INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
On automated query modification techniques for databases

Du, Kaizheng, Ph.D.

Case Western Reserve University, 1993
ON AUTOMATED QUERY MODIFICATION

TECHNIQUES FOR DATABASES

by

KAIZHENG DU

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Thesis Advisor: Dr. Gultekin Ozsoyoglu

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY
May, 1993
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

KAIZHENG DU

candidate for the Ph. D.
degree.*

(signed) [signature]
(chair)

[signature]

[signature]

date 4/9/93

*We also certify that written approval has been obtained for any proprietary material contained therein.
I grant to Case Western Reserve University the right to use this work, irrespective of any copyright, for the University's own purposes without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Kaiheng Du
ON AUTOMATED QUERY MODIFICATION TECHNIQUES FOR DATABASES

Abstract

by

KAIZHENG DU

In many cases, in order to satisfy certain constraints specified in a database query, the query needs to be modified. However, users' "manual" modification is an extra burden on users and, sometimes, may not be correct when users lack knowledge about the database. In a real-time environment, users' manual modification may be too slow to make a query satisfy a given time deadline. In this thesis, a general automated database query modification model is proposed for automatically modifying database queries with constraints. Five types of query modification constraints, namely, time constraints, error constraints, aggregate function constraints, PCF constraints and count proportionality constraints are introduced. To enable query modification to be performed automatically, two query modification protocols, namely, the use of superset/subset chains based on relation fragmentation and the use of sampling data, are specified.

Based on the two query modification protocols, a detailed query modification mechanism for enforcing time and error constraints are given. An iterative query evaluation technique is used to process nonperiodically occurring queries with time constraints. An incremental query evaluation technique is used to process periodically occurring queries with time constraints. Error removal
and error estimation techniques are introduced to enforce error constraints. Finally, extending the above listed techniques into enforcing the remaining three types of constraints is briefly summarized.

In this thesis, query estimation techniques for aggregate relational algebra queries COUNT, SUM and AVERAGE are also presented. Two statistical estimators, the Jackknife estimator and the Chao’s estimator, for COUNT queries with projection are used. Estimators using double sampling technique for SUM and AVERAGE queries are introduced, and new sampling plans based on systematic sampling and stratified sampling to increase the estimation accuracy are investigated. These new estimators and sampling plans are extensively used in automated query modification.

Some of the techniques and associated algorithms proposed in this thesis have been implemented in CASE-DB, which is a prototype relational database management system developed at Case Western Reserve University.
For my parents and my lovely wife Yan
ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Gultekin Ozsoyoglu, my Ph.D. thesis advisor, for his continuous support and encouragement during my Ph.D. program at CWRU. His valuable directions and suggestions were very helpful in my Ph.D. study and research.

Second. I would like to thank Dr. George W. Ernst, Dr. Kenneth A. Loparo, and Dr. Kumar Vadaparty for reviewing my thesis and being in my program committee.

Next. I would also like to extend my appreciation to both the Li Foundation for providing me a scholarship for my Ph.D. study and Tianjin University for encouraging me to go abroad to study for a Ph.D. degree.

Last but not least. my deepest gratitude goes to my family for their never ending support and love.
Contents

1. Introduction ................................................................................. 1
   1.1 Overview .............................................................................. 1
2. Query Modification ........................................................................ 10
   2.1 Modified Queries .................................................................. 10
   2.2 Query Enlargement and Query Reduction .............................. 11
   2.3 Automated Query Modification Model ................................. 12
   2.4 Modification Protocols ....................................................... 13
      2.4.1 Protocol One: Use of Superset/Subset Chains ................ 13
      2.4.2 Protocol Two: Use of Sampled Data .............................. 15
   2.5 Modification Constraints .................................................... 16
   2.6 Modification Operators and Modification Strategies ............ 22
   2.7 The General Algorithm for Enforcing Constraints ................. 23
3. Enforcing Time Constraints for Non-Aggregate Queries

3.1 Introduction

3.2 Fragment Selection Problem

3.3 Iterative Query Evaluation Transformations

3.3.1 Transformations for Single-Operator Queries

3.3.2 Transformations for Multiple-Operator Queries

3.4 Query Evaluation Algorithms for Iterative Evaluation

3.4.1 Algorithms and costs for query transformations

3.4.2 Using Indexed Files

3.4.3 Transformation Optimization

3.5 Other Issues About Fragmenting Relations

3.5.1 Diamond-Structured Lattices

3.5.2 Arbitrary Relation Fragmentation Lattices and Stratification

3.6 Experimental Results for Iterative Query Evaluation

3.7 Incremental Query Evaluation Transformations

3.7.1 Transformations for Insertion-Only Single-Operator Queries

3.7.2 Transformations for Deletion-Only Single-Operator Queries

3.7.3 Transformations for Single-Operator Queries Containing Increments

3.8 Algorithm for Processing Arbitrary, Periodically Occurring Queries

4. Enforcing Error Constraints
4.1 Introduction ...................................................... 70
4.2 Error Removal Technique ....................................... 71
4.3 Error Estimation Technique ..................................... 84
4.4 Simplification of $Q - Q_m$ and $Q_m - Q$ .................... 88
   4.4.1 Constructing $Q'_m$ for Arbitrary Queries ............... 90
   4.4.2 Optimizing $Q'_m$ ........................................ 94
4.5 Fragmentation Attribute Value Occurrence Probability Method to Estimate
   Error Size for Single Set Difference Queries .................. 97
   4.5.1 Equal-Probability Assumptions .......................... 97
   4.5.2 Non-Equal-Probability Distribution ..................... 101
   4.5.3 Multiple Fragmentation Attributes ...................... 103

5. Estimating COUNT, SUM, and AVERAGE RA Queries............ 105
   5.1 Introduction .................................................. 105
   5.2 New Estimators for COUNT($\pi_X(R)$) ...................... 106
      5.2.1 Goodman's Estimator ................................ 107
      5.2.2 Jackknife Estimators .................................. 108
      5.2.3 Chao's Estimator ..................................... 110
   5.3 Estimating $SUM_A(\pi_X(R))$ Using Double Sampling ........ 111
   5.4 New Sampling Plans .......................................... 113
      5.4.1 Sampling Plan Based on Systematic Sampling ........ 113
      5.4.2 Sampling Plan Based on Stratified Random Sampling .. 114
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Generic Estimators for AGG Queries</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>5.5.1 Estimator for Select-Join-Intersection-Expressions</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>5.5.2 Estimator for Project-Select-Join-Intersection-Expressions</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>5.5.3 General Algorithm for Estimating AGG(E)</td>
<td>117</td>
</tr>
<tr>
<td>5.6</td>
<td>Experimental Results</td>
<td>119</td>
</tr>
<tr>
<td>6.</td>
<td>Enforcing Aggregate Function Constraints, PCF Constraints and Count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proportionality Constraints</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>6.1 Enforcing Aggregate Function Constraints</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>6.2 Enforcing PCF Constraints</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>6.3 Enforcing Count Proportionality Constraints</td>
<td>126</td>
</tr>
<tr>
<td>7.</td>
<td>Conclusions and Future work</td>
<td>128</td>
</tr>
<tr>
<td>8.</td>
<td>Appendix</td>
<td>140</td>
</tr>
</tbody>
</table>
List of Figures

1.1 An Example of a Relation ........................................ 6
2.1 The Subset and Superset Chains. ............................... 14
2.2 Logical and Physical Fragmentations of $R$. .................... 15
2.3 Error Characterization in Query Reduction and Enlargement of a Single Set Difference Query. .......................... 19
2.4 The General Algorithm for Enforcing Modification Constraints. .................. 24
3.1 Relation Fragmentation Chains .................................. 27
3.2 Query Evaluation Algorithm for Real-Time, Non-Aggregate Queries in CASE-DB ........................................... 28
3.3 Algorithm for $f_{i,k} \cap g_{j,m}$ Using Indexed Files. ............ 49
3.4 Algorithm for $f_{i,k} \cup g_{j,m}$ Using Indexed Files. ............... 50
3.5 Algorithm for $f_{i,k} - g_{j,m}$ Using Indexed Files. ............... 50
3.6 Diamond-Structured Fragmentation Lattice. ...................... 53
3.7 Relation Fragmentation Lattice of the FURNACES Relation. ........ 55
3.8 Conversion from an Arbitrary Relation Fragmentation Lattice to a Chain. .......... 56
3.9 Algorithm for Converting an Arbitrary Relation Fragmentation Lattice into a Chain-Structured Relation Fragmentation Lattice. ............. 57
3.10 Venn Diagram for a Single Relation $R$ with $\Delta R$ and $\nabla R$. ................. 64
3.11 Venn Diagram for Two Overlapped Relations $R$ with $\Delta R$ and $\nabla R$ and $S$ with $\Delta S$ and $\nabla S$. ........................................... 65

3.12 Incremental Query Evaluation Algorithm for Time-Constrained, Periodically occurring Queries. ........................................... 69

4.1 Subset Chains for Relations $R$ and $S$. ..................................... 74

4.2 Algorithm A-ERROR.REMOVAL. .............................................. 82

4.3 Subset Chains for Relations $R$, $S$ and $T$. .................................. 83

4.4 The Venn Diagram of $R - S$. .................................................. 87

4.5 Algorithm A-ERROR.SIZE.ENFORCEMENT. ................................... 89

4.6 Venn Diagram for $Q = Q_1 - Q_2$ and $Q_m = Q_{1m} - Q_{2m}$. ......... 92

4.7 Algorithm for $Q_m'$ Optimization. ............................................ 95

4.8 Algorithm for $Q_m'$ Construction. ............................................ 98


4.10 A three-dimensional hypercube for $D_A$. .................................. 103

5.1 Algorithm for Computing the Jackknife Estimator $\hat{J}$. .................. 110

5.2 Algorithm for Computing the Chao's Estimator $\hat{C}$. ...................... 111

5.3 Simple Random Sampling-Based Estimator Evaluation Algorithm for $AGG(E)$, where $E$ is an SJI-expression. ................................. 117

5.4 A Simple-Random-Sampling-Based Estimator Construction Algorithm for $AGG(E)$, where $E$ is a PSJI-expression. ............................. 118

5.5 A Simple-Random-Sampling-Based Estimator Construction and Evaluation Algorithm for $AGG(E)$, Where $E$ is an Arbitrary RA-expression. .... 120
7.1 The Topologic Structures of Local Computer Networks. ................. 132

8.1 Algorithm for Testing the Null Hypothesis $H_{ab}$ of the Jackknife Estimator. . 141
Chapter 1

Introduction

1.1 Overview

Query modification has long been discussed in the area of semantic query optimization, where the modified query produces the same output as the original query [Hamm 80, King 81, Chak 85]. In recent years, new research has emerged in query modification with the goal of modifying a query into another one in order to satisfy users' expectations on the output of the query [HoOT 89, SmiL 89, Chau 90, OzDu 91, OzDG 92]. In this type of query modification, the semantics and the output of the modified query may be different from that of the original query. In this thesis, we discuss the second type of query modification.

Generally speaking, in the second type of query modification, a database query has to be modified in order to satisfy certain constraints on the query (e.g., timing constraints, security constraints, etc.). The user specifies the constraints (also called modification constraints) to emphasize his or her expectations about the desired features of the output relation. We now give three reasons for a query to be modified:

- **Query with a Time Constraint.**

  A time-constrained query is of the form "get the information x in at most t time units". If the given query can not be finished within the given time, it needs to be modified into another query which meets the time constraint. For example, the user may need within 10 seconds the names of factory furnaces with "dangerously high" temperature readings in the last week. The database system, judging that the query cannot be
evaluated in 10 seconds, may modify it into the names of "high-security" furnaces with "dangerously high" temperature readings in the last week.

- **Security Enforcement.**

A given query may violate the security of the database; and needs to be modified. For instance, a query may request the average salary of engineers with Ph.D. The database system may change the query to the average salary of all engineers with B.S., M.S. and Ph.D. so as to protect the salary information of a small number of engineers with Ph.D.

- **Exploratory Data Analysis.**

Sometimes a user may be performing exploratory data analysis and may present an initial query with a set of constraints. The user's expectations are that if the output of the initial query does not satisfy the given constraints, the database system automatically (using pre-defined modification protocols) modifies the query such that the output of the modified query satisfies the given constraints. We give an example.

**Example 1.1.** The precision of a product is determined by the precision of the machine used, and the skill level of the operator. Suppose the product with a certain precision requirement can be made by either a low-skilled operator operating a high-precision machine, or a high-skilled operator operating a low-precision machine. An exploratory data analysis query may ask at least K combinations of operators and machines such that the product precision is not less than P. Due to resource considerations, the operators are preferred to be with skill level X or low, and the machines are preferred to be with precision Y or low.

If the number of combinations found from the operators with skill level X or low plus machines with precision Y or low is less than K, then the initial query needs to be (repeatedly) modified by relaxing the requirement either on operators' skills or machines' precision (e.g., by adding high-skilled operators and/or high-precision machines into consideration).
In the case that the output of a user's original query does not satisfy the given constraints, the user may need to repeatedly modify his or her query until the given constraints are satisfied. The query modification work is obviously an extra burden on users. Also, users' "manual" modification may not be correct when users lack knowledge about the database. In a real-time environment, users' manual modification may be too slow to make a query satisfy a given time deadline. Therefore, an automated query modification mechanism may be beneficial in such environments. For instance, in example 1.1, if there are pre-defined modification protocols between the user and the database system, say, preference orders on operators and machines, then the query modification can be automatically performed by the database system.

Automated query modification (such as those in exploratory data analysis) may be beneficial in applications such as marketing analysis, decision making, industrial control, and scientific data analysis.

The automated query modification model to be introduced in the thesis consists of four main components, namely, a database instance, a set of query modification protocols, a set of query modification constraints and a set of query modification methods. To simplify the problem, in this thesis, we only consider the relational data model. The query modification protocols are query input data replacement rules followed by both the user and the database management system during query modification. Two types of query modification protocols are introduced in the thesis, namely, use of superset/subset chains and use of sampled data. The query modification constraints are specified by a user to express his or her expectations on the database response to his or her query. If the result of the user's original query does not satisfy the given constraints, the query is to be automatically and repeatedly modified until the constraints are satisfied. Five types of query modification constraints are introduced in this thesis, namely, the real-time constraints, the error constraints, the aggregate function constraints, the PCF constraints and the count pro-
portionality constraints. The query modification methods are automatic constraint enforcement strategies and algorithms, query rewrite rules, etc.

The goal of this thesis is to provide a framework that formalizes the basic ideas about the automated query modification model and introduce its enforcement issues.

The main contributions of this thesis are:

- an automated query modification model (chapter 2);
- automated query modification protocols (chapter 2);
- classification of query modification constraints (chapter 2);
- iterative query evaluation techniques for processing real-time, non-periodically occurring queries (chapter 3);
- incremental query evaluation techniques for processing real-time, periodically occurring queries (chapter 3);
- query modification error removal and estimation techniques for enforcing error constraints (chapter 4);
- new query estimation techniques for processing aggregate queries, which are used in automated query modification (chapters 4 and 5).

Some of the techniques and associated algorithms proposed in this thesis have been implemented in CASE-DB, a real-time DBMS that is prototyped at Case Western Reserve University.

1.2 Related Work

There are many papers addressing query modification [Hamm 80, King 81, Chak 85, HoOT 89, SmiL 89, Chau 90, OzDu 91, OzDG 92]. However, among these papers, there are only a few of them that address automated query modification. In [Chau 90], Chaudhuri introduces a query generalization framework
aimed at providing tools and techniques to perform query modification automatically. But, automated query modification mechanisms are not mentioned in his paper. Smith and Liu [SmiL 89] and Vrbsky and Liu [VrbL 91] give a methodology for finding approximate answers to relational algebra queries. In their approach, with an increase in the amount of time used, the accuracy of the approximate result is improved. But their work does not contain any risk control mechanism that is used in our query modification system [OzDG 92].

Hou and Oszoyoglu [HoOT 88, HoOT 89, HouO 91] and Lipton and Naughton [LiNa 89, LiNa 90, LiNS 90] present different statistical estimators and sampling techniques to estimate query sizes. Olken and Rotem [OlkR 86] discuss how to obtain samples from the results of relational queries without first forming the queries. In this thesis, we introduce new query estimation techniques using new statistical estimators (Jackknife estimator [BuOv 79] and the Chao's estimator [Chao 84]) and new sampling plans (double sampling and stratified sampling) for estimating aggregate queries.

Rosenthal, Chakravarthy and Blaustein [RoCB 89] use incremental relations to evaluate the net changes to stored or derived relations. In [QiWi 91], Qian and Wiederhold also present an algorithm for incremental recomputation of active relational expressions based on finite differencing techniques. In comparison, in this thesis, our incremental evaluation method focuses on using incremental and decremental relations to efficiently evaluate the current results of periodically occurring queries.

Constraint specification and enforcement in Constraint Logic Programming (CLP) are introduced by Jaffar, Michaylov, Walinsky, Lassez, Borning, Maher, etc. in their papers [JaMi 86, JaLa 87, Wali 89, BMMW 89]. Constraint hierarchies are discussed by Borning, Duisberg, etc. in [BoDF 87]. Chaudhuri defines extended queries which express additional constraints on the answer set, and designates some of the conditions in the relational query as flexible [Chau 90]. In this thesis, we define five types of query modification constraints
that can be used with relational algebra queries. Using these constraints enables users to express their expectations on the database responses of their queries.

1.3 Relational Data Model

The relational model was first introduced by Codd [Codd 70]. The mathematical concept underlying the relational data model is the set-theoretic relation, which is a subset of the Cartesian product of a list of domains. A domain is simply a set of values. For example, the set of integers is a domain. The Cartesian product of domains \( D_1, D_2, \ldots, D_k \), written \( D_1 \times D_2 \times \ldots \times D_k \), is the set of all \( k \)-tuples \((v_1, v_2, \ldots, v_k)\) such that \( v_1 \) is in \( D_1 \), \( v_2 \) is in \( D_2 \), and so on.

A relation is any subset of the Cartesian product of one or more domains. The members of a relation are called tuples. Each relation that is a subset of \( D_1 \times D_2 \times \ldots \times D_k \) is said to have degree \( k \). A tuple \((v_1, v_2, \ldots, v_k)\) has \( k \) components: the \( i^{th} \) component is \( v_i \).

A relation can be viewed as a table, where each row is a tuple and each column corresponds to one component. The columns are often given names, called attributes. The set of attribute names for a relation is called the relation scheme. If we name a relation REL, and its relation scheme has attributes \( A_1, A_2, \ldots, A_k \), the relation scheme is often written as REL\((A_1, A_2, \ldots, A_k)\). A relation is an instance of a relation scheme. An example of a relation with relation scheme EMPLOYEE\((\text{NAME}, \text{SSN}, \text{SALARY})\) and three tuples is given in Figure 1.1.
The collection of relation schemes used to represent information is called a relational database scheme, and the instances of the corresponding relation schemes is called the relational database.

Relational Algebra (RA) is a query language for manipulating relation instances. We now describe the five basic RA operators:

Difference (−): The difference between two relations A and B, \( A - B \), is the set of tuples belonging to A, but not to B.

Natural Join (⊗): The natural join of two relations A and B, \( A \bowtie B \), combines a tuple belonging to A and a tuple belonging to B whenever their common attributes have identical values, and eliminates duplicate columns.

Intersection (∩): The intersection of two relations A and B, \( A \cap B \), is the set of all tuples belonging to both A and B.

Projection (π): The projection operation selects chosen attribute(s) from a given relation and discards the other column(s).

Selection (σ): The selection operation is used to select a subset of the tuples in a relation satisfying a selection formula.

Union (∪): The union of two relations A and B, \( A \cup B \), is the set of tuples t belonging to either A or B (or both).

An aggregate function \( f_A(R) \), where f is one of COUNT, SUM, AVERAGE, MAX and MIN, takes a group of attribute A values in relation R as input, and produces an aggregate value as output.

1.4 Statistics Terminology

In this section, we present a brief description of the statistics terminology used in this thesis. Let \( \Psi \) be a parameter of interest such as a population mean or a population total. An estimate of the parameter \( \Psi \) is a number computed
from the sample information, which serves as an estimate for the value of $\theta$. The function of the observation, which is used to obtain an estimate is called an \textit{estimator}, and is denoted by $\hat{\theta}$.

Two of the desirable properties of an estimator [Coch 77] are \textit{unbiasedness} and \textit{consistency}. We say that an estimator $\hat{\theta}$ is an \textit{unbiased estimator} when the expected value of the estimator is equal to the parameter $\theta$ to be estimated (i.e., $E(\hat{\theta}) = \theta$) $.^1$ If $\hat{\theta}$ is biased, the difference $E(\hat{\theta}) - \theta$ is called the \textit{bias} of $\theta$. If $E(\hat{\theta}) > \theta$, $\hat{\theta}$ is said to be \textit{positively biased}; if $E(\hat{\theta}) < \theta$, it is said to be \textit{negatively biased}. When an estimate (i.e., $\hat{\theta}$) approaches the parameter that is to be estimated (i.e., $\theta$) as the sample size increases, it is said to be a \textit{consistent estimator} of the parameter. In other words, for a consistent estimator, when the sample consists of the whole population, the estimate becomes exactly equal to the true population value.

The \textit{mean square error} of an estimator is a frequently used concept to compare two estimators, and is defined to be $E(\hat{\theta} - \theta)^2 = Var(\hat{\theta}) + bias^2$, where $Var$ denotes the variance. The precision of any estimate computed from a sample depends not only on the estimator, but also on the plan of sampling, i.e., the sampling method, such as simple random sampling or systematic sampling [Coch 77]. \textit{Simple random sampling} is a method of selecting $m$ elements (sample size) out of $N$ (population size) such that each one of $\frac{N!}{m!(N-m)!}$ possible samples has an equal chance of being selected. A simple random sampling method according to which a selected element is removed from the population for all subsequent draws is called the \textit{random sampling without replacement} (RSWR). All the sampling methods used in this thesis employ sampling without replacement.

$^1$We use the notation $E$ for expected value to prevent any confusion with the RA expression $E$
1.5 Arrangement of the Thesis

The rest of the thesis is arranged as follows. In chapter 2, we introduce the automated query modification model, automated query modification protocols, query modification constraints, query modification strategies. In chapter 3, we introduce strategies and algorithms for enforcing time constraints. Chapter 4 introduces strategies and algorithms for enforcing error constraints. In chapter 5, we introduce new query estimation techniques for processing aggregate queries. Chapter 6 briefly summarizes strategies and algorithms for enforcing aggregate function constraints. PCF constraints and count proportionality constraints. Chapter 7 discusses the future work and concludes this thesis.
Chapter 2

Query Modification

2.1 Modified Queries

A modifiable database query is denoted as \{Q(R_1, R_2, ..., R_k), C\}, where \(Q(R_1, R_2, ..., R_k)\) is called the original (base) query as contrasted with its modified versions, \(R_i\) is the \(i^{th}\) relation involved in the query, \(C\) is a conjunction of a set of modification constraints on the query. \(Q(R_1, R_2, ..., R_k)\) is abbreviated as \(Q\) if there is no ambiguity. For a given query \{\(Q, C\)\}, if \(Q\) does not satisfy \(C\), then \(Q\) needs to be changed into a query \(Q_m\) such that \(Q_m\) satisfies \(C\). In this thesis, we assume that only the original query is to be modified, not the constraints.

From the viewpoint of query formation, modified queries are of two types, namely, the isolog type and the non-isolog type.

**Definition 2.1.** (Isologous queries) Two queries, \(Q(R_1, R_2, ..., R_k)\) and \(Q'(R'_1, R'_2, ..., R'_k)\), are said to be isologous, or, \(Q(R_1, R_2, ..., R_k)\) is an isolog of \(Q'(R'_1, R'_2, ..., R'_k)\), if (1) for \(1 \leq i \leq k\), \(R_i\) is a superset or subset of \(R'_i\) and (2) when \(R'_i\) in \(Q'\) is replaced by \(R_i\) for all \(i, 1 \leq i \leq k\), we obtain \(Q\).

Definition 2.1 implies that the parse trees \(Q(R_1, R_2, ..., R_k)\) and \(Q'(R'_1, R'_2, ..., R'_k)\) are identical, except at the leaves.

For example, in relational algebra, \(R \cap S\) and \(r \cap s\) are isologous queries if \(r\) is a subset of \(R\) and \(s\) is a superset of \(S\). However, \(R \cap S\) and \(R\) are not. In Datalog, \(engineer(x, \{PhD\}) \land earns(x, y)\) and \(engineer(x, \{PhD, Master\}) \land earns(x, y)\) are isologous queries.
If we modify a query only to its isolog, then it is an isologous query modification. This thesis discusses only isologous query modification.

2.2 Query Enlargement and Query Reduction

There are two basic strategies used in isologous query modification, namely, query enlargement and query reduction. For an arbitrary query \( Q(R_1, R_2, ..., R_k) \), query enlargement means that \( Q(R_1, R_2, ..., R_k) \) is transformed into its isolog \( Q(R'_1, R'_2, ..., R'_k) \) such that \( R_i \subseteq R'_i \) for all \( 1 \leq i \leq k \). Similarly, query reduction means that \( Q(R_1, R_2, ..., R_k) \) is transformed into its isolog \( Q(R'_1, R'_2, ..., R'_k) \) such that \( R'_i \subseteq R_i \) for all \( 1 \leq i \leq k \). One can also use both query enlargement and reduction strategies to modify the same query, i.e., a query \( Q(R_1, R_2, ..., R_k) \) is transformed into \( Q(R'_1, R'_2, ..., R'_k) \) such that \( R_i \subseteq R'_i \) for some \( 1 \leq i \leq k \) and \( R'_j \subseteq R_j \) for \( 1 \leq j \leq k \), \( j \neq i \). This is called enlargement-reduction modification.

Query enlargement may be used, for example, to satisfy the requirement that number of the output tuples of a query is not less than a certain value. Or, query enlargement may be to enforce a security constraint (e.g., the example given in section 1.), or to obtain other information related to the original query. For example, a hospital may ask for the names and addresses of at least 10 students whose blood type is “AB” by a time deadline of T. The query may be extended to involve both students and employees if the number of qualified students is not enough, and there is still time left to evaluate the query.

Query reduction may be used for processing queries with time constraints. In this case, we actually have a trade-off between the input size and the timing requirement. Enlargement-reduction modification strategy may be useful for enforcing error constraints or in exploratory data analysis; a typical example is given in example 1.1.
2.3 Automated Query Modification Model

One can think of various approaches to perform the query modification automatically. However, the algorithm for automated query modification should be efficient; and the resulting modified query should be efficiently executable.

We model the automated query modification mechanism as a quadruplet \((D, P, C, M)\), where \(D\) is a relational database instance, \(P\) is the set of query modification protocols which are followed by both the user and the database system during query modification, \(C\) is a set of modification constraints given by the user and satisfied by the modified queries, \(M\) denotes the modification methods such as automatic constraint enforcement strategies, query rewrite rules, etc.

A language independent query expression frame used in an automated query modification system is

**Query:**

\{a query written in any relational database language\}

**Constraints:**

\{a set of constraints for Query to satisfy\}

**Protocols:**

\{the protocols used for modifying Query.\}

with semantics that the Query must satisfy the Constraints. Otherwise, the Query is modified into another query following the Protocols such that the modified query satisfies the Constraints.

In the following sections, we introduce the details about modification protocols, modification constraints and modification methods.
2.4 Modification Protocols

We give two modification protocols used for automated isologous query modification, namely, use of superset/subset chains and use of sampled data.

2.4.1 Protocol One: Use of Superset/Subset Chains

A relation can be partitioned (or fragmented) into small subsets. A subset \( f_i \), \( 1 \leq i \leq n - 1 \), of a relation \( f_n \) is defined using a selection formula on attribute(s) of \( f_n \) as

\[
 f_i = \{ t | t \in f_n \land PCF(f_i) \}
\]

where \( PCF(f_i) \) is a Propositional Calculus Formula containing conjunctions of atoms \( C_j \theta_j t.A_j \theta'_j C'_j \), \( 1 \leq j \leq k \), where \( A_j \) is an attribute of \( f_n \) and called the fragmentation attribute, \( C_j \) and \( C'_j \) are constants, \( \theta_j \) and \( \theta'_j \) are arithmetic comparison operators \( =, < \) and \( \leq \). A superset/subset chain involves a set of relations \( f_1, f_2, ..., f_n \) such that \( f_i \subset f_{i+1} \), i.e., \( PCF(f_i) \land PCF(f_{i+1}) = PCF(f_i) \) and \( PCF(f_i) \lor PCF(f_{i+1}) = PCF(f_{i+1}) \). Figure 2.1.a contains the subset chain of relation Furnaces. Figure 2.1.b contains the superset chain of relation Engineer-with-PhD. Figure 2.1.c contains the subset chain of relation Employees.

**Example 2.1** Relation FURNACES in figure 2.1.a is fragmented on attributes STATUS, PRIORITY-LEVEL and ENVIRONMENT-LEVEL. For example, for \( f_3 \), \( PCF(f_3) \) is given as STATUS = "CRITICAL". And, for \( f_2 \), \( PCF(f_2) \) is STATUS = "CRITICAL" \( \land \) PRIORITY = "HIGH". It is easy to check that in this subset chain

\[1\]When an attribute \( A_j \) does not have a numeral domain (e.g., a character string domain), we may assume that a total ordering exists in the domain of \( A_j \) (e.g., lexicographic ordering), and treat operators \( < \) and \( \leq \) as the "order-testing" operators.
Let the (finite) domain of attribute \( A \) be \( D_A \). For a fragment \( f \) of \( R \) fragmented on \( A \), let \( D_f \) denote the subdomain of \( D_A \) from which the tuples of fragment \( f \) get their \( A \) values. In other words, \( D_f \) is the set of \( A \) values in \( D_A \) that make formulas \( PCF(f) \) true. For example, for \( PCF(f) \) is \( 20 \leq \text{Age} \leq 30 \), \( D_f = [20, 30] \). In \( PCF(f) \) and \( D_f \), \( f \) can also be a relation, say, \( PCF(R) \) and \( D_R \). We assume that, for \( R \) fragmented on attribute \( A \), \( D_R = D_A \).

A (superset/subset) chain exists logically (conceptually) and physically. A node \( f_i \) in a logical chain is defined and located by using a selection formula (PCF). However, for evaluation efficiency purposes, we maintain physical files \( f'_i \) for \( f_i \), where \( f'_i = f_i - f_{i-1} \) and \( f'_1 = f_1 \) in the superset/subset chain. We call \( f'_i \) the complement set of \( f_i \), or, the \( i^{th} \) physical set in the chain. Suppose relation \( R \) is physically fragmented into \( n \) physical sets \( f'_i \), \( 1 \leq i \leq n \). For each \( f_i \), \( f_i = f'_i \cup f_{i-1} \), \( 2 \leq i \leq n \), where \( \cup \) denotes disjoint union. Therefore, \( R = f'_1 \cup f'_2 \cup ... \cup f'_n \). The logical fragmentation of \( R \) is shown in figure 2.2.2.a and the corresponding physical fragmentation of \( R \) is illustrated by the Venn diagram in figure 2.2.2.b.
One relation may correspond to multiple logical superset/subset chains built on different set of attributes. However, only one of these logical chains is allowed to be physically maintained. This physically maintained superset/subset chain is called the main superset/subset chain. The others are auxiliary superset/subset chains.

