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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

University of Alberta

Distributed Query Scheduling in The Context of DIOM: An Experiment

by

Kirill Richine ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 1997

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-21204-1

Canada

University of Alberta

Library Release Form

Name of Author: Kirill Richine

Title of Thesis: Distributed Query Scheduling in The Context of DIOM: An Experiment

Degree: Master of Science

Year this Degree Granted: 1997

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Kirill Richine
59 Sierra Vista Circle
Calgary, Alberta
Canada, T3H 3A4

Date: 17 April 1997

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Distributed Query Scheduling in The Context of DIOM: An Experiment** submitted by Richine Kirill V. in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .
Dr. L. Liu

. . . .
Dr. F. Lau

. . . .
Dr. M.T. Özsu

. . . .
Dr. X. Li

Date: 17 April 1997

Abstract

One of the key issues for query processing in distributed open environments is the query scheduling problem. Given a user query, after we know that there are n sources that are relevant to the answer of this query, the first issue we need to address is how to decompose the query into n subqueries, each targeting at one single source. The second issue is how to synchronize these n subqueries in the presence of inter-site joins. The third issue is how to package and assemble the results from n information sources according to the original query posed by the user.

In this thesis, we discuss the first two issues in the context of DIOM, a distributed and interoperable query mediation system [12]. Our main contribution is the systematic development of the two-tier distributed query scheduling framework that produces the relatively best query schedule according to the given combination of cost parameters, including the total query response time, the local query processing cost, and the communication cost. Our main focus is on queries that contain inter-site joins. The first tier is called the heuristic-driven query processing, which produces a heuristic-based optimal schedule. The second tier is referred to as the cost-driven query processing, which generates a cost-based optimal schedule. We implement a subset of our query scheduling algorithms in Java accessible from any Java-compliant GUI viewer such as Netscape 3.0. The URL for the demo is `ugweb.cs.ualberta.ca/~diom/query/EQ.html`. The most interesting features of our Java implementation is the functionality to allow users to trace the query scheduling process through trace program interface and the trace logs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Motivation

Over the last few years there has been a drastic increase in exchange of electronic information. The Internet connection as well as the browsing tools, originally designed for specialized needs of scientific community, have become an expected component of an average computer system. More and more information, previously accessible only via conventional means, become available through the world-wide-web. For example, many service requests, ranging from booking an airplane ticket and making a hotel reservation half-way across the world to career planning and banking, can now be done from one's home computer connected to the world-wide networks called Internet. The amount and diversity of the information available on-line is astounding. One can spend days just browsing and still not be able to even keep up with the megabytes of the new Internet resources that become available every day.

On the one hand, the sheer amount and diversity of the available information have increased the opportunities to get the most authentic and complete information on any topic of our interest. On the other hand, our ability to obtain only the relevant information among the large collection of available resources is limited by the time we can afford to spend surfing on the Internet.

A variety of search engines available on the world-wide web as well as some Internet search utilities try to address this problem. However, most of the search engines and utilities available today use keyword-based search - the user supplies a set of keywords that he or she thinks are specific to the information being sought. The keyword-based search uses partial string matching techniques and typically selects the repositories that match the keyword in their title or some content description fields.

Needless to say, this type of search is quite primitive. It is not sufficient for most of the structured information sources because it can not provide adequate support for sophisticated queries such as those that require inter-site joins. For example, if we are interested in purchasing a Toyota car of 1997 model, what we would like to do first is to look into the consumer reviews for such car sales. Very often, we can find the car reviews from the consumers' service information sources or local AAA offices, but need to obtain information about the 1997 car model and current sale price from car dealers. Thus, inter-site joins are required. There are two key difficulties in processing inter-site joins in distributed open environments such as Internet: (1) Given a user query and a growing collection of information sources, how to reduce the search space of answering a query to those information sources that are relevant to the query answer; and (2) given a set of relevant information sources, how to find a relatively efficient query execution schedule (plan) that has the quickest response time and/or the lowest total processing cost. This thesis will address the second issue by providing a theoretical analysis of the two-tier query scheduling architecture, and by applying and extending the heuristic-based and the cost-based query optimization techniques used in relational database management systems [18] and in distributed database systems [1, 5, 16].

1.1.1 Query Optimization in Relational Database Systems

In a centralized relational database system, given a query, there are a variety of methods for computing the answer. Each way of expressing the query "suggests" a strategy for finding the answer. However, we do not expect the users to write their queries in a way that suggests the most efficient strategy. Thus, it becomes the responsibility of the system to transform the query as entered by the user into an equivalent query which can be computed more efficiently. This "optimizing" or, more accurately, improving of the strategy for processing a query is called query optimization. There is a close analogy between code optimization by a compiler and query optimization by a database system.

Query optimization is an important issue in any database system since the difference between a good strategy and a bad strategy is often substantial, and may be several orders of magnitude.

Before query processing can begin, the system must translate the query into a usable form. A language like SQL is suitable for human use, but not suited to be the system's internal representation of a query. A more useful internal representation is the one based on the relational algebra, which indicates the order of evaluating operations. The first action the relational system must take on a query is to translate it into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser replaces all references to the view name with the relational algebra expression in order to compute the view.

Once the query has been translated into an internal relational algebra form, the optimization process begins. The first phase of the optimization is done at the relational algebra level. Since for a given query, there are many equivalent and yet correct algebraic expressions in terms of algebraic rewriting rules. An attempt is made to find an expression that is equivalent to the given expression but more efficient to execute. The second phase involves the selection of a detailed strategy for processing the query. A choice must be made as to exactly how the query will be executed. The specific indices to use must be chosen. The order in which tuples are processed must be determined. The final choice of a strategy is based primarily on the number of disk accesses required.

1.1.2 Query Optimization in Distributed Database Systems

The query processing problem in distributed database systems is more difficult than in centralized ones, because a larger number of parameters affect the performance of distributed queries. The role of a distributed query processor is to map a high-level query on a distributed database (i.e., a database whose fragments may be stored in different physical locations) into a sequence of database operations on relational fragments. Then each database query operation on relational fragments in one location is translated to bear on local data. Finally the set of query operations must be extended with communication operations and optimized with respect to a cost function which typically refers to computing resources such as disk I/Os, CPUs, and communication networks.

Due to the critical role of communication parameter in the distributed query processing cost, relational algebra is augmented by operations such as semijoins for exchanging data between sites to reduce communication cost. In addition to the choice of ordering relational algebra operations, the distributed query processor must also select the best sites to process data, and possibly the way data should be transformed. This increases the solution space from which to choose the distributed execution strategy, making distributed query processing significantly more sophisticated. For a concrete introduction of distributed query processing, readers may refer to Özsu and Valduriez's book [16].

1.1.3 Open Interoperable Database Systems

Over the last two decades, there have been many research publications in the field of distributed databases and federated databases.

Traditional distributed database systems have a dedicated central site that keeps the schema of the database, including the fragmentation and location information schemas. This site coordinates the processing of all query requests coming from the users of the distributed database. The query is distributed among the local databases according to the fragmentation schema.

Federated database system differs from traditional distributed database systems in the sense that the component databases, of which the federated system consists, are autonomous, that is, they perform irrespective of whether they have been included into the federation. At the same time, the schema of all the components is still maintained at federation level. In other words, if a source database exercises its autonomy by changing its schema information, the federation's schema becomes inconsistent until either the source uploads the necessary schema updates to the federation or the federation downloads them.

The federated approach requires the source to send information to all the federations of which the source is a member whenever there is a change at the source. On the other hand, the federated database server must be ready to process the updates at all times, irrespective of whether they are relevant to the current task of the server. Both of the above conditions may become a serious predicament to a database's performance and extensibility.

Quite differently, in an open and interoperable system, such as the DIOM, a project for building scalable and extensible query mediation services [7, 21], DIOM server does not create and maintain an integrated view schema of all the sources' schemas. In contrast, it allows the consumers to request services through the domain-specific mediator. The interconnection of consumers to the relevant information producers is established dynamically at the query processing stage. Every information source registered with DIOM must have a wrapper set up, which acts as a DIOM agent to the information source.

The main difference between distributed or federated database systems and open interoperable database systems is the open world assumption. The distributed database systems and the federated database systems [19] are based on the closed-world assumption, whereas in open environments such as Internet, the information sources available on-line are changing in numbers, volume, contents and query capabilities dynamically. The distributed query scheduling system must not only deal with the selection of the best sites to process data, but also the dynamic increase of the solution space from which to choose the distributed execution schedule, due to the rapid growing number of information sources available on-line. Furthermore the statistic information that is accessible in distributed database systems may not be available in open environments due to the autonomy of individual information sources.

Thus the effectiveness of the distributed query scheduling is measured in terms of (1) the reduction in the number of candidate plans that are enumerated, (2) the discovery of a query execution schedule that is relatively efficient with respect to either the total cost that will be incurred in processing the query (i.e., the sum of all times incurred in processing the operations of the query at various sites and in inter-site communication), or the response time of the query (i.e., the time elapsed for executing the query) or a weighted combination of cost components. Since operations can be executed in parallel at different sites, the response time of a query may be significantly less than its total cost. For more detail see Chapter 3 of this thesis.

1.2 Scope and Organization of The Thesis

This thesis presents the design and implementation of a DIOM *Distributed Query Scheduling prototype*, namely *DQS* utility. The theoretical model and the architecture for query scheduling are developed based on the previous result of the DIOM [14, 12] project. This prototype illustrates the ability to automate the query scheduling process and the ability to allow experienced users to tune the query performance by adjusting the unit costs, the statistics information, and the number of sources accessed to answer the query.

The main contribution of this thesis project is the systematic development and implementation of the two-tier distributed query scheduling framework that produces the relatively best query schedule according to the given combination of various cost parameters, including the total query response time, the local query processing cost, and the communication cost. Our main focus is on queries that contain inter-site joins. The first tier is called heuristic-driven query processing, which produces

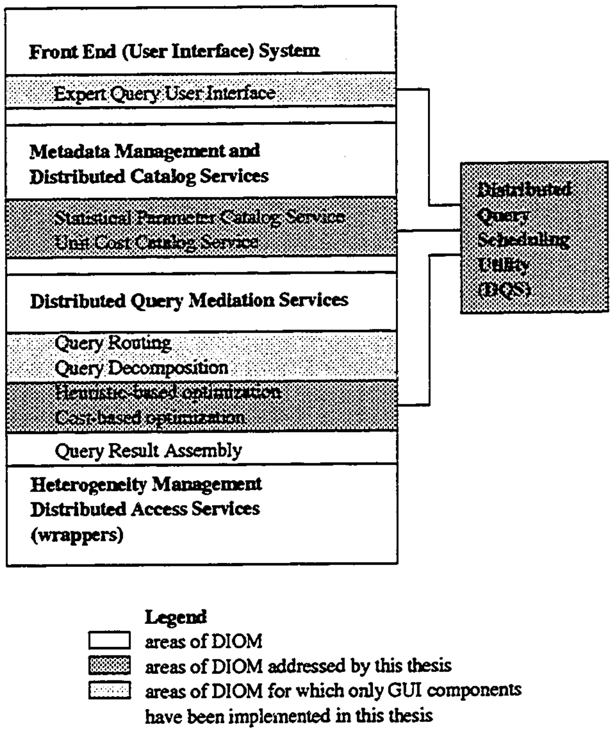

Front End (User Interface) System
Expert Query User Interface
Metadata Management and
Distributed Catalog Services
Statistical Parameter Catalog Service
Unit Cost Catalog Service
Distributed Query Mediation Services
Query Routing
Query Decomposition
Heuristic-based optimization
Cost-based optimization
Query Result Assembly
Heterogeneity Management
Distributed Access Services
(wrappers)
Distributed
Query
Scheduling
Utility
(DQS)
Legend
□ areas of DIOM
▩ areas of DIOM addressed by this thesis
▨ areas of DIOM for which only GUI components
have been implemented in this thesis

Figure 1-1: DIOM research subdivisions. Highlights show focus of this research.

a heuristic-based optimal schedule. The second tier is referred to as cost-driven query processing, which generates a cost-based optimal schedule. We implement a subset of our query scheduling algorithms in Java accessible from any Java-compliant GUI viewer such as Netscape 3.0. The most interesting features of our Java implementation is the functionality to allow users to trace the query scheduling process through trace program interface and the trace logs.

More concretely, the design and implementation of the *DQS* prototype system addresses the following issues:

- The design of a set of heuristics-driven query optimization strategies and the implementation of a subset of the proposed heuristics for obtaining a relatively optimal algebraic query expression.
- The design and implementation of the cost functions for realization of cost-driven query optimization in DIOM, which provides a weighted combination of the three key cost parameters that affect the distributed query processing performance: (1) the total cost of local processing incurred by all information sources involved in query execution, (2) the total communication cost of transferring the intermediate results of query execution among the sources, and (3) the total response time cost, which is the maximum combination of local and communication costs incurred at query execution.
- The implementation of interactive interface programs that allow experienced users to trace the query scheduling process and to tune the query processing performance by dynamically incorporating the changes in the unit cost involved in a query and the local statistic information.
- The methods for computing the intermediate query results as well as the strategies for selection of the best sites to process inter-site joins or inter-site unions so that the total cost is minimized.

Figure 1-1 shows the major areas of research for the DIOM project. The highlighted boxes represent the areas that the *DQS* prototype attempts to address.

The DQS prototype chooses to implement the user interface as a Java-based WWW application. The main technologies used in the prototype implementation of the DQS utility include Java for the main modules of the DQS and Oraperl, SQL/Plus, and Oracle DBMS for accessing statistics such as unit cost information and local statistics for the lower-level query scheduling process as well as the heuristics used for the DIOM high-level query scheduling process.

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the DIOM system architecture, and query mediation approach. A running example is used to illustrate the DIOM query mediation framework and the theme of this thesis project. Chapter 3 describes the two-tier approach we propose to handle the distributed query scheduling issue. The first part of this Chapter contains the detailed description of the proposed heuristic-based query processing methods, and the second part covers the intrinsics of cost-based query scheduling. Chapter 4 describes the object-oriented analysis, design and implementation issues of the *Distributed Query Scheduling (DQS) Utility* software package that has been developed to demonstrate the viability of the query scheduling algorithm proposed in Chapter 3. Chapter 5 concludes the thesis with a brief overview of the state of the art research, and a summary of the contributions of the *DQS* prototype, the issues addressed by the *DQS* prototype implementation, and a discussion of possible future improvement.

Chapter 2

DIOM Architecture

The Distributed Interoperable Object Model (DIOM) [12, 14] introduced the approach that explicitly defines the interfaces of an information consumer and an information producer, matching them dynamically to achieve interoperability in heterogeneous information systems with growing number of autonomous data sources as components. Although the DIOM interoperable architecture and its adaptive query mediation framework has been extensively covered in [10, 12, 14], to make this thesis self-contained, in this chapter we present a brief overview of the fundamental points of the DIOM project. Our attention is more concentrated on the parts of the DIOM previous research that are directly related to the investigation of the distributed query processing and optimization in DIOM.

2.1 Overview of the Information Mediation Architecture

In DIOM we view an advanced distributed information system as a dynamic interconnection between information consumers and information producers, instead of just functioning as a static data delivery system. Two issues that arise immediately are: (1) heterogeneity of information producers' data sources and information consumers' query requests, and (2) scalability of distributed query services in the presence of a growing number of information sources and the evolving requirements of both information producers and information consumers.

To deal with both the heterogeneity issues and the exponential growth of the available information, scalability and extensibility become very critical. DIOM proposes two independent but complementary strategies for achieving better scalability and higher extensibility in the development of distributed and interoperable query services:

- Use an incremental approach to construction and organization of information access through a network of domain-specific application mediators; and support the dynamic linking of mediators to heterogeneous information sources via repository-specific wrappers (see Figure 2-1).

- Provide a collection of facilities to allow information consumers to specify their queries in terms of how they would like their query results be received and represented, rather than relying on a global integrated view of all the participating information sources.

The first strategy guarantees a seamless incorporation of new information sources into the DIOM system. The second strategy allows the distributed query services to be developed as source-independent middleware services which establish the interconnection between consumers and a variety of information producers' sources at the query processing time. As a result, the addition of any new sources into the system only requires each new source to have a DIOM wrapper installed. The DIOM services can dynamically capture the newly available information sources and incorporate them into the distributed query scheduling process.

Figure 2-2 shows a concrete example network of information mediators collaborating through the DIOM mediator network architecture.

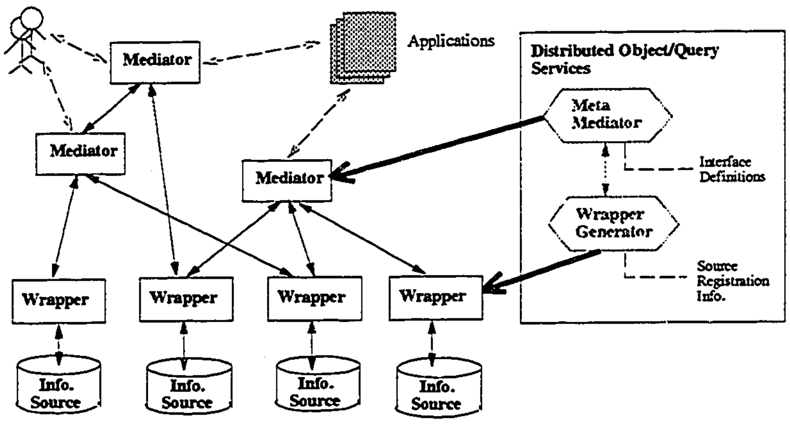


Figure 2-1: The cooperation architecture of network of mediators

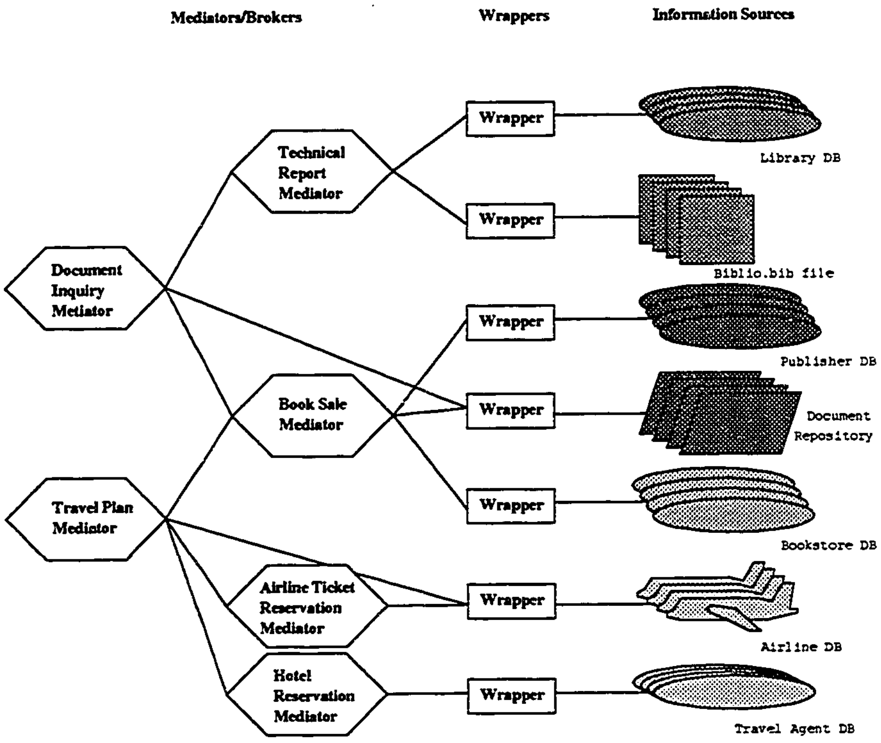


Figure 2-2: The DIOM system architecture: an example

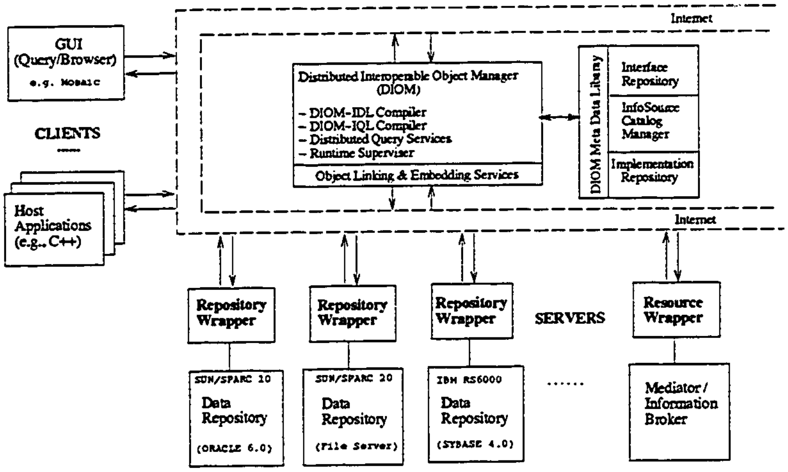


Figure 2-3: The DIOM meta mediator architecture

This network includes simple wrapper-based mediators such as a wrapper to BookStore data repository which only provides information about and access to the BookStore repository. This wrapper-based mediator is in turn used to construct a BookSale mediator. The BookSale mediator is again used to build both a document inquiry mediator and a travel plan mediator. This recursive construction and organization of information access have the following important features.

- The individual mediators can be independently built and maintained. Each specialized mediator represents a customized personal view of an information consumer over the large amount of information accessible from the growing number of information sources.
- These mediators can be generated automatically or semi-automatically by using the DIOM IDL/IQL interface specification language and the associated incremental compilation techniques. These features also make the DIOM architecture scale well to the large and growing number of information sources and to the varying information customization needs from diverse information consumers.

To build a network of specialized information mediators, an architecture for a single mediator (or so called meta mediator) is needed. Such meta mediator can be instantiated to build multiple specialized mediators. They are responsible for the following functions in DIOM system:

1. provide a uniform interface description language;
2. create a suite of interface composition meta operations;
3. provide the query processing services that are customizable and can be utilized by a number of application-specific mediators to facilitate the access to multiple heterogeneous information sources.

A typical DIOM mediator has a two-tier architecture and it offers services at both the mediator level and the wrapper level. Figure 2-3 illustrates the mediator object server, whose main responsibility is to provide the following services:

1. the coordination and information exchange between information consumer's domain usage model and the relevant information source models,
2. the distributed metadata library of mediated metadata and the correspondence to the metadata of information source models, and
3. the maintenance of the metadata catalog in the presence of changes or upon the arrival of new information sources.

Mediators in DIOM are application-specific. Each mediator consists of a consumer's domain model and many information producer's source models and are described in terms of the DIOM interface definition language (DIOM IDL) [13]. The consumer's domain model specifies the querying interests of the consumer and the preferred query result representation. The producer's source models describe the information sources in terms of DIOM internal object representation generated by the DIOM interface manager. The consumer's domain model and the information producer's source models constitute the general knowledge of a mediator and are used to determine how a consumer's information request is processed. The main task of the mediator sub-system is to utilize the metadata provided by both information consumers and information producers for efficient processing of distributed queries.

Each wrapper serves one information source. Wrappers are software modules that need to be built around the existing source in order to make it available to the network of mediators – this will turn the system into a DIOM local agent responsible for accessing that information source and obtaining the required data for answering the query. The main task of a wrapper is to control and facilitate external access to the information repositories by using the local metadata maintained in the implementation repository and the wrapper functions. Services provided by a wrapper include:

- translating a subquery in consumer's query expression into an information producer's query language expression,
- submitting the translated subquery to the target information source, and
- packaging the result of a subquery obtained from the source in terms of the objects understandable by the corresponding mediator.

Building a wrapper around the existing system turns the local system into a cooperative database agent.

The information sources at the bottom of the diagram in Figure 2-3 may be one of the following types of sources:

well structured: such as relational or object-oriented database management systems,

semi-structured: such as HTML files, bibliographical record files, other text-based records, or

nonstructured: such as technical papers or reports, ascii files, a collection of raw image files, etc.

Each information source is autonomous – it may make changes without approval from the mediators. If, however, an information source makes a change in its export schema, including logical structure, naming, or semantic constraints, then it must notify the DIOM object server.

2.2 The DIOM Distributed Query Scheduling Framework

2.2.1 An Overview of the Main Steps

The main task of a distributed query mediation manager is to coordinate the communication and distribution of the processing of information consumer's query requests among the root mediator and its component mediators or wrappers (recall Figure 2-1).

[14] has proposed the general procedure of a distributed query scheduling process in DIOM. It primarily consists of the following steps to process a user query submitted to the DIOM server:

1. query routing,
2. query decomposition,
3. parallel access plan generation,
4. subquery translation and execution, and
5. query result assembly.

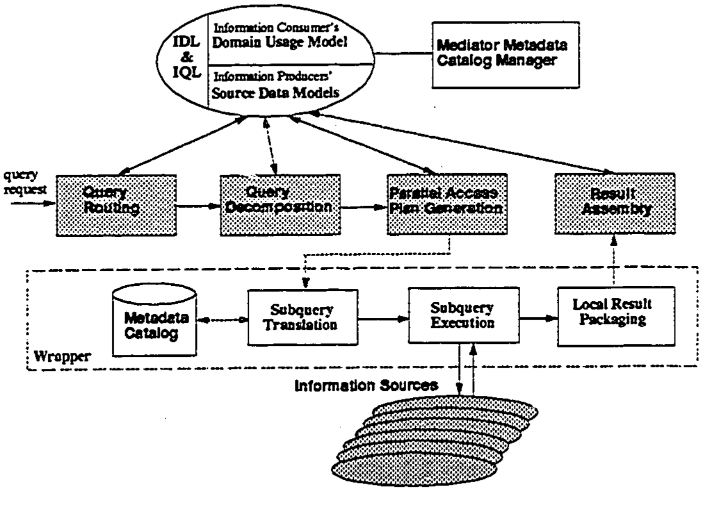


Figure 2-4: The distributed query scheduling framework in DIOM

Figure 2-4 presents a sketch of how a query is processed inside of the DIOM system.

Query routing is the first step. The main task of query routing is to select relevant information sources from available ones for answering the query. This is done by mapping the domain model terminology to the source model terminology, by eliminating null queries, which return empty results, and by transforming ambiguous queries into semantic-clean queries. Consumers' query profiles and producers' data source profiles play an important role in establishing the interconnection between a consumer's query request and the relevant information sources.

