Abstract

Title of Dissertation: Adaptive Database Systems Based On Query Feedback and Cached Results

Chung-Min Chen, Doctor of Philosophy, 1994

Dissertation directed by: Professor Nick Roussopoulos
Department of Computer Science

This dissertation explores the query optimization technique of using cached results and feedback for improving performance of database systems. Cached results and experience obtained by running queries are used to save execution time for follow-up queries, adapt data and system parameters, and improve overall system performance.

First, we develop a framework which integrates query optimization and cache management. The optimizer is capable of generating efficient query plans using previous query results cached on the disk. Alternative methods to access and update the caches are considered by the optimizer based on cost estimation. Different cache management strategies are also included in this framework for comparison. Empirical performance study verifies the advantage and practicality of this framework.

To help the optimizer in selecting the best plan, we propose a novel approach for providing accurate but cost-effective selectivity estimation. Distribution of attribute values is regressed in real time, using actual query result sizes obtained as feedback, to make accurate selectivity estimation. This method avoids the expensive off-line

database access overhead required by the conventional methods and adapts fairly well to updates and query locality. This is verified empirically.

To execute a query plan more efficiently, a buffer pool is usually provided for caching data pages in memory to reduce disk accesses. We enhance buffer utilization by devising a buffer allocation scheme for recurring queries using page fault feedback obtained from previous executions. Performance improvement of this scheme is shown by empirical examples and a systematic simulation.

Adaptive Database Systems Based On Query Feedback and Cached Results

by

Chung-Min Chen

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1994

c/o Dept. of Computer Science

Advisory Committee:

Professor Nick Roussopoulos, Chairman/Advisor
Professor Michael Ball
Associate Professor Jim Purtilo
Associate Professor Christos Faloutsos
Assistant Professor Michael Franklin

Dedication

To my wife Sophie P. Wang

Acknowledgements

First I would like to thank my advisor Nick Roussopoulos for guiding me through the adventure of my research on the field of databases at The University of Maryland, College Park. It was he who provided me with an intellectual and pleasant working atmosphere and it was his guidance, encouragement, and support which made this work possible.

I am grateful to the faculty members of the database research group; Christos Faloutsos, Timos Sellis, and Mike Franklin all gave me helpful advises during my research. I would also like to thank Ken Salem, Jim Purtilo, and Mike Ball, who served in my dissertation committee, for giving me valuable feedback.

Special thanks are due to the colleges whom I have been working with on the ADMS project. Steve Kelley and I have been cooperating throughout the project since the very beginning in numerous discussion and consultations. Wenfeng Li, Zhaohui Yao, and Michael Tan helped developing and enhancing the system on various aspects. I would like to thank my colleges Alex Delis, Ibrahim Kamel, David Lin, Kyuseok Shim, and Konstantinos Stathatos for their friendship. I would also remember Nancy Lindley, Helen Papadopoulos and Sue Elliott who have helped me in numerous administrative affairs.

I want to express my deepest thanks to my parents and brothers in Taiwan. They gave me the support and freedom of pursuing my study in the United States without any burden.

Finally and most importantly, I thank my wife Sophie Wang, who has been with me throughout the final stage of my graduate study. Without

her generous support, encouragement, patience, understanding, and love, none of this work would have ever been possible. She deserves my greatest appreciation. This dissertation is dedicated to her.

TABLE OF CONTENTS

Section	Page
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Motivation	1
1.2 Contributions and Dissertation Outline	3
2 Survey	6
2.1 Query Language and Optimization	6
2.2 Caching and Using Query Results	10
2.3 Selectivity Estimation	12
2.4 Buffer Management	14
3 The ADMS Query Optimizer—Integrating Result Caching and Matching	16
3.1 Introduction	16
3.2 Related Work	18
3.3 Cache Management	19
3.3.1 Cache Representation Methods	19
3.3.2 Cache Replacement Strategies	20
3.3.3 Cache Update Strategies	21
3.4 Query Optimization Using Cached Results	22
3.4.1 Query Graph	23
3.4.2 Logical Access Path Schema	24
3.4.3 Query Graph Reductions	25

3.4.4 The Search Engine 31
3.5 Performance Evaluation 33
3.5.1 Experiment Design 33
3.5.2 Experiment Results 36
3.6 Conclusions 44

4 Selectivity Estimation Using Query Feedback **46**
4.1 Introduction 46
4.2 Related Work 48
4.3 Adaptive Selectivity Estimation 50
4.3.1 Customizing Recursive Least-Square-Error Approximation for Query Feedback 51
4.3.2 Accommodating Update Adaptiveness 53
4.3.3 An Example 57
4.4 Experimental Results 59
4.4.1 Adaptiveness to Various Distributions 60
4.4.2 Adaptiveness to Query Locality 63
4.4.3 Adaptiveness to Updates 64
4.5 Conclusions 68

5 Buffer Allocation Using Query Feedback **70**
5.1 Introduction 70
5.2 Related Work 71
5.3 The Feedback Mechanism 75
5.3.1 The Faulting Characteristic Model 75
5.3.2 Collecting Faulting Characteristics 77
5.3.3 Adapting Faulting Characteristics 79
5.4 Buffer Allocation Based on Average Marginal Gain Ratio 80

5.5 Performance Evaluation 86
5.5.1 System Configuration and Buffer Management Algorithms . . . 86
5.5.2 Simulation Results 89
5.6 Conclusions 93

6 Conclusions **95**

7 Future Research **98**

A Recursive Solution for Weighted LSE **100**

B Update Workload Specifications **103**

References **105**

LIST OF TABLES

Number		Page
3.1	Direct Elimination Tables for $A_i \rightarrow A_j$	29
3.2	Synthetic Relations for CMO Benchmark	34
3.3	Synthetic Databases for CMO Benchmark	34
3.4	Optimization Overhead Comparison of CMO and STD	43
4.1	Notations of Distribution	61
4.2	Customized Parameters for Experimental Distribution	61
4.3	Estimate Errors of ASE and SLR under Various Data Distributions	61
4.4	Three Levels of Query Localities	64
4.5	Estimate Errors of ASE and SLR under Various Query Localities	64
4.6	Characteristics of Three Update Workloads	66
4.7	Estimate Errors of Three Variations of ASE under Various Update Loads	66
5.1	Outline of Various Buffer Management Algorithms	72
5.2	Adaptation of MGR Allocation Scheme	82
5.3	Page Faults of `MGR` and `MG-x-y`	85
5.4	Parameters for Simulation	86
5.5	Four Query Types	87
5.6	Three Query Mixes for Simulation Workload	87

LIST OF FIGURES

Number		Page
3.1	Query Graph Reductions and Searching	26
3.2	Three Levels of Query Correlation	35
3.3	Effect of Cache Management on CMO Performance	37
3.4	Comparison of Variations of CMO under Different Update Loads	39
3.5	Comparison of CMO and STD under Different Query Correlations	41
3.6	Comparison of CMO and STD under Different Query Selectivities	41
3.7	Comparison of CMO and STD under Different Database Sizes	42
3.8	CMO Optimization Overhead	43
4.1	Outline of ASE	56
4.2	Adaptation Dynamics of ASE — an Example	58
4.3	ASE Adaptation under Normal Distribution	62
4.4	ASE Adaptation under Chi-Square Distribution	62
4.5	ASE Adaptation under F Distribution	63
4.6	ASE Adaptation under a bi-modal Distribution	63
4.7	ASE Adaptation under Low Query Locality	65
4.8	ASE Adaptation under Medium Query Locality	65
4.9	ASE Adaptation under High Query Locality	65
4.10	ASE Adaptation under Update Load 1	67
4.11	ASE Adaptation under Update Load 2	68
4.12	ASE Adaptation under Update Load 3	68
5.1	Faulting Characteristic Model	76
5.2	Adapting Faulting Characteristics	79
5.3	Page Fault Comparison among `MGR`, `MG-x-y` and `OPT`	85

5.4 Simulation Configuration for Buffer Management Algorithms 86
5.5 Throughput Comparison under Query Mix M1, No Data Sharing . . . 90
5.6 Throughput Comparison under Query Mix M2, No Data Sharing . . . 91
5.7 Throughput Comparison under Query Mix M3, No Data Sharing . . . 92
5.8 Throughput Comparison under Query Mix M3, Partial Data Sharing . 92
5.9 Effect of Buffer Pool Size under Full Data Sharing 93

Chapter 1

Introduction

1.1 Motivation

A *database* is a collection of data. A *database management system* (DBMS) is a system that is used to manipulate the database. One of the primary goals of a DBMS is to provide an environment that is both convenient and efficient to use in retrieving information from and storing information into the database. A *relational* database system represents the data and the relationships among them as a collection of *tables* (often called relations).

Relational database systems have been used with much success in various applications in the past decade. One of the factors that makes them such a success is the support of a *declarative* Data Manipulation Language (DML), or *query language*, by most relational DBMS vendors; such a language allows users to access the database by specifying the desired information in a query. Such a system frees the user from concern over how to access the database efficiently; it is the responsibility of the DBMS to find the most efficient way to evaluate the query.

To process a query expressed in a declarative language such as SQL [A+76] or Quel [SWK76], the DBMS must select, during a query optimization phase, an efficient plan for processing the query. Without query optimization, most relational database

systems would be highly inefficient. The result of query optimization is a query *plan* which can be interpreted and executed by the underlying database access method and storage management modules. The access method module supports alternative means of accessing the relations (such as indices); the storage management module organizes database files on the disks, provides an interface between the access method module and the file system, and is also responsible for transferring database pages between memory and disks. We refer to this whole subsystem of a DBMS as the Query Optimization and Evaluation Module (QOEM).

In a traditional database system, the function of a QOEM is to optimize a query, execute the produced plan, and return the result to the users. This uni-directed function flow overlooks the potential advantage of recycling useful query results for speeding up subsequent queries and of feeding back useful experience or information learned from execution. This dissertation commences research in this direction. In particular, we focus on three techniques that would extend or improve the function of a QOEM from this aspect.

- We extend the QOEM to caching query or intermediate results for reuse in order to save query evaluation time for follow-up queries. This idea has been proposed in previous literature for different applications. However, to the best of our knowledge, no concrete work has ever been reported, and cached results have never before been integrated into the optimizer and the access path module efficiently.

- We propose a novel approach of regressing attribute value distribution using query feedback to make accurate selectivity estimation. *Selectivity estimation* is one of the most important factors affecting the correctness of query optimization and thus the quality of the output plans. Traditional methods for selectivity estimation usually involve the enormous off-line database access and computation

overhead of collecting relevant data statistics and maintaining data distribution. We feel that using the knowledge about data distribution obtained from query execution to estimate selectivities is much more efficient than the traditional methods.

- We explore the technique of using page fault feedback to adjust buffer allocation for recurring queries and thus to increase buffer utilization. Recent research on database buffer management has proposed various allocation schemes based on the specific page reference patterns exhibited by relational queries. However, the allocation strategies of these methods were based on probabilistic analysis where crude assumptions such as uniformity had to be made. Instead of using this assumption, we seek to use real page reference characteristics obtained from query feedback to adjust buffer allocation.

In short, the purpose of this study is to explore the query optimization technique of using cached results and to investigate the use of query feedback in selectivity estimation and buffer allocation.

1.2 Contributions and Dissertation Outline

The results of this study and the contents of this thesis have been published in [CR94b, CR94a, CR93]. Although the experiments we conducted were based on a centralized database environment, the techniques proposed here could be applied to a distributed environment as well. We briefly discuss the contributions of this dissertation as follows.

Previous research on the issue of utilizing cached results has focused on either cache management or the theoretical discussion of how to determine if one query condition implies another one. Little attention has been paid to the problem of how to optimize a query using previously cached query results. This issue is important because the blind use of cached results may result in a query plan even worse than that created without

using them. To the best of our knowledge, no substantial framework of this sort has ever been developed or prototyped in any database product or research system, and thus the practicality of this technique is still questionable. On this basis, we investigated and developed a query optimizer that can optimize queries using cached results and that can generate more efficient plans. This optimizer was implemented in a DBMS prototype, and the experimental results showed that, under a variety of query workloads, the saving of query evaluation time is significant, and the optimization overhead is usually extremely small.

We then turn to the selectivity estimation problem. To make accurate selectivity estimation, we need to maintain the value distributions for those attributes of interest. Existing methods all require off-line database accesses and computation in order to collect the statistics and approximate the distributions. When updates to the relations occur, these maintained distributions are not usually modified until the updates exceed a given threshold. This system results in two problems: (1) because of the presence of the outdated information, the query optimization will operate in a degraded mode until the distributions are updated, and (2) the updating of the distributions will incur more overhead. To solve these problems, we propose a novel approach of regressing attribute value distributions using actual query result sizes as feedback. With this technique, the distribution, which is adjusted gradually as queries proceed, adapts well to the updates. This approach requires no off-line database accesses and makes accurate estimations comparable to those of the existing methods.

Finally, we present a buffer allocation scheme for recurring queries. Queries such as compiled queries and non-materialized (or pointer-based) views are very likely to recur in a database environment. For these queries, we can use information about page reference behavior learned from previous executions to adjust the buffer allocation. We use a quantitative model to characterize the page reference behavior that a query imposes on a relation, and based on this, we devise an allocation scheme which allocates

buffers to individual relation instances according to their quantified characteristics. Simulation results have shown that the use of this technique can lead to significant performance improvement.

This dissertation is organized as follows. In Chapter 2 we review the concepts of relational databases and query optimization and survey the relevant issues of caching query results, selectivity estimation, and buffer management. In Chapter 3 we investigate the query optimization technique of using cached query results. We present an implementation framework which integrates various cache management techniques and query optimization. An empirical performance study which has shown the advantage of this framework is also presented. In Chapter 4 we describe the technique of using actual query result sizes as feedback to regress attribute distribution and make accurate selectivity estimation. We also show how this method adapts gracefully to updates and query locality. In Chapter 5 we describe the buffer allocation scheme for recurring queries based on feedback of page fault information. We show the simulation results which indicate the performance improvement of this scheme over the existing ones. Finally, we offer conclusions and issues for future research in Chapters 6 and 7.

Chapter 2

Survey

2.1 Query Language and Optimization

A *relational database* consists of a collection of relations by which users can store data and represent relationship between data. Each relation is associated with a *schema* which specifies the name of the relation, the attributes of the relation, and the domain of each attribute. A *query language* is a language in which a user requests information from the database. Query languages can be categorized as being *procedural* or *nonprocedural.* In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a nonprocedural language, the user describes the information desired without giving a specific procedure for obtaining that information.

The *relation algebra* is a procedural query language. It consists of three fundamental operators: *select, project,* and *cartesian-product.* In the following, we introduce these operators using demonstrating examples from [KS86] and [Shi93]. Suppose a company stores its records of employees, departments and projects in three separate relations with the following schemas.

```
DEPT(dept_no, dept_name,floor_no)
EMP(name, dept_no, project, salary)
```

```
PROJ(proj_name, contract_type)
```

The *select* operation selects tuples that satisfy a given predicate. A select operation is denoted by σ_p, where p is the predicate. To select all departments that are located in the first floor, we write

$$\sigma_{\texttt{floor\_no}=1}(\texttt{DEPT})$$

The *project* operation extracts certain attributes from the relation. A project operation is denoted by Π_A, where A is the list of attributes that would appear in the result. To project the names of all employees from relation EMP, we write

$$\Pi_{\texttt{name}}(\texttt{EMP})$$

Given two relations R and S, the *cartesian-product* of R and S, denoted by $R \times S$, is the set of all tuples that are concatenated by one tuple of R and one tuple of S. A *join* operation, written as $R \bowtie_p S$, is a shorthand for $\sigma_p(R \times S)$, where predicate p is called the join predicate. The following expression retrieves the names and salaries of all employees who work on the first floor.

$$\Pi_{\texttt{name,salary}}(\sigma_{\texttt{floor\_no}=1}(\texttt{EMP} \bowtie_{\texttt{dept\_no=dept\_no}} \texttt{DEPT}))$$

Most database system products provide a more "user-friendly" query language other than relational algebra. Perhaps the most influential commercial query language is the SQL, introduced as the query language for System R [A+76]. A typical SQL query has the form:

SELECT $a_1, a_2, \ldots, a_n$

FROM $R_1, R_2, \ldots, R_m$

WHERE P

The a_is represent attributes (referred to as the *target-list*), the R_is represent relations (referred to as the *relation-list*), and P is a predicate (referred to as the *condition-list* or the *qualification*). This query produces the same result as the following relational

algebra expression:

$$\Pi_{a_1,a_2,\ldots,a_n}(\sigma_P(R_1 \times R_2 \times \ldots, \times R_m))$$

Unlike relational algebra which specifies exactly in which sequence the operations are to be performed, SQL simply gives declarative information about what is desired. For example, the relational algebra expression

$$\Pi_{\texttt{name,salary}}(\sigma_{\texttt{floor\_no=1}}(\texttt{EMP} \bowtie_{\texttt{dept\_no=dept\_no}} \texttt{DEPT}))$$

can be represented in SQL statement as

QUERY 1:

```
SELECT EMP.name, EMP.salary
FROM EMP, DEPT
WHERE DEPT.floor_no =1
      AND EMP.dept_no = DEPT.dept_no
```

When a SQL query is presented to a database system, it is necessary to find an efficient way, if not the best, to compute the answer using the existing database access methods. Each alternative way of computing a query is called a *query plan.* For example, two query plans are possible for the above SQL query.

Plan 1:

$$\texttt{TEMP1} := \Pi_{\texttt{dept\_no}}(\sigma_{\texttt{floor\_no=1}}(\texttt{DEPT}));$$

$$\texttt{RESULT} := \Pi_{\texttt{EMP.name,EMP.salary}}(\texttt{EMP} \bowtie_{\texttt{dept\_no=dept\_no}} \texttt{TEMP1});$$

Plan 2:

$$\texttt{TEMP1} := \texttt{EMP} \bowtie_{\texttt{dept\_no=dept\_no}} \texttt{DEPT};$$

$$\texttt{RESULT} := \Pi_{\texttt{EMP.name,EMP.salary}}(\sigma_{\texttt{dept\_floor=1}}(\texttt{TEMP1}));$$

In this case, Plan 1 is expected to be more efficient than Plan 2 because it reduces the number of DEPT tuples to be joined with EMP by applying the select operator

to DEPT before the join. If there is an index built on attribute DEPT.floor_no, we also need to decide if it is more efficient to use this index than not to use it when performing $\sigma_{\texttt{floor\_no=1}}(\texttt{DEPT})$.

There are generally a large number of possible query plans for evaluating a query, especially if the query is complex. It is not uncommon that the difference in execution time between a good plan and a bad plan may be huge. Thus, it is worthwhile for the database system to trade a small amount of time on the selection of a good plan for the significant execution time saved by the good plan. This "optimizing", or more accurately, improving of the strategy for processing a query, is called *query optimization.*

Given a query, the query optimizer must find out the most efficient plan for computing the query. There are two things that can affect the cost of a query plan: *access methods* and *join order.* Database systems usually provide alternative access methods for retrieving data from tables (aside from sequential scanning, there are access methods like indexing, hashing, and sort-merging), and the query optimizer must determine the best method to access each relation in the context of a specific query. For example, an index can be used in Plan 1 to select the departments on floor 1 from DEPT, or we can simply scan through DEPT sequentially and select all those qualified tuples. Similarly, if an index exists on attribute EMP.dept_no, then we can use this index to join each qualified tuple from DEPT with tuples in EMP that match on their dept_no values. The order in which the relations are taken into join might as well affect the cost of the resulting plan. This is because different join orders access different amount of tuples within each relation and produce different sizes of intermediate results. The amount of tuples accessed must be minimized to reduce the number of disk accesses required.

Query optimization has been well studied in the past [WY76, S+79, IK84, GW89, YL90]. Among them, perhaps the most influential one is the System R optimizer

[S+79], which is based on the technique of *dynamic programming*. In System R, a join with N relations is considered as a sequence of 2-way joins. Two relations are first joined together and the resulting relation is joined with the third relation, etc. The optimization algorithm proceeds by considering increasing larger subsets of the set of all join relations. It begins with finding the optimal plans for all subsets of 2 relation, and then finds the optimal plans for all subsets of k relations ($k \geq 3$). Dynamic programming is used so that the optimal plan of a k relation join is extended from one of the optimal plans for the $k-1$ relation joins. At the same time, certain heuristics are adopted to limit the search space. For example, the famous "push-down" heuristics attempts to apply selections and projections as early as possible and delay cartesian product as late as possible. The search space is at worst of 2^N. However, for typical join queries of less than 10 relations, the optimization overhead is not that significant [1].

2.2 Caching and Using Query Results

Caching the result of a query on disk can potentially save the execution time for follow-up queries. For example, suppose QUERY 1 has been executed and the result is saved in a relation RESULT1. Consider the following query which lists the names, salaries of all the employees who work on the first floor, earn more than 50K dollars, and are in a project of government contract.

```
SELECT EMP.name, EMP.salary
FROM EMP, DEPT, PROJ
WHERE EMP.salary ≥ 50,000
```

[1] For large join queries that involve tens or even hundreds of relations, another class of optimization algorithms must be adopted [IW87, SG88, Swa89, IK90], and they are not within the scope of this dissertation.

```
        AND EMP.dept_no = DEPT.dept_no
        AND DEPT.floor_no =1
        AND EMP.project = PROJ.proj_name
        AND PROJ.contract_type = 'government'
```

It is not hard to see that we can take advantage of the cached result RESULT1 and rewrite the above query as

```
SELECT EMP.name, EMP.salary
FROM RESULT1, PROJ
WHERE EMP.salary ≥ 50,000
      AND EMP.project = PROJ.proj_name
      AND PROJ.contract_type = 'government'
```

This expression avoids the re-computation of join between EMP and DEPT and is more likely to be less costly than the first expression. However, it is not expected that the users will be skilled enough to make this kind of query transformation, and even if they are, they might not be aware of or able to keep track of all the pre-existing cached results that can be made use of. It should be the responsibility of the query optimizer to handle the task of caching and using query results and make it transparent to the users.

Finkelstein in [Fin82] described an algorithm for comparing a query with existing views. Once a view matching a subexpression of the query is found, the query is rewritten by substituting the subexpression with that view. Larson and Yang [LY85] transformed the problem of finding a matched view into an equivalent graph theory problem. Roussopoulos in [Rou82a, Rou82b] proposed to cache query results in views and store the access paths in a Logical Access Path Schema (LAPS). The LAPS is a logical entity which provides an integrated representation of all the possible logical access paths to derive the views. In practice, the LAPS is physically recorded in

several catalog tables, and is used as a basis for common access paths recognition in query optimization. [J+93] proposed to support transaction time using differential techniques and cached results, and outlined an optimization algorithm which takes into account cached results.

The profit of using cached results is not totally free. Certain amount of disk space must be reserved for the cache, and the cached results must be managed in an efficient way. There are different ways to store the query results. In [AL80, BLT86], query results are stored in views as regular data, called *materialized views*. [Rou82b, Val87] proposed to store query results by pointers or *Tuple IDs* which are addresses of tuples in relations. Sellis [Sel88] discussed the issue of cache replacement strategy and suggested several useful heuristics. A cache must be updated when its underlying base relations are modified. Different strategies and techniques as of when and how to update derived relations were explored in [Shm84, RK86, BLT86, LHM86, Han87, BCL89]. Incremental updates of the pointer-based representation of views is fully developed in [Rou91]. We will discuss more about them in Chapter 3.

Despite of the intensive literature on the issues of caching, managing, and using query results, there are few that actually integrate them all in a concrete framework. In Chapter 3, we will describe an implementation which integrates cache management and query optimization using cached results in a DBMS prototype.

2.3 Selectivity Estimation

The cost of a query plan depends heavily on the *selectivities* — the size (number of tuples) of the result of a selection or a join. Selectivity estimation has a significant impact on the selection of best plan. Consider Plan 1 as an example. If 90% of the company's departments are on the first floor, then we might choose to scan sequentially through relation DEPT instead of using the index on `DEPT.floor_no` to read the

qualified tuples since in both cases, almost all pages of DEPT will be read, and using index will simply incur more disk accesses to the index structure. On the other hand, if only 10% of the company's departments are on first floor, then it might be more efficient to use the index to select those 10% tuples from DEPT than to read through the whole relation.

Selectivity depends on the distribution of values within attributes. However, it is not realistic to maintain *exact* distribution for each attribute in the database. For simplicity, many query processors make the assumption of *uniform* distribution which assumes that each value of an attribute domain appears with equal probability [S+79, KS86]. As today's database applications involve more complex and massive amount of data, the uniformity assumption is no longer satisfactory since skew data distribution is not unusual. In order to make correct decisions during query optimization, accurate selectivity estimation is desired.