In this protocol, when modifying a query, we require that the new relation used for replacing the base relation must be in the superset/subset chain of the base relation. The motivation for using superset/subset chains as query modification protocols is that the superset/subset chains provide an efficient way to find a superset or a subset of a base relation in a query to be modified, making systematic and automatic query modification possible.

2.4.2 Protocol Two: Use of Sampled Data

In some cases, a base relation may be replaced by a sample (e.g., a simple random sample) drawn from the base relation or its superset instead of the prefixed superset/subset chains. A typical example of using a sample to modify a query is in aggregate query estimation [HoOT 89]. In the following chapters, we show other examples that use sampling protocols to enforce constraints.
2.5 Modification Constraints

In this section, we introduce five types of constraints based on the modification protocols, namely, the time constraint, error constraint, aggregate function constraint, PCF constraint and count proportionality constraint. Among these, the time constraint and the aggregate function constraint can be specified independently. The error constraint, the count proportionality constraint and the PCF constraint must be specified together with at least one of the former type of independently specifiable constraints. Combinations of these constraints can be specified to express users' expectations in isologous query modification.

- Time Constraints

In many cases, particularly in real-time databases [Son 88, Ram 92], a database query is associated with a time constraint. For example, a query on a tracking data of a satellite must be processed within the given deadline: otherwise, the information provided could be of little or no value.

In our system, a time constraint is specified as \( (T = t, \alpha = a) \), where \( T \) is the time quota for processing a given query, \( \alpha \) is the maximum risk that can be taken such that the modified query overspends the given time quota \( T \). \( \alpha \) specifies an upper bound to the probability that \( Q \) can not be completed within \( t \) time. For example, \( \{Q, T = 10 \text{ seconds}, \alpha = 0.6\} \) means that \( Q_m \) (the modified version of \( Q \)) must be evaluated within 10 seconds such that the risk that \( Q_m \) can not be evaluated in 10 seconds is 0.6 or less. Both the time quota \( T \) and the risk \( \alpha \) are specified by users.

- Aggregate Function Constraints

For some queries, the user may expect some aggregate properties to hold by the output of a given query. For example, for a query \( Q \) whose output has an Age attribute, the user may specify that the average of Age values
of the output tuples is greater than or equal to 25, but less than or equal to 28.

Such an aggregate function constraint is defined as \( c_1 \theta_1 \text{Agg}(A) \theta_2 c_2 \), where \( \text{Agg}(A) \) denotes an aggregate function on the attribute \( A \) of the output of the query \( Q \) (Agg can be COUNT, SUM or AVG), \( \theta_1 \) and \( \theta_2 \in \{<, \leq, >, \geq, =\} \), \( c_1 \) and \( c_2 \) are constants. For example, in the above example, the user’s query is specified as \( \{Q, 25 \leq \text{AVG}_{Age(Q)} \leq 28\} \).

The aggregate function constraint can also be specified on an input relation in the query as well. For example, if a user’s query involves the relation EMPLOYEE(...,Salary,...), the user may specify that the tuples from relation EMPLOYEE or one of its superset/subset satisfy that the sum of the salaries of the input tuples is greater than or equal to $2M but less than or equal to $3M.

An aggregate function constraint on an input relation is defined as 
\( c_1 \theta_1 \text{Agg}_{A}(R) \theta_2 c_2 \), where \( \text{Agg}_{A}(R) \) denotes an aggregate function on the attribute \( A \) of the input relation \( R \). For example, in the above example, the user’s aggregate function constraint is expressed as

\[
$2M \leq \text{SUM}_{Salary}(\text{EMPLOYEE}) \leq $3M.
\]

Aggregate function constraints can be used in security enforcement or in exploratory data analysis. The cardinality constraint [Chau 90] is only a special case (COUNT) of the aggregate function constraint. An interesting query using both output and input aggregate function constraints is “List the names of 100 stocks such that their Price/Earnings Ratio over the last 365 days is less than 20.”.

• **Error Constraints**

A modified query is different from the original query in both the semantics and the output. The difference in the output is called the modification error (or error in short).
In terms of the expectations of the user who posed query $Q$, the query modification performed by the database system leads to two types of errors. First, some tuples that are in the original query $Q$ do not appear in the output of the modified query $Q_m$. We call this **Deleted-Information Error**, or, **D-error** for short, and characterize it as $Q - Q_m$. Second, some tuples of the modified query $Q_m$ are not among the tuples of the original query $Q$. These tuples, characterized by $Q_m - Q$, constitute **Added-Information Error**, or, **A-error** for short. We define the (relative) sizes of A- and D-errors as

\[
\text{The size of A-error} = \frac{|Q_m - Q|}{|Q_m|} \quad (f.2.1)
\]

\[
\text{The size of D-error} = \frac{|Q - Q_m|}{|Q|} \quad (f.2.2)
\]

**Example 2.2.** Consider the simple relational algebra query $Q = R - S$ and the modified query $Q_m = f_i - g_j$ where $f_i$ ($1 \leq i \leq n$) and $g_j$ ($1 \leq j \leq m$) are members in the superset/subset chains of $R$ and $S$, respectively. For query reduction, $R = f_n$ and $S = g_m$. For query enlargement, $R = f_i$ and $S = g_l$. Figures 2.3.a and 2.3.b illustrate the errors in query reduction and query enlargement using Venn diagrams, respectively.

According to the definitions of A- and D-errors, a monotone query, when modified, produces only D-error in query reduction, and only A-error in query enlargement. This is easily proved by the monotonicity of the query. However, a nonmonotone modified query may produce both A- and D-errors in both query enlargement and query reduction.

Sometimes, due to various reasons (e.g., time constraint or an impatient user), an exact answer to the original query is not necessary. Instead, the user presents an error bound together with his query. Any modified query satisfying the error constraint is acceptable to the user.
Figure 2.3: Error Characterization in Query Reduction and Enlargement of a Single Set Difference Query.

For aggregate queries, the error in a modified query is measured as the relative error using a certain statistical estimation approach. An error-constrained COUNT query evaluation method is introduced in [HoOD 91].

For non-aggregate queries, the **error constraint** is specified as (A-error $\leq c_1$, D-error $\leq c_2$), where $c_1$ and $c_2$ are constants, the user can specify the desired sizes of A- and D-errors to control the error due to query modification.

A special error constraint, namely, the **safety constraint**, is useful:

**Safety Constraint**

In some cases, the A-error in query reduction (and the D-error in query enlargement) is unacceptable. For example, consider a time-constrained query of a hospital database asking for the names of people with blood type "O". Clearly, names of people whose blood type are not "O" are unacceptable to be included in the answer.
In the cases that the A-error in query reduction and the D-error in query enlargement are unacceptable, the user requires that the output of any modified query $Q_m$ must be (i) a subset of the output of $Q$ in query reduction (no A-error); and (ii) a superset of the output of $Q$ in query enlargement (no D-error). Such a modified query is called a safe modified query. Otherwise, it is an unsafe modified query. Such constraints are called the safety constraints.

A safety constraint is specified as (A-error = 0, D-error $\leq c_1$) for query reduction and (A-error $\leq c_2$, D-error = 0) for query enlargement.

- **PCF Constraints**

When a query is being modified using a superset/subset protocol, the user may specify the semantic relationship between the replacements of two relations in the original query. For example, if the relations RESEARCHERS(Name, Field, ...) and PROFESSORS(Name, Field, ...) in a query need to be replaced by their subsets, respectively, the user may desire that the researchers and the professors in the two subsets work in the same field.

In section 3, we define the superset/subsets of a relation with propositional calculus formulas (PCF). We can also use PCFs to specify the desired semantic relationship between the replacements in a modified query. Such a constraint is called a PCF constraint.

Let $\text{replace}(R, f)$ denote the predicate that $f$ is a superset/subset of $R$ in the superset/subset chain of $R$. Three other predicates are used in a PCF Constraint, namely, $\text{PCF\_equal}$, $\text{PCF\_cover}$ and $\text{PCF\_exclude}$. $\text{PCF\_equal}(f, g)$ means $\text{PCF}(f) \equiv \text{PCF}(g)$, where $f$ and $g$ are two superset/subsets. $\text{PCF\_cover}(f, g)$ means $\text{PCF}(f) \land \text{PCF}(g) \equiv \text{PCF}(f)$. $\text{PCF\_exclude}(f, g)$ means $\text{PCF}(f) \land \text{PCF}(g) \equiv \emptyset$. For example, a user query in relational algebra using PCF constraints is
\{R \cup S, \text{replace}(R, f) \land \text{replace}(S, g) \land \text{PCF_equal}(f, g) \land \text{COUNT}(f \cup g) \geq 100\}

The function of the PCF constraint in this query is that if \(|R \cup S| < 100\), then DBMS can choose to replace \(R\) and \(S\) with their supersets \(f\) and \(g\), respectively, such that \(PCF(f) \equiv PCF(g)\) and \(|f \cup g| \geq 100\).

Furthermore, when a relation (or fragment) is fragmented on more than one of its attributes (e.g., relations RESEARCHERS and PROFESSORS may be fragmented on attributes Age and Field), we utilize multiple superset/subset chains. In this case, we need also to specify particular superset/subset chains used in a PCF constraint. This is done by specifying a particular superset/subset chain to each relation in a PCF constraint in protocol specification section.

- **Count Proportionality Constraint**

In order to infer from the output of a modified query what the output of the original query looks like, the user may require that the "tuple ingredients" in the output of the modified query are proportional to that in the output of the original query. We call such a constraint the count proportionality constraint. Let \(R\) be the output relation of a query \(Q\), and \(r\) be the output relation of a modified query \(Q_m\) of \(Q\). For a function \(F\) applied on the output of a query, we expect that the count proportionality constraint guarantees that \(F(r)\) provides an unbiased estimation or approximation of \(F(R)\). Such a function \(F\) can be, for example, the ratio of men to women in a population survey, the frequency of occurrences of the names of presidential candidates in a poll, etc. Example 2.3 below shows a case where count proportionality constraint is necessary.

**Example 2.3.** Suppose an investment decision with an urgent time deadline is to be based on the result of a query on a very large database relation. If the query can not be completely evaluated by the deadline, a subset of the relation is allowed to replace the relation. If the correctness
of the investment decision depends on the proportionality of the tuple ingredients in the output of the modified query to that in the output of the original query, a subset which does not satisfy the count proportionality constraint may lead to a wrong investment decision.

The enforcements of time constraints, error constraints, aggregate function constraints, PCF constraints and count proportionality constraints are to be discussed in chapters 3, 4 and 6.

2.6 Modification Operators and Modification Strategies

For any relational language chosen, we can incorporate two new operators, namely, enlargement operator (denoted as \( \uparrow \)) and reduction operator (denoted as \( \downarrow \)). The operands of the two operators are single relations.

If superset/subset protocols are used, \( \uparrow R \) specifies a possible transformation of the relation \( R \) into one of its subsets. Similarly, \( \downarrow R \) specifies a possible transformation of the relation \( R \) into one of its subsets. For examples, \( \uparrow \) ENGINEERS-WITH-PH.D may result in relation ENGINEERS-WITH-PH.D being replaced by its superset ENGINEERS-WITH-M.S.-OR-PH.D. Similarly, \( \downarrow \) FURNACES may result in the relation FURNACES being replaced by its subset CRITICAL-STATUS-FURNACES.

If sampling protocols are used, \( \uparrow R \) transforms relation \( R \) into a sample drawn from one of its supersets. \( \downarrow R \) transforms relation \( R \) into a sample drawn from itself.

Query enlargement, query reduction and query enlargement-reduction can be easily extended to the cases where sampling protocols are used: In such a case, for an arbitrary query \( Q(R_1, R_2, ..., R_k) \), a \( R_i \) is replaced by a sample drawn from one of its supersets or itself.

Now we give the definitions of query enlargement, query reduction and
query enlargement-reduction using enlargement and reduction operators, which are suite for both of the two modification protocols.

**Definition 5.1. (Query Enlargement)**

For each $R_i$ in $Q(R_1, ..., R_k)$, $R_i$ is modified using enlargement operator ($\uparrow R_i$) or remains the same.

**Definition 5.2. (Query Reduction)**

For each $R_i$ in $Q(R_1, ..., R_k)$, $R_i$ is modified using reduction operator ($\downarrow R_i$) or remains the same.

**Definition 5.3. (Query Enlargement-Reduction)**

For each $R_i$ in $Q(R_1, ..., R_k)$, $R_i$ is modified using enlargement operator for some $1 \leq i \leq k$; and $R_j$ is modified using reduction operator for some $1 \leq j \leq k$, $j \neq i$; the rest $R_k$, $k \neq i, j$, remains the same.

In the next section, we will present the general algorithm for enforcing modification constraints.

### 2.7 The General Algorithm for Enforcing Constraints

Enlarging or reducing a relation in a given query depends on $Q$ and the given modification constraints. We design a procedure $er(R, Q, C, S)$ which decides the modification strategy $S$ to be used and assigns a modification operator $\uparrow$ or $\downarrow$ (or null if $R$ needs no changes) to each base relation $R$ in $Q$ according to $S$ and $C$. The enlargement and reduction operators can also be assigned to relations explicitly by the user. For example, in example 1.1, the user can assign an $\uparrow$ operator to the base relation OPERATOR.WITH.X.LEVEL.SKILLS.OR_LOW and a $\downarrow$ operator to the base relation MACHINE.WITH.Y.PRECISION.OR_LOW.

Once the modification "direction" (the modification strategy) for each re-
Algorithm ENFORCING_CONSTRAINTS \((Q, C, P, Q'_m)\)

**Input:** Original query \(Q\), modification constraints \(C\) and modification protocol \(P\);

**Output:** A modified query \(Q'_m\) satisfying \(C\);

begin

\(Q'_m := Q; \{l = R_1, R_2, \ldots, R_k, \text{ the original relations in } Q.\}\)

\(T = "YES";\)

while \(satisfy(Q'_m, C) != "TRUE"\) and \(T == "YES"\) do begin

Call \(er(R, Q, C, S);\) \{to decide the modification strategy \(S.\}\)

Call \(select(l, P, S, T);\) \{to construct a replacement list \(l\) according to \(P\) and \(S; select\) also returns YES/NO in \(T.\}\)

end;

if \(T == "YES"\) return \(Q'_m; \{Q'_m\ \text{satisfies}\ \(C.\}\}

else Report "\(C\) cannot be satisfied."; \(T\) is "NO".

end.

Figure 2.4: The General Algorithm for Enforcing Modification Constraints.

Relation in \(Q\) is determined, we use procedure \(select(l, P, S, T)\) to construct a replacement list \(l\) (i.e., supersets/subsets or samples) used for replacing the base relations in \(Q\) according to \(P\) and \(S\). Then, we modify query \(Q\) into \(Q'_m\) which uses \(l\). \(select(l, P, S, T)\) returns a "YES" value in \(T\) if a replacement list is selected; otherwise, \(select(l, P, S, T)\) returns a "NO" value in \(T\).

Once a modified query \(Q'_m\) is formed, function \(satisfy(Q'_m, C)\) is triggered to test the satisfaction of \(Q'_m\) to \(C\). That is, \(satisfy(Q'_m, C)\) returns "TRUE" if \(Q_m\) satisfies \(C\); otherwise, it returns "FALSE". If "FALSE" is returned, the "select-test" loop is repeatedly performed until a "TRUE" value is returned or, a conclusion that the given query cannot be modified is made.

The general algorithm ENFORCING_CONSTRAINTS for enforcing modification constraints is provided in figure 2.4.
Chapter 3

Enforcing Time Constraints for Non-Aggregate Queries

3.1 Introduction

In a single-user DBMS, the issue of time-constraint satisfaction is equivalent to controlling the evaluation time of a query precisely. There are two points to observe. First, query evaluation times for RA queries are unknown prior to evaluation and can only be estimated with a certain probabilistic confidence. Consider, for example, selecting from a relation a set of tuples satisfying a boolean formula $F$. The number of tuples satisfying $F$ may vary significantly, and whether the selection can be completed within a given time quota cannot be known a priori. In general, the evaluation time of a query changes not only with different input data, but also with the selectivities\footnote{Selectivity of an RA operation (or an expression) $E$, denoted by $\text{sel}_E$, is the ratio of the number of output tuples of $E$ to the product of the number of tuples in the operand relations in $E$.} of the RA operators of the query. Second, a given time constraint $T$ for a query may be so small (or the query so time-consuming) that the probability that it cannot be completed within $T$ time units (hereafter, called the risk of overspending) may be extremely high. For example, the probability that the join of two disk-resident relations, each with 1,000 blocks, can be performed in 10 seconds is likely to be almost zero, and, thus the risk of overspending the time quota of 10 seconds for such a join is almost one.

One can think of various approaches to approximating or modifying a time-constrained RA query. Our approach is as follows.

1. Before the user specifies any time-constrained query, she or he is asked
to specify the subsets (fragments) of each database relation $R$ that the DBMS can use for replacing $R$ in a time-constrained query. The user can also specify additional information about these fragments in order to provide better guidance for the DBMS in the replacement.

2. With each query, in addition to the time-constraint, the user specifies the maximum risk of overspending to be taken by the DBMS in evaluating (either) the query (or one of its modified versions).

3. During query evaluation, the DBMS modifies the original query by replacing base relations with their fragments (called the fragment selection problem) such that the risk of overspending in the revised query is closest to and less than the risk specified by the user.

4. If, after evaluating the modified query, there is time left then step (3) is performed repetitively (called the iterative query evaluation) with higher risks of overspending until the time quota is completely used.

Example 3.1. In CASE-DB, each RA query has the keyword parameter "T=" which specifies the time constraint (or time quota), and the keyword parameter "R=" which specifies the risk of overspending. Now consider the database relation FURNACES (fnumber, fname, priority, status, environment) that contains information about furnaces, and the relation TEMPERATURES (fnumber, temperature, time, date) that maintains the recorded temperatures of furnaces. Assume that the user has specified the relation fragmentation chains shown in figure 3.1. Consider the query $Q =$ "List the furnace names and their temperatures in 10 seconds with the risk at 0.5 or less" which is specified in RA as

$$Q = \pi_{\text{name}, \text{temperature}}(\text{FURNACES} \bowtie \text{TEMPERATURES}) \text{ T = 10s R = 0.5}$$

CASE-DB finds the risks of evaluating the query with different combinations of fragments from the two chains. Assume that, among these risks, the risk that comes closest to and is less than 0.5 is 0.48, and it is for the query "List
the last day temperatures of high-priority-and-critical-status-furnaces" which is

\[ Q_{m_1} = \pi_{fname.temperature(HIGH-PRIORITY-AND-CRITICAL-STATUS-FURNACES \bowtie \text{ LAST-3-DAYS-TEMPERATURES})} \]

Then CASE-DB evaluates \( Q_{m_1} \). Assume that the evaluation of \( Q_{m_1} \) took 8 seconds. Then, for the remaining 2 seconds, CASE-DB chooses larger fragments from the two chains using a very high risk of overspending (e.g., 0.95) and repeats the query evaluation. The reason for choosing high risks in later iterations is to reduce the number of additional iterations, and thus to control the overhead of iterations. On the average, the number of iterations are always upper bounded by 4. □

Figure 3.1: Relation Fragmentation Chains

Figure 3.2\(^2\) presents an outline of the non-aggregate, real-time query evaluation algorithm used in CASE-DB. Please note that the major random variables that introduce an error in query evaluation time and thus cause multiple query evaluation steps are the selectivities of RA operators in the query. At the end of each query evaluation step, we have better information about operator selectivities, which is used to revise the selectivity estimations.

---

\(^2\)When the timer interrupt occurs, interrupt service routine returns the control to the statement after the while loop. Therefore, the while loop in the algorithm is an infinite loop.
Algorithm  Time-Constrained-Non-Aggregate-Query-Evaluation(Q, T, α)
input: Q: an arbitrary relational algebra query.
T: a given amount of clock time quota.
α: (upper bound for) the risk of overspending.
Output: a revised query Q_m and its response produced within T clock time units.
begin
    Initialize the estimated selectivities for operators in Q;
s := 1; \{s denotes the current step\}
Set the timer interrupt to T units:
while TRUE do begin \{iterative evaluation\}
    if s = 1 then
        Choose the fragments for the risk α:
        begin
            for each relation R in Q do
                begin
                    Replace R with the chosen fragment f;
                    Evaluate the resulting revised query Q_m;
                end;
            end;
        else
            Choose a very high risk α':
            Choose the fragments for the risk α' and Call the resulting query Q_m;
            Transform Q_m into Q_m' such that Q_m' uses the previous step's response;
            Evaluate Q_m'; \{section 3.3\}
        endif;
    endwhile;
end.

Figure 3.2: Query Evaluation Algorithm for Real-Time, Non-Aggregate Queries in CASE-DB
Please note that, in the algorithm in figure 3.2, there is a transformation of the modified query $Q_m$ into $Q'_m$ such that $Q'_m$ uses "the previous step's response". The first revision of the query $Q$ obtained by replacing each relation with its required fragment and the evaluation of the revised query constitutes the first query evaluation step. CASE-DB then spends the remaining time by iteratively improving the query with additional steps. Clearly, from step 2 onwards, the DBMS may save time if, instead of evaluating the current step's query with base relations, it can revise the current step's query such that previous step's output can be used in the current step's output. Section 3.3 discusses the transformation problem.

The motivation for our approach is that

1. there is a compromise between the sizes of operand relations of the query and the risk of overspending. Under the expected case (with the possible exception of the set difference operator of the RA), as the relations are replaced by their subsets (i.e., fragments), the query evaluation time and hence the risk of overspending get smaller.

2. By specifying the fragments of relations and how much risk (s)he is willing to take for overspending in a query, the user guides the DBMS in choosing the modified query.

3. The modified query is semantically meaningful, and represents the "best" query that the DBMS can answer for the given risk and the given time constraint.

With the exception of [SmiL 89 VrbL 91], all other real-time database literature deals with the multi-user environment and transaction management such as maximizing the number of transactions that complete within their deadlines. Smith and Liu [SmiL 89] and Vrbsky and Liu [VrbL 91] give a methodology for finding approximate answers to relational algebra queries. In
their approach, with the amount of time used increasing, the accuracy of the approximate result is improved. But their work does not contain any risk control mechanism.

The rest sections in this chapter are organized as follows. In sections 3.2, we introduce fragment selection problem. Sections 3.3 and 3.4 discuss the iterative query evaluation techniques used for processing nonperiodically occurring queries. Section 3.5 introduces more general issues about relation fragmentation. Section 3.6 introduces the experimental results and performance analysis for iterative query evaluation. Sections 3.7 and 3.8 introduce the incremental query evaluation techniques for processing periodically occurring queries.

3.2 Fragment Selection Problem

For each relation $R$, let $S_R$ denote the fragments in the subset chain of $R$, i.e., $S_R = \{ f_i | f_i \subseteq R \}$. Consider $Q$ with input relations $R_i$. For a query evaluation step of $Q$, let us say we choose the fragment $f_i$ from $(S_{R_i})$ of each relation $R_i$. We call the resulting list of fragments $l = \{ f_1, \ldots, f_n | f_i \subseteq R_i, f_i \in S_{R_i} \}$ the fragment list of $Q$.

Assume that we are at the $i$th query evaluation step. Let $l_i = \{ f_1, f_2, \ldots, f_n \}$ be a fragment list of $Q$ selected at step $i$. The risk of overspending at step $i$, denoted by $\alpha_i$, is introduced by Hou in his thesis [Hou 89].

Definition 3.1. (Fragment Selection Problem) Let $\alpha$ be the given risk of overspending at step $i$. Let $X$ denote the set $\{ l_j | l_j$ is a fragment list, $0 \leq \alpha - \alpha_j \leq \varepsilon, \varepsilon$ is a pre-chosen small constant $\}$. Choose from all fragment lists in $X$ the list $l_i$ with the risk $\alpha_i$ such that $\alpha - \alpha_i$ is minimum.

The Fragment Selection Problem has been proven to be NP-Complete by Guruswamy in her Masters thesis [Guru 92].
Thus, the complexity of finding \( F_i \) with \( \alpha_i \) among all possible fragment lists is high. We need to consider a heuristic approach to locate a qualified \( l_i \). We consider the following properties for our heuristics:

(i) Selectivity,  
(ii) types of operators involved,  
(iii) time costs of subqueries where \( R \) is involved  
(iv) file organization type. and  
(v) position of the relations in the parse tree.

We use "selectivity" because if the selectivity of an operator is high, a slight increase in the fragment size of the relation involved with the operator would drastically increase the output, thereby increasing the time cost. and might overspend the allocated time. So, we would like to increase the fragment sizes of input relations whose associated operator has a high selectivity, when we are ready to take a large risk. We use "the types of operators to determine the monotonicity property of the subquery involved. For some relations and some operators, if the fragment size is increased (decreased) then we may observe a priori an increase (decrease) in the output size, and hence in the time cost. For some relations the reverse is true: an increase (decrease) in the fragment size decreases (increases) the output size. We would like to increase the fragment size of those relations which increase the time cost (maximize) when the available time is larger.

The time-cost of the subqueries involving only base relations is expected to have smaller variance. Hence, the time-cost of a query involving few operators would have smaller variance.

The type of operator in the subquery plays a part in the time-cost of the subquery. the file organization of the relation involved in the subquery also
plays a role in determining the time-cost of the subquery. For example, in case of the operator 'selection' if there is an indexed file whose index is over the same attribute used in the selection formula, then the time-cost is much less than that of evaluating the 'selection' operator on a non-indexed file.

In stepwise optimization, we begin with a fragment list whose fragments are all "required". We use the weights of the input relations and the sizes of fragment complements to find successive fragment lists such that the associated query evaluation costs progressively increase, but yet are smaller than \( T \). This process is repeated until we cannot find a "better" fragment list or a preset time limit is reached.

3.3 Iterative Query Evaluation Transformations

We illustrate the iterative query evaluation transformations with an example.

Example 3.2. Consider the query \( Q_{m_1} \) of Example 3.1.

\[
Q_{m_1} = \pi \text{name, temperature} (\text{HIGH-PRIORITY-AND-CRITICAL-STATUS-FURNACES} \bowtie \text{LAST-3-DAYS-TEMPERATURES}) \\
= \pi \text{name, temperature}(f_2 \bowtie g_1)
\]

Assume that the evaluation of \( Q_{m_1} \) took 8 seconds, and there still are 2 seconds left in the time quota. In the second step, using the risk of 0.95, the DBMS chooses to evaluate

\[
Q'_{m_2} = \pi \text{name, temperature}(\text{CRITICAL-STATUS-FURNACES} \bowtie \text{LAST-4-DAYS-TEMPERATURES}) \\
= \pi \text{name, temperature}(f_3 \bowtie g_2)
\]

We can transform \( Q'_{m_2} \) into an equivalent query \( Q_{m_2} \) that uses \( Q_{m_1} \) as follows.
\( Q_{m_2} = Q_{m_1} \cup \pi_{fname.temperature} (\{CRITICAL-STATUS-FURNACES - HIGH-PRIORITY-AND-CRITICAL-STATUS-FURNACES\} \bowtie \text{LAST-4-DAYS-TEMPERATURES}) \cup \pi_{fname.temperature} (\{\text{CRITICAL-STATUS-FURNACES} \bowtie \text{LAST-4-DAYS-TEMPERATURES} \} - \text{LAST-3-DAYS-TEMPERATURES})) \)

\[ = Q_{m_1} \cup \pi_{fname.temperature} ((f_3 - f_2) \bowtie g_1) \cup \pi_{fname.temperature} (f_3 \bowtie (g_2 - g_1)) \]

We make two observations. First, each of the two union operators in the right hand side of equation of \( Q_{m_2} \) is a union of two disjoint sets. Therefore, there is no need for duplicate tuple elimination which leads to a very fast implementation. Second, in the implementation of relation fragmentation chains for FURNACES and TEMPERATURES, we actually maintain (physical files for) \( f'_i \) and \( g'_i \), \( i > 1 \), where \( f'_i = f_i - f_{i-1} \) and \( g'_i = g_i - g_{i-1} \) at each node. Thus, when evaluating \( (f_3 - f_2) \) and \( (g_2 - g_1) \), we substitute \( f'_3 \) and \( g'_2 \), respectively, that are already stored in the database and available; this leads to a very fast implementation of \( Q_{m_2} \). \[ \square \]

Consider a single-operator query with input relation \( R \). CASE-DB evaluates \( Q \) using fragments \( f_1, f_2, \ldots, f_n \) from \( R \) such that \( f_i \subset f_{i+1}, 1 \leq i \leq n - 1 \), iteratively.

**Example 3.3.** Consider a relation \( R \) with a subset chain for \( R \). Assume we have already evaluated \( Q(f_1) \), and there is still time left in the time quota. Let \( f_2 \) be the next fragment chosen. We then evaluate \( Q(f_2) \) in terms of \( Q(f_1) \) which in turn, is used in evaluating \( Q(f_i), 2 < i, \) and so on. \[ \square \]

The evaluation of \( Q(f_{i+1}) \) in terms of \( Q(f_i) \) is done (almost always) as follows. Through algebraic manipulations, \( Q(f_{i+1}) \) is converted into \( Q(f_i) \cup Q'(f_i, f'_i) \) such that \( f'_{i+1} = f_{i+1} - f_i \), \( f_i \subset f_{i+1} \), and \( Q'(f_i, f'_i) \cap Q(f_i) = \phi \). In other words,
1. $Q'$ uses $f_i$ and $f'_{i+1}$ in its evaluation—two relations each strictly smaller than $f_{i+1}$.

2. The union operation between $Q'$ and $Q$ is a union of two disjoint sets.

Let us denote the union of two disjoint sets by $\cup$, and call it the disjoint union.

**Definition 3.3. (Disjoint Union)** Let $R$ and $S$ be two relations. Then $R \cup S \equiv R \cup S$ where $R \cap S = \emptyset$.

Please note that disjoint union can be implemented very fast since, unlike union, it does not require duplicate tuple elimination, which is normally implemented by sorting in databases—an expensive task. Therefore, whenever possible, we use disjoint union over union. We illustrate with an example.

**Example 3.4.** Let $Q(R, S) = R \cup S$ with $R$ and $S$ having the subset chains $f_1, f_2, \ldots, f_n$ and $g_1, g_2, \ldots, g_m$, respectively. Assume, at a previous iteration, $Q(f_i, g_j) = f_i \cup g_j$ is evaluated, and, at the current step, $f_{i+1}$ and $g_{j+1}$ are chosen to evaluate $Q$, i.e., $Q(f_{i+1}, g_{j+1}) = f_{i+1} \cup g_{j+1}$ is to be evaluated. We transform $Q(f_{i+1}, g_{j+1})$ as

$$Q(f_{i+1}, g_{j+1}) = Q(f_i, g_j) \cup (g_{j+1} - f_i) \cup (f'_{i+1} - g_j)$$

where $g'_{j+1} = g_{j+1} - g_j$ and $f'_{i+1} = f_{i+1} - f_i$, and the subset chains of $R$ and $S$ contain $g'_{j+1}$ and $f'_{i+1}$ computed and stored in the database already (before the query session starts).