The second step is called *Query decomposition.* It is done by decomposing a user query expressed in terms of the DIOM interface query language (IDL) into a group of subqueries, each targeted at a single data source.

The third step is *Parallel query planning* where optimization of distributed query takes place. The goal of generating a parallel access plan for a group of subqueries is to find a relatively optimal schedule that makes use of the parallel processing potentials and the useful execution dependencies between subqueries, resulting from built-in heuristics, to minimize the overall response time and reduce the total query processing cost. This is the main focus and contribution of the research and implementation project reported in this thesis.

Since the problem of optimization is *NP-complete* [9, 20], to achieve feasible query optimization as well as to enhance the effect of the knowledge the user possesses about the idiosyncrasies of the system, heuristic search for solution is performed initially, followed by a full cost-based search in thus reduced solution space. The purpose of applying the heuristics is to restrict the solution search space for optimization at the early stage of optimization. However, it is not the purpose of the heuristics-based optimization to finalize the search. The intention is to restrict it well enough so that the cost-based search is feasible in the new solution space, e.g., it leaves three or four alternatives.

Given a query request, there are often more than one way to reduce the search space of the query, even with applying heuristics such as proposed in [14]. For instance, one of the possible tactics in reducing the solution space is to discard the query execution plans that are obviously non-optimal, thus leaving only the promising plan candidates. This includes discarding certain non-optimal orders of execution of query operators. [14] gives an example of such tactics that when two query operators, with strict and non-strict selections, need to be ordered, the priority is given to the strict one, which has greater selectivity than the non-strict, e.g., = operator is more selective than > operator, therefore query operators that have the former should be executed first to reduce the size of the solution space at early stages of query execution.

The next step is called *Subquery translation and execution.* The translation process basically

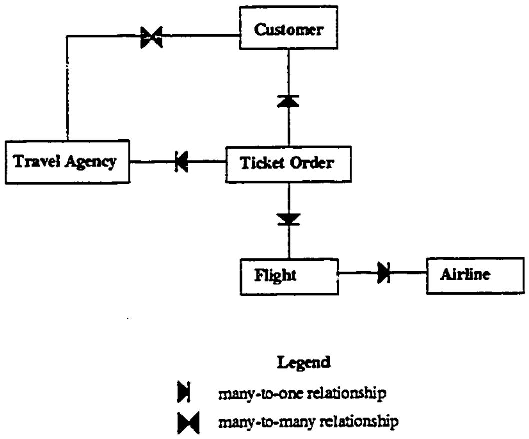


Figure 2-5: Entity–Relationship Diagram of Airline Reservation domain model.

converts each subquery expressed in the interface query language into the corresponding information producer's query language expression, and adds the necessary join conditions required by the information source system.

After submitting the subqueries, the DIOM query server is responsible for (1) packaging each individual subquery result into a DIOM object (done at wrapper level) and (2) assembling results of the subqueries in terms of the consumers' original query statement (done at mediator level). The semantic attachment operations and the consumers' query profiles are the main techniques that we use for resolving semantics heterogeneity implied in the query results. This step is referred to as *Query result packaging and assembly.*

2.2.2 An Application Scenario: The Airline Reservation Example

In this thesis we use an application scenario taken from an Airline Reservation application domain. Figure 2-5 presents an entity-relationship diagram of this domain model. Each box in this diagram represents a class of objects. For instance, objects of class *Customer* are related to objects of class *Travel Agency* with a many-to-many relationship. In other words, a travel agency has a database of many customers, and each customer may use many travel agencies.

Consider a sample query in the *Airline Reservation* domain:

> *find all available reservation information about all customers who flew to a destination in Europe with* `Canadian Airlines International` *last year.*

A DIOM customer may issue this query using the interface query language (IQL). The main advantages of using IQL are that the customer need not be aware of the many different naming conventions and terminology used in the underlying information sources nor does the customer need to specify the join conditions. For a detailed definition of IQL, see [12]. The above query can be expressed in IQL as follows:

```
SELECT *
FROM Customer, Flight, Ticket Order
WHERE Flight->destination CONTAINS 'Europe' AND
      Flight->date BETWEEN '01-JAN-96' AND '31-DEC-96' AND
      Flight->airline = 'Canadian Airlines International'
```

As was described in section 2.2.1, DIOM query processing consists of five main steps. We illustrate these steps using the sample query in the *Airline Reservation* application.

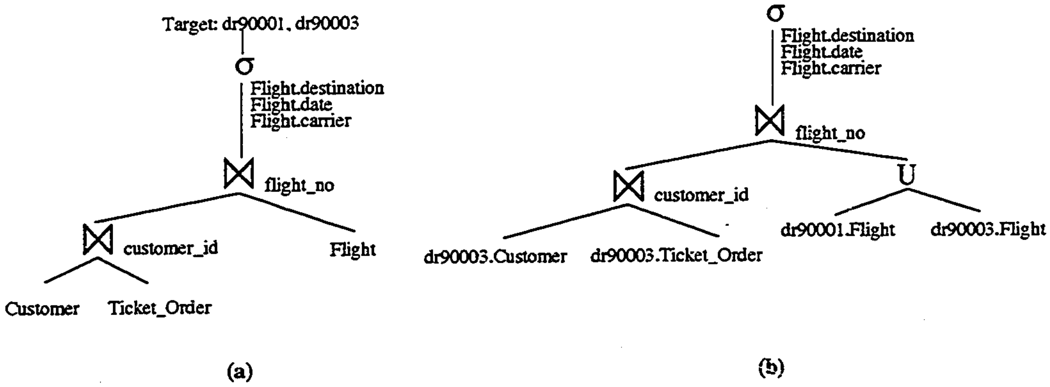


Figure 2-6: Query Decomposition Step.

2.2.2.1 Query Routing

To answer each query in *Airline Reservation* domain, the potential number of sources may be huge. There may be many airlines and travel agencies registered with DIOM server. Since query routing algorithms are not the theme of this thesis, for simplicity in this example we assume that the DIOM mediator identifies the following information repositories to answer the given query:

- dr90003 information repository, registered with DIOM as *Europa Travel* travel agency on-line database, contains objects of type *Customer*, *Flight*, *Ticket Order*;
- dr90001 information repository, registered with DIOM as *Canadian Airlines International* on-line reservation system, contains objects of type *Flight*;

The original query posed by the DIOM user then needs to be decomposed into subqueries which are against each of these two repositories. The following IQL is the result of query routing:

```
TARGET dr90001, dr90003
SELECT *
FROM Customer, Flight, Ticket Order
WHERE Flight->destination CONTAINS 'Europe' AND
      Flight->date BETWEEN '01-JAN-96' AND '31-DEC-96' AND
      Flight->airline = 'Canadian Airlines International'
```

2.2.2.2 Query Decomposition

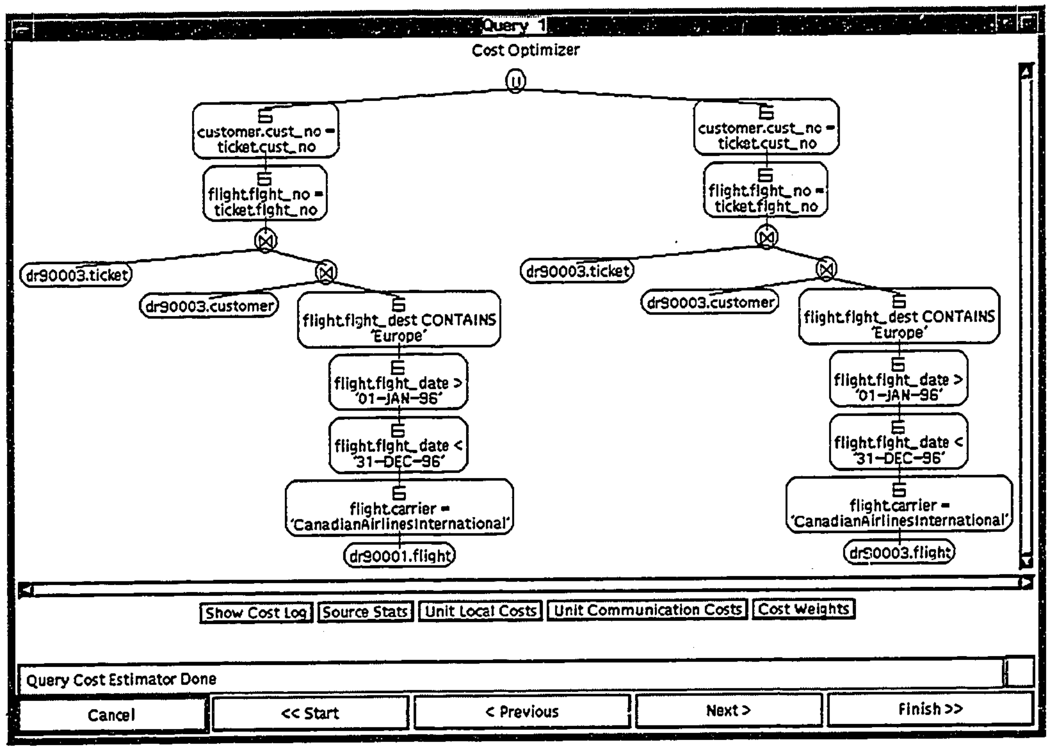
The query decomposer takes the IQL query represented in the tree structure as shown in Figure 2-6(a), and generates the query decomposition tree shown in Figure 2-6(b), which takes into account the results of query routing. It basically decomposes the leaf nodes of the query tree in Figure 2-6(a) by associating the sources specified in the TARGET clause.

The query tree contains the necessary information for obtaining the result of the query but it does not provide the concrete query execution plan.

2.2.2.3 Parallel Plan Generation

The next step of query processing is the generation of query execution plan, a concrete set of instructions to be sent to each of the information sources participating in the query. As pointed out in [12, 14], to generate an efficient query execution plan, two problems need to be solved:

- the query processor must find the most optimal (efficient) order in which the query operators are executed, and
- each of the query operators need to be assigned to a site where it can be executed.


Query 1
Cost Optimizer
customer.cust_no = ticket.cust_no
flight.flight_no = ticket.flight_no
dr90003.ticket
dr90003.customer
flight.flight_dest CONTAINS 'Europe'
flight.flight_date > '01-JAN-96'
flight.flight_date < '31-DEC-96'
flight.carrier = 'CanadianAirlinesInternational'
dr90001.flight
customer.cust_no = ticket.cust_no
flight.flight_no = ticket.flight_no
dr90003.ticket
dr90003.customer
flight.flight_dest CONTAINS 'Europe'
flight.flight_date > '01-JAN-96'
flight.flight_date < '31-DEC-96'
flight.carrier = 'CanadianAirlinesInternational'
dr90003.flight
Show Cost Log
Source Stats
Unit Local Costs
Unit Communication Costs
Cost Weights
Cancel
<< Start
< Previous
Next >
Finish >>

Query Cost Estimator Done

Figure 2-7: The final result of the *Airline Reservation* domain query.

We formally address these problems in detail in Chapter 3. In Chapter 4, we provide an experimental design of the *DIOM Query Scheduling Utility* that implements the query scheduling algorithm proposed in Chapter 3. Figure 2-7 shows the final result of the cost-based query scheduling obtained by the *DIOM Query Scheduling Utility*. Each node in the query tree shown there has been assigned to the site where it can be executed.

2.2.2.4 Subquery Translation and Execution

Once the query execution plan is generated, subqueries are issued to each information repository that participates in the plan. These subqueries are still expressed in DIOM IQL and therefore need to be translated by the wrappers and executed by the individual repositories. The issue of subquery translation and execution is beyond the scope of this thesis and is covered in detail in [10, 12, 14].

2.2.2.5 Query Result Assembly

The results of each individual subquery need to be assembled and presented to the user. The specifics of query result assembly are also beyond the scope of this thesis.

In this chapter we have covered the main steps of query processing in DIOM and presented the application domain that we use throughout the rest of the thesis. The purpose is to present the general picture, in which our work takes place. The remaining chapters present a concrete solution for the query scheduling component in the context of this general picture.

Chapter 3

Distributed Query Scheduling

3.1 Global Optimization Criteria

Traditionally, the distributed database query optimization was primarily aimed at reducing the communication cost. Little attention has been paid to other costs incurred at query execution in a distributed environment such as the local processing cost as well as the cost associated with the response time.

However, due to drastic advancements in network communication speeds and bandwidth, these other costs have become as significant factors as the communication costs.

Moreover, to cater to the needs of various database users, it is desirable to provide a flexible framework for query optimization, which allows to plug in efficient components of query cost estimation on demand, thereby providing the user-driven and customizable query optimization.

In general, the cost of query execution consists of three independent factors: communication cost, local processing cost, and total response time cost. They may be combined additively into a generic goal formula shown in Equation 3.1.

$$Cost = a_{cc} \cdot C + a_{lqp} \cdot L + a_{rt} \cdot R = A^T \cdot \begin{pmatrix} C \\ L \\ R \end{pmatrix}, \text{ where} \qquad (3.1)$$

- C is the total amount of communications over the network spanning the distributed database expressed in time units;
- L is the total amount of local query processing, also expressed in time units;
- R is the total response time of the query.

The coefficients associated with each of these components are the indicators of the desired optimization profile. They can be controlled by the user of the database by setting the profile via the components of vector A^T. For example, if the user's primary concern in finding an optimal query execution plan is the response time, then A^T is set to $(\,0 \quad 0 \quad 1\,)$. Vector $A^T = (\,0.3 \quad 0 \quad 0.7\,)$ would be specified by a user who is also concerned with keeping the communication cost low, allocating 30% of the total cost to it and 70% to the response time.

The general cost estimation formula 3.1 serves as the goal function of the optimization process. As was pointed out in chapter 2, the problem of query optimization is *NP-complete*, therefore direct application of this goal function is not feasible since the number of possible solutions is far too great even for queries of average complexity.

In this chapter we propose a scheduling technique that reduces the solution space prior to cost estimation thus reducing the expense of query processing itself, we call it *two-phase reduction.*

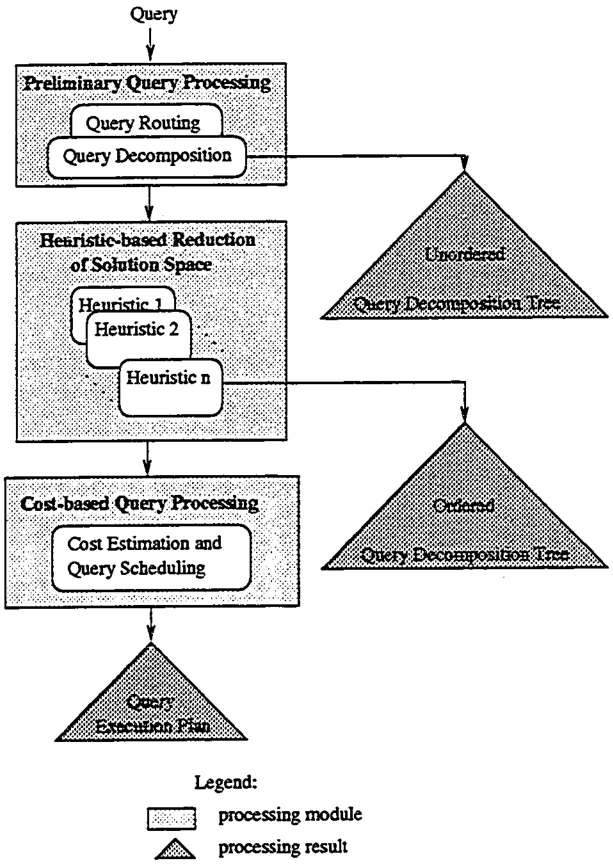

Query
Preliminary Query Processing
Query Routing
Query Decomposition
Heuristic-based Reduction of Solution Space
Heuristic 1
Heuristic 2
Heuristic n
Cost-based Query Processing
Cost Estimation and Query Scheduling
Unordered Query Decomposition Tree
Ordered Query Decomposition Tree
Query Execution Plan
Legend:
processing module
processing result

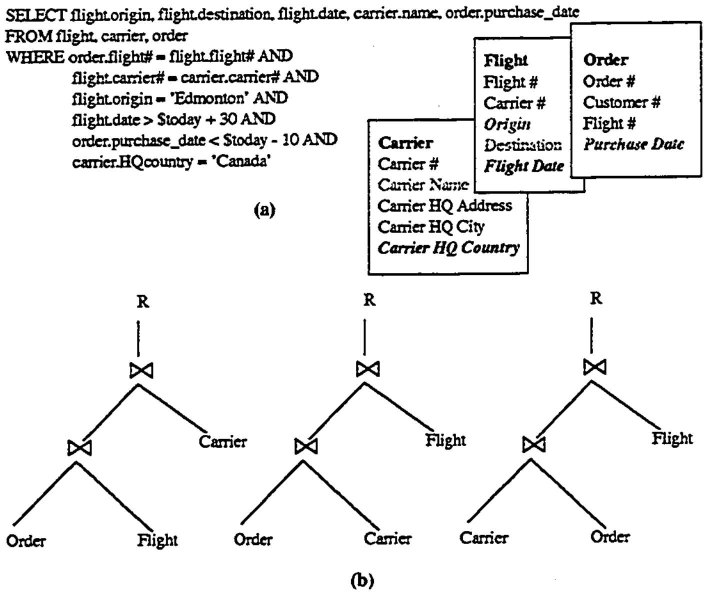
Figure 3-1: Two-phase query processing technique, a diagram.

3.2 Two-Phase Reduction Approach to Distributed Query Processing

A sketch of the proposed query scheduling architecture is shown in Figure 3-1. The first processing phase is based on the heuristic query processing and optimization techniques. This phase is covered in detail in section 3.3. The second processing phase is the cost estimation and access plan scheduling. Section 3.4 covers this phase in detail.

3.3 Heuristic-Based Optimization

In the last section we have mentioned the two-phase reduction approach to query processing. The cost-based phase is responsible for computing costs of various possible scenarios for query execution. In particular, a main manipulating factor that affects cost of a query execution plan is the site assignment of the binary operators of the query. At the same time the query execution plan may also be affected by changing the order of the relational operators, as long as the expression resulting from query rewriting is equivalent to the original query. In this section we will consider, in detail, the possibilities of heuristic-based optimization that involves the use of query rewriting rules to generate the "optimal" query expression that is equivalent to the original query.



```
SELECT flight.origin, flight.destination, flight.date, carrier.name, order.purchase_date
FROM flight, carrier, order
WHERE order.flight# = flight.flight# AND
        flight.carrier# = carrier.carrier# AND
        flight.origin = 'Edmonton' AND
        flight.date > $today + 30 AND
        order.purchase_date < $today - 10 AND
        carrier.HQcountry = 'Canada'
```

Figure 3-2: Example of rebalancing the query tree with three-way join.

3.3.1 Motivation

Consider a case in which three relations are joined. The three-way join is often presented in terms of two binary joins. The question then is, which two relations must be joined first.

A common tactic that is used in most state of the art research in query optimization, not only in distributed but also in centralized databases is to perform join on the relations which do not have any common attributes last. It is simply because their join is identical to the operation of *Cartesian product.*

Another common tactic is to reduce the size of the query result at the earliest stage of query processing. This leads to a heuristics that the join producing the smallest result should be performed earliest in the query execution plan.

The impact of the above two heuristic rules on the cost of query processing can be very significant. Let us consider an example shown in Figure 3-2(a). The given query asks to

> *select flight, carrier, and order information for all flights originating in Edmonton that were booked more than ten days ago, will fly in more than a month, and will be performed by a Canadian carrier company.*

Figure 3-2(b) shows the possible permutations of the two joins of the query tree. In fact, there are three choices in this query optimization task:

1. to join *Order* with *Flight* first, followed by joining the result of *Order*$\bowtie$*Flight* with *Carrier*;
2. to join *Order* and *Carrier* first followed by joining the result with *Flight*;
3. to join *Flight* and *Carrier* first followed by joining the result with *Order*;

Assume the following statistics are available or can be gathered about these three relations:

	Order	*Flight*	*Carrier*
$card_{domain}$	10,000	1,000	100
$width_{domain}[bytes]$	4	4	2
$card_{relation}$	6,000	700	20
$card_{Flight\#}$	500	700	N/A
$card_{Carrier\#}$	N/A	15	20

From the schema description in Figure 3-2(a) we see that *Order* and *Carrier* do not have any common attributes, therefore join between them is equivalent to Cartesian product. The size of a Cartesian product is equal to the product of the sizes of its two inputs, therefore the second choice does not reduce the result at all, which means that this choice is not beneficial in terms of cost. The size of the intermediate result would be $(4 \times 6,000) \times (2 \times 20) = 960,000$ [bytes].

On the other hand, the first choice yields the size of the intermediate result to be $6,000 \times 700/700 \times (4+4) = 48,000$, and the third, $700 \times 20/20 \times (4+2) = 4,200$. Thus based on the second heuristic rule, the third scenario is chosen because its reduction power is at least ten times greater than that of the first choice, which, in turn, is greater than that of the third choice. Note that this analysis is based on heuristic rules, not on the actual computation of the cost. It is on the assumption that the intermediate result will probably have to be moved that we make the reduction power the most crucial in our decision, which in most cases is true.

In the following two sections we will discuss some of the heuristic rewriting rules that we anticipate to be valuable in reducing the search space in the distributed environment of DIOM. Before we start our semantic rewriting rules we below present a number of common heuristics that are used in DIOM to limit the search space for the query processor:

Heuristic 3.1 (Common Heuristics)
(a) moving relational selection and projection operators down the query tree to reduce the expected size of the intermediate result of the query [5, 9, 20];
(b) considering only such join orderings that do not result in Cartesian product between relations [18];
(c) performing the joins whose estimated result is smaller before the joins that are expected to have larger intermediate result [1].

3.3.2 Heuristics Based on Semantic Rewriting

According to [14], the query expression passed to the query optimizer after query decomposition and routing phase includes all relevant database sources that were selected to answer the query.

The first semantic-based rewriting rule is to eliminate the sources that do not contain join attributes required in the query.

For the keyword-based scenarios of information retrieval, which presently dominate the consumers' world-wide information systems, missing/absence of join attributes is not detrimental since queries do not involve inter-site information processing. The missing attributes are retrieved as empty fields, and the consumer understands that this type of information is not available at this source.

However, when a query involves inter-site joins of the objects or relations, the join attributes may play a crucial role in answering the query.

Let us consider an example in which we will show how such knowledge can effectively reduce the number of sources involved in the query and therefore the complexity of cost-based query optimization. Consider query:

> *find origin, destination, and date of all flights that have been booked on date* pd *and that fly before date* fd,

whose algebraic representation is shown in the top part of Figure 3-3. The bottom part of the figure provides the schema information of the *Flight* and *Order* objects.

Although the schemas are different, the information covered by them is relevant to the query, therefore a DIOM query router will select the following three source databases to answer the query:

Travel Agent database, which is primarily concerned with customers and their relationship to flights;

Airline database, whose concern is on plane and flight-related information;

Airport database, focusing on departure schedule and plane types (because it has to service the planes).

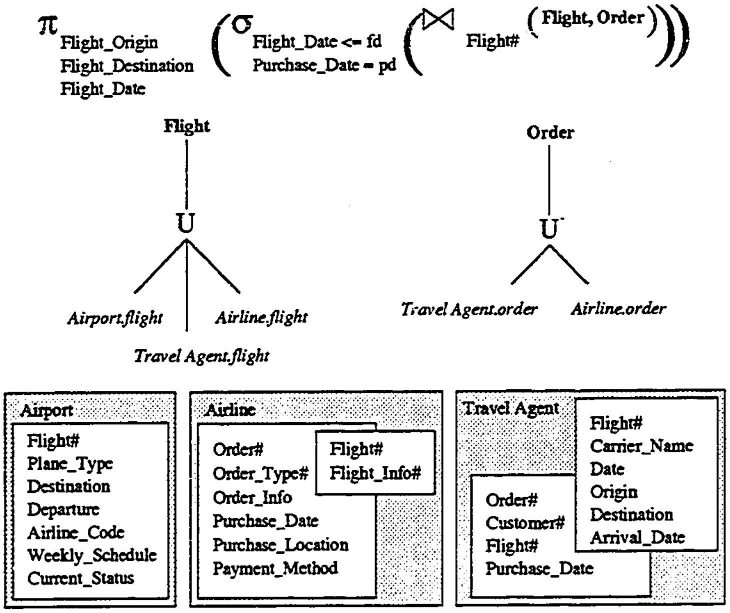


Figure 3-3: Example of semantic incompleteness.

As seen from Figure 3-3, the *Order* information in the airline database does not contain any means of joining it with *Flight* information, assume the airline is not involved with ticket-order information. Therefore the required join between *Flight* and *Order* is not possible for the source at *Airline* database, thus whether the **Airline** database takes part in the query assembly does not affect the final result of the query. Intuitively, we can stipulate that in a multilevel query tree the same observation holds for all sources and all levels of binary operators that require selection by a certain attribute that must be present in the source. If a source relation, an object class, or a file selected by the mediator does not provide all of the attributes required for all binary operators, we would exclude the participation of that source in the query answer. This is performed by the query router.

For instance in the example of Figure 3-3 attributes Flight_Date and Purchase_Date are required for selection operation. Each source's wrapper would try to provide this attribute in some form, be it direct mapping of *Travel Agent*'s Date attribute, or creating a method for *Airline* database that would draw the flight date. As a result the query optimizer looks up the corresponding wrappers and discards the ones that fail to provide the necessary information for this operator.

On the other hand, projection operation, which results from SELECT clause in *SQL*-like queries, semantically does not require presence of the attribute. Quite reasonably, a record may still be part of the result even if some of its fields are missing. Therefore we would allow the projected attributes to appear as blank fields in the query result. Looking back at Figure 3-3, Flight_Origin and Flight_Destination would appear missing on the information originating at *Airline* database unless its wrapper could fill in the blanks.

3.3.3 Heuristics Based on Semantic Rewriting and Estimated Cost

As most research results in query optimization indicate, the task of finding the absolute best query plan is a combinatorial problem in general. Our task in this section is to come up with several heuristic rules that would cut down the amount of search to be done during the query processing and to produce nearly, or almost, optimal plans, for the cases which in our opinion will be the most common in the DIOM environment. We will consider two examples.