Various methods have been proposed to approximate the distribution of attribute values and thus make accurate selectivity estimation. The most common method is the *histogram*, which divides an attribute domain into intervals and counts the number of tuples holding values which fall into each of the intervals. Variations of histograms are proposed in [MO79, PSC84, MD88, Lyn88, Ioa93]. Another group of methods [S+79, SB83, Chr83b, Chr83a, LST83, Fed84, SLRD93] used certain mathematical functions (such as polynomials or normal distribution) to approximate actual distribution. The above methods all require off-line database accesses for collecting certain statistics. This task is expensive because it incurs intensive disk accesses and has to be re-performed periodically at the presence of updates. Recently, *sampling* methods were proposed to estimate query result sizes [HOT88, LN90, HS92]. Sample tuples are taken from the relations, and the query is performed against these samples to collect statistics for estimating the selectivity. However, sampling methods are basically used in answering statistical queries rather than in the context of query optimization where

the overhead of performing sampling is prohibitive.

In Chapter 4, we propose a novel approach of selectivity estimation which regresses the distribution using actual query result sizes as feedback in real time. This method avoids the cumbersome off-line database access overhead and gives accurate estimation comparable to the existing methods.

2.4 Buffer Management

When an optimal query plan is generated, it is sent to the underlying access method and storage management module for execution. This module reads data pages from the disk and performs desired computation to produce the final result. Since accesses to disk pages are much more costly than accesses to memory, most database systems provide a memory buffer pool for caching data pages and thus reducing disk accesses. Early works on database buffer management [Rei76, SB76, TLF77, Kap80, EH84] adopted the conventional strategies that are previously used in virtual memory systems (such as LRU, Working-Set, etc). In these methods, database buffers are managed based on certain simple page reference statistics. The treatment is "page-oriented" rather than "query-oriented" in the sense that each database page is treated equally regardless of which relation it resides or in what context of query it is accessed. This basically overlooks the specific page reference behavior exhibited by relational queries which can be used to improve the buffer utilization.

Let us use an example to show how the buffer allocation strategy can affect the effectiveness of the execution of a relational query plan. Consider Plan 1 again, this time assume there is no index on any attribute of either relation. For the query $\texttt{EMP} \bowtie_{\texttt{dept\_no=dept\_no}} \texttt{TEMP1}$, suppose the optimizer chooses to read one tuple at a time from TEMP1 and compares it with every tuples of EMP. It is obvious that to best utilize the buffers in assisting this join operation, we should allocate only 1 buffer

page to TEMP1 (since TEMP1 is accessed sequentially, and each page will be read only once), and allocate the rest to EMP (since the data pages of EMP are accessed iteratively for many times).

[SS82, SS86] first proposed to allocate buffers to individual relation instances based on the *reference patterns* exhibited by the query plan. [CD85] refined this work and showed the advantage of this approach over the traditional methods. Ng, Faloutsos, and Sellis improved the work of [CD85] by using a more flexible allocation policy which takes into account buffer availability [NFS91], and augmented it in [FNS91] by considering the effect of load control.

In Chapter 5, we propose a buffer allocation scheme for recurring queries using feedback of page fault information. Instead of relying on ad hoc reference pattern classification, we use a quantitative model to characterize the reference behavior of queries and use such obtained characteristics to adjust buffer allocation. This technique complements the pattern-based methods on the allocation strategy for "random" references.

Chapter 3

The ADMS Query Optimizer—Integrating Result Caching and Matching

3.1 Introduction

Relational database query languages allow users to save final query results in relations [S+79, SWK76]. Under certain situations, for example, when sorting is performed or nested queries are present, query intermediate results must also be produced to facilitate the query computations. It is then profitable to cache these query results on disk over a longer time for potential reuse. Caching query (intermediate) results for speeding up follow-up query processing has been proposed for different applications in previous literature. In [AL80, Fin82, LY85, Rou91], cached query results are used in relational database systems to avoid repeated computations. [Sel87, Jhi88] addressed the issue of caching query results to support queries with procedures, rules and functions. In a distributed client-server environment, caching query results on client workstations can reduce both the network contention and the server request bottleneck [DR92]. Recently in extended relational databases, this technique was suggested to save evaluation time of expensive predicates which involve large and complex attributes [HS93].

Research on this topic must address to two major issues: (1) cache management,

and (2) query matching and optimization. The two issues have been investigated separately in previous literature, where [AL80, Rou82b, Val87, Rou91, Sel88, Jhi88, RK86, Han87, BLT86] focused on the cache management, and [Rou82a, Fin82, LY85, S+89] addressed the query matching. However, no substantial work or experimental results have been reported on the integration of caching, matching and optimization, except for algorithmic discussion. Therefore, it is not clear if such an integration can be fulfilled in practice efficiently.

In this chapter, we describe the design and implementation of the ADMS[1] Cache and Match Optimizer (CMO)— the first prototyped optimizer that integrates query result caching and matching. ADMS adopts an *enhanced client-server* architecture which utilizes both the CPU and the I/O of the client workstations. Each server is dedicated to running a DBMS and maintaining a main centralized database. The users access the database on servers through their workstations (the client workstations) via a local area network and can create their own private databases on the local disk. The system off-loads disk accesses from the servers by having the clients run a limited DBMS (a centralized version of ADMS) and by caching results of both client and server queries to the client disk. To provide cache transparency to the users, we integrated the ADMS query optimizer with the underlying cache manager and with the matching mechanism so that the users do not need to worry about which caches to use and how. The integrated optimizer can generate efficient query plans using previously cached results automatically. In particular, it (1) can use multiple cached results in computing a query, (2) allows dynamic cache update strategies, depending on which is better, and (3) provides options for different cache management strategies. While the ADMS CMO query optimizer is running in a client-server environment, in this chapter, we only present the performance evaluation results from a centralized

[1]ADMS is a relational DBMS prototype developed at the Department of Computer Science of University of Maryland, College Park.

case where the user runs queries against his private database and caches the query results on the local disk.

The rest of this chapter is organized as follows: Section 3.2 briefs the related work, Section 3.3 describes the underlying ADMS cache manager, Section 3.4 details the integration of query matching and optimization in the CMO optimizer, Section 3.5 presents the performance evaluation results, and finally in Section 3.6, we summarize this work.

3.2 Related Work

Different issues concerning cache management have been widely studied before. [AL80, Rou82b, Val87, Rou91] proposed alternative methods for storing the cached data; [Sel88, Jhi88] discussed the problem of selective caching and cache replacement strategies; in [RK86, Han87, BLT86], different cache update strategies are explored. More details about these issues will be discussed in Section 3.3.

The problem of identifying useful cached results that can be used in computing a query, referred to as the query matching problem, was addressed in [Fin82, LY85, S+89]. The essential issue here is to find an algorithm that can test if the qualification clause of a query logically implies that of a view definition. The solution usually involves some theorem proving techniques whose computational complexity, in general, is exponential, though polynomial algorithms exist for certain special cases of the problem. Optimization, however, was not the issue in this work and thus was not addressed satisfactorily.

Optimization can not be neglected when using cached results not only because there may be different combinations of matched caches from which the query can be computed, but also because it is not always beneficial to use caches. A possible solution, as mentioned in [Fin82], is a *two phase* approach; during the first phase,

the query is transformed into a number of equivalent queries using different cached temporaries, and during the second phase, all the revised queries are fed to a regular optimizer to generate an optimal plan. Without elaborate pruning, this approach is not satisfactory because the search space for both phases is extremely large. Even if only a small number of revised queries are produced from the first phase, these revised queries can still duplicate their search spaces and effort during the following phase. A better approach is to integrate the matching phase with the optimization and thus, unify the search spaces and avoid duplicate effort. [J+93] first described this approach and used a *state transition network* to explore the space, along with some pruning heuristics. Rather than being implemented as a working module in any DBMS, their work is merely an algorithmic piece.

In the following, we describe the design and implementation of the ADMS CMO framework which integrates query result caching, matching, and optimization.

3.3 Cache Management

Intermediate and final query results cached on disk are referred to as *temporaries*. Cached temporaries are collected and maintained by the cache manager. In this section, we review relevant cache management issues and describe the approach we adopted in the ADMS cache manager.

3.3.1 Cache Representation Methods

The simplest way to store the temporaries is to store them as regular relations [AL80, BLT86]. This is called *materialized view* or *data caching*. Another approach is to store for each resulting tuple of the temporary, a number of pointers or *Tuple Identifiers (TID)*, instead of materialized values, which point to the corresponding tuples in the base relations (possibly through several levels) that constitute the resulting tuple. We

refer to this pointer based approach as *pointer caching*. Variations of this approach have been proposed in [Rou82b, Val87], and was called *ViewCache* in [Rou91].

Pointer caching is more space-effective since each tuple is represented by a small number of fixed length pointers. However, extra page references to higher level relations or temporaries are required when materializing tuples from pointer caches. In view of query matching, pointer caching is more attractive than data caching because (1) more temporaries can be retained in a limited cache space, and (2) unlike data caches which have only projected attributes, pointer caches virtually serve as indices to the base tuples and thus can select any attributes from the underlying relations. This makes pointer caching more versatile than data caching.

In ADMS, both data caching and pointer caching are supported. The default can be set to either one, and the user can explicitly specify in a query whether the final result is to be stored in a pointer cache (ViewCache) or a data cache.

3.3.2 Cache Replacement Strategies

In a system which provides unbounded disk space, we can simply cache everything generated and leave the task of how to use these temporaries to the query optimizer. However, a more realistic situation is to bound the available space for caching. In this situation, a *cache replacement strategy* must be employed to decide which temporaries to replace when the cache space is full. The problem of choosing a good replacement strategy so that the most profitable query results can always be cached was addressed in [Sel88]. Basically, he proposed to associate each temporary with a *rank*, which is a weighted sum of certain relevant statistics (such as the cache size, the time since the temporary is last referenced, etc.). The lowest rank cache(s) should be discarded at a point where cache space is needed. Unfortunately, they did not tell how to derive the proper weights and simply left it as an open problem. [Kam87] compared the different heuristics proposed in [Sel88] based on a simulation study. Still, no empirical

evaluation was reported.

With no evidence of which strategy performs the best, we simply equipped the ADMS cache manager with three options of replacement strategies:

LRU: replace the least recently used — in case of a tie, LFU, LCS

LFU: replace the least frequently used — in case of a tie, LRU, LCS

LCS: replace the largest cache space — in case of a tie, LRU, LFU

The purpose here, however, is not to compare amongst the different replacement strategies, but rather observe the CMO performance change under different available cache spaces without biasing towards any replacement strategy, though the results might shed a light into the choice of proper replacement strategy under certain query workload.

3.3.3 Cache Update Strategies

Cached temporaries become outdated when the base relations from which they are derived are modified and thus must be updated before they can be further used. There are three different strategies regarding *when* to update the outdated caches: (1) *immediate update* (i.e., right after every update to the base relations), (2) *periodic update*, and (3) *lazy update* (i.e., only when access to the outdated cache is requested). As for the cache update method, it can be *re-execution* or *incremental* [LHM86, Rou91]. While the re-execution method re-computes an outdated temporary from scratch, the incremental method can efficiently update a temporary if only a few tuples are modified in the base relations. However, incremental update logs must be maintained to support incremental updates.

It was analyzed in [Han87] that none of the combinations of the update time strategies and the update methods is superior to all the others under all situations. As it is practically prohibitive to experiment with all the possible combinations, the lazy

update strategy has been adopted in our implementation because it can batch consecutive updates into a single update (and thus reduce the excessive overhead of multiple smaller updates) and always prevents the unnecessary updates to never-used caches. The ADMS, however, supports both incremental and re-execution methods for View-Caches, but supports only re-execution method for data caches [2]. The choice between incremental and re-execution updates is made by the query optimizer, depending on the estimated costs.

3.4 Query Optimization Using Cached Results

In this section, we describe the ADMS CMO query optimizer, which currently handles only the class of *SPJ-queries*— queries which involve only projections, selections and joins. The optimizer employs a graph search-based algorithm [Nil80] (referred to as state transition network in [J+93, LW86]), where each query is represented by a *query graph* (or *state*). The optimizer has two components: the *reduction module* and the *search engine*. The reduction module consists of the procedures for performing query graph transformations. The search engine controls the exploring sequence of the search space, i.e., the order in which the query graphs will be explored for reductions. When a part of a query graph is computed either by performing a join or using a matched cache, this query graph is said to be *reduced (transformed)* to a new one. The access cost of the join or the cache is estimated and accumulated into the successive state. Thus, starting from the initial state, the searching algorithm generates successive states until a final state which represents the totally computed query is reached. The path with the lowest cost is selected as the optimal plan. We formalize the framework in the following.

[2] In the client-server ADMS± environment, incremental update is supported for downloaded data cached on client workstation disks.

3.4.1 Query Graph

We represent a PJS-query q as $q = (R_q, A_q, f_q)$, where $R_q = r_1, r_2, \ldots, r_{n_q}$ is the operand list (each r_i is the name of a base relation or a derived relation), $A_q = a_1, a_2, \ldots, a_{l_q}$ are attributes projected from the relations (referred to as the *target-list*), and f_q is a boolean formula for which the resulting tuples must satisfy (usually called the *qualification*). In the CMO optimizer, we use a graphic representation called *query graph* to model the queries.

Definition 1 A *query graph* (or a *state*) is a connected, undirected graph $G = (V, E)$ where

1. Each node $x \in V$ denotes a relation, a cached temporary, or an intermediate result.

   x is associated with a projected attribute list π_x, where π_x is a subset of $schm(x)$, the schema of x.

2. Each *hyperedge* $e \in E$ connects a subset of nodes $V_e \subseteq V$, and is labelled with a boolean formula f_e. □

Query graphs are used to model the original query as well as any partially processed queries during the optimization. More precisely, a query graph (V, E) represents a query $q = (R_q, A_q, f_q)$ where $R_q = V, A_q = \cup_{x \in V} \pi_x$, and $f_q = \wedge_{e \in E} f_e$. Note that the above semantic implies that a query graph usually represents a query whose qualification is a conjunction of sub-formulas. This will not lose any generalization because a formula can always be transformed into a conjunctive normal form [CL73]. Therefore, a query can always be represented by a query graph. We say e is a *k-connector* if it connects k nodes, i.e., $|V_e| = k$. An edge is a *join edge* if $k \geq 2$ and is a *selection edge* if $k = 1$.

In the following, we introduce the concept of *induced sub-query (graph)*, which is the basic unit to be collapsed during a query graph transformation (see Section 3.4.3).

Given a state $q = (V, E)$ and a join edge $e \in E$, let Ext_e be the set of edges which connect at least one node from V_e with at least another node *not* in V_e,

$$Ext_e = \{e' \in E \mid \exists x_1, x_2 \in V_{e'} \textit{ such that } x_1 \in V_e \textit{ and } x_2 \notin V_e\}.$$

We use $attr(Ext_e)$ to denote the set of attributes that appear in the formulas of Ext_e,

$$attr(Ext_e) = \{a \mid a \textit{ is an attribute appearing in } f_{e'}, \textit{ where } e' \in Ext_e\}.$$

Also, let Int_e be the selection edges incident on any node in V_e,

$$Int_e = \{e' \in E \mid V_{e'} = \{x\} \textit{ and } x \in V_e\}$$

Definition 2 Given a query graph $G = (V, E)$, the *sub-query induced* by a join edge $e \in E$ is a query $q_e = (R_{q_e}, A_{q_e}, f_{q_e})$ such that

1. $R_{q_e} = V_e$
2. $A_{q_e} = \bigcup_{x \in V_e} \pi_x \bigcup(schm(V_e) \wedge attr(Ext_e))$
3. $f_{q_e} = f_e \bigwedge(\bigwedge_{e' \in Int_e} f_{e'})$ □

The query graph corresponding to an induced sub-query can be accordingly defined and is called the *induced sub-query graph.*

3.4.2 Logical Access Path Schema

To facilitate the searching of useful temporaries from the cache pool, a structure called Logical Access Path Schema (LAPS) [Rou82a] is adopted. The LAPS is used to keep track of the cached temporaries efficiently. Instead of recording each cached temporary independently, the LAPS integrates the cached temporaries along with their logical access paths which captures the logical and derivation relationships among the temporaries.

Formally, a LAPS is a directed graph whose nodes, which reference to existing base relations and cached temporaries, are connected with edges that represent the derivation paths. In the following definition, we use $v = (R_v, A_v, f_v)$ to denote that temporary v is directly computed from R_v, which may contain some other temporaries, without producing any intermediate results in between.

Definition 3 A LAPS is a *directed* graph $LAPS = (N, E)$ where N is a set of nodes corresponding to base relations and cached temporaries, E is a set of directed *hyperedges* corresponding to logical access paths, and for any temporary $v = (R_v, A_v, f_v) \in N$,

1. $R_v \subset N$, and
2. there exists a hyperedge $e = (R_v, v) \in E$, which leads from the set of operand nodes R_v toward v and is labelled with f_v. □

Initially, the LAPS contains base relations only. When subsequent queries are processed, it is augmented by integrating the cached temporaries along with their logical access paths. The integration of new cached temporaries and logical access paths into the LAPS is straightforward and has been developed in [Rou82a]. A LAPS subcomponent has been embedded in the CMO and allows the coexistence of multiple and equivalent caches which may have been derived from different paths.

3.4.3 Query Graph Reductions

There are two kinds of reductions: the *selJoin-reduction*, which corresponds to performing a regular join with selection(s) done on the fly (called a *selJoin* operator), and the *match-reduction*, which corresponds to using a matched cached temporary. Both reductions reduce a query graph by replacing an induced sub-query graph with a new node, which might have an incident selection edge on it in the case of a match-reduction.

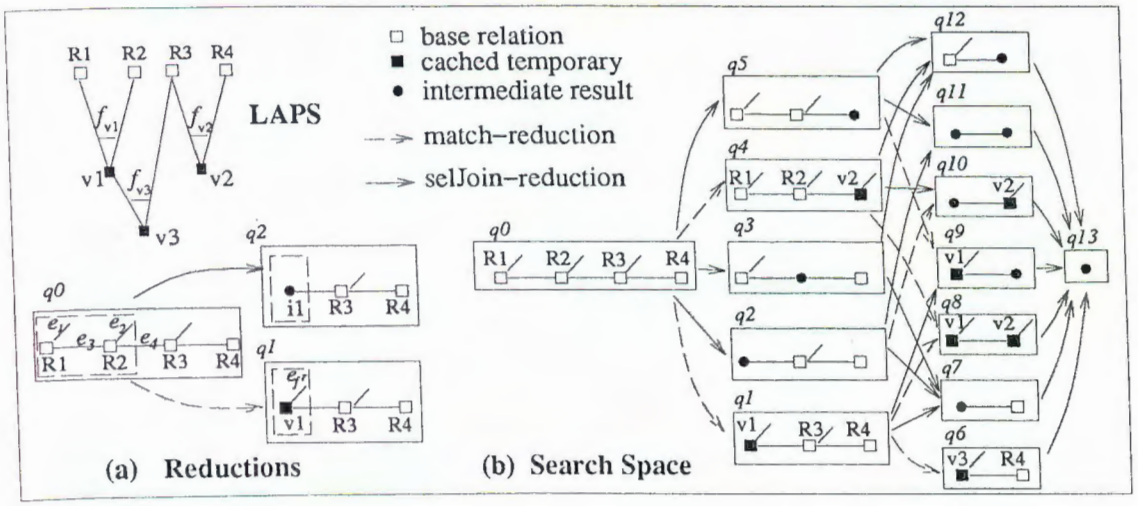

Figure 3.1: Query Graph Reductions and Searching

Definition 4 (selJoin Reduction) When applying a selJoin-reduction to a state $q = (V, E)$ on a join edge $e \in E$, the optimizer first (1) finds the optimal selJoin access path for evaluating the induced sub-query q_e, and then (2) reduces $q = (V, E)$ to a new state $q' = (V', E')$ such that

- $V' = V - V_e \cup \{v'\}$, where $v' \notin V$ is a new node and is labelled by a projected attribute list $\pi_{v'} = \bigcup_{x \in V_e} \pi_x$,
- $E' = E - \{e\} - Int_e - Ext_e \cup Ext'_e$, where Ext'_e is a new set of edges formed from Ext_e by replacing each occurrence of node in V_e with the new node v'. □

A selJoin-reduction is illustrated in Figure 3.1.(a), where state q_0 is reduced to q_2 on the join edge e_3. Note the induced sub-graph q_{e_3}, bounded by dashed rectangle in q_0, is replaced by a new node $i1$ in q_2 which corresponds to a new intermediate result. The optimal access path to evaluate the induced sub-query q_{e_3} is decided depending on the estimated costs of alternative join and selection access methods.

A match-reduction, instead of performing a regular join, uses an existing cached temporary to replace an induced sub-query of a query graph. However, for this to happen, we must find the cached temporary that can be used to *derive* the induced sub-

query. Formally, we say an induced sub-query q is *derivable* from a cached temporary v (or v is a *match* of q) if there exists an attribute set A and a formula f such that, for any database instance D,

$$q(D) = \pi_A(\sigma_f(v(D))),$$

where $q(D)$ denotes the result of q, and $v(D)$ denotes the content of v under database instance D respectively. A temporary $v = (R_v, A_v, f_v) \in LAPS$ is a match of a sub-query $q = (R_q, A_q, f_q)$ if it satisfies all the three conditions described below.

Condition 1 (Operand Coverability) $R_v = R_q$

Rather than using a looser condition that requires only the equivalence of the underlying base relations, this condition requires exactly the same set of parent operands. However, this will not miss any candidates when we capitalize on the LAPS and the match-reductions to identify the matched temporaries.

Condition 2 (Qualification Coverability) $\forall x_1, x_2, \ldots, x_n\ (f_q \rightarrow f_v)$, and, there exists a *restricting* formula f^r on v such that $\forall x_1, x_2, \ldots, x_n\ (f_q \leftrightarrow f_v \wedge f^r)$.

In the above formulas, $x_1, x_2, \ldots, x_n$ are the attributes appearing in the respective formulas in parentheses, and the universal quantifier '$\forall$' bounds these attributes variables to values from their respective domains. Symbols '$\rightarrow$' and '$\leftrightarrow$' stand for *logical implication* and *logical equivalent* respectively. This condition ensures that every tuple t in the result of q has a corresponding tuple t' in v such that t is a sub-tuple of t', and there exists a formula f^r through which these t' can be selected from v.

To check Condition 2, we need an algorithm to test if $f_1 \rightarrow f_2$ is true. This is known as the *implication problem* and is NP-hard even for a very restrictive problem instance [RH80, S$^+$89]. If the above test is true, we also need to find a restricting formula f^r which will make $f_1 \leftrightarrow f_2 \wedge f^r$ true. [LY85] and [Fin82] considered boolean formulas

which consist of only attributes, constants, comparison operators ($>, =, <$), and logical connectives ($\wedge$, $\vee$). This is satisfactory since a large set of database queries are in this form. [LY85] solved this problem by transforming it into a graph theory problem. They did not, however, give a constructive way to find the restricting formula, and the efficiency of their algorithm is not known. [Fin82] allowed *arithmetic additions* to appear in the formulas and referred to the *resolution* method. Resolutions are expensive theorem proving technique, and their effectiveness in this respect is not justified. We argue here that a simpler but weaker algorithm is more appropriate than those complicated and cost prohibitive ones. For this reason, we embedded in CMO a matching algorithm that is *sound*, in the sense that it answers affirmatively only when the implication is valid, but *not complete*, in the sense that it may respond negatively for some valid statements. Note that this will not affect the correctness of the query matching except that in certain cases, mostly when query predicates are complicated, some useful cached temporaries might be missed for potential optimization. In the following, we describe the theoretical basis of this algorithm.

An *atom*, denoted by A_i, is a predicate of the form $x\ \theta\ y$, where x is an attribute, y is either an attribute or a constant, and θ is a comparison operator ($>, =, <$). A *clause* is a disjunction of atoms, denoted as $C = A_1 \vee A_2 \vee \ldots A_n$. Using elementary logic [CL73], we can transform any boolean formula into a conjunctive form as $f = C_1 \wedge C_2 \wedge \ldots C_m$. The following lemma gives sufficient conditions for checking $f_1 \rightarrow f_2$ when both f_1 and f_2 are in conjunction form. We omit the universal quantifiers and the quantified attribute variables in the following formulas, as they are clear from the content.

Lemma 1 *1. if for each $C_{2,j} \in f_2$, $f_1 \rightarrow C_{2,j}$, then $f_1 \rightarrow f_2$*

2. if there exists a $C_{1,i} \in f_1$ such that $C_{1,i} \rightarrow C_{2,j}$, then $f_1 \rightarrow C_{2,j}$

3. if for each $A_i \in C_{1,i}$, there exists a $A_j \in C_{2,j}$ such that $A_i \rightarrow A_j$, then $C_{1,i} \rightarrow C_{2,j}$

□

What remains to be solved is the testing of $A_i \rightarrow A_j$. Obviously, it evaluates to false if A_i and A_j contain different set of attributes. Otherwise, a look-up table, as shown in Table 3.1, is used to decide the truth value. This table is adopted from [Fin82], where he termed it *direct elimination.* If both atoms are selection predicates, let $A_i = x\ \theta_1\ c$ and $A_j = x\ \theta_2\ c'$. The entries in table (a) indicate the relationship between constants c and c' under which $A_i \rightarrow A_j$ is true; a blank entry means under no situations can it be true. If $A_i = x\theta_1 y$ and $A_j = x\theta_2 y$, where both x and y are attributes, a checked entry in table (b) indicates that $A_i \rightarrow A_j$ is true.

