9	2873	321.1	1	EQUAL
0.2	4345.9	2143.1	983.5	1781.5	207.1	1	EQUAL
0.1	3016.2	1498.2	331	611.1	54.5	1	EQUAL

Table G.3: Other results when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done only at the processors.

	High Rates						
DB Size Factor	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
1	58103.2	29002.7	15378.1	29622.9	1181.3	1	EQUAL
0.9	43003.7	21459.9	11394.3	21780.1	1049	1	EQUAL
0.8	31905.7	15904	8310.4	15746.3	920	1	EQUAL
0.7	21888.5	21888.5	6143.2	6143.2	794.3	1	LOW
0.6	14435.6	14435.6	4515.3	4515.3	670.3	1	LOW
0.5	9587.1	9587.1	3248.3	3248.3	551.4	1	LOW
0.4	6310.7	6310.7	2251.8	2251.8	434	1	LOW
0.3	4112.4	4112.4	1577.9	1577.9	321.1	1	LOW
0.2	2406.4	2406.4	983.5	983.5	207.1	1	LOW
0.1	1560.1	1560.1	331	331	54.5	1	LOW

Table G.4: Other results when varying DB sizes for the OLDNODE rebalancing decision algorithm for workload WORK1 when PR priority scheduling is done at all the resources.

Util.	T_{be}	**Reorg. Time**	**Bef. Time**	**Dur. Time**	**Aft. Time**	**Rel. Group**	**Pri.**
0.9	329007	329007	13740.6	13740.6	6798	1	LOW
0.7	99854.2	35096.7	7392.4	14429.7	3578.4	14	EQUAL
0.5	77249.2	24041.4	5092.5	9836.4	2949	14	EQUAL
0.3	64959.3	18185.9	3873.6	7405.3	2500.5	14	EQUAL
0.1	57474.6	14750	3157.9	5980	2183.6	14	EQUAL

Table G.5: Other results when varying the bottleneck utilization for the OLDNODE rebalancing decision algorithm for workload WORK2 using priority scheduling at all resources.