It is widely accepted that the query optimization strategy in centralized database systems is to perform the most expensive and least effective operators last in the query. In other words, the goal is to reduce the size of the result as early as possible, for which we will put the most reducing operators first and the least last. What are the operators that are the least effective in terms of reducing the result? The most notoriously known one is Cartesian product between two relations. Indeed, the output size of this operator is the product of sizes of the operands. Another one is the binary union operator. It results in a relation or object extent that is at least the size of the largest of the operands. And, finally, join operator falls into this category.

All three operators are expensive, and require certain preprocessing of the operands to achieve some performance improvements. For example, by properly ordering the operands or the execution sequence of these operators, we may still obtain performance improvement.

We observe that both union and Cartesian product should always be performed at the site where the result is expected – see section 3.5.1 on page 32 – because the communication cost of transferring the result is greater than the communication cost of transferring any one of the operands. However, the union is still a better reducer than Cartesian product because $l(\cup(Q_i, Q_j)) \leq l(Q_i) + l(Q_j)$ while $l(\times(Q_i, Q_j)) = l(Q_i) \times l(Q_j)$, (see page 36 for definition of $l(Q)$). The latter amount is greater except for empty Q_i or Q_j. Therefore, it would be more beneficial to perform Cartesian product after the union, except for the cases when one or both of the operands is empty. Thus,

Heuristic 3.2 (Union & Cartesian Product)
(a) Given a subquery of type $\cup(\ldots(\cup(Q_1, Q_2), \ldots, Q_n))$*, the total cost of its execution is invariant to the order in which the unions are performed.*
(b) Given a subquery of type $\times(\ldots(\times(Q_1, Q_2), \ldots, Q_n))$*, the total cost of its execution is minimum if* $size(Q_1) \leq size(Q_2) \leq \cdots \leq size(Q_n)$.
(c) Given a subquery $\times(\cup(Q_i, Q_j), Q_k)$*, the cost of it is less than the cost of its permutation* $\cup(\times(Q_i, Q_k), \times(Q_j, Q_k))$.

Consider an example query:

> *perform Cartesian product of* flight *and* order*, i.e., find all the possible flight-order combinations*

Assume that to answer this query three sources have been selected, two of which contain *flight* objects, and the third contains *order* objects. Then the query decomposition expression is as follows:

$$\times(\cup(flight_1, flight_2), order).$$

Based on heuristic 3.2, the following expression is equivalent but less beneficial:

$$\cup(\times(flight_1, order), \times(flight_2, order)).$$

The situation with join is more complex. The problem is that in most cases it is impossible to predict exactly the extent of the join result, only approximate estimation may be obtained. Therefore in some cases it may be more beneficial to interchange union and join, while in others it is not.

Due to the nature of DIOM system, the query trees produced by decomposition and routing stages are most likely to have unions at the bottom of the tree, closer to the leaf nodes. In some cases these unions will encompass many information sources. The following rewriting rule can be applied to produce an equivalent query expression by moving join before union:

$$\bowtie(\cup(Q_{11}, Q_{12}), Q_2) \Rightarrow \cup(\bowtie(Q_{11}, Q_2), \bowtie(Q_{12}, Q_2)).$$

Apparently, the number of join operations increases twice with each union-join order exchange. To justify this increase, the expected performance gain must be greater than the additional cost of the extra join operator. There are two factors that may be used to estimate the required performance gain.

First, the join itself has to be a good reducer.

Heuristic 3.3 (Good Reducer)
Given a subquery $\bowtie (\cup(Q_{11}, \ldots Q_{1n}), \cup(Q_{21}, \ldots Q_{2m}))$, *it is beneficial to rewrite it into* $\cup(\bowtie (Q_{11}, Q_{21}), \ldots, \bowtie (Q_{1n}, Q_{2m}))$ *if the expected size of join result is smaller than any of its inputs.*

Consider an example query:

> *select all ticket order and flight information about all ticket orders that were made on the same day as the flight for which they were booked was to depart.*

Assume the same three sources have been selected, two with *flight* objects, and one with *order* objects. The query decomposition expression for this query is

$$\sigma_{flight.date=order.date}(\bowtie_{flight\#} (\cup(flight_1, flight_2), order)).$$

It is beneficial to rewrite this expression into

$$\sigma_{flight.date=order.date}(\cup(\bowtie_{flight\#} (flight_1, order), \bowtie_{flight\#} (flight_2, order)))$$

because the join operator may substantially reduce the size of the query result. More concretely, if the extent of *order* is very small, the join result is small and the new expression is accepted. Otherwise, if the estimated size of the join result is large, the old expression is kept.

Second, the interchange, if done, must result in fewer site accesses, thus,

Heuristic 3.4 (Same Sites)
Given a subquery $\bowtie (\cup(Q_{11}, \ldots Q_{1n}), \cup(Q_{21}, \ldots Q_{2m}))$, *it is beneficial to interchange join with the unions if*
$(n \geq m) \wedge (i \geq n/2 + 1) \wedge (loc(Q_{11}) = loc(Q_{21})) \wedge \ldots \wedge (loc(Q_{1i}) = loc(Q_{2i}))$, *or if*
$(m > n) \wedge (j \geq m/2 + 1) \wedge (loc(Q_{21}) = loc(Q_{11})) \wedge \ldots \wedge (loc(Q_{2j}) = loc(Q_{1j}))$,
where loc is a function that returns the site location of its argument.

In other words, if the interchange of the join and the union does not lead to an increase in the number of the inter-site joins, then such interchange is performed, otherwise, the order of operations is kept unchanged.

Note, however, that Heuristic 3.4 may only be applied to the unions which are immediately connected to the leaf nodes of the tree because only for these unions the location is known. If a union is not of this nature, then this heuristic is ignored.

Heuristics 3.3 and 3.4 are targeted at the same rewriting rule. It may so happen that either one of them will trigger the transformation of the query expression. Thus if one of them triggers the tranformation that moves a particular join operator down, the second heurisitc does not need to be applied.

Consider the same query example but assume now that one of the sources of the *flight* objects is the same as the source of the *order* objects (source *1*). Then applying Heuristic 3.4 to the query decomposition expression $\bowtie (\cup(flight_1, flight_2), order_1))$, we have

$$\sigma_{flight.date=order.date}(\cup(\bowtie_{flight\#} (flight_1, order_1), \bowtie_{flight\#} (flight_2, order_1))), \text{ where}$$

$\bowtie (flight_1, order_1)$ is a local join operation on site 1. Thus, by changing the order of join and union, expression $\cup(\bowtie (), \bowtie ())$ is more beneficial because the number of inter-site joins has not increased as a result of rewriting.

In this section we have considered some of the heuristic rules that may be applied to reduce the search space for the optimal query execution plan. As a result of applying the heuristics the decomposition trees that are produced from the original query expression may be pruned thereby reducing the search space in the optimization problem. Usually, however, this reducing does not lead to a single solution to the problem but provides a number of alternatives. Second phase of the query optimization is to find an execution plan that is optimal for answering the query, using cost-based optimization approaches. This is the theme of the next section.

3.4 Cost-Based Optimization

3.4.1 Statistics Required for Cost Estimation

To effectively estimate the cost of query execution it is important to determine the size of the operands of the query predicates, as well as the size of all the partial results. This problem is of statistical nature, and therefore requires a statistical model of the database, along with certain assumptions about the database that make such statistical model justifiable.

In this section we will consider how such statistical model may be put together for three most common types of database systems, relational, object-oriented, and a file retrieval system. A running example is used in this section to illustrate the notions and issues involved.

3.4.1.1 Statistical Model of Relational System

[1] proposes a statistical model for relational databases, according to which the attribute values in relations are

1. distributed uniformly,
2. statistically independent of the values of other attributes, and
3. statistically independent of other values of the same attribute.

The following statistics may be used to estimate the size of the relational operands:

Cardinality of domain – fully defined by the type of the attribute. In relational database each attribute is associated with a domain, for example, if an attribute is of type string with length n then n unambiguously defines the cardinality of this domain, which is the number of distinct values the attribute may take, see section 3.4.4;

Width of domain – defines the length of one element of the domain in bytes, for example, a decimal integer number is 16 bytes long;

Cardinality of relation – the actual number of tuples in the relation. This number may be updated periodically by a system which supports such statistical information;

Cardinality of relation's attribute – the number of relation tuples having distinct values in the given attribute.

The latter statistic requires the existence of an index on the particular attribute. If the attribute is a primary key then the index on it is implemented as a table, other indexes are implemented as binary trees, therefore many systems maintain the required information for the indexed attributes. Most commercial relational database management systems measure and maintain the statistical information listed above. Figure 3-4 shows an example of a simple relation of *Order-Flight-Customer* and the related statistics.

As stated earlier, the main purpose of having the statistical information about the database is to estimate the cost of processing a query, which is affected by the size of all the relations that take part in evaluating the query. For instance, the following query

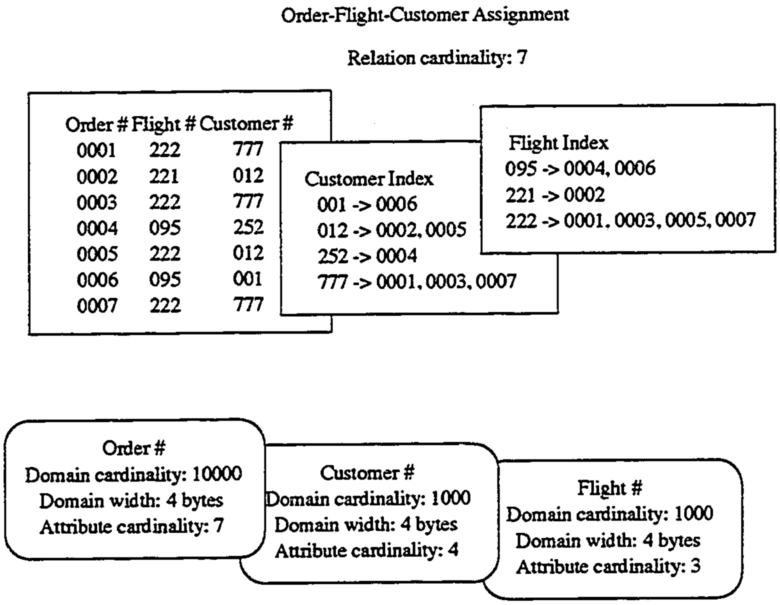
```
SELECT *
FROM Order-Flight-Customer
WHERE Flight# = 222 OR Flight# = 221
```

can be expressed in the relational algebra expression as follows:

$$E = \sigma_{Flight\#=222\ or\ Flight\#=221}(Order\_Flight\_Customer)$$

To compute the cost of this query expression E, we first compute the cost of the relational select operator σ, then the relational project operator π.

Since relation `Order-Flight-Customer` has cardinality 7, and `Flight#` has cardinality of 3, and the estimated selectivity of selection over `Order-Flight-Customer` with selection condition on `Flight#` is 2/3, the estimated cardinality of E is $7 * \frac{2}{3} = 4\frac{2}{3} \approx 5$.



Order-Flight-Customer Assignment

Relation cardinality: 7

Order #	Flight #	Customer #
0001	222	777
0002	221	012
0003	222	777
0004	095	252
0005	222	012
0006	095	001
0007	222	777

Customer Index
001 -> 0006
012 -> 0002, 0005
252 -> 0004
777 -> 0001, 0003, 0007

Flight Index
095 -> 0004, 0006
221 -> 0002
222 -> 0001, 0003, 0005, 0007

Order #
Domain cardinality: 10000
Domain width: 4 bytes
Attribute cardinality: 7

Customer #
Domain cardinality: 1000
Domain width: 4 bytes
Attribute cardinality: 4

Flight #
Domain cardinality: 1000
Domain width: 4 bytes
Attribute cardinality: 3

Figure 3-4: An example of database statistics required for cost estimation.

3.4.1.2 Statistical Model of Object-Oriented System

Figure 3-5 shows how the relational schema shown in Figure 3-4 may be expressed in an object schema. Although it is impossible to consider the above two schemas without the context of the entire database, it is clear that it represents a cross-reference relationship among entities of three types: customer, flight, and order. Therefore the relations per se are the links between the objects of these three types, although in a strictly object-oriented implementation these relationships are represented as objects of a special type. Depending on the constraint of the relationship, a link-object may be one of `one-to-one`, `one-to-many`, or `many-to-many` types.

The statistics then may be listed as follows,

Class cardinality is based entirely on the intensional definition of the class, e.g., a class of dates lying in the range of the century has cardinality 36,500;

Object size is simply the number of bytes an object of a certain class occupies;

Class extent is the number of instances of a certain class in the database;

Object inter-reference extent is the number of instances of link-type objects that reflect relationships of objects of a class with objects of either other classes or the same class.

Although there are similarities between the statistics of the relational schema and the statistics of the object schema, they are different in several aspects. The domain characteristics are independent of the schema and therefore are the same as in relational schema. However in object-oriented approach, the attributes which facilitate entity relationships by means of value-based equality, are replaced with concrete and unique objects, which relate to other objects in the schema by means of id-based equality, and the cross-reference relation is not explicitly present in the schema. In other words, in relational schema, the cardinality of a relation is simply the number of tuples in the relation, while in object schema, this number defines the complexity of the relationship between the objects of two classes. For example, the complexity of relationship between `Order` and `Customer` objects is defined by the number of all possible links that connect them, as shown in Figure 3-4.

Generally, for a *many-to-many* relationship between objects of class A and objects of class B, let $Card_{AB}$ denote the cardinality of this relationship, let $card(Link_{A,B})$ denote the number of links

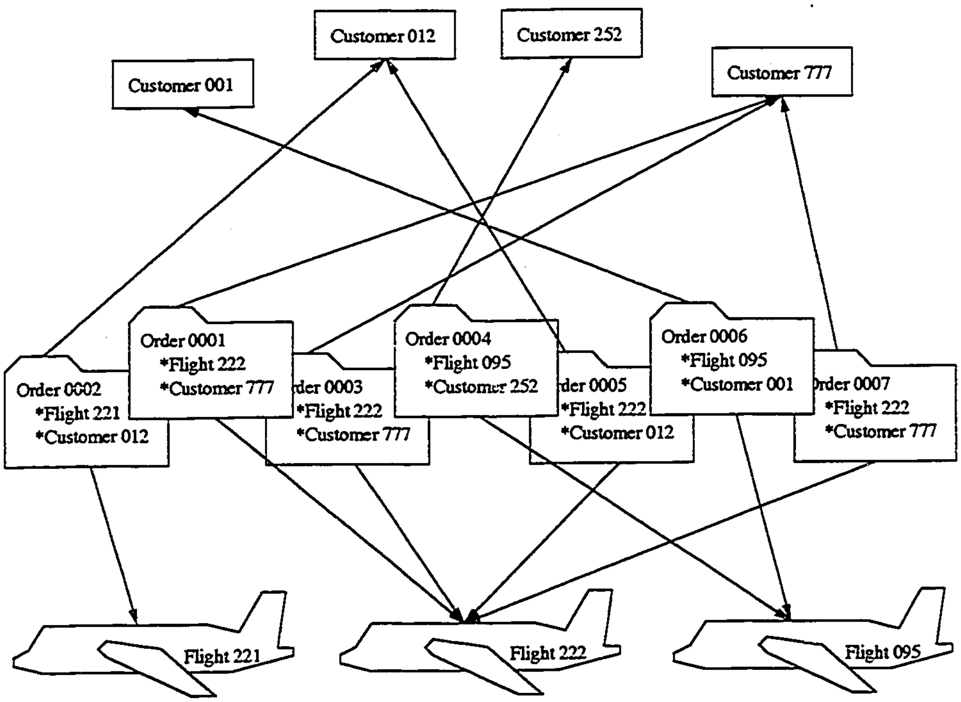


Figure 3-5: Object version of the relational schema.

an object i of class A has with objects of class B, and let n_A denote the cardinality of class A, then

$$Card_{AB} = \sum_{i=0}^{n_A-1} card(Link_{A,B}). \tag{3.2}$$

The formula is true for any of the three types of entity relationships. For instance, Flight-Order relationship has inter-reference extent 7 since there are a total of 7 connections between the Flight and Order objects. In the example of Figure 3-4 the Order is the primary key of the relation. In terms of object-oriented schema, the objects of class Order have many-to-one relationships with objects of both Flight and Customer.

Given an attribute A of class C, the cardinality of A is estimated by the cardinality of C.

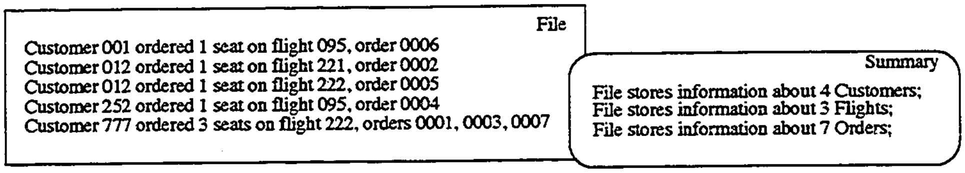
3.4.1.3 Statistical Model of File System

File-based information storage and retrieval systems do not implement the notion of logical entity relationships. All information stored in files is in record form where records are in some way delimited and possibly have distinct record fields that allow to distinguish between various attributes of the information stored in them. However, such storage and retrieval systems do not allow for the records in these files to relate to the information in other files, or at least not at the logical level. Every record exhaustively contains all the possible information that can be inferred from such system.

Moreover, the tools to define, manipulate, and query the data in the file systems are very closely coupled with the particular storage system, therefore an application accessing the data hides the intrinsics of query processing on the system level. Query optimization in such systems is very system-specific and the user generally does not have access to it. The statistics mentioned above are not available, they are either hidden or are not used at all.

Therefore it is reasonable to designate the corresponding wrapper responsible for:

- establishing the necessary statistics, or



File

Customer 001 ordered 1 seat on flight 095, order 0006
Customer 012 ordered 1 seat on flight 221, order 0002
Customer 012 ordered 1 seat on flight 222, order 0005
Customer 252 ordered 1 seat on flight 095, order 0004
Customer 777 ordered 3 seats on flight 222, orders 0001, 0003, 0007

Summary

File stores information about 4 Customers;
File stores information about 3 Flights;
File stores information about 7 Orders;

Figure 3-6: File system version of the relational schema.

- providing the default statistics based on its experience with relational and/or object-oriented systems of a similar range.

The left part of Figure 3-6 contains a variant of how the relational schema shown in Figure 3-4 may look in the file storage system. The right part of of Figure 3-6 shows what kind of statistical information is needed from the file storage system.

3.4.2 Cardinality Estimation of Basic Algebraic Operators

In this section we only consider the following basic algebraic operators: =, IN, OR, AND, NOT, $\cup$, and $\bowtie$. For other operators that are specific to particular datatypes see section 3.4.3.

As stated earlier, the necessary elements of cost-based optimization are,

- the computation of cardinality of the expected result, and
- the computation of the cost of materializing the result.

These factors must be computed for each intermediate step of query processing. Note that the intermediate cardinality of an intermediate result directly influences the cost of performing the subsequent operation to which this result is used as an input.

Since all the optimization steps have to be completed before the query execution may start, it is impossible to know the actual numbers corresponding to the intermediate results of the processing. It is however possible to estimate these numbers using the statistical information and the model described in [1, 9, 20].

The cardinality of the query result depends on the selection power of the qualifying clause of the query. The more the qualifier restricts the result, the greater this selection power is. Generally, the qualifying clause of the query may consist of multiple conjunct or disjunct predicates, each of which is characterized by its selection power, so called *selectivity factor*. One needs to estimate the selectivity factor of each of such components as well as of their combination to effectively estimate the cardinality of the query result. *System R* [18], a research prototype of database management system *DB2*, was the first one that used selectivity factors for cost-based query optimization.

In this section we have tried to generalize the formulae for estimating the selectivity factors, and subdivide them based on their category and the types of their operands. The formulae are given in tables 3-1 and 3-2 on pages 25 and 26. There,

SF denotes the selectivity factor of the predicate or a combination of predicates,

$Icard(A)$ denotes the cardinality of attribute A,

$domain(A)$ denotes the domain of attribute A,

$card(list)$ denotes the cardinality of $list$, and

$card(Q)$ denotes the estimated cardinality of the result of query Q.

The computation of most of the selectivity factors involves knowledge of the statistical information about the relation, the class of objects, or the file to be accessed, see sections 3.4.1.1 through 3.4.1.3 for discussion on how this information is obtained.

Predicate	**Selectivity Factor SF**	**Comment**
$A = v$, where A is the attribute name, v is a value, $v \in domain(A)$	$SF = \frac{1}{Icard(A)}$	The adopted statistical model assumes that values of A are uniformly distributed in the domain of A, therefore probability of hitting the right value is equal to the inverse of the number of values in A
$A \neq v$, where A is an attribute, v is a value within the range of A	$SF = \frac{Icard(A) - 1}{Icard(A)}$	If v lies within the domain of A and the values of A are uniformly distributed over its domain, then the event $A \neq v$ is complementary to $A = v$, therefore its probability is $1 - P(A = v)$
$A_i = A_j$, where A_i and A_j are attributes, $domain(A_i) = domain(A_j)$	$SF = \frac{1}{\max\{Icard(A_i), Icard(A_j)\}}$	We assume that all values in the attribute with smaller cardinality have matching values in the attribute with larger cardinality, which is a reasonable assumption because the statistical model assumes uniform distribution of attribute values from their domain
A IN $\{list\}$, where A is the attribute name and $\{list\} = \{v_1, v_2, \ldots\}$ is a list of values each of which belongs to the domain of A	$SF = \frac{card(list)}{Icard(A)}$	This is analogous to $A = v_1$ OR $A = v_2$ OR $\ldots$; since the disjuncts are mutually exclusive, the selectivity factors add up, see the formula for OR operator in the next table
A IN Q_A, where A is the attribute name and Q_A is a subquery over relations $R_0 \ldots R_{n-1}$ that returns tuples whose type is the same as the type of A	$SF = \frac{card(Q_A)}{\prod_{i=0}^{n-1} card(R_i)}$	The ratio of the subquery cardinality to the cardinality of all possible subquery answers is called the selectivity of the subquery. This selectivity defines the selectivity of the entire IN operator. Assume the subquery results in a set of values that is a superset of A, then all values in attribute A are matched, the selectivity factor is 1. If subquery Q_A restricts the domain of A, then the portion of the result of Q_A which will match the values in A will be restricted proportionately. Therefore SF for this operator is the selectivity factor of Q_A, which is the ratio of the cardinality of the answer to the cardinality of all possible answers. This reasoning is extended to include subqueries that involve multiple relations and aggregate functions over attribute A

Table 3-1: Formulae for computing selectivity factors of query predicates.

Boolean Operator	Selectivity Factor SF	Comment
P OR Q, where P and Q are predicates, which have their selectivity factors SF	$SF = SF(P) + SF(Q) - SF(P \wedge Q)$	If P and Q are mutually exclusive then $SF(P \wedge Q) = 0$ and the selectivity factor of the operator is the sum of the selectivity factors of P and Q
P AND Q, where P and Q are predicates, which have their selectivity factors SF	$SF = SF(P) * SF(Q/P)$	$SF(Q/P)$ is the selectivity factor of predicate Q given that predicate P is true. Generally, P and Q are not independent, therefore this selectivity factor is different from $SF(Q)$ as the result set is already restricted by P. If P and Q are independent, then $SF(Q/P) = SF(Q)$. The adopted statistical model assumes that if P and Q predicates involve different attributes then they are considered independent
NOT P, where P is a predicate, and $SF(P)$ is its selectivity factor	$SF = 1 - SF(P)$	Let query with P result in a third of the relation's tuples being selected , then $SF(P) = 1/3$, consequently query with NOT P will result in $SF(\neg P) = 2/3$ because query without any predicate would result in all tuples being selected, or $SF = 1$

Table 3-2: Formulae for computing selectivity factors of combinations of predicates.

Table 3-1 does not cover a number of query predicates that are specific to numerical domains. Operators like <, BETWEEN, *high*, and *low* are covered later when we describe the selectivity factors of numerical datatypes, see section 3.4.3.

It is also important to be able to estimate the cardinality of the results of binary relational operators, such as union and join. First, let us consider the simpler case with union. The very nature of the operator suggests that the cardinality of the union result is at least the cardinality of the largest set participating in the union,

$$card(\cup(Q_i, Q_j)) \quad = \quad \max\{card(Q_i), card(Q_j)\}, \tag{3.3}$$

where Q_i, Q_j are the operands of the union, and $card(S)$ is the cardinality of set S.

The other statistical parameters, namely domain cardinality, width of domain, and attribute cardinality, of the result of the union are defined in the same way – the largest relation or class extent dominates in defining the characteristics of the union result.

[20, 1] give the following rules for computing the cardinality of the join result,

$$card(\bowtie_A (Q_i, Q_j)) \quad = \quad \frac{card(Q_i) \cdot card(Q_j)}{\max\{card(Q_i.A), card(Q_j.A)\}}, \tag{3.4}$$

where Q_i, Q_j are the operands of the join, and A is the set of attributes on which the join is performed, $card(Q_i.A)$ is cardinality of the projection of A on relation Q_i.

So far, we have considered estimating cardinality of the result of query operators assuming that the result is final, i.e., that it does not serve as an input to other query operators. However, most queries will consist of more than one level of query operators, the intermediate results of the operators at the lower levels will need to be used as input at a higher level, where we will need to repeat the estimation process applying the selectivity factors corresponding to the operators at this level. As was mentioned earlier, computation of selectivity factors requires full knowledge of the statistics about the operands of the query operator. Therefore, one needs to estimate not only the cardinality of the intermediate results but also the other statistics – such as the domain properties of each of the attributes in the result as well as cardinality of the attribute.