$\theta_1 \backslash \theta_2$	$=$	$<$	$\leq$	$>$	$\geq$
$=$	$=$	$<$	$\leq$	$>$	$\geq$
$<$		$\leq$	$\leq$		
$\leq$		$<$	$\leq$		
$>$				$\geq$	$\geq$
$\geq$				$>$	$\geq$

(a) Unary Atoms

$\theta_1 \backslash \theta_2$	$=$	$<$	$\leq$	$>$	$\geq$
$=$	$\surd$		$\surd$		$\surd$
$<$		$\surd$	$\surd$		
$\leq$			$\surd$		
$>$				$\surd$	$\surd$
$\geq$					$\surd$

(b) Binary Atoms

Table 3.1: Direct Elimination Tables for $A_i \rightarrow A_j$

When $f_1 \rightarrow f_2$ is true, we also need to find a restricting formula so that the tuples satisfying f_1 can be extracted from the tuples satisfying f_2. The simplest restricting formula is f_1 itself since $f_1 \leftrightarrow f_2 \wedge f_1$. However, in certain cases, only a subset of f_1 is required. This will save the evaluations of some redundant predicates during query execution. The following lemma tells how to form a restricting formula by eliminating redundant clauses from f_1.

Lemma 2 *Suppose $f_1 \rightarrow f_2$ is true. If there exists clauses $C_1 \in f_1, C_2 \in f_2$ such that $C_1 \leftarrow C_2$, then $f^r = f_1 - \{C_1\}$ is a restricting formula, i.e. $f_1 \leftrightarrow f_2 \wedge f^r$.* □

In the above lemma, $C_1 \leftrightarrow C_2$ can be checked by checking both $C_1 \rightarrow C_2$ and $C_2 \rightarrow C_1$ (as described in Lemma 1) . We have integrated Lemma 1 and 2 into an

algorithm so that if $f_1 \rightarrow f_2$ is true, the restricting formula f^r is computed at the same time and returned as the result. The algorithm is constructed directly from the above two lemmas using a nested loop, though in most cases only few iterations will be invoked.

Condition 3 (Attribute Coverability) $A_v \supseteq (A_q \cup attr(f^r))$, where $attr(f^r)$ are attributes appearing in f^r.

This condition assures that the schema of temporary v contains all the attributes that are to be projected in query q, as well as those required to evaluate f^r. It is not hard to see that if all the above three conditions are satisfied, $q(D) = \pi_{A_q}(\sigma_{fr}(v(D)))$ for all database instance D. That is, the result of query q can be derived from temporary v via a selection and a projection. Therefore, if q is an induced sub-query graph under consideration during the optimization, we can replace it with a node and a selection edge which correspond to v and f^r respectively.

Definition 5 (Match Reduction) Given a state $q = (V, E)$ and a join edge $e \in E$ under consideration. If the induced sub-query q_e is *derivable* from a temporary $v = (R_v, A_v, f_v) \in LAPS$ through a restricting formula f^r (i.e., the above three coverability conditions are satisfied), then we can *match-reduce* sate q to a new state $q' = (V', E')$ where

1. $V' = V - V_e \cup \{n_v\}$, $n_v \notin V$ is a new node corresponding to temporary v and is labelled with $\pi_v = \bigcup_{x \in V_e} \pi_x$,
2. $E' = E - \{e\} - Int_e - Ext_e \cup Ext'_e \cup \{e_{fr}\}$, where e_{fr} is a new selection edge incident to n_v and labelled with f^r. □

A match-reduction is shown in Figure 3.1.(a), where state q_0 is reduced to q_1 on edge e_3. The induced sub-query q_{e_3} is replaced by a cached temporary $v1$ from LAPS

and a selection edge e_{fr}, where $v1$ is a match of q_{e_3} and f^r is the corresponding restricting formula. Note that $schm(v_1)$, the schema of v_1, must contain those attributes from $R1$ and $R2$ which appear in f^r or f_{e_4}. This is assured in the attribute coverability check (Condition 3) for $v1$ and q_{e_3}.

3.4.4 The Search Engine

The search engine of the ADMS CMO optimizer adopts the *dynamic programming* technique to sequence the query graph reductions and find the optimal plan[3]. It performs a breadth-first search and restricts the state space by eliminating *isomorphic* states. Formally, two states $q_x(V_x, E_x), q_y(V_y, E_y)$ are isomorphic, denoted as $q_x \cong q_y$, if there exists a 1-to-1 mapping $\psi : V_x \rightarrow V_y$ such that for each pair of $v \in V_x$ and $\psi(v) \in V_y$, they either denote the same base relation or cached temporary, or they are both intermediate results and have the same set of underlying base relations[4].

We also customized the dynamic programming search so that the LAPS is traversed top-down in parallel with the applications of selJoin-reduction and match-reduction, and such that no node in the LAPS is visited more than once. This saves the effort of searching for potential matches in two ways: (1) once a temporary is known not to be a match, all its descendants can be rejected automatically without further checking, and (2) the qualification coverability checks are performed in an amortized style in the sense that for each temporary $v = (R_v, A_v, f_v)$, we involve only the incremental formula f_v (probably with some propagated restricting formulas), which is upon the immediate parent nodes R_v, instead of the expanded formula upon the base relations.

The ADMS CMO employs an extended cost model which takes into account the

[3] An A^* algorithm was used in an early version of the ADMS optimizer; however, experiments showed that it aided little in reduction of the searching space.

[4] Different definitions of isomorphism were implemented and experimented in ADMS. The one given here turned out to have a manageable searching overhead without too great a sacrifice in the quality of the output plan.

costs of pointer cache materialization and incremental updates. During the optimization of a query, the cost of a state q, denoted as $cost(q)$, is the least cost among all paths leading from the initial state to q. The search algorithm is outlined in the following.

Step 1 Let $q_0(V_0, E_0)$ be the initial state. Set $T := \{q_0\}$, $S := \emptyset$. Repeat Step 2 for $|V_0| - 1$ times.

Step 2 $S := T$, $T := \emptyset$. For each state $q(V, E) \in S$, and each join edge $e \in E$, do the following,

2.1 Apply selJoin-reduction to q on e, and let q_1 be the reduced state; Apply match-reduction to q on e if applicable, and let q_2 be the reduced state.

2.2 If there exists no $q' \in T$ such that $q' \cong q_1$, then $T := T \cup \{q_1\}$. Otherwise, if $q' \cong q_1$ (note there can be at most one such q') and $cost(q') > cost(q_1)$, set $T := T - \{q'\} \cup \{q_1\}$. Do the same thing for q_2.

Step 3 Output the path leading from q_0 to the final state in T as the optimal plan.

Figure 3.1.(b) draws the search space for the query and LAPS given in figure (a). The selJoin-reductions and match-reductions are drawn in solid and dashed arrows respectively. Isomorphic states are reflected by those arrows that lead to the same state. Three iterations are performed, with a final state generated at the lowest level. In this figure, q_1 is further match-reduced to q_6 by using a matched temporary v_3; q_8 corresponds to the plan of using two matched temporaries v_1 and v_2.

3.5 Performance Evaluation

We conducted an empirical study in order to evaluate the performance of CMO under a variety of query workloads. The experimental design and detailed results are given as follows.

3.5.1 Experiment Design

We implemented the CMO optimizer on top of the ADMS access method module which provides a single `binary selJoin` operator. A `binary selJoin` is a join operator between two tables with selections and projections done on the fly. If the inner table is not given, it becomes a straightforward selection operator on the outer table. Projection of a subset of attributes and duplicate elimination are supported on the fly during output using main memory hashing. Two access methods, sequential and index access, are provided for single table scan. Three join methods—nested loop, index, and hash join—are supported for the `binary selJoin` operator.

The experiments were carried out by running a centralized version of ADMS on a Sun SPARCstation 2. All the experiments were run under a single user stand-alone mode so that the system performance can be measured in terms of elapsed time. Different databases and query workloads were used throughout the experiments to observe the performance impact from the CMO parameters as well as from the system environment.

Databases

We created a set of synthetic relations based on the Wisconsin Benchmark schemas [BDT83]. Table 3.2 outlines the cardinality, and size of each relation. The set of short version is obtained from the regular one by eliminating the last two string attributes, which are 104 bytes in length. Throughout the experiments, each tested database

Relation	100	1k	2k	5k	10k
Cardinality	100	1,000	2,000	5,000	10,000
Size (KB)	20	208	408	1,008	2,016

Relation	100s	1ks	2ks	5ks	10ks
Cardinality	100	1,000	2,000	5,000	10,000
Size (KB)	10	95	200	496	976

Table 3.2: Synthetic Relations

Databases	Relations
DBMIX	1k, 2k, 5k, 10k
DBMIX-S	1ks, 2ks, 5ks, 10ks
DB100	100, 100′, 100″, 100‴
DB1k	1k, 1k′, 1k″, 1k‴
DB5k	5k, 5k′, 5k″, 5k‴

Table 3.3: Five Different Databases

consists of four relations from Table 3.2. Table 3.3 lists all the tested databases. The primes (′) indicate different relation instances of the same relation schema, cardinality and attribute value distributions.

Query Workloads and Characteristics

Synthetic query workloads are generated using a customized random query generator. By specifying desired query characteristics to the generator, different instances of *query streams* that all satisfy the given characteristics can be generated. In our experiment, each query stream has an equal number of single-table selections, 2-way joins, 3-way joins, and 4-way joins in it. These queries are randomly distributed within a query stream.

We restrict the number of permitted join attributes so that common sub-expressions can recur within a query stream and so that the effectiveness of CMO can be observed. We define *query correlation* of a query stream as the number of distinct equal-join predicates appearing in the query stream. Figure 3.2 shows three levels of query correlations used in generating the tested query streams. The circles denote the relations, the nodes denote the permitted join attributes, and the edges denote the permitted join predicates. Note that a maximum of 6, 16, and 24 distinct join predicates can be generated in HighQC, MedQC, and LowQC respectively. For each n-way join query,

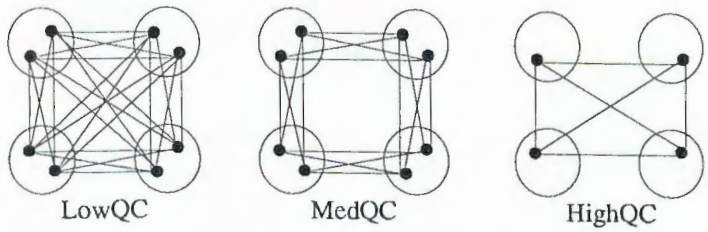

Figure 3.2: Three Levels of Query Correlation

the qualification statement is formed by $n-1$ binary join predicates and a number of random selection predicates. The join predicates are selected randomly from the permitted join predicates; the selection predicates are chosen from random attributes, and the selection ranges are chosen to satisfy a specified query selectivity. When not otherwise mentioned, the query selectivities are set over a wide range so that the cardinalities of query results range from less than a hundred to several hundred thousands.

To allow best utilization for query result caching, every query is projected on all attributes of its participating relations. This process makes no difference in pointer caching, but necessitates more space for data caching. Updates are restricted to modifications on the last three string attributes of each relation in order to keep the cardinalities roughly unchanged during the experiments. The qualification predicates in the update queries, however, are set against randomly chosen attributes. Throughout the experiments, each query stream contains at least 50 queries. When not mentioned explicitly, the defaults for the tested database and the query correlation are DBMIX and MedQC respectively.

Performance Metrics

The total *elapsed time* of a query stream, including query optimization time and query evaluation time, is taken as the main metric in evaluating the performance outcome. Throughout the whole experiment, each run (query stream) was repeated several times and the average elapsed time was computed.

3.5.2 Experiment Results

We performed three sets of experiments. The first set evaluated variations of CMO (with different cache management strategies) under different sizes of available cache space. The second set was conducted to observe the performance degradation of CMO (with three different cache update strategies) under different degrees of update rates. Finally, we observed the impact of database sizes and query characteristics on CMO performance. The optimization overhead of CMO, recorded from the three sets of experiments, was also compared to that of a standard optimizer.

Effect of Cache Management

In this set, we compared data caching (DC) with pointer caching (PC) using three different replacement strategies: LRU, LFU, and LCS. These are shown on the six curves labelled accordingly in each of the figures of Figure 3.3. Two databases, DBMIX and DBMIX-S, and two query streams, QS1 and QS2, were used in this set. We varied the available cache pool size and measured the elapsed time for the query streams.

In Figure 3.3.a (where DBMIX and QS1 are tested), pointer caching runs faster than data caching under all tested cache sizes ranging from 0 to 25 MB. This suggests that when a moderate amount of intermediate results are generated and written to and read from the disk, the materialization cost of PC is compensated by its efficient write cost. As the cache space increases, PC reduces the elapsed time more sharply than DC. In this case, with 2 MB cache space, all useful temporaries were cached under the PC/LCS strategy. In contrast, with even more than 10MB cache space, the performance of DC/LCS is still worse than that of PC/LCS with only 2MB. The inferiority of data caching can be attributed to its large overhead in writing and reading the intermediate results. To make data caching more competitive, the same experiment was performed again on a smaller database DBMIX-S whose tuple lengths were only almost half the length of the original ones. The results are shown in Figure 3.3.b,

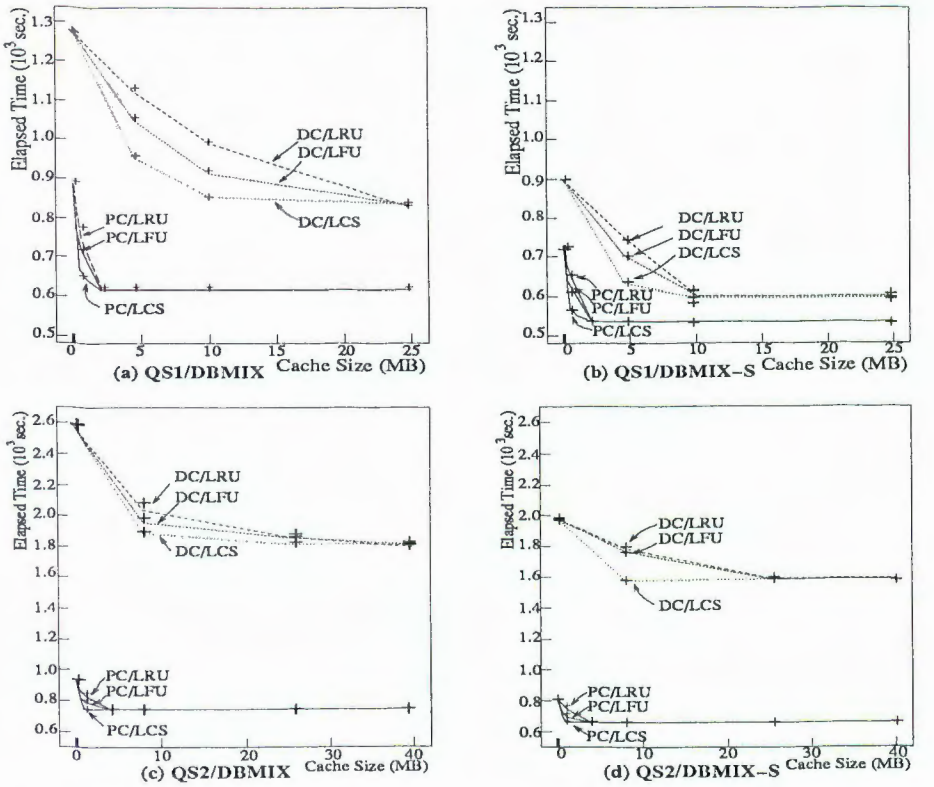

Figure 3.3: Effect of Cache Management on CMO Performance

where the curves of DC now better approximate those of PC. Figures 3.3.c and 3.3.d show the results from another query stream QS2 which, as can be seen, are similar to those from QS1. It is also worth mentioning that in all figures, the replacement strategies are consistently ordered with LCS as the best followed by LFU and then LRU. This suggests that cache space is the most critical factor in enhancing the cache utilization under random query workloads. It does not imply, however, that LCS is the best under all situations. For example, in a session where queries are issued by a user in response to previous query results, LRU might be the best choice.

For data caching, gains in available disk space and decrease in write cost can be achieved by not storing (i.e. projecting out) some of the non-useful attributes of the intermediate results. However, such a projection reduces the potential reuse of these intermediate results in other queries which may need these attributes. Thus, for data

caching, there is a dilemma between reducing the intermediate size and enhancing the chance of cache reuse. Pointer cache, on the other hand, does not have this problem at all, since all attributes are implicitly inherited in the non-materialized cache. If *cache utilization* is measured as the time reduced in query execution divided by the size of the cache pool, it is clear that pointer caching has much better cache utilization than data caching, according to the above results. For this reason, we used PC/LCS as the underlying cache management system and set the default cache size to 4MB for all the rest of our experiments.

Effect of Relation Updates

In this experiment, we evaluated the CMO performance degradation under relation updates. Three variations of CMO were evaluated under different degrees of update frequencies and selectivities. CMO/INC uses incremental update only, CMO/REX uses re-execution update only, and CMO/DYN chooses between incremental and re-execution methods, depending upon which is less expensive. The performance results of a standard optimizer STD, which uses no caching and matching technique, is also included for comparison.

Update frequency is defined as the number of update queries divided by the number of total queries in a query stream; *update selectivity* is defined as the number of tuples affected by an update query divided by the relation cardinality. We experimented with three levels of update selectivities: LS ($1\% - 5\%$), MS ($6\% - 10\%$), and HS ($6\% - 10\%$ for $2/3$ of the update queries, and $40\% - 50\%$ for the other $1/3$). Within each selectivity level, five degrees of update frequencies, including $0\%, 5\%, 10\%, 15\%$ and 25%, were tested. Therefore, for each raw query stream (which contains no update queries), 15 variations were produced by interleaving it with the different combinations of the three update selectivities and five update frequencies. The *query throughput*, measured as the average number of queries completed per minute, was used for the performance

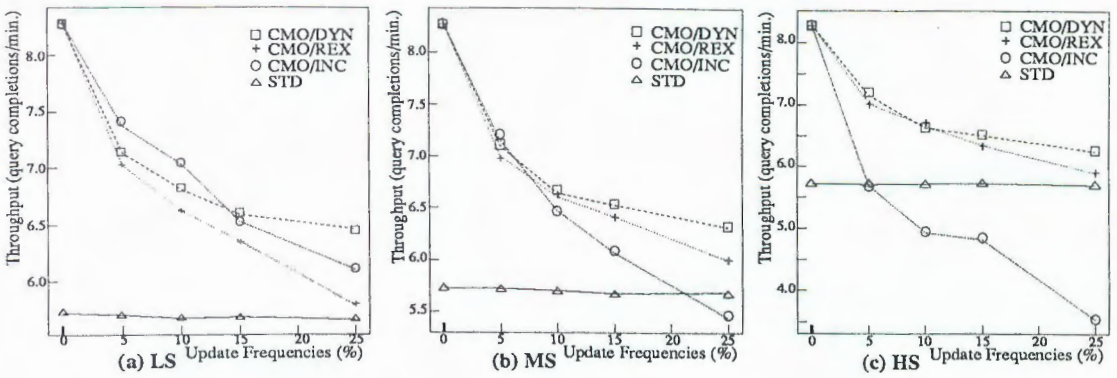

Figure 3.4: Comparison of Variations of CMO under Different Update Loads

comparison.

Figure 3.4 depicts the average query throughput (among four query streams) as a function of update frequencies, under three different update selectivity levels. It is obvious that the CMO curves, no matter what update strategies were used, performed better than STD in all figures (except for CMO/INC in the case of HS), and declined elegantly as update frequency increased. This is no surprise since the cost of updating an outdated temporary was amortized among those subsequent queries that were able to use it before it became outdated again. As update frequency increased, the cost was amortized among fewer queries and thus the throughput decreased.

In Figure 3.4.a, which depicts low update selectivity, CMO/REX performed worse than CMO/INC and CMO/DYN since the re-execution update did not take advantage of the incremental update scheme, which is very efficient under low update workloads. In this figure, CMO/INC is better than CMO/DYN at 5% and 10% update frequencies, but was outperformed by CMO/DYN at higher frequencies. This is attributable to the increasing cost of the incremental update log processing as a result of update frequency increase. In Figure 3.4.b, which depicts medium update selectivity, CMO/DYN performs the best except at the frequency of 5%. In this figure, CMO/INC swaps positions with CMO/REX from the LS figure. Lastly, in Figure 3.4.c, which depicts high selectivity, the throughput of CMO/INC declines drastically as update frequency

increases and becomes even worse than that of STD for update frequencies greater than 10%. In this set, CMO/DYN still performs the best.

The readers might wonder why CMO/DYN, which is theoretically the best scheme under all circumstances, is inferior to CMO/INC at update frequencies 5% and 10% in Figure 3.4.a. We analyzed the statistical profile and found that CMO/DYN sometimes chose less efficient paths than CMO/INC and/or CMO/REX. This was due to the inaccuracy of cost estimation; such inaccuracies may have caused CMO/DYN to choose incorrectly between incremental and re-execution updates when their costs were close. Another reason is that optimizing individual queries does not guarantee a global optimum over all queries. In effect, CMO/DYN might have incorrectly decided not to update an outdated cache (because of its high update cost) but rather to run from scratch, even if the high update cost of the outdated cache actually could be compensated by the time saved from other follow-up queries which could use it. These two problems, referred to as the problem of *accurate cost estimation* and *multiple query optimization*, are generic issues to all query optimization algorithms and are not within the scope of this work. Nonetheless, CMO is cost effective in most environments where queries do not arrive in batch and thus the technique of multiple query optimization can not be applied.

Effect of Query Correlation, Selectivities, and Database Sizes

In this set of experiments, we observed the impact of the database environment on CMO performance. Figure 3.5 shows the performance improvement of CMO over STD under the three levels of query correlation LowQC, MedQC, and HighQC. For each level, the results from three random query streams (QS1, QS2, QS3), each of which consists of 70 queries, are presented. These results demonstrate that CMO reduces the total elapsed time by a significant amount in all three correlation levels. In particular, the improvement increases as query correlation increases.

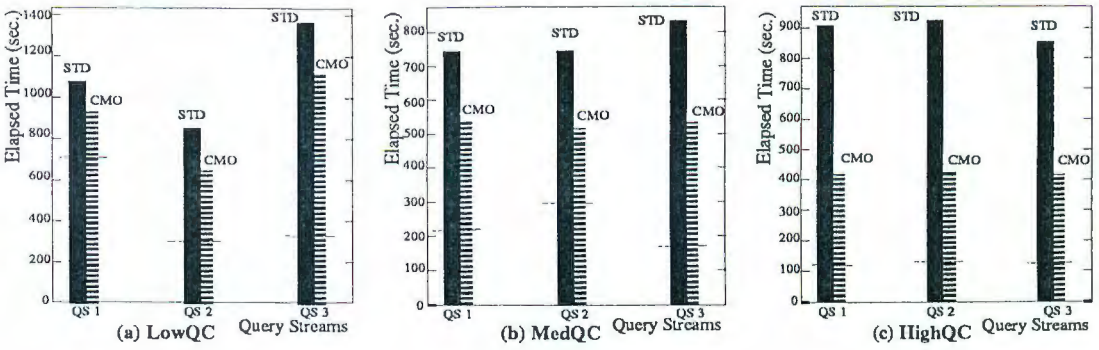

Figure 3.5: Comparison of CMO and STD under Different Query Correlations

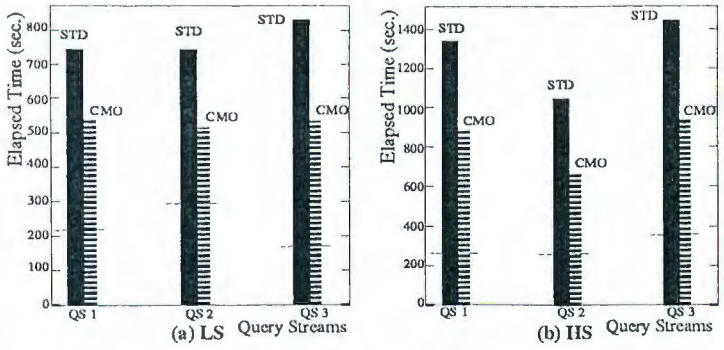

Figure 3.6: Comparison of CMO and STD under Different Query Selectivities

We also observed the effect of query selectivities. Figure 3.6 compares the results between two classes of query selectivities: low selectivities (LS) of $0.0001 - 0.05$ and high selectivities (HS) of $0.0001 - 0.3$. The results show that the relative performance improvement of CMO over STD in high selectivity (HS) is as good as that in low selectivity (LS), though the elapsed time has almost doubled in HS.