### 3.3.1 Transformations for Single-Operator Queries

We now generalize our approach for single-operator queries. Assume $Q(f_i, g_j)$ is evaluated before, and $Q(f_k, g_m)$, $k > i, m > j$, is to be evaluated. Let $f_i, k$ denote $f_i \cup f'_{i+1} \cup \ldots \cup f_k$ and $g_j, m$ denote $g_j \cup g'_{i+1} \cup \ldots \cup g'_{m}$. We trans-
form \( Q(f_k, g_m) \) (with few exceptions) into \( Q(f_i, g_j) \theta Q'(f_i, f_{i+1,k}, g_j, g_{j+1,m}) \) where \( \theta \) is an RA operator, and evaluate the transformed form. We now list for each single-operator query \( Q \) such transformations.

**Union:** \( Q(R, S) = R \cup S. \)

\[
\begin{align*}
Q(f_k, g_m) &= f_k \cup g_m = \\
Q(f_i, g_j) \cup (f_{i+1,k} \cup g_{j+1,m}) &= \text{t.n.u.1} \\
Q(f_i, g_j) \uplus ((f_{i+1,k} \cup g_{j+1,m}) - Q(f_i, g_j)) &= \text{t.n.u.2} \\
Q(f_i, g_j) \uplus (g_{j+1,m} - f_i) \uplus (f_{i+1,k} - g_m) &= \text{t.n.u.3} \\
((Q(f_i, g_j) - f_{i+1,k}) - g_{j+1,m}) \uplus (f_{i+1,k} - g_{j+1,m}) \uplus g_{j+1,m} &= \text{t.n.u.4}
\end{align*}
\]

**Difference:** \( Q(R, S) = R - S. \)

\[
\begin{align*}
Q(f_k, g_m) &= f_k - g_m = \\
(Q(f_i, g_j) \uplus (f_{i+1,k} - g_j)) - g_{j+1,m} &= \text{t.n.d.1} \\
((Q(f_i, g_j) \uplus f_{i+1,k}) - (f_{i+1,k} \cap g_j)) - g_{j+1,m} &= \text{t.n.d.2} \\
(Q(f_i, g_j) - g_{j+1,m} \uplus (f_{i+1,k} - g_m) &= \text{t.n.d.3}
\end{align*}
\]

**Intersection:** \( Q(R, S) = R \cap S. \)

\[
\begin{align*}
Q(f_k, g_m) &= f_k \cap g_m = \\
Q(f_i, g_j) \uplus (f_i \cap g_{j+1,m}) \uplus (f_{i+1,k} \cap g_j) \uplus (f_{i+1,k} \cap g_{j+1,m}) &= \\
Q(f_i, g_j) \uplus f_i \cap g_{j+1,m} \uplus f_{i+1,k} \cap g_m &= \text{t.n.i.1}
\end{align*}
\]

**Join:** \( Q(R, S) = R \Join S. \)

\[
\begin{align*}
Q(f_k, g_m) &= f_k \Join g_m = \\
Q(f_i, g_j) \uplus (f_i \Join g_{j+1,m}) \uplus (f_{i+1,k} \Join g_j) \uplus (f_{i+1,k} \Join g_{j+1,m}) &= \\
Q(f_i, g_j) \uplus (f_i \Join g_{j+1,m}) \uplus (f_{i+1,k} \Join g_m) &= \text{t.n.j.1}
\end{align*}
\]

**Selection:** \( Q(R) = \sigma_{\text{condition}}(R). \)
\[ Q(f_k) = \sigma_{\text{condition}}(f_k) = Q(f_i) \cup \sigma_{\text{condition}}(f_{i+1,k}) \quad (\text{t.n.s.1}) \]

**Projection:** \[ Q(R) = \pi_{\text{attributes}}(R). \]

\[ Q(f_k) = \pi_{\text{attributes}}(f_k) = Q(f_i) \cup \pi_{\text{attributes}}(f_{i+1,k}) \quad (\text{t.n.p.1}) \]

### 3.3.2 Transformations for Multiple-Operator Queries

Consider an RA query with multiple operators and its parse tree, e.g., the RA query \( Q = (R_1 \Join R_2) \cup R_3 \). At each query evaluation step, internal nodes of the parse tree are associated with (output) relation instances obtained by evaluating the operator at that node. Our approach is to use whenever possible the last instances of such relations. For monotone queries, such an approach is quite efficient.

#### 3.3.2.1 Monotone Queries

Assume that the RA expression does not have any set difference operators (i.e., a monotone query). Let \( o_i \) and \( o_{i+1} \) be the output relations of an internal node in the parse tree obtained in two consecutive query evaluation steps. We now summarize the query transformations at each node of the parse tree. Let \( E \) and \( \tilde{E} \) be arbitrary RA expressions (possibly relations) that are evaluated at the \( i^{\text{th}} \) step to give \( e_i \) and \( \tilde{e}_i \), respectively, and, at the \( (i+1)^{\text{th}} \) step to give \( e_{i+1} \) and \( \tilde{e}_{i+1} \), respectively. From the subset chains of the relations involved in \( E \) and \( \tilde{E} \), we can compute \( e'_{i+1} = e_{i+1} - e_i \) and \( \tilde{e}'_{i+1} = \tilde{e}_{i+1} - \tilde{e}_i \). Clearly, for \( E \theta \tilde{E} \) (or \( \theta(E) \) in unary operator case), where \( \theta \in \{ \cup, \cap, \Join, \pi, \sigma \} \), \( o_i = e_i \theta \tilde{e}_i \) (or \( o_i = \theta(e_i) \)). Let \( o_i, e_i, \tilde{e}_i, e_{i+1}, \tilde{e}_{i+1}, e'_{i+1} \) and \( \tilde{e}'_{i+1} \) correspond to \( Q(f_i, g_j) \) (or by \( Q(f_i) \)), \( f_i, g_j, f_k, g_m, f_{i+1,k} \) and \( g_{j+1,m} \), respectively. Then we use exactly the same transformations given in section 3.3.2 for evaluating \( e_{i+1} \theta \tilde{e}_{i+1} \) (or...
\(\theta(e_{i+1})\). The transformations for the query \(Q = E\theta E\) or \(Q = \theta E\) (only one for each query type) are given below, where \(E\) and \(\tilde{E}\) are arbitrary monotone RA expressions. \(\theta \in \{\cup, \cap, \bowtie, \pi, \sigma\}\).

**Union**: \(e_{i+1} \cup e_{i+1}\)

\[a_{i+1} = a_i \cup (e'_{i+1} - e_i) \cup (e'_{i+1} - e_i)\]

**Intersection** \(e_{i+1} \cap e_{i+1}\)

\[a_{i+1} = a_i \cap (e_i \cap e'_{i+1}) \cap (e'_{i+1} \cap e_{i+1})\]

**Join** \(e_{i+1} \bowtie e_{i+1}\)

\[a_{i+1} = a_i \bowtie (e_i \bowtie e'_{i+1}) \bowtie (e'_{i+1} \bowtie e_{i+1})\]

**Selection** \(\sigma_{condition}(e_{i+1})\)

\[a_{i+1} = a_i \cup \sigma_{condition}(e'_{i+1})\]

**Projection** \(\pi_{attributes}(e_{i+1})\)

\[a_{i+1} = a_i \cup \pi_{attributes}(e'_{i+1})\]

### 3.3.2.2 Nonmonotone Queries

Whenever a set difference operator appears in the parse tree (i.e., a non-monotone query), we may have \(a_i - a_{i+1} \neq \phi\) where \(a_i\) and \(a_{i+1}\) are two consecutive output relations of the set difference operator. This results in complicated transformations if we are to use \(a_i\) in the computation of \(a_{i+1}\), thus making the iterative evaluation too costly. Note that, in \(R - S\), the consecutive evaluations of \(f_1 - g_1, f_2 - g_1, f_3 - g_1, \ldots\) etc., do create monotonously increasing output relations \(a_1, a_2, \ldots, a_i, a_{i+1}, \ldots\) such that \(a_i \subset a_{i+1}\). Thus, our approach
in CASE-DB for any subexpression $E - \bar{E}$ in the query is to evaluate $E - \bar{E}$ once, and afterwards, to evaluate $E - \bar{E}$ with new fragments only in $E$ (but not in $\bar{E}$). Such an approach guarantees that consecutive output relations $o_i$ and $o_{i+1}$ of any set difference operator satisfy $o_i \subseteq o_{i+1}$. In this case, to compute $o_{i+1} = e_{i+1} - \bar{e}_{i+1}$ we use the transformation

$$o_i \cup (e'_{i+1} - \bar{e}_i)$$

(t.n.d.4)

3.4 Query Evaluation Algorithms for Iterative Evaluation

At each iterative query evaluation step, CASE-DB selects the proper fragment for each relation involved in the given query; transforms the query into the form which uses the previously obtained (last step) result; and, then evaluates the transformed query. In this section, we introduce the algorithms for query transformation and evaluation.

3.4.1 Algorithms and costs for query transformations

The condition for a transformation to be used is determined by comparing the cost formulas of all transformations for an operator if it has more than one transformation. Once a transformation is determined, the query evaluation algorithms used are rather straightforward and identical to query evaluation algorithms of standard DBMSs with the exception that we use main-memory-based operations as much as possible.

3.4.1.1 File Organization and Notations

We use sequential files sorted on the key to store the fragments of a relation. As a notation, for file $F$, $\| F \|$ denotes the number of disk blocks occupied by $F$: $| F |$ denotes the number of records of $F$. $C_d(Q)$ denotes the number of
disk accesses and $C_c(Q)$ denotes the number of comparisons for processing the query $Q$. Below we derive upper bounds for $C_d$ formulas. However, since the cost of a comparison is much smaller than the cost of a disk access, we will only derive big-Oh, i.e., $O(f(x))$, notation upper bounds for $C_c$ formulas.

We have two ways to maintain the intermediate results obtained during an iterative evaluation step: in main memory until the evaluation is over or on the disk. Since we use the iterative evaluation method to process the fragments of a relation, intermediate results are repeatedly used in each iterative step. Therefore, we keep intermediate results in main memory. The final results obtained from each iterative step are kept on the disk.

3.4.1.2 Basic Algorithms

In a subset chain, $R = f'_1 \cup f'_2 \ldots \cup f'_m$. Therefore, we store as separate files $f'_i$, $i=1, 2, \ldots, n$, where each of $f'_i$ is held in a separate file. We have a fragment selection and risk control mechanism (FSRCM) as discussed in section 3.2 which estimates using the heuristic approach how large a fragment can be evaluated in the given time quota and selects a fragment from each input relation for query evaluation. Below we give the basic algorithms to be used in the transformations.

Algorithm Union1($f_{i,k}, g_{j,m}$)
{$f_{i,k} \cup g_{j,m}$. The result is held in the main memory.}

Begin

s := i; t := j;

Empty A; Empty B; \{A and B are two arrays.\}

flagF := 'Load'; flagG := 'Load'; \{flagF and flagG are two flags.\}

while $s \leq k$ and $t \leq m$ do

begin

if flagF = 'Load' then

begin Load $f_i$ to $A$: $x := 1$: end:
if flagG = 'Load' then
    begin Load $g'_i$ to $B$: $y := 1$: end:
while $A[x] \neq \phi$ and $B[y] \neq \phi$ do
    begin
      if $A[x] < B[y]$ then
        begin Output $A[x]$: $x := x + 1$: end
      else if $A[x] = B[y]$ then
        begin Output $A[x]$: $x := x + 1$: $y := y + 1$: end
      else begin Output $B[y]$: $y := y + 1$: end
    end
if $A[x] = \phi$ then
    begin Empty $A$: flagF := 'Load': $s := s + 1$: end
else flagF := ' ': 
if $B[x] = \phi$ then
    begin Empty $B$: flagG := 'Load': $t := t + 1$: end
else flagG := ' ': 
if $s \leq k$ then Load and output the rest $f'_i$;
else Load and output the rest $g'_i$;
End.

$C_d(\text{Union1}) = O(||f_{i,k}|| + ||g_{j,m}||)$; $C_c(\text{Union1}) = O(|f_{i,k}| + |g_{j,m}|)$, where $|f_{i,k}| = \sum_{s=1}^{k} |f'_s|$ and $|g_{j,m}| = \sum_{s=j}^{m} |g'_s|$.

Similar to algorithm Union1, we have algorithm Union2($A$, $g_{j,m}$) for $A \cup g_{j,m}$, where $A$ is an array residing in the main memory and $g_{j,m} = g'_j \cup g'_{j+1} \cup \ldots \cup g'_m$. $C_d(\text{Union2}) = O(||g_{j,m}||)$. $C_c(\text{Union2}) = O(|A| + |g_{j,m}|)$.

Algorithm Difference1($f_{i,k}$, $g_{j,m}$)

\{ $f_{i,k} - g_{j,m}$. The result is held in the main memory.\}

Begin
s := i; t := j;
Empty A: Empty B: \{A and B are two arrays.\}
flagF := 'Load'; flagG := 'Load'; \{flagF and flagG are two flags.\}
while \(s \leq k\) and \(t \leq m\) do
begin
if \(flagF = 'Load'\) then
begin Load \(f'_s\) to A; \(x := 1\); end:
if \(flagG = 'Load'\) then
begin Load \(g'_t\) to B; \(y := 1\); end:
while \(A[x] \neq \phi\) and \(B[y] \neq \phi\) do
begin
if \(A[x] < B[y]\) then
begin Output \(A[x]\); \(x := x + 1\); end
else if \(A[x] = B[y]\) then
begin \(x := x + 1\); \(y := y + 1\); end
else \(y := y + 1\); \{\(A[x] > B[y]\}\}
endwhile:
if \(A[x] = \phi\) then
begin Empty A; flagF := 'Load'; \(s := s + 1\); end
else flagF := "."
if \(B[x] = \phi\) then
begin Empty B; flagG := 'Load'; \(t := t + 1\); end
else flagG := "."
endwhile:
if \(s \leq k\) then Load and output the rest \(f'_s\);
End.

\[ C_d(\text{Difference}1) = O(\|f_{i,k}\| + \|g_{j,m}\|); \]
\[ C_c(\text{Difference}1) = O(|f_{i,k}| + |g_{j,m}|). \]

Similar to algorithm Difference1, we have algorithm Difference2(A, \(g_{j,m}\))
for \(A - g_{j,m}\), where A is an array residing in the main memory and \(g_{j,m} = \]
\(g'_1 \cup g'_{j+1} \ldots \cup g'_m\). \(C_d(\text{Difference}2) = O(||g_{j,m}||). \ C_c(\text{Difference}2) = O(|A| + |g_{j,m}|)\).

**Algorithm** \(\text{Intersection1}(f_{i,k}, g_{j,m})\)

\(\{f_{i,k} \cap g_{j,m}\}. \ \text{The result is held in the main memory.}\)

**Begin**

\[s := i; \ t := j;\]

\(\text{Empty A: Empty B:}\) \quad \{A and B are two arrays.\}

\(\text{flagF := 'Load': flagG := 'Load':}\) \quad \{flagF and flagG are two flags.\}

\[\text{while } s \leq k \text{ and } t \leq m \text{ do}\]

**begin**

\[\text{if flagF = 'Load' then}\]

**begin**

Load \(f'_i\) to A; \(x := 1;\) \text{ end};

\[\text{if flagG = 'Load' then}\]

**begin**

Load \(g'_i\) to B; \(y := 1;\) \text{ end};

\[\text{while } A[x] \neq \phi \text{ and } B[y] \neq \phi \text{ do}\]

**begin**

\[\text{if } A[x] < B[y] \text{ then } x := x + 1\]

\[\text{else if } A[x] = B[y] \text{ then}\]

**begin**

Output \(A[x]; x := x + 1; y := y + 1;\) \text{ end}

**else** \(y := y + 1;\) \quad \{A[x] > B[y]\}

**endwhile;**

**if** \(A[x] = \phi\) **then**

**begin**

Empty A; flagF := 'Load'; \(s := s + 1;\) \text{ end}

**else** flagF := ' ';

**if** \(B[x] = \phi\) **then**

**begin**

Empty B; flagG := 'Load'; \(t := t + 1;\) \text{ end}

**else** flagG := ' ';

**endwhile;**

**End.**

\(C_d(\text{Intersection1}) = O(||f_{i,k}|| + ||g_{j,m}||); \ C_c(\text{Intersection1}) = O(|f_{i,k}| + |g_{j,m}|)\).
Algorithm Disjoint_Ution(D, A_1, A_2, ..., A_m)
{D denotes a disk-resident file. A_i is an array in the main memory.
Disjoint_Ution adds A_i (i = 1, 2, ..., m) to D, i.e., D U A_1 U ... U A_m.}
Begin
for i = 1 to m do
    add A_i without checking duality to D;
End.

C_d(Disjoint_Ution) = \sum_{i=1}^{m} ||A_i||; C_e(Disjoint_Ution) = 0.

Algorithm Join(f_{i,k}, g_{j,m})
{f_{i,k} \bowtie g_{j,m}.}
Begin
    Load f_{i,k};
    Load g_{j,m};
    Sort f_{i,k} on the join attributes;
    Sort g_{j,m} on the join attributes;
    Output f_{i,k} \bowtie g_{j,m};
End.

It is difficult to give the expected time complexity for the join operation since we do not know how much common information that the two input files have on the join attributes. In the worst case, the number of comparisons or the number of tuples of the results for f_{i,k} \bowtie g_j is |f_{i,k}| \times |g_j|. But, such a case is perhaps not common. We assume that the size of the output relation is proportional to the sizes of the input files. Therefore,

C_d(Join) = O(||f_{i,k}|| + ||g_{j,m}||)
C_e(Join) = O(||f_{i,k}|| \log_2 |f_{i,k}| + ||g_{j,m}|| \log_2 |g_{j,m}|).

Algorithm Selection(f_{i,k}, Condition)
\{ \sigma_{Condition}(f_{i,k}) \} 

Begin
for s = i to k do
    begin
        Load \(f'_s\);
        Output the tuples of \(f'_s\) meeting Condition to file F;
    end;
End.

\( C_d(Selection) = O(||f_{i,k}||) \). \( C_e(Selection) = O(|f_{i,k}|) \).

\textbf{Algorithm} \textit{Projection}(f_{i,k}, \textit{Attributes})
\{ \pi_{Attributes}(f_{i,k}) \} 
T: a balanced binary search tree initialized with empty:

Begin
for s = i to k do
    begin
        Load \(f'_s\);
        for each tuple t in \(f'_s\)
            if \(t.Attribute\) is in T then
                Insert \(t.Attribute\) into T:
        Output T;
    end:
End.

\( C_d(Projection) = O(||f_{i,k}||) \). \( C_e(Projection) = O(|f_{i,k}| \log_2 |f_{i,k}|) \).

3.4.1.3 Time-Costs for the Transformations

Using the above algorithms and their time costs, we compute the time costs for the transformations given in section 3.3.2. For union and set difference which have more than one transformation, we choose the transformation
with the smallest time cost. Since the query is evaluated in an iterative fashion, we only compute the costs of the transformations in a certain iteration step. In what follows, let Evaluate($Q(f_k, g_m)$) (or Evaluate($Q(f_k)$)) denote a procedure evaluating query $Q(f_k, g_m)$ (or $Q(f_k)$) with $Q(f_i, g_j)$ (or $Q(f_i)$) obtained in the previous step. Procedure Evaluate contains the algorithms introduced above. The execution order of procedure Evaluate is strictly sequential.

**Union:** $Q(R, S) = R \cup S$

For the four equivalent transformations for union listed in section 3.4.2. transformation t.n.u.3, $Q(f_i, g_j) \cup (g_{j+1,m} - f_i) \cup (f_{i+1,k} - g_m)$, has the smallest time costs. Transformation t.n.u.3 is implemented as

\[
\text{Evaluate}(Q(f_i, g_j) \cup (g_{j+1,m} - f_i) \cup (f_{i+1,k} - g_m))
\]

**Begin**

\[
\text{Difference1}(g_{j+1,m} - f_i);
\]
\[
\text{Disjoint_Union}(Q(f_i, g_j), g_{j+1,m} - f_i);
\]
\[
\text{Difference1}(f_{i+1,k} - g_m);
\]
\[
\text{Disjoint_Union}(Q(f_i, g_j) \cup (g_{j+1,m} - f_i), (f_{i+1,k} - g_m));
\]

**End.**

with time costs

\[
C_d(\text{t.n.d.3}) = ||f_{i+1,k}|| + ||g_m||;
\]
\[
C_c(\text{t.n.d.3}) = O(||f_k|| + ||g_m||).
\]

**Set Difference:** $Q(R, S) = R - S$

For the three equivalent transformations for union listed in section 3.4.2. transformation t.n.d.3, $(Q(f_i, g_j) - g_{j+1,m}) \cup (f_{i+1,k} - g_m)$, has the smallest time costs. Transformation t.n.d.3 is implemented as
Evaluate\((Q(f_i, g_j) - g_{j+1,m}) \cup (f_{i+1,k} - g_m)\)

Begin

\begin{align*}
\text{Difference} & 1(f_{i+1,k}, g_m); \\
\text{Load} & \ Q(f_i, g_j); \\
\text{Difference} & 2(Q(f_i, g_j), g_{j+1,m}); \\
\text{Save} & \ Q(f_i, g_j) - g_{j+1,m} \text{ in the disk;}
\end{align*}

\text{Disjoint} \cup \text{Union}(Q(f_i, g_j) - g_{j+1,m}, (f_{i+1,k} - g_m));

End.

with time costs

\begin{align*}
C_d(t.n.d.3) &= \|Q(f_i, g_j)\| + \|f_{i+1,k}\| + \|g_m\|; \\
C_c(t.n.d.3) &= O(|Q(f_i, g_j)| + |f_{i+1,k}| + |g_m|).
\end{align*}

**Intersection:** \(Q(R, S) = R \cap S\)

The only transformation for \(f_k \cap g_m\) is \(Q(f_i, g_j) \cup f_i \cap g_{j+1,m} \cup f_{i+1,k} \cap g_m\) (t.n.i.1). Transformation t.n.i.1 is implemented as

Evaluate\((Q(f_i, g_j) \cup f_i \cap g_{j+1,m} \cup f_{i+1,k} \cap g_m)\)

Begin

\begin{align*}
\text{Intersection} & \ 1(f_i, g_{j+1,m}); \\
\text{Intersection} & \ 1(f_{i+1,k}, g_m); \\
\text{Disjoint} \cup \text{Union}(Q(f_i, g_j), f_i \cap g_{j+1,m}, f_{i+1,k} \cap g_m); \\
\end{align*}

End.

with time costs

\begin{align*}
C_d(t.n.i.1) &= \|f_k\| + \|g_m\|; \\
C_c(t.n.i.1) &= O(|f_k| + |g_m|).
\end{align*}

**Join:** \(Q(R, S) = R \bowtie S\)

\(f_k \bowtie g_m\) is transformed into \(Q(f_i, g_j) \cup (f_i \bowtie g_{j+1,m}) \cup (f_{i+1,k} \bowtie g_j) \cup (f_{i+1,k} \bowtie g_m)\).
\( g_{j+1,m} \) (t.n.j.1). It is implemented as

\[
\text{Evaluate}(Q(f_i, g_j) \bowtie (f_i \bowtie g_{j+1,m}) \bowtie (f_{i+1,k} \bowtie g_j) \bowtie (f_{i+1,k} \bowtie g_{j+1,m}))
\]

\text{Begin}

\begin{align*}
&\text{Join}(f_i, g_{j+1,m}); \\
&\text{Join}(f_{i+1,k}, g_j); \\
&\text{Join}(f_{i+1,k}, g_{j+1,m}); \\
&\text{Disjoint Union}(Q(f_i, g_j), f_i \bowtie g_{j+1,m}, f_{i+1,k} \bowtie g_j, f_{i+1,k} \bowtie g_{j+1,m});
\end{align*}

\text{End.}

with time costs

\[
C_d(\text{t.n.j.1}) = \|f_k\| + \|g_m\|;
\]

\[
C_c(\text{t.n.j.1}) = O(|f_i|\log_2|f_i| + |f_{i+1,k}|\log_2|f_{i+1,k}| + |g_j|\log_2|g_j| + |g_{j+1,m}|\log_2|g_{j+1,m}|).
\]

**Selection:** \( Q(R) = \sigma_{\text{condition}}(R) \)

Using algorithm Selection\((f_{i+1,k}, \text{Condition})\), Transformation t.n.s.1 is implemented as

\[
\text{Evaluate}(Q(f_i) \bowtie \sigma_{\text{condition}}(f_{i+1,k}))
\]

\text{Begin}

\begin{align*}
&\text{Selection}(f_{i+1,k}, \text{Condition}); \\
&\text{Disjoint Union}(Q(f_i), \sigma_{\text{condition}}(f_{i+1,k}));
\end{align*}

\text{End.}

with time costs

\[
C_c(\text{t.n.s.1}) = \|f_{i+1,k}\|;
\]

\[
C_d(\text{t.n.s.1}) = O(|f_{i+1,k}|).
\]

**Projection:** \( Q(R) = \pi_{\text{attributes}}(R) \)

Using algorithm Projection\((f_{i+1,k}, \text{Attributes})\), Transformation t.n.p.1 is implemented as
Evaluate($Q(f_i) \cup \pi_{\text{Attributes}}(f_{i+1,k})$)

Begin

Projection($f_{i+1,k}$, Attributes);
Load $Q(f_i)$;
Union2($Q(f_i)$, $\pi_{\text{Attributes}}(f_{i+1,k})$);
Save the result on the disk;

End.

with time costs

$C_d(\text{t.n.p.1}) = \|Q(f_i)\| + \|f_{i+1,k}\|;$
$C_c(\text{t.n.p.1}) = O(|Q(f_i)| + |f_{i+1,k}| \log_2 |f_{i+1,k}|).$

We must perform the union of $Q(f_i)$ and $\pi_{\text{attributes}}(f_{i+1,k})$ since the intersection of $Q(f_i)$ and $\pi_{\text{attributes}}(f_{i+1,k})$ may be nonempty.

3.4.2 Using Indexed Files

To further increase the (time) effectiveness of transformations, we can use indexed files for storing fragments of a relation because some RA operations such as union, intersection and set difference benefit heavily from using indexes. For example, we check the intersection of two files through their indexes. There are two parts for an indexed file, the main file which holds records (tuples) and the index file which holds the key value and a pointer to each record of the main file. The index files are of fast-access data structures such as B-tree [Ullm 88]. The main file is large, and is kept in secondary storage. The index files reside in main memory during a query evaluation session.

Using indexed files can greatly speed the evaluations for union, intersection
Algorithm Intersection2($f_{i,k}$, $g_{j+1,m}$)
{$i, k \cap g_{j,m}$. The result is held in the main memory.}
Begin
for s = i to k do
    for each key value v in the index of $f_s'$ do
        if v is in the index of a $g_t'$, $t = j, j+1, ..., m$ then
            Output $< v, p >$ {p points to the tuple with the key value v.}
End.

Figure 3.3: Algorithm for $f_{i,k} \cap g_{j,m}$ Using Indexed Files.

and set difference\(^3\). To see this, please see the algorithm Intersection2 shown in Figure 3.3.

In algorithm Intersection2, $f_{i,k} = f_i' \cup f_{i+1}' \cup \ldots \cup f_k'$ and $g_{j,m} = f_j' \cup f_{j+1}' \cup \ldots \cup g_m'$.

We store as separate files $f_s'$, s=i, i+1, ..., k, (and $g_t'$, t=j, j+1, ..., m) where each $f_i'$ ($g_t'$) is held in a separate indexed file. All such files have identical index structures. The tuples of the output relation are identified by the pairs with the format $< v, p >$, where $p$ points to the output tuple with the key value $v$.

Since we assume that the index files reside in the main memory during the query evaluation (actually, they are loaded in the first query evaluation step), we do not consider the time for loading index files into the main memory in algorithm Intersection2. Therefore, the time costs of Intersection2 are

$$C_d(\text{Intersection2}) = 0;$$

$$C_c(\text{Intersection2}) = O(|f_{i,k}| \times I_{g_{j,m}}).$$

where $I_{g_{j,m}}$ denotes the expected number of comparisons for searching a record through the index of $g_{j,m}$. $I_{g_{j,m}} = \sum_{s=j}^{m} I_{g_s'}$.

Clearly, algorithm Intersection2 has better time cost than algorithm Intersection1 (without index), a very important factor in real-time systems.

\(^3\) Usually, natural join, selection and projection do not benefit from using indices because the key attributes are usually not contained in the join attributes, selection condition or project attributes.
Algorithm Union3($f_{i,k}, g_{j,m}$)
{\{f_{i,k} \cup g_{j,m}. The result is held in array A with format <v,p>\}}
Begin
for s = i to k do
  for each <v,p> in the index of $f_s$ do
    if v is not in the index of every $g_t', t = j, j+1, ..., m$, then
      Save <v,p> in array A;
  for t = j+1 to m do save all <v,p> of in the index of $g_t'$ in array A:
End.

Figure 3.4: Algorithm for $f_{i,k} \cup g_{j,m}$ Using Indexed Files.

Algorithm Difference2($f_{i,k}, g_{j,m}$)
{\{f_{i,k} - g_{j,m}. The result is held in array A with format <v,p>\}}
Begin
for s = i to k do
  for each <v,p> in the index of $f_s$ do
    if v is not in the index of every $g_t', t = j, j+1, ..., m$ then
      Save array A in the disk;
End.

Figure 3.5: Algorithm for $f_{i,k} - g_{j,m}$ Using Indexed Files.

Similarly, algorithms Union3 shown in figure 3.4 and Difference3 shown in figure 3.5 perform union and set difference operations on indexed files, respectively. Algorithms Union3 and Difference3 have the same time costs as algorithm Intersection2.

### 3.4.3 Transformation Optimization

We have designed the transformations for each RA query type in iterative evaluation. All of these transformations are designed for general cases. These transformations can be further optimized if more information about operand fragments is given. Particularly, for the binary RA operations union, intersection, set difference and natural join\(^4\) operations, if the two operand relations are fragmented on the same attribute, we can simplify the corresponding transformations by checking the intersection of the two operand fragments.

\(^4\)If the join attributes include the fragmentation attribute
The intersection of two fragments can be detected by comparing their propositional calculus formulas on the fragmentation attribute.

**Lemma 3.1.** For two fragments $f_i$ and $g_j$, if $PCF(f_i) \land PCF(g_j) = \phi$, then

\[
\begin{align*}
&f_i \cap g_j = \phi; \quad \text{(o.1)} \\
&f_i \cup g_j = f_i \cup g_j; \quad \text{(o.2)} \\
&f_i - g_j = f_i; \quad \text{(o.3)} \\
&f_i \triangleright g_j = \phi. \quad \text{(o.4)}
\end{align*}
\]

**Proof.** Straightforward. Omitted. Q.E.D.

In run-time, we optimize the transformations by using lemma 3.1.