To estimate the attribute cardinalities we use the approach taken in [22], which presents the problem in the following way:

> Given n records grouped into m blocks $(1 < m \leq n)$, each contains n/m records, if k records $(k \leq n - n/m)$ are randomly selected from the n records, the expected number of blocks hit by the selection (blocks with at least one record selected) is
>
> $$m \cdot \left[1 - \prod_{i=1}^{k} \frac{nd-i+1}{n-i+1}\right], \text{ where } d = 1 - 1/m, \text{ or}$$
> $$m \cdot \left[1 - \prod_{i=1}^{k} \left(1 - \frac{n}{m \cdot (n-i+1)}\right)\right].$$

Let us translate this statement into our query terms. Given a relation R whose cardinality is $card(R)$ that has attribute A with cardinality $Icard(R.A)$, where $1 < Icard(R.A) \leq card(R)$, each value of A is present in $card(R)/Icard(R.A)$ tuples of the relation, if k tuples, where $k \leq card(R) - card(R)/Icard(R.A)$ are randomly selected from R, the expected cardinality of A in the resulting relation Q is

$$Icard(Q.A) = Icard(R.A) \cdot \left[1 - \prod_{i=1}^{card(Q)} (1 - \frac{card(R)}{Icard(R.A) \cdot (card(R) - i + 1)})\right] \tag{3.5}$$

where $card(Q)$ is the cardinality of the result, $card(Q) = k$.

In this section we have covered the basics of estimating the cardinality of query predicates and relational union and join operators. This discussion is generic in the sense that we have not considered the specifics of arithmetic datatype operands in the predicates or in the relational operators. This is done in the next section.

3.4.3 Cardinality for Number and Date Datatypes (Arithmetic)

Let us limit our study with decimal integer numbers. For true real numbers the term of cardinality does not apply since such numbers represent continuous values, for each pair of which there may be found a number that is between the two. Whereas the machine representation of real numbers is discrete, all the discussion below may be applied to machine real numbers as well as it is done for integers.

First, let us define the domain cardinality for the arithmetic datatype. Domain cardinality of a number is the set of all values that lie within the range of this domain. In the machine representation the domain of an arithmetic number is defined by the size of the number. For instance, *short int* is usually represented by data whose length is one byte, or eight bits, the valid range of this number lies in $[-128, 127]$. In this case the width of domain is one byte and domain cardinality is 256.

The upper and lower bounds of the range define the *high* and *low* values of the range, which in our case are $low(short\ int) = -128$, and $high(short\ int) = 127$. In this section we will show that it is important to know these domain parameters to successfully estimate the selectivity factors of most query operators defined on operands of arithmetic type.

It is also beneficial to keep track of the actual *high* and *low* values of the attributes or objects of numerical datatype. Generally, non-numeric datatypes do not define the comparison operators other than = and ≠, see Table 3-1. Numerical datatypes allow inequality-based comparisons as well as all arithmetic operands defined on each particular datatype.

Most query languages, including *SQL*, query by example, data manipulation languages of file systems supports this special class of operators. The interface query language of DIOM also supports them. To effectively evaluate the cost of a query plan that uses these operators one needs to estimate their selectivity factors.

Let $l(domain)$ and $h(domain)$ denote the low and high values of the domain of a numeric attribute. Also, let $l(A)$ and $h(A)$ denote the actual minimum and maximum values of numeric attribute A. These type-specific statistics may be collected together with the main statistics about

Predicate	Selectivity Factor SF	Comment
$A > v$, where A is a numeric attribute and v is a numeric value, $l(A) < v \leq h(A)$	$SF = \frac{h(A) - v}{h(A) - l(A)}$	v must be known at optimization time, otherwise see below. The probability that a value of A is greater than v is determined by the distribution of A, which is uniform. Therefore the probability is equal to the rectangular area bound by the probability density function and v
$A > B$, where A and B are numeric attributes	$SF = 1/2$	See discussion and formulae 3.6 on page 30
A BETWEEN v_1 AND v_2, where A is a numeric attribute, v_1, v_2 are values in the domain of A, $v_2 > v_1$	$SF = \frac{\min[v_2, h(A)] - \max[v_1, l(A)]}{h(A) - l(A)}$	BETWEEN AND operator is a short form of $A > v_1$ AND $A < v_2$, therefore the formula for selectivity is a particular case of the combination of the inequality operators

Table 3-3: Formulae for computing selectivity factors for predicated defined over numerical attributes.

each of the numeric attribute, they make it possible to estimate cardinalities of results of quantitative comparison operators, which are listed in Table 3-3.

As seen from the Table, the case of $A > B$ requires special attention. We will address it in the following discussion. In essence, the problem is to estimate the probability that a random value A is greater than a random value B, given that they are uniformly distributed in their respective ranges $l(A) \leq A < h(A)$ and $l(B) \leq B < h(B)$. This problem can be stated in a more presentable way by introducing random variable $C = -B$, $l(C) \leq C < h(C)$, thus $l(C) = -h(B)$, and $h(C) = -l(B)$. Then substituting the new random variable for B, the goal event of $A > B$ may be expressed as $A + C > 0$, thus probability of $A > B$ equals to probability of $A + C > 0$.

Based on our statistical model, the probability of a certain predicate being true is the selectivity factor of this predicate since it represents the portion of all samples of the random variable that satisfy the predicate, i.e., the percentage of all the tuples that are *selected* by this predicate. For instance, probability of 0.5 suggests that half of all tuples in relation will be selected.

To find probability of event $A + C > 0$, where A and C are uniformly distributed random variables, let us introduce a new random variable $Z = A + C$ and investigate its probability function denoted as $p(z)$. Figure 3-7 on page 29 presents the problem graphically.

The rectangular area bound by $l(A), h(A), l(C), h(C)$ (dotted lines) represents all possible values of A and C random variables. Then the thick line marked as $a + c = z$ divides this rectangle into two subareas one of which corresponds to $A + C \leq z$ (shaded) and the other to $A + C > z$. Then the probability of $Z \leq z$ is equal to the area corresponding to $Z \leq z$ divided by the area of the entire rectangle. Conversely, the probability of $Z > z$ is the a: ea above the thick line divided by the total area.

The range of z spans five different intervals, sepa ated from each other by dashed lines. Let us consider the probabilities for each of these intervals denoting $P(Z \leq z)$ as the probability that random variable Z is less than or equal to value z.

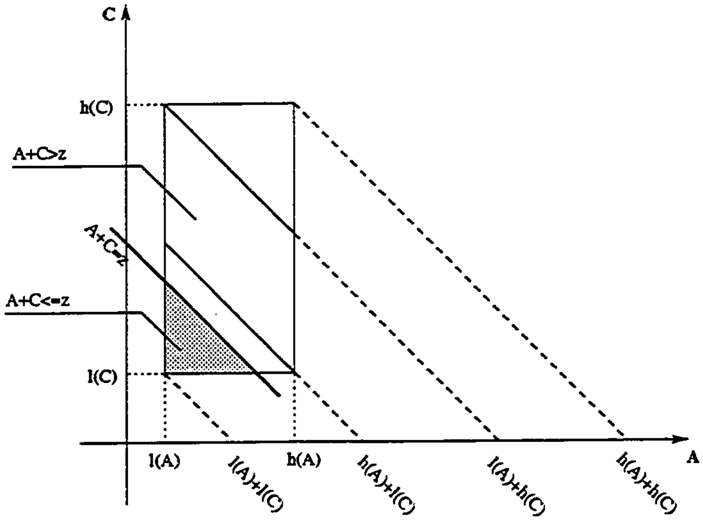


Figure 3-7: Graphical representation of probability problem $Z = A + C$.

$$P(Z \leq z) = \begin{cases} 0, & z < l(A) + l(C) \\ \frac{(z-(l(A)+l(C)))^2}{2(h(A)-l(A))(h(C)-l(C))}, & l(A) + l(C) \leq z < h(A) + l(C) \\ \frac{2z-h(A)-l(A)-2l(C)}{2(h(C)-l(C))}, & h(A) + l(C) \leq z < l(A) + h(C) \\ 1 - \frac{(h(A)+h(C)-z)^2}{2(h(A)-l(A))(h(C)-l(C))}, & l(A) + h(C) \leq z < h(A) + h(C) \\ 1, & h(A) + h(C) \leq z \end{cases}$$

Knowing the probabilities for these intervals we can determine the probability function $p(z)$ by finding the derivatives of the above formulae. Note that if $h(A) - l(A) > h(C) - l(C)$ then the picture will change in that $l(A) + h(C)$ will be to the right of $h(A) + l(C)$, therefore we use max and min of these values to preserve generality.

$$p(z) = \begin{cases} 0, & z < l(A) + l(C) \\ \frac{z-(l(A)+l(C))}{(h(A)-l(A))(h(C)-l(C))}, & l(A) + l(C) \leq z < \min[l(A) + h(C), l(C) + h(A)] \\ \frac{1}{\max[h(A)-l(A), h(C)-l(C)]}, & \min[l(A) + h(C), l(C) + h(A)] \leq z \\ & z < \max[l(A) + h(C), l(C) + h(A)] \\ \frac{(h(A)+h(C))-z}{(h(A)-l(A))(h(C)-l(C))}, & \max[l(A) + h(C), l(C) + h(A)] \leq z < h(A) + h(C) \\ 0, & h(A) + h(C) \leq z \end{cases}$$

The plot of this function is shown in Figure 3-8 on page 30. The area under the trapezoid is equal to one and represents the probability of random variable Z. Then probability that Z is greater than zero equals to the area of the positive portion under the trapezoid. Depending on the values of $l(A), h(A), l(C)$, and $h(C)$, the zero may be in either of the five intervals shown in the figure.

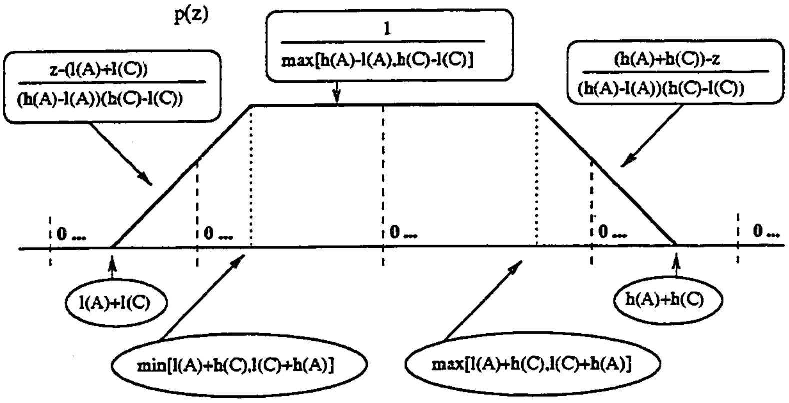

p(z)
z-(l(A)+l(C)) / (h(A)-l(A))(h(C)-l(C))
1 / max[h(A)-l(A),h(C)-l(C)]
(h(A)+h(C))-z / (h(A)-l(A))(h(C)-l(C))
0 ...
0 ...
0 ...
0 ...
0 ...
l(A)+l(C)
h(A)+h(C)
min[l(A)+h(C),l(C)+h(A)]
max[l(A)+h(C),l(C)+h(A)]

Figure 3-8: Probability function $p(z)$ of random variable $Z = A + C$.

After calculating the areas for each of the five cases, we substitute $-h(B)$ for $l(C)$ and $-l(B)$ for $h(C)$ and arrive at the formulae given in Equation 3.6.

$$SF = \begin{cases} 1, & 0 < l(A) - h(B) \\ 1 - \frac{(l(A)-h(B))^2}{2(h(A)-l(A))(h(B)-l(B))}, & \begin{array}{l} l(A) - h(B) \leq 0 \\ 0 < \min[l(A) - l(B), h(A) - h(B)] \end{array} \\ \frac{\min\left[\frac{h(A)-l(A)}{2}, \frac{h(B)-l(B)}{2}\right] + \max[l(A)-l(B), h(A)-h(B)]}{\max[h(A)-l(A), h(B)-l(B)]}, & \begin{array}{l} \min[l(A) - l(B), h(A) - h(B)] \leq 0 \\ 0 < \max[l(A) - l(B), h(A) - h(B)] \end{array} \\ \frac{(h(A)-l(B))^2}{2(h(A)-l(A))(h(B)-l(B))}, & \begin{array}{l} \max[l(A) - l(B), h(A) - h(B)] \leq 0 \\ 0 < h(A) - l(B) \end{array} \\ 0, & h(A) - l(B) \leq 0 \end{cases} \tag{3.6}$$

Let us, for instance, consider a case when $l(A) = l(B)$ and $h(A) = h(B)$ and apply the formulae in Equation 3.6. Quite amazingly, all these calculations reduce to a mere 1/2 selectivity factor for this trivial case, which is quite what we expected. This fact suggests that we should assume the selectivity factor of 1/2 for the cases when two numerical attributes whose domains are the same are compared and of which no actual high and low statistics are available.

3.4.4 Cardinality for Character and String Datatypes

For simplicity, let us assume the Oracle Corporation's definition of datatype representing a string of variable length[1].

Let us consider a number of possible operations applicable to objects of this datatype and their selectivity, i.e., the cardinality of the result of the operation contrasted with that of the initial cardinality of the operand.

For any string whose maximum length is n characters, each of which can assume one of l values, the total number of possible strings, including the empty string is given in formula 3.7.

$$S = 1 + l + l^2 + \cdots + l^n = \sum_{j=0}^{n} l^j = \frac{l^{n+1} - 1}{l - 1}. \tag{3.7}$$

[1] *VARCHAR*.

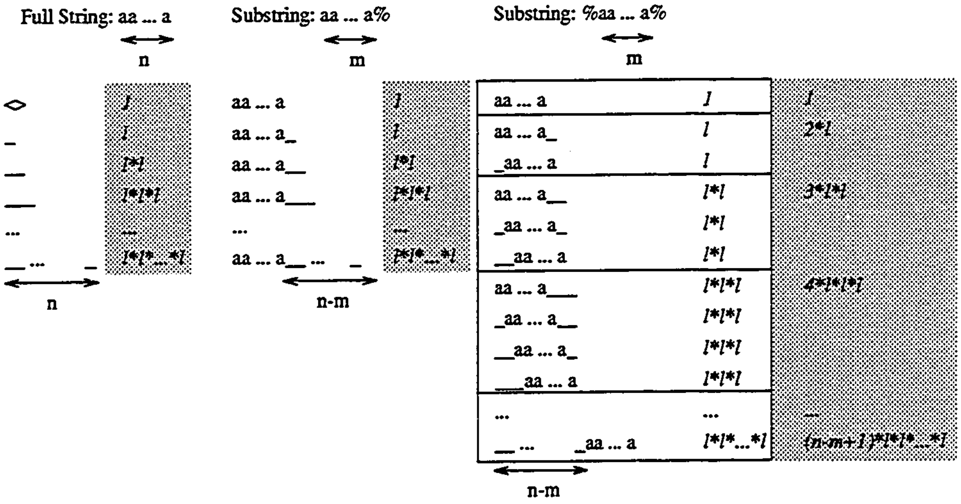


Figure 3-9: Restrictions of Various Substring Operations.

This equation is illustrated in the left part of Figure 3-9, where _ represents any character that can be a member of the string, and <> represents the empty string.

Consider the substring restriction operation of the form **aa ... a%**, where % stands for any string, including an empty string, **a** stands for any character defined in the substring, which must be a legal character for the original string, and the length of the substring is m. The center part of Figure 3-9 illustrates the resulting set of strings. The cardinality of this set is shown in formula 3.8.

$$S_{lead} = 1 + l + l^2 + \cdots + l^{n-m} = \sum_{j=0}^{n-m} l^j = \frac{l^{n-m+1}-1}{l-1}, n \geq 1, m \geq 0, n \geq n. \tag{3.8}$$

Consider more general operation – **%aa ... a%**. The right part of Figure 3-9 illustrates the result, and 3.9 gives its cardinality.

$$\begin{aligned} S_{general} &= 1 + 2 \cdot l + 3 \cdot l^2 + \cdots + (n-m+1) \cdot l^{n-m} = \sum_{j=0}^{n-m} (j+1) \cdot l^j = \\ &= \sum_{j=0}^{n-m} l^j + l \cdot \sum_{j=0}^{n-m-1} l^j + l^2 \cdot \sum_{j=0}^{n-m-2} l^j + \cdots + l^{n-m-1} \cdot \sum_{j=0}^{1} l^j + l^{n-m} \cdot \sum_{j=0}^{0} l^j = \\ &= \frac{l^{n-m+1}-1}{l-1} + l \cdot \frac{l^{n-m-1+1}-1}{l-1} + l^2 \cdot \frac{l^{n-m-2+1}-1}{l-1} + \cdots + l^{n-m} \cdot \frac{l-1}{l-1} = \\ &= \frac{(l^{n-m+1}-1) + (l^{n-m+1}-l) + (l^{n-m+1}-l^2) + \cdots + (l^{n-m+1}-l^{n-m})}{l-1} = \\ &= \frac{(n-m+1) \cdot l^{n-m+1} - \sum_{j=0}^{n-m} l^j}{l-1} = \frac{((n-m+1) \cdot (l-1) - 1) \cdot l^{n-m+1} + 1}{(l-1)^2}. \end{aligned} \tag{3.9}$$

Consider a query selection operator where user wants to select all flights whose origin starts with `'Ed'`, `flight.origin LIKE 'Ed%'`. Assume that the domain of `flight.origin` is a string whose maximum length is $n = 10$ characters, and that there are $l = 128$ possible character values. Then the total number of possible strings, according to Equation 3.7, is

$$S = \frac{128^{11}-1}{127} \approx 128^{10}$$

```
SELECT customer.name, customer.address
FROM customer
WHERE customer.city != 'Edmonton'  AND
        customer.country = 'Canada'
```

Customer
Customer #
Customer Name
Customer Address
Customer City
Customer Country

Figure 3-10: The query example.

The estimated number of selected strings is computed according to Equation 3.8 as

$$S_{lead} = \frac{128^9 - 1}{127} \approx 128^8$$

Then the selectivity factor of `flight.origin LIKE 'Ed%'` equals

$$SF = \frac{S}{S_{lead}} = \frac{(128^9 - 1) \cdot 127}{127 \cdot (128^{11} - 1)} \approx \frac{1}{16384}$$

3.5 Cost Estimation for Inter-site Single-Operator Queries

3.5.1 Single Union Inter-Site Query

Most of the previous work in query optimization in distributed databases considers enumeration of all possible permutations of operators in query expression. Such permutations are done based on equivalent transformations of unary and binary relational algebraic operations, e.g. [4] gives an example of enumeration of all possible query plans based on equivalent query transformations. Such transformations are also called algebraic transformations.

The standard way to approach the problem of choosing the optimal query plan is to have all equivalent plans resulting from query transformations enumerated, then to eliminate those that are obviously non-optimal based on some heuristics, to evaluate the cost of each of the remaining plans, and finally choose the plan with minimal cost among them.

Consider the query example shown in Figure 3-10. The query is expressed in an *SQL*-like form and has `SELECT`, `FROM`, and `WHERE` clauses. The relational model is assumed in this example, but the same example may be given for the object model. Note that the query involves only one relation, or an object class, *customer*.

Let us assume that after the query shown in Figure 3-10 has been submitted to the *DIOM* query processor, has undergone query routing and query decomposition, two information source sites registered with *DIOM* server are selected to answer this query as shown in Figure 3-11(a).

In DIOM, information sources may have heterogeneous representations for the same real world attributes. For example, class *customer* requested by the DIOM users may have different data representation in different information sources. DIOM delays the resolution of heterogeneity issues at the query packaging and result assembly stage. Therefore, at query optimization stage, we assume all subqueries are expressed uniformly.

The problem at the cost-based optimization phase is to find an optimal query execution plan that is most efficient for answering the query. A query execution plan is considered optimal when the cost associated with its realization is the lowest possible cost among all the plans that realize the given query. As was pointed out earlier (see section 3.1), the cost of a query plan is a function of three cost components, Equation 3.1. An optimal query execution plan then would be such that its cost is the minimum among the costs of all known plans that realize the query. Let us proceed with the analysis of our example query and try to apply some of the known query processing tactics to the given query.

A well-known tactic for optimizing *SQL*-like queries is to apply unary operations such as selection and projection before binary operations such as join, union whenever possible. This is mainly because

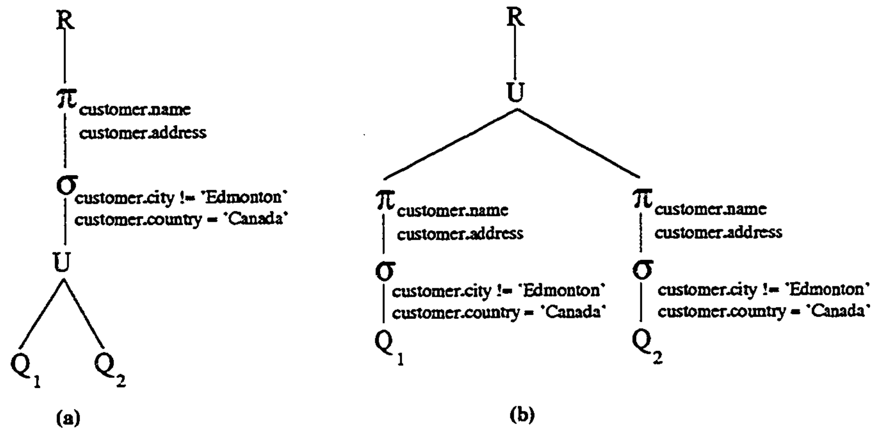

R
π customer.name customer.address
σ customer.city != 'Edmonton' customer.country = 'Canada'
U
Q1
Q2
(a)
R
U
π customer.name customer.address
σ customer.city != 'Edmonton' customer.country = 'Canada'
Q1
π customer.name customer.address
σ customer.city != 'Edmonton' customer.country = 'Canada'
Q2
(b)

Figure 3-11: The query tree resulted from query decomposition and routing.

selection operation reduces the size of the operands of binary operations fast, and thus the size of the binary operation result. Figure 3-11(b) shows the result of applying this heuristic. The unary operations of selection and projection are moved below the union operation, meaning that in the resulting query execution plan these operations will be executed before the union. The size of the input to the union operation will then be reduced by the selectivity factor of each of the selection and projection operators.

Previous studies in relational DBMS optimization, e.g., [18], have shown that moving the unary operations in the query tree towards the bottom of the tree is the necessary condition for producing an optimal query execution plan.

The given expression with unary operations pushed down may still result in a number of execution plans. The possibility to assign the binary query operators to different sites accounts for this multiplicity, while the feasibility of each particular placement constrains the number of the execution plans. For instance, let us assume that both Q_1 and Q_2 represent the *customer* subqueries on two different information sources, while result R of the query is expected at a remote third site. Then there are at least three possible realizations of this query tree, three execution plans:

1. placing union at the site of Q_1,
   - materialize Q_1 at its site, perform selection and projection there;
   - materialize Q_2 at its site, perform selection and projection there, and ship the result to the first site;
   - perform union at the first site, ship the result to the site where result is expected;
2. placing union at the site of Q_2,
   - materialize Q_1 at its site, perform selection and projection there, and ship the result to the second site;
   - materialize Q_2 at its site, perform selection and projection there;
   - perform union at the second site, ship the result to the site where result is expected;
3. placing union at the result delivery site,
   - materialize Q_1 at its site, perform selection and projection there, and ship the result to the result site;
   - materialize Q_2 at its site, perform selection and projection there, and ship the result to the result site;
   - perform union at the result site;

Thus the following factors are considered to establish all possible execution plans,

- the hypothetical possibility to assign the binary operation to a site;
- the set of all sites that are able to take the binary operation under their capability – such as the appropriate processing power, temporary disk space, and query and data definition compatibility;
- the set of sites immediately involved in the current query – all server sites pointed to after the query routing plus the client site where the result is expected.

The intersection of these sets will yield the tentative workspace where the optimizer will generate the possible execution plans. For example, the set of sites to be considered in our query optimization are site one, site two, and the client site to receive the results.

Let us assume that there are no constraints on any of the hypothetical site to take the union. There are three possible scenarios of query execution for this simple case and the costs associated with each of these scenarios.

1. Assume the result is expected at site one. The set of sites that will then possibly participate in query execution is {1,2}. Inclusion of any third-party sites will inevitably increase the communication costs to move data to and from them, thereby increasing the total execution cost. In fact, the increase is by at least twice the communication cost required to send the greater of Q_1 or Q_2. Left part of Figure 3-12 graphically shows how query operations are initially assigned to the two sites.
2. Assume the result is expected at site two. The disposition of sites and operators is exactly symmetrical to the one for site 1. The right part of Figure 3-12 reflects the situation.
3. Assume the result is expected at a site different from both site 1 and 2. In this case the center part of Figure 3-12 should be used.

At this point the cost-based optimization should decide which site the union operation should be assigned to, based on all the costs associated with each of the possible decisions. The optimal execution plan would be the one which offers the least cost and the fastest response time.

According to Equation 3.1, the three components of the overall cost have to be computed. Then they are added, each with the assigned weight factor indicating its perceived importance, to form the overall query execution cost.