To see the effect of database size, three different databases were tested in another set of experiments. We adjusted the query selectivities for each query stream so that the query result sizes did not diverge greatly among the three database sizes. The purpose of doing so was to observe the improvement trends under different database sizes, under the supposition that the query result sizes were fairly small and unchanged. Figure 3.7 shows the results, where CMO consistently exhibited a shorter elapsed time

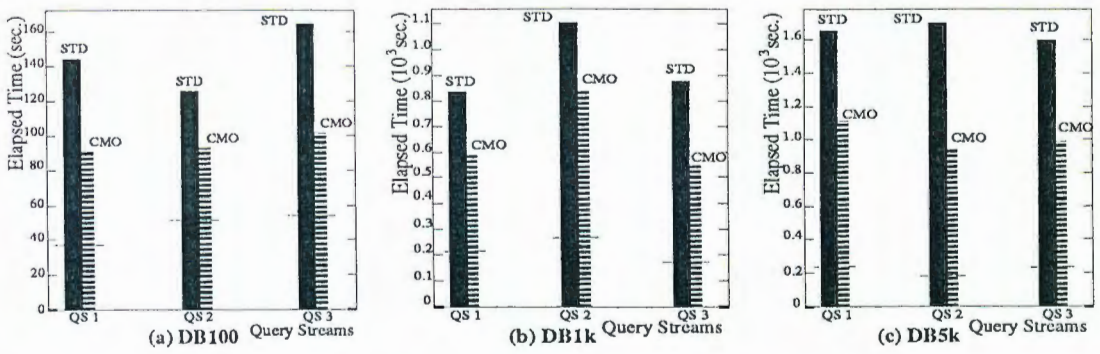

Figure 3.7: Comparison of CMO and STD under Different Database Sizes

than did STD. No drastic differences in relative performance improvement, however, can be observed among the three database sizes. From this observation, it is reasonable to argue that the relative performance gains of CMO over STD should remain as the database size scales up, so long as the cache space scales up proportionally.

Figure 3.8.a plots the average optimization overhead per query versus available cache size based on the statistics obtained from the experiments corresponding to the PC/LCS curves in Figure 3.3. The results at cache size 0 correspond to the runs where the caching and matching engine was not enabled. It can be seen that introducing the matching mechanism into the standard optimizer (as the cache size increases from 0 to 1 MB) does increase optimization overhead. This is attributable to the additional search space of matching. However, as the cache size increases from 1 to 10 MB, the optimization overhead does not increase accordingly. This is because when an intermediate cache is to be replaced, we can free its physical storage space but can not delete its entry from the system catalog or the LAPS, since the intermediate cache could be part of the logical access paths of other caches. Therefore, increasing cache size does not necessarily increase the number of entries in the LAPS, which number is the actual factor that affects the matching overhead; increasing cache size thus does not have much impact on the optimization overhead. Figure 3.8.b shows the average optimization overhead per query for the three different levels of query correlation. The

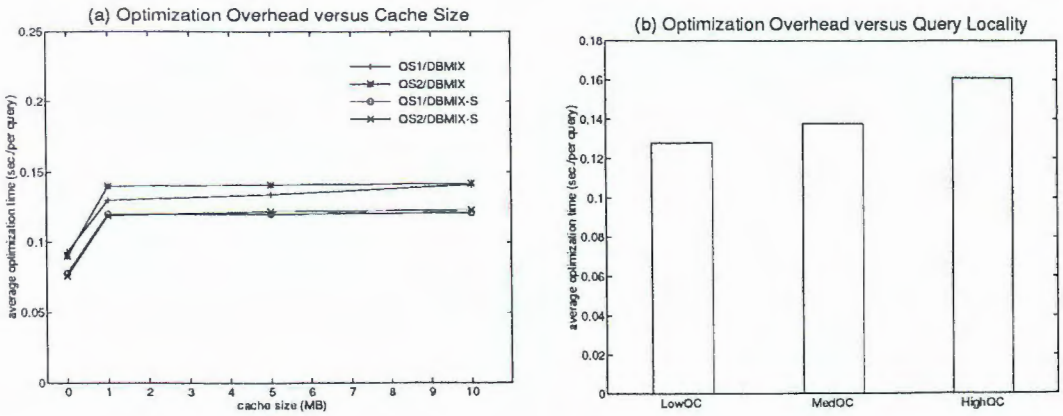

Figure 3.8: CMO Optimization Overhead

	Exp. Set 1	Exp. Set 2	Exp. Set 3
STD	0.084	0.087	0.088
CMO	0.133	0.135	0.149

Table 3.4: Average Optimization Overhead (second/per query)

overhead increases as query correlation increases since a high query correlation has a higher chance of cache re-use and thus larger optimization search space.

Finally, we compared the optimization overhead of CMO with STD. Table 3.4 lists the average optimization time per query for each set of experiments we described above. Though CMO incurs around 50% – 60% more optimization time than STD, the extra overhead introduced by CMO (an average of 1/100 – 1/10 sec. per query) is relatively small when compared to the time saved in query evaluation (an average of 10–100 sec. per query). Note that this has been demonstrated in all the above experimental results, where the elapsed time includes both query optimization and evaluation time.

We summarize the results as follows:

- Pointer caching is much better than data caching due to its compact size and much lower write cost.

- CMO with dynamic use of re-execution and incremental cache update strategies has the best performance under all query loads. It improves system performance under low to medium update loads and loses nothing under high update loads.
- CMO improves system performance under different query correlations, selectivities, and database sizes.
- The optimization overhead of CMO is insignificant. This justifies its practical use in general query loads, even when only few matches would occur.

3.6 Conclusions

We have described the design and implementation of the ADMS CMO query optimizer — an optimizer that is capable of matching and integrating in its execution plans query results cached from previous queries. Based on two kinds of query graph transformations, one of which corresponds to using a regular join and the other of which corresponds to using a matched cache, the optimizer performs a dynamic programming search strategy to generate optimal query plans. The optimizer also features data caching and pointer caching, different cache replacement strategies (LRU, LFU, and LCS), and incremental and re-execution cache update methods.

A comprehensive set of experiments was conducted using a benchmark database and synthetic queries. The results showed that pointer caching and dynamic cache update strategies substantially saved query execution time and thus increased query throughput under situations with moderate query correlation and update load. The requirement of disk cache space was relatively small, and the extra optimization overhead introduced by ADMS CMO was more than offset by the time saved in query evaluation. To the best of our knowledge, this work is the first of its kind that has been integrated and implemented in a DBMS product or prototype. It offers evidence that the technique of using cached query results in query optimization is advantageous

and can be implemented efficiently.

Chapter 4

Selectivity Estimation Using Query Feedback

4.1 Introduction

As demonstrated in Section 2.3, one of the most important factors that affects query plan cost is *selectivity*, which is the number of tuples satisfying a given predicate. Therefore, the accuracy of selectivity estimate directly affects the choice of best plan. A study on error propagation [IC91] revealed that selectivity estimation errors can increase exponentially with the number of joins and thus affect the decisions in query optimization. Accurate selectivity estimation has become even more important in today's systems of much larger database sizes, possibly distributed over a LAN or a WAN. In such systems, the query plans are expected to diverge much more in cost due to the database size and the volume of data transmission. Therefore, accurate selectivity estimation is even more crucial.

The issue of selectivity estimation has attracted popular interest, and different methods have been proposed [MO79, Chr83b, Chr83a, PSC84, KK85, HOT88, Lyn88, MD88, LN90, SLRD93, Ioa93]. Although accuracy is very important for selectivity estimates, the cost of obtaining such estimates must be confined if they are to be cost effective. In all the above methods, however, extra I/O accesses to the database are required for the very purpose of collecting statistics. This procedure might be expen-

sive and, as suggested, should be done off-line or when the system is light-loaded. In a static database where updates are rare or in a system where off-line (slack) time is affordable with a reasonable frequency (e.g. once a day), this overhead is acceptable. However, many systems do not afford to have sufficient slack time for collecting and maintaining the required statistics. Moreover, in the presence of updates, the procedure must be re-run either periodically or whenever the updates exceed a given threshold. This process not only incurs more overhead, but also degrades the query optimizer before the out-dated statistics are refreshed.

In this chapter, we present a novel approach which approximates the attribute value distribution using query feedbacks and totally avoids the overhead of statistics collection. The idea is to use subsequent *query feedbacks* to "regress" the distribution gradually, in the hope that as queries proceed, the approximation will become more and more accurate. We say that the adaptive approximation "learns" from the query executions in the sense that it not only "remembers" and "recalls" the selectivities of repeating query predicates, but can also "infer" (predict) the selectivities of new query predicates. This approach is advantageous in the following respects:

- Efficiency — Unlike the previous methods, no off-line database scans or on-line sampling are needed to form the value distribution. Also, unlike all the other methods where the statistics collection and computation overhead is proportional to the relation size, the overhead of our method has a negligible cost in constant time for each query feedback, regardless of the relation size.

- Adaptation — The technique we use here adapts the approximating value distribution to queries and updates. None of the previous methods achieve this. They neither take into account query information when approximating the value distribution (only relations are scanned), nor continuously adjust the distribution to updates (re-computation is invoked only after the updates exceed a threshold).

The rest of this chapter is organized as follows: Section 4.2 reviews the related work. Section 4.3 describes the adaptive selectivity estimator in detail. Section 4.4 presents the experimental results. Finally, summary is given in Section 4.5.

4.2 Related Work

The existing methods for selectivity estimation can be categorized into four classes: the *non-parametric method*, the *parametric method*, *sampling*, and *curve fitting*. In the following paragraphs, we review the essential approaches for each of these four classes. A detailed survey of the first two classes can be found in [MCS88].

Non-Parametric Method Methods in this class maintain attribute value distributions using ad hoc data structures and algorithms. The most common method is the *histogram*, which divides an attribute domain into intervals and counts the number of tuples holding values which fall into each of the intervals. Variations of the histogram method can be found in [MO79, PSC84, MD88, Lyn88, Ioa93]. The histogram is simple, but tradeoff between the computation/storage overhead and the estimation accuracy must be considered. Satisfactory accuracy will not be reached until the domain is divided into a sufficient large number of small intervals. In addition to the histogram, a pattern recognition technique was used by [KK85] to construct discrete cells of distribution table, and [Lyn88] used a keyterm-oriented approach to keep counts of the most frequently queried attribute values.

Parametric Method Parametric methods approximate the actual distribution with a mathematical distribution function of a certain number of free statistical parameter(s) to be estimated (we call such a function a *model* function). Examples of the model function include the uniform, normal, Pearson family and Zipf distributions. In these methods, statistics must be collected, either by scanning

through or by sampling from the relation, in order to estimate the free parameter(s). These methods usually require less storage overhead and provide more accurate estimation than non-parametric methods (if the model function fits the actual distribution). The disadvantage of this method is that the "shape" of the actual distribution must be known a priori in order to choose a suitable model function. Moreover, when the actual distribution is not shaped like any of the known model functions, any attempt to approximate the distribution by this method will be in vain. Contributions to research of parametric methods can be found in [S+79, SB83, Fed84, Chr83b, Chr83a].

Curve Fitting In order to overcome the inflexibility of the parametric method, [LST83] and [SLRD93] used a general polynomial function and applied the criterion of least-square-error to approximate attribute value distribution. First, the relation is exhaustively scanned, and the number of occurrences of each attribute value is counted. These numbers are then used to compute the coefficients of the optimal polynomial that minimizes the sum of the squares of the estimation errors over all distinct attribute values. Polynomial approximation has been widely used in data analysis; however, care must be taken here to avoid the problem of oscillation (which may lead to negative values) and rounding error[1] (which may propagate and result in poor estimation when the degree of the polynomial is high, say, more than 10).

Sampling The sampling method has recently been investigated for estimating the resulting sizes of queries. Sample tuples are taken from the relations, and queries are performed against these samples to collect the statistics. Sufficient samples

[1]The problem caused by rounding errors is usually termed a case of being *ill-conditioned*. This can always be avoided by representing the approximating polynomial with a more *numerically stable* basis. For example, the Legendre polynomials are used as the basis in [LST83].

must be examined before desired accuracy can be achieved. Variations of this method have been proposed in [HOT88, LN90, HS92]. Though the sampling method usually gives more accurate estimation than all other methods (suppose sufficient samples are taken), it is primarily used in answering statistical queries (such as `COUNT(...)`). In the context of query optimization where selectivity estimation is much more frequent, the cost of performing sampling on every predicate of queries is prohibitive and thus might prevent its practical use.

4.3 Adaptive Selectivity Estimation

In this section, we describe the implementation of an Adaptive Selectivity Estimator (ASE). At the heart of our approach is a technique called *recursive least-square-error* (RLSE), which is adopted to adjust the approximating distribution according to subsequent feedbacks. Before exploring the details, we first define some notations used throughout this paper.

Let A be an attribute of relation R, and let range $D = [d_{min}, d_{max}]$ be the *domain* of A. In this study, we consider only numerical domains (either discrete or continuous).[2] Let D' be the collection of all sub-ranges of D, and define $f_A : D' \rightarrow N$ as the actual distribution of A, i.e., for each sub-range $d \subseteq D$, $f_A(d) = |\{t \in R \ : \ t.A \in d\}|$ is the number of tuples in R whose values of attribute A belong to range d. Notice that the above notation is well-defined for both discrete and continuous cases. We denote a *selection query* $\sigma_{l \leq R.A \leq h}(R)$, where $l \leq h$, as $q = (l, h)$. The *selectivity* of query q, defined as $s = f_A([l, h])$, is the number of tuples in the query result. The *query feedback* from query q is then defined as $\zeta = (l, h, s)$.

[2] Non-numerical domains can be mapped into numerical ones using certain mapping techniques. The mapping functions should be provided by the database creators who know the semantic meaning of the attributes.

4.3.1 Customizing Recursive Least-Square-Error Approximation for Query Feedback

The goal of our approach is to approximate f_A by an easily evaluated function f which is able to self-adjust from subsequent query feedbacks. Thus, given a sequence of queries $q_1, q_2, \ldots$, we can view f as a sequence $f_0, f_1, f_2, \ldots$ where f_{i-1} is used to estimate the selectivity of q_i, and, after q_i is optimized and executed, f_{i-1} is further adjusted into f_i using feedback ζ_i (which contains the actual selectivity, s_i, of query q_i obtained after the execution).

We use a general form $f(x) = \sum_{i=0}^{n} a_i \phi_i(x)$ as the underlying approximating function, where $\phi_i(x)$, $i = 0, \ldots, n$, are $n+1$ pre-chosen functions (called *model functions*), and a_i are coefficients to be adjusted from the query feedbacks. The corresponding *cumulative* distribution of $f(x)$ is given as $F(x) = \sum_{i=0}^{n} a_i \Phi_i(x)$, where $\Phi_i(x)$ is the indefinite integral of $\phi_i(x)$. Using this form of approximation, the estimated selectivity of query $q = (l, h)$, denoted by $\hat{s}$, is computed as:

$$\hat{s} = \int_{l}^{h+1} f(x)dx = F(h+1) - F(l) = \sum_{j=0}^{n} a_j [\Phi_j(h+1) - \Phi_j(l)].$$

Now suppose a sequence of query feedbacks $\zeta_1, \ldots, \zeta_m$, where $m \geq n$, have been collected. A reasonable criterion for tuning $f(x)$ is to find the optimal coefficients a_i that minimize the sum of the squares of the estimation errors (thus referred to as *least-square-error* (LSE)):

$$\sum_{i=1}^{m} (\hat{s}_i - s_i)^2 = \sum_{i=1}^{m} (\sum_{j=0}^{n} a_j [\Phi_j(h_i + 1) - \Phi_j(l_i)] - s_i)^2. \tag{4.1}$$

The above problem can be reformulated in linear algebra form as:

$$\textit{find the optimal } A \textit{ that minimize } ||X * A - Y||^2, \tag{4.2}$$

where $||\cdot||^2$ denotes the sum of the squares of all elements in the vector, and

$$X = \begin{bmatrix} \Phi_0(h_1+1)-\Phi_0(l_1) & \dots & \Phi_n(h_1+1)-\Phi_n(l_1) \\ \Phi_0(h_2+1)-\Phi_0(l_2) & \dots & \Phi_n(h_2+1)-\Phi_n(l_2) \\ \dots & \dots & \dots \\ \Phi_0(h_m+1)-\Phi_0(l_m) & \dots & \Phi_n(h_m+1)-\Phi_n(l_m) \end{bmatrix}, Y = \begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_m \end{bmatrix}, A = \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_n \end{bmatrix}. \tag{4.3}$$

Let X^t be the transpose of X, the solution to Eq. 4.2 is obtained as [KMN89]:

$$A^* = (X^t X)^{-1} X^t Y. \tag{4.4}$$

The above computation has the drawback that the space requirement of X and Y increases in proportional to the number of query feedbacks m, and each time a new query feedback is added, the whole thing must be re-computed. This concern can be relieved with some rearrangement of the above computation. Let $P = X^t X$ and $N = X^t Y$. It is not hard to see that P is a $n \times n$ matrix and N is a $n \times 1$ vector—both of whose dimensions are independent from the number of feedbacks m. A more careful look into P and N shows that

$$P = X^t X = \sum_{i=1}^{m} X_i^t X_i, \quad N = \sum_{i=1}^{m} X_i^t s_i, \tag{4.5}$$

where X_i is the ith row of X, and X_i^t its transpose. Now, let $\zeta_1, \zeta_2, \dots, \zeta_i, \dots$ be a sequence of query feedbacks, and A_i^* be the optimal coefficients of $f(x)$ corresponding to the first i feedbacks. According to Eq. 4.4 and 4.5 we have

$$A_i^* = P_i^{-1} N_i, \text{ for } i = n+1, n+2, \dots \text{ where} \tag{4.6}$$

$$P_i = P_{i-1} + X_i^t X_i, \quad N_i = N_{i-1} + X_i^t s_i, \text{ for } i = 1, 2, \dots, \tag{4.7}$$

with initial condition $P_0 = N_0 = 0$. Note that for $i \leq n$, P_i^{-1} dose not exist and thus a default distribution (e.g., uniformity) must be used temporarily. Later in this context, we will relax this restriction. Also notice that by using Eqs. 4.6 and 4.7, only two constant size arrays, P and N, need to be maintained.

The above equations can be further transformed into another form where the expensive matrix inversion P_i^{-1} need not be explicitly computed. [You84] derived the following recursive formulas, referred to as *Recursive Least-Square-Error* (RLSE), from Eqs. 4.6 and 4.7 :

$$A_i^* = A_{i-1}^* - G_i X_i^t (X_i A_{i-1}^* - s_i), \tag{4.8}$$

$$G_i = G_{i-1} - G_{i-1} X_i^t (1 + X_i G_{i-1} X_i^t)^{-1} X_i G_{i-1}, \tag{4.9}$$

for $i = 1, 2, \ldots$, while A_0 and G_0 can be of any arbitrary values. In this expression, no explicit matrix inverse operation is needed, and only an $n \times n$ matrix G (called a *gain matrix*) needs to be maintained (actually, $G = P^{-1}$). The computation complexity is in the order of $O(n^2)$. Since n is a pre-chosen small integer, the computation overhead per query feedback is small and is considered constant, regardless of the relation size. The initial values G_0 and A_0 may affect the convergence rate of A_i^* and, thus, the rate at which f_i converges to f_A. We describe later in this section how to initialize G_0 and A_0 with appropriate values. It is interesting to see that the computation of A_i^* resembles the technique of *stochastic approximation* [AG67], in the sense that A_i^* is adjusted from A_{i-1}^* by subtracting a *correction term* which is the product of the estimation error $(X_i A_{i-1}^* - s_i)$ and the *gain* value $G_i X_i^t$. Because of their simplicity and efficiency in both space requirement and computation, Eqs. 4.8 and 4.9 were adopted in the ASE.

4.3.2 Accommodating Update Adaptiveness

The RLSE can be further generalized to accommodate adaptability to updates. We accomplish this by associating different *weights* with the query feedbacks so that the outdated feedbacks can be suppressed by assigning smaller weights to them. In Eq. 4.1, we now associate an *importance weight* β_i to the estimation error of the ith query, and a *fading weight* α_i to the estimation errors of all the preceding queries. That is, instead

of minimizing Eq. 4.1, we now want to minimize:

$$\sum_{i=1}^{m}[(\prod_{j=i+1}^{m}\alpha_j)\cdot\beta_i\cdot(\hat{s}_i - s_i)]^2. \tag{4.10}$$

The recursive solution to the above is similar to Eqs. 4.8 and 4.9 (see Appendix A for derivation detail):

$$A_i^* = A_{i-1}^* - \beta_i^2 G_i X_i^t (X_i A_{i-1}^* - s_i), \tag{4.11}$$

$$G_i = (\frac{1}{\alpha_i})^2 G_{i-1} - (\frac{\beta_i}{\alpha_i})^2 G_{i-1} X_i^t (\alpha_i^2 + \beta_i^2 X_i G_{i-1} X_i^t)^{-1} X_i G_{i-1}, \tag{4.12}$$

for $i = 1, 2, \ldots$ Intuitively, β_is determine the "importance" of individual feedbacks; α_is determine the "forgetting" rate of previous feedbacks. Note that Eqs. 4.8 and 4.9 offer a special case of Eqs. 4.11 and 4.12 with $\alpha_i = \beta_i = 1$, for all i. Apparently, different weights affect the adaptation behavior of the approximating function. As an innovation, we consider only fixed-value weights. We set $\beta_i = \alpha_i = 1$ for all i, except that α_i is assigned another positive number less than 1 if ζ_i is the first feedback after update. The smaller the α_i, the more the knowledge from previous feedbacks is to be forgotten. Note that we cannot set $\alpha_i = 0$, because it appears as a denominator in Eq. 4.12. Nonetheless, the same effect (of discarding all previous knowledge) can be achieved by assigning an extremely small number to α_i. Experiments with different values of α_i are given in the next section.

We would like to point out that the weighting scheme we described here is strongly related to the so called "moving window" weighting technique used in the *time series analysis* [You84]. The "moving window" weighting technique adapts the function only to the most recent S (the window size) outcomes. In our weighted formula, however, we do not keep a moving window, but rather fade out the memory, which records the knowledge of all the previous outcomes, with a fading weight each time as an update occurs.

Initializing A_0 and G_0

The initial values of G_0 and A_0 must be determined before the recursive formulas in Eqs. 4.11 and 4.12 can be used. Theoretically, arbitrary initial values can be used for G_0 and A_0 [You84], though they differ greatly in convergence rates. To speed up convergence, we compute $G_0(=P_0^{-1})$ and A_0^* using Eqs. 4.4 and 4.5 by substituting the following $(n+1)$ *manual* feedbacks into Eq. 4.3:

$$l_i = h_i = d_{min} + (i-1) * \frac{(d_{max} - d_{min})}{(n-1)}, \quad s_i = \frac{|R|}{(d_{max} - d_{min})}, \quad i = 1 \ldots n \tag{4.13}$$

$$l_{n+1} = d_{min}, \; h_{n+1} = d_{max}, \qquad s_{n+1} = |R|, \tag{4.14}$$

where $|R|$ denotes the number of tuples in relation R. The intention here is to force ASE to begin with a uniform distribution (enforced by Eq. 4.13), and to keep knowledge of the relation cardinality in the gain matrix (enforced by Eq. 4.14).

Choosing the Model Functions

The remaining problem now is to choose the model functions $\phi_i(x)$. The polynomial function is a good candidate due to its generality and simplicity and has been used in [LST83] and [SLRD93]. We adopted polynomials of degree 6 throughout our experiments, i.e., the approximating function is of the form $f(x) = \sum_{i=0}^{6} a_i x^i$. Whereas polynomials of higher degrees have the potential problem of being ill-conditioned, polynomials of lower degrees might not be flexible enough to fit the variety of actual distributions. Therefore, our choice of degree 6 is a compromise between these concerns [3]. Another interesting class of functions is the *spline* functions [dB78], which are piecewise polynomial functions. Splines have many advantages over polynomials in the aspects of adaptability and numerical stability. However, they are more complex

[3] In our experiments using degree 6, the "ill-conditioned" problem did not arise. However, for higher degrees we might need to use another basis (such as Legendre polynomials or B-splines) since the basis of $x^i, i = 1, \ldots, n$ is in general ill-conditioned for large values of n.

Variables

f: a polynomial of degree 6; F: the indefinite integral of f;

A: the (adaptable) coefficients of f; G: the gain matrix;

Initialization

Use the manual feedbacks listed in Eqs. 4.13 and 4.14 to compute the initial values for A and G from Eqs. 4.3, 4.4 and 4.5.

Selectivity Estimation

The selectivity of query $q_i = (l_i, h_i)$ is estimated as $F(h_i + 1) - F(l_i)$; if it is negative, simply return 0.

Feedback and Adaptation

After the execution of q_i, get feedback $\zeta_i = (l_i, h_i, s_i)$ where s_i is the actual selectivity of q_i obtained from execution. If q_i is the first query after the latest update, set the fading weight α_i to a positive number less than 1. Use ζ_i to adjust A and G, as shown in Eqs. 4.11 and 4.12.