**Example 3.5.** Consider the query $f_k \theta g_m$, $\theta \in \{\cup, \cap, -, \triangleright\}$ with $f_i \theta g_j$ evaluated at last iteration. If $PCF(f_i) \land PCF(g_j) = \phi$, then

transformation t.n.u.1, $Q(f_i, g_j) \cup (f_{i+1,k} \cup (g_{j+1,m})$, can be simplified to

\[Q(f_i, g_j) \cup ((f_{i+1,k} \cup (g_{j+1,m})); \quad \text{by (o.2)}\]

transformation t.n.u.3, $Q(f_i, g_j) \cup (g_{j+1,m} - f_i) \cup (f_{i+1,k} - g_m)$ can be simplified to

\[Q(f_i, g_j) \cup g_{j+1,m} \cup (f_{i+1,k} - g_{j+1,m}); \quad \text{by (o.3)}\]

transformation t.n.d.1, $(Q(f_i, g_j) \cup (f_{i+1,k} - g_j)) - g_{j+1,m}$, and transformation t.n.d.3, $(Q(f_i, g_j) - g_{j+1,m}) \cup (f_{i+1,k} - g_m)$, can be simplified to

\[Q(f_i, g_j) \cup (f_{i+1,k} - g_{j+1,m}); \quad \text{by (o.3)}\]

transformation t.n.i.1, $Q(f_i, g_j) \cup f_i \cap g_{j+1,m} \cup f_{i+1,k} \cap g_m$, can be simplified to

\[Q(f_i, g_j) \cup f_{i+1,k} \cap g_{j+1,m}; \quad \text{by (o.1)}\]
transformation t.n.j.1, \( Q(f_i, g_j) \uplus (f_i \Join g_{j+1,m}) \uplus (f_{i+1,k} \Join g_m) \), can be simplified to

\[
Q(f_i, g_j) \uplus (f_{i+1,k} \Join g_{j+1,m}). \tag{by (0.4)}
\]

\[\square\]

3.5 Other Issues About Fragmenting Relations

General speaking, we have several ways to specify the inclusion relationships of the fragments of a relation. Such inclusion relationships correspond to different fragmentation graphs. The superset/subset chain is one type of fragmentation graphs. A fragmentation graph basically consists of vertices (nodes) and arcs. In a fragmentation graph, each node represents a fragment; each arc represents a logical inclusion relationship of the two directly connected fragments. In some cases, the nodes of a fragmentation graph are assigned weights to indicate the priorities that the fragments are chosen for query evaluation. Such fragmentation graphs are called weighted fragmentation graphs.

A fragmentation graph of a relation \( R \) is a lattice. That is, any two nodes in the graph have a greatest lower bound (GLB) and a least upper bound (LUB). The source node is the relation itself. The sink node is the fragment which must be included by any (modified) query involving \( R \). The simplest lattice structure is a chain.

3.5.1 Diamond-Structured Lattices

In addition to chain-structured lattices, another commonly-used fragmentation graph type is diamond-structured lattice shown in figure 3.6.
\[ f_i \subseteq R, \quad i = 1, 2, ..., n-1 \]
\[ f_i \subseteq f_i, \quad i = 1, 2, ..., n \]
\[ f_i \cap f_j = f_i, \quad i \neq j, \quad i, j = 1, 2, ..., n-1 \]

Figure 3.6: Diamond-Structured Fragmentation Lattice.

In a diamond-structured lattice, \( R = f_n, f_1 \subseteq R, f_1 = f'_1, f'_i = f_i - f_1, \quad i = 2, 3, ..., n-1 \), and \( f'_n = f_n - f_2 - ... - f_{n-1} \). Therefore, \( R = f_n = f'_1 \cup f'_2 \cup ... \cup f'_n \). For a relation \( R \) having a diamond-structured lattice \( L_R \) with \( f'_i \)'s as its fragments, in evaluating a query involving \( R \), we always first consider \( f_1 \). Then, if there is still time left, in iterative step, we pose the same query upon one or more \( f_i \)'s and merge the result with the one obtained in the previous step. This procedure continues until the time quota is used up.

If a diamond-structured lattice is not a weighted one, the order that fragments \( (f_i) \) are incorporated in the evaluation is determined by DBMS. Without losing generality, we assume that the order in which \( f_i \)'s are selected by DBMS from \( L_R \) is \( f_1, f_2, f_3, ..., f_n \). Let \( F_i = f'_1 \cup f'_2 \cup ... \cup f'_i \) and \( F_{i,k} = f'_i \cup f'_{i+1} \cup ... \cup f'_k \) \((i, k = 1, 2, 3, ..., n; i < k)\). We define a mapping \( M \) to map a diamond-structured lattice to a chain-structured lattice.

\[ M: \quad f_i \text{ in chain-structured lattice} \leftarrow F_i \text{ in diamond-structured lattice}, \quad i = 1, 2, ..., n \]
Based on this mapping, it is easy to see that one can use the same query transformations and algorithms designed for queries using chain-structured lattices in the cases where diamond-structured lattices are used.

3.5.2 Arbitrary Relation Fragmentation Lattices and Stratification

One can use more complex relation fragmentation lattices for a better representation of subset relationship among fragments. We give an example.

Example 3.6. Figure 3.7 shows a stratified relation fragmentation lattice for the relation FURNACES. Notice that the lattice is divided into three fragment groups, namely, required, strongly preferred, and preferred groups which are used as follows. When the user asks a query with a time constraint and involving FURNACES, the user (a) prefers that the query is evaluated with FURNACES. (b) strongly prefers that the query is evaluated with one of the six fragments in the strongly-preferred group, and (c) absolutely requires that the query is evaluated with at least the fragment HIGH-PRIORITY-AND-CRITICAL-STATUS-AND-DANGEROUS-ENVIRONMENT-FURNACES. It is the responsibility of DBMS to ensure that (a), (b) and (c) are satisfied.

CASE-DB allows the specification of stratified lattices as in figure 3.7. And the very first query step of a query evaluation in CASE-DB is the one that evaluates the given query with required fragments of each operand relation. Furthermore, nodes in a single stratum of a relation fragmentation lattice can also be assigned weights to provide more general model. In what follows, we introduce an algorithm for converting an arbitrarily stratified relation fragmentation lattice with varying weights of nodes in each stratum into a chain-structured lattice.
Figure 3.7: Relation Fragmentation Lattice of the FURNACES Relation.

Consider the arbitrary relation fragmentation lattice \( L \) shown in figure 3.8.a. We stratify \( L \), denoted as vertical priority, and assign weights to nodes, denoted as horizontal priority. The horizontal priorities are assigned in level order in the lattice, and are sequential.

In an arbitrary relation fragmentation lattice \( L \) with the highest vertical priority \( n \), a node with vertical priority \( i \) and horizontal priority \( j \) is denoted as \( f_{i,j} \). \( f_{1,1} \) is the sink node. \( f_{n,1} \) is the source node. Please see figure 3.8. \( f_{i,j} \) has a higher priority than \( f_{k,l} \) if \( i > k \), or \( i = k \) but \( j > l \). Let \( f'_{i,j} \) be the complement fragment of \( f_{i,j} \). We have \( f_{1,1} = f'_{1,1} \). For an arbitrary node \( f_{i,j} \) \((i > 1)\) having \( m \) descendants \( f_{x,y} \) in \( L \) (i.e., there is a path from \( f_{i,j} \) to \( f_{x,y} \), \( i > x \)), we define \( f'_{i,j} \) by

\[
f'_{i,j} = f_{i,j} - (\cup_{x,y} f_{x,y}) \tag{f.3.1}
\]

For any two different nodes \( f_{i,j} \) and \( f_{k,l} \) in \( L \), we require

\[
f'_{i,j} \cap f'_{k,l} = \phi \tag{f.3.2}
\]
Figure 3.8: Conversion from an Arbitrary Relation Fragmentation Lattice to a Chain.
Algorithm LATTICE\_CONVERSION(L)

\textit{Input:} An arbitrary relation fragmentation lattice \(L\).

\textit{Output:} A chain-structured relation fragmentation lattice.

\begin{verbatim}
begin
  \(k := 1;\)
  \(F_k := f_{i,1};\)
  for \(i = 2\) to \(n\) do \(\{\text{\(m\) is the largest horizontal priority in level \(i\).}\}\)
    begin
      \(k := k + 1;\)
      \(F'_k := f'_{i,j};\)
      \(F_k := F_{k-1} \cup F'_k;\)
    end;
  \end{verbatim}

Return the constructed chain-structured lattice \(\{F_1, \ldots, F_N\};\) \(\{N\) is the total number of nodes in \(L\}.\)

end.

Figure 3.9: Algorithm for Converting an Arbitrary Relation Fragmentation Lattice into a Chain-Structured Relation Fragmentation Lattice.

Therefore, for any node \(f_{i,j}\) in \(L\),

\[ f_{i,j} = (\cup f_{x,y}) \cup f'_{i,j} \]  \hspace{1cm} (f.3.3)

where \(f_{x,y}\) is a descendant of \(f_{i,j}\). Now, we can convert an arbitrary weighted relation fragmentation lattice \(L\) into a chain-structured relation fragmentation lattice by visiting every node of \(L\) in the priority order and converting the visited nodes into a chain. The algorithm for converting \(L\) into a chain-structured relation fragmentation lattice is given in figure 3.9.

Lemma 3.2. Algorithm LATTICE\_CONVERSION converts any arbitrary relation fragmentation lattice into a chain-structured relation fragmentation lattice with each node in the chain being a superset of any of its descendants.

Proof. Suppose \(F_{k-1} = \cup f_{x,y} \cup f'_{x,y}.\) \(F_k\) includes one more node than \(F_{k-1}\), say, \(f_{i,j}\). By f.3.2, \(f'_{i,j}\) has no intersection with any \(f_{x,y}'\) included in \(F_{k-1}\). Therefore, we can define \(F_k\) as \(F_{k-1} \cup F'_{k} (F'_k = f'_{i,j}).\) This leads to \(F_{k-1} \subset F_k\), for any
\( k > 1 \). Therefore, the output of algorithm LATTICE\_CONVERSION is a
chain-structured relation fragmentation lattice. Q.E.D.

Figure 3.8.b is the chain-structured relation fragmentation lattice converted
from the relation fragmentation lattice in figure 3.8.a. The dotted line in
figure 3.8.a indicates the processing order during the conversion. For
\( F_i = (\exists_{x,y} f_{x,y}') \cup f_{i,j}' \), \( PCF(F_i) = (\forall_{x,y} PCF(f_{x,y}')) \cup PCF(f_{i,j}') \). For example, in
figure 3.8. \( F_4 = f_{1,1}' \cup f_{2,1}' \cup f_{2,2}' \cup f_{3,1}' \). If \( PCF(f_{1,1}') = 1 \leq A \leq 10 \), \( PCF(f_{2,1}') = 11 \leq A \leq 15 \), \( PCF(f_{2,2}') = 16 \leq A \leq 20 \), and \( PCF(f_{3,1}') = 21 \leq A \leq 30 \), then \( PCF(F_4) = PCF(f_{1,1}') \cup PCF(f_{2,1}') \cup PCF(f_{2,2}') \cup PCF(f_{3,1}') = 1 \leq A \leq 30 \).

3.6 Experimental Results for Iterative Query Evaluation

The iterative query modification technique has been implemented by Gu-
ruswamy [Guru 92]. The experimental results and performance analysis are
presented in her Master thesis [Guru 92].

3.7 Incremental Query Evaluation Transformations

In some applications of databases, there are queries that are repeatedly
evaluated at fixed intervals. For example, the periodical queries may be used
in a periodical test system for an automated production line, in the stock
information tracing systems, and in inventory shortage forecast systems. etc.

For periodically occurring queries, CASE-DB uses incremental query
evaluation techniques so that the previous response to the query is stored
and used in the next response. To express the fact that the query is periodical,
we append the keyword "P=" to the query. We give two examples.

**Example 3.7.** Assume that CASE-DB is asked to evaluate the query \( Q = \) “at
every hour, list the temperatures of all high priority furnaces in 2 seconds”.
Therefore, the query in relational algebra is

$$\pi_{name, temperature} \left( \text{HIGH-PRIORITY-FURNACES} \bowtie \text{TEMPERATURES} \right)$$

$$T = 2s \quad P = 1hr$$

Assume that the most recent response to $Q$ is available and stored in $q$, and within the last hour, the new tuples of temperatures are also stored in the relation $\Delta \text{TEMPERATURES}$. Then, the present temperatures of all high priority furnaces can be obtained by first evaluating

$$\pi_{name, temperature} \left( \text{HIGH-PRIORITY-FURNACES} \bowtie \Delta \text{TEMPERATURES} \right)$$

Then one can obtain the current result by modifying those tuples of $q$ which have new temperature values.

\[ \square \]

**Example 3.8.** Assume, for every company stock, we keep the change of stock prices (as a percentage) in the last minute in attribute $\text{SP\_Change}$ of relation $\text{C\_Info}$. Consider the query "list at the end of every minute those companies with $\text{SP\_Change}$ greater than 1% and their stock values in ten seconds" which is

$$Q: \pi_{\text{Company, Stocks, SP\_Change}}(\sigma_{\text{SP\_Change} > 1\%} (\text{C\_Info} \bowtie \text{Stocks}) \quad T = 10s \quad P = 60s$$

CASE-DB evaluates $Q$ once, and stores the response into, say, $q$. It then determines every minute which relations have additions and deletions. Assume relation $\text{C\_Info}$ has both additions and deletions. Thus, every minute, the new tuples of $\text{C\_Info}$ are stored in $\Delta \text{C\_Info}$, the increment relation of $\text{C\_Info}$; and the tuples of $\text{C\_Info}$ that are to be deleted are stored into $\nabla \text{C\_Info}$, the decrement relation of $\text{C\_Info}$. Then $Q$ is revised and evaluated as

$$Q_m: q \quad \pi_{\text{Company, Stocks, SP\_Change}} (\sigma_{\text{SP\_Change} \leq 1\%} (\nabla \text{C\_Info} \bowtie \text{Stocks}) \cup$$

$$\pi_{\text{Company, Stocks, SP\_Change}}(\sigma_{\text{SP\_Change} > 1\%} (\Delta \text{C\_Info}) \bowtie \text{Stocks} \quad T = 10s \quad P = 60s$$

Note that $Q_m$ can possibly be evaluated in 10 seconds while $Q$ most proba-
bly can not be evaluated in 10 seconds. \(\square\) Consider a query \(Q\) with the relation \(R\) and the most recent response \(Q(R)\). Assume that, after the response \(Q(R)\), new tuples for \(R\) are stored into \(\Delta R\), the increment relation, and the tuples to be deleted from \(R\) are stored into \(\nabla R\), the decrement relation. One can translate the new query \(Q((R - \nabla R) \cup \Delta R)\) into \(Q(R)\) and \(Q'((R, \Delta R, \nabla R)\) such that the new translated form permits a faster and better time-controlled query evaluation.

We use the previous response \(Q(R)\) such that the transformed form of \(Q((R - \nabla R) \cup \Delta R)\) involves either unary relational algebra operators operating on \(\Delta R\) and \(\nabla R\), or binary operators one operand of which is \(\Delta R\) or \(\nabla R\). i.e., relations much smaller than \(R\). Also, during the translation, we avoid as far as possible, the relational algebra operators in the order of importance. join, projection (since it requires sorting to eliminate duplicates), union (which also requires sorting), and difference (since it is a nonmonotone operator).

It is reasonable to assume that \(R \cap \Delta R = \emptyset, \nabla R \cap \Delta R = \emptyset, R \cap \nabla R = \nabla R, |\Delta R| \ll |R|\) and \(|\nabla R| \ll |R|\). since one can always revise (at a cost, however) \(\Delta R\) and \(\nabla R\) such that these equalities hold. Thus, the query to be evaluated is \(Q((R - \nabla R) \uplus \Delta R)\).

### 3.7.1 Transformations for Insertion-Only Single-Operator Queries

In the "insertion-only" case, \(\nabla R = \emptyset\) and \(\Delta R \neq \emptyset\) for any operand relation. Consider the following correspondences: Let relations \(R, S, \Delta R\) and \(\nabla R\) correspond to \(f_i, g_j, f_{i+1,k}\) and \(g_{j+1,m}\) discussed in section 3.3, respectively (this implies that \(R \uplus \Delta R\) and \(S \uplus \Delta S\) correspond to \(f_k\) and \(g_m\), respectively). Clearly, in the "insertion-only" case, the transformations are already specified in section 3.3. Here, we only simply list the corresponding transformations for insertion-only, single-operator, periodically occurring queries.
Union: $Q((R \uplus \Delta R), (S \uplus \Delta S)) = (R \uplus \Delta R) \cup (S \uplus \Delta S)$.

$$(R \uplus \Delta R) \cup (S \uplus \Delta S) =$$

$Q(R, S) \cup (\Delta R) \cup (\Delta S) =$ \hspace{1cm} (t.i.u.1)

$Q(R, S) \uplus ((\Delta R \cup \Delta S) - Q(R, S)) =$ \hspace{1cm} (t.i.u.2)

$Q(R, S) \uplus (\Delta S - R) \uplus (\Delta R - (S \uplus \Delta S)) =$ \hspace{1cm} (t.i.u.3)

$((Q(R, S) - \Delta R) - \Delta S) \uplus (\Delta R - \Delta S) \uplus \Delta S$ \hspace{1cm} (t.i.u.4)

Difference: $Q((R \uplus \Delta R), (S \uplus \Delta S)) = (R \uplus \Delta R) - (S \uplus \Delta S)$.

$$(R \uplus \Delta R) - (S \uplus \Delta S) =$$

$Q(R, S) \uplus (\Delta R - S) - \Delta S =$ \hspace{1cm} (t.i.d.1)

$((Q(R, S) \uplus \Delta R) - (\Delta R \cap S)) - \Delta S =$ \hspace{1cm} (t.i.d.2)

$(Q(R, S) - \Delta S) \uplus (\Delta R - (S \uplus \Delta S)) =$ \hspace{1cm} (t.i.d.3)

Intersection: $Q((R \uplus \Delta R), (S \uplus \Delta S)) = (R \uplus \Delta R) \cap (S \uplus \Delta S)$.

$$(R \uplus \Delta R) \cap (S \uplus \Delta S) =$$

$Q(R, S) \uplus (R \cap \Delta S) \uplus (\Delta R \cap S) \uplus (\Delta R \cap \Delta S) =$

$Q(R, S) \uplus R \cap \Delta S \uplus \Delta R \cap (S \uplus \Delta S)$ \hspace{1cm} (t.i.i.1)

Join: $Q((R \uplus \Delta R), (S \uplus \Delta S)) = (R \uplus \Delta R) \Join (S \uplus \Delta S)$.

$$(R \uplus \Delta R) \Join (S \uplus \Delta S) =$$

$Q(R, S) \uplus (R \Join \Delta S) \uplus (\Delta R \Join S) \uplus (\Delta R \Join \Delta S) =$

$Q(R, S) \uplus (R \Join \Delta S) \uplus (\Delta R \Join (S \uplus \Delta S))$ \hspace{1cm} (t.i.j.1)

Selection: $Q((R \uplus \Delta R)) = \sigma_{\text{condition}}(R \uplus \Delta R)$.

$$\sigma_{\text{condition}}((R \uplus \Delta R)) = Q(R) \uplus \sigma_{\text{condition}}(\Delta R)$$ \hspace{1cm} (t.i.s.1)

Projection: $Q((R \uplus \Delta R)) = \pi_{\text{attributes}}(R \forall R)$

$$\pi_{\text{attributes}}((R \uplus \Delta R)) = Q(R) \cup \pi_{\text{attributes}}(\Delta R)$$ \hspace{1cm} (t.i.p.1)
3.7.2 Transformations for Deletion-Only Single-Operator Queries

In the “insertion-only” case, $\Delta R = \phi$ and $\nabla R \neq \phi$ for any operand relation. Below we list the possible transformation(s) for each query type.

Union: $Q(R - \nabla S - \nabla S) = (R - \nabla R) \cup (S - \nabla S)$.

\[
(R - \nabla R) \cup (S - \nabla S) = \\
Q(R.S) - (\nabla R - S) - (\nabla S - R) - (\nabla R \cap \nabla S) \tag{t.d.u.1}
\]

Difference: $Q(R - \nabla R, S - \nabla S) = (R - \nabla R) - (S - \nabla S)$.

\[
(R - \nabla R) - (S - \nabla S) = \\
(Q(R, S) - \nabla R) \uplus ((\nabla S - \nabla R) \cap R) \tag{t.d.d.1}
\]

\[
(Q(R, S) \uplus (R \cap \nabla S)) - \nabla R \tag{t.d.d.2}
\]

Intersection: $Q(R - \nabla R, S - \nabla S) = (R - \nabla R) \cap (S - \nabla S)$.

\[
(R - \nabla R) \cap (S - \nabla S) = \\
Q(R.S) - \nabla R \tag{t.d.i.1}
\]

Join: $Q(R - \nabla R, S - \nabla S) = (R - \nabla R) \Join (S - \nabla S)$.

\[
(R - \nabla R) \Join (S - \nabla S) = \\
(Q(R, S) - (R \Join \nabla S)) - (\nabla R \Join S) \tag{t.d.j.1}
\]

Selection: $Q(R - \nabla R) = \sigma_{condition}(R - \nabla R)$.

\[
\sigma_{condition}((R - \nabla R)) = Q(R) - \sigma_{condition}(\nabla R) \tag{t.d.s.1}
\]

Projection: $Q(R - \nabla R) = \pi_{attributes}(R - \nabla R)$

Consider $Q = \pi_{attributes}(R)$. Converting $Q(R - \nabla R) = \pi_{attributes}(R - \nabla R)$
into a combination of $Q(R)$ and $Q'(R, \nabla R)$ presents a problem since the projection does not distribute over difference operation. Our approach is to revise projection and difference operations such that duplicate counts (i.e., frequencies) of tuples after a projection are retained. Let us define $\tilde{\pi}$ to be a projection operation that retains the frequency of each projected tuple. For example, for the relation $R$ instance $\{(a,b), (a,c), (d,e)\}$ with the attributes $\{A,B\}$, $\tilde{\pi}_A(R)$ gives $\{(a,2), (d,1)\}$ where the second component of each tuple contains the frequency information. Let us also define $\tilde{\sim}$ as the difference operation acting on frequencies: If the frequency of a tuple in the subtrahend relation is greater than or equal to frequency of the same tuple in the minuend relation then we delete the tuple from the output. Otherwise, we keep the tuple in the minuend relation in the output. For example,

$$\{(a.2), (b.3)\} - \{(a.2), (b.1), (c.3)\} = \{(b.2)\}$$

We transform $\pi_{\text{attributes}}(R - \nabla R)$ into

$$\tilde{\pi}_{\text{attributes}}(R) \tilde{\sim} \tilde{\pi}_{\text{attributes}}(\nabla R)$$ (t.d.p.1)

3.7.3 Transformations for Single-Operator Queries Containing Increments and Decrements

Now, let's consider the case where $\Delta R \neq \phi$ and $\nabla R \neq \phi$, i.e., there are tuples which have been either deleted from or added into the relation $R$ since last use of $R$. In such a case, we assume that $\Delta R \cap \nabla R = \phi$. Even though we can do this when processing $R$, $\Delta R$ and $\nabla R$, the more efficient way is to keep this feature ($\Delta R \cap \nabla R = \phi$). whenever we perform deletion and addition on $R$. We define the deletion and addition operations for this purpose.
Deletion

To delete a tuple from $R$, we put the tuple into $\nabla R$ only if the tuple is not in $\Delta R$. If the tuple is in $\Delta R$, then we delete it from $\Delta R$, not from $R$.

Addition

To add a tuple into $R$, we put the tuple into $\Delta R$ only if the tuple is not in $\nabla R$. If the tuple is in $\nabla R$, then we delete it from $\nabla R$.

Both of the two definitions are under the assumption that $R \cap \Delta R = \emptyset$ and $R \cap \nabla R = \nabla R$. Figure 3.10 shows the Venn diagrams for a single relation $R$ with $\Delta R$ and $\nabla R$. Figure 3.11 shows the Venn diagrams for two overlapped relations $R$ with $\Delta R$ and $\nabla R$ and $S$ with $\Delta S$ and $\nabla S$.

The simplified transformations for single-operator queries containing increments and decrements are given below.

**Union:** $Q((R - \nabla R) \uplus \Delta R, (S - \nabla S) \uplus \Delta S) = ((R - \nabla R) \uplus \Delta R) \uplus ((S - \nabla S) \uplus \Delta S)$

$$\begin{align*}
((R - \nabla R) \uplus \Delta R) \uplus ((S - \nabla S) \uplus \Delta S) &= \\
(Q(R, S) - (\nabla R \cap \nabla S) - (\nabla R - (\nabla S \uplus S)) - (\nabla S - (\nabla R \uplus R)) \\
(\nabla R - S) - (\nabla S - R) - (\nabla R \cap \nabla S))
\end{align*}$$
Figure 3.11: Venn Diagram for Two Overlapped Relations R with ΔR and ∇R and S with ΔS and ∇S.

ψ(ΔR - S) ∪ ((∇S - R) - ΔR) (t.id.u.1)

**Difference:** Q(((R - ∇R) ∪ ΔR. (S - ∇S) ∪ ΔS) = ((R - ∇R) ∪ ΔR) - ((S - ∇S) ∪ ΔS)

((R - ∇R) ∪ ΔR) - ((S - ∇S) ∪ ΔS) =

(((Q(R, S) ∪ (R ∩ ∇S)) - ∇R) ∪ (ΔR - S)) - ΔS (t.id.d.1)

**Intersection:** Q(((R - ∇R) ∪ ΔR. (S - ∇S) ∪ ΔS) = ((R - ∇R) ∪ ΔR) ∩ ((S - ∇S) ∪ ΔS)

((R - ∇R) ∪ ΔR) ∩ ((S - ∇S) ∪ ΔS) =

(Q(R, S) ∪ R ∩ ΔS ∪ S ∩ ΔR) ∪ ΔR ∩ ΔS) - ∇R - ∇S (t.id.i.1)

**Join:** Q(((R - ∇R) ∪ ΔR. (S - ∇S) ∪ ΔS) = ((R - ∇R) ∪ ΔR) ⋈ ((S - ∇S) ∪ ΔS)

(R - ∇R ∪ ΔR) ⋈ (S - ∇S ∪ ΔS) =

(Q(R, S) ∪ R ⋈ ΔS ∪ S ⋈ ΔR)

ψΔR ⋈ ΔS) - R ⋈ ∇S - ∇R ⋈ S (t.id.j.1)
Selection: \( Q((R - \nabla R) \uplus \Delta R) = \sigma_{\text{condition}}((R - \nabla R) \uplus \Delta R) \)

\[
\sigma_{\text{condition}}((R - \nabla R) \uplus \Delta R) = (Q(R) - \sigma_{\text{condition}}(\nabla R)) \uplus \sigma_{\text{condition}}(\Delta R)
\]  
(t.id.s.1)

Projection: \( Q((R - \nabla R) \uplus \Delta R) = \pi_{\text{attributes}}((R - \nabla R) \uplus \Delta R) \)

\[
\pi_{\text{attributes}}((R - \nabla R) \uplus \Delta R) = (\hat{\pi}_{\text{attributes}}(R) - \hat{\pi}_{\text{attributes}}(\nabla R)) \cup \pi_{\text{attributes}}(\Delta R)
\]  
(t.id.p.1)

3.8 Algorithm for Processing Arbitrary, Periodically Occurring Queries

In this section, we give the algorithm for processing arbitrary multi-operator periodically occurring queries. We assume that \( \Delta R \neq \phi \) and \( \nabla R \neq \phi \) for every relation \( R \) involved in a query.

Lemma 3.3. For an arbitrary query \( Q \), \( Q = (q - \nabla q) \uplus \Delta q \), where \( q \) is the response obtained from last evaluation of \( Q \), \( \nabla q \) is the set of tuples which are in \( q \) but not in \( Q \), \( \Delta q \) is the set of tuples which are in \( Q \) but not in \( q \).

Proof. Straightforward. Omitted. Q.E.D.

\( q \) is obtained by fully evaluating \( Q \) at the first time. We need to investigate the way to find \( \nabla q \) and \( \Delta q \).

Lemma 3.4. For arbitrary relations \( R \) and \( S \) with corresponding \( \Delta R \), \( \nabla R \), \( \Delta S \) and \( \nabla S \),

(a) Union. For \( Q((R - \nabla R) \uplus \Delta R),(S - \nabla S) \uplus \Delta S) = ((R - \nabla R) \uplus \Delta R) \uplus ((S - \nabla S) \uplus \Delta S), \)

\[
\nabla q = \nabla R \cap \nabla S \cup \nabla R - (\Delta S \uplus S) \cup \nabla S - (\Delta R \uplus \nabla) \]  
(d.id.u.d)

\[
\Delta q = (\Delta R - S) \uplus ((\nabla S - R) - \Delta R) \]  
(d.id.u.i)
(b) **Difference.** For $Q((R - \nabla R) \cup \Delta R), (S - \nabla S) \cup \Delta S) = ((R - \nabla R) \cup \Delta R) - ((S - \nabla S) \cup \Delta S),$

\[
\nabla q = \nabla R - S \cup \nabla S \cap (R - \nabla R) \quad \text{(d.id.d.d)}
\]
\[
\Delta q = \Delta R - (\nabla S \cup S) \cup (\nabla S - (R - \nabla R)) \quad \text{(d.id.d.i)}
\]

(c) **Intersection.** For $Q((R - \nabla R) \cup \Delta R), (S - \nabla S) \cup \Delta S) = ((R - \nabla R) \cup \Delta R) \cap ((S - \nabla S) \cup \Delta S),$

\[
\nabla q = \nabla R \cap S \cup R \cap \nabla S \quad \text{(d.id.i.d)}
\]
\[
\Delta q = \Delta R \cap \Delta S \cup \Delta R \cap (S - \nabla S) \cup \Delta S \cap (R - \nabla R) \quad \text{(d.id.i.i)}
\]

(d) **Join.** For $Q((R - \nabla R) \cup \Delta R), (S - \nabla S) \cup \Delta S) = ((R - \nabla R) \cup \Delta R) \Join ((S - \nabla S) \cup \Delta S),$

\[
\nabla q = \nabla R \Join S \cup R \Join \nabla S \quad \text{(d.id.j.d)}
\]
\[
\Delta q = \Delta R \Join \Delta S \cup \Delta R \Join (S - \nabla S) \cup \Delta S \Join (R - \nabla R) \quad \text{(d.id.j.i)}
\]

(e) **Selection.** For $Q((R - \nabla R) \cup \Delta R) = \sigma_{\text{condition}}((R - \nabla R) \cup \Delta R),$

\[
\nabla q = \sigma_{\text{condition}}(\nabla R) \quad \text{(d.id.s.d)}
\]
\[
\Delta q = \sigma_{\text{condition}}(\Delta R) \quad \text{(d.id.s.i)}
\]

(f) **Projection.** For $Q((R - \nabla R) \cup \Delta R) = \pi_{\text{attributes}}((R - \nabla R) \cup \Delta R),$

\[
\nabla q = \hat{\pi}_{\text{attributes}}(\nabla R) \quad \text{(d.id.p.d)}
\]
\[
\Delta q = \hat{\pi}_{\text{attributes}}(\Delta R) \quad \text{(d.id.p.i)}
\]

**Proof.** Straightforward. Omitted. Q.E.D.

To derive $\nabla q$ and $\Delta q$ for an arbitrary query $Q$, we repeatedly apply the corresponding $\nabla q$ and $\Delta q$ derivation rules given in lemma 3.2 on the subexpressions of $Q$ from internal to external, until $\nabla q$ and $\Delta q$ are obtained.
Example 3.9. Consider query \( Q = (R \cap S) - T \) where \( R, S \) and \( T \) are base relations with the corresponding increment and decrement relations \( \nabla R, \nabla S, \nabla T, \Delta R, \Delta S \) and \( \Delta T \). Let \( Q_1 = R \cap S, q_1 \) be the prior response of \( Q_1, \nabla q_1 \) and \( \Delta q_1 \) be the increment and decrement of \( q_1 \). \( \nabla q \) and \( \Delta q \) are derived from the following steps.

\[
\nabla q_1 = \nabla R \cap S \cup R \cap \nabla S \quad \{\text{d.id.i.d}\}
\]

\[
\Delta q_1 = \Delta R \cap \Delta S \cup \Delta R \cap (S - \nabla S) \cup \Delta S \cap (R - \nabla R) \quad \{\text{d.id.i.i}\}
\]

\[
\nabla q = \nabla q_1 - T \cup \nabla T \cap (q_1 - \nabla q_1) \quad \{\text{d.id.d.d}\}
\]

\[
\Delta q = \Delta q_1 - (\nabla T \cup T) \cup (\nabla T - (q_1 - \nabla q_1)) \quad \{\text{d.id.d.i}\}
\]

\( \Box \)

When \( \nabla q_1 \) and \( \Delta q_1 \) are obtained, in order to improve evaluation efficiency, the RA expression \( Q = (q - \nabla q) \cup \Delta q \), may need to be simplified before being evaluated.