The formulae for deriving the component costs for the scenarios shown in Figure 3-12 are given

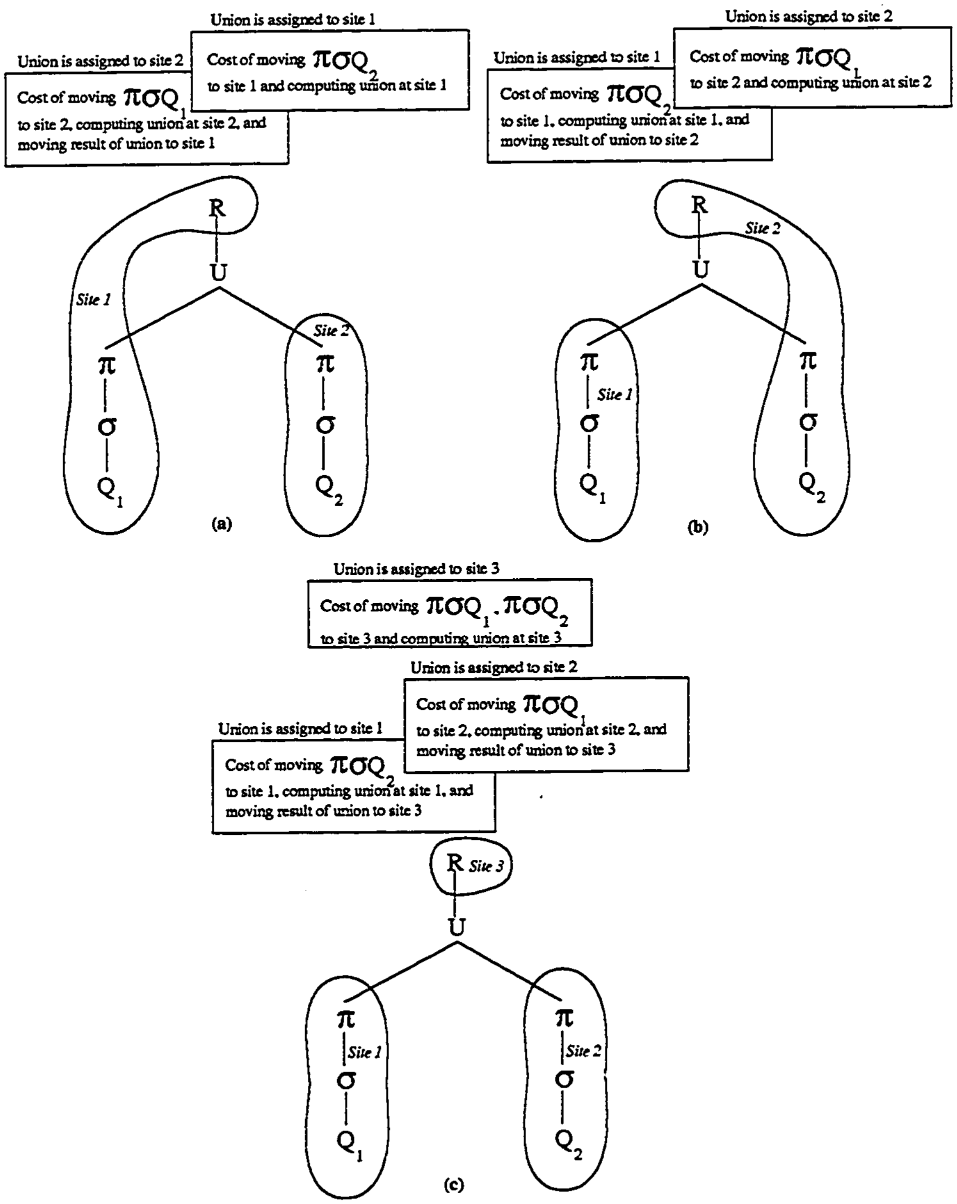

Union is assigned to site 1
Cost of moving $\pi\sigma Q_2$ to site 1 and computing union at site 1
Union is assigned to site 2
Cost of moving $\pi\sigma Q_1$ to site 2, computing union at site 2, and moving result of union to site 1
R
U
Site 1
Site 2
π
σ
Q_1
Q_2
(a)
Union is assigned to site 2
Cost of moving $\pi\sigma Q_1$ to site 2 and computing union at site 2
Union is assigned to site 1
Cost of moving $\pi\sigma Q_2$ to site 1, computing union at site 1, and moving result of union to site 2
R
U
Site 2
Site 1
π
σ
Q_1
Q_2
(b)
Union is assigned to site 3
Cost of moving $\pi\sigma Q_1 , \pi\sigma Q_2$ to site 3 and computing union at site 3
Union is assigned to site 2
Cost of moving $\pi\sigma Q_1$ to site 2, computing union at site 2, and moving result of union to site 3
Union is assigned to site 1
Cost of moving $\pi\sigma Q_2$ to site 1, computing union at site 1, and moving result of union to site 3
R Site 3
U
π
Site 1
Site 2
σ
Q_1
Q_2
(c)

Figure 3-12: Three possible site distributions for the single union query case: (a) – result is expected at site 1, (b) – at site 2, and (c) – at site 3.

in Equation 3.10.

$$
\begin{aligned}
C_{11} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) \\
C_{12} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{21} \cdot l(\cup(Q_1, Q_2)) \\
C_{21} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{12} \cdot l(\cup(Q_1, Q_2)) \\
C_{22} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) \\
C_{31} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{13} \cdot l(\cup(Q_1, Q_2)) \\
C_{32} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{23} \cdot l(\cup(Q_1, Q_2)) \\
C_{33} &= C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{13} \cdot l(Q_1) + cc_{23} \cdot l(Q_2) \\
\\
L_{ij} &= L(Q_1) + L(Q_2) + cl_{\cup j} \cdot (l(Q_1) \cdot l(Q_2)), \; i = 1,2,3, \; j = 1,2,3 \\
\\
R_{1\star}(Q_1) &= L(Q_1) + C_{1\star}(Q_1) \\
R_{2\star}(Q_2) &= L(Q_2) + C_{2\star}(Q_2) \\
R_{11} &= \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\cup 1} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{12} &= \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\cup 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{21} \cdot l(\cup(Q_1, Q_2)) \\
R_{21} &= \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\cup 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{12} \cdot l(\cup(Q_1, Q_2)) \\
R_{22} &= \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\cup 2} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{31} &= \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\cup 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{13} \cdot l(\cup(Q_1, Q_2)) \\
R_{32} &= \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\cup 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{23} \cdot l(\cup(Q_1, Q_2)) \\
R_{33} &= \max\{R_{1\star}(Q_1) + cc_{13} \cdot l(Q_1),\ R_{2\star}(Q_2) + cc_{23} \cdot l(Q_2)\} + cl_{\cup 3} \cdot (l(Q_1) \cdot l(Q_2))
\end{aligned}
\tag{3.10}
$$

where

C_{km} is the total communication cost for the case where the result is expected at site k and the union is assigned to site m;

L_{km} is the total local processing cost for the case where the result is expected at site k and the union is assigned to site m;

R_{km} is the total response time cost for the case where the result is expected at site k and the union is assigned to site m;

$C_{i\star}(Q_j)$ is the communication cost of delivering the result of Q_j to site i – this cost is necessary to express the recursive nature of the optimization problem;

$L(Q_i)$ is the local processing cost of obtaining the result of subquery Q_i – this cost also expresses the recursive nature of the problem and is the same for all scenarios of the current example;

cc_{ij} is the unit *communication cost* for transferring a unit of data from site i to site j;

$l(Q_i)$ is the length of the result of operator Q_i[1];

cl_{OPi} is the operator-OP-specific unit *cost of local* processing at site i.

Note that response time cost is computed according to possible parallelization of the tasks as a maximum of the two concurrent sequential subplans executing on one site.

The formulae given in Equation 3.10, on the one hand, provide a general means of computing the comprehensive cost of each of the possible query plans for the case of two-way inter-site union. On the other hand, however, these formulae do not give any clue in derivation of plan generating rules for this query case. To achieve the latter task, one needs to make assumptions concerning the inter-site communication and local on-site processing, similar to the ones made in [4],

- the cost of communication of one unit of data between any pair of sites in the distributed system is constant and the same for all sites in the distributed system, or $\forall\ i,j\ cc_{ij} = cc$, and

[1] Note that $l(\cup(Q_i, Q_j)) = \max(l(Q_i), l(Q_j))$.

- the cost of local query processing for same query operators is proportional to the size of the operands and the same for all sites in the distributed system, or $\forall\ i\ cl_{opi} = cl_{op}$.

These assumptions, in part, are unrealistic for real-life distributed systems. For example, the cost of communication depends on the current load of the network connection. The same cost increase is observed for the local processing cost whenever the CPU is on high demand. Ideally, a distributed query optimizer should provide the flexibility to accommodate the changing system parameters. However, when the actual cost parameters are not available, it is useful to make these assumptions.

Let us consider a case of three sites and the query plan selection rule given in Equation 3.11, which follows from the cost formulae of 3.10:

$$\begin{array}{lll} C_{11} \leq C_{12} & L_{11} = L_{12} = L & R_{11} \leq R_{12} \\ C_{22} \leq C_{21} & L_{22} = L_{21} = L & R_{22} \leq R_{21} \\ C_{33} \leq C_{31}, C_{32} & L_{33} = L_{31} = L_{32} = L & R_{33} \leq R_{31}, R_{32}, \end{array} \tag{3.11}$$

These rule amount to say that it is always beneficial if the union is assigned to the same site as the site where the result if the query is expected.

Note that $\max(a,b) \geq a$, $\max(a,b) \geq b$, and $\max\{a,b\} + c = \max\{a+c, b+c\}$, where c is the cost of the inter-site moving of the result of the union.

The best query plans for the scenarios of Figure 3-12(a), (b), and (c) are 11, 22, and 33 respectively. In other words, as a rule, for queries that involve two-way binary union, the optimal execution plan is to perform the union at the site where the result is expected. This observation may be further used as a heuristic for such queries.

Consider the example query in Figure 3-10. Denote customer subquery at site 1 as Q_1, and customer subquery at site 2 as Q_2, and assume that the following statistics are true for Q_1 and Q_2:

$$\begin{array}{rcl rcl} card(Q_1) & = & 1000 & card(Q_2) & = & 5000 \\ Icard(Q_1.country) & = & 12 & Icard(Q_2.country) & = & 52 \\ Icard(Q_1.city) & = & 100 & Icard(Q_2.city) & = & 1000 \\ Icard(Q_1.address) & = & 1000 & Icard(Q_2.address) & = & 4985 \\ Icard(Q_1.name) & = & 991 & Icard(Q_2.name) & = & 4993 \\ width(customer.name) & = & 20 & width(customer.address) & = & 40 \end{array}$$

Applying the selectivity factors from Tables 3-1,3-2,3-3 obtain,

$$card(\pi(\sigma(Q_1))) = \frac{card(Q_1)(Icard(Q_1.city)-1)}{Icard(Q_1.city)Icard(Q_1.country)} \approx 83$$

$$l(Q_1) = 83 \cdot (width(customer.name) + width(customer.address)) = 4960$$

$$card(\pi(\sigma(Q_2))) = \frac{card(Q_2)(Icard(Q_2.city)-1)}{Icard(Q_2.city)Icard(Q_2.country)} \approx 97$$

$$l(Q_2) = 97 \cdot (width(customer.name) + width(customer.address)) = 5820$$

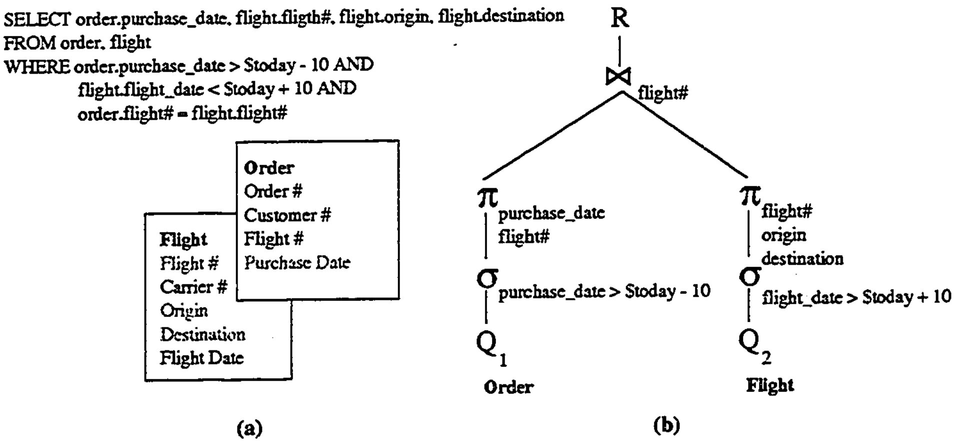
$$l(\cup(Q_1, Q_2)) = \max[l(Q_1), l(Q_2)] = 5820$$

Then, substituting the values in Equation 3.10 and assuming the unit communication cost to be one and the unit local cost to be 0.0001 obtain,

$$\begin{array}{lll} C_{11} = 5820 & C_{12} = 10780 & \\ C_{21} = 11640 & C_{22} = 4960 & \\ C_{31} = 11640 & C_{32} = 10780 & C_{33} = 10780 \end{array}$$

$$L_{ij} = 2887 \quad i = 1,2,\ j = 1,2 \quad i = 3,\ j = 1,2,3$$

$$\begin{array}{lll} R_{11} = 8707 & R_{12} = 13667 & \\ R_{21} = 14527 & R_{22} = 7847 & \\ R_{31} = 14527 & R_{32} = 13667 & R_{33} = 8707 \end{array}$$



```
SELECT order.purchase_date, flight.fligth#, flight.origin, flight.destination
FROM order, flight
WHERE order.purchase_date > $today - 10 AND
        flight.flight_date < $today + 10 AND
        order.flight# = flight.flight#
```

Figure 3-13: The single join query example.

Thus the lowest communication costs of 5820, 4960, and 10780 are observed for the plans C_{11}, C_{22}, and C_{33} respectively, which correspond to three best plans for each of the three site distributions. Because of simplicity of this example local processing cost is invariant. The response time cost is lowest for R_{11}, R_{22}, and R_{33} plans, which are three optimal plans for each of the distributions.

The total cost of each of the query plans may then be obtained by substituting components C, L, and R into Equation 3.1 on page 14. The total cost also depends on the user's perception of the importance of each of these components, which is defined by the weight factors for each of the components, see section 3.1.

The assumption we made about the ratio of local processing and communication costs would not have changed the optimization decision, it would only affect the actual figures. Therefore the rule obtained given in Equation 3.11 may be used for choosing an optimal plan.

However, since this rule is based on the assumptions of equal unit communication and equal local processing costs, there may be cases where it will not lead to the minimum cost. For instance, if local processing at site one is a lot more expensive than on site two, $cl_{\cup_1} >> cl_{\cup_2}$, possibly due to high CPU load averages, then the local processing component may affect the total cost given in Equation 3.1 in a way that plans with processing on site one will be more expensive than those with processing on site two. Therefore, applying the rule given in Equation 3.11 will not result in the best execution plan. In this case the best plans will place the binary union operation on site two.

In short, without the assumptions of equal unit communication cost and equal local operator processing cost the above rule does not hold, and one needs to compute the corresponding cost for each case, compare them, and select the most optimal query execution plan.

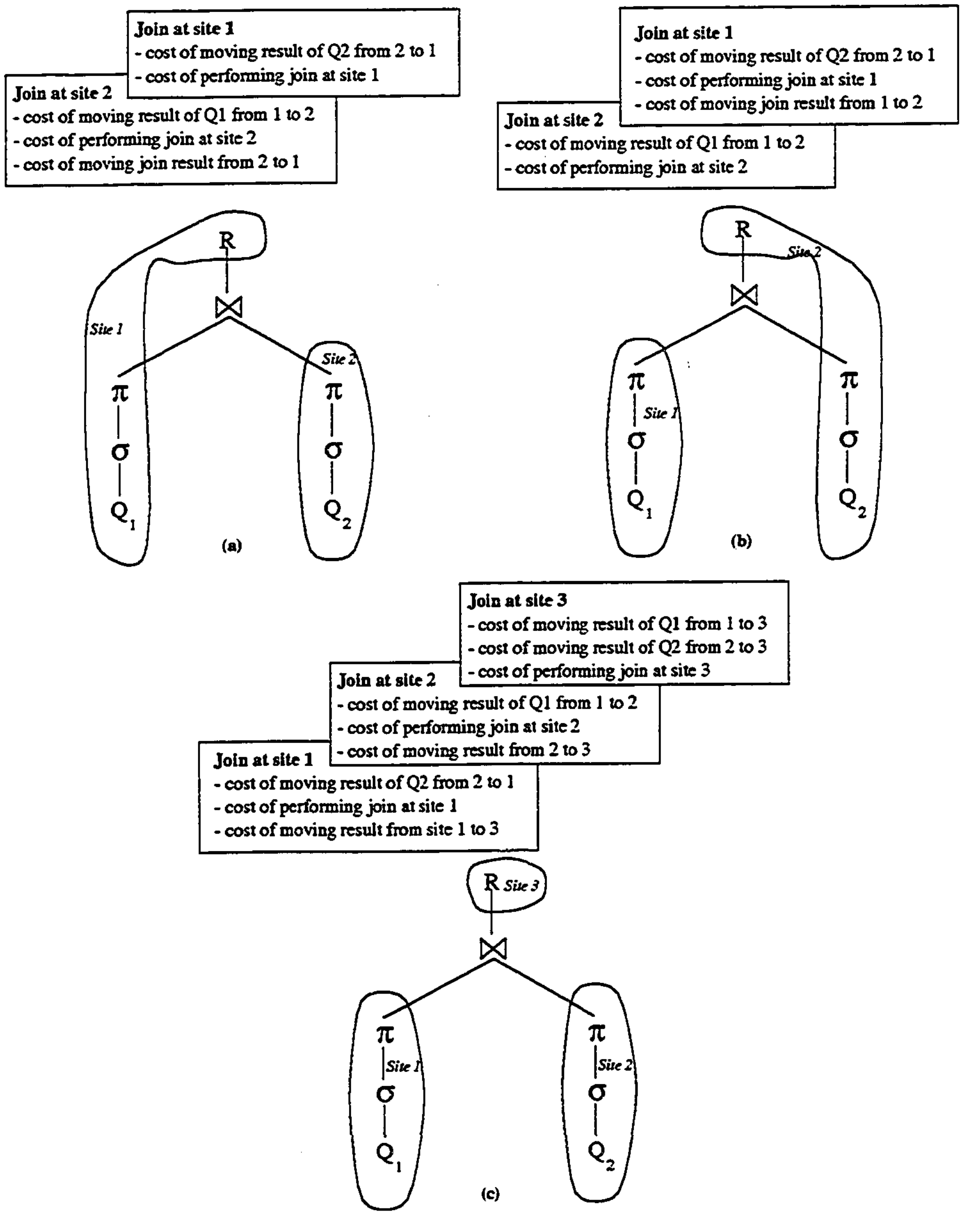
3.5.2 Single Join Query Example

Consider a query containing a single join shown in Figure 3-13(a):

> *find the date of purchase, flight number, origin, and destination for all flights booked fewer than 10 days ago and flying within the next 10 days.*

The tree corresponding to this query is shown in Figure 3-13(b).

As in the single union example, we will present the possible site distributions for this query processing task, with the costs associated with each of the scenario. The scenarios and the costs associated with them are laid out in Figure 3-14.



Join at site 1
- cost of moving result of Q2 from 2 to 1
- cost of performing join at site 1

Join at site 2
- cost of moving result of Q1 from 1 to 2
- cost of performing join at site 2
- cost of moving join result from 2 to 1

Join at site 1
- cost of moving result of Q2 from 2 to 1
- cost of performing join at site 1
- cost of moving join result from 1 to 2

Join at site 2
- cost of moving result of Q1 from 1 to 2
- cost of performing join at site 2

Join at site 3
- cost of moving result of Q1 from 1 to 3
- cost of moving result of Q2 from 2 to 3
- cost of performing join at site 3

Join at site 2
- cost of moving result of Q1 from 1 to 2
- cost of performing join at site 2
- cost of moving result from 2 to 3

Join at site 1
- cost of moving result of Q2 from 2 to 1
- cost of performing join at site 1
- cost of moving result from site 1 to 3

Figure 3-14: Three possible site distributions for the single join query case: (a) – result is expected at site 1, (b) – at site 2, and (c) – at site 3.

$$
\begin{array}{l}
C_{11} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) \\
C_{12} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{21} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{12} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{22} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) \\
C_{31} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{13} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{32} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{23} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{33} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{13} \cdot l(Q_1) + cc_{23} \cdot l(Q_2) \\
\\
L_{ij} = L(Q_1) + L(Q_2) + cl_{\bowtie j} \cdot (l(Q_1) \cdot l(Q_2)),\ i = 1,2,3,\ j = 1,2,3 \\
\\
R_{1\star}(Q_1) = L(Q_1) + C_{1\star}(Q_1) \\
R_{2\star}(Q_2) = L(Q_2) + C_{2\star}(Q_2) \\
R_{11} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{12} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{21} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{12} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{22} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{31} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{13} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{32} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{23} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{33} = max\,\{R_{1\star}(Q_1) + cc_{13} \cdot l(Q_1),\ R_{2\star}(Q_2) + cc_{23} \cdot l(Q_2)\} + cl_{\bowtie 3} \cdot (l(Q_1) \cdot l(Q_2))
\end{array}
\tag{3.12}
$$

The single join case, unlike the single union, does not present a uniform rule for selecting the best execution plan. The difference is that the estimated size of the result of join depends not only on the size of its operands, like in union, but also on the selectivity of the join condition and the statistical information about the operands. In the worst scenario, when the selectivity is one – every tuple from the first relation is joined with all tuples from the second relation – the result is Cartesian product of the operands, whose cardinality is the product of the cardinalities of the operands. In the best scenario, when the selectivity is approaches zero – only one tuple from each of the input relations is selected to form the result, whose cardinality is one.

The cost optimizer must estimate the cardinality and attribute cardinality of the result of the join to make the decision about the best plan. We have covered the computation of these statistical estimates for the join operator in section 3.4.2, Equations 3.4 and 3.5, now let us consider a concrete example and apply these formulae to the case shown in Figure 3-14(c) on page 39. Let *pd* denote *purchase_date*, *fd* denote *flight_date*, *f* denote *flight#*, *o* and *d* denote *origin* and *destination* respectively, and let the relevant statistical information be the following:

$card(Q_1)$	=	1000	$card(Q_2)$	=	200
$Icard(Q_1.f)$	=	100	$Icard(Q_2.f)$	=	200
$Icard(Q_1.pd)$	=	10	$Icard(Q_2.fd)$	=	20
$\min(pd)$	=	1997/01/01	$\max(pd)$	=	1997/04/01
$\min(fd)$	=	1997/01/01	$\max(fd)$	=	1997/12/31
$Icard(Q_2.o)$	=	50	$Icard(Q_2.destination)$	=	40
$width(f)$	=	4	*$today*	=	1997/03/15
$width(pd)$	=	8	$width(fd)$	=	8
$width(o)$	=	20	$width(d)$	=	20

Let $l(S)$ in this context denote the length of relation S. The length, or size, of a relation is the product of its width and height, where width is the sum of the domain widths of the attributes of S and height is the cardinality of S. Based on the selectivity factor for > predicate given in Table 3-3 on page 28, let us first compute the cardinalities of the operands of join:

$$\begin{aligned} \mathit{width}\,(\pi\sigma(Q_1)) = \mathit{width}(pd) + \mathit{width}(f) &= 12 \\ \mathit{card}\,(\pi\sigma(Q_1)) = \tfrac{\mathit{card}(Q_1)\cdot(\max(pd)-(\$today-10))}{\max(pd)-\min(pd)} &= 217 \\ l\,(\pi\sigma(Q_1)) &= 2604 \end{aligned}$$

$$\begin{aligned} \mathit{width}\,(\pi\sigma(Q_2)) = \mathit{width}(f) + \mathit{width}(o) + \mathit{width}(d) &= 44 \\ \mathit{card}\,(\pi\sigma(Q_2)) = \tfrac{\mathit{card}(Q_2)\cdot(\max(fd)-(\$today+10))}{\max(fd)-\min(fd)} &= 154 \\ l\,(\pi\sigma(Q_2)) &= 6776 \end{aligned}$$

To compute the parameters of the join result R we need to compute $Icard\,(\pi\sigma(Q_1.f))$ and $Icard\,(\pi\sigma(Q_2.f))$, which requires the use of Equation 3.5. To put the probabilistic problem in our language, 1000 tuples of Q_1 contained 100 different values of *flight#* attribute, how many different values of this attribute will be left if 217 tuples are randomly selected from Q_1. Substituting these values into Equation 3.5, obtain

$$Icard\,(\pi\sigma(Q_1.f)) = 100\left[1 - \prod_{i=1}^{217}\left(1 - \frac{1000}{100\cdot(1001-i)}\right)\right] \approx 92$$

Analogously for Q_2 we have 200 tuples with 200 different values of *flight#* and we randomly select 154 tuples. Substituting this into Equation 3.5, obtain

$$Icard\,(\pi\sigma(Q_2.f)) = 200\left[1 - \prod_{i=1}^{154}\left(1 - \frac{1}{201-i}\right)\right] = 154$$

Note that the last result is absolutely predictable because attribute *flight#* is apparently the primary key for relation **Flight**.

Now we are finally ready to compute the parameters of the result of join. Substituting the parameters computed earlier into Equation 3.4, obtain

$$\begin{aligned} \mathit{width}(R) = \mathit{width}(pd) + \mathit{width}(f) + \mathit{width}(o) + \mathit{width}(d) &= 52 \\ \mathit{card}(R) = \tfrac{\mathit{card}(\pi\sigma(Q_1))\cdot \mathit{card}(\pi\sigma(Q_2))}{\max\{Icard(\pi\sigma(Q_1.f)), Icard(\pi\sigma(Q_2.f))\}} = \tfrac{217\cdot 154}{154} &= 217 \\ l(R) = \mathit{card}(R)\cdot \mathit{width}(R) &= 11284 \end{aligned}$$

At this point all the parameters needed for computation of cost components have been obtained. Note that we consider the query shown in Figure 3-14(c) where the relations **Flight** and **Order** are located on the separate source sites (*Site 1* and *Site 2*) and the result R of join is expected at the client site (*Site 3*). We assume that the source relations are readily available therefore $C_{1\star}(Q_1) = 0$, $C_{2\star}(Q_2) = 0$, and $L(Q_1) = L(Q_2) = 0$. We also assume that the unit communication and local processing costs are the same for all sites and communication links in the network, the unit communication cost is assumed 1, and the unit local processing cost is assumed 0.0001 as in the single union query example presented earlier. Then, substituting the precomputed parameters into the formulae for cost components of Equation 3.12, obtain

$$\begin{aligned} &C_{31} = 18060 \quad C_{32} = 13388 \quad C_{33} = 9380 \\ &L_{3j} = 1765 \\ &R_{31} = 19825 \quad R_{32} = 15653 \quad R_{33} = 8541 \end{aligned}$$

We can see from this example that the lowest cost is achieved by the plan that places the join operation on the client site, mainly because the result of join has the size that is greater than the sum of the sizes of its operands, which dominates the communication costs of the two other plans.

However, despite the sheer advantage the third plan shows in this example, we may not possibly generalize it to the level of universally applicable heuristic. Indeed, if the size of the result of join were not greater than the sum of the sizes of its operands, then the outcome would be quite different.