Figure 4.1: Outline of ASE

in computation and particularly in representation. We are currently investigating this approach and will not discuss it here.

A practical problem of polynomials is the negative values which are undesired in distribution approximations. This poses no problem so long as the negative values occur only outside the attribute domain, or so long as the resulting estimated selectivity of the query of interest is still positive (even if some negative values do occur within the domain). If a negative selectivity is ever estimated for a query, we simply use zero instead (and note that if the error is large, it will be tuned through feedback). Finally, we summarize ASE in Figure 4.1.

Comparison with [SLRD93]

Sun, Ling, Rishe, and Deng proposed in [SLRD93] a method of approximating the attribute distribution using a polynomial with the criterion of least-square-error. While both their method and ours use polynomial approximations, there are several differences between the two methods. First, their approach is *static* in the sense it is necessary to scan the database and count the frequencies of distinct attribute values, and, once computed, the approximating distribution remains unchanged until the next re-computation. Our method is *dynamic* and depends only on query feedbacks, with no access to the database. For a relation which is large and/or is updated regularly, the overhead of collecting or refreshing the statistics can be very expensive. Our approach totally avoids such overhead. Besides, in an environment where queries exhibit highly temporal or spatial locality on certain attribute ranges, ASE's dynamic adaptation to queries will perhaps be of greater benefit. Finally, ASE's adaptiveness to updates not only eliminates the overhead of statistics re-collection, but also provides a more graceful performance degradation for selectivity estimations through a query session interleaved with updates.

4.3.3 An Example

We use an example to demonstrate how the ASE works by using successive query feedbacks to approximate the data distribution. The experimental data is from a movie database, courtesy of Dr. Wiederhold of Stanford University, which records 3424 movies produced during the years 1890–1989. Figure 4.2 snapshots the evolution of the approximating distribution for a sequence of query feedbacks. The queries are listed in the table, where $[l_i, h_i]$ denotes the range of the ith query, and $\hat{s_i}$ and s_i denote the selectivities estimated (by ASE) before and obtained after the query execution respectively. In each frame, the curve of the approximating distribution f_i, drawn in solid line, is compared to the real distribution, drawn in discrete points.

query sequence	1	2	3	4	5
$[l_i, h_i]$	[1935,1966]	[1925,1950]	[1904,1939]	[1890,1923]	[1908,1913]
$\hat{s_i}$	1073	1138	1248	567	2
s_i	1872	1399	890	136	14

6	7	8	9
[1948,1989]	[1957,1980]	[1964,1989]	[1916,1981]
1956	1103	1041	3173
2033	1130	1134	3045

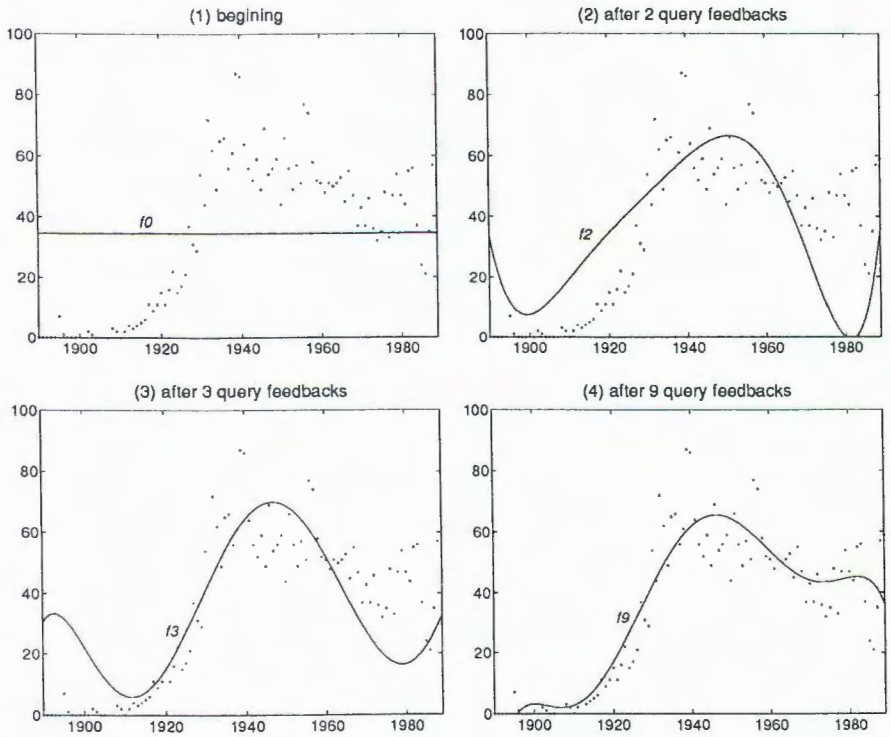

Figure 4.2: Adaptation Dynamics of ASE — an Example

In frame 1, uniform distribution is assumed at the very beginning, as no queries have been issued. Note that knowledge of the relation cardinality (3424 tuples) has been implicitly enforced in the initial approximating distribution f_0, using the initialization scheme explained in the previous section. After the execution of two queries, as shown in frame 2, the approximating curve becomes closer to the actual distribution. However, f_2 is relatively inaccurate for attribute ranges outside [1925, 1966] which have not been queried yet (and, thus, no distribution information is yet known). The third query and its feedback $\zeta_3 = (1904, 1939, 890)$ tunes f_2 into f_3 with better accu-

racy for range [1904, 1939]. It is worth mentioning that at the same time, f_3 improves the distribution of years greater than 1966, though no queries against this range have ever been posed. This is attributed to ASE's ability to infer and properly shape the unknown ranges using knowledge about the relation cardinality and distribution information obtained from queries on other attribute ranges. Subsequently, frame 4 shows the curve after nine query feedbacks, by which time the approximation has become even closer to the real distribution.

4.4 Experimental Results

A comprehensive set of experiments was performed to evaluate the ASE. We ran the experiments using the mathematics package MAPLE, developed by the Symbolic Computation Group of the University of Waterloo; MAPLE was chosen for its provision of immediate access to matrix operations and random number generators. We experimented also with the method proposed in [SLRD93] (referred to as SLR in what follows) for comparisons whenever appropriate. The selectivity estimation errors and the adaptation dynamics of ASE were observed and graphed for demonstration. However, to interpret and compare the estimation errors correctly, both *normalized error* and *relative error* are presented; they are calculated as:

$$\text{nor. err.} = \frac{|\hat{s} - s|}{|R|} \times 100, \quad \text{rlt. err.} = \frac{|\hat{s} - s|}{s} \times 100,$$

where $\hat{s}$ and s are the estimated and actual query result sizes, respectively; $|R|$ is the cardinality of the queried relation. Our reason for using both is that neither one alone can provide evidence of good or poor estimation in all cases. For example, a 200% relative error for a query of selectivity of 1 tuple by no means represents a poor estimate; in fact, it is the stringent selectivity (of 1 tuple) that causes such an exaggerated relative error. It must be pointed out that we do not compare the computation overhead since our method, which costs only negligible CPU time for query feedback computation, is definitely superior to all other methods which require

extra database accesses (either off-line or on-line) for statistics gathering or sampling.

Both real and synthetic data were used in the experiments. The use of real data validates the usefulness of our method in practice (as has been demonstrated in the example); the use of synthetic data allows systematic evaluation of ASE under diverse data and query distributions. Throughout the experimentation, only selection queries were considered. Each query is represented as a range $[x - \delta/2, x + \delta/2]$, where $d_{min} \leq x \leq d_{max}$, $0 \leq \delta \leq d_{max} - d_{min}$. In this paper, we report only results from those experiments where x and δ are generated randomly from their respective domains using a random number generator. Experimental results regarding the impacts of different distributions of x and δ on the convergence rate of ASE are prepared in a more detailed version of this paper.

Three sets of experimental results are presented here. The first set shows the adaptability of ASE to various data distributions. The second set shows how ASE adapts to *query locality*, in the sense that it provides more accurate selectivity estimates for the attribute sub-ranges which are queried most. In the last set, we demonstrate ASE's elegant adaptation through database updates which require no overhead for database re-scan and statistics re-computation.

4.4.1 Adaptiveness to Various Distributions

To observe ASE's adaptability to various data distributions, synthetic data generated from each of the following four customized distributions were tested: normal distribution, chi-square distribution, the F distribution, and a "bi-modal" distribution.[4] The notations and customized parameters of each distribution are described in Tables 4.1 and 4.2. For each data distribution, three random query streams (each of which contains 50 queries) were run for both ASE and SLR.

[4] We do not present the results of uniform distribution since the ASE assumes uniform distribution from the very beginning.

Notations	Meaning
$N(\mu, \sigma)$	Normal distribution with mean μ, standard deviation σ
$\chi^2(n)$	chi-square distribution with n degrees of freedom
$F(m, n)$	F distribution with m/n degrees of freedom for numerator/denominator
$B(u_1, \sigma_1, u_2, \sigma_2)$	a bi-modal distribution which is an overlap of $N(u_1, \sigma_1)$ and $N(u_2, \sigma_2)$

Table 4.1: Notations of Distribution

Distribution	$[d_{min}, d_{max}]$	cardinality
$N(200, 150)$	$[-150, 550]$	10,000
$\chi^2(10)$	$[0, 1200]$	20,000
$F(10, 4)$	$[0, 800]$	10,000
$B(250, 150, 450, 50)$	$[-150, 550]$	12,500

Table 4.2: Customized Parameters for Experimental Distribution

	1st - 50th queries				10th - 50th queries			
	ASE		SLR		ASE		SLR	
	nor. err.	rlt. err.	nor. err.	rlt. err.	nor. err.	rlt. err.	nor. err.	rlt. err.
N	0.73	4.43	0.16	2.4	0.16	3.66	0.16	2.73
χ^2	1.36	13.0	0.33	8.0	0.33	8.36	0.40	8.93
F	2.2	28.6	1.7	28.2	1.10	15.3	1.76	30.1
B	1.40	8.75	0.60	3.08	0.80	5.11	0.60	3.13

Table 4.3: Estimate Errors of ASE and SLR under Various Data Distributions

Table 4.3 lists the average error per query of ASE and SLR under each data distribution. In order to achieve a fair comparison between ASE and SLR, the average errors, which exclude the first 10 queries of each query stream (during which ASE is still in its "learning" stage), are also calculated for comparison. The first set of columns shows that ASE is slightly inferior to SLR in estimation accuracy; however, the second set of columns shows that after ASE converges (after 10 queries), its accuracy is very comparable to that of SLR. Figures 4.3 through 4.6 depict the corresponding dynamics of ASE and SLR for one of the query streams under each data set. In the figures to the left marked (a), curve g corresponds to the approximating distribution computed from SLR; f_i denotes the adaptive approximating distribution from ASE after i query

feedbacks. Figures (b) compare the estimation errors of ASE and SLR by plotting them along with the query streams. The adaptiveness of ASE can be clearly observed from the decreasing trend of errors as queries proceed. The occasionally high relative errors of ASE are either caused by stringently small selectivities (as evidenced by the high relative errors of SLR for the same queries), or are indications of the moments where feedbacks take place for the first time on the queried ranges. However, as can be seen from all the figures, after sufficient query feedbacks have covered the whole attribute domain, ASE converges the approximating distribution to a stable curve and provides estimations with constantly small errors.

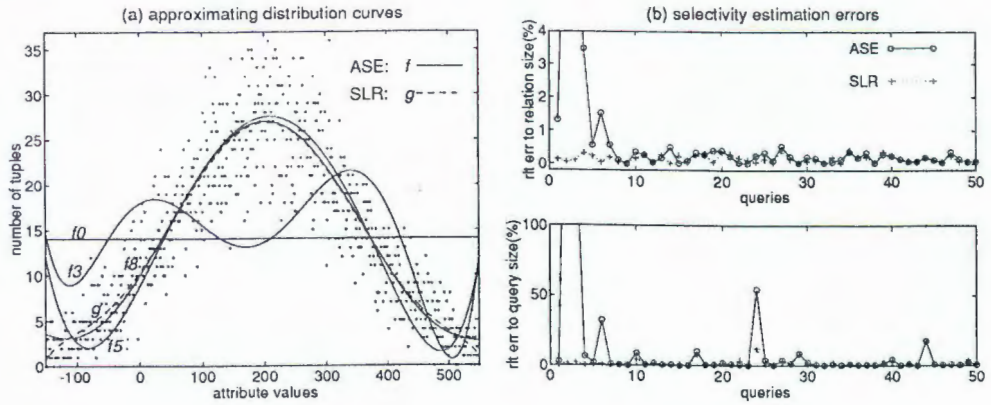

Figure 4.3: Normal Distribution

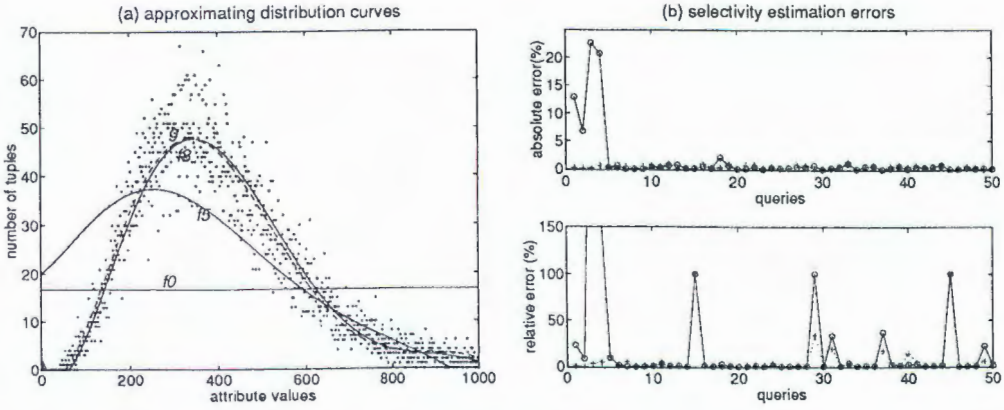

Figure 4.4: Chi-Square Distribution

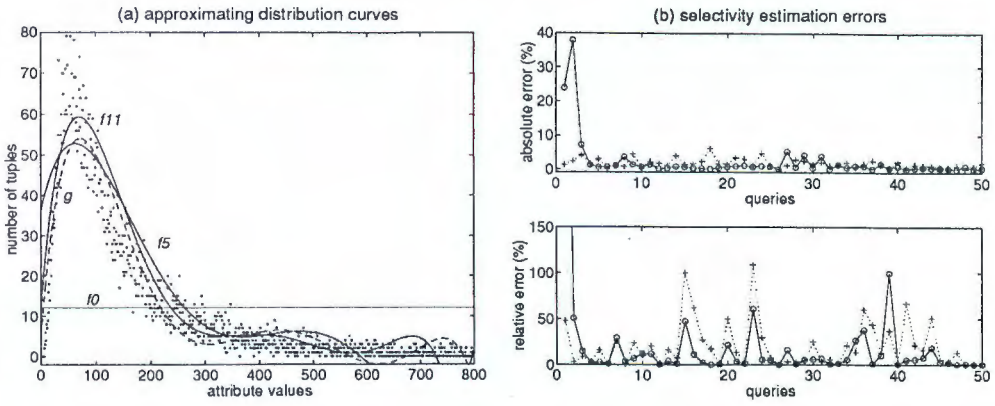

Figure 4.5: the F Distribution

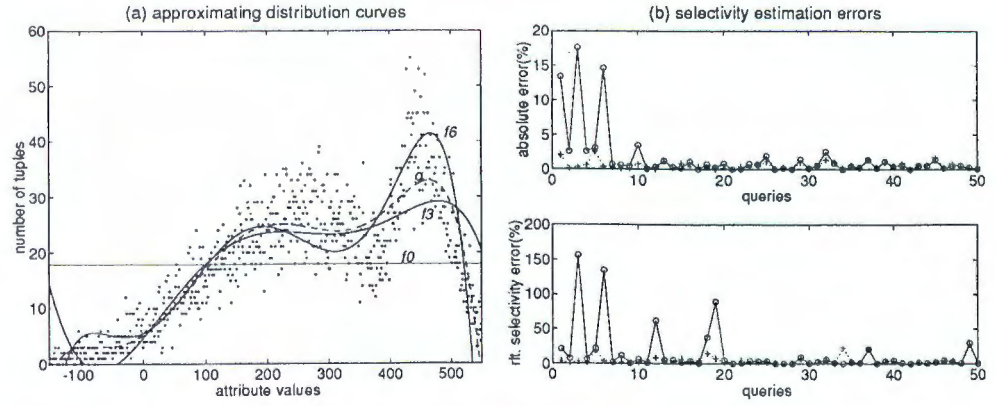

Figure 4.6: a bi-modal Distribution

4.4.2 Adaptiveness to Query Locality

No matter what method is used to estimate the data distribution, the computation *capacity* of the method is always limited (e.g., the number of intervals in a histogram, the degree of a polynomial). It is not uncommon for the distribution to be estimated to be too detailed to be modeled by the limited capacity. Therefore, we believe that instead of approximating the overall distribution evenly, the limited capacity should be used to approximate more accurately the local distribution of a rather narrow attribute sub-range which imposes either a temporal or spatial query locality. ASE inherits this merit: the more query feedbacks obtained from a local area, the more accurate the resulting approximating distribution for this area.

	Queried Range	Locality
LowQL	Jan. 1948 – Dec. 1978	Low
MedQL	Jan. 1948 – June 1960	Medial
HighQL	Jan. 1948 – Jan. 1953	High

Table 4.4: Three Levels of Query Localities

	ASE		SLR	
	nor. err.	rlt. err.	nor. err.	rlt. err
LowQL	1.1	5.6	0.93	5.0
MedQL	0.33	6.3	0.66	10.6
HighQL	0.086	12.8	0.14	21.3

Table 4.5: Estimate Errors of ASE and SLR under Various Query Localities

An "event" database which contains 431,258 records of events during 1948-1978 was used in this experiment. Three levels of query localities, as outlined in Table 4.4, were designed to compare ASE and SLR. For each level of locality, three random query streams (each of which contains 50 queries) were tested for both ASE and SLR. Table 4.5 summarizes the average errors for the 10th to 50th queries (we excluded the first 10 queries during which ASE has not yet converged). The curves of the approximating functions and the estimation errors of ASE and SLR are graphed for comparison, according to the three levels of localities, in Figures 4.7 , 4.8, and 4.9. It can be seen both from the tables and figures that ASE and SLR behave almost the same for low locality, but that as locality increases, ASE turns out to be better. This is because ASE is computed dynamically according to the query feedbacks and thus implicitly takes into account the query locality; in contrast, SLR is statically computed from the underlying data.

4.4.3 Adaptiveness to Updates

In this section, we show the elegant adaptation of ASE to updates. The normal distribution data from Section 4.4.1 is used again. Table 4.6 briefs the characteristics of

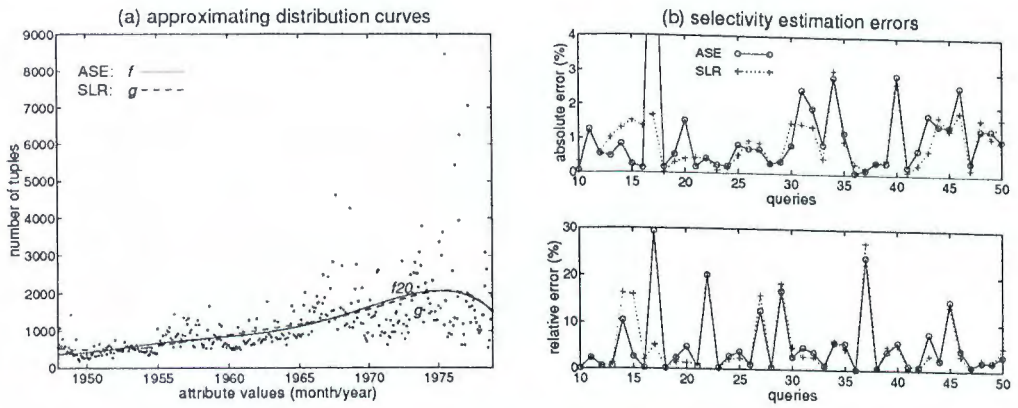

Figure 4.7: Adaptation under LowQL (Low Query Locality)

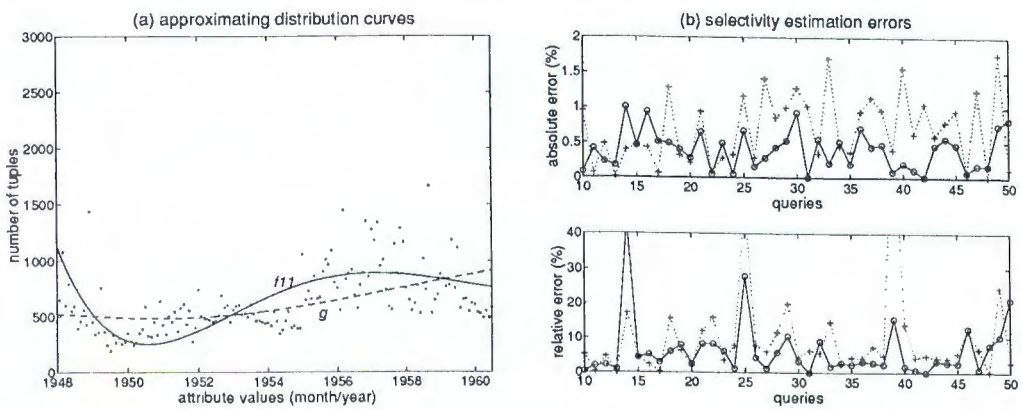

Figure 4.8: Adaptation under MedQL (Medium Query Locality)

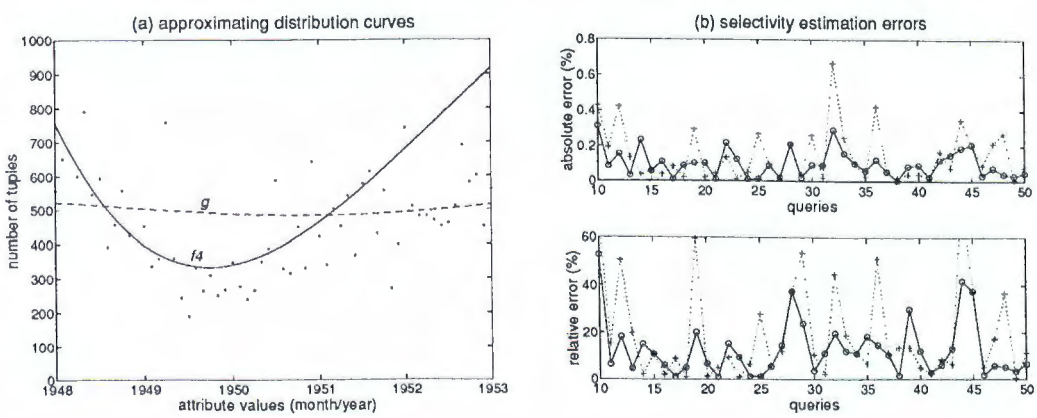

Figure 4.9: Adaptation under HighQL (High Query Locality)

	updates occur at	no. of total tuples updated	change of distribution shape	update transition
LOAD1	11	4,500	local, big increase	in batch
LOAD2	11, 17, 23, 29	9,000	global, slightly increase	gradual
LOAD3	11, 17, 23, 29	9,000	global, drastic	gradual

Table 4.6: Characteristics of Three Update Workloads

Update Workload	$ASE_{\alpha=0.01}$		$ASE_{\alpha=0.1}$		$ASE_{\alpha=0.5}$	
	nor. err.	rlt. err.	nor. err.	rlt. err.	nor. err.	rlt. err.
LOAD1	3.38	16.7	3.58	25.7	4.71	30.0
LOAD2	3.35	22.2	2.66	17.2	2.59	15.9
LOAD3	5.58	31.0	4.19	21.3	4.24	21.6

Table 4.7: Estimate Errors of Three Variations of ASE under Various Update Loads

three different update workloads to be interleaved with the query streams (more details about the update workloads are given in Appendix B). Orthogonal to the update loads are three versions of ASE, namely, $ASE_{0.01}$, $ASE_{0.1}$, and $ASE_{0.5}$, with different fading weights (as indicated in the subscripts). For each update workload, three query streams (each of which contains 40 selection queries interleaved with updates) are generated, and each of them is tested with all three fading weights. Table 4.7 tabulates the average errors; Figures 4.10, 4.11, and 4.12 correspond to the adaptation dynamics of ASE in the three different update loads. The corresponding curves for the three fading weights are grouped and graphed in each figure.