The general algorithm for processing arbitrary (multi-operator) periodically occurring queries is given in figure 3.12.

Different from using iterative query evaluation techniques to process non-periodically occurring queries, in which the semantics of the original queries may be modified to meet the time constraints, using incremental query evaluation techniques keeps the semantics of the original queries unchanged and makes the given queries be completely evaluated. Incremental query evaluation requires that the current responses (including the intermediate relations generated during query evaluation) of the given queries be kept. For complex queries, incremental query evaluation may need large storage space, and hence, is only suitable for evaluating periodically occurring queries.
Algorithm Periodical-Query-Evaluation($Q, P_Q$)

Input: $Q$: an arbitrary relational algebra query.  
$P_Q$: specifies the periodicity of $Q$.

Output: the response of $Q$.

begin
  if $P_Q = 1$ then \{ $Q$ is a new query. \}
  begin
    Evaluate $Q$;
    Save the response of $Q$ in $q$;
    Save the intermediate relations generated during evaluating $Q$;
    $P_Q := 2$; \{ $P_Q > 1$ means $Q$ has been evaluated at least once. \}
    Save $P_Q$ into the dictionary;
    Return the response to $Q$;
  end;
  else \{ $Q$ is not a new query. \}
  begin
    Derive $\Delta q$;
    Derive $\nabla q$;
    Simplify $Q = (q - \nabla q) \cup \Delta q$;
    Evaluate $Q = (q - \nabla q) \cup \Delta q$;
    Save the current response of $Q$ in $q$;
    Save the intermediate relations generated during evaluating $Q$;
    Return the response to $Q$;
  end;
end.

Figure 3.12: Incremental Query Evaluation Algorithm for Time-Constrained, Periodically occurring Queries.
Chapter 4

Enforcing Error Constraints

4.1 Introduction

The error constraint enforcement issues for aggregate queries have been discussed in [HoOD 91]; in this chapter, we only discuss the error constraint enforcement issues for non-aggregate queries in relational algebra (RA) queries.

The error constraints are enforced by two techniques, namely, the error removal and the error estimation. Error removal eliminates the A-error in query reduction and the D-error in query enlargement. A semantic method using the PCFs of the supersets/subsets involved in a modified query is used in error removal. When errors of certain type are chosen not to be eliminated, error estimation is used to estimate the sizes of errors. New sampling statistical techniques and estimators are designed to estimate the error in query modification.

We make two assumptions.

**Assumption 4.1.** For $R\theta S$, where $\theta \in \{\cup, \cap, -, \mathcal{M}\}$, $R$ and $S$ are fragmented on the same attribute.

**Assumption 4.2.** For $R\theta S$, where $\theta \in \{\cup, \cap, -, \mathcal{M}\}$ and $R$ and $S$ are fragmented on the same attribute, we have $R = f'_1 \cup f'_2 \cup \ldots \cup f'_n$ and $S = g'_1 \cup g'_2 \cup \ldots \cup g'_n$ with $PCF(f_i) = PCF(g_i)$, i.e., $D_{f_i} = D_{g_i}$, $1 \leq i \leq n$. 

70
4.2 Error Removal Technique

A modified query is $A_{\text{safe}}$ if it does not have any $A$-errors. Similarly, a modified query is $D_{\text{safe}}$ if it does not have any $D$-errors. $A$- and $D$-errors in a modified query can be eliminated by checking the PCFs of the supersets/subsets.

A query is monotone when adding tuples to its input relations does not make it lose any of its output tuples: otherwise, it is nonmonotone. For example, RA queries with unions, intersections, projections, selection and joins are monotone [Ullman 88]. However, the inclusion of the set difference operator makes an RA query nonmonotone as shown in the lemma below.

Lemma 4.1. Consider a monotone query $Q$ with superset/subset chains for each relation $R_i$ in $Q$. For any modified query $Q_m$, there is no $A$-error in Query Reduction and there is no $D$-error in Query Enlargement.

Proof. Considering a monotone query $Q$ and its modified version $Q_m$ in Query Reduction, we have $Q_m \subseteq Q$. To prove this, let's construct $Q$ by adding tuples one by one into input relations. Clearly, such a construction can be arranged to first form $Q_m$ and then form $Q$. If $Q_m \not\subseteq Q$, there must be a tuple $t$ of $Q_m$ which is not in $Q$. Since $Q_m$ finally becomes $Q$, therefore, there must be a tuple $t'$ of a certain input relation $R$ whose addition into $R$ removes $t$ from $Q_m$. This contradicts the monotonicity of $Q$. $Q_m \subseteq Q$ leads to $Q_m - Q = \emptyset$, which means that there is no $A$-error, i.e., $\frac{|Q_m - Q|}{|Q_m|} = 0$.

Using a similar way, we can prove that there is no $D$-error in Query Enlargement. Q.E.D.

Thus, we only need to perform error removal for nonmonotone queries, i.e..
RA queries that contain set difference\(^1\).

Let us first discuss the error removal for a simple RA query \(R - S\).

**Definition 4.1. (Safe Replacement Pair)** Given \(Q = R - S\) and \(Q_m = f_i - g_j\),

(a) for query reduction, \(Q_m\) produces no A-error if \(f_i - g_j \subseteq R - S\). Then, we say that \(< f_i, g_j >\) is an **addition-safe pair** (denoted as \(a\_safe\) pair);

(b) for query enlargement, \(Q_m\) produces no D-error if \(R - S \subseteq f_i - g_j\). Then, we say that \(< f_i, g_j >\) is a **deletion-safe pair** (denoted as \(d\_safe\) pair).

We now characterize errors of both types in terms of formulas defining superset/subset chains.

**Lemma 4.2.** Given \(Q = R - S\), \(Q_m = f_i - g_j\),

(a) there is no A-error in query reduction, i.e., \(< f_i, g_j >\) is an \(a\_safe\) pair, if

\[
PCF(f_i) \land PCF(g_j) = PCF(f_i).
\]

(b) there is no D-error in query enlargement, i.e., \(< f_i, g_j >\) is a \(d\_safe\) pair, if

\[
PCF(R) \land PCF(S) = PCF(R) \land PCF(g_j).
\]

**Proof.** Let us first prove (a). Suppose that \(PCF(f_i) \land PCF(g_j) = PCF(f_i)\), but \(< f_i, g_j >\) is not an \(a\_safe\) pair. According to definition 4.1, there exists a

---

\(^1\)The error removal and error estimation techniques used in RA can also be applied to the relational calculus (RC) and (stratified) Datalog (with negation and no recursion). In Datalog, a nonmonotone query contains at least one negated subgoals. For example, \(truecousin(X,Y) : -cousin(X,Y) \& -sibling(X,Y)\). The same query can also be expressed in Relational Calculus and Relational Algebra as \(XY \mid cousin(X,Y) \land -sibling(X,Y)\) and \(truecousin = cousin - sibling\), respectively.
tuplet \( t \) of \( f_i - g_j \) which is not in \( R - S \). Since both \( R \) and \( S \) are fragmented on attribute \( A \), and the tuples in \( R \) satisfying \( PCF(f_i) \) must be in \( f_i \) and the tuples in \( S \) satisfying \( PCF(g_j) \) must be in \( g_j \), then \( t \) being in \( f_i \) but not in \( g_j \) implies that \( t.A \) satisfies \( PCF(f_i) \), but not \( PCF(g_j) \), which leads to \( PCF(f_i) \land PCF(g_j) \neq PCF(f_i) \). Contradiction to the hypothesis.

According to assumption 4.1 and 4.2 and definition 4.1.b. it is straightforward. Q.E.D.

Lemma 4.3. If \( < f_i, g_j > \) is an a_safe pair, then for any \( x \leq i \), \( < f_x, g_j > \) is an a_safe pair.

Proof. If \( < f_i, g_j > \) is an a_safe pair, then \( f_i \subseteq g_j \). By the definition of fragmentation chain, \( f_x \subseteq f_i \). Therefore, \( f_x \subseteq g_j \). Q.E.D.

Lemma 4.4. Consider \( R - S \) and query enlargement. Suppose relations \( R \) and \( S \) satisfy \( PCF(R) \land PCF(S) = PCF(R) \). For any \( f_i \) and \( g_j \) chosen from the superset chains of \( R \) and \( S \), respectively, \( f_i - g_j \) produces no D-errors.

Proof. According to lemma 4.2, \( f_i - g_j \) produces no D-errors if \( PCF(R) \land PCF(S) = PCF(R) \land PCF(g_j) \). Since \( g_j \) is a superset of \( S \), we have \( PCF(R) \land PCF(S) = PCF(R) \land PCF(g_j) \) Q.E.D.

To summarize, when choosing \( f_i \) and \( g_j \) for query enlargement, if zero D-error is desired then, by lemma 4.4, we should choose \( g_j \) such that \( PCF(R) \land PCF(S) = PCF(R) \land PCF(g_j) \). If \( PCF(R) \land PCF(S) = PCF(R) \) then, according to lemma 4.4, there will not be a D-error for any choice of \( g_j \). By assumption 4.2, this condition is always satisfied.

When choosing \( f_i \) and \( g_j \) for query reduction, if zero A-error is desired then, by lemma 4.2, we should choose \( PCF(f_i) \land PCF(g_j) = PCF(f_i) \). According to lemma 4.3, the strategy to satisfy \( PCF(f_i) \land PCF(g_j) = PCF(f_i) \) is to
assign a $\downarrow$ operator to the minuend $f_i$ and/or a $\uparrow$ operator to the subtrahend $g_j$, and keep modifying $f_i$ and $g_j$ until $PCF(f_i) \land PCF(g_j) = PCF(f_i)$. We give an example.

**Example 4.1.** Suppose that relations $R$ and $S$ have the scheme (Name, Age, Salary) and are fragmented on Age. The subset chains for $R$ and $S$ are shown in figures 4.1.a and 4.1.b, respectively.

Suppose that $R = \{(a, 10), (b, 25), (c, 32)\}$, $S = \{(a, 10), (d, 24), (c, 32)\}$. Clearly, $R - S = \{(b, 25)\}$. According to figure 4.1, we have

- $f_1 = \{(a, 10)\}$, $PCF(f_1)$ is Age $\leq$ 20;
- $f_2 = \{(a, 10), (b, 25)\}$, $PCF(f_2)$ is Age $\leq$ 30;
- $f_3 = \{(a, 10), (b, 25), (c, 32)\}$, $PCF(f_3)$ is Age $\leq$ 40;
- $g_1 = \{(a, 10)\}$, $PCF(g_1)$ is Age $\leq$ 20;
- $g_2 = \{(a, 10), (d, 24)\}$, $PCF(g_2)$ is Age $\leq$ 30;
- $g_3 = \{(a, 10), (d, 24), (c, 32)\}$, $PCF(g_3)$ is Age $\leq$ 40.

$< f_2, g_2 >$ is an a-safe pair since $PCF(f_2) \land PCF(g_2) = PCF(f_2)$.

$f_2 - g_2 = \{(b, 25)\} \subset R - S$, having no A-error. But, $< f_3, g_2 >$ is not an a-safe pair because $PCF(f_3) \land PCF(g_2) \neq PCF(f_3)$.

$f_3 - g_2 = \{(b, 25), (c, 32)\} \not\subset R - S$, where the tuple (c, 32) is an A-error. $< f_3, g_2 >$ can be modified into an a-safe pair by either changing $f_3$ to $f_2$ or
changing $g_2$ to $g_3$. $\square$

From example 4.1, we have the following observations:

1. By using lemma 4.2.a, checking and generating an a_safe pair does not need any database access. All operations are on the given subset chains. Therefore, this approach has good time efficiency, a very important feature in real time systems.

2. In addition to $< f_2, g_2 >$, $< f_2, g_1 >$ is also an a_safe pair because $f_2 - g_1 = \{(b.25)\} \subset R - S$ even though $PCF(f_2) \land PCF(g_1) \neq PCF(f_2)$. This shows that $PCF(f) \land PCF(g) = PCF(f)$ is a sufficient condition for $< f, g >$ being an a_safe pair.

Now we generalize error removal to the general case, i.e., for a multi-operator nonmonotone query.

For a query $Q = Q(R_1, R_2, ..., R_n)$, $Q_m^l$ denotes a modified query of $Q$ using fragment list $l = [f^{R_1}_1, f^{R_2}_j, ..., f^{R_n}_l]$, where $f^{R_i}_g$ is the $g^{th}$ fragment in the superset/subset chain of relation $R_i$. In modifying a multi-operator nonmonotone query, instead of finding a safe replacement pair, we find a safe (replacement) list for $Q_m^l$.

**Definition 4.2. (Safe Replacement List)** Given a nonmonotone query $Q(R_1, R_2, ..., R_n)$ and its modified query $Q_m^l$,

(a) for query reduction, $l$ is an addition-safe (replacement) list (denoted as a_safe list) for $Q_m^l$ if $Q_m^l$ produces no A-errors:

(b) for query enlargement, $l$ is a deletion-safe (replacement) list (denoted as d_safe list) for $Q_m^l$ if $Q_m^l$ produces no D-errors.

Let a derived fragment be an RA expression $e$ with fragments and base
relations.

Lemma 4.5. In query reduction (query enlargement) of $Q$, $l$ is an a_ (d_)
fail list for query $Q'_m$ if for any subexpression $e = e_m - e_s$ (i.e., $< e_m, e_s >$
is a subtrahend-minuend fragment pair) in $Q'_m$, $e_m$ and $e_s$ are both A-safe
(D-safe), respectively.

Proof. Suppose $Q'_m$ contains $n$ subtrahend-minuend fragment pairs, all of
them are safe pairs. But $Q'_m$ is not a safe modified query. According to the
definition of safe modified query, there must exist a tuple $t$ in the output of
$Q'_m$ such that $t$ is not in the output of $Q$. Assume tuple $t$ is produced by the
subtrahend-minuend fragment pair $p$ of $Q'_m$. According to lemma 4.2, $p$ is not
a safe fragment pair. Contradiction to the hypothesis. Q.E.D.

For example, in query $Q = \left( (R - S) \bowtie T - U \right) \cup V)$, there are two subtrahend-
minuend fragment pairs, $< R, S >$ and $< (R - S) \bowtie T, U >$. For a fragment
list $l = [r, s, t, u, v]$, if both $< r, s >$ and $< (r - s) \bowtie t, u >$ are safe pairs, then
$Q'_m = \left( (r - s) \bowtie t - u \right) \cup v)$ is a safe modified query of $Q$.

In order to check the safety of a fragment pair at least one of which is an RA
expression $e$ (i.e., a derived fragment), we need to derive a PCF characterizing
the expression $e$. Similar to the query reduction safety check of a fragment pair,
for $e_m - e_s$, where $e_m$ and $e_s$ are derived fragments, one expects that $PCF(e_m) \land
PCF(e_s) = PCF(e_m)$ provides a sufficient condition that $< e_m, e_s >$ is an
a_safe pair. Note that, using the transformation rules given in lemma 4.6, we
can transform a query $Q$ into $Q'$ such that there is no union operator in all
subtrahend and minuend subexpressions in $Q'$. Therefore, we only need to
provide PCF derivation rules for intersection, difference, join, selection and
projection operations. The derivation rules are given in definition 4.3.

Lemma 4.6. Let $e_1$, $e_2$ and $e_3$ be arbitrary relational algebra expressions. The
following equalities always hold.

\[(e_1 \cup e_2) - e_3 = (e_1 - e_3) \cup (e_2 - e_3) \quad (t.4.1)\]

\[e_1 - (e_2 \cup e_3) = (e_1 - e_2) \cap (e_1 - e_3) \quad (t.4.2)\]

\[(e_1 \cup e_2) \cap e_3 = e_1 \cap e_3 \cup e_2 \cap e_3 \quad (t.4.3)\]

\[\sigma_C(e_1 \cup e_2) = \sigma_C(e_1) \cup \sigma_C(e_2) \quad (t.4.4)\]

\[\pi_A(e_1 \cup e_2) = \pi_A(e_1) \cup \pi_A(e_2) \quad (t.4.5)\]

\[(e_1 \cup e_2) \Join e_3 = e_3 \Join (e_1 \cup e_2) = (e_1 \Join e_3) \cup (e_2 \Join e_3) \quad (t.4.6)\]

**Proof:** Simple and omitted. Q.E.D.

**Lemma 4.7.** Equalities are sufficient to transform any RA expression \( E \) into \( E_1 \cup E_2 \cup \cdots \cup E_m \) such that there are no union operators in \( E_i \), \( 1 \leq i \leq m \).

**Proof:** Lemma 4.7 can be proved by induction on the number of operators in \( E \). Simple and omitted. Q.E.D.

**Definition 4.3 (PCF Derivation Rules).** Let \( f \) and \( g \) be either base fragments (i.e., fragments of base relations) or derived fragments (of arbitrary RA expressions). The PCF derivation rules for fragments derived from intersection, difference, join, selection and projection operations are

\[PCF(f \cap g) \equiv PCF(f) \land PCF(g);\]  
\[PCF(f \Join g) \equiv PCF(f) \land PCF(g);\]  
\[PCF(f - g) \equiv PCF(f) \text{ with } PCF(f) \land PCF(g) \equiv PCF(f);\]  
\[PCF(\sigma_C(f)) \equiv PCF(f);\]  
\[PCF(\pi_X(f)) \equiv PCF(f); \text{ the fragmentation attribute of } f \text{ is in } X.\]
For example, consider the derived fragment \( f - (g \cap h) \) where \( f, g \) and \( h \) are base fragments fragmented on the same attribute \( A \). Suppose \( PCF(f) = 20 \leq A \leq 30, PCF(g) = 10 \leq A \leq 40 \) and \( PCF(h) = 20 \leq A \leq 50 \). We have \( PCF(g \cap h) = 20 \leq A \leq 40 \) and \( PCF(f - (g \cap h)) = 20 \leq A \leq 30 \).

**Lemma 4.8.** Let \( E \) be the original (isologous) RA expression for an arbitrary derived fragment \( e \). \( \sigma_{PCF(e)}(E) = e \).

**Proof.** We prove by induction on \( n \), the number of RA operators in \( e \).

**Base:** \( n = 1 \).

In the base case, \( E = R \theta S \) with \( \theta \in \{\cap, -, \Join\} \) or \( E = \alpha(R) \), with \( \alpha \in \{\sigma, \pi\} \). Then \( e = f\theta g \) with \( f = \sigma_{PCF(f)}(R) \) and \( g = \sigma_{PCF(g)}(S) \) or \( e = \alpha(f) \) with \( f = \sigma_{PCF(f)}(R) \).

**Case 1:** \( e = f\theta g \) with \( \theta \in \{\cap, \Join\} \). Then

\[
\sigma_{PCF(e)}(E) = \sigma_{PCF(f) \wedge PCF(g)}(R\theta S)
\]

Since \( PCF(f) \wedge PCF(g) \) are both over the same fragmentation attribute (assumption 4.1) we have

\[
\sigma_{PCF(f) \wedge PCF(g)}(R\theta S) = \sigma_{PCF(f)}(R) \theta \sigma_{PCF(g)}(S) = f\theta g = e
\]

**Case 2:** \( e = f - g \). Then

\[
\sigma_{PCF(e)}(E) = \sigma_{PCF(f)}(R - S)
\]

Since \( PCF(f) \wedge PCF(g) = PCF(f) \), we have

\[
\sigma_{PCF(e)}(E) = \sigma_{PCF(f)}(R - S) = \sigma_{PCF(f) \wedge PCF(g)}(R - S) =
\]

\[
\sigma_{PCF(f)}(R) - \sigma_{PCF(g)}(S) = f - g = e
\]

**Case 3:** \( e = \sigma_C(f) \). Since \( PCF(f) = PCF(\sigma_C(f)) \), we have
\[ \sigma_{PCF(e)}(E) = \sigma_{PCF(\sigma_X(f))}(\sigma_X(R)) = \sigma_{PCF(f)}(\sigma_X(R)) = \sigma_X(\sigma_{PCF(f)}(R)) = \sigma_X(f) = e \]

Case 4: \( e = \pi_X(f) \) where the fragmentation attribute \( A \) belongs to \( X \). Since \( PCF'(f) = PCF(\pi_X(f)) \), we have

\[
\sigma_{PCF(e)}(E) = \sigma_{PCF(\pi_X(f))}(\pi_X(R)) = \sigma_{PCF(f)}(\pi_X(R)) = \pi_X(\sigma_{PCF(f)}(R)) = \pi_X(f) = e
\]

Induction step: Assume \( \sigma_{PCF(e)}(E) = e \) is true when the number of operators in \( E \) is \( k-1 \). That is, for any tuple \( t \) in \( E \) with \( t.A \in D_e, t \in e \). We now prove that \( \sigma_{PCF(e)}(E) = e \) is also true when \( n = k \). We will prove the induction step in the cases where the incorporated operator is \( \cap, \cdot, \times, \pi \) and \( \sigma \), separately.

Case 1: \( e' = e \cap f \). According to the induction hypothesis, the tuples in \( E \) satisfying \( PCF(e) \) are in \( e \) and the tuples in \( R \) satisfying \( PCF(f) \) are in \( f \). Therefore, any tuple \( t \) in \( E \cap R \) with \( t.A \in D_e \cap D_f \) is in \( e \cap f \). Since \( D_e \cap D_f \) is equivalent to \( PCF(e) \land PCF(f) \), we have that the tuples in \( E' \) satisfying \( PCF(e') = PCF(e) \land PCF(f) \) are also in \( e' \), i.e., \( \sigma_{PCF(e')}(E') = e' \).

Case 2: \( e' = e \times f \). According to the induction hypothesis, the tuples in \( E \) satisfying \( PCF(e) \) is in \( e \) and the tuples in \( R \) satisfying \( PCF(f) \) is in \( f \). Therefore, any tuple \( t \) in \( E \times R \) with \( t.A \in D_e \cap D_f \) must be in \( e \times f \). Since \( D_e \cap D_f \) is equivalent to \( PCF(e) \land PCF(f) \), we have that the tuples in \( E' \) satisfying \( PCF(e') = PCF(e) \land PCF(f) \) must be in \( e' \), i.e., \( \sigma_{PCF(e')}(E') = e' \).

Case 3: \( e' = e - f \) or \( e' = f - e \). According to the induction hypothesis, the tuples in \( E \) satisfying \( PCF(e) \) is in \( e \) and the tuples in \( R \) satisfying \( PCF(f) \) is in \( f \). Therefore, any tuple \( t \) in \( E - R \) (\( R - E \)) with \( t.A \in D_e \) (or \( t.A \in D_f \)) and \( PCF(e) \land PCF(f) = PCF(e) \land PCF(f) \) must be in \( e - f \) (\( f - e \)). Since \( D_e \land D_f \) is equivalent to \( PCF(e) \land PCF(f) \), we have
that the tuples in \( E' = E - R \) satisfying \( PCF(e') = PCF(e) \) must be in \( e' \), or, the tuples in \( E' = R - E \) satisfying \( PCF(e') = PCF(f) \) must be in \( f \), i.e., \( \sigma_{PCF(e')}(E') = e' \).

**Case 4:** \( e' = \sigma_C(e) \). According to the induction hypothesis, the tuples in \( E \) satisfying \( PCF(e) \) is in \( e \). Therefore, any tuple \( t \) in \( \sigma_C(E) \) with \( t.A \in D_e \) must be in \( \sigma_C(e) \), no matter what \( C \) is. Since \( D_e \) is equivalent to \( PCF(e) \), we have that the tuples in \( E' \) satisfying \( PCF(e') = PCF(e) \) must be in \( e' \), i.e., \( \sigma_{PCF(e')}(E') = e' \).

**Case 5:** \( e' = \pi_X(e) \). According to the induction hypothesis, the tuples in \( E \) satisfying \( PCF(e) \) is in \( e \). Since the fragmentation attribute of \( e \) is in \( X \), any tuple \( t \) in \( \pi_X(E) \) with \( t.A \in D_e \) must be in \( \pi_X(e) \). Since \( D_e \) is equivalent to \( PCF(e) \), we have that the tuples in \( E' \) satisfying \( PCF(e') = PCF(e) \) must be in \( e' \), i.e., \( \sigma_{PCF(e')}(E') = e' \). **Q.E.D.**

**Lemma 4.9.** For \( e_m - e_s \), where \( e_m \) and \( e_s \) are either base or derived fragments, \( <e_m, e_s> \) is a safe pair if \( PCF(e_m) \land PCF(e_s) = PCF(e_m) \).

**Proof.** By lemma 4.8, the tuples in \( E_m \) (\( E_s \)) satisfying \( PCF(e_m) \) (\( PCF(e_s) \)) are in \( e_m \) (\( e_s \)). The correctness of lemma 4.9 follows from lemma 4.2.a. **Q.E.D.**

In definition 4.3, rule d.5 requires that the fragmentation attribute of \( f \) is in \( X \). If the schema of the derived fragment \( \pi_X(f) \) does not include the fragmentation attribute used by \( f \), using the PCF obtained by the derivation rule d.5 in the safety check may not be correct. For example, assume relations \( R \) and \( S \) have the schema (A, B), and are fragmented on B. Suppose that \( f \) is a fragment of \( R \) with \( PCF(f) = B \leq 20 \) and \( g \) is a fragment of \( S \) with \( PCF(g) = B \leq 40 \). Consider the query \( \pi_A(f) - \pi_A(g) \). By the derivation rule d.5, \( PCF(\pi_A(f)) = PCF(f) = B \leq 20 \), \( PCF(\pi_A(g)) = PCF(g) = B \leq 40 \). This
shows that $\text{PCF}(\pi_A(f))$ is safely covered by $\text{PCF}(\pi_A(g))$. However, assume that $A$ values in both $R$ and $S$ are unique and, in $R$, there is a tuple $(a, 20)$ and, in $S$, there is a tuple $(a, 50)$. The tuple $(a, 20)$ will remain in the output of $\pi_A(f) - \pi_A(g)$. Obviously, the tuple $(a, 20)$ constitutes an A-error since it does not belong to $\pi_A(R) - \pi_A(S)$.

One solution for the above described problem is as follows. Suppose there is a subexpression $\pi_X(f)$ in the expression of a modified query $Q_m$. The fragmentation attribute of $f$ is not in $X$. Let $F$ be the original expression (i.e., all operands in $F$ are base relations) of $f$. Consider an arbitrary expression $e_m - e_s$. If $\pi_X(f)$ is in $e_s$, to guarantee the safety of $e_m - e_s$, we change $\pi_X(f)$ to $\pi_X(F)$. If $\pi_X(f)$ is in $e_m$, we replace all fragments in $e_s$ by their base relations.

Lemmas 4.6 and 4.9 provide an approach to perform a safety check for arbitrary nonmonotone modified queries. Based on lemmas 4.6 and 4.9, for a nonmonotone query $Q$, the A-error removal algorithm for a multi-operator modified query $Q_m^l$ is now given in figure 4.2.

The correctness of algorithm A-ERROR-REMOVAL is given in lemma 4.10.

**Lemma 4.10.** Algorithm A-ERROR-REMOVAL eliminates A-errors in query reduction.

**Proof.** First, we show that the algorithm A-ERROR-REMOVAL stops. Suppose there are $k$ relations involved in $Q_m^l$: $f_{11}, f_{21}, ..., f_{k_1}$ are the smallest fragments in the $k$ subset chains, respectively. By assumptions 4.1 and 4.2, all of the $k$ relations are fragmented on the same attribute and $\text{PCF}(f_{11}) = \text{PCF}(f_{21}) = ... = \text{PCF}(f_{k_1})$. Clearly, for any $e_m - e_s$ in $Q_m^l$, eventually

$$\text{PCF}(e_m) \land \text{PCF}(e_s) = \text{PCF}(e_m)$$  \hspace{1cm} \text{(f.4.1)}

will be satisfied since, in the worst case, all $f_j$ in $e_m$ will be reduced to $f_{j_1}$,
Algorithm A-ERROR.REMOVAL($Q'_{m}$)

Input: The modified RA query $Q'_{m}$.
Output: An A-safe modified query.

begin
In bottom-up direction, for each subtree $e_m - e_s$ of the parse tree of $Q'_{m}$ do
\{ $e_m$ and $e_s$ are RA expressions.\}
while $PCF(e_m) \land PCF(e_s) \neq PCF(e_m)$ do
\begin{align*}
& \text{begin} \\
& \text{reduce} (e_m); \\
& \text{Derive} \ PCF(e_m); \\
& \text{end:}
\end{align*}
Return the resulting modified a-safe version of the query $Q'_{m}$;
end.

Procedure reduce($e$)

Input: An RA subexpression $e$ with fragments.
Output: A modified version of $e$ with some reduced fragments.

begin
if $e$ is monotone or a base relation or a fragment then
assign operator \(\downarrow\) to some or all of the base relations or the fragments in $e$;
else
Traverse the parse tree of $e$ in preorder (root-left-right) except when a set difference node $e_m - e_s$ is encountered, in which case do not traverse the subtree for $e_s$.
During the traversal, assign operator \(\downarrow\) to some or all of the base relations and fragments that are encountered;
Replace in $e$ each of the fragments or relations with \(\downarrow\) operator by a subset from its subset chain;
Return the modified $e$;
end.

Figure 4.2: Algorithm A-ERROR.REMOVAL.
\[ r_3 \cdot \text{Age} \leq 40 \quad s_3 \cdot \text{Age} \leq 40 \quad t_3 \cdot \text{Age} \leq 40 \]
\[ r_2 \cdot \text{Age} \leq 30 \quad s_2 \cdot \text{Age} \leq 30 \quad t_2 \cdot \text{Age} \leq 30 \]
\[ r_1 \cdot \text{Age} \leq 20 \quad s_1 \cdot \text{Age} \leq 20 \quad t_1 \cdot \text{Age} \leq 20 \]

(a) Subset chain for \( R \) \hspace{1cm} (b) Subset chain for \( S \) \hspace{1cm} (c) Subset chain for \( T \)

Figure 4.3: Subset Chains for Relations \( R, S \) and \( T \).

thus automatically satisfying the while condition in the algorithm.