Let us try to list some observations based on the cost formulae of Equation 3.12 and the assumptions we made for this example using the terms of that equation:

$$\begin{array}{rcl} l(Q_2) < l(\bowtie(Q_1,Q_2)) & \rightarrow & C_{11} < C_{12}, R_{11} < R_{12}, \quad C_{33} < C_{32}, R_{33} < R_{32} \\ l(Q_2) < l(Q_1) & \rightarrow & C_{11} < C_{12}, \quad R_{11} < R_{12} \\ l(Q_1) < l(\bowtie(Q_1,Q_2)) & \rightarrow & C_{22} < C_{21}, R_{22} < R_{21}, \quad C_{33} < C_{31}, R_{33} < R_{31} \\ l(Q_1) < l(Q_2) & \rightarrow & C_{22} < C_{21}, \quad R_{22} < R_{21} \\ l(Q_1) + l(Q_2) < l(\bowtie(Q_1,Q_2)) & \rightarrow & C_{11} < C_{12}, \quad C_{22} < C_{21}, \quad C_{33} < C_{31}, C_{32} \\ & & R_{11} < R_{12}, \quad R_{22} < R_{21}, \quad R_{33} < R_{31}, R_{32} \end{array} \tag{3.13}$$

As in the case of single union, the assumption of equal local processing power among sites leads to the observation that $\forall i,j \quad L_{ij} = L$. This also leads to a possible reduction in computation – local query processing cost may be left out of the process of optimization because they just uniformly offset the total cost, which does not affect the decision of the best execution plan.

In the last two sections we have presented the two simple query cases and used examples to illustrate the cost-based optimization process. In the next section we extend our discussion to inter-site queries and give a general formula for estimating the cost of query plans.

3.5.3 Cost Functions for Inter-site Queries

3.5.3.1 General Cost Estimation

As we notice in the previous two sections, the formulae for cost estimation may be generalized in such a way that the specifics of the binary operator considered are taken out of computation and are localized in operator-specific factors. Equation 3.14 is a general formula to compute costs of any binary query operator. The operator-specific information is localized in the following factors

$l(OP(Q_i,Q_j))$ – the size of the result of applying operator OP to the results of queries Q_i, Q_j;

cl_{OPi} – the unit cost of local processing to perform operator OP at site i;

$$\begin{array}{l} C_{11} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) \\ C_{12} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{21} \cdot l(OP(Q_1,Q_2)) \\ C_{21} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{12} \cdot l(OP(Q_1,Q_2)) \\ C_{22} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) \\ C_{31} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{13} \cdot l(OP(Q_1,Q_2)) \\ C_{32} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{23} \cdot l(OP(Q_1,Q_2)) \\ C_{33} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{13} \cdot l(Q_1) + cc_{23} \cdot l(Q_2) \\ \\ L_{ij} = L(Q_1) + L(Q_2) + cl_{OPj} \cdot (l(Q_1) \cdot l(Q_2)), \ i = 1,2,3, \ j = 1,2,3 \\ \\ R_{1\star}(Q_1) = L(Q_1) + C_{1\star}(Q_1) \\ R_{2\star}(Q_2) = L(Q_2) + C_{2\star}(Q_2) \\ R_{11} = \max\{R_{1\star}(Q_1), \ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{OP1} \cdot (l(Q_1) \cdot l(Q_2)) \\ R_{12} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{OP2} \cdot (l(Q_1) \cdot l(Q_2)) + \\ \qquad + cc_{21} \cdot l(OP(Q_1,Q_2)) \\ R_{21} = \max\{R_{1\star}(Q_1), \ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{OP1} \cdot (l(Q_1) \cdot l(Q_2)) + \\ \qquad + cc_{12} \cdot l(OP(Q_1,Q_2)) \\ R_{22} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{OP2} \cdot (l(Q_1) \cdot l(Q_2)) \\ R_{31} = \max\{R_{1\star}(Q_1), \ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{OP1} \cdot (l(Q_1) \cdot l(Q_2)) + \\ \qquad + cc_{13} \cdot l(OP(Q_1,Q_2)) \\ R_{32} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{OP2} \cdot (l(Q_1) \cdot l(Q_2)) + \\ \qquad + cc_{23} \cdot l(OP(Q_1,Q_2)) \\ R_{33} = \max\{R_{1\star}(Q_1) + cc_{13} \cdot l(Q_1), \ R_{2\star}(Q_2) + cc_{23} \cdot l(Q_2)\} + cl_{OP3} \cdot (l(Q_1) \cdot l(Q_2)) \end{array} \tag{3.14}$$

To illustrate the notation in Equation 3.14 let us use the query decomposition tree shown in Figure 3-13(b). Suppose the leaf of the tree marked Q_1 is, in fact, a subtree that contains another binary query operator and its two operands denoted as Q_{11} and Q_{12}. This means that the costs of obtaining the result of Q_1 at site 1, which are denoted here as $C_{1*}(Q_1)$ for communication, $L(Q_1)$ for local processing, and $R_{1*}(Q_1)$ for response time costs, may not be assumed zero as was done in previously considered examples of union and join cases.

But the overall optimization problem hinges on computing the costs for the alternative query execution plans and selecting the plan with the lowest cost, therefore the underlying costs must be computed as well. Suppose now that Q_{11} is also a subquery that has a binary operator. Then the computation must propagate until it has reached the leaf nodes of the query decomposition tree. Each of the levels of the tree represents a subquery whose communication, local processing, and response time costs need to be computed and the optimal execution plan generated before the query at the higher level may commence the cost-based optimization.

Equation 3.14 gives the general formulae for computing the costs of the query execution plans at any level of the decomposition tree. In the following section we explain how the optimal execution plan may be generated for the decomposition trees that have more than one level of query operators.

3.5.3.2 The Recursive Approach

As indicated earlier, the total cost of a simple subquery depends on the costs of obtaining the inputs to this subquery. The process of cost optimization starts at the top of the given decomposition tree, having as a parameter, only the top part of the decomposition tree. If the decomposition tree is a binary tree, i.e., if each of its operator nodes has two subsequent nodes, then the process of optimization may be organized as a downward traversal of the entire decomposition tree by the optimizer. At each step of the traversal the optimizer analyses the leaf nodes of the current subtree. If one of the nodes is itself a subtree, then the optimizer recursively invokes another optimizer, a copy of itself, and passes the subtree to it as the optimization task. As soon as all the invoked optimizers with subtrees have returned, the optimizer may assemble the result and pass it back, one level up, to the optimizer that invoked it.

Therefore, the search for optimal query plan may start at the top of the decomposed query tree and then be recursively developed as each branch of the tree presents multiple choices of concrete query plans.

Then at each step of optimization the algorithm may try to apply the precomputed solutions, like the ones worked out in the previous two sections thus speeding the process.

To formalize the approach of applying the above optimization recursive decomposition we introduce a predicate D, which stands for *decomposable.* This predicate applies to each of the operator nodes of the decomposition tree and indicates whether the node of the tree is a decomposable subtree or a leaf node that is assigned to the information source. Although a leaf node of the decomposition tree may itself be a subquery, since it does not involve inter-site query operations, DIOM treats it as non-decomposable because local query processing is not affected by the query execution plan.

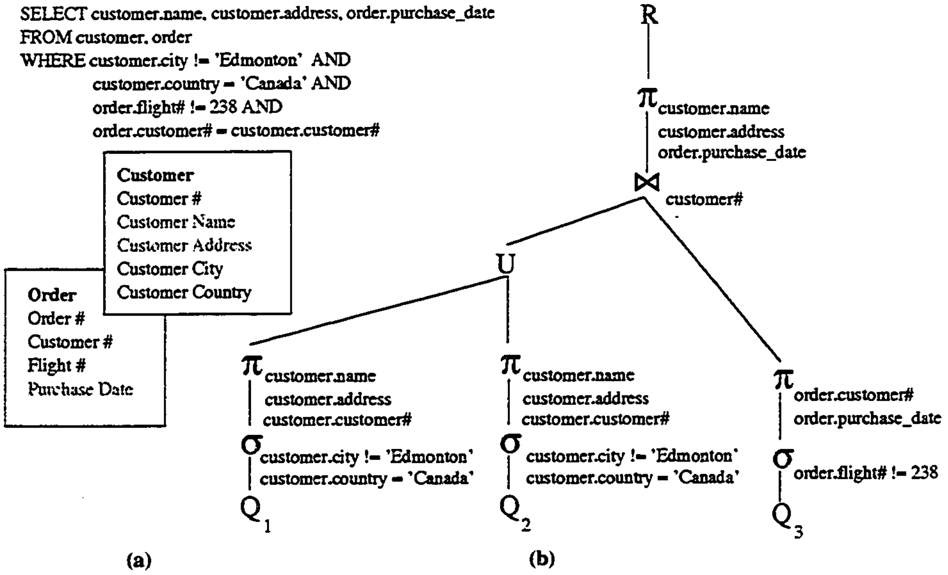
3.5.3.3 An Example

Consider the query shown in Figure 3-15(a):

> *find names and addresses of all customers living outside Edmonton, Canada, and the date that they booked a flight except for flight number* 238.

It contains two inter-site binary operators. Suppose that DIOM `customer` objects are present at both site one and two, and `order` objects are present on site 1 only. Assume the query decomposition tree for this query is as shown in Figure 3-15(b), where Q_1 and Q_2 denote customer subqueries on sites 1 and 2 respectively, and Q_3 denotes order subquery at site 1.

Let us consider the problem in a top-down manner, starting the optimization process from the top of the site distribution tree. The top part of the query tree, combined with the given site distribution represents the problem considered in the previous section.



```
SELECT customer.name, customer.address, order.purchase_date
FROM customer, order
WHERE customer.city != 'Edmonton' AND
        customer.country = 'Canada' AND
        order.flight# != 238 AND
        order.customer# = customer.customer#
```

Customer
Customer #
Customer Name
Customer Address
Customer City
Customer Country
Order
Order #
Customer #
Flight #
Purchase Date
R
π customer.name customer.address order.purchase_date
⋈ customer#
U
π customer.name customer.address customer.customer#
σ customer.city != 'Edmonton' customer.country = 'Canada'
Q1
π customer.name customer.address customer.customer#
σ customer.city != 'Edmonton' customer.country = 'Canada'
Q2
π order.customer# order.purchase_date
σ order.flight# != 238
Q3

(a) (b)

Figure 3-15: The single join and union query example.

The location of Q_1, the intermediate result of the union, is not assigned to any site at this stage. The cost optimizer has to consider all possible placements and work out the one leading to the minimum overall cost of query execution. The approach we may take that would allow the optimizer to consider all possible scenarios is a *depth-first* search based on recursion. First, the optimizer is called with the task of finding the optimal execution plan for the top node of the decomposition tree. It should analyze the tree and detect that one of the leaf nodes is a decomposable subquery. Then the optimizer should suspend its execution and determine whether the decomposable leaf node has been assigned to a site. If yes, the optimizer should call a copy of itself with new task – to find the best execution plan for the decomposable subquery. If not, then the optimizer should replicate itself as many times as there are possible site assignments to this node. Then for each one of them the leaf node is fixed and the optimizer should call a copy of itself with the corresponding subquery. And so forth, until the optimizer does not detect any decomposable leaf nodes in the given parameter.

This type of depth-first search is based on recursion and the final result is available when the last of the initially invoked optimizers returns with its version of the best execution plan and its cost.

Formally speaking, on identifying the binary operator and its inputs, we consider all possible placements of these inputs on all sites capable of accommodating them. Each of the options thus results in either a well defined inter-site optimization subproblem, a kind discussed previously, or in a trivial single-site problem whose cost reduces to computing the local processing cost of the binary operator in question.

Coming back to our example, consider the initial site distribution shown in Figure 3-16. **customer** objects are present at two different sites, site 1 and site 2, and **order** objects are all located at site 1. This distribution is one of the possible distributions that may be derived from the decomposition tree shown in Figure 3-15.

Let us consider the options for the top of the query.

1. Q_1 is placed at site 1 – we have the following situation,

   (a) ⋈ subquery turns into a single-site optimization task, whose solution is trivially found by assigning the binary operator to the very site all the components are present at, i.e., site 1 in this case. See formulae below for the detailed cost formulation;

   (b) ∪ subquery is the case when the union result as well as the first input are placed on site 1, while the second input (result of Q_{12}) is at site 2. This case has been considered

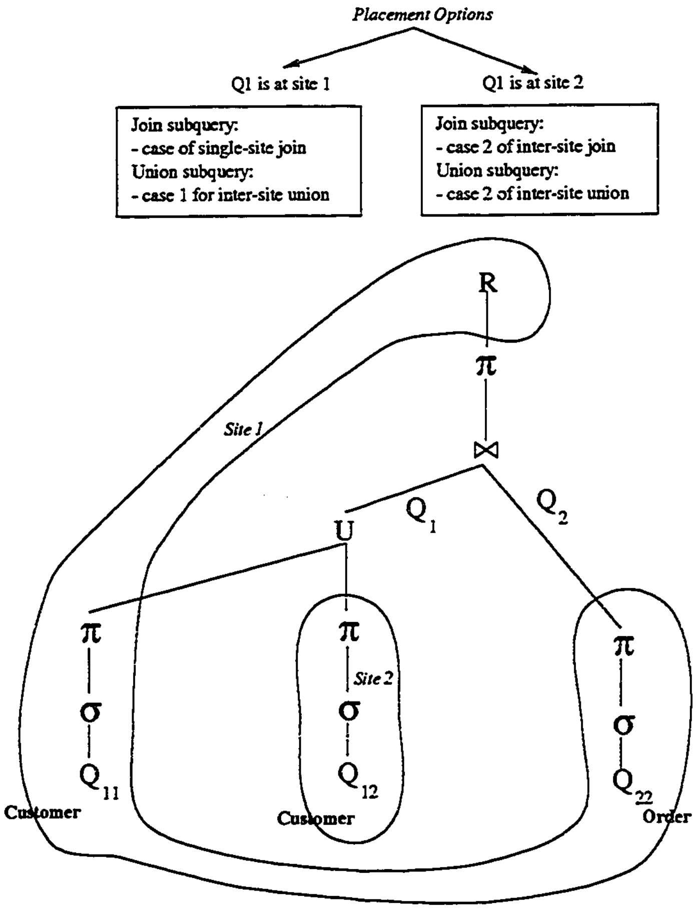


Figure 3-16: Layouts of costs for the first site distributions.

previously and is shown in Figure 3-12(a) on page 35;

2. Q_1 is at site 2,

   (a) $\bowtie$ subquery represents the site distribution in which the first input (Q_1) is placed at site 1, and the second input as well as the result of the join are placed at site 2. The site distribution is shown in Figure 3-14(b) on page 39;

   (b) $\cup$ subquery is similar to that shown in Figure 3-12(b) on page 35 – the second input and the result of union are at site 2, while the first input is at site 1.

Let us consider the costs associated with each of the subqueries for both options. Let us denote, for the purpose of this example, C^i the total communication cost for the scenario when the intermediate result is assigned to site i. In general, if there is more than one intermediate result whose placement is not fixed, then the costs will be denoted with subscript $C^{i_1 i_2 \dots i_n}$ where $i_1 \dots i_n$ are the respective placements of these intermediate results, and n is the number of them. Equation 3.15 gives the formulae for the costs for the $\bowtie$ subquery.

$$\begin{aligned}
C^1_{11} &= C_{1\star}(Q_1) + C_{1\star}(Q_2) \\
L^1_{ij} &= L(Q_1) + L(Q_2) + cl_{\bowtie j} \cdot (l(Q_1) \cdot l(Q_2)),\ i = 1,\ j = 1,2 \\
R^1_{11} &= \max\{C_{1\star}(Q_1) + L(Q_1),\ C_{1\star}(Q_2) + L(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) \\
\\
C^2_{11} &= C_{2\star}(Q_1) + C_{1\star}(Q_2) + cc_{21} \cdot l(Q_1) \\
C^2_{12} &= C_{2\star}(Q_1) + C_{1\star}(Q_2) + cc_{12} \cdot l(Q_2) + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) \\
L^2_{ij} &= L(Q_1) + L(Q_2) + cl_{\bowtie j} \cdot (l(Q_1) \cdot l(Q_2)),\ i = 1,\ j = 1,2 \\
R^2_{11} &= \max\{C_{2\star}(Q_1) + L(Q_1) + cc_{21} \cdot l(Q_1),\ C_{1\star}(Q_2) + L(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) \\
R^2_{12} &= \max\{C_{2\star}(Q_1) + L(Q_1),\ C_{1\star}(Q_2) + L(Q_2) + cc_{12} \cdot l(Q_2)\} + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) + \\
&\quad + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2))
\end{aligned} \tag{3.15}$$

where

C^k_{ij} is the communication cost for the scenario when Q_1 has been placed on site k, the join result is expected at site i, and the join has been assigned to site j;

L^k_{ij} – similar to the previous item but for the total local processing cost;

R^k_{ij} – same as above but for the total response time cost;

$C_{i\star}(Q_j)$ is the total communication cost incurred in delivering the result of Q_j to site i. This cost is either decomposable, in which case it equals to the total subcost of the corresponding scenario it decomposes to, or, if not decomposable, either is equal to zero – for the cases when Q_j is already at site i, or is equal to the cost of moving, that is, $c_{ki} \cdot l(Q_j)$, where k is the site where Q_j currently resides;

$L(Q_i)$ is the total local processing cost incurred in obtaining the result of Q_i. This cost is either decomposable, in which case it equals to the total local processing subcost of the corresponding scenarios, or, if not decomposable, equals to the cost of local processing at the site where it currently resides, which means that this cost may be mutually reduced from all cost estimations;

As shown in Equation 3.15, the costs expressed in $C_{1\star}(Q_1)$, $C_{2\star}(Q_1)$, and $C_{1\star}(Q_2)$, as well as $L(Q_1)$ and $L(Q_2)$ are not explicitly defined. This means that the query processing algorithm, at this stage, needs to deepen the search into the second layer of the query tree and consider the subqueries whose roots are Q_1 and Q_2.

For each of the queries on the second level, we need to disclose the following information:

- whether the query is decomposable into an inter-site optimization task, i.e., whether it is indeed a root of some subtree or have we reached a leaf node of the tree, and

- if the query is in fact decomposable then we need to define a new subquery and consider the new optimization task, whose solution will be the input to the formulae given in Equation 3.15.

It is here that the recursive nature of the problem is truly revealed. However, being recursive does not mean that the search necessarily has to reach every leaf node in the query tree. Note the heuristic rules discovered for the case with union summarized in Equation 3.11 on page 37, as well as for the case with join given in Equation 3.13 on page 42.

In this chapter we have described the theoretical foundations and the proposed framework for distributed query processing and optimization. The next chapter is dedicated to the system analysis and design specifics of our proposed system as well as to the implementation points worth noting.

Chapter 4

System Analysis, Design, and Implementation Issues

In this chapter we will discuss the issues involved in the process of design and implementation of the *DIOM Query Scheduling Utility.* As any software system [17], the fundamental steps of this engineering process require analysis, design, and the actual coding of the software. The task of implementing the *DIOM Query Scheduling Utility* can be seen as a demonstration of viability of the ideas and principles presented in Chapter 3.

The theoretical results obtained and shown in Chapter 3 also serve as the baselines for the software development covered in this chapter. Thus we omit the technical feasibility study phase of software engineering and start from system requirements analysis.

4.1 System Requirements Analysis

The success of a software project depends to great extent on how well and thoroughly the requirements to the projected software have been analysed. The requirements analysis is a top-down, or general-to-particular process, in which software requirements are gradually refined and finalized in some form that is acceptable to the next phase, software design [17].

4.1.1 Analysis of Non-Functional Requirements

4.1.1.1 Portability

Since the current implementation is a component of a greater project, there are certain portability requirements that need to be fulfilled. The overall DIOM framework implemented in [10] exemplifies such requirements by using a highly portable *html-cgi-oraperl* combination of programming environment. Our goal is to create a value-added software package for distributed query scheduling, which is portable across platforms, and can be run by the applications via a network connection without a need for installation or compilation. All the necessary components must be downloadable and executable, possibly with the use of the necessary application viewing tool that is platform-specific but ubiquitous enough to be present at most platforms.

We need to provide a graphical user interface, which uses such standard interface components as control buttons, text components, drawable components, etc. Portability of such interfaces, at least for major platforms, must be achieved without having to implement separate versions of GUI for each platform.

4.1.1.2 Extensibility to Add New Heuristics or Cost Parameters

The proposed implementation software is an experiment – it does not cover all aspects of query processing and optimization. However, it must provide an extensible framework that would allow to add new heuristics and cost processing modules with a reasonable flexibility. It is preferred that these modules be downloaded dynamically at run time, without the need to recompile the entire package.

4.1.1.3 Modularity through OO Design and OO Development

To proceed with the analysis process, it is necessary to decide on the method and the model for the analysis. Object model provides the most flexibility to represent the problem, therefore, in this implementation we use the object-oriented approach to systems requirements analysis.

The software must be modular and with as much reusability as possible. Of all software engineering techniques available today, object-oriented design and development provides the greatest opportunity for modularity and reusability. To ensure a good and robust implementation of the *Distributed Query Scheduling* software package the object-oriented techniques have been used throughout the entire course of analysis, design, and development.

4.1.2 Functional System Requirements

To cover the functionality of the *Distributed Query Scheduling* software utility we use the data flow diagrams [17] to identify the main components of the system architecture. The diagram representing the top-level functionality of *DIOM Query Scheduling Utility* is shown in Figure 4-1.

Each of the entities in the diagram relates to one or more entities in another group, which is shown with the lines connecting them. The semantics of these connections is defined at the next stages of the systems analysis, where each of the entities in the diagram is analysed further and the details are further refined.

The entities of this diagram are grouped according to their relevance to one of the following categories of operations:

4.1.2.1 User Interface Processing Components (UIP)

The objects of this component are responsible for providing the graphical interface to the user. This includes the query entry form components, the controls for displaying and updating the query optimization parameters such as cost weight factors, communication and local processing costs, and data repository statistical information. Each of the major components of the query processing must have a display component for showing its result and the log information that would allow the user to follow the details of the processing at this step. For more details on the user interface components, see sections 4.2.2 and 4.5.

4.1.2.2 Input/Output Processing Components (IOP)

The objects of this component are responsible for input and output. Some of the main required IO components are the query object, objects representing the result of the query routing, objects representing the query tree, the detailed query plan, and the query execution result. Section 4.2.3 provides the detailed design specification of the components in this group.

4.1.2.3 Distributed Query Processing Components (DQP)

These are the main functional components in this application. The router, decomposer, heuristic, and cost-based query processors are the main objects in this group of components. They must be able to communicate with the query manager that organizes their operation and ensures that the necessary objects are passed to and from the user interface components, and to and from each of the query processing components. Section 4.2.4 contains the detailed specifications for each of the distributed query components.

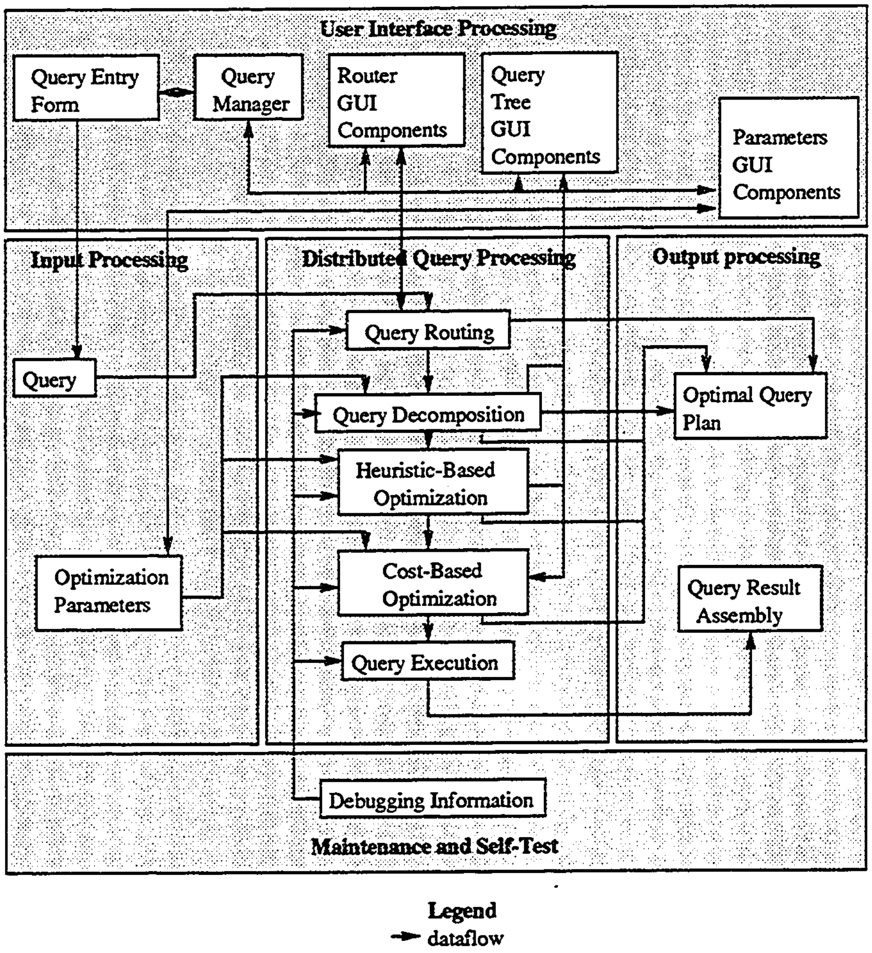


Figure 4-1: Architecture Flow Diagram of The *DIOM Query Scheduling Utility* Application.