It can be seen from the figures that ASE adapts elegantly to all update loads. For example, in Figure 4.10.b, the errors go up over a few queries after the 10th query where update occurs, and then decline back to a stable low level. This adaptation can also be observed in Figure 4.10.a, where frames 2 through 4 show the adaptation of the approximating curves to the local distribution change at interval $[-50, 250]$. It is interesting to note from Table 4.7 that $ASE_{0.01}$, $ASE_{0.1}$, and $ASE_{0.5}$ are respectively the best in update loads LOAD1, LOAD3, and LOAD2. This is no surprise since

in LOAD1, a vast amount of update is done at once and thus it is advantageous to forget previous feedbacks and rely mainly on new ones. Therefore, the smallest fading weight $ASE_{0.01}$ (which forgets previous feedbacks to the greatest extent) outperforms the other two in this case. Similarly, in LOAD2, the shape of the distribution does not change too much during successive updates, and thus $ASE_{0.5}$ benefits the most by using old knowledge during transition. Finally, in LOAD3, where the distribution shape changes greatly through gradual updates, the use of $ASE_{0.1}$ offers a compromise between the two extremes.

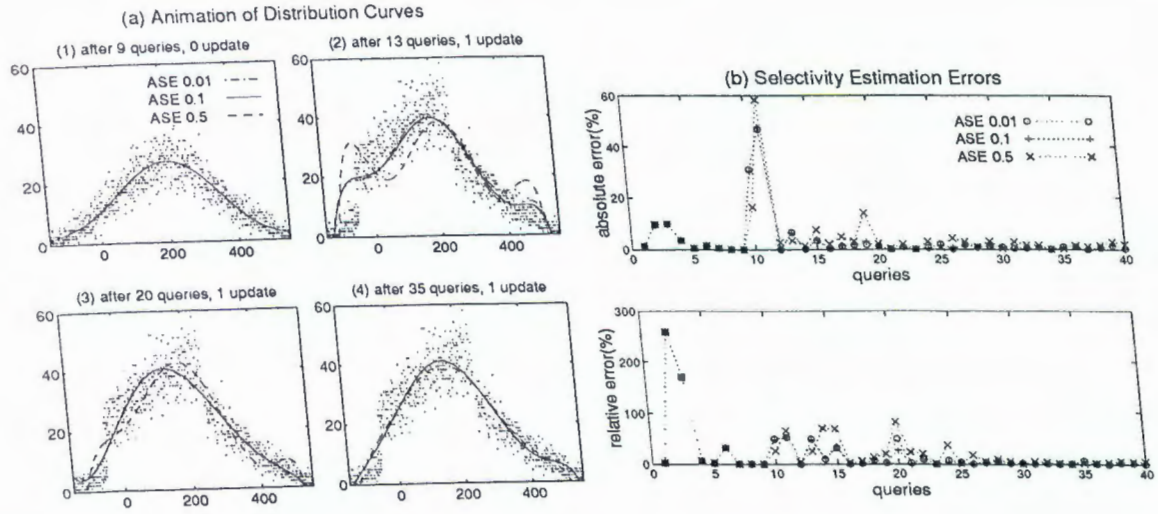

Figure 4.10: Adaptation under Update LOAD1

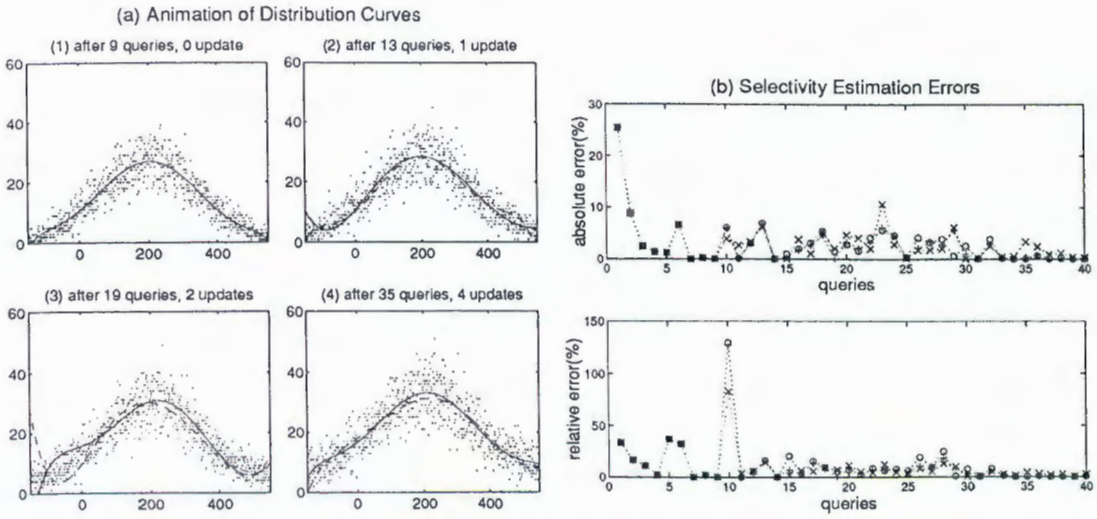

Figure 4.11: Adaptation in Update LOAD2

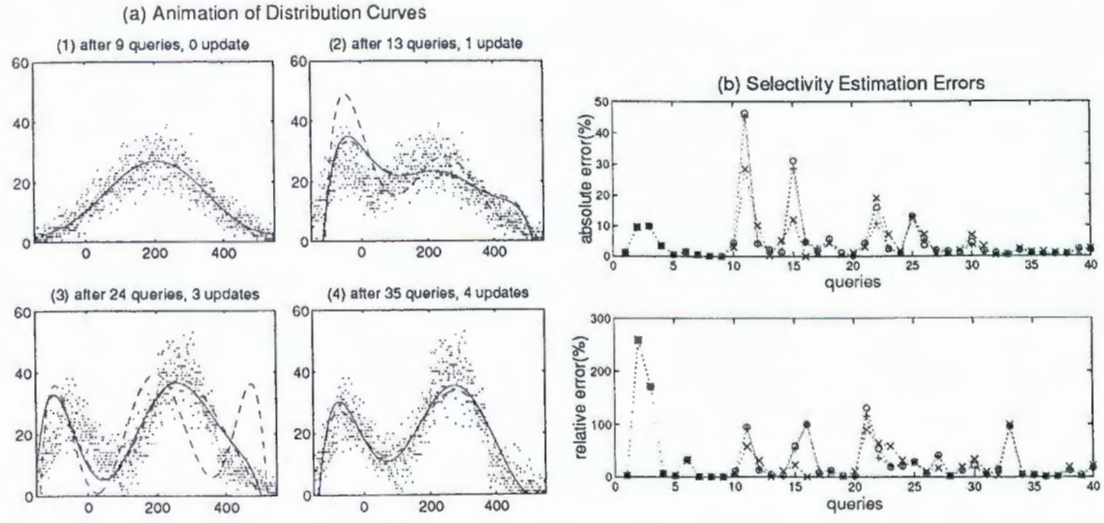

Figure 4.12: Adaptation in Update LOAD3

4.5 Conclusions

In this chapter, we have presented a novel approach for selectivity estimation. Capitalizing on the technique of recursive weighted least-square-error, we devised an adaptive selectivity estimator which uses actual query result sizes as feedback to approximate

the actual attribute distribution and to provide efficient and accurate estimations. The most significant advantage of this approach over traditional methods is that it incurs no extra cost for gathering database statistics. Furthermore, it adapts better to updates and query localities.

Throughout the experiments, ASE converges its approximation to the actual distribution of attribute values after around 10–15 queries. This is based on the uniform distribution of query ranges among the entire attribute range. In a load with skew query range distribution, the convergence might take longer. For example, if most of the queries have very large ranges, then the ASE might only give a coarse approximation to the actual distribution; if most of the queries are of point ranges, the ASE might take a long time to converge to the actual distribution. However, we think this is a feature rather than a weakness of ASE in the sense that ASE learns from and adapt to the actual query load.

Finally, We hope this study has inspired a new direction for data knowledge acquisition, especially in systems where statistics gathering is cost prohibitive because of large data sizes (such as tertiary databases).

Chapter 5

Buffer Allocation Using Query Feedback

5.1 Introduction

Buffer management has been the subject of much investigation in database management systems. The goal of such research has been to find suitable buffer management strategies for database systems and thus to enhance system performance. Early works [Rei76, SB76, TLF77, Kap80, EH84] adapted the conventional techniques used in virtual memory systems (such as LRU) to database management systems. Recently, another group of algorithms [CD85, SS86, NFS91] based on the prediction of *page reference patterns* was proposed. By taking into account the specific reference patterns exhibited by relational queries, methods from the second group are more suitable to database environments and perform better than the conventional strategies.

The methods based on pattern prediction have a major problem in that the proposed patterns are oversimplified for characterization of the page reference behavior of complicated queries. Moreover, it is usually difficult to predict a priori what the page references of a query will be like. For occasional queries, there is little opportunity for improvement because the exact reference behavior is not known in advance. However, for a large set of recurring queries such as compiled queries and pointer-materialized views [Rou82b, Val87, Rou91], whose page reference strings are more likely to re-

cur, we can use the information about their reference behavior learned from previous executions to adapt allocations for later executions.

In this chapter, we propose a buffer allocation scheme for recurring queries based on feedback of query page fault statistics. This method characterizes the page reference behavior of queries using page fault statistics obtained from prior query executions; it then refines the buffer allocation to achieve better buffer utilization and to improve the overall system throughput. This approach is practical because most database systems have a software-based buffer manager which can be extended to include the proposed feedback mechanism with minimal overhead.

The rest of this chapter is organized as follows: Section 5.2 reviews the related work. Section 5.3 describes a quantitative model for characterizing query page reference behavior and the feedback mechanism for obtaining those characteristics. Section 5.4 describes a buffer allocation scheme based on the feedback of those reference characteristics. Simulation results which show the advantages of this allocation scheme are given in Section 5.5. Section 5.6 concludes this work.

5.2 Related Work

In a database environment where concurrent queries arrive and compete for limited buffer resources, the buffer manager's goal is to reduce the disk operations and to enhance the system's throughput by utilizing a dedicated buffer pool for caching the data pages. To achieve this goal, three issues must be considered: (1) Load Control: When a query arrives, the buffer manager must decide if the query can be admitted for execution directly, or whether it should be blocked in a waiting queue until sufficient buffers are available. (2) Buffer Allocation: How should the buffers be allocated to queries or relations ? (3) Buffer Replacement: How should a victim buffer page be selected for replacement when the buffer pool is full and a new page is requested ?

algorithms	allocation/replacement policies for different patterns			admission policy
	sequential	looping	random	
	$[l_{min}, l_{max}]$, rpl	$[l_{min}, l_{max}]$, rpl	$[l_{min}, l_{max}]$, rpl	
DBMIN [CD85]	$[1, 1]$, –	$[t, t]$, MRU	$[1, 1]$, RAN,	$\sum l_{min} \leq A$
MG-x-y [NFS91]	$[1, 1]$, –	$[x\% * t, t]$, MRU	$[1, y]$, RAN	$\sum l_{min} \leq A$
EDU [FNS91]	$[1, 1]$, –	$[f(load), t]$, MRU	$[f(load), b_{yao}]$, RAN	$\sum l_{min} \leq A$

t: number of distinct pages referenced in a looping pattern

x: a constant between 1 and 100; y: a constant greater than or equal to 1

$f(load)$: a function which returns a number based on the current system load

b_{yao}: the expected number of distinct pages referenced in a random pattern

Table 5.1: Buffer Management Algorithms

In [Rei76, SB76, Tue76, Kap80, EH84], variations of traditional replacement techniques adapted from virtual memory systems such as LRU and Working-Set are applied to a global database buffer pool. These strategies, though successful in virtual memory mechanisms, do not perform satisfactorily in database systems because database applications have less page reference *locality* than that found in virtual memory activities [RR76, EH84, CD85].

Another group of algorithms [SS82, SS86, CD85, NFS91, FNS91] proposed the allocation of buffers based on specific reference patterns exhibited by relational queries. Some of these algorithms are summarized in Table 5.1, where all references are classified into three patterns. A *sequential* pattern is a sequence of distinct page references, a *looping* pattern denotes iterations of a sequential patter, and everything else is termed a *random* pattern. Each pattern is associated with an allowable buffer allocation range $[l_{min}, l_{max}]$, and a suggested replacement strategy *rpl*. For each relation instance r_i accessed in a query, the page reference pattern on r_i induced by this query is determined based on the analysis of the query plan, and at least l_{min} buffers are allocated to r_i.

A query is *admitted* (*activated*) for execution only if the current available buffers, A, is greater than $\sum l_{min}$ — the sum of the minimum buffer requirement of each relation accessed in the query. Otherwise, it must be *blocked* in a waiting queue until sufficient buffers are available. More than l_{min} buffers can be allocated to individual relation instances of an admitted query, but they should not exceed the upper bound l_{max} associated with the individual patterns. For all algorithms, MRU (Most-Recently-Used) replacement is adopted for looping patterns, RAN replacement—which selects a random page for replacement—is used for random patterns, and no explicit replacement strategy is needed for sequential patterns since only one buffer page is allocated.

Basically, these algorithms differ only in the allocation ranges $[l_{min}, l_{max}]$ for looping and random patterns. DBMIN allocates a *fix* number of buffers to each pattern (reflected by $l_{min} = l_{max}$). While DBMIN was shown to outperform the traditional algorithms, its strict allocation policy might result in poor buffer utilization. For example, a query with a looping pattern of $t = 100$ (100 distinct pages referenced) will not be admitted to execution even if there are 90 buffer pages available. MG-x-y is more flexible in the sense that it reduces the minimum buffer requirement for looping patterns to $x\% * t$, and allocates up to y buffers to a random pattern as long as the *expected marginal gain*[1] is still positive. While MG-x-y has improved DBMIN, keeping x and y as global constants for all queries is not appropriate because different reference strings, even though of the same pattern, can have completely different faulting behavior. EDU is one of the class of *predictive* load control algorithms proposed in [FNS91]. Subject to the current buffer availability, a query is activated only if activation will result in better *expected* system performance. $l_{min} = f(load)$ is computed as the minimum number of buffers required to activate a query so as to enhance the current system performance, based on the current system load. For random patterns,

[1]The expected marginal gain is the expected number of page faults reduced per extra buffer allocated.

$l_{max} = b_{yao}$ is the expected number of distinct pages referenced based on Yao's formula [Yao77]. This approach was shown to be more adaptive to different query loads than MG-x-y. The computation of b_{yao} and the expected system performance, however, are based on the assumption of *uniform page accesses*, which in general is not true[2].

Obviously, the main weakness of the above algorithms lies in their inability to characterize *random* reference strings more finely. In these schemes, all reference strings, other than sequential and looping ones, are categorized as *random* and are treated equally based on the assumption of uniform page accesses. This overlooks the benefit of allocating more buffers to those relations which reduce page faults most and of allocating fewer buffers to the others. For example, in a multi-relation join where non-clustered indices or hash tables are used, the reference strings on the indexed or hashed relations will be classified simply as random, while the actual page references may indeed turn out to be of a certain locality instead of uniformity.

To cope with the above problem, we propose a feedback mechanism to capture the page reference behavior of queries by collecting simple statistics during executions. This model associates each recurring reference string with a characteristic record, which is used to adjust the buffer allocation for later executions. A simple load controller is also adopted; however, the scheme we propose here is basically an allocation-oriented approach rather than a load-control-oriented one. LRU is used under all circumstances unless a looping pattern is detected, in which case MRU is used instead.

[2]The assumption of *uniform page accesses* assumes that a sequence of page references to a relation are distributed uniformly among all pages of this relation. Computed based on this assumption, b_{yao} is in general much higher than the actual number of pages referenced.

5.3 The Feedback Mechanism

In this section, we propose a quantitative model for characterizing query page faulting behavior and describe a mechanism for collecting the characteristics during query executions.

5.3.1 The Faulting Characteristic Model

Definition 6 A *reference string* $\mathcal{R} = \{r_1, r_2, \ldots\}$ is a finite sequence of page references, where r_i denotes the page number of the referenced page. We use $|\mathcal{R}|$ to denote the *normalized length* of $\mathcal{R}$ where consecutive references to the same page are counted as one reference, and let $C(\mathcal{R})$ be the *number of distinct pages* referenced in $\mathcal{R}$. □

For example, suppose $\mathcal{R} = \{3, 2, 2, 1, 8, 1\}$, then $|\mathcal{R}| = 5$ because page 2 is referenced twice in a row and should be counted as only one reference. In this case, $C(\mathcal{R}) = 4$.

Definition 7 Suppose b buffer pages are dedicated to reference string $\mathcal{R}$ and are manipulated under a buffer management strategy B. We use $f_{\mathcal{R},B}(b)$ to denote the number of page faults as a function of b. We call $f_{\mathcal{R},B}$, or simply f when $\mathcal{R}$ and B are understood, the *faulting function* of $\mathcal{R}$ under B. □

From the above definitions, it is not hard to see that for $1 \leq b \leq C(R)$,

$$C(\mathcal{R}) = f(C(\mathcal{R})) \leq f_{\mathcal{R},B}(b) \leq f(1) = |\mathcal{R}|.$$

This formula simply says that no matter what kind of buffer management strategy is used for a reference string $\mathcal{R}$, at least $C(\mathcal{R})$ disk reads must be performed to access all the distinct pages, and that at most $|\mathcal{R}|$ page faults can occur (which is the case when only one buffer page is allocated). It will be useful to find and keep the faulting function for each reference string if possible. Since there is no easy way to express

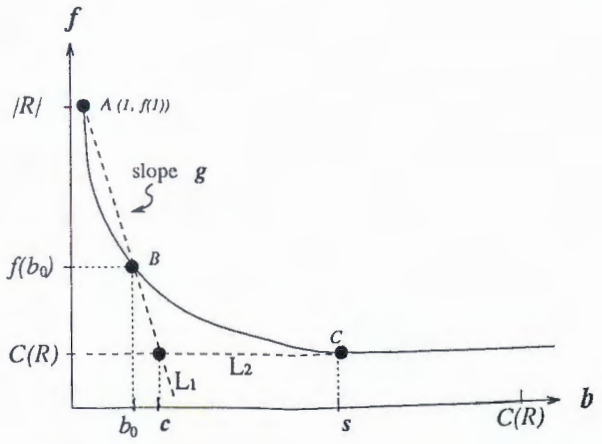

Figure 5.1: Faulting Characteristic Model

a faulting function in terms of a compact mathematical formula, however, some kind of approximation must be sought. In the following, we introduce a simple model to characterize the faulting function.

Definition 8 (Faulting Characteristic Model (FCM)) The *faulting characteristic* of a faulting function $f_{\mathcal{R},\mathrm{B}}$ *at buffer size* b_0 is a triple $\rho_{b_0} = (g, c, s)$, where

$$\begin{aligned} g &= (f_{\mathcal{R},\mathrm{B}}(1) - f_{\mathcal{R},\mathrm{B}}(b_0))/(b_0 - 1), \\ c &= 1 + (b_0 - 1)\frac{f_{\mathcal{R},\mathrm{B}}(1) - C(\mathcal{R})}{f_{\mathcal{R},\mathrm{B}}(1) - f_{\mathcal{R},\mathrm{B}}(b_0)}, \\ s &= \textit{the minimum } b \textit{ such that for all } b' > b, f_{\mathcal{R},\mathrm{B}}(b') = f_{\mathcal{R},\mathrm{B}}(b) \end{aligned}$$

We call g the *average marginal gain*, c the *critical size*, and s the *saturated size*. □

Intuitively, ρ_{b_0} characterizes the general behavior of $f_{\mathcal{R},\mathrm{B}}$ between the buffer range from 1 to b_0. This is depicted in Figure 5.1, which shows the faulting characteristics of a typical faulting function at buffer size b_0. The faulting curve is approximated by three points A, B, and C, which can easily be computed during the course of running the reference string (we will discuss how these values can be obtained later). The average marginal gain g is the slope of line L_1, which connects points A and B, and

thus represents the average number of page faults reduced per extra buffer allocated in the range of 1 to b_0 buffers. The saturated size s is the smallest buffer size beyond which the faulting curve becomes horizontal. That is, adding buffers beyond s will not lead to any additional reduction of page faults. Since s depends only on $\mathcal{R}$ and B, we use $s_{\mathcal{R},\mathtt{B}}$ for clarity when needed. It is easy to see that $f(s) = f(C(\mathcal{R})) = C(\mathcal{R})$. In practice, s is usually smaller than $C(\mathcal{R})$[3]. Critical size c is the x-axis value of the intersection point of line L_1 and L_2, where L_2 is the horizontal line $f = C(\mathcal{R})$. The intention here is to treat the critical size as the expected buffer size around which the rate of page fault reduction slows considerably.

Before going into the details of how to use faulting characteristics to adjust allocations, we shall first describe how to compute the characteristics of a query execution and how to obtain the most informative characteristics.

5.3.2 Collecting Faulting Characteristics

According to the definition of FCM, there are four basic numbers to be computed for each reference string during its course of execution: $f_{\mathcal{R},\mathtt{B}}(b_0)$, $|\mathcal{R}|$, $C(\mathcal{R})$, and $s_{\mathcal{R},\mathtt{B}}$. We describe how each of them can be obtained during the execution.

number of page faults $f_{\mathcal{R},\mathtt{B}}(b_0)$:

We need a counter to keep the number of page faults. The counter will increase by one each time a page fault occurs.

normalized length $|\mathcal{R}|$:

To obtain the normalized length, we must detect the consecutive references to the same page. This can be achieved by keeping a counter for $|\mathcal{R}|$ and increasing it by one only when the current page reference is different from the previous one (we can use a variable to keep track of the previous page reference).

[3] In our experiments, if LRU is used and $\mathcal{R}$ is a random pattern, then $s_{\mathcal{R},\mathtt{LRU}}$ ranges from $C(\mathcal{R})/3$ to $C(\mathcal{R})$

number of distinct pages referenced $C(\mathcal{R})$:

A data structure must be maintained to record all the page numbers that have been accessed so far. Initially this structure will be empty. As a page fault occurs, the structure will be searched to see if the faulted page is a new one; if it is not in the structure, the counter for $C(\mathcal{R})$ will increase by one and the page number of this new faulted page will be entered into the structure. This can be maintained efficiently using a bitmap structure where each bit corresponds to a page and a zero bit is set to one when the corresponding page is first faulted.

saturated buffer size $s_{\mathcal{R},B}$:

Saturated size depends both on the reference string and the underlying buffering strategy. Its exact value is not easy to calculate in general. Iteratively tracing the whole reference string to find the number of page faults at all buffer sizes and thus to obtain the saturated size is impractical. If the underlying strategy is LRU, however the saturated size can be found efficiently. Actually, LRU is a member of a class of replacement algorithms called *stack algorithms* whose faulting function for any reference string can be found during one pass trace of that string [M+70]. According to that paper, if we simulate $\mathcal{R}$ under LRU replacement strategy using a LRU-stack of $C(\mathcal{R})$ pages, then

$$f_{\mathcal{R},\mathrm{LRU}}(b) = C(\mathcal{R}) + \sum_{i=b+1}^{C(\mathcal{R})} hit(i),$$

where $hit(i)$ is the frequency of page hits on the i'th page to the *top* (the most recently used or the *growing* end) of the LRU-stack. Since the saturated size is the minimum s such that $f_{\mathcal{R},\mathrm{LRU}}(s) = C(\mathcal{R})$ (by Definition 8), according to the above equation we have $hit(i) = 0$ for all $(s+1) \leq i \leq C(\mathcal{R})$. Therefore, the saturated size can be computed as

$$s_{\mathcal{R},\mathrm{LRU}} = \max\{i | hit(i) \neq 0, 1 \leq i \leq C(\mathcal{R})\}.$$

Actually, this formula is valid as long as the simulated stack size is greater than or

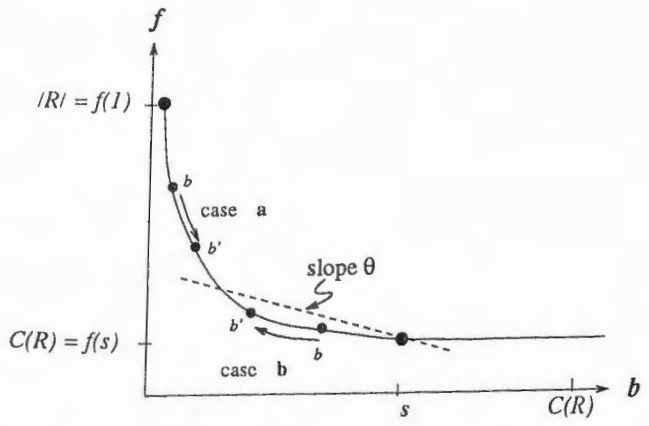

Figure 5.2: Adapting Faulting Characteristics

equal to the saturated size (instead of $C(\mathcal{R})$). If the stack size is less than the saturated size or the strategy is not LRU, we can simply set $s_{\mathcal{R},B} = C(\mathcal{R})$ as a loose upper bound.