Second, when the algorithm stops, the while condition and the bottom-up traversal of the parse tree of \( Q'_m \) guarantees that no set difference operator in \( Q'_m \) with an A-error remains. Thus \( Q'_m \) is free of A-errors. \hspace{1cm} \text{Q.E.D.}

Example 4.2 illustrates the safety check for a multi-operator modified query.

**Example 4.2.** Three relations \( R, S \) and \( T \) are fragmented on person’s age.

The corresponding fragment chains for \( R, S \) and \( T \) are shown in figures 4.3.a. 4.3.b and 4.3.c, respectively. For query \( Q(R, S, T) = (R - (S - T)) \bowtie U \), we have two candidate fragment lists, \( l_1 = \{r_2, s_2, t_2, u_1\} \) and \( l_2 = \{r_1, s_2, t_1, u_2\} \).

The two "−" subtrees in the parse tree are \( \{< S, T >, < R, S - T >\} \).

According to figure 4.3, the \( PCF(r_2) = PCF(s_2) = PCF(t_2) = \text{Age} \leq 30 \).

Since \( PCF(s_2) \) is safely covered by \( PCF(t_2) \), \( < s_2, t_2 > \) is an \( a\text{-safe} \) pair. Using the derivation rule d.4. we have \( PCF(s_2 - t_2) = \text{Age} \leq 30 \).

Since \( PCF(r_2) \) is safely covered by \( PCF(s_1 - t_2) \), \( < r_2, s_2 - t_2 > \) is also an \( a\text{-safe} \) pair. Hence, according to lemma 4.5, \( l_1 \) is an \( a\text{-safe} \) list of \( Q \).

However, \( l_2 \) is not an \( a\text{-safe} \) list of \( Q \) because \( PCF(s_2) \) is not safely covered by \( PCF(t_1) \). \( l_2 \) can be changed into a safe one by reducing \( s_2 \) to \( s_1 \). \hspace{1cm} \Box
4.3 Error Estimation Technique

In some cases, users may allow A-errors in query reduction and D-errors in query enlargement as long as the errors have a bound. In order to meet the user-specified error bounds in an error constraint, or to report the error size in a modified query, we need to design error estimation techniques.

In our earlier work [HouO 91, HoOT 89, HoOT 88, OzDu 91], we introduced methods based on sampling and statistical techniques to estimate the values of RA aggregate functions such as COUNT, SUM and AVERAGE. Here we design new estimators for D- and A-errors. In this section, we only discuss the error estimation in only query reduction. Error estimation in query enlargement is quite similar.

Let us first consider the query $R - S$ and its modified version $f_i - g_j$. According to the definitions given in chapter 2, the sizes of D- and A-errors for $f_i - g_j$ are

$$\text{Size of D-error} = \frac{|(R - S) - (f_i - g_j)|}{|R - S|} = \frac{\text{COUNT}((R - S) - (f_i - g_j))}{\text{COUNT}(R - S)} \quad (f.4.2)$$

$$\text{Size of A-error} = \frac{|(f_i - g_j) - (R - S)|}{|(f_i - g_j)|} = \frac{\text{COUNT}((f_i - g_j) - (R - S))}{\text{COUNT}(f_i - g_j)} \quad (f.4.3)$$

where $\text{COUNT}(\text{RA-expression})$ gives the number of tuples in the RA-expression. We use $\text{COUNT}(\text{RA-expression})$ to replace $\text{COUNT}(\text{RA-expression})$ in f.4.2 and f.4.3, where $\text{COUNT}(\text{RA-expression})$ is an estimator for $\text{COUNT}(\text{RA-expression})$.

From the performance analysis in [OzDu 91], the accuracy of $\text{COUNT}(\text{RA-expression})$ is dependent not only on the sample size, but also on the sampling method used and the RA-expression structure. The variance of $\text{COUNT}$ for a simpler RA expression is usually smaller than that for a complicated one.
Therefore, whenever it is possible, we simplify the RA-expression, i.e., make it contain as fewer operators as possible.

Stratified random sampling [Coch 77] may produce a precision gain in the estimates of characteristics of the whole population. When a population is stratified, if each stratum is homogeneous in that the measurements vary little from one unit to another, a precise estimate of any stratum mean can be obtained from a small sample in that stratum. These estimates can then be combined into a precise estimate for the whole population. Please note that the relations in our problem have already been stratified by relation fragmentation. The stratum is \( f'_i \) for unary operations and \( (f'_i, g'_i) \) for binary operations.

According to the experimental results from [OzDu 91], when relations are stratified (or fragmented), the stratified random sampling for COUNT outperforms significantly the other sampling plans. This is because for union, set difference, intersection, selection and join and projection\(^2\) operations, each stratum is more homogeneous than the whole population (higher selectivity).

Taking advantage of the relation fragmentation, we transform formulas f.4.2 and f.4.3 into the versions in which (1) the RA-expression being counted is greatly simplified; (2) the stratified random sampling can be used.

**Lemma 4.11.** For \( f_{i,j} \subseteq R \) and \( g_{i,j} \subseteq S \), \( 1 \leq i < j \leq n \), \( f_{i,j} = f'_i \cup f'_{i+1} \cup \ldots \cup f'_j \) and
\[
g_{i,j} = g'_i \cup g'_{i+1} \cup \ldots \cup g'_j,
\]
\( \text{COUNT}(f_{i,j} \theta g_{i,j}) = \sum_{k=i}^{j} \text{COUNT}(f'_k \theta g'_k) \) where \( \theta \in \{-, \cap\} \).

**Proof.** If there exist a tuple \( t \) in both \( f_{i,j} \) and \( g_{i,j} \), then \( t \) being in fragment \( f'_k \) of \( f_{i,j} \) implies that \( t \) is in fragment \( g'_k \) of \( g_{i,j} \) because both \( R \) and \( S \) are fragmented on the same attribute \( A \). Therefore, \( f_{i,j} \theta g_{i,j} = (\cup_{k=i}^{j} f'_k) \theta (\cup_{k=i}^{j} g'_k) = \)

\(^2\)The join and projection operations whose join or projection attributes include the fragmentation attribute.
\( w_{k=i}^j (f_k^j \theta g_k^j) \).

For two sets \( A \) and \( B \), if \( A \cap B = \phi \), then \( A \cup B = A \cup B \). According to the inclusion and exclusion principle, \( |A \cup B| = |A| + |B| - |A \cap B| = |A| + |B| \). Therefore, \( \text{COUNT}(f_i^j \theta g_i^j) = |w_{k=i}^j (f_k^j \theta g_k^j)| = \sum_{k=i}^j |(f_k^j \theta g_k^j)| = \sum_{k=i}^j \text{COUNT}(f_k^j \theta g_k^j) \). Q.E.D.

**Lemma 4.12.** If for \( k = i + 1 \), \( i + 2 \), \ldots, \( j \), the sample total \( \text{COUNT}(f_k^j \theta g_k^j) \) is unbiased, then \( \text{COUNT}(f_i^j \theta g_i^j) = \sum_{k=i}^j \text{COUNT}(f_k^j \theta g_k^j) \) is an unbiased estimate of \( \text{COUNT}(f_i^j \theta g_i^j) \), where \( \theta \in \{-, \cap\} \).

**Proof.** Let \( Y = \text{COUNT}(f_i^j \theta g_i^j) \), the total of a population. Let \( \hat{Y} = \text{COUNT}(f_i^j \theta g_i^j) \), the estimate of \( Y \). and \( \hat{y}_k = \text{COUNT}(f_k^j \theta g_k^j) \), the sample total of the \( k^{th} \) stratum of the population. The proof of lemma 4.12 is provided in [Coch 77]. Q.E.D.

**Lemma 4.13.** For \( Q = R - S \), \( Q_m = f_i^j - g_j^j \), and query reduction

\[
\text{Size of D-error} = \frac{|(R-S)-(f_i^j-g_j^j)|}{|(R-S)|} = \frac{\sum_{k=m+1}^n \text{COUNT}(f_k^j-g_k^j)}{\sum_{k=m+1}^n \text{COUNT}(f_k^j-g_k^j)} \tag{4.4}
\]

\[
\text{Size of A-error} = \frac{|(f_i^j-g_j^j)-(R-S)|}{|(f_i^j-g_j^j)|} = \frac{\sum_{k=m+1}^{i-1} \text{COUNT}(f_k^j-g_k^j)}{\sum_{k=m+1}^{i-1} \text{COUNT}(f_k^j-g_k^j) + \sum_{k=m+1}^n |I_k^j|} \tag{4.5}
\]

where \( j < i \).

**Proof.** Referring to the Venn diagram shown in figure 4.4, we have

\[
f_i^+ = w_{k=i+1}^n f_k^j, \quad g_i^+ = w_{k=i+1}^n g_k^j, \quad f_i^- = w_{k=i+1}^{i-1} f_k^j \quad \text{and} \quad g_i^- = w_{k=i+1}^{i-1} g_k^j.
\]

It is evident to see from figure 4.4 that \( (R-S) - (f_i^j - g_j^j) = f_i^+ - g_i^+ \), \( (f_i^j - g_j^j) - (R-S) = f_i^- \cap g_j^- \), and \( f_i - g_j = (f_j - g_j) \cup f_i^+ \). According to lemma 4.11, we have

\[
|f_i^+ - g_i^+| = \sum_{k=i+1}^n \text{COUNT}(f_k^j - g_k^j), \quad |f_i^- \cap g_j^-| = \sum_{k=j+1}^{i-1} \text{COUNT}(f_k^j \cap g_k^j)
\]

and
$|f_i - g_i| = \sum_{k=1}^{l} COUNT(f_i - g'_k)$. Replacing $COUNT(RA-expression)$ by $COUNT(RA-expression)$ completes the proof. Q.E.D.

One can find that the RA-expressions to be counted ($f'_k - g'_k$ or $f'_k \cap g'_k$) in formulas f.4.4 and f.4.5 are much simpler than those in formulas f.4.2 and f.4.3, respectively. According to the performance analysis of $COUNT(RA-expression)$ in [OzDu 91], using formulas f.4.4 and f.4.5, one can achieve higher accuracy than using formulas f.4.2 and f.4.3.

Another advantage of formulas f.4.4 and f.4.5 is their high efficiency in sample utilization. For example, both $R$ and $S$ are equally fragmented into $l$ fragments. Suppose $|R| = |S| = N$. For every $f'_i$ or $g'_i$, $|f'_i| = |g'_i| = \frac{N}{l}$. Clearly, the number of points in $|R| \times |S|$ point space is $N^2$: the number of points in $|f'_i| \times |g'_i|$ point space is $\frac{N^2}{l^2}$. Assume we use both non-stratified random sampling and stratified random sampling for $COUNT(R - S)$ with the same sample rate, say $\lambda$. Using non-stratified random sampling, the sample size is $\lambda \times N^2$. Using stratified random sampling, the sample size for each stratum is $\lambda \times \frac{N^2}{l^2}$. Therefore, the total sample size using stratified random sampling is $l \times \lambda \times \frac{N^2}{l^2} = \lambda \times \frac{N^2}{l}$, $l$ times less than that using non-stratified random sampling. Therefore, formulas f.4.4 and f.4.5 have better time efficiency.

The general formulas for evaluating the sizes of A- and D-errors for an
arbitrary query $Q$ and its modified version $Q_m$ are given in chapter 2 (f.2.1 and f.2.2). For an arbitrary query $Q$, $Q - Q_m$ and $Q_m - Q$ may be very complex. For example, for $Q = R - S \bowtie T$ and $Q_m = r - s \bowtie t$, $Q - Q_m$ is $Q = (R - S \bowtie T) - (r - s \bowtie t)$. The accuracy of estimating the output size of a query decreases as the query becomes more complex. Therefore, we need to investigate ways of simplifying $Q - Q_m$ and $Q_m - Q$ and making estimations of $|Q - Q_m|$ and $|Q_m - Q|$ more accurate. The query simplification techniques and COUNT estimation techniques using relation stratification (fragmentation) used in estimating the errors for an arbitrary modified query are discussed in next section.

For a given fragment list $Q^i_m$ with an a_error bound in query reduction, suppose $Q^i_m$ does not meet the given error bound. We design algorithm ERROR_ENFORCEMENT shown in figure 4.5 to make the modified query meet the given a_error constraint (bound of a_error size).

4.4 Simplification of $Q - Q_m$ and $Q_m - Q$

For an arbitrary query $Q$, $Q - Q_m$ and $Q_m - Q$ may be very complicated. This leads the costs of estimating $|Q - Q_m|$ and $|Q_m - Q|$ to be high, e.g., sample size is large. As we have mentioned above, the accuracy of estimating the size of a complicated RA expression $E$ is also low. Therefore, we need to investigate a possible way to simplify $Q - Q_m$ and $Q_m - Q$ if there is any.

For an arbitrary query $Q$, $Q_m \nsubseteq Q$ in general (has A-error). Let $Q_m = Q'_m \uplus Q''_m$ such that $Q'_m \cap Q = Q'_m$ and $Q''_m \cap Q = \phi$.

Lemma 4.14. For an arbitrary query $Q$ and its modified version $Q_m = Q'_m \uplus Q''_m$,

\[
\text{the size of D-error } = \frac{|Q - Q_m|}{|Q|} = 1 - \frac{|Q'_m|}{|Q|} \quad (f.4.6)
\]
Algorithm **A-ERROR_SIZE_ENFORCEMENT**(\(Q_m^l, \mathcal{A}\))

\textbf{Input:} The modified RA query \(Q_m^l\) and the a-error bound \(\mathcal{A}\).

\textbf{Output:} A modified query satisfying the given error constraint.

\textbf{begin}

\textbf{if} \(\hat{a}(Q_m^l) > \mathcal{A}\) \textbf{then} \quad \{\(a(Q_m^l)\) denotes the a-error size of \(Q_m^l\).

\quad \hat{a}(Q_m^l)\) denotes the estimate of \(a(Q_m^l)\).\}

\textbf{repeat do}

\quad In top-down direction. \textbf{for} each subtree \(e_m - e_s\) of \(Q_m^l\) \textbf{do}

\quad \textbf{begin}

\quad \quad reduce(\(e_m\)) \text{ and/or enlarge}(\(e_s\));

\quad \quad Estimate \(\hat{a}(Q_m^l)\);

\quad \textbf{end;}

\quad until \(\hat{a}(Q_m^l) \leq \mathcal{A}\);

\quad Return the resulting modified query;

\textbf{end.}

\textbf{Procedure} \textbf{enlarge}(e)

\textbf{Input:} An RA subexpression \(e\) with fragments.

\textbf{Output:} A modified version of \(e\) with \(PCF(e)\) increased.

\textbf{begin}

\quad Assign operator \(\uparrow\) to some or all of the fragments in \(e\);

\quad Replace each of the fragments with \(\uparrow\) operator by a superset

\quad \text{from its subset chain};

\quad Return the modified \(e\);

\textbf{end.}

Figure 4.5: Algorithm A-ERROR_SIZE_ENFORCEMENT.
the size of A-error $= \frac{|Q_m - Q|}{|Q_m|} = 1 - \frac{|Q'_m|}{|Q_m|}$ (f.4.7)

**Proof.** According to the inclusion/exclusion principle,

$$|Q - Q_m| = |Q| - |Q \cap Q_m| = |Q| - |Q \cap (Q'_m \cup Q''_m)| =$$

$$|Q| - |Q \cap Q'_m \cup Q \cap Q''_m| = |Q| - |Q'_m| \text{ and}$$

$$|Q_m - Q| = |Q_m| - |Q_m \cap Q| = |Q_m| - |(Q'_m \cup Q''_m) \cap Q| =$$

$$|Q_m| - |Q'_m \cap Q \cup Q''_m \cap Q| = |Q_m| - |Q'_m|.$$

This completes the proof. Q.E.D.

If $Q$ is monotone, $Q''_m = \emptyset$. Therefore, $Q_m = Q'_m$. For an arbitrary query $Q$, if $Q'_m$ is simpler than $Q - Q_m$ and $Q_m - Q$, then formulas f.4.6 and f.4.7 are more efficient than f.2.1 and f.2.2, respectively. In the next subsection, we show that such $Q'_m$ indeed exists and introduce the method to construct $Q'_m$ for an arbitrary modified query $Q_m$.

### 4.4.1 Constructing $Q'_m$ for Arbitrary Queries

For the simple nonmonotone query $Q = R - S$ and its modified version $Q_m = f - g$, $Q'_m = f - S$. This is true because for any tuple $t$ belonging to $(f - g) \cap (R - S)$, it belongs to $f - S$ (refer to the Venn diagram in figure 2.3). Clearly, $Q'_m = f - S$ is much simpler than $Q_m - Q = (f - g) - (R - S)$ and $Q - Q_m = (R - S) - (f - g)$. We generalize this $Q'_m$ construction method to the general case. If there is no ambiguity, in what follows, $Q$ (also $Q_m$, $Q'_m$ and $Q''_m$) denotes either an RA expression or the output tuples of the RA expression.

**Lemma 4.15.** For $Q = Q_1 - Q_2$ and $Q_m = Q_{1m} - Q_{2m}$, $Q'_m = Q_{1m} - (Q_{2m} \cup Q_2)$. 


Proof. Any tuple in \( Q'_m \) must be in \( Q_{1m}' \), i.e., \( Q'_m \subseteq Q_{1m}' \). Any tuple in \( Q'_m \) must not be in \( Q_{2m} \) and \( Q_2 \), i.e., \( Q'_m \cap Q_{2m} = \emptyset \) and \( Q'_m \cap Q_2 = \emptyset \). We have \( Q'_m = Q_{1m}' - (Q_{2m} \cup Q_2) \). This is also illustrated by the Venn diagram in figure 4.6. Q.E.D.

From lemma 4.15, we have two important corollaries.

Corollary 4.15.1. For \( Q = Q_1 - Q_2 \) and \( Q_m = Q_{1m} - Q_{2m} \) with \( Q_1 \) is monotone. \( Q'_m = Q_{1m} - (Q_{2m} \cup Q_2) \).

Proof. If \( Q_1 \) is monotone, then \( Q_{1m}' = Q_{1m} \). Q.E.D.

Example 4.3. For \( Q = R - (S - T) \) and \( Q_m = r - (s - t) \), where \( r, s \) and \( t \) are fragments of \( R, S \) and \( T \), respectively. By corollary 4.15.1, \( Q'_m = r - ((S - T) \cup (s - t)) \). \( Q'_m = r - ((S - T) \cup (s - t)) \) is simpler than \( Q - Q_m = (r - (S - T)) - (r - (s - t)) \) and \( Q_m - Q = (r - (s - t)) - (r - (S - T)) \). \( \square \)

Corollary 4.15.2. For \( Q = Q_1 - Q_2 \) and \( Q_m = Q_{1m} - Q_{2m} \) with \( Q_2 \) is monotone. \( Q'_m = Q_{1m}' - Q_2 \).

Proof. If \( Q_2 \) is monotone, then \( Q_{2m} \subseteq Q_2 \) (\( Q_{2m} \cup Q_2 = Q_2 \)). Q.E.D.

Corollary 4.15.3. For \( Q = Q_1 - Q_2 \) and \( Q_m = Q_{1m} - Q_{2m} \) with both \( Q_1 \) and \( Q_2 \) are monotone. \( Q'_m = Q_{1m}' - Q_2 \).

Proof. Follows corollary 4.15.1 and corollary 4.15.2. Q.E.D.

Example 4.4. For \( Q = (R \cap S) - T \) and \( Q_m = (r \cap s) - t \). By corollary 4.15.3, \( Q'_m = (r \cap s) - T \). \( Q'_m = (r \cap s) - T \) is simpler than \( Q - Q_m = ((R \cap S) - T) - ((r \cap s) - t) \). \( \square \)
Figure 4.6: Venn Diagram for $Q = Q_1 - Q_2$ and $Q_m = Q_{1m} - Q_{2m}$.

Let $Q$ be a nonmonotone RA expression. A complete nonmonotone subexpression of $Q$ is a nonmonotone subexpression of $Q$ such that it is not an operand of any set difference operator. For example, for $Q = (R - S) - (T - U) \cup V$, $(R - S) - (T - U)$ is the only complete nonmonotone subexpression while either $R - S$ or $T - U$ is not. Let $Q_{1m}$ be a modified query of a complete nonmonotone subexpression $Q_1$. $Q_{1m}'$ is constructed by repeatedly using lemma 4.15, corollaries 4.15.1, 4.15.2 and 4.15.3 starting from the leftmost "−" operator in the RA expression of $Q_{1m}$ to right, until all "−" operators of $Q_{1m}$ are processed. For example, for $Q_{1m} = (r - s) - (t - u)$, we first convert $r - s$ to $r - S$ (corollary 4.15.3) then convert $(r - S) - (t - u)$ to $(r - S) - ((t - u) \cup (T - U))$ (corollary 4.15.1).

**Lemma 4.16.** For $Q = Q_1 \theta Q_2$ and $Q_m = Q_{1m} \theta Q_{2m}$ where $\theta \in \{\cap, \cup, \setminus\}$, $Q_m' = Q_{1m}' \theta Q_{2m}'.$

**Proof.** Consider $\theta = \cap$. For $Q_{1m} = Q_{1m}' \uplus Q_{1m}''$ and $Q_{2m} = Q_{2m}' \uplus Q_{2m}'',

$$Q_m = Q_m' \uplus Q_m'' = Q_{1m} \cap Q_{2m} = (Q_{1m}' \uplus Q_{1m}'') \cap (Q_{2m}' \uplus Q_{2m}'') =$$

$$Q_{1m}' \cap Q_{2m}' \uplus Q_{1m}'' \cap Q_{2m}' \uplus Q_{1m}'' \cap Q_{2m}''.$$

Since $Q_{1m}' \subset Q_1$ and $Q_{2m}' \subset Q_2$, $Q_{1m}' \cap Q_{2m}' \subset Q$. Therefore, $Q_{1m}' \cap Q_{2m}' \subset Q_m'$. 


Since $Q_{2m} \not\subseteq Q_2$, $Q_{1m} \cap Q_{2m} \not\subseteq Q$. Therefore, $Q_{1m} \cap Q_{2m} \not\subseteq Q'$. Similarly, $Q_{1m} \cap Q_{2m} \not\subseteq Q'$ and $Q_{1m} \cap Q_{2m} \not\subseteq Q'$. Therefore, we have $Q' = Q_{1m} \cap Q_{2m}$.

Similarly, we can prove that $Q'_m = Q_{1m} \emptyset Q_{2m}$ when $\emptyset$ is $\cup$ or $\emptyset$. Q.E.D.

**Corollary 4.16.1.** For $Q = Q_1 \theta_1 Q_2 \theta_2 \cdots \theta_{k-1} Q_k$ and

$Q_m = Q_{1m} \theta_1 Q_{2m} \theta_2 \cdots \theta_{k-1} Q_{km}$ where $Q_i$ is an arbitrary RA expression, $1 \leq i \leq k$, and $\theta_i \in \{\cap, \cup, \emptyset\}$, $1 \leq i \leq k - 1$. $Q'_m = Q_{1m} \theta_1 Q_{2m} \theta_2 \cdots \theta_{k-1} Q_{km}$.

**Proof.** Obtained by repeatedly using lemma 4.16. Q.E.D.

**Lemma 4.17.** For $Q = \alpha(Q_1)$ and $Q_m = \alpha(Q_{1m})$ where $\alpha \in \{\pi, \sigma\}$,

$Q'_m = \alpha(Q_{1m})$.

**Proof.** Simple and omitted. Q.E.D.

**Lemma 4.18.** For $Q = Q(Q_1, Q_2, \ldots, Q_k, q_1, q_2, \ldots, q_l)$ and

$Q_m = Q(Q_{1m}, Q_{2m}, \ldots, Q_{km}, q_1, q_2, \ldots, q_l)$ where $Q_i$ is a monotone RA expression ($1 \leq i \leq k$) and $q_j$ is a complete nonmonotone RA expression ($1 \leq j \leq l$),

$Q'_m = Q(Q_{1m}, Q_{2m}, \ldots, Q_{km}, q_1, q_2, \ldots, q_l)$.

**Proof.** Follows from lemmas 4.15, 4.16 and 4.17. Q.E.D.

For an arbitrary query $Q$ and its modified version $Q_m$, we use lemma 4.18 to convert $Q_m$ into $Q'_m$.

**Example 4.5.** For $Q = (R_1 \emptyset (R_2 - (R_3 - R_4)) \cup (\pi(R_5 - R_6) \cap R_7)$ and $Q_m = (r_1 \emptyset (r_2 - (r_3 - r_4)) \cup (\pi(r_5 - r_6) \cap r_7)$ where $r_i$ is a fragment of $R_i$, we convert $Q_m$ into $Q'_m$ as the follows. There are two complete nonmonotone subexpressions in $Q$, $R_2 - (R_3 - R_4)$ and $R_5 - R_6$. Let $Q_1 = R_2 - (R_3 - R_4)$ and $Q_{1m} = r_2 - (r_3 - r_4)$. Let $Q_2 = R_5 - R_6$ and $Q_{2m} = r_5 - r_6$. By corollary 4.15.1, $Q_{1m} = r_2 - (r_3 - r_4 \cup R_5 - R_6)$. By corollary 4.15.3, $Q_{2m} = r_5 - R_6$.
By lemma 4.18, $Q'_m = (r_1 \times (r_2 - (r_3 \cup r_4 \cup R_5 \cup R_6)) \cup (\pi(r_5 \cup R_8) \cap r_7)$.
\[\square\]

4.4.2 Optimizing $Q'_m$

Under certain conditions, $Q'_m$ can be further optimized. In what follows, we introduce two methods for optimizing $Q'_m$.

4.4.2.1 Optimization by Transformation

Consider a query $Q = Q_1 - Q_2$ and its modified version $Q_m = Q_{1m} - Q_{2m}$. If $Q_2$ is nonmonotone, according to lemma 4.15, in $Q'_m$, $Q_{2m}$ is changed to $Q_{2m} \cup Q_2$. For example, for $Q = R - (S - (T \cap U))$ and $Q_m = r - (s - (t \cap u))$, $Q'_m = r - (s - (t \cap u)) \cup S - (T \cap U)$). Now, let us investigate a transformation given in lemma 4.19.

**Lemma 4.19.** For three arbitrary relational algebra expressions $e_1$, $e_2$ and $e_3$,

$$e_1 - (e_2 - e_3) \equiv (e_1 - e_2) \cup (e_1 \cap e_2 \cap e_3)$$

**Proof:** Simple and omitted. Q.E.D.

Using this transformation rule, the given $Q$ can be equivalently transformed into $(R - S) \cup R \cap S \cap T \cap U$. Clearly, for $Q_m = (r - s) \cup r \cap s \cap t \cap u$, $Q'_m = (r - S) \cup r \cap s \cap t \cap u$. In the $Q'_m$ for the transformed $Q$, only one relation needs to be fully evaluated. It is much simpler than that for the untransformed $Q$. Therefore, estimating the $Q'_m$ for the transformed $Q$ is obviously more accurate and time efficient.
Algorithm $Q'_m$-OPTIMIZATION ($Q'_m$)

Input: The unoptimized $Q'_m$.

Output: The optimized $Q'_m$.

begin
  for each complete nonmonotone subexpression $e$ in $Q'_m$ do
    In bottom-up direction, for each subtree $e_m - e_s$ of the parse tree of $e$ do
      begin
        Derive $PCF(e_m)$;
        {using the derivation rules given in definition 4.3.}
        for each base fragment (or relation) $f$ in $e_s$
          if $PCF(f) \land PCF(e_m) \neq PCF(f)$ then
            change $f$ to $f'$ such that $PCF(f') = PCF(e_m)$;
          end:
        Return the optimized $Q'_m$;
      end.
  end.

Figure 4.7: Algorithm for $Q'_m$ Optimization.

4.4.2.2 Optimization by Common Fragmentation Attribute

Consider query $Q = R - S$ and its modified version $Q_m = f_i - g_j$, $i > j$. By corollary 4.15.3, $Q'_m = f_i - S$. Since we have assumed that both $R$ and $S$ are fragmented on the same attribute, say, attribute $A$, and $PCF(f_i) = PCF(g_i)$, $1 \leq i \leq n$, referring to figure 4.4, it is easy to see that $Q'_m = f_i - g_k$, $i \leq k \leq n$. In other words, $Q'_m = f_i - g_k$ such that $PCF(f_i) \land PCF(g_k) = PCF(f_i)$. Clearly, $g_i$ is the smallest fragment satisfying $PCF(f_i) \land PCF(g_k) = PCF(f_i)$, hence $f_i - g_i$ is the simplest one for replacing $f_i - S$. Therefore, we optimize a subexpression $f_i - g_j$ with $i < j$ in $Q'_m$ by replacing $g_j$ with $g_i$ (note that $S = g_n$).

This optimization approach can be generalized to optimize arbitrary complete nonmonotone expressions in $Q'_m$. The general optimization algorithm is given in figure 4.7.

The correctness of algorithm $Q'_m$-OPTIMIZATION is straightforward. We
give an example to illustrate the utilization of algorithm $Q'_m$-OPTIMIZATION.

**Example 4.6.** Suppose in $Q'_m$, we have a complete nonmonotone subexpression $e = (r - S) - ((t - u) \cup (T - U))$. Relations $R$, $S$, $T$ and $U$ are fragmented on the same attribute $A$. Assume $PCF(r) = A \leq 40$, $PCF(t) = A \leq 30$, $PCF(u) = A \leq 10$, and $PCF(S) = PCF(T) = PCF(U) = A \leq 100$. Using algorithm $Q'_m$-OPTIMIZATION, we first optimize the subexpressions $r - S$, $t - u$ and $T - U$. $r - S$ is changed to $r - s$ with $PCF(s) = A \leq 40$ because $PCF(S) \land PCF(r) \neq PCF(S)$. $t - u$ and $T - U$ remain the same because $PCF(u) \land PCF(t) = PCF(u)$ and $PCF(U) \land PCF(T) = PCF(U)$. In the second round, we optimize $(r - s) - ((t - u) \cup (T - U))$. Since $PCF(r - s) = A \leq 40$, we change $T$ to $t'$, and $U$ to $u'$ with $PCF(t') = PCF(u') = A \leq 40$. $t$ and $u$ remain the same because $PCF(t) \land PCF(r - s) = PCF(t)$ and $PCF(u) \land PCF(r - s) = PCF(u)$. The optimized version of $(r - S) - ((t - u) \cup (T - U))$ is $(r - s) - ((t - u) \cup (t' - u'))$. \[ \square \]

In example 4.6, if $PCF(r) = A \leq 20$, we can further simplify $e$. In the first round, $r - S$ is changed to $r - s$ with $PCF(s) = A \leq 20$. $t - u$ and $T - U$ remain the same. In the second round, since $PCF(r - s) = A \leq 20$, we change $t$ to $t'$. $T$ to $t''$. and $U$ to $u'$ with $PCF(t') = PCF(t'') = PCF(u') = A \leq 20$. $u$ remains the same because $PCF(u) \land PCF(r - s) = PCF(u)$. The optimized version of $(r - S) - ((t - u) \cup (T - U))$ is $(r - s) - ((t' - u) \cup (t'' - u'))$. Since $t' = t''$ and $u \subseteq u'$, $(t' - u) \cup (t'' - u') = t' - u$. Therefore, $(r - s) - ((t' - u) \cup (t'' - u'))$ can be further simplified to $(r - s) - (t' - u)$. We characterize this simplification in lemma 4.20.