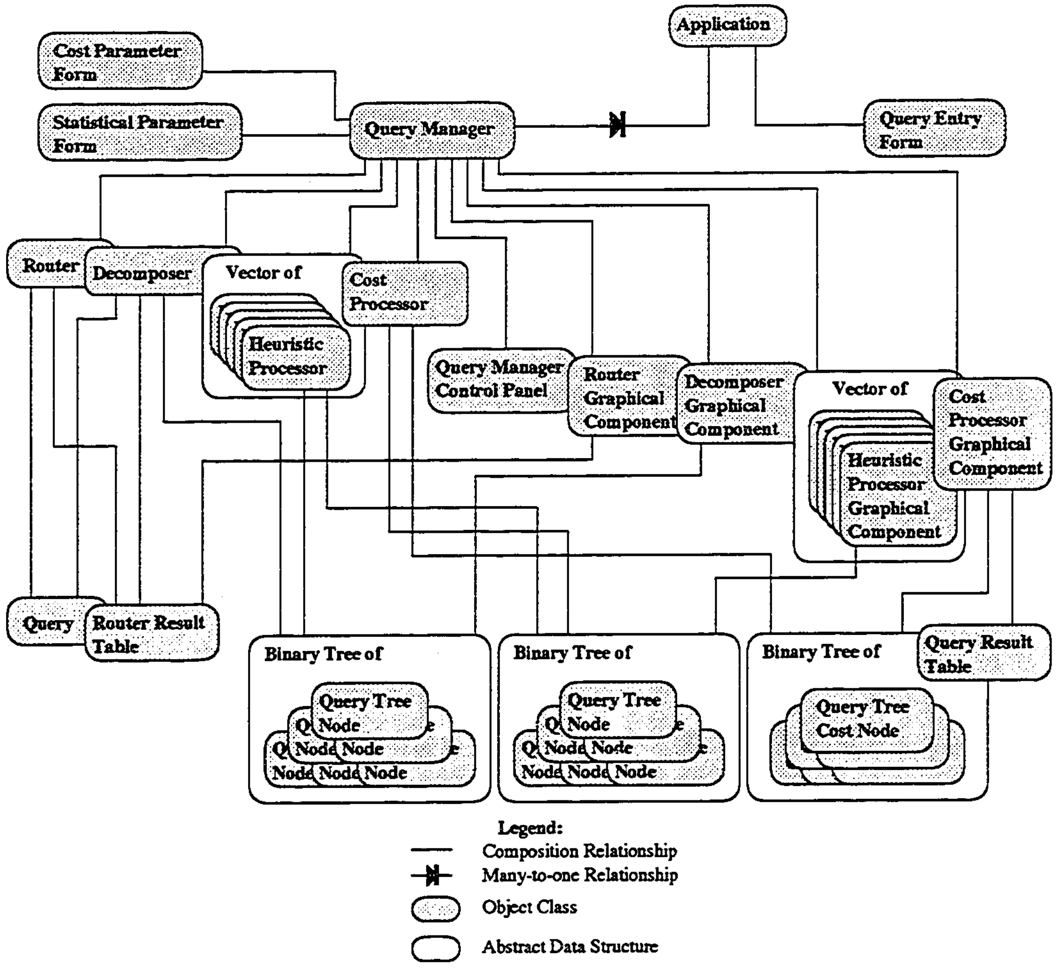


Figure 4-2: Detailed Requirements Analysis Diagram for *Query Manager* Object.

4.1.2.4 Maintenance and Testing Components

The components that allow the user to test and diagnose all other functional components. This may include query processing scenarios that test for certain features in the processing components, etc. The detailed specification of all maintenance and testing components is given in Section 4.2.5

4.2 System Architecture Design

4.2.1 Architecture Overview

Based on the requirements listed in Section 4.1, we generate a top-level object composition layout for all components of the application. This layout is shown in Figure 4-2.

The links between the objects and abstract data structures represent the composition relationships. For example, the *Router* object contains references to the *Query* and *Router Result Table* objects.

In the following sections we consider, in detail, the design issues of each of the components in user interface, input/output, and query processing component groups.

4.2.2 UIP Components

This section gives the detailed design specifications for all user interface components of the *DIOM Query Scheduling Utility*.

4.2.2.1 Query Entry Form

The *Query Entry Form* is an object in the user interface object group, and it has the following GUI components, which are also shown in Figure 4-7:

- The *choice box* component for selecting a query posed by the user with respect to an application domain. On selecting an item in this component, the query form is automatically filled out with one of the pre-installed queries;

- The TARGET selection *text field* component. This component is optional and may be left blank by the user. If the user enters the names or uniform resource locators of the information producers, then the query manager considers only them as the candidates for the relevant source selection, otherwise it will consider all registered information producers;

- The SELECT *text field* component. This component is for the entry of the select string of the IQL query. All IQL syntax rules apply;

- The FROM *text field* component. This component is for the entry of the from string of the IQL query expression. The height of this component is one line and its width must be sufficient to display a full line of text;

- The WHERE *text area* component. This component is designed for accepting the user input of the where string in the IQL query expression. It is a multi-line component since the length of the string for most user queries may exceed the length of the line. The user must be able to navigate through the entered text in this component by using the arrow keys as well as scrollbars;

- The *Result Processing Strategy* selection *checkbox*. It is an optional component. If the user clicks on this box, its state changes, which is indicated by changing color and shape of the checkbox;

- The GROUP BY *text field* component. This is an optional component. It is designed to take the user input of the corresponding IQL clause of the query expression. It contains a single line of text;

- The ORDER BY *text field* component. This is an optional component. It is designed to take the user input of the corresponding IQL clause of the query expression. It contains a single line of text;

- The *Install Query checkbox* component. This is an optional component. It reverses its state in response to the user button click, which is indicated by changing color and shape of the checkbox;

- The *radio box group* for control of the query processing mode. This group includes two *checkboxes*, *Run Query*, and *Execute Query*. The state of these checkboxes is inter-exclusive, i.e., one of them must be checked, and when one is checked then the other one will be unchecked, which is indicated by changing the color and the shape of the corresponding checkbox;

- The *Submit Query button* component. This component is designed to respond to user mouse and keyboard events. The event that happened in the area of this component is caught and the control is passed to the corresponding event handling method that creates a new query manager;

- The *Reset button* component. If an action event happens in the area of this component, the control is passed to the corresponding event handling method, which clears all the text components and sets the states of all the checkboxes to the original state;

These components are laid out on the query form panel in the same way as has been designed in the IQL interface implementation[1] of [10]. The user can navigate through these components by either mouse or keyboard events. The focus is programmatically transferred through the components whenever the user presses the *Next Component* key, which is assigned to the platform-specific default navigation key. Otherwise, if the mouse is clicked in the component area, it automatically requests focus and processes the event.

4.2.2.2 Cost and Statistical Parameter Forms

The parameters required to compute the cost of the query are separated into four groups, *Cost Weight Parameters, Local Unit Cost Parameters, Communication Unit Cost Parameters*, and *Statistical Parameters*. Their design specifications have been given in Section 4.2.3. This section gives the design specification of the GUI components that are used to provide the interface to these parameters. All three of these parameter groups reuse the same GUI component, which has the following properties:

- it contains the editable text fields for each parameter, which are laid out on a panel;
- the first row of text fields is reserved for display of the name of the parameter and is not editable;
- the first column of text fields is reserved for display of the name of the data repositories and is not editable;
- it contains a user control that allows to update the parameters to the current values of the displayed parameters;
- the component contains a reference to the data structure of the parameters, a two-dimensional array of strings, the data from it being read at the time when an event that triggers the display of this component occurs.

4.2.2.3 Query Manager Control Panel

Query Manager Control Panel is a collection of GUI components that allow the user to control and observe the query scheduling process. These components are organized as follows:

button panel contains the main controls for navigating through the steps of query processing:

- Cancel button is used for cancellation of the current query processing session and invoking the *Query Manager* event handler method *handle cancel button* that does that;
- Start button is designed to set the current state of query manager to the initial state, trigger the display of the results of this step, and generate a message informing the user of the action that has been just taken. On completion of these actions, this and Previous buttons become disabled, Next and Finish buttons become enabled;
- Previous button brings the user one step back and triggers the display of the results of the previous step, also sending a message about the performed action. Next and Finish buttons become enabled, and if the new state is the first state, this and Start buttons become disabled;

[1] The difference between IQL and Relational SQL is that the join conditions are not required for IQL expressions at the time when the query is posed. They can be derived by the IQL parser. This topic is beyond the focus of this thesis. See [12].

- Next button event causes the query manager to advance the current state by one, enable Previous and Start buttons, and if the new state becomes the last state, disable this and Finish buttons;
- Finish button event triggers the execution of all remaining steps of query processing and displaying the results. This and Next buttons become disabled, Start and Previous buttons become enabled.

display panel contains the components that display the current status and progress of the query processing. It consists of two graphical components:

- status *text field*, which provides a method of setting its contents to the programmatically defined text;
- progress graphical component, which shows an animated sequence of images and provides methods for starting and stopping the animation programmatically.

4.2.2.4 Router Graphical Component

The result of query routing step is best represented in a table form, where each row of the table corresponds to a separate information source (see Figure 4-8 on page 72). The detailed design specification of the router result component is given in Section 4.2.3. For the purpose of display this component is treated as a two-dimensional array of variable-length strings. The first row of the array consists of the column headings. Each of the rows is highlighted depending on whether the corresponding source has been selected by the router automatically, or by the user, manually.

This graphical component contains the following elements:

- the main display area, where the visible portion of the router table is displayed. The display may be shifted horizontally or vertically according to the offset controlled by the scrollbars;
- the scrollbars, which graphically display the current horizontal and vertical offset of the main display area and react to the user events targeted at them.

This component defines the following methods:

- *measure* – to properly display multi-line, column-formatted text, the component must know the height of each line and the width of each column. This measurement requires that the font parameters are known, therefore it must be called before the display but after the font information is available.
- *handle mouse event* – if the coordinates of the mouse event are within the main display area of this component, this method computes the row number based on these coordinates and the current offset and highlights the corresponding row of the table. This row is marked as user-selected so that future screen updates redisplay it properly.
- *resize* method, which is called whenever the size of this component changes, updates the main display area, the shape of the scrollbars, and the offset values.

4.2.2.5 Decomposer Graphical Component

The result of the query decomposition step is a collection of query operators organized in a binary tree data structure. An example of a query tree is shown in Figure 3-2 on page 16. The detailed design specification of the elements in this data structure is given in Section 4.2.3, and here we list the design issues related with the proper display of this structure. We assume that each node in the binary tree contains references to the subnodes, and if there are no subnodes, the corresponding reference is NIL.

The decomposer graphical component consists of two main elements:

- the main display area, where the visible portion of the tree is displayed. The display may be shifted horizontally or vertically according to the offset controlled by the scrollbars;

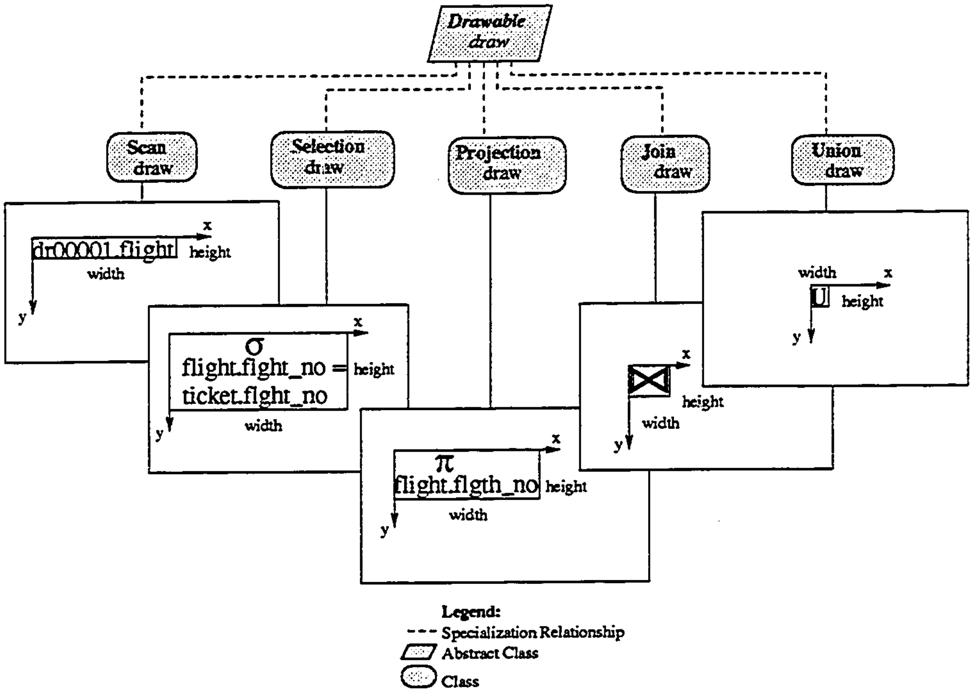


Figure 4-3: *Drawable* abstract class and *draw* method overriding

- the scrollbars, which graphically display the current horizontal and vertical offset of the main display area and react to the user events.

The display of a binary tree, which is not a linear structure, is more complex than the display of a table. We take into account the following design issues:

1. this component must know the total width and height of the entire tree, which is computed recursively starting from the root of the tree and ending at its leaf nodes. A query tree node provides a method of computing these parameters;
2. the tree is drawn in a recursive manner, starting from the root, and ending at the leaf nodes. If the width of the node being drawn is greater than the total width of this node's subtrees, then the subtrees are shifted to be centered under this node, otherwise, this node is shifted to be centered above the subtrees;
3. links are drawn as lines connecting the center of the bottom edge of this node to the center of the top of the underlying subtree(s);
4. to actually draw the node, its drawing method is called, with the required coordinates; Each of the defined node types is a subclass of an abstract *Drawable* class, it overrides the draw method. Figure 4-3.

4.2.2.6 Heuristic Processor Graphical Components

Like query decomposer result, the result of each heuristic-based transformation is an instance of a binary query tree, therefore here we reuse the design specifications of the *Decomposer Graphical Component*.

4.2.2.7 Cost Processor Graphical Component

The cost-based query tree processor results in a data structure that may be viewed as an extension of the decomposition and heuristic-based query trees (see Figure 4-11). Each node in a cost-based query tree has additional information. The detailed design specification of the cost node is given in Section 4.2.3. In this section we deal with the user interface-related issues of the cost node design. We assume that cost node provides the methods for retrieving the extended elements of the node.

This component is an extension of the *Decomposer Graphical Component* and has all of the design features of it. In addition, this component has the following design characteristics:

- each node, is drawn with the color-coded background that corresponds to the site to which this node has been assigned. Nodes that have been assigned to the same site use the same background color. The color value is computed dynamically based on the total number of distinct sites in the cost query tree.

- *handle mouse event* method is defined for this component. Whenever the user generates the mouse event in the display area of this component, this method tries to find the node corresponding to the mouse coordinates. If the node is found, a *dialog window* is displayed. This dialog window contains the detailed cost-related information about this tree node. The background color of the dialog repeats the background color of the node on the main display area.

4.2.2.8 Additional Graphical Components

In addition, the following user interface components have been designed to allow expert users to trace the heuristics used in the query tree transformations and the cost estimation used in the cost-based optimization (see Figure 4-12 for an example):

Log View Dialog Window Each of the query processors produces the log information that contains the query processing details and is intended for user viewing. This component is designed to provide a flexible display facility that allows the user to view the log with variable degree of detail. This component is a subclass of a dialog window, it contains the following GUI elements:

- the main display *text area*, where the corresponding log text is displayed. Since the log is a multi-line variable-width text, the text area provides the horizontal and vertical scrollbars for controlling the position of the visible portion of the text.
- +, or *increase log detail level button*, and −, or *decrease log detail level button*. The action events in these buttons are handled in such a way that the log detail level is changed and only the information that has been logged with the level up to the current detail level is displayed in the main text area. If the maximum or minimum level is reached, the current value of the log detail overwraps to minimum or maximum respectively.
- CLOSE *button* is designed to allow the user to dispose of this dialog window and free the system resources.

Query Result Graphical Component allows the user to see the result of the query execution. This component reuses the *Router Graphical Component* design specification since the query result has the same structure as the query router result, a two-dimensional array of strings that are formatted into columns and rows.

Query Manager Main Display Panel provides the GUI container for each of the main graphical components of the query manager. The main display panel has the layout that allows the user to view the contained components one at a time in the same display area. It provides a method for putting the specified component on top of others and making it visible. The *Query Manager Control Panel* button handling methods call this method to make the graphical component of the corresponding processor visible.

4.2.3 IOP Components

4.2.3.1 Query

Query is the object that is instantiated by the *Query Entry Form* (see Figure 4-7) and passed to the *Query Manager* object. It contains all the information about the user query, which includes:

- SELECT, FROM, WHERE IQL clauses, and
- the entire IQL query expression that is formed from the above three clauses;

Query class defines the access methods for each of its components.

4.2.3.2 Cost Parameters

Cost Parameters consist of the following three groups of parameters:

- Cost Weight Parameters – see Equation 3.1 on page 14 for explanation of this group of parameters. The parameters of this group are user-defined because they represent the profile of the user priorities in evaluating the total query cost.
- *Local Unit Cost Parameters* include the unit costs for each of the query-related operations. This application considers local unit costs of *scan*, *join*, and *union.*
- *Communication Unit Cost Parameters* – see Equations 3.10 – 3.15 on pages 36 – 46 for the explanation of these parameters – are the parameters provided by the information producers and are collected by DIOM. This parameter is provided for each combination of the information sources, otherwise, a default unit cost is used.

4.2.3.3 Statistical Parameters

Statistical Parameters contain the statistical information about the class, relation, or attribute objects that are stored in the information sources registered with DIOM. Only the statistics information for the information sources selected by query router are included in the *Statistical Parameter* component of the current query (see Figure 4-13. Section 3.4.1 provides a more detailed information about the nature of these parameters. For the purposes of this application we supply the following parameters for each of the objects in the source database:

- *class name* – a string containing the name of the class, relation, or file name;
- *cardinality* – number of objects of this class in the repository;
- *width* – size of one object, row in the relation, or scannable unit in the file;
- *size* – the total size of the extent;
- *attributes*, for each of which the following data is collected:
  - *attribute name* – a string containing the name of the attribute, the column name, or the field name;
  - *cardinality* – if the objects of this class are indexed by this attribute, it contains the number of distinct entries in the index;
  - *width* – the size of one element of this attribute;
  - *size* – the total size of the index;
  - *type* – this is an optional element, which is used in determining whether the following two elements are valid; it distinguishes between enumerated and non-enumerated datatypes;
  - *min* and *max* values – valid only for the enumerated datatypes.

4.2.3.4 Router Result

This component is instantiated by the *Router* query processing component, modified by the *Router Graphical Component*, and is passed to the *Decomposer* query processing component. It consists of the following elements:

- the table which stores the information about the data repositories and is displayed in *Router Graphical Component.* Each column of this table contains the specific type of information about the source, e.g., source id, its URL, full name, owner, keywords, etc.;
- the one-dimensional array of boolean values that mark the data repositories which have been selected by the *Router*;
- the one-dimensional array of boolean values that mark the data repositories which have been selected by the user through the *Router Graphical Component.*

We call the one-dimensional boolean array generated by the *Router* DQP component *Router Vector*

4.2.3.5 Query Tree Node

This is the main operating component of the *Decomposer*, and the *Heuristic Processors* query processing components. It consists of the following elements:

- *Query operator* object – one of the subclasses of *Drawable* abstract class that implements the *draw* method, see Figure 4-3.
- Reference links to *parent*, *left*, and *right Query Tree Nodes.* If this node is the root of the query tree, then its PARENT link is NIL, if it contains a unary query operator, then its *right* links is NIL, and if this is a leaf node, both its *left* and *right* links are NIL.

It defines the following methods:

- *clone* method creates a copy of this object and returns it. As we will see in the design specifications of *Heuristic Processors*, Section 4.2.4, quite often we will need to make a copy of the node to properly process the tree;
- *draw* method simply calls the *draw* method of this node's *query operator* object;
- *width* method simply calls the *width* method of this node's *query operator* object;
- *totalWidth* method is needed to compute the total width of the subtree whose root is this node. If this is the leaf node, then its width plus the width of the margin is computed, otherwise, the maximum of this node's width and the sum of the *totalWidths* of its subnodes;
- *totalHeight* method computes the total height of the subtree whose root is this node. This and the previous methods are used in the UIP components that display the query tree, see Section 4.2.2.

4.2.3.6 Query Tree Cost Node

This component is an extension of the *Query Tree Node* IOP component. It adds the following elements to its design specification:

- cost-related elements, *total cost*, *local processing cost*, *communication cost*, and *response time cost*; The computation of these parameters is covered in the design specifications of the *Cost Query Processor* in Section 4.2.4;
- site information, where the name of the site to which the query operator of this cost node is assigned;

- statistical information of the current subquery result. This information is initialized with the statistical parameter values that are collected by DIOM from the information repositories. For non-leaf nodes of the cost query tree this information is computed by the *Cost Query Processor.* This information is a set of attributes that are defined in the intermediate query objects.

The functionality of this component is extended by the following methods:

- *clone* – this method overrides the *clone* method of the *Query Tree Node* component, it ensures deep-level cloning of all the attributes of the current cost node;
- *get new extent cardinality* method is responsible for computing the new cardinality of the objects that are represented by this tree node. Given the selection operator, this method computes the statistical selectivity factor based on the attribute information of the attribute on which the selection is done. The detailed process of statistical parameter estimation is specified in Section 3.4 of Chapter 3;
- *update attribute cardinalities* method updates the statistical information of all the attributes of this query tree node. Given are the old and the new extent cardinalities. The computation is done according to Equation 3.5 on page 27;
- *get attribute superset* method takes two attribute vectors and makes a superset consisting of all attribute elements of the first attribute vector and the elements of the second attribute vector not found in the first attribute vector. This type of processing is done by the union query operator and is covered in detail in Section 4.2.4;
- *append attribute set* method appends the given attribute vector to the end of this node's attribute vector. This type of processing is required by the product query operator, which serves as the basis for the join query operator;
- *get log information* method instantiates a log object that may be passed to the *Log View Dialog Window* UIP component. This method is called by the *Cost Processor Graphical Component* when the user generates a mouse event that is targeted on this cost node;

4.2.3.7 Additional IOP Components

Log component is used throughout the application by most of the main UIP components to generate a multi-level detail log information string. This component is then passed to the corresponding *Log View Dialog Window* component, which displays it. The format of this component is defined by the following rules:

- the level of detail is defined by the first character of the log line, this character is not displayed in the *Log View Dialog Window*;
- every line of the log information starts with a character that is converted to the number, and interpreted as the level of detail, if the first character may not be converted into the number format, then the string is discarded by the *Log View Dialog Window*;
- the rest of the string of the logged information is indented according to the level of detail, e.g., the indentation of a line whose detail level is 2 equals to two indentation units. This rule is optional and is designed only for user convenience.

Selection Object components are derived from the WHERE clause of the IQL expression. Each of the tokens in the clause represent one *selection object. Router* DQP component instantiates them and stores them in the *Vector* data structure at the time of query parsing. Intensionally, they are comprised of three elements:

- *left argument* is an instance of *projection object,*

- *operator* is a string representation of the operator taken directly from the WHERE expression, and
- *right argument* is an instance of *projection object*;

Projection Object components are derived from the SELECT clause of the IQL expression. Each of the tokens in the clause represent one *projection object*. At the time of parsing the query Router instantiates the objects of this class and stores them in the *Vector*. This component consists of three elements:

- *schema* is a string representing the name of the schema of the database object, relation, or file,
- *object name* is a string containing the name of the object, relation, or file,
- *attribute name* is a string containing the name of the attribute, column, or the field;

Join Object components are derived from the FROM clause of the IQL expression. Each of the tokens in the clause represent one *join object*. The *Router* component instantiates the objects of this class and stores them in the *Vector* at the time of query parsing; This component consists of two elements:

- *schema* is a string representing the name of the schema of the database object, relation, or file, and
- *object name* is a string containing the name of the object, relation, or file

Query Result Table component reuses the specifications of the table component of the *Router Result* IOP component.

4.2.4 DQP Components

This section covers the main components of the distributed query processing done in this application.

4.2.4.1 Query Manager

The *Query Manager* is the main component that coordinates the work of both DQP and UIP components. A new query manager object is instantiated in the *Query Entry Form* whenever the user submits a query. For performance tuning purpose we allow a query to be submitted several times, and each time, be optimized using different parameters. For each query form we preset the maximum number of query manager objects that the form can instantiate. This restriction can be used to set the upper limit on the possible CPU and memory requirements this application can impose on the system.

Since *Query Manager* is composed of both DQP and UIP components (see Figure 4-2), its functionality combines the query processing-related functionality with the GUI-related functionality. Namely, the *Query Manager* class both specializes a *window* class and implements the methods of the *runnable* interface. Thus DQP components of the *Query Manager* are independent of the UIP components (recall the modularity requirement in Section 4.1.1). In detail, *Query Manager* object is composed of the following elements:

- reference to the *Query Entry Form* that launched this query manager is needed to pass the status messages to that component;
- reference to the *Query* IOP component, which is passed from the form at the instantiation;
- *ID* number of this query manager is used for identification in the messages to the query form and in the title of this window;
- *Router*, a *Vector* of *Heuristic Processors*, and the *Cost Processor* DQP components, whose design specification is given later in this section;

- *Query Manager Control Panel* UIP component and all its elements, whose design specifications are covered in Section 4.2.2;
- *Main Display Panel*, all UIP components covered in Section 4.2.2 that are used to display the result of each of the query processing steps, and their containing panels, parameter forms, and buttons used to activate them;
- *Vector* of *Log View Dialog Windows* that is used to store the references to all currently displayed log windows. The query manager needs to keep track of these windows so that if the user cancels the current session, they may be disposed of and the system resources freed;
- *current state* of the query processor is a numeric representation of the current state of this component, e.g., initial state is zero, the state on completion of query routing is one, etc.
- *finish state* of the query processor is a numeric representation of the desired state on achieving which the query manager suspends its execution, e.g., in the tracing mode the finish state is always greater than the current state by one, while in the run mode the finish state equals to the maximum number of query processing steps. This parameter is initialized based on the value of the *trace/run* radiobox group.

The last two elements are designed to enable and control the multiple sessions of the *Query Manager*.

Methods of the *Query Manager* class include

- *router process*, *decomposer process*, *heuristic optimizer process*, and *cost optimizer process* methods facilitate communication with the corresponding DQP components. The semantics of these components will be specified in detail below;
- *run* is the main processing method that is called whenever the user presses either Next or Finish buttons in the *Query Manager Control Panel* UIP component (see Figure 4-8). Based on the values of the *current state* and *finish state*, this method incrementally checks all the states of the query processing and calls the processing methods that correspond to the states between the current and the finish state; Since *Query Manager* implements the runnable interface, this method must be defined and is called whenever the query manager object is *started*;
- *init* is the method that is called at the instantiation time of the query manager object. It instantiates and lays out all the GUI components of the *Query Manager*;
- *handle event* method is called whenever an event occurs within the *Query Manager* window. Depending on the GUI component the event is targeted to, its handling is delegated to that component's *handle event* method.