5.3.3 Adapting Faulting Characteristics

The values of the faulting characteristics ρ_b depend heavily on b—the number of buffers allocated. If b is too small (close to 1) or too large (close to the saturated size), the obtained characteristics, especially the average marginal gain, may not reflect the overall faulting function well. Therefore, we need to find "better informed" characteristics which will reflect the average faulting behavior more accurately. Formally, let ρ_b be the best characteristics so far associated with a reference string under consideration. Now suppose b' is the buffer size allocated for the next execution and $\rho_{b'}$ is composed of the newly obtained characteristics, then $\rho_{b'}$ replaces ρ_b only if

a) $b' > b$ and $r_{b'} > \theta$, or

b) $b' < b$ and $r_{b'} < \theta$,

where $r_{b'} = \frac{f(b')-C(\mathcal{R})}{s-b'}$ is the slope between points $(b', f(b'))$ and $(s, C(\mathcal{R}))$. We call $r_{b'}$ the *residual gain* at b'. θ is a global threshold set for all reference strings. Intuitively, $\rho_{b'}$ is a more informed feedback than ρ_b if b' is closer to the buffer size beyond which the

residual gain becomes smaller than the pre-defined threshold θ. These two conditions assure that the adaptation, initiated by either an under-allocation (case **a**) or an over-allocation (case **b**), eventually converges to a well-informed characteristic record whose residual gain is close to θ. This is shown in Figure 5.2

5.4 Buffer Allocation Based on Average Marginal Gain Ratio

Since the purpose of this work is to demonstrate the strength of adaptive buffer allocation based on feedback information, a simple load control mechanism is used. Let c_q be the critical size of query q; this size is computed from the combined reference string of q. Let A be the current available buffers, then query q is activated only if

$$A \geq min(w_1 * c_q, w_2 * \texttt{MAX\_BUF}),$$

where `MAX_BUF` is the size of the buffer pool. We set $w_1 = 0.5$ and $w_2 = 0.9$ so that a query will be admitted only if either half of the critical size or, in the case where the critical size is much bigger than the total buffers, 90% of the total buffers must be available. Once admitted, query q is allocated with $min(w_3 * c_q, A)$ buffers, where we set $w_3 = 0.8$. This allows the admitted query to acquire more than half of but not greater than 0.8 of the critical size of buffers.

At the same time as a query is admitted, the buffer manager must determine how many buffers to allocate to each relation instance of the query. DBMIN and MG-x-y are examples of such algorithms whose allocation strategies for a random reference pattern are based on the assumption of uniform page access distribution. As an alternative, we introduce an allocation algorithm based on the faulting characteristics defined in FCM. Since FCM quantifies the reference behavior from feedback, it provides more accurate information than that based on uniformity.

Allocation based on Marginal Gain Ratio (MGR) For each query, we allocate buffers to individual relations *in proportion to* their average marginal gains subject to the following constraints:

a) never allocate more than the saturated size (avoid waste), and

b) when the demand for buffers is high, never exceed the critical size of each reference string.

We demonstrate the MGR allocation policy, which has been implemented in ADMS, with a complete example and show the adaptation process of the feedback mechanism FCM. A join query on three base relations is taken as the example:

```
SELECT *
FROM 10k1, 10k2, 10k3
WHERE10k1.un1 = 10k2.un1 and 10k2.un2 < '500'
      AND 10k1.5000 = 10k3.5000
```

Each relation contains 10,000 tuples spanning 2,500 pages, and the query result has 1,000 tuples. The query is chosen such that none of the reference strings on any of the relations is of sequential or looping pattern. Table 5.2 tabulates the allocations and characteristics for a sequence of executions of this query performed on ADMS, using MGR and FCM.

Each table denotes an execution, where A is the number of buffers available for this very execution instance. Column b corresponds to the buffers allocated to each relation; Column $f(b)$ shows the resulting page faults; Column r denotes the residual gain computed based on b and $f(b)$ after the execution. After each execution the faulting characteristics ρ_b are computed (but are not shown in the table), and the *best so far informed* faulting characteristic record $\rho_{b^*} = (g, c, s)$ is revised and stored in columns b^*, g, c, and s. Note that b^* is used in the future execution to determine if a

Execution 1, $A = 50$									
	b	$f(b)$	r	b^*	g	c	s	$\lvert\mathcal{R}\rvert$	$C(\mathcal{R})$
10k1	17	951	1.39	17	2.75	196	372	995	458
10k2	17	879	6.98	17	7.06	123	125	992	125
10k3	16	897	0.06	16	0.00	1	719	897	853
total		2727							

Execution 2, $A = 100$							
	b	$f(b)$	r	b^*	g	c	s
10k1	28	924	1.35	28*	2.62*	206*	372
10k2	71	455	6.11	71*	7.67*	114*	125
10k3	1	897	0.06	16	0.00	1	719
total		2276					

Execution 3, $A = 300$							
	b	$f(b)$	r	b^*	g	c	s
10k1	174	663	1.03	174*	1.91*	282*	372
10k2	125	125	0.00	71	7.67	114	125
10k3	1	897	0.06	16	0.00	1	719
total		1685					

Execution 4, $A = 50$							
	b	$f(b)$	r	b^*	g	c	s
10k1	9	979	1.43	174	1.91	282	372
10k2	40	701	6.77	71	7.67	114	125
10k3	1	897	0.06	16	0.00	1	719
total		2577					

Execution 5, $A = 100$							
	b	$f(b)$	r	b^*	g	c	s
10k1	19	944	1.37	174	1.91	282	372
10k2	80	402	6.15	80*	7.47*	117*	125
10k3	1	897	0.06	16	0.00	1	719
total		2243					

Table 5.2: Adaptation of MGR Allocation Scheme

new feedback should replace the current one (see Section 5.3.3). New adaptations of characteristics are marked with asterisks. The threshold θ for the residual gain is set to 1.0 in this experiment.

At the first execution, since no feedback information is yet available, MGR simply allocates the 50 available buffers evenly among all relations. After the execution, $f(b), |\mathcal{R}|, C(\mathcal{R})$, and s are obtained, and based on these, r, g, and c are computed. Since it is the first execution, all the above numbers are saved. Note that $|\mathcal{R}|, C(\mathcal{R})$, and s are kept unchanged thereafter.

At the second execution, where 100 buffers are available, MGR allocates the buffers among the relations in proportion to the marginal gains obtained from the previous execution. As a result of this, 10k1 is allocated $100*2.75/(2.75+7.06+0) = 28$ buffers, 10k2 is allocated $100 * 7.06/(2.75 + 7.06 + 0) = 71$ buffers, and 10k3 is allocated one buffer since its marginal gain is 0. After this execution, the residual gains for 10k1 and 10k2 are computed to be 1.35 and 6.11, both of which are still greater than the threshold value 1.0. Since the buffers allocated to 10k1 and 10k2 (column b in the second table) are greater than those recorded in column b^* in the previous table, the new characteristics should replace the old feedback (according to Section 5.3.3). Therefore, the characteristic record of 10k1 gets adjusted from $\rho_{17} = (2.75, 196, 372)$ of Execution 1 to a more informed one $\rho_{28} = (2.62, 206, 372)$ of Execution 2. The same adaptation takes place for 10k2. However, for 10k3, since the faulting characteristics at buffer size 1 are meaningless, the characteristics obtained from Execution 1 $\rho_{16} = (0.00, 1, 719)$ will remain unchanged.

At the third execution, 300 buffers are available, according to the marginal gain ratios obtained from Execution 2; 10k1 is first allocated $300 * 2.62/(2.62 + 7.67) = 76$ buffers and 10k2 is allocated $300 * 7.67/(2.62 + 7.67) = 223$ buffers. According to the MGR scheme, however, no allocation can exceed the saturated size; thus only 125 buffers will be allocated to 10k2. The remaining $223 - 125 = 98$ buffers are

then redistributed among 10k1 and 10k3, but since 10k3 has zero marginal gain and a critical size of 1, 10k1 receives all the remaining buffers and has a total of 76+98 = 174 buffers. After the execution, the residual gain of 10k1 is 1.03, which is greater than the threshold, so the characteristic record is again adjusted from $\rho_{28} = (2.62, 206, 372)$ to $\rho_{174} = (1.91, 282, 372)$. The characteristic record of 10k2 is not changed since its residual gain $r = 0$ is less than the threshold.

At the fourth execution, where 50 buffers are available, MGR allocates buffers based on the characteristics resulting from the prior three executions. As a result, it produces 2577 page faults, which is fewer than the 2727 page faults produced by Execution 1 where the same buffer size is provided but no feedback is available. No characteristics are adapted after this execution since neither condition **a** nor condition **b** of the adaptation guideline given in Section 5.3.3 is satisfied. For example, for 10k2, the allocated buffer size $b = 40$ at Execution 4 is less than $b^* = 71$ at Execution 3, but the residual gain $r_b = (701 - 125)/(125 - 40) = 6.77$ is greater than the threshold θ of value 1.

Finally, at the fifth execution, where 100 buffers are provided, 2243 page faults are produced. This is slightly smaller than 2276 — the number of page faults produced during Execution 2, which used the same number of buffers. This can be attributed to the adaptation of the characteristic record through Executions 2 and 3. Also note that the characteristics of 10k2 are adjusted again after this execution because the allocated buffer size $b = 80$ is greater than $b^* = 71$ (as recorded in Execution 4), and the residual gain $r_b = 6.15$ is also greater than the threshold $\theta = 1$.

We also experimented using the MG-x-y algorithm with the same sequence of executions above. Since the example query induces random reference strings of equal length on all three relations, MG-x-y simply results in allocating buffers equally among the relations. We compare the page faults of MGR with those of MG-x-y in Table 5.3. It can be seen that MGR has fewer page faults than MG-x-y as executions recur.

	Exec. 1	Exec. 2	Exec. 3	Exec. 4	Exec. 5
available buffer size	50	100	300	50	100
page faults of MGR	2727	2276	1685	2577	2243
page faults of MG-x-y	2727	2572	1951	2727	2572

Table 5.3: Page Faults of `MGR` and `MG-x-y`

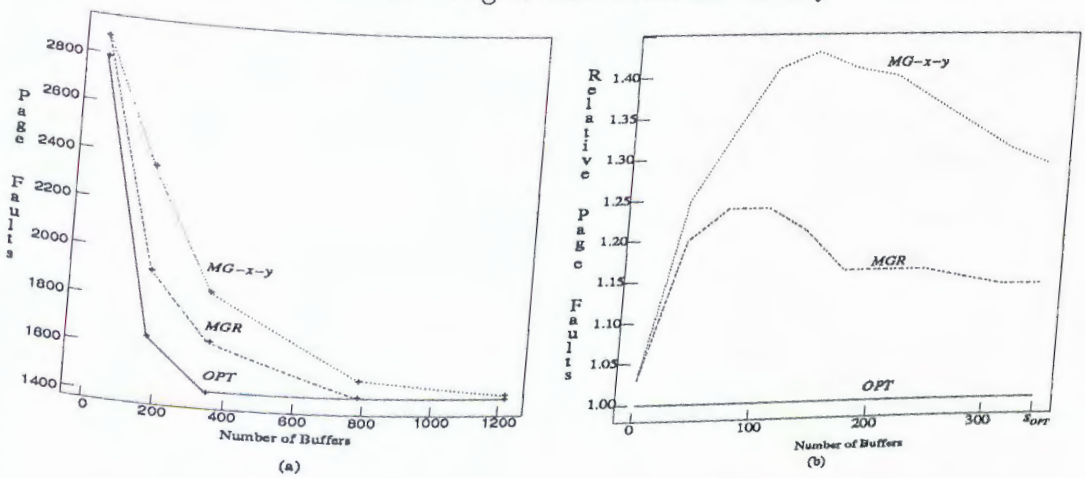

Figure 5.3: Page Fault Comparison among `MGR`, `MG-x-y` and `OPT`

Using the same example query above, we compare the page faults of MGR and MG-x-y at different numbers of buffers in Figure 5.3. For MGR, we used the characteristics obtained after Execution 5 from Table 5.2. The optimal replacement algorithm OPT [Bel66], which replaces the page that will not be used for the longest time in the future, is also graphed for comparison. The relative performance of MGR and MG-x-y to OPT (number of page faults of MGR and MG-x-y divided by that of OPT) is drawn in Figure 5.3.(b), with number of buffers ranging from 1 to s_{OPT}, where s_{OPT} is the saturated size under OPT replacement strategy. It can be seen that MGR performs much better than MG-x-y. Especially when the number of buffers is around 200, the page faults of MGR are reduced from those of MG-x-y to half of the page faults above OPT.

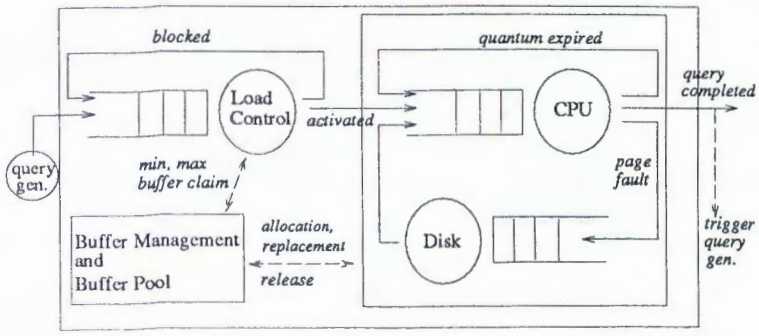

Figure 5.4: Simulation Configuration

Parameters	Values
disk transfer time	10–20 ms (uniform)
page size	1024 bytes
CPU quantum for round robin	10ms
interval between two consecutive logical page accesses	0.5–1.5 ms (uniform)
average FCM overhead per page reference	0.018 ms

Table 5.4: System Parameters

5.5 Performance Evaluation

5.5.1 System Configuration and Buffer Management Algorithms

In this section we present the results from a simulation study which evaluated the performance of different database buffer management algorithms including MGR, MG-x-y, and EDU. The simulation is similar to the one used in [CD85, NFS91], which simulated a closed system with concurrent queries competing for buffers. Figure 5.4 shows the system configuration of the simulation. The number of concurrent queries (concurrency level) varied from 1 to 32. When the system starts, successive queries are generated and enter the system until the concurrency level is reached. As the system continues to run, no new query can enter the system until one of the running queries finishes and leaves the system. Concurrent queries are scheduled for the CPU using a round robin policy. Unless otherwise mentioned, the size of the buffer pool is set to

query type	no. of result tuples	no. of base relations	reference type
Q1	1000	1	*Sequential*
Q2	1000	2	*Sequential, Looping*
Q3	1000	2	*Random*
Q4	1000	3	*Random*

Table 5.5: Four Query Types

query mix	Q1	Q2	Q3	Q4
M1	40%	40%	10%	10%
M2	10%	10%	40%	40%
M3	25%	25%	25%	25%

Table 5.6: Three Query Mixes for Simulation Workload

1,000 pages. In all cases, the query mixes along with the configurations simulate an IO-bound closed system. Other system parameters are shown in Table 5.4.

We used four basic query types to mix the query streams. The reference strings of the query types are collected from executing them against the Wisconsin Benchmark database [BDT83] on ADMS. The number of participating relations in each query varies from one to three, and each relation contains 10,000 tuples spanning 2,500 pages. Table 5.5 shows the characteristics of the page references for each of the query types we chose. The query types have been chosen such that each of them accesses various numbers of distinct pages ranging from 100 to 1,500. Q1 accesses 250 different pages, Q2 accesses 10 pages of the outer relation and has a looping access on a set of 98 pages of the inner relation. Q3 and Q4 perform random accesses to their relations, with totals of 1110 and 1416 distinct pages referenced respectively.

Three different query mixes are generated according to different percentages of the four query types. They are shown in Table 5.6. Query mix M1 represents the situation where most of the reference strings are either sequential or looping, M2 simulates the

situation where random references dominate, and in M3, all query types have the same frequency.

One of the factors that affects buffer utilization is the degree of data sharing [BDT83, CD85, NFS91, FNS91]. We adopted the same methodology from [BD84, CD85, NFS91, FNS91] for classifying the degree of data sharing in our simulation. Three degrees of data sharing are tested. In *no data sharing*, all concurrent queries access different copies of the relations or completely different databases. In *partial data sharing*, every two of the concurrent queries access the same copy of database. In *full data sharing*, all queries access the same database. The higher the degree of sharing, the better buffer utilization due to the fact that pages in the buffers are used by several concurrent queries.

For the purpose of establishing a baseline comparison, LRU is selected as the representative for traditional strategies. According to [EH84, CD85], all the traditional strategies including LRU, Working-Set, and Clock make no significant performance difference when applied to database applications. Two schemes, *local LRU* (LLRU) and *global LRU* (GLRU) are tested. Local LRU keeps an LRU list for every relation instance of every running query. Global LRU manipulates the entire buffer pool using a single LRU list for all queries. There is no explicit load control for global LRU; a query is admitted immediately as it arrives. For local LRU, however, a query is admitted only when there are available buffers.

Since DBMIN has been outperformed by MG-x-y using a flexible allocation [NFS91], we do not include DBMIN in the comparison. By trial and error, we have adjusted the values of parameters x and y of MG-x-y so that it reached its best overall performance. Six pairs of (x, y) are experimented: (50, 100), (50, 200), (50, 400), (100, 100), (100, 200), and (100, 400). A query is admitted if the number of available buffers is greater than the sum of each of its accessed relation's minimum buffer requirements (see Table 5.1). After a query is admitted, MG-x-y allocates as many buffers as possible to

each relation but never exceeds a specified upper bound (see Table 5.1).

Among the class of predictive load control algorithms proposed in [FNS91], the algorithm with the best overall performance, called EDU, is chosen as a representative in our simulation. Subject to the current number of available buffers, EDU activates a query only if doing so will result in better *effective disk utilization* than that of the current state. After the query is admitted, it allocates buffers in the same way as does MG-x-y.

As described earlier, MGR uses a simple load controller based on FCM. MGR uses the marginal gain ratios for buffer allocation for both queries and relations. It uses LRU replacement unless a looping pattern is detected, in which case MRU is used instead. The overhead of collecting and computing characteristics is also estimated and included in the simulation, though it is insignificant when compared to the whole query evaluation time (according to our experiment from the implementation of MGR on ADMS). It should be pointed out that for the first time a query runs, MGR simply uses MG-x-y strategy for allocation since no feedback is yet available. After the first query feedback, however, MGR uses the faulting characteristics to adjust the allocations for recurring queries. Because MGR uses more information about the behavior of the queries, it is expected to do better than all other techniques.

In the rest of this section, we present the results of the simulation. In all figures, the *throughput* refers to the average number of queries finished per minute.

5.5.2 Simulation Results

Effect of Sequential and Looping References

Figure 5.5 shows the results under query mix M1. The throughput of each scheme increases until the number of concurrent queries reaches four, after which buffer contention occurs, and thus the throughput starts to decline. Due to the load control, however, the throughput reaches a saturated level around twelve concurrent queries.

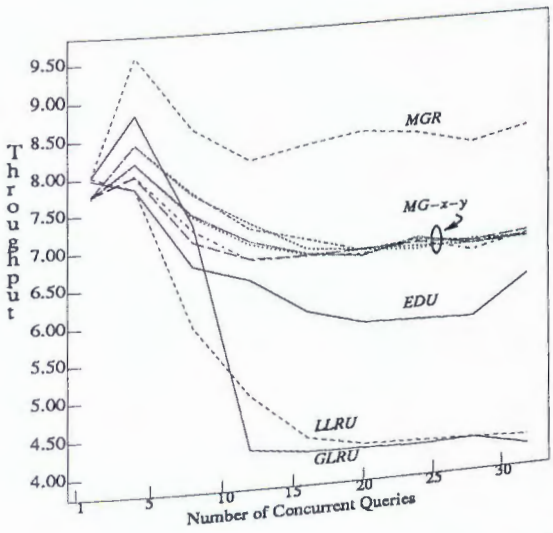

Figure 5.5: Throughput Comparison under Query Mix M1, No Data Sharing

MGR performs better than all the others because it makes more effective allocation for random references than do the others. Since sequential and looping references dominate in this load, MG-x-y, which makes allocation for both based on their *exact* patterns, outperforms the load control-based scheme EDU which makes admission decisions based on an *estimated* performance pointer. This implies that load control does not have as much impact as does buffer allocation on the performance under sequential-and-looping-dominated query mixes.

Effect of Random References

Figure 5.6 compares the throughput under query mix M2, where the random references dominate. As opposed to Figure 5.5, where the throughput increases for a small range of current queries, the throughput in this load declines as the number of concurrent queries increases since random references usually consume more buffers than do sequential and looping references. MGR still provides substantial performance improvement because of its ability to characterize random access behavior correctly. In this case, the improvement of MG-x-y over LRUs is less substantial when compared to the previous figure. This decline in improvement is attributable to MG-x-y's inability to characterize random references. EDU now outperforms MG-x-y because the

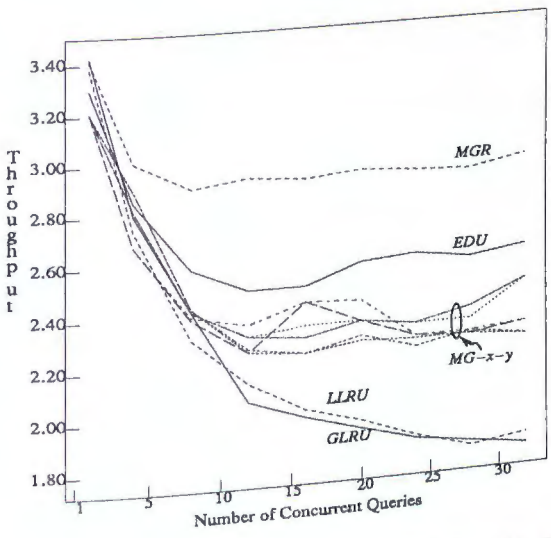

Figure 5.6: Throughput Comparison under Query Mix M2, No Data Sharing

load controller of EDU is able to block temporarily certain random reference queries to avoid performance degradation due to a lack of available buffers, while the static upper bound y, imposed by MG-x-y on all random references, is less flexible.

Equal Frequency Query Mix

Figure 5.7 depicts the results of running query mix M3, where sequential, looping, and random references occur with equal frequency. Because of its ability to characterize more finely on random references, MGR again performs much better than all other strategies. However, the gap between MGR and the group of EDU and MG-x-y is larger than the gap in Figure 5.6. This is because, based on the assumption of uniform page accesses, MG-x-y and EDU tend to overallocate for random references, whereas based on characteristics feedback, MGR makes more effective use of buffers for random references. MGR is thus able to admit more concurrent queries of sequential and looping references, which usually consume fewer buffers than random references, and throughput is enhanced. The group of MG-x-y and EDU algorithms have similar performance, which is significantly better than those of LLRU and GLRU.

Effect of Data Sharing

We used query mix M3 again in this set of experiments. Figure 5.8 shows the

results of partial data sharing. EDU now again performs fairly better than does MG-x-y after the concurrency level reaches 12. MGR remains much better than all the others.

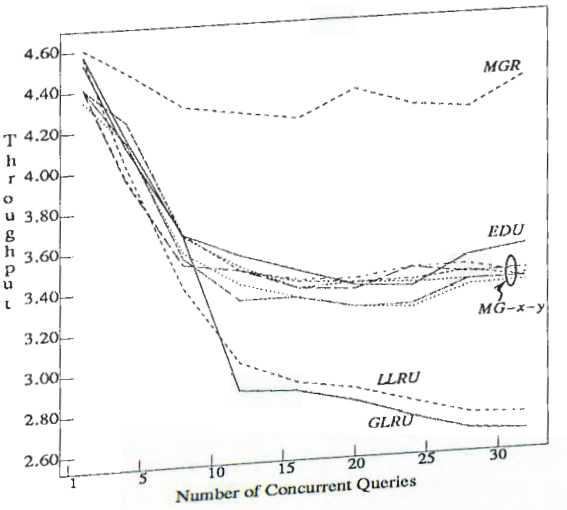

Figure 5.7: Throughput Comparison under Query Mix M3, No Data Sharing

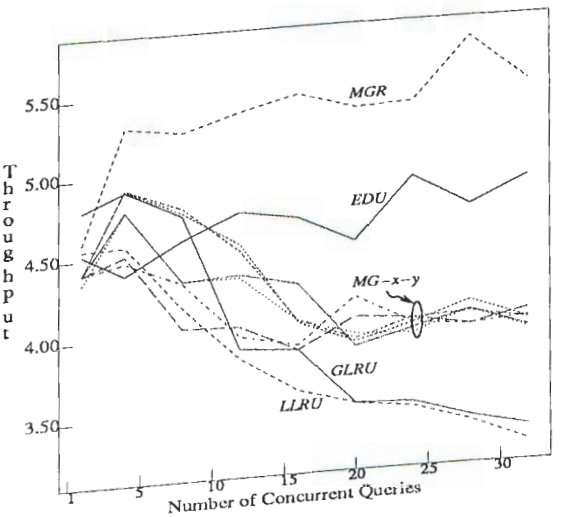

Figure 5.8: Throughput Comparison under Query Mix M3, Partial Data Sharing

Figure 5.9.a shows the effects of full data sharing. For concurrency levels between 1 and 8, GLRU outperform all other strategies including MGR. This is because in full data sharing, global LRU can keep the *locality* sets of several queries in buffers at a time. When the number of concurrent queries increases, however, this advantage

disappears as the overall locality set becomes too large to fit into the buffer pool, and thus MGR and EDU again perform best. In this simulation, the performance of EDU is close to that of MGR, which indicates that when data sharing increases, the impact of buffer allocation decreases.