**Lemma 4.20.** Consider any subexpression $e_m - (e_s \cup e'_s)$ in an optimized complete nonmonotone subexpression $e$ by algorithm $Q'_m$-OPTIMIZATION, where $e'_s$ is changed from $E_s$ and $E_s$ contains no recursive set difference subexpression. If, except the fragments in a subtrahend subexpression of $e_s$, for any
other fragment \( f \) in \( e_s, PCF(f) = PCF(e_m), \) then \( e_m - (e_s \cup e'_s) = e_m - e_s. \)

**Proof.** Since \( e_m - (e_s \cup e'_s) \) is in an optimized complete nonmonotone subexpression. Since \( e \) is obtained by using algorithm \( Q'_m \)-OPTIMIZATION.

\( PCF(f) \land PCF(e_m) = PCF(f) \) for any fragment \( f \) in \( e. \) Since \( e'_s \) is changed from \( E_s, PCF(f) = PCF(e_m) \) for each fragment \( f \) in \( e'_s. \) \( e_s \) and \( e'_s \) are isologus and different only in some fragments in the subtrahend subexpressions of \( e_s \) (or \( e'_s \)). For \( f \) in \( e_s \) and \( f' \) in \( e'_s \) being such a pair of corresponding fragments, \( f \subseteq f'. \) This makes \( e'_1 - e'_2 \subseteq e_1 - e_2, \) where \( f \) is in \( e_2 \) and \( f' \) is in \( e'_2. \)

Since \( E_s \) contains no recursive set difference subexpression, we have \( e'_s \subseteq e_s. \) Q.E.D.

We can add this \( Q'_m \) optimization approach to algorithm \( Q'_m \)-OPTIMIZATION.

Now we give the general algorithm \( Q'_m \)-CONSTRUCTION in figure 4.8 for constructing and optimizing \( Q'_m \) for arbitrary queries as a summarization of simplification of \( Q - Q_m \) and \( Q_m - Q. \)

4.5 **Fragmentation Attribute Value Occurrence Probability Method to Estimate Error Size for Single Set Difference Queries**

In this section, we introduce another method for estimating the error size for single set difference queries. This new method estimates modification errors by using the probability of occurrence of the fragmentation attribute values in a relation. hence is called the fragmentation attribute value occurrence probability method (denoted as the FP method).

4.5.1 **Equal-Probability Assumptions**

Assume that \( Q_m = f_i - g_j \) has an A-error. In this section, we define estimators for A- and D-errors of query reduction for a single set difference
Algorithm $Q'_m$-CONSTRUCTION ($Q_m$)

Input: An arbitrary modified query $Q_m$.
Output: The optimized $Q'_m$.

begin
  Optimize $Q_m$ by repeatedly using lemma 4.19;
  for each complete nonmonotone subexpression $e_m - e_s$ in $Q_m$ do
    begin
      if both $e_m$ and $e_s$ are monotone then
        change $e_s$ to $E_s$; {Corollary 4.15.3.}
      else if $e_m$ is monotone then
        change $e_s$ to $e_s \cup E_s$; {Corollary 4.15.1.}
      else if $e_s$ is monotone then
        begin
          change $e_s$ to $E_s$;
          $Q'_m$-CONSTRUCTION ($e_m$); {Corollary 4.15.2.}
        end;
      else {both $e_m$ and $e_s$ are nonmonotone}
        begin
          change $e_s$ to $e_s \cup E_s$;
          $Q'_m$-CONSTRUCTION ($e_m$); {Lemma 4.15.}
        end;
    end;
  $Q'_m$-OPTIMIZATION ($Q'_m$);
  Return the optimized $Q'_m$;
end.

Figure 4.8: Algorithm for $Q'_m$ Construction.
query. We make the following assumptions (to be relaxed later).

**Assumption 4.3.** There is only one fragmentation attribute, say $A$, used in the subset chain definitions of $R$ and $S$.

**Assumption 4.4.** Let the (finite) domain of attribute $A$ be $D_A$. For any tuple $t$ in $R$ and $S$, $t.A$ is equally likely to be any value in $D_A$.

Let $z$ be a random variable for selection of a value from $D_A$. Let $p_R(z)$ denote the probability $Pr\{t \in R, t.A = z \in D_A\}$. We call $p_R(z)$ the occurrence probability of $z$ in $R$. Let $n_R(z)$ denote the expected number of tuples in $R$ associated with $z$. Then

$$n_R(z) = |R| \cdot p_R(z) \quad (\text{f.4.8})$$

Let $n_R$ denote the expected number of tuples per $A$ value for $R$. Then

$$n_R = \frac{1}{|D_A|} \sum_{z \in D_A} n_R(z) \quad (\text{f.4.9})$$

Clearly, the expected number of tuples in the fragment $f_i$ of $R$ is

$$|D_{f_i}| \cdot n_R \quad (\text{f.4.10})$$

Under the equal-probability assumption 4.4, $t.A$ is equally likely to be any value in $D_A$. Therefore, we have

$$p_R(z) = \frac{1}{|D_A|}$$

$$n_R(z) = |R| \cdot \frac{1}{|D_A|}$$

$$n_R = \frac{1}{|D_A|} \sum_{z \in D_A} |R| \cdot \frac{1}{|D_A|} = |R| \cdot \frac{1}{|D_A|} = n_R(z)$$

**Lemma 4.21.** Consider $Q = R - S$, $Q_m = f_i - g_j$. Let $D_{f_i}$ and $D_{g_j}$ be the subdomains of $D_A$ that make formulas $PCF(f_i)$ and $PCF(g_j)$ true, respectively. Then, under assumptions 4.3 and 4.4.
the size of D-error = \frac{|D_A - D_f|}{|D_A|} \quad (f.4.11)

the size of A-error = \frac{|D_A - D_f|}{|D_A| + |D_f|(|c-1|)} \quad (f.4.12)

where \( c = \frac{|R|}{|R \cap S|} \).

**Proof.** According to figure 2.3.a, the size of D-error equals \( \frac{|D|}{|BD|} \). According to figure 4.4, the A subdomain for the tuples in D is \( D_A - D_f \), and the A subdomain for the tuples in BD is \( D_A \) itself. Therefore, according to f.4.10, the expected number of tuples in D equals \( |D_A - D_f| (n_R - n_{RNS}) \), and the expected number of tuples in BD equals \( |D_A| (n_R - n_{RNS}) \). This completes the proof of f.4.11.

According to figure 2.3.a, the size of A-error equals \( \frac{|A|}{|AB|} \). According to figure 4.4, the A subdomain for the tuples in A is \( D_f - D_g \), and the A subdomain for the tuples in B is \( D_f \). Therefore, according to f.4.10, the expected number of tuples in A equals \( |D_f - D_g| n_{RNS} \), and the expected number of tuples in B equals \( |D_f| (n_R - n_{RNS}) \).

\[
\frac{|A|}{|AB|} = \frac{|D_f - D_g| n_{RNS}}{|D_f - D_g| n_{RNS} + |D_f| (n_R - n_{RNS})} = \frac{|D_f - D_g|}{|D_f| + |D_f| (\frac{n_R}{n_{RNS}} - 1)}
\]

By f.4.9, \( n_{RNS} = \frac{|A|}{|AB|} \sum_{r \in D_A} \frac{|R| \cdot p_R(r)}{|R \cap S| \cdot p_R(r)} = \frac{|R|}{|R \cap S|} = c \). This proves f.4.11. Q.E.D.

In the equal-probability case, the size of D-error defined by f.4.11 is only relevant with the sizes of domains \( D_A \) and \( D_A - D_f \). If \( c \) is obtained from \( \frac{\text{COUNT}(R)}{\text{COUNT}(R\cap S)} \) where \( \text{COUNT}(R) \) and \( \text{COUNT}(R \cap S) \) are estimates of \(|R|\) and \(|R \cap S|\), respectively, then the accuracy of f.4.12 is dependent on the accuracy of \( c \). Once \( c \) is obtained, it does not need to be recalculated unless the tuples in \( R \) and \( S \) have been changed.
Figure 4.9: Occurrence Probability Density Functions of a Single Fragmentation Attribute for R, S and R \cap S in the Non-Equal-Probability Case.

4.5.2 Non-Equal-Probability Distribution

Now, let us relax the equal-probability assumption 4.4. When assumption 4.4 is relaxed, t.A is no longer equally likely to be any value in \( D_A \). Therefore, the occurrence probabilities for different \( A \) values are no longer a constant \( \frac{1}{|D_A|} \). For example, for relation PERSON(Name, Age, Salary) fragmented on Age, the occurrence probability of Age usually has a normal distribution. Figures 4.9.a, 4.9.b and 4.9.c show the occurrence probability density functions of a single fragmentation attribute for \( R \), \( S \) and \( R \cap S \), respectively, in a general non-equal-probability case.

In the non-equal-probability case, the formulas f.4.11 and f.4.12 can not be used since for two \( A \) values \( x \) and \( y \), \( p_R(x) \) may not be equal to \( p_R(y) \). This leads the conclusion that the ratio of the sizes of two fragments is not proportional to the ratio of the sizes of the corresponding \( A \) domains.

For \( Q = R - S \) and \( Q_m = f_i - g_j \) with \( p_R(z) \) and \( p_R\cap S(z) \) given, we derive the formulas for estimating expected D- and A-error sizes in non-equal-probability
case as follows.

In non-equal-probability case, the expected number of tuples in the fragment \( f_i \) of \( R \) is

\[
\sum_{z \in D_{f_i}} |R| \cdot p_R(z) \quad (f.4.13)
\]

Therefore, according to figures 2.3.a and 4.4, the expected number of tuples in \( D \) equals

\[
\sum_{z \in D_{A - D_{f_i}}} |R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)
\]

The expected number of tuples in \( BD \) equals

\[
\sum_{z \in D_{A}} |R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)
\]

Therefore,

the expected size of D-error

\[
1 - \frac{\sum_{z \in D_{A - D_{f_i}}} |R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)}{\sum_{z \in D_{A}} |R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)} \quad (f.4.14)
\]

The expected number of tuples in \( A \) equals

\[
\sum_{z \in D_{f_i - D_{g_j}}} |R \cap S| \cdot p_{R \cap S}(z)
\]

The expected number of tuples in \( B \) equals

\[
\sum_{z \in D_{f_i}} |R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)
\]

Therefore,

the expected size of A-error

\[
\frac{\sum_{z \in D_{f_i - D_{g_j}}} |R \cap S| \cdot p_{R \cap S}(z)}{\sum_{z \in D_{f_i}} |R| \cdot p_R(z) - \sum_{z \in D_{g_j}} |R \cap S| \cdot p_{R \cap S}(z)} \quad (f.4.15)
\]

If the fragmentation attribute \( A \) values are continuous in the fragmentation attribute domain \( D_A \), f.4.14 and f.4.15 are rewritten as

the expected size of D-error

\[
1 - \frac{\int_{D_A} (|R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)) \, dz}{\int_{D_A} (|R| \cdot p_R(z) - |R \cap S| \cdot p_{R \cap S}(z)) \, dz} \quad (f.4.16)
\]
the expected size of A-error = \[\frac{\int_{D_f} |R \cap S| \cdot |p_{R \cap S}(z)| \, dz}{\int_{D_f} |R| \cdot |p_R(z)| \, dz} \]

(4.4)

4.5.3 Multiple Fragmentation Attributes

Finally, we relax the assumption 4.3. For \(R - S\), suppose \(R\) and \(S\) are fragmented on the same set of attributes \(\mathcal{A} = A_1, A_2, \ldots, A_m\). \(D_{Ai} = [a_{i1}, a_{in_i}]\), where \(n_i\) is the size of domain \(A_i\). \(D_{\mathcal{A}} = [D_{A_1}, D_{A_2}, \ldots, D_{A_m}]\). The \(\mathcal{A}\) domain, \(D_{\mathcal{A}}\), forms an \(m\)-dimensional hypercube. A point in the hypercube has a coordinate value \(a_{1k_1}, a_{2k_2}, \ldots, a_{nk_n}, a_{mk_m}\), where \(1 \leq k_i \leq n_i\). The size of domain \(\mathcal{A}\) is \(n_1 \cdot n_2 \cdot \ldots \cdot n_m\). A three-dimensional hypercube for \(D_{\mathcal{A}}\) is shown in figure 4.10. For fragment \(f_i\), the subdomain of \(D_{f_i}\) of \(D_{\mathcal{A}}\) is

\[D_{f_i} = [a_{i1} \leq A_1 \leq a_{i2}, a_{i1} \leq A_2 \leq a_{i2}, \ldots, a_{im} \leq A_m \leq a_{im}]\]

where \(a_{jk}\) is a value of attribute \(A_j\). Let \(z_i\) be a random variable for selection of a value from \(D_{A_i}, i = 1, 2, \ldots, m\). If the occurrence probabilities \(p_R(z_1, z_2, \ldots, z_m)\) and \(p_{R \cap S}(z_1, z_2, \ldots, z_m)\) are given, the formulas 4.4.11 to 4.4.17 can be directly used in the multiple fragmentation attributes case.
The advantage of using the FP method to estimate errors is that it does not need any database access provided that occurrence probability density functions of fragmentation attributes are known. All operations are on fragmentation chains. Therefore, it has a better time efficiency, a very important feature in real-time systems. Unfortunately, the FP method is only good for simple queries, because for a complicated query, it is difficult to obtain the occurrence probability density functions of fragmentation attribute for intermediate relations produced during query evaluation. The costs of finding these occurrence probability density functions may be high.
Chapter 5

Estimating COUNT, SUM, and AVERAGE RA Queries

5.1 Introduction

In real-time or time-constrained databases, queries have to be completed within a given time period. When an aggregate query in such an environment cannot be evaluated within the given time period, one approach is to evaluate a statistical estimator and produce a statistical estimate to the answer of the query. Such an approach has been proposed for COUNT relational algebra queries by Lipton and Naughton [LiNa 89], Lipton, Naughton and Schneider [LNS 90] and by Hou and Ozsoyoglu [HoOT 88, HoOT 89, HoO 91]. Hou [Hou 89] has implemented the approach given in [HoOT 88, HoOT 89, HoO 91] in CASE-DB.

This chapter extends the work on statistical estimators for aggregate relational algebra queries in CASE-DB with the following features.

1. **New statistical estimators for COUNT(E) queries with projection.** In Hou's work, whenever the relational algebra (RA) expression E contained the projection operator, he used a revised version of the Goodman's estimator [Good 49] which did not always perform well. In this chapter, we introduce two new estimators, namely, the Jackknife estimator [BuOv 79], and the Chao's estimator [Chao 84].

2. **Extending the methodology for SUM and AVERAGE aggregate queries.** Hou's work introduced estimators for COUNT
queries. We now propose and evaluate estimators for SUM and AVERAGE queries. To obtain estimators for SUM(E) and AVG(E) queries where E has a projection operator, we use the double sampling technique with the acceptance/rejection method.

3. New sampling plans based on systematic sampling and stratified sampling. To obtain samples for evaluating an estimator, earlier work used simple random sampling [HoOT 88], simple random sampling with an adaptive stopping criteria [LNS 90], and cluster sampling [HoO 91]. We now add systematic sampling and stratified sampling into the list of possible sampling techniques for aggregate query evaluation. The stratified sampling is particularly useful in enforcement of error constraints introduced in chapter 4.

Section 5.2 introduces the new estimators for COUNT(\(\pi(R)\)) queries. Generalization into estimators for COUNT(E) is given later in section 5.6. Section 5.3 introduces the double sampling approach to estimate SUM(A(\(\pi(R)\))). In section 5.4, we briefly summarize systematic sampling and stratified sampling. Finally, in section 5.5, we present the algorithms for evaluating estimators for AGG(E) where AGG \(\in\) \{COUNT, SUM, AVE\} and E is an arbitrary RA expression. Section 5.6 summaries the experimental results provided by Tjahjana [Atj 91].

5.2 New Estimators for \(\text{COUNT}(\pi_X(R))\)

Consider the query \(\text{COUNT}(\pi_X(R))\). Conceptually, the projection operation \(\pi_X(R)\) eliminates from each tuple those attributes not in X (thus creating classes of identical tuples), reduces each class of identical tuples to a single tuple, and returns the resulting tuples. That is, the projection operation on \(R\) may produce (and later eliminate) duplicate tuples. In the case that the sizes
of distinct classes produced by projection are different, the duplication makes
the probabilities of inclusion in the sample unequal; hence, introducing bias
to the estimate for COUNT(\(\pi_X(R)\)).

Goodman [Good 49] proposed a nonparametric estimation of the number
of groups in a population with known size. In [HoOT 89], Hou and Ozsoyoglu
used the Goodman's estimator to estimate COUNT(\(\pi_X(R)\)). However, when
the sampling fraction is low. Goodman's estimator is unstable for a population
with heavy duplicates.

Below we present two different nonparametric estimators to estimate
COUNT(\(\pi_X(R)\)), namely, the Jackknife estimator and the Chao's estimator.
K. P. Burnham and W. S. Overton [BuOv 79] use the Jackknife method to
estimate the population size (e.g., in our case, \(\pi_X(R)\) is the population, and
COUNT(\(\pi_X(R)\)) is the population size) when sample inclusion probabilities
vary among population elements. Chao [Chao 84] proposes another nonpara-
metric method to estimate the number of classes in a population. In what
follows, we briefly state the Goodman's estimator for comparison purposes.
and adapt the Jackknife and Chao's estimators to estimate COUNT(\(\pi_X(R)\)).

5.2.1 Goodman's Estimator

Goodman's estimator, denoted by \(\hat{G}\) and based on simple random sampling
is

\[
\hat{G} = \sum_{i=1}^{m} A_i x_i
\]

(5.1)

where \(A_i = 1 - (-1)^i \frac{(N-m+i-1)!}{m!(i-1)!} \), \(x_i\) is the number of classes containing \(i\) elements (the number of resulting tuples in a class) in a sample of size \(m\) from
relation $R$. $N$ is the number of tuples in $R$, and

$$m^{(i)} = \begin{cases} 
  n(n-1) \cdots (n-i+1) & \text{for } i > 0 \\
  1 & \text{for } i = 0 
\end{cases}$$  

(5.2)

### 5.2.2 Jackknife Estimators

The Jackknife estimator, denoted by $\hat{J}$, is developed by Burnham and Overton [BuOv 79] to be used for estimating the number of $N$ distinct animals in live-trapping studies. The studies are done by trapping animals on $c$ occasions. Capture frequencies, denoted by $x_i$ for $i = 1, \ldots, c$, are then computed. Each $x_i$ represents the number of animals captured exactly $i$ times, while $x_0$ is the number of animals never trapped. Accordingly, $d = \sum_{i=1}^{c} x_i$ is the number of individuals seen during the study. Thus, $N = d + x_0$ is equal to the size of the animal population.

The animal population size estimation problem is mapped to our problem as follows: $N$ is $\text{COUNT}(\pi_X(E))$; $x_i$ is the number of tuples which appear exactly $i$ times in the sample; $x_0$ is the number of distinct tuples which are not seen in the sample. As an analogy for $c$ in our problem, we could theoretically have taken $c$ as the size of the original sample (yielding many zero values for the $x_i$'s). Equivalently, for convenience, we took $c$ as the highest appearing frequency of sample elements. Therefore, in our problem: $d = \sum_{i=1}^{c} x_i$ equals the number of distinct tuples seen in the sample, and $N = d + x_0$ represents the number of distinct tuples in the population (i.e., $\text{COUNT}(\pi_X(E))$).

The $k^{th}$ order Jackknife estimator $\hat{J}_k$ of $J_k$ is defined as

$$\hat{J}_k = d + \sum_{i=1}^{k} a_{ik} x_i$$  

(5.3)

where $a_{ik} = 0$ for $i > k$. The first three orders of the Jackknife estimators are
\[ \hat{J}_1 = d + \frac{(c-1)}{c} x_1 \]  
(5.4) 
\[ \hat{J}_2 = d + \frac{(2c-3)}{c} x_1 - \frac{(c-2)^2}{c(c-1)} x_2 \]  
(5.5) 
\[ \hat{J}_3 = d + \frac{(3c-6)}{c} x_1 - \frac{(3c^2 - 15c + 19)}{c(c-1)} x_2 + \frac{(c-3)^3}{c(c-1)(c-2)} x_3 \]  
(5.6) 

From Burnham and Overton [BuOv 79], for any fixed value of \( c \), the higher order Jackknife estimators (i.e., increasing \( k \) values) lead to greater bias reduction, but at the cost of increased sampling variance. Conversely, for any fixed value of \( k \), as \( c \) increases, \( \hat{J}_k \) is a consistent estimator of \( J \) asymptotically, and its sampling variance will decrease as \( c \) increases.

Clearly, one should choose [BuOv 79] a higher order Jackknife and a greater \( c \) if one wants to obtain a good bias reduction with small variance. But, Burnham and Overton suggest [BuOv 79] that there will generally be a minimum mean square error (MSE) at a small value of \( k \); that is, there is a “best” \( \hat{J}_k \) at a small order of Jackknife estimators. Experimental results show that the minimum MSE is usually achieved at \( k = 1.2 \) or 3.

A selection procedure was also presented by Burnham and Overton [BuOv 79] to select the “best” \( \hat{J}_k \) among \( k = 1, 2, \) or 3. First, the null hypothesis is tested to investigate if there are any differences between the expected values of \( \hat{J}_1 \) and \( \hat{J}_2 \), i.e., test \( H_{01} \): \( \text{E}(\hat{J}_2 - \hat{J}_1) = 0 \) versus the alternative \( H_{a1} \): \( \text{E}(\hat{J}_2 - \hat{J}_1) \neq 0 \). If \( H_{01} \) is accepted, the interpretation is that the reduction in the absolute bias achieved by using \( \hat{J}_2 \) rather than \( \hat{J}_1 \) is small relative to the variance of \( \hat{J}_2 \). Since \( \hat{J}_1 \) has a smaller variance than \( \hat{J}_2 \), \( \hat{J}_1 \) is more preferable than \( \hat{J}_2 \), and it should be taken as the estimator.

The rejection of \( H_{01} \) implies a significant decrease in absolute bias relative to the increased variance of \( \hat{J}_2 \). Therefore, \( \hat{J}_2 \) should be chosen instead of \( \hat{J}_1 \).
Algorithm \textsc{Jackknife-Estimator} \((z)\)

\textbf{Input} : An array \(x\) where each element \(x_i\) represents the number of distinct tuples appearing exactly \(i\) times in the sample.

\textbf{Output} : Jackknife estimator \(\hat{J}\) of the number of distinct tuples.

\begin{algorithm*}
\begin{algorithmic}
\STATE Compute \(c\) which is the highest appearing frequency in the sample:
\quad \{i.e., \(x_i = 0\) for \(i > c\}\).
\STATE \(d := \sum_{i=1}^{c} x_i\);
\STATE \(J_1 := d + \frac{(c-1)}{c} x_1\);
\STATE \(J_2 := d + \frac{(2c-3)}{c} x_1 - \frac{(c-2)^2}{c^2-1} x_2\);
\STATE \(J_3 := d + \frac{(2c-6)}{c} x_1 - \frac{3c^2-15c+19}{c^2-1} x_2 + \frac{(c-3)^3}{c^2-1}(c-2) x_3\);
\FOR {\(k := 1\) to \(2\)}
\STATE \textbf{if} \textsc{Test-Null-Hypothesis} = \textit{"Accept"}
\STATE \{\(E(J_{k+1} - J_k) = 0\}\)
\STATE \textbf{then} \(J_E := J_k\) and exit:
\STATE \textbf{end}
\STATE \(J(E) := J_3\);
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm*}

Figure 5.1: Algorithm for Computing the Jackknife Estimator \(\hat{J}\).

However, the absolute bias may be reduced even further. Before choosing \(\hat{J}_2\), \(\hat{J}_2\) needs to be tested against \(\hat{J}_3\). If this test results in rejection, the selection procedure continues. The selected \(\hat{J}_i\) for \(i=1, 2,\) or \(3\), is called the Jackknife estimator. Appendix A describes the general selection procedure for choosing \(\hat{J}\). Algorithm \textsc{Jackknife-Estimator} presented in Figure 5.1 shows the computation of the Jackknife estimator.

\subsection{Chao’s Estimator}

Chao proposes a nonparametric method to estimate the number of classes in a population [Chao 84]. The method was intentionally devised for those cases where most of the information is concentrated on the first two occupancy numbers, i.e., \(x_1\) and \(x_2\). Chao’s estimator, denoted by \(\hat{C}\), is formulated as

\begin{equation}
\hat{C} = d + \frac{x_1^2}{2x_2}
\end{equation} (5.7)
Algorithm CHAO-ESTIMATOR($x$)

**Input**: An array $x$ where each its element $x_i$, $1 \leq i \leq c$, represents the number of tuples appearing exactly $i$ times in the sample.

**Output**: Chao's estimator $\hat{C}$ of the number of distinct tuples.

**begin**

Compute $c$ which is the highest appearing frequency in the sample:

$$d := \sum_{i=1}^{c} x_i;$$

if $x_2 \neq 0$ then

$$\hat{C} := d + \frac{x_2^2}{2x_2};$$

else

Print ("error");

**end**

Figure 5.2: Algorithm for Computing the Chao's Estimator $\hat{C}$.

where $d$, $x_1$, and $x_2$ are as defined in Section 5.2.2. Clearly, Chao's estimator is easier to compute compared with Jackknife estimators. Comparison studies have shown that Chao's estimator has a better performance than the Jackknife estimator when $(d, x_1, x_2)$ carry most of the information [Chao 84]. However, the estimator as formulated can easily give an infinite value when $x_2$ is equal to 0. The algorithm CHAO-ESTIMATOR for computing the Chao's estimator is presented in Figure 5.2.

5.3 Estimating $\text{SUM}_A(\pi_X(R))$ Using Double Sampling

In this section, we present the use of double sampling technique to estimate $\text{SUM}_A(\pi_X(R))$. Note that the aggregate function $\text{SUM}_A(R)$ has the relation $R$ as input, and produces a single value which is the sum of all the $A$ attribute values in all tuples of $R$.

Consider the query $\text{SUM}_A(\pi_X(R))$, where $A \subseteq X$. We would like to obtain a sample of tuples from $R$, where the inclusion probability of each distinct attribute $A$ value in the sample is proportional to its frequency of occurrence in $\pi_X(R)$. For this purpose, we use the acceptance/rejection sampling method, which is a double sampling technique. This sampling technique was used by Olken and Rotem [OlkR 86] to estimate the cost of disk accesses.
in auditing and statistical analysis of large databases. The main steps of acceptance/rejection sampling are as follows.

1. Draw a reasonably large (e.g., 10%) simple random sample from \( R \). Each tuple \( t_i \) in \( \pi_X(R) \) has the inclusion probability of \( \frac{N_t}{N} \), where \( N_t \) is the number of tuples in \( R \) with attribute \( X \) value of \( x_i \), and \( N \) is the number of tuples in \( R \).

2. Estimate the frequency distribution of each of the distinct tuples in \( \pi_X(R) \) (i.e., obtain an estimate \( \hat{N}_t \) for \( N_t \) for each \( t_i \) in \( \pi_X(R) \)).

3. Sample again from \( R \) with the acceptance probability \( p_i = \frac{1}{\hat{N}_t} \) for a tuple of \( R \) whose \( X \) attribute values are \( x_i \).

Hence, the inclusion probability for each distinct \( t_i \) in \( \pi_X(R) \) is \( p = \left( \frac{N_t}{N} \right) \times \left( \frac{1}{\hat{N}_t} \right) = \frac{1}{N} \). Moreover, the inclusion probability for each distinct value \( a \) of attribute \( A \) is \( \sum_{i=1}^{N_a} \frac{1}{N} = \frac{N_a}{N} \) where \( N_a \) is the number of distinct tuples in \( \pi_X(R) \) with attribute \( A \) value of \( a \).

There is one potential problem with the above acceptance/rejection sampling approach. It is possible to sample a tuple from \( R \) in step 3 whose \( X \) components do not match the \( X \) components of any tuple sampled in step 1. Rejecting such tuples may effect the randomness of the sample, and may result in increased sample construction time, which may not be desirable. Another alternative is to draw tuples in step 3 from the sample of step 1 (instead of \( R \)) which eliminates such a problem. Such an alternative approach makes the assumption that the first sample includes most of the distinct \( X \) values in \( R \).

Let \( s \) denote the tuples that are sampled (and accepted) from \( R \) in step 3 of the acceptance/rejection sampling. Let \( N_s \) denote the number of tuples of \( s \). Then the estimator for \( \text{SUM}_A(\pi_X(R)) \) we use is

\[
\text{SUM}_A(\pi_X(R)) = \frac{\text{SUM}_A(\pi_X(s)) \times \text{COUNT}_A(\pi_X(R))}{N_s}
\] (5.8)
where $\text{COUNT}_A(\pi_X(R))$ is a nonparametric estimator from section 5.2 (such as the Jackknife estimator) for the total number of distinct tuples of $\pi_X(R)$. Also note that $\text{SUM}_A(\pi_X(s))/N$, in the above equation is an unbiased estimator for $\text{AVG}_A(\pi_X(R))$, which has good performance even at very low sampling fractions. For example, at a sampling fraction of 5%, the relative error of $\text{AVG}_A(\pi_X(R))$ is less than 2%.

In section 5.5, we generalize the above approaches, and present an algorithm for estimating $\text{SUM}_A(E)$ where $E$ is an arbitrary RA expression.

5.4 New Sampling Plans

In this section, we discuss two different sampling techniques, namely, systematic and stratified sampling.

5.4.1 Sampling Plan Based on Systematic Sampling

Suppose we would like to draw a systematic sample from a population of size $N$. Assume that we have an ordering among the population elements. The systematic sampling procedure takes a unit at random from the first $k$ elements and every $k^{th}$ unit from there on. The performance of systematic sampling depends on the properties of the population [Coch 77], which may greatly improve the estimate for some populations, and it may deteriorate the estimate for others. Two main types of populations are Random Order Populations and Ordered Populations.

We obtain a systematic sample of size $m$ from a relation $R$ with $N$ tuples ($N >> m$) as follows. We take a tuple out at random from the first cluster of size $c$ and every $k^{th}$ thereafter. The period $k$ of relation $R$ is $\lceil N/m \rceil$. The first tuple in the sample is determined by drawing at random from the first $c$
tuples of relation \( R \), where \( c \) is
\[
c = \begin{cases} 
  k & \text{if } k = \frac{N}{m} \\
  k + (N \mod m) & \text{if } k \neq \frac{N}{m}
\end{cases}
\] (5.9)

### 5.4.2 Sampling Plan Based on Stratified Random Sampling

In stratified sampling, the population of \( N \) units is broken down into \( L \) non-overlapping subpopulations of \( N_1, N_2, \ldots, N_L \) units, respectively. These subpopulations, called strata, are disjoint, i.e.,
\[
N_1 + N_2 + \cdots + N_L = N
\] (5.10)

It is known [Coch 77] that the variability or heterogeneity among the units of the population has an effect on the precision of an estimator. In addition to increasing the sample size, one possible way to obtain greater precision is to divide the population into several strata each of which is more homogeneous than the whole population and draw a sample from each of the stratum.

We use stratified sampling when a relation is already stratified (e.g., employment relation is stratified and stored as research-personnel and development-personnel). That is, we do not perform any stratification or post-stratification during estimator evaluation.