The complete code of this component is listed in Appendix A.

4.2.4.2 Router Object

Router DQP component is responsible for query routing step of the distributed query scheduling, it is instantiated by the *Query Manager* and contains the following elements:

- *query* component is passed to the *Router* by the *Query Manager* at the time of instantiation;
- *Vectors* of *selection objects*, *projection objects*, and *join objects*, which represent the result of query parsing;
- *Router Result* component, whose specification is given in Section 4.2.3.

This component implements the following methods:

- *get new query vectors* method parses the query of the router object and makes three new vectors containing *Selection Objects*, *Projection Objects*, and *Join Objects* components. Query string parsing is done by separating the string into tokens based on the syntactically valid IQL separators;

- *get new router table* method is responsible for contacting the DIOM server and obtaining the list of information producers registered with DIOM dynamically, i.e., each time when the query is evaluated. The format of the entries in the list is coordinated between the server and this method at run time of the query, and the list is parsed and stored in the table, which is an element of the *Router Result* component specified in Section 4.2.3;
- *get new router auto-select vector* method scans the router table, tries to match the content description of each entry with one of the elements in the vector of *Join Objects*, and produces a boolean array, in which each element indicates whether the corresponding information producer has been selected by the router as a relevant information source for the current query;

The complete code of this component is listed in Appendix A.

4.2.4.3 Query Tree Processor Object

This is an abstract component whose main function is to process a binary tree of *Query Tree Node*. *Query Tree Processor* consists of the following elements:

- *query tree* is the binary tree of *Query Tree Node* IOP components;
- *name* is the name of the current tree processor. It is used in the log information and for user messages;
- *log* is the *Log* IOP component.

Query Tree Processor implements the following methods:

- *set query tree* simply sets the reference to the given query tree;
- *get query tree* simply returns the reference to this object's query tree component;
- *set new log* resets the log information;
- *get log* simply returns the *Log* component of this object;
- *copy* method recursively traverses the given tree, and effectively copies each node of the tree to a new instance.

All subclasses of *Query Tree Processor*, e.g., *Decomposer*, *Move Selections Heuristic Processor*, *Move Joins Heuristic Processor*, and *Cost Processor*, inherit all its components and methods. In addition, all non-abstract subclasses of *Query Tree Processor* must implement the following methods:

- *get new query tree* contains the intrinsics of the current component's query processing. Given the query tree, a tree processor object custom-processes it and generates a new query tree;

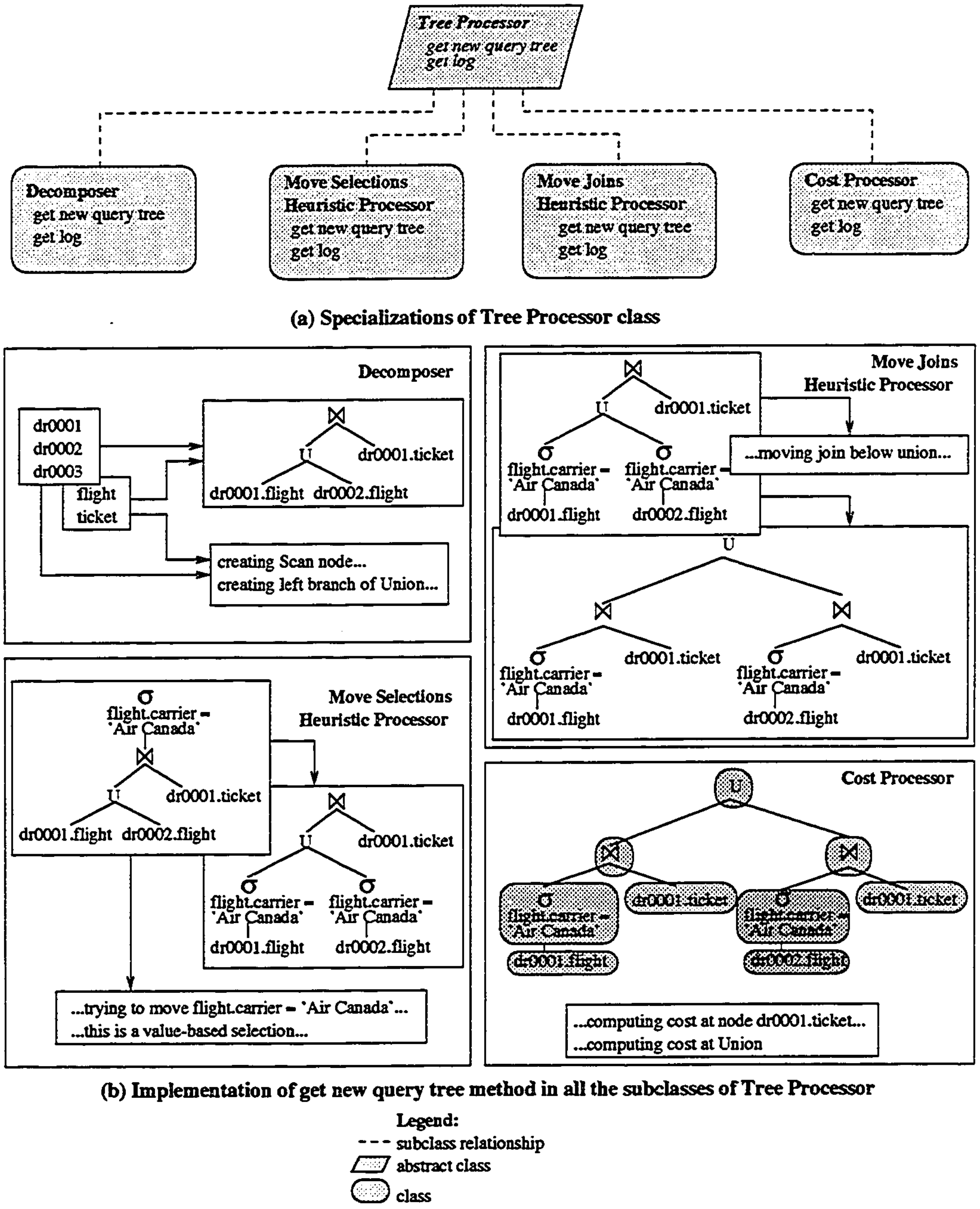
Figure 4-4 illustrates how the subclasses of *Query Tree Processor* implement the *get new query tree* method. The detailed design specifications of this method for each of its subclasses are given in the subsequent sections. The complete code of this component is listed in Appendix A.

4.2.4.4 Decomposer Object

Decomposer is responsible for the second step of distributed query processing session in *DIOM Query Scheduling Utility*. It has the following components:

- *Router*, a reference link to this component is necessary to access the *Router Result* objects;
- *Vector* of *Selection*, *Projection*, and *Join*, these vectors are needed to store the intermediate results of decomposition;

The design of its *get new query tree* method has the following details:



(a) Specializations of Tree Processor class

(b) Implementation of get new query tree method in all the subclasses of Tree Processor

Figure 4-4: Tree Processor Classification.

1. create new vectors of *Selection*, *Projection*, and *Join* objects that are based both on the same vectors of the *Router* component and on the *Router Result*. These objects are accessed through the reference link to the router object; Only those elements from the router vectors are selected that have supporting entries in the router table. The following rules are observed:
   - each of the *Join Objects* in the *Router* vector (recall Section 4.2.3 is included if it matches at least one entry in the router table. For each of the matching table entry the source of this entry is added to the vector of available sources for that *Join Object*;
   - each of the *Selection Objects* in the *Router* vector is included if it is a value-based selection and if the selection argument object has been included into the *Decomposer's* vector of *Join Objects*, otherwise, if it is a join-based selection and both of its arguments are included;
   - each of the *Projection Objects* in the *Router* vector is included if the object component of the projection has been included into the *Decomposer's* vector of *Join Objects*.
2. create the decomposition tree according to the following rules:
   - all source-based scans of every *Join Object* form a subtree with *Union* as the internal node and *Scan* as leaf nodes;
   - all of these subtrees are *joined* together to form the query tree;
   - all of the *Selection* and *Projection* objects form the *unary node group* which is put on top of this tree. The result is assigned to the *Query Tree* component of the *Decomposer*.

This constitutes the last preliminary step of the query scheduling. The main processing that generates the query execution plan is done by the heuristic-based and cost-based query processing.

The current version of *DIOM Query Scheduling Utility* supports two of the heuristics covered in Chapter 3, moving selections down and moving joins below unions. Their design specifications are presented below.

The complete code of this component is listed in Appendix A.

4.2.4.5 Move Selections Heuristic Processor

This component's *get new query tree* method has the following design specifications:

1. create a copy of the given query tree and locate all nodes that contain *Selection Objects*;
2. for each of the found nodes, traverse its subtree recursively performing the following operations until a leaf node is reached:
   - if the child node contains a *Union* object, create a copy of this *Selection Object*, put the original into the left subbranch of the *Union*, and the copy into the right subbranch of the *Union*, recursively traverse each of the subbranches until a leaf node is reached;
   - if the child node contains another *Selection* object or a *Projection* object, put this node below it and recursively traverse its subbranch until a leaf node is reached;
   - if the child node contains a *Join* object, and if this is a value-based selection, move it into the subbranch that contains a *Scan* of the same objects as this node selects, if neither subbranch terminates with these scans, leave this *Selection* object and stop the tree traversal, otherwise, if this is a join-based selection and neither of the join subbranches is terminated with a scan of any of the selection arguments, perform the same operations as for the *Union* node, else stop the tree traversal.
3. assign the transformed tree to the *Query Tree* component.

The complete code of this component is listed in Appendix A.

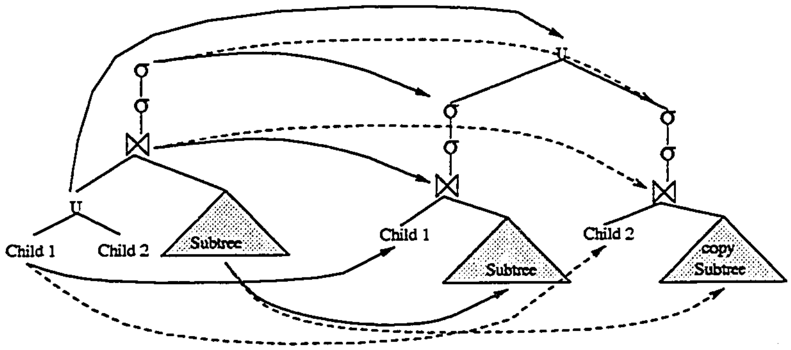


Figure 4-5: Moving the Join Group below the Union, an example.

4.2.4.6 Move Joins Heuristic Processor

The purpose of this heuristic is to move the joins below the unions in the query tree. Chapter 3 provides the theoretical coverage of this heuristic-based processing rule. This component has the following design specifications of its *get new query tree* method:

1. create a copy of the given query tree and locate all nodes that contain *Join Objects.* If there are such nodes, analyse the query tree, compute the total number of scan nodes and the total number of different sources on which these scans are performed. If more than half of all the scan operations are done at the same sources, then do the processing, otherwise terminate the processing without modifications to the tree;

2. for each of the found nodes, traverse its subtree recursively performing the following operations until no more union nodes are found below the node containing a *Join Object*, if the left child of this node is not a union, try the right child, otherwise, there are no more *Union*-containing nodes below. If it is a union, do the following recursively:

    - clone the node containing this *Join Object*, and all the *Selection* nodes above it, thus creating a new copy of the join group;
    - locate the other child node of this *Join Object* and make a copy of the entire subtree whose root is that node;
    - attach the left child of the union as the left child of the original *Join Object*, and the right child of the union as the left child of the clone *Join Object*;
    - attach the copy of the join's right subtree as the right child of the clone *Join Object*;
    - attach the top of the join group of *Join Object* as the left, and its clone as the right child of the *Union* node;
    - recursively process these newly formed subtrees;
    - return the *Union* node as the new root of this subtree.

    This operation is graphically illustrated in Figure 4-5.

3. assign the transformed tree to the *Query Tree* component.

The complete code of this component is listed in Appendix A.

4.2.4.7 Cost Processor Object

This component is responsible for the final step in distributed query scheduling. It is also the most complex component since it requires two phases of cost estimation: (1) computation of statistical parameters for each of the nodes of the query tree and (2) computation of the cost of evaluating the entire query trees that are considered as potential candidates for the selection of an "optimal" query schedule. The theoretical model for this step is covered in sections 3.1 and 3.4.

Cost Processor component operates on *Query Tree Cost Node*, an extension of *Query Tree Node* component whose design specifications are covered in Section 4.2.3. The main processing method of *Cost Processor*, *get new query tree* (see bottom right box of Figure 4-4 on page 63), consists of the following steps:

1. create a copy of the given query tree and locate all nodes that contain *Scan*, and *Projection* objects, including *Projections* that are part of *Selection* objects. Get the *Statistical Parameters* of each of all sources from which scans are performed in this query. Fill in the missing statistical information with the default values if any parameter is missing;

2. fill in the statistical information at each of the *Query Tree Cost Nodes* of the binary tree, the following rules are observed in computing the statistics:

   - if the current node contains a *Scan*, fill in the statistics directly from the *Statistical Parameters* component obtained from the server;
   - if the current node contains a *Selection*, compute the new statistics by using methods *get new extent cardinality* and *update attribute cardinalities* of *Statistical Parameter* IOP component;
   - if the current node contains a *Projection*, remove the non-projected attributes and update the statistics accordingly;
   - if the current node contains a *Union*, update the statistics according to method *get attribute superset* of *Statistical Parameter* IOP component, which also applies formula of Equation 3.3 on page 26;
   - if the current node contains a *Join*, update the statistics according to methods *append attribute set* of *Statistical Parameter* IOP component, and applying formula of Equation 3.4 on page 26.

3. by default, the result of the query is required at the *client* site. We obtain the unit cost information from *Cost Parameters* IOP Component. We compute the cost information and assign a site to each of the nodes in the cost query tree recursively, starting from the root of the tree, whose preferred result site is *client*:

   - if the current node is a unary node non-leaf node, i.e., it contains either *Selection* or *Projection*, compute the cost information and assign the site of this node's child, tell the child that the preferred site of its result is the same as the preferred site of this node's result. Then assign this node to the same site as the child is assigned to, compute the cost components and the total cost. Unary query operator is always performed at the same site as its child;
   - if the current node is a leaf node, i.e., node containing *Scan* object, just compute the cost components and the total cost. Scans are always performed at their sites;
   - if the current node is a binary node, i.e., it contains either *Join* or *Union*, first compute the costs and assign the sites of each of the subtrees, then compute the total cost for each of the possible three placement options, as described in Section 3.5.3, Equation 3.14 on page 42, and choose the minimum option. If the placement resulting from this option differs from the current preferred result placement, recompute the costs for each of the subnodes for the new preferred site. In this case, the binary operator will be performed at the same site as one of its children's, therefore the cost changes and the preferred site for all of this node's children now becomes the site to which this node has been assigned.

4. assign the transformed tree to the *Query Tree* component. This result constitutes the detailed query schedule, where each site has been assigned a defined set of operations.

The complete code of this component is listed in Appendix A.

4.2.5 Maintenance and Diagnostic Components Specification

To develop bug-free, robust software, there needs to be a facility that allows to test and diagnose each of its design components. Each of the components of *DIOM Query Scheduling Utility* has been equipped with uniform diagnostic and bug detection methods.

One of the approaches used in tackling this problem is to provide the component methods that allow easy tracing of the component's elements. For instance, we have designed a method to convert the contents of a *Query Tree Node* component into a printable form to allow users to trace the work performed by one of the *Tree Processor* components. Another approach used in the design of our testing package is to provide a set of self-tests for each of the components. These tests check for exception conditions that are not parts of the main processing the components have been designed for but which could undermine the correctness of the processing. For instance, a test that checks for the presence of both children in a binary query tree node ensures that the tree is correctly generated.

Conditional compilation allows the application developer to remove the maintenance and diagnostic components from the production version of the software package, which ensures that performance is not traded in for correctness of the design.

4.3 Code Implementation Design

Based on the analysis of the system requirements, and the architecture design of the Distributed Query Scheduling Utility, we chose Java programming language as the coding tool for implementing our system. In this section we first give a brief overview of the Java language [6], and then discuss the coding issues that we encountered during the coding phase with Java.

4.3.1 Java Programming Language: a Brief Overview

Originated as a programming language for consumer electronics in 1990, Java's main goals were to be small, reliable, and architecture independent. With the advent of the World Wide Web on the Internet, Java became ideal for programming on the Internet as its original design goals suited perfectly for it. At present most of the major software companies have shown interest in Java.

Java syntax looks very similar to C and C++ syntax. At the same time, it has fewer syntactical constructs, which makes the language easier to learn and simpler and more reliable to code.

Java is an object-oriented programming language. This fact imposes the analysis and design constraints on the application developers. To use the language most effectively, a developer must adopt the object-oriented approach at both analysis and design phases of software engineering.

Java is a distributed language, it supports applications on networks at various levels of network connectivity.

To enforce platform independence of Java, its compiler generates a platform-neutral program code, also known as *byte* code. This code is executed by the language interpreter, commonly referred to as *Java Virtual Machine.* Thus Java is an interpreted programming language.

Java is designed for writing highly reliable or robust programs due to the following three features:

- it is a strongly typed language, it enforces explicit type casts and method declarations;
- it has a highly reliable memory model, e.g., the programmer does not need to worry about memory deallocation and memory leaks as well as checking array boundaries;
- it has a convenient exception handling mechanism that uses the `try/catch/finally` construct and allows the programmer to simplify the task of error handling.

A compiled Java program can run on any platform that implements the *Java Virtual Machine.* This feature originated from the initial language design and was amplified with the development of the Internet. Also, this feature has found support in the fact that most software developers want to create software that runs on any platform. Java provides a package called *Abstract Windowing Toolkit* (referred to as `java.awt`), which allows the programmer to develop application user interfaces that look native on a platform on which they run.

At the same time Java is a portable language, i.e., it explicitly defines each of the primitive datatypes and thus provides full independence of the particular implementation.

There are a number of other attributes of the Java programming language, such as security, high-performance, multi-threaded functionality, dynamic extensibility, and others. [6] provides a good reference that explains each of these language attributes in detail.

4.3.2 Distributed Query Scheduling Utility Software Package in Java

Since Java is an object-oriented language, a software package written in this language is a collection of class definitions, each of which is a specialization of the system built-in classes. Thus all program units in a Java program form a class hierarchy that is built on top of the Java's own class hierarchy. A collection of the classes that is designed to perform a certain task is usually combined into a Java package.

The Java implementation of the *Distributed Query Scheduling Utility* is such a Java package. It is named (according to the language naming conventions) `ca.ualberta.diom.query` and can be accessed using a Java-enabled browser by pointing at URL
`http://ugweb.cs.ualberta.ca/~diom/query/EQ.html`.

Figure 4-6 shows the class hierarchy of this package in the context of class hierarchies of standard Java packages. The class hierarchy as well as full software documentation in the form of *javadoc* html files is also available at URL
`http://www.cs.ualberta.ca/~diom/query/html/tree.html`. Some of the main classes in this diagram directly correspond to the design components covered in Section 4.2, e.g. **Router** class is the code implementation of the *Router* DQP component (see Section 4.2.4), while others, such as Java class **Utils** implement a collection of utility methods that are being used throughout the *DQS* package. Full source code of the entire package is available on-line at URL
`http://www.cs.ualberta.ca/~diom/query/src/`.

This software package was developed and tested on **Solaris** platform using **Sun JDK** version **1.1**. The byte-code has been tested on the following platforms:

- Windows NT v. 3.51, using Netscape Navigator v. 2.01;
- Sun OS v. 4.1.4, using Netscape Navigator v. 3.0.

4.4 Implementation Remarks

4.4.1 Generality

It is worth mentioning that although in the first version of *DQS* we only use the sample query in the Flight-Order application, the complete implementation of *DQS* package, including the **Router**, **Decomposer**, **MvSlctnDwn**, **MvJoinDwn**, and **Cost** classes, is generic. Namely, it can be applied to any query in any domain application. As long as the relevant information sources are identified, and the statistical information about their content and capabilities is available, *Distributed Query Scheduling Utility* may be used to generate the distributed query execution plan.

4.4.2 Software Extension and Further Development

Object-oriented model provides the best environment for extension and further development of the software through mechanisms of class specialization and aggregation. Since *DQS* software package uses object-oriented approach in its design and coding, such extensions are made easy. For instance, to add a new heuristic to the package a developer needs to further specialize the *Tree Processor* class, likewise, adding a new type of query operation would involve further specialization of the *Drawable* class.

Although the current version of the package solves the initial task to provide evidence of viability of query processing techniques proposed in Chapter 3, there is still room for improvement and

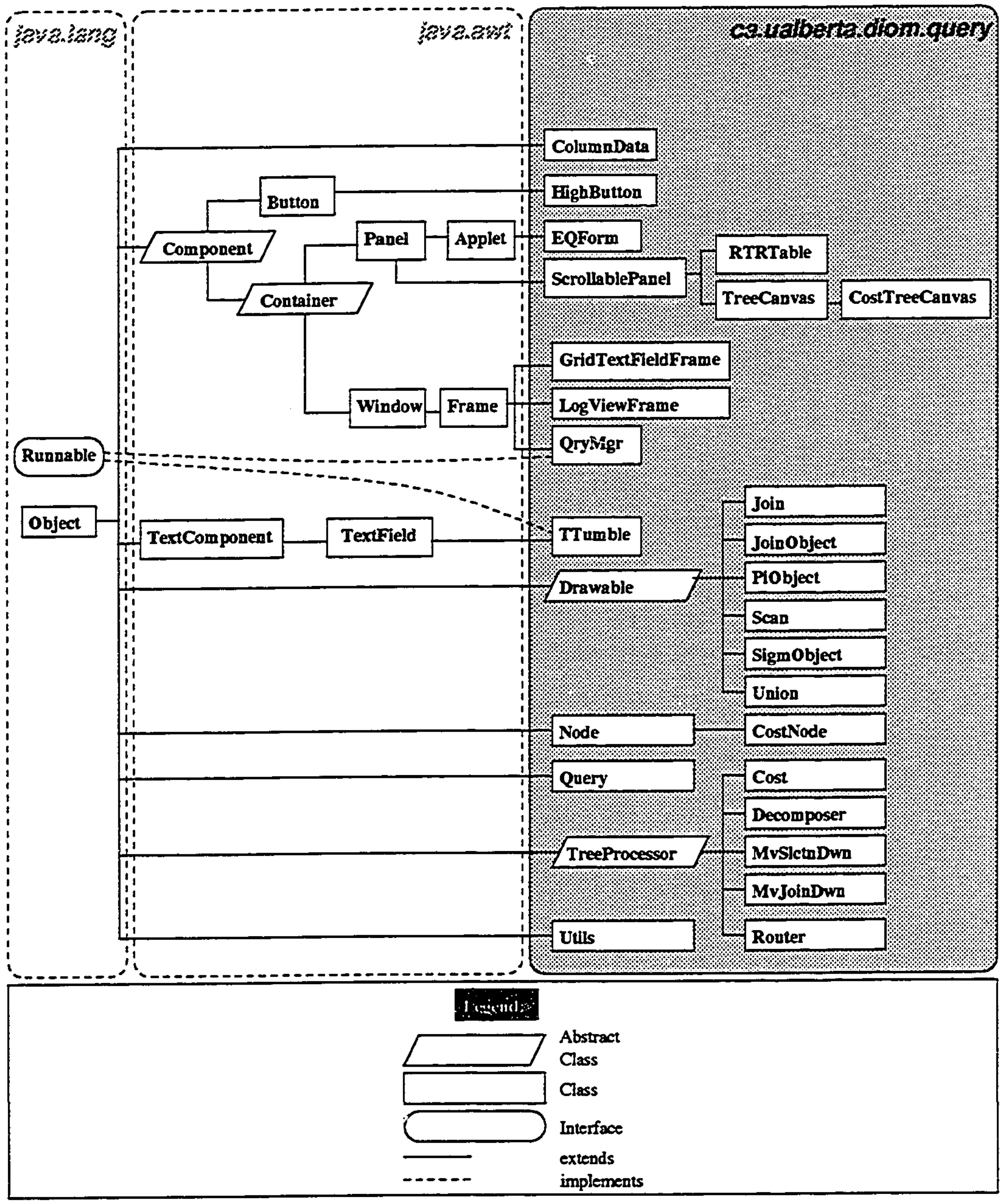


Figure 4-6: Class Hierarchy of Distributed Query Scheduling Utility software: Implementation in Java.

extension. Chapter 5 contains a brief outline of the possible further developments of this software package.

4.5 User Interface

In this section we present, in detail, the user interface of the *DIOM Query Scheduling Utility.* The material of this section is targeted at the users of this software package and may also serve as the user's manual.

4.5.1 Query Entry Form Screen

The front-end of *DIOM Query Scheduling Utility,* the query entry form, is shown in Figure 4-7. It contains the following user controls:

Application Domain choice box is located at the top of the page. It allows the user to enter the pre-installed queries into the form without having to type. On selecting any of the choices in that box the corresponding text fields are filled with the query strings;

TARGET text field is the text field where user can enter the name, the id, or the URL of the information source(s), to which the query is addressed. This field is optional;

SELECT , FROM, and WHERE text components are where the user types the IQL query. The WHERE text is optional;

Select Result Processing Strategy checkbox is for use of the query result assembly, and currently is not implemented;

GROUP BY and ORDER BY text fields allow the user to enter the corresponding IQL clauses;

Install this query checkbox is used if the user wants to install the currently posed query;

Run Query and Trace Query radiobox group allows the user to choose the query execution mode. If the Trace Query is checked, the execution will stop after the first step, routing, is completed, and its result is displayed, otherwise, all steps are done consecutively and the result of the last step is displayed. This checkbox only sets the appropriate flag for query manager but it does not actually start query processing, which is done by Submit button;

Reset button clears all text components of this form and resets all checkboxes to their default state.

Submit Query button submits the currently displayed query to DIOM and launches a new query scheduling session. A new query manager window titled Query#, where # is the ID number of the query defined by the *Query Manager.* Depending on the state of Trace Query box, the new query manager window displays the result of one of the steps of query scheduling.