When we shrink the buffer pool from 1,000 to 600 pages, the effect of full data sharing becomes less significant. This is shown in Figure 5.9.b, where GLRU now degrades drastically. The performance improvement of MGR over EDU now increases again. This implies that when buffer contention occurs, those buffer management algorithms which can characterize the reference behavior more accurately achieve better buffer utilization.

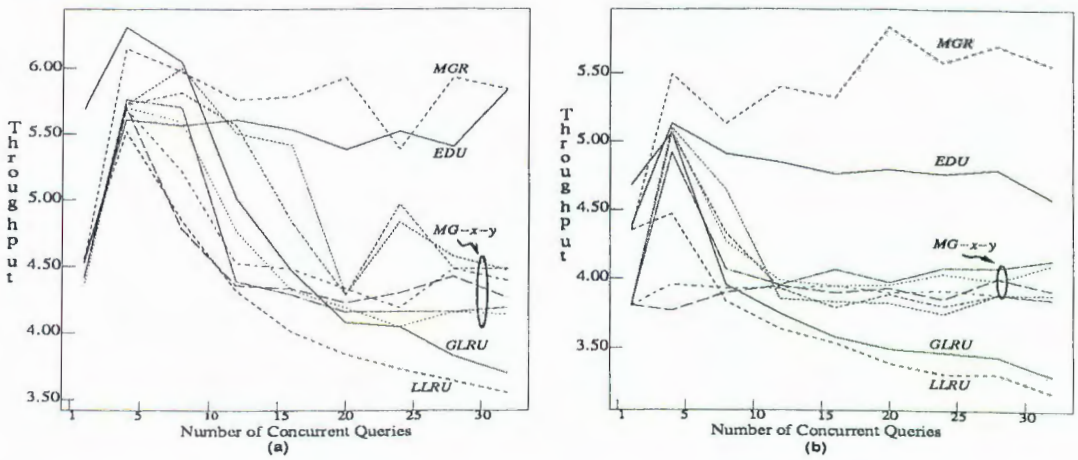

Figure 5.9: Query Mix M3, Full Data Sharing, (a) 1000 buffers, (b) 600 buffers

5.6 Conclusions

In this chapter, we have proposed a buffer allocation algorithm, MGR, for recurring queries using feedback of page faults from executions. This algorithm uses a quantitative model to characterize the page reference behavior of queries and allocates buffers to individual reference strings according to their quantified characteristics. The simu-

lation results show that MGR demonstrates significant performance improvement over the pattern prediction-based algorithm MG-x-y and the load control-based algorithm EDU. In all cases of query mixes with no data sharing, MGR outperforms the second best strategy with 15% – 30% throughput improvement on average. We also observed the effect of data sharing. The results showed that, except for the cases involving full data sharing with a very large buffer availability and small concurrency level, MGR is still favored.

The advantage of MGR can be attributed to the tuning of the buffer allocation technique based on the real access behavior obtained by query feedback rather than probabilistic query path analysis where crude assumptions such as uniformity have to be made. Furthermore, since queries are treated as reference strings, our approach is applicable not only to relational algebra access paths but also to access paths of other more advanced database systems such as deductive and object-oriented databases.

Chapter 6

Conclusions

To enhance system performance, traditional relational database systems have adopted various techniques to improve and tune the query optimization and evaluation phases. Such systems, however, overlook the potential advantage of recycling useful query results and of using informative feedback learned from query execution. This dissertation extends the traditional query optimizer to integrate query result caching and matching and explores the technique of using query feedback to adapt selectivity estimation and buffer allocation.

In Chapter 3, we developed an integrated framework of query optimization using cached results. The extended query optimizer is able to match and integrate in its execution plans query results cached from previous queries. The framework supports data-based caching and pointer-based caching, re-execution and incremental cache update strategies, and alternative cache replacement strategies. The best plan of a query, which may or may not use previously cached results, is selected based on an extended cost model which takes into account the costs of incremental updates and cache materialization. This work was implemented on a DBMS prototype, and the empirical performance study showed that, by using pointer-based caching and a dynamic update strategy, query throughput can be substantially improved under moderate query correlations and moderate update loads. The requirement of the disk

cache space is relatively small, and the extra optimization overhead introduced is more than offset by the time saved in query evaluation.

In Chapter 4, we proposed a novel approach for selectivity estimation. Capitalizing on the technique of recursive, weighted least-square-error, we devised an adaptive selectivity estimator which uses actual query result sizes as feedback to approximate the attribute distribution and to provide efficient and accurate estimations. The distribution is adjusted gradually and with little CPU overhead after each query execution in real time. The most significant advantage of this approach over traditional methods is that it incurs no off-line database access overhead for gathering statistics. Moreover, it adapts fairly well to updates and query locality, while such adaptation usually can not be achieved easily by traditional methods.

In Chapter 5, we proposed an adaptive buffer allocation algorithm for recurring queries. This algorithm uses a quantitative model and simple statistics obtained from executions to characterize the page reference behavior of queries; the algorithm allocates buffers to individual relation instances (of a query) according to their quantified characteristics. The advantage of this scheme is attributable to the tuning of the buffer allocation based on the real access behavior obtained by query feedback rather than probabilistic query path analysis where crude assumptions such as uniformity have to be made. Simulation results show that this scheme demonstrates significant performance improvement over the existing methods which are based on reference pattern classification or load control.

The techniques of using cached results and query feedback are not restricted to centralized RDBMSs only. In the emerging technology of *replicated systems* [D+94, Die94, GWD94] which support materialized views as data replica at different sites, the optimizer we proposed here can be easily adopted to provide replica transparency (to the users) and to make best use of the replica. In a multi-database or heterogeneous database system where independent database systems interoperate while preserving

their autonomy, the problem of *global* or *heterogeneous* query optimization is a great challenge since the participating DBMSs are autonomous and may not be able to provide the necessary information (such as query selectivities and costs) for global query optimization [DKS92, L+92, ZL94]. We believe that using query feedback is a proper solution to this problem because it can be implemented efficiently without changing the kernels of the individual participating DBMSs and because it provides accurate and dynamic information which reflects the current database image and system load. The above are just two of the many areas where using cached results and query feedback is profitable and/or can achieve much more than the existing methods. In the future, we expect to explore and apply the technology of query feedback to emerging database areas with dynamic environments where the traditional static and probabilistic methods are deemed inappropriate.

Chapter 7

Future Research

We would like to extend the ADMS CMO query optimizer to a client-server distributed environment ADMS$\pm$[RK86, RES93]. Caching query results on client workstations has the following advantages: (1) it reduces the data transmission over the network, (2) it reduces the server contention, and finally (3) it provides concurrent and parallel access to data cached on multiple disks of workstations. To fulfill these advantages, the query optimizer must be able to find a most efficient way to compute a query using the computation power of both client and server workstations and the results cached on the disks of client workstations.

The adaptive selectivity estimator can be improved and explored in several directions. First, we would like to extend this work to complex queries which involve joins or logical operators such as AND, OR, etc. We can refine the query feedback mechanism so that adaptation will stop after the approximating distribution converges and will be triggered again after updates. To achieve this, a condition for testing convergence and an appropriate threshold of estimation error for the adaptation trigger must be sought. Mathematical analysis of ASE is also desired in order to give deeper insight into its performance behavior under diverse query distributions and into its theoretical limits. As another possibility, we can use B-splines [dB78] as the model functions. The distribution thus formed is a piecewise polynomial, which is called a spline. Piece-

wise polynomials are much more suitable than regular polynomials in approximating attribute value distribution. In addition, since both histograms and polynomials are special cases of splines, using B-splines allows the database administrator to have the flexibility of choosing the most appropriate function, with the consideration for space overhead and actual data skewness.

We are also investigating the potential use of query feedback for query optimization in heterogeneous or multi-database systems. In a multi-database system, the internals of individual host DBMSs usually can not be accessed or altered from the outside. This makes the problem of query optimization in a multi-database system much more difficult than that in a single database system. For example, to join relations R1 and R2 which reside in two different DBMS hosts, it is not easy to determine whether R1 should be imported into the DBMS host of R2 to do the join or vice versa, since we do not know the optimization and access strategies of individual DBMS hosts. However, if we can predict the costs of both alternatives, we can make an appropriate decision. For this problem, we can use the elapsed time of query execution as feedback to regress the cost formulas. We classify queries into a few types and associate each type with a cost formula which contains certain parameters to be estimated. When a query is executed, we measure the elapsed time and use it to adapt the cost parameters. We believe that the use of query feedback is a promising solution for query optimization in a heterogeneous environment.

Appendix A

Recursive Solution for Weighted LSE

Using the notations defined in Section 4.3, for each query feedback $\zeta_i = (l_i, h_i, s_i)$, let

$$X_i = \left[\begin{array}{cccc} \Phi_0(h_i+1) - \Phi_0(l_i) & \Phi_1(h_i+1) - \Phi_1(l_i) & \ldots & \Phi_n(h_i+1) - \Phi_n(l_i) \end{array} \right]$$

Now suppose m query feedbacks ζ_i, $i = 1, 2, \ldots, m$ are given. It is not hard to see that Eq. 4.10 can be rearranged and expressed as

$$||\mathcal{X}^m * A - \mathcal{Y}^m||^2, \tag{A.1}$$

where $\mathcal{X}^m$ is a $m \times n$ weighted matrix, $\mathcal{Y}^m$ is a $m \times 1$ weighted vector, and the respective ith rows of $\mathcal{X}^m$ and $\mathcal{Y}^m$ are defined as

$$\mathcal{X}_i^m = (\prod_{j=i+1}^{m} \alpha_j) \cdot \beta_i \cdot X_i, \quad \mathcal{Y}_i^m = (\prod_{j=i+1}^{m} \alpha_j) \cdot \beta_i \cdot s_i \tag{A.2}$$

According to Eqs. 4.7 and A.2, the optimal values of A that minimize Eq. A.1 is computed as

$$A_m^* = G_m N_m \tag{A.3}$$

$$G_m = [\sum_{i=1}^{m} (\mathcal{X}_i^m)^t \mathcal{X}_i^m]^{-1} = [\alpha_m^2 \sum_{i=1}^{m-1} (\mathcal{X}_i^{m-1})^t \mathcal{X}_i^{m-1} + \beta_m^2 X_m^t X_m]^{-1} \tag{A.4}$$

$$N_m = \sum_{i=1}^{m} (\mathcal{X}_i^m)^t \mathcal{Y}_i^m = \alpha_m^2 \sum_{i=1}^{m-1} (\mathcal{X}_i^{m-1})^t \mathcal{Y}_i^{m-1} + \beta_m^2 X_m^t s_m \tag{A.5}$$

First, we derive the recursive formula for G_m. From the above equations we have

$$G_m^{-1} = \alpha_m^2 G_{m-1}^{-1} + \beta_m^2 X_m^t X_m \tag{A.6}$$

$$N_m = \alpha_m^2 N_{m-1} + \beta_m^2 X_m^t s_m \tag{A.7}$$

Do $G_m \times$ (Eq.A.6) $\times G_{m-1}$, we obtain

$$G_{m-1} = \alpha_m^2 G_m + \beta_m^2 G_m X_m^t X_m G_{m-1} \tag{A.8}$$

Multiply the above equation by X_m^t, and we get

$$G_{m-1} X_m^t = G_m X_m^t [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t] \tag{A.9}$$

Rearrange Eq.A.9 and multiply it by $X_m G_{m-1}$,

$$G_{m-1} X_m^t [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} X_m G_{m-1} = G_m X_m^t X_m G_{m-1} \tag{A.10}$$

Finally, substitute Eq.A.10 into Eq.A.8 and rearrange, we obtain the recursive formula for G_m:

$$G_m = (\frac{1}{\alpha_m})^2 G_{m-1} - (\frac{\beta_m}{\alpha_m})^2 G_{m-1} X_m^t [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} X_m G_{m-1} \tag{A.11}$$

Now we can derive the recursive formula for A^*. Substitute Eqs. A.7 and A.11 into Eq. A.3 and let $\Delta = [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1}$, we obtain

$$\begin{aligned}
A_m^* &= G_{m-1} N_{m-1} - \beta_m^2 G_{m-1} X_m^t \Delta X_m G_{m-1} N_{m-1} \\
&\quad + (\frac{\beta_m}{\alpha_m})^2 G_{m-1} X_m^t s_m - (\frac{\beta_m^4}{\alpha_m^2}) G_{m-1} X_m^t \Delta X_m G_{m-1} X_m^t s_m \\
&= A_{m-1}^* - G_{m-1} X_m^t \Delta [\beta_m^2 X_m A_{m-1}^* - (\frac{\beta_m}{\alpha_m})^2 (\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t) s_m \\
&\quad + (\frac{\beta_m^4}{\alpha_m^2}) X_m G_{m-1} X_m^t s_m] \\
&= A_{m-1}^* - G_{m-1} X_m^t \Delta (\beta_m^2 X_m A_{m-1}^* - \beta_m^2 s_m) \\
&= A_{m-1}^* - \beta_m^2 G_{m-1} X_m^t \Delta (X_m A_{m-1}^* - s_m)
\end{aligned} \tag{A.12}$$

The term $G_{m-1}X_m^t\Delta$ in the above expression can be further simplified as

$$
\begin{aligned}
& G_{m-1}X_m^t[\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} \\
= \; & [G_m G_m^{-1}] G_{m-1} X_m^t [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} \\
& \text{now substitue } G_m^{-1} \text{ above with Eq. A.6} \\
= \; & G_m[\alpha_m^2 G_{m-1}^{-1} + \beta_m^2 X_m^t X_m] G_{m-1} X_m^t [\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} \\
= \; & G_m[\alpha_m^2 X_m^t + \beta_m^2 X_m^t X_m G_{m-1} X_m^t][\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} \\
= \; & G_m X_m^t[\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t][\alpha_m^2 + \beta_m^2 X_m G_{m-1} X_m^t]^{-1} \\
= \; & G_m X_m^t
\end{aligned}
\tag{A.13}
$$

Therefore, from Eq.A.12 and A.13 we get the recursive formula for A^*:

$$
\underline{A_m^* = A_{m-1}^* - \beta_m^2 G_m X_m^t (X_m A_{m-1}^* - s_m)} \tag{A.14}
$$

Appendix B

Update Workload Specifications

In the experiments, an update query is simulated by its effect on the value distribution of the attribute of interest. An update query is specified by five parameters:

$$(i, N, D, [min, max], prob_{INS}),$$

where i means this update takes place immediately before the ith query in the query stream. N is the number of tuples updated (either inserted or deleted). Each tuple's attribute value is randomly generated from range $[min, max]$ according to a distribution D. A tuple is inserted with probability $prob_{INS}$ or deleted with probability $1 - prob_{INS}$. Three different update workloads are tested, each of which is interleaved with another 40 random selection queries. The three update workloads are specified in the following, where $U(x, y)$ denotes the uniform distribution among range $[x, y]$ and $N(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ.

LOAD1: $(11, 4500, U(-50, 250), [-50, 250], 1.0)$.

LOAD2: $(11, 2250, U(-150, 550), [-150, 550], 0.75)$,
$(17, 2250, U(-150, 550), [-150, 550], 0.75)$,
$(23, 2250, U(-150, 550), [-150, 550], 0.75)$,
$(29, 2250, U(-150, 550), [-150, 550], 0.75)$.

LOAD3: $(11, 3000, N(-63, 50), [-150, 25], 0.9)$,

$(17, 1500, N(112, 40), [25, 200], 0.1),$

$(23, 2250, N(290, 60), [200, 375], 1.0),$

$(29, 2250, N(455, 50), [375, 550], 0.4).$

Bibliography

[A+76] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[AG67] A.E. Albert and L.A. Gardner. *Stochastic Approximation and Nonlinear Regression.* M.I.T. Press, Cambridge, Massachusetts, 1967.

[AL80] M.E. Adiba and B. G. Lindsay. Database snapshots. In *Procs. of the 6th Intl. Conf. on Very Large Data Bases (VLDB)*, 1980.

[BCL89] J.A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, 1989.

[BD84] H. Boral and D. DeWitt. A methodology for database system performance evaluation. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1984.

[BDT83] D. Bitton, D.J. DeWitt, and C. Turbyfill. Benchmarking database systems, a systematic approach. In *Procs. of the 9th Intl. Conf. on VLDB*, 1983.

[Bel66] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[BLT86] J.A. Blakeley, P. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1986.

[CD85] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 127–141, 1985.

[Chr83a] S. Christodoulakis. Estimating block transfers and join sizes. In *Procs. of 1983 ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–54, New York, 1983.

[Chr83b] S. Christodoulakis. Estimating record selectivities. *Inf. Syst.*, 8(2):105–115, 1983.

[CL73] C. L. Chang and C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[CR93] C.M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Procs. of the 19th Intl. Conf. on Very Large Data Bases*, 1993.

[CR94a] C.M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[CR94b] C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Procs. of the 4th Intl. Conf. on Extending Database Technology*, 1994.

[D+94] D. Daniels et al. Oracles's symmetric replication technology and implications for application. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[dB78] C. de Boor. *A practical guide to splines.* Springer-Verlag, New York, 1978.

[Die94] D.J. Dietterich. DEC Data Distributor: for data replication and data warehousing. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Procs. of the 18th VLDB Conference*, 1992.

[DR92] A. Delis and N. Roussopoulos. Evaluation of an enhanced workstation-server DBMS architecture. In *Procs. of the 18th Intl. Conf. on VLDB*, 1992.

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM TODS*, 9(4):560–595, 1984.

[FCL93] M. Franklin, M. Carey, and M. Livny. Local disk caching for client-server database systems. In *Procs. of the 19th Intl. Conf. on Very Large Data Bases*, 1993.

[Fed84] J. Fedorowicz. Database evaluation using multiple regression techniques. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 70–76, Boston, MA, 1984.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, 1982.

[FNS91] C. Faloutsos, R. T. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Procs. of the 17th Intl. Conf. on VLDB*, pages 265–274, 1991.

[GW89] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the 1989 ACM SIGMOD Conference on the Management of Data*, pages 358–366, Portland, OR, 1989.

[GWD94] A. Gorelik, Y. Wang, and M. Deppe. Sybase Replication Server. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 440–453, 1987.

[HOT88] W. Hou, G. Ozsoyoglu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Procs. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 276–287, 1988.

[HS92] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 341–350, San Diego, CA, 1992.

[HS93] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1993.

[IC91] Y.E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, Denver, Colorado, 1991.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM TODS*, 9(3):482–502, 1984.

[IK90] Y.E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1990.

[Ioa93] Y.E. Ioannidis. Universality of serial histograms. In *Procs. of the 19th Intl. Conf. on VLDB*, Dublin, Ireland, 1993.

[IW87] Y.E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1987.

[J+93] C. S. Jensen et al. Using differential techniques to efficiently support transaction time. *VLDB Journal*, 2(1):75–111, 1993.

[Jhi88] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Procs. of the 14th Intl. Conf. on VLDB*, 1988.

[Kam87] M. Kamimura. Caching query results: A survey based on the simulation of caching query results in a database. Class project report, Dept. of Computer Science, University of Maryland at College Park, MD, 1987.

[Kap80] J. Kaplan. Buffer management policies in a database environment. Master's thesis, University of California, Berkeley, 1980.

[KK85] N. Kamel and R. King. A method of data distribution based on texture analysis. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–325, Austin, Texas, 1985.

[KMN89] D. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[KS86] H. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill Book Company, New York, NY, 1986.

[L+92] H. Lu et al. On global query optimization in multidatabase systems. In *2nd Intl. Workshop on Res. Issues on Data Engineering*, 1992.

[LHM86] B. Lindsay, L. Haas, and C. Mohan. A snapshot differential refresh algorithm. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 53–60, 1986.

[LN90] R. J. Lipton and J. F. Naughton. Practical selectivity estimation through adaptive sampling. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 1–11, Atlantic City, NJ, 1990.

[LST83] E. Lefons, A. Silvestri, and F. Tangorra. An analytic approach to statistical databases. In *Procs. of the 9th Intl. Conf. on VLDB*, 1983.

[LW86] S. Lafortune and E. Wong. A state transition model for distributed query processing. *ACM TODS*, 11(3):294–322, 1986.

[LY85] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In *Procs. of the 11th Intl. Conf. on VLDB*, pages 259–269, 1985.

[Lyn88] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Procs. of the 14th Intl. Conf. on VLDB*, pages 240–251, Los Angeles, CA, 1988.

[M+70] R. Mattson et al. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[MCS88] M.V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3), 1988.

[MD88] M. Muralikrishma and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 28–36, Chicago, Illinois, 1988.

[MO79] T.H. Merrett and E. Otoo. Distribution models of relations. In *Procs. of the 5th Intl. Conf. on VLDB*, pages 418–425, Rio De Janero, Brazil, 1979.

[NFS91] R. T. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Procs. of 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 387–396, 1991.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Pub. Co., 1980.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 256–275, Boston, MA, 1984.

[Rei76] A. Reiter. A study of buffer management policies for data management systems. Technical Report TR-1619, Mathematics Research Center, University of Wisconsin-Madison, 1976.

[RES93] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A testbed for incremental access methods. *Appeared in IEEE Trans. on Knowledge and Data Engineering*, 1993.

[RH80] D.J. Rosenkrantz and H.B. Hunt. Processing conjunctive predicates and queries. In *Procs. of the 6th Intl Conf. on VLDB*, 1980.

[RK86] N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS±. *Computer*, 19(12):19–25, 1986.

[Rou82a] N. Roussopoulos. The logical access path schema of a database. *IEEE Trans. on Software Engineering*, SE-8(6):563–573, 1982.

[Rou82b] N. Roussopoulos. View indexing in relational databases. *ACM TODS*, 7(2):258–290, 1982.

[Rou91] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, 1991.

[RR76] J. Rodriguez-Rosell. Empirical data reference behavior in data base systems. *IEEE Computer*, 9(11), Nov. 1976.

[S+79] P. G. Selinger et al. Access path selection in a relational database management system. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[S+89] X. Sun et al. Solving implication problems in database applications. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 185–192, 1989.

[SB76] S.W. Sherman and R.S. Brice. Performance of a database manager in a virtual memory system. *ACM TODS*, 1(4), 1976.

[SB83] W. Samson and A. Bendell. Rank order distriburions and secondary key indexing. In *Procs. of the 2nd Intl. Conf. on Databases*, Cambridge, England, 1983.

[Sel87] T. Sellis. Efficiently supporting procedures in relational database systems. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1987.

[Sel88] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Inform. Systems*, 13(2), 1988.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1988.

[Shi93] K. Shim. *Advanced Query Optimization Techniques for relational database systems.* PhD thesis, Dept. of Computer Science, University of Maryland, College Park, 1993.

[Shm84] O. Shmueli. Maintenance of views. In *Procs. of the ACM SIGMOD Intl. Conf. on the Management of Data*, pages 240–255, Boston, MA, 1984.

[SLRD93] W. Sun, Y. Ling, N. Rishe, and Y. Deng. An instant and accurate size estimation method for joins and selection in a retrieval-intensive environment. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 79–88, Washington, DC, 1993.

[SS82] G. Sacca and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Procs. of the 8th Intl. Conf. on VLDB*, pages 257–262, 1982.

[SS86] G. Sacca and M. Schkolnick. Buffer management in relational database systems. *ACM TODS*, 11(4):474–498, 1986.

[Swa89] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Procs. of the ACM SIGMOD Intl. Conf. on the Management of Data*, pages 367–376, Portland, Oregon, 1989.

[SWK76] M. Stonebraker, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM TODS*, 1(3):189–222, 1976.

[TLF77] C. Wood T. Lang and E. Fernandez. Database buffer paging in virtual storage systems. *ACM TODS*, 2(4), 1977.

[Tue76] W. Tuel. An analysis of buffer paging in virtual storage systems. *IBM Journal of Research and Development*, 1976.

[Val87] P. Valduriez. Join indices. *ACM TODS*, 12(2):218–246, 1987.

[WY76] E. Wong and K. Youssefi. Decomposition : A strategy for query processing. *ACM TODS*, pages 223–241, Sept. 1976.

[Yao77] S.B. Yao. Approximating block accesses in database organizations. *Communications of ACM*, 20(4), 1977.

[YL90] H. Yoo and S. Lafortune. Enhancing efficiency and modularity in join query optimization. Technical Report CSE-TR-77-90, University of Michigan, 1990.

[You84] P. Young. *Recursive estimation and time-series analysis.* Springer-Verlag, New York, 1984.

[ZL94] Q. Zhu and P.-A. Larson. A query sampling method for estimating local cost parameters in a multidatabase system. In *Procs. of the 10th Intl. Conf. on Data Engineering*, 1994.