**Stratified Relation**: A stratified relation \( R \) is defined as a relation consisting of \( j, j > 1 \) non-overlapping subrelations (or strata) \( s^1, s^2, \ldots, s^j \) with the following properties: (1) \( R = s^1 \cup s^2 \cup \cdots \cup s^j \); (2) \( s^m \cap s^n = \emptyset, m \neq n \), \( 1 \leq m \leq j, 1 \leq n \leq j \)

Clearly, \( N = N^1 + N^2 + \cdots + N^j \), where \( N \) is the number of tuples of the stratified relation \( R \) and \( N^j \) is the number of tuples of the subrelation \( s^j \).
Our design utilizes the proportional allocation method: a sample from each stratum is allocated proportionally according to the size of the strata.

Consider an RA expression $E$ with $n$ operand relations $R_1, R_2, \ldots, R_k, r_{k+1}, r_{k+2}, \ldots, r_n$, where $R_i$, $1 \leq i \leq k$, are stratified relations and $r_j$, $k + 1 \leq j \leq n$, are non-stratified original relations. Suppose $s_i$ represents the number of subrelations of $R_i$. In the stratified random sampling design, a set of $d_i^m$ for $1 \leq m \leq s_i$, tuples is randomly drawn from each of the subrelations of $R_i$, for $1 \leq i \leq k$, and a random sample of $d_j$ tuples is obtained from each of $r_j$, for $k + 1 \leq j \leq n$. The number of sample elements constructed in the $n$-dimensional space $R_1 \cdots R_k r_{k+1} \cdots r_n$ is then

$$m = \left( \prod_{i=1}^{k} r_i \right) \times \left( \prod_{j=k+1}^{n} d_j \right)$$

(5.11)

5.5 Generic Estimators for AGG Queries

In this section, we present the general methodology for estimating aggregate queries $AGG(E)$, where $AGG \in \{SUM, COUNT, AVG\}$. The expression $E$ in the query is an arbitrary relational algebra expression containing union ($\cup$), difference ($-$), intersection ($\cap$), selection ($\sigma$), projection ($\pi$), and natural join ($\bowtie$) operators. The attribute $A$ is an output attribute of expression $E$.

5.5.1 Estimator for Select-Join-Intersection-Expressions

We first discuss the estimator for $AGG(E)$, where $E$ is an expression with arbitrarily many select, join, and intersection operators. which, for simplicity, we will call an SJI-expression from now on.

Suppose that $E$ has $n-1$ join operators with relations denoted by $R_1, R_2, \ldots, R_n$. Let $|R_i|$ denote the number of tuples in $R_i$. An SJI-expression with $n$ operand
relations is modeled as an \( n \)-dimensional "point space" \( R_1 R_2 \cdots R_n \). Given a
tuple \( t_i \in R_i, 1 \leq i \leq n \), if \( E(\{t_1\}, \{t_2\}, \cdots, \{t_n\}) \) produces an output tuple \( t \),
a "point" \( p(t_1, t_2, \cdots, t_n) \) in the space is assigned a value of 1 (else 0) in the
case of \( AGG = COUNT \), or the attribute \( A \) value of the tuple \( t \) (else 0) in
the case of \( AGG = SUM_A \). \( N = |R_1| \times |R_2| \times \cdots \times |R_n| \) is the total number
of points in the point space \( R_1 R_2 \cdots R_n \). Assume that the points in the point
space of \( E \) are represented by \( p_1, p_2, \cdots, p_N \). Let \( v_i \) denote the value of point
\( p_i \), \( V(E) = v_1 + v_2 + \cdots + v_N \). One can see that \( COUNT(R_1, R_2, \cdots, R_n) \) or
\( SUM(R_1, R_2, \cdots, R_n) \) is exactly \( V(E) \). An estimator \( \hat{V}(E) \) for \( V(E) \) based on
a simple random sample of \( m \) points from the space \( R_1 R_2 \cdots R_n \) is then
\[
\hat{V}(E) = N \frac{\sum_{i=1}^{m} v_i}{m}
\]
(5.12)

Intersection (\( \cap \)) is incorporated into the above approach by considering it
as a special case of the join operation. Furthermore, by taking into account
the qualification specified in the selection formula of the selection operator (\( \sigma \))
during the \( v_i \) value assignment process, the operator (\( \sigma \)) can also be incorpo-
rated into the methodology. Figure 5.3 gives the algorithm \texttt{ESTIMATE-SJI-
RANDOM(E)} where \( E \) is an arbitrary SJI-expression.

### 5.5.2 Estimator for Project-Select-Join-Intersection-Expressions

Let \( E \) be an expression with arbitrarily many project, select, join and
intersection operators, which we call a PSJI-expression. The one-to-one rela-
tionship between the points in the point space and the output tuples of \( E \) is
violated due to the duplication in the projected domain. Each group of dupli-
cates eventually produces a single tuple in the projected result. For \texttt{COUNT},
we use the previously discussed nonparametric estimators, i.e., the Goodman's,
the Jackknife and the Chao's estimators for estimating the number of distinct
Algorithm ESTIMATE-SJI-RANDOM(E, AGG, ATTR)
Input: An SJI-expression E, the aggregate function type AGG,
and the aggregation attribute ATTR (if any).
Output: An estimate AGG(E) of AGG(E).
begin
Obtain a simple random sample of m points p_i, 1 ≤ i ≤ m.
from the point space R_1 R_2 ⋯ R_n;
{Note that each p_i uniquely identifies the ordered n-tuple (t_1, t_2, ⋯, t_n)
such that t_j ∈ R_j, 1 ≤ j ≤ n}
case AGG of
  SUM: for each point p_i in the sample do
         if E({t_1}, {t_2}, ⋯, {t_n}) produces an output tuple t then
         v_i := t[ATTR] else v_i := 0;
  COUNT: for each point p_i in the sample do
           if E({t_1}, {t_2}, ⋯, {t_n}) produces an output tuple then
           v_i := 1 else v_i := 0;
endcase;
return ( \( \frac{N}{m} \) \sum_{i=1}^{m} v_i ); {N is the total number of points in R_1 R_2 ⋯ R_n}
end.

Figure 5.3: Simple Random Sampling-Based Estimator Evaluation Algorithm for AGG(E), where E is an SJI-expression.

groups in the point space. For SUM, we generalize the double sampling technique. The algorithm ESTIMATE-PSJI-RANDOM of Figure 5.4 implements the estimation methodology for PSJI-expressions.

5.5.3 General Algorithm for Estimating AGG(E)

We now incorporate difference (−) and union (∪) operators into an SJI-expression E, and call such an expression a DUSJI-expression. We apply the Principle of Inclusion and Exclusion and the following equalities

\[ AGG(R_1 ∪ R_2) = AGG(R_1) + AGG(R_2) - AGG(R_1 \cap R_2) \]

\[ AGG(R_1 - R_2) = AGG(R_1) - AGG(R_1 \cap R_2) \]

where AGG ∈ \{SUM, COUNT\}. Clearly, we can compute AGG(E_1 ∪ E_2) and AGG(E_1 - E_2) indirectly by making use of AGG(E_1), AGG(E_2), and AGG(E_1 ∩ E_2). Thus, to obtain an estimator for AGG(E), where E is a DUSJI-expression, we decompose AGG(E) into a set of subexpressions AGG(E_i)
Algorithm ESTIMATE-PSJI-RANDOM (E, AGG, ATTR)

*Input*: A PSJI-expression $E$, the aggregate function type AGG, and the aggregation attribute ATTR (if any).

*Output*: An estimate $AGG(E)$ of $AGG(E)$.

*Global var*: projector-estimator

begin

case AGG of

COUNT: begin

Obtain a simple random sample of $m$ points $p_i, 1 \leq i \leq m$.

from the point space $R_1 R_2 \cdots R_n$;

{Note that each $p_i$ uniquely identifies the ordered $n$-tuple $(t_1, t_2, \ldots, t_n)$ such that $t_j \in R_j, 1 \leq j \leq n$}

Initialize the multi-set MULTI-SET as empty;

for each $p_i$ if $E(\{t_1\}, \{t_2\}, \ldots, \{t_n\})$ produces an output tuple $t_0$

then add $t_0$ into MULTI-SET;

Scan MULTI-SET and compute $x_i, 1 \leq i \leq m$, which is the number of groups containing $i$ elements in MULTI-SET;

case project-estimator of

  Jackknife: return (JACKKNIFE-ESTIMATOR(x));

  Chao: return (CHAO-ESTIMATOR(x));

  Goodman: return (GOODMAN-ESTIMATOR(x));

eendcase;
end {COUNT};

SUM: begin

Obtain a simple random sample of $M$ points $p_i, 1 \leq i \leq M$.

from the point space $R_1 R_2 \cdots R_n$;  \{M is pre-chosen\}

Initialize an array $A$ as empty;

for each $p_i$ do

if $E(\{t_1\}, \{t_2\}, \ldots, \{t_n\})$ produces an output tuple $t_0$

then add $t_0$ into $A$; \{$A$ is unsorted\}

Scan $A$ and compute the frequency $f_u$, for each distinct value $u_i, 1 \leq i \leq M$ in $A$;

ACCEPTED-SUM := ACCEPTED-COUNT := 0;

while ACCEPTED-COUNT $< M$ do

begin

Sample randomly again from $A$ with the acceptance probability $\frac{1}{f_u}$ for a tuple $t$:

if $t$ is accepted then

begin

ACCEPTED-SUM := ACCEPTED-SUM + $t$[ATTR];

ACCEPTED-COUNT := ACCEPTED-COUNT + 1;

end;
end;

COUNT := ESTIMATE-PSJI-RANDOM (E, COUNT, ATTR);

return (COUNT x ACCEPTED-SUM/ACCEPTED-COUNT);
end {SUM};

eendcase;
end.

Figure 5.4: A Simple-Random-Sampling-Based Estimator Construction Algorithm for $AGG(E)$, where $E$ is a PSJI-expression.
combined by +’s and −’s, where $E_i$ does not contain ∪’s and −’s. We call this transformation $\text{TRANSFORM}(\text{AGG}(E))$. When the projection operator is added into $E$, the transformation still works if projections do not precede differences in $E$. Thus, for each $\text{AGG}(E_i)$, the expression $E_i$ is either an SJI or a PSJI-expression. The estimator for $\text{AGG}(E)$ is then determined as $\text{AGG}(E) = \sum_i(\pm)\text{AGG}(E_i)$.

Now we give the general algorithm for estimating $\text{AGG}(E)$. Algorithm $\text{ESTIMATE-RANDOM}$ of Figure 5.5 evaluates an estimator for $\text{AGG}(E)$ for any arbitrary RA expression $E$.

5.6 Experimental Results

The experimental work about the new estimators and sampling plans introduced in this chapter has been conducted by Tjahjana in his Masters thesis [Tjah 91]. Two types of input relations (with normal and uniform tuple distributions, separately) were used in the experiments. And eight factors, estimator, sampling fraction, sampling technique, distributions of attribute values, ordering of attribute values, selectivities of operations, and sizes of input relations, were used to test the performance of estimates. From the experimental results, we have the following observations:

1. At low sampling fractions, both the Chao’s estimator and Jackknife estimators perform significantly better than the Goodman’s estimator, especially when tuples are heavily duplicated (low selectivities). But at high selectivities, in either case (normal and uniform), the performances of the Chao’s and Jackknife estimators are bad. The Jackknife and Chao’s estimators perform better when attributes of input relations are uniformly

\footnote{We revise the expression and the estimator further when projections precede differences. See [HoOT 88] for details.}
Algorithm ESTIMATE-RANDOM(E, AGG, ATTR)

Input: An arbitrary expression $E$; the aggregate function type AGG,
and the aggregation attribute ATTR (if any).

Output: An estimate $AGG(E)$ of $AGG(E)$, where $AGG \in \{\text{SUM, COUNT, AVG}\}$.

begin
  case AGG of
    COUNT: begin
      TRANSFORM(COUNT(E));
      \{Now, E is of the form $E_1 \theta E_2 \ldots \theta E_m$, \( \theta \in \{\cup, -\} \) and $E_j$ is a SJI or PSJI exp.\}
      for each $E_j$ do
        case $E_j$ of
          SJI-exp: $COUNT(E_j) :=$ ESTIMATE-SJI-RANDOM($E_j$, COUNT, -);
          PSJI-exp: $COUNT(E_j) :=$ ESTIMATE-PSJI-RANDOM($E_j$, COUNT, -);
        endcase;
        $COUNT(E) := \sum_j(\pm)COUNT(E_j)$;
      end {COUNT};
    end
    SUM: begin
      TRANSFORM(SUM(E));
      for each $E_j$ do
        case $E_j$ of
          SJI-exp: $SUM(E_j) :=$ ESTIMATE-SJI-RANDOM($E_j$, SUM, ATTR);
          PSJI-exp: $SUM(E_j) :=$ ESTIMATE-PSJI-RANDOM($E_j$, SUM, ATTR);
        endcase;
        $SUM(E) := \sum_j(\pm)SUM(E_j)$;
      end {SUM};
    end
    AVG: begin
      $S :=$ ESTIMATE-RANDOM(E, SUM, ATTR);
      $C :=$ ESTIMATE-RANDOM(E, COUNT, -);
      $AVG(E) := \frac{S}{C}$;
      end
  endcase
end.

Figure 5.5: A Simple-Random-Sampling-Based Estimator Construction and Evaluation Algorithm for $AGG(E)$, Where $E$ is an Arbitrary RA-expression.
distributed. Especially, Chao's estimator behaves like a consistent one.

2. the $\tilde{D}$ estimator based on the double sampling performs better than simple random sampling, particularly when the input data is uniformly distributed.

3. With systematic sampling on ordered input data, relative errors converge more rapidly compared with simple random sampling.

4. As the subrelations become more homogeneous, estimates with lower relative errors are obtained by using the stratified random sampling as opposed to other sampling methods.
Chapter 6

Enforcing Aggregate Function Constraints, PCF Constraints and Count Proportionality Constraints

Comparatively, enforcing aggregate function constraints, PCF constraints and count proportionality constraints are easier than enforcing time constraints and error constraints. In this chapter, we summarize the enforcement issues of aggregate function constraints. PCF constraints and count proportionality constraints.

6.1 Enforcing Aggregate Function Constraints

Unlike the enforcement of time constraints in which only query reduction strategy is used, we can use query reduction, query enlargement and query enlargement-reduction strategies in enforcing aggregate function constraints. Either the sampling protocol or the superset/subset chain protocol can be used to enforce the aggregate function constraints.

When arithmetic comparison operators used in a given aggregate function constraint do not contain "$\neq\$", we use the superset/subset chain protocol to enforce the aggregate function constraints. The general strategy for enforcing the aggregate function constraints using the superset/subset chain protocol may be as follows.

1. Check whether the original query satisfies the given aggregate function constraints;

2. If YES, stop. Otherwise, decide the modification strategy to be used...
(query reduction, query enlargement or query enlargement-reduction);

3. Assign a proper modification operator (\(\uparrow\), \(\downarrow\)) to some of the base relations according to the chosen modification strategy;

4. Construct a fragment list as follows:
   
   (a) For a relation with a \(\uparrow\) operator, replace the relation by one of its superset from its superset chain;
   
   (b) For a relation with a \(\downarrow\) operator, replace the relation by one of its subset from its subset chain;

5. Check the satisfaction of the modified query to the given constraints;

6. Repeatedly perform steps 3 and 5 until a qualified modified query is obtained, or report that the given aggregate function constraints can not be satisfied.

We check the satisfaction of given constraints for a (modified) query by either fully evaluating the query or estimating the query with a sample of its input. Fully evaluating the query and checking for the constraints takes a long time; estimating the query with a sample takes a shorter time. However, the accuracy of the estimated result is dependent on the related probabilistic and statistical factors such as tuple probability distributions, sample size and estimators used.

Choosing a modification strategy is dependent on the check result. For example, assume that a given query \(Q\) has the aggregate function constraint \(\text{COUNT}(Q) > 100\). If \(|Q|\) is less than 100 then, a query enlargement strategy is chosen when \(Q\) is a monotone query; query enlargement-reduction strategy is chosen when \(Q\) is a nonmonotone query. In the second case, we enlarge those relations in the minuend subexpressions of \(Q\) and reduce those relations in the subtrahend subexpressions of \(Q\).
When an arithmetic comparison operator "=" is used in a given aggregate function constraint, the sampling protocol may be used, i.e., replace a base relation by a sample drawn from the relation or its supersets. This is because using supersets/subsets of base relations usually cannot make the input or (output) of a modified query satisfy an exact aggregate value. e.g., \( COUNT(Q) = 100 \).

The aggregate function constraint enforcement strategy using the sampling protocol can be designed similar to that using the superset/subset chain protocol. They are only different in step 4, i.e., instead of modifying the query by using a fragment list, we use a set of samples drawn from the relations or their supersets.

6.2 Enforcing PCF Constraints

Unlike time constraints and aggregate constraints, PCF constraints are not independently specifiable constraints. Therefore, PCF constraints must be used accompanying some independently specifiable constraints. For a given query \( Q \) and a set of constraints \( C \), suppose a replacement list \( l \) of fragments is chosen such that \( Q'_{\text{m}} \) satisfies the independently specifiable constraints in the given constraint set \( C \). The general enforcement strategy for PCF constraints is given below.

1. Check whether the original query satisfy the independently specifiable constraints in the given constraint set \( C \);
2. If YES, stop. Otherwise, decide modification strategy to be used (query reduction, query enlargement or query enlargement-reduction);
3. Assign a proper modification operator (\( \uparrow, \downarrow \)) to some of the base relations according to the chosen modification strategy;
4. Construct a replacement list as follows:

For each $PCF_x(f, g)$ in the given PCF constraints,

(a) if $x = 'equal'$, then choose $f'$ for replacing $f$ and $g'$ for replacing $g$
    from the corresponding superset/subset chains such that $PCF(f') \equiv
    PCF(g')$;

(b) if $x = 'cover'$, then choose $f'$ for replacing $f$ and $g'$ for replacing $g$
    from the corresponding superset/subset chains such that $PCF(f') \wedge
    PCF(g') \equiv PCF(f')$;

(c) if $x = 'exclude'$, then choose $f'$ for replacing $f$ and $g'$ for replacing $g$
    from the corresponding superset/subset chains such that $PCF(f') \wedge
    PCF(g') \equiv \phi$;

5. Check the satisfaction of the modified query for the given independently
    specifiable constraints in $C$;

6. Repeatedly perform steps 2 and 3 until obtaining a qualified modified
    query or report that the given constraints can not be satisfied.

Since the PCF constraints are not independently specifiable, after con-
structing a new replacement list which makes the modified query satisfy the
given PCF constraints, we need to check whether the new constructed replace-
ment list still makes the modified query satisfy the given independently spec-
ifiable constraints in $C$. If the new constructed replacement list does not make
the modified query satisfy the given independently specifiable constraints, we
need to (repeatedly) adjust the replacement list until obtaining a qualified
modified query or report that the given constraints can not be satisfied.
6.3 Enforcing Count Proportionality Constraints

In order to enforce count proportionality constraints, we first need to define a way of checking for the validity of count proportionality constraints.

**Definition (Count Proportional Modified Query)** Consider query $Q$ with the output relation $R$. Let $r$ be the output relation of a modified query $Q_m$ of $Q$. $Q_m$ is called a **count proportional modified query** of $Q$ if $r$ is a randomly distributed subset of $R$.

According to the definition of count proportional constraint, count proportional modified queries satisfy count proportional constraint. Two features of count proportional modified query (from the definition of count proportional modified query) are:

1. $r$ is subset of $R$, i.e., $r \subseteq R$.
2. the tuples in $r$ are randomly distributed in $R$.

Clearly, the count proportionality constraint can be enforced by using random sampling protocol. For a monotone query $Q$, we replace each of the base relations involved in $Q$ by a random sample drawn from the base relation with a fixed sampling ratio $\rho$. We call such a set of samples the **fully proportional sample set with sampling ratio $\rho$**. For a nonmonotone query $Q$, we replace each of the base relations which are not involved in a subtrahend subexpression by a random sample drawn from the base relation with a fixed sampling ratio $\rho$. However, we fully use the relations which are involved in a subtrahend subexpression of $Q$. We call such a set of samples and relations the **partially proportional sample set with sampling ratio $\rho$**.

**Remark 6.1** For a monotone query $Q$ and a fully proportional sample set $I$, the modified query $Q'_m$ is a count proportional query.
**Proof.** Straightforward. Q.E.D.

**Remark 6.2** For a nonmonotone query $Q$ and a partially proportional sample set $l$, the modified query $Q_m^l$ is a count proportional query.

**Proof.** Straightforward. Q.E.D.

Now, we give the strategy for enforcing the count proportionality constraint.

1. Check whether the original query satisfies the independently specifiable constraints in the given constraint set $C$;

2. If YES, stop. Otherwise, assign a $\downarrow$ operator to each of the base relations which are not involved in a subtrahend subexpression;

3. Choose a proper sampling ratio $\rho$;

4. Construct a $Q_m^l$ by drawing a fully proportional sample set with sampling ratio $\rho$ for each relation with $\downarrow$ operator;

5. Check the satisfaction of the modified query to the given independently specifiable constraints;

6. Repeatedly perform steps 3 and 5 until a qualified modified query is obtained or report that the given constraints cannot be satisfied.
Chapter 7

Conclusions and Future work

In this thesis, we have developed an automated query modification model for modifying database queries with constraints. Five types of query modification constraints, namely, time constraints, error constraints, aggregate function constraints, PCF constraints and count proportionality constraints are introduced. In order to have automatic query modification, we have introduced relation fragmentations, two query modification protocols, namely, the use of superset/subset chains use and the use of sampled data, and an isologous query modification mechanism. We have proposed strategies and algorithms for enforcing the five individual query modification constraints. We have designed iterative and incremental query evaluation techniques for processing queries with time constraints and the error detection and control for enforcing error constraints. In this thesis, we have also (i) presented new statistical estimators, the Jackknife estimator and the Chao’s estimator, for COUNT queries with projection, (ii) designed estimators using double sampling technique for SUM and AVG queries, and (iii) investigated new sampling plans based on systematic sampling and stratified sampling to increase the estimation accuracy.

Some of the techniques and algorithms proposed in this thesis have been implemented in CASE-DB which is a prototype relational database management system implemented at Case Western Reserve University.

There is much research work to be conducted in automated query modification. Below, we briefly discuss some possible future work related to this
thesis.

1. Enforcement Issues for Queries with Multiple Constraints

In this thesis, we have introduced ways to enforce individual types of modification constraints. As a future work, one may further investigate strategies and algorithms for enforcing multiple types of constraints specified for the same query. Some of the research topics in the multiple constraints case are given below.

When a user's query is associated with a set of constraints, say, \( \{c_1, c_2, \cdots, c_k\} \), there may be conflicts among these constraints in their semantics or in the selection of fragments. For instance, constraint \( c_i \) may require that relation \( R \) be reduced while constraint \( c_j \) may require that \( R \) be enlarged. First, the complexity and algorithm deleting such constraints needs to be studied. Second, we need to find a way to solve such conflicts. A possible method is to assign priorities to the constraints. When a constraint conflicts with another, the constraint with a higher priority can override the requirements of the constraint with a lower priority. Thus, conflict detection rules and priority specification rules are needed.

Sometimes, there is redundancy in the user-specified constraint set. For example, consider the query \( Q = R - S \). If the user specifies both the PCF equality constraint and the safety constraint in the constraint set with \( Q \) then the safety constraint is redundant because the PCF equality constraint guarantees the safety of any modified query \( Q_m \) of \( Q \). Thus redundancy detection and removal rules are also needed.
2. Optimization Problem — Finding the Most Informative Modified Query

For a query associated with a set of constraints, there may be more than one fragment replacement list which makes the modified query satisfy the given constraints. However, in this thesis, we only choose one of these fragment replacement lists to evaluate. It is expected that the modified query with the selected replacement list satisfies all of the given constraints to the maximum, i.e., provides as much user “expected” information as possible and as few user “unexpected” information as possible. We consider such a modified query as the most informative one to the user.

For a query \( Q \), a set of constraints and a set of available fragment replacement lists \( l_1, l_2, \ldots, l_n \), \( Q^l_m \) is said to be more informative than \( Q^l_n \) if \( Q^l_m \) “better” satisfies the given constraints than \( Q^l_n \). \( Q^l_m \) is said to be the most informative modified query if there exists no \( l_j \) such that \( Q^l_j \) is more informative than \( Q^l_m \). Ways of defining what “better” query satisfaction and of finding the most informative modified query need to be investigated.

3. Modifying Queries in Other Relational Languages

In this thesis, all algorithms are designed for queries written in the Relational Algebra (RA). For other equivalent relational database languages, e.g., safe Relational Calculus (RC) and nonrecursive Datalog, the queries expressible in Relational Algebra can also be expressed in these languages [Ullm 88]. When a given query is written in one of such languages, one can first transform the query into Relational Algebra, then enforce the given constraints. The modified query can be transformed back to its original language.

For languages whose functions and queries are not expressible in Relational Algebra, e.g., recursive Datalog queries, one needs to design different algo-
rithms to modify queries, and it is a possible future direction.

4. Parallel Query Evaluation Mechanism

One can increase the query evaluation speed further by introducing distributed and parallel evaluation mechanisms in iterative query evaluation. Parallel and distributed evaluation of a modified query can be supported by the query transformations introduced in chapter 3. Notice that most of the transformation formulas are disjoint unions of intersection, set difference, natural join and selection subexpressions.

As defined in chapter 3, for two arbitrary RA expressions $e_1$ and $e_2$ taking a disjoint union, $e_1 \cap e_2 = \emptyset$. Therefore, one can evaluate $e_1$ and $e_2$ independently then put their results together without checking duplication. This provides the possibility for parallel and distributed evaluation of the transformed queries. For example, suppose we have three independent processors in our local computer network with topologic structures shown in figure 7.1. At an iterative query evaluation step, processor A evaluates query $f_k \bowtie g_m$. Suppose $f_i \bowtie g_j$ has been evaluated at the last iteration. Using transformation t.n.j.1, processor A transforms $f_k \bowtie g_m$ into

$$Q(f_i, g_j) \uplus (f_i \bowtie g_{j+1,m}) \uplus (f_{i+1,k} \bowtie g_j) \uplus (f_{i+1,k} \bowtie g_{j+1,m})$$

Then, processor A distributes the subexpression $f_{i+1,k} \bowtie g_j$ to processor B and the subexpression $f_{i+1,k} \bowtie g_{j+1,m}$ to processor C. Processor A evaluates the subexpression $f_i \bowtie g_{j+1,m}$. Once processors B and C finish evaluating the forwarded subexpressions, they send the results back to processor A. Processor A adds the separately evaluated results to $Q(f_i, g_j)$ on the disk.

The parallel and distributed query evaluation approach introduced above can not only be used to evaluate modified queries, but also can be used to
Figure 7.1: The Topologic Structures of Local Computer Networks.

evaluate original queries (i.e., all tuples of each relation involved in the query must be processed) provided that the relations are partitioned into smaller fragments.

5. Using the Occurrence Probability Method to Estimate Errors for Arbitrary Queries

In chapter 4, we have introduced the occurrence probability method to estimate errors for simple set difference queries. We have indicated advantages and problems of using this method to estimate errors for arbitrary queries, i.e., the difficulty in deriving the occurrence probabilities of the fragmentation attribute values for those intermediate relations produced during query evaluation. One can try to investigate efficient ways to obtain occurrence probabilities of the fragmentation attribute values for intermediate relations.

6. Other Query Modification Methods

This thesis discusses isologous query modification. One can also investigate non-isologous query modification methods. Possible methods may include conjunction removal for the purpose of increasing the size of the output of a query, and disjunction removal for the purpose of decreasing the size of the output.
of a query.
Bibliography


[Olke 86] Olken, F., “Physical Database Support for Scientific and Statistical Databases”, Third Int. Scientific and Statistical Databases Workshop,
1986.


[StZa 88] J. Stankovic, W. Zhao, "On Real-Time Transactions", *ACM SIG*


Appendix

A.1. The general selection procedure for choosing $\hat{J}$ developed by Burnham and Overton is as follows:

"Test the null hypotheses $H_{ok}$: $E(\hat{J}_{k+1} - \hat{J}_k) = 0$ versus $H_{ak}$: $E(\hat{J}_{k+1} - \hat{J}_k) \neq 0$ sequentially for $k \leq 3$, and choose $\hat{J} = \hat{J}_k$ such that $H_{ok}$ is the first null hypotheses not rejected. The actual test of $H_{ok}$ is conditional on $d$, and is based on the fact that $\hat{J}_{k+1} - \hat{J}_k = \sum_{i=1}^{c} b_i x_i$ for the constants $b_i = a_{i,k+1} - a_{i,k}$. Given the null hypothesis $H_{ok}$, the test statistic

$$T_k = \frac{\hat{J}_{k+1} - \hat{J}_k}{[\text{var}(\hat{J}_{k+1} - \hat{J}_k)|d|]^{1/2}}$$

(8.1)

has approximately a standard normal distribution and

$$\text{var}(\hat{J}_{k+1} - \hat{J}_k|d) = \frac{d}{d-1} \left[ \sum_{i=1}^{c} (b_i)^2 x_i - \frac{(\hat{J}_{k+1} - \hat{J}_k)^2}{d} \right].$$

(8.2)

Because this test is conditional upon $d$, it does not depend upon the unknown value of $N."$ [BuOv 78]

Rejection of $H_{ok}$ is indicated by large $|T_K|$. Let $P_k$ be the probability of a value of $|T_k|$ larger than the observed value. $P_k$ can be determined by

$$P_k = 2[1 - \int_{-\infty}^{\left|T_k\right|} \frac{1}{\sqrt{2\pi}} e^{-u^2/2} du]$$

(8.3)

$H_{ok}$ is rejected when the value of $P_k$ is less than 0.05. We give an algorithm for accepting or rejecting the hypothesis $H_{ok}$ in figure 8.1.

Using the hypothesis test algorithm, we are now able to select a Jackknife estimator among $\hat{J}_k$'s, for $k = 1, 2, 3$. Algorithm JACKKNIFE-ESTIMATOR presented in figure 5.1 shows the computation of the Jackknife estimator. The
Algorithm TEST-NULL-HYPOTHESIS
{TEST-NULL-HYPOTHESIS tests the null hypothesis $H_{0k}$}

Input: $c, d, k, a_{i,k}, a_{i,k+1}, x_i, i = 1, \ldots, c.$

Output: "Accept" or "Reject".

\begin{align*}
T_k & := \frac{\sum_{i=1}^{c}(a_{i,k+1}-a_{i,k})x_i}{\sum_{i=1}^{c}(a_{i,k+1}-a_{i,k})^2} \\
P_k & := 2 \left( 1 - \int_{T_k}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-u^2/2} du \right) \\
\end{align*}

if $P_k \geq 0.05$

then return "Accept";

else return "Reject";

end

Figure 8.1: Algorithm for Testing the Null Hypothesis $H_{0k}$ of the Jackknife Estimator.

Estimator for the sampling variance of $\hat{J}_k$ given by Burnham and Overton [BuOv 79] is

\begin{equation}
\text{var}(\hat{J}_k) = \sum_{i=1}^{c} (a_{i,k})^2 x_i - \hat{J}_k 
\end{equation} (8.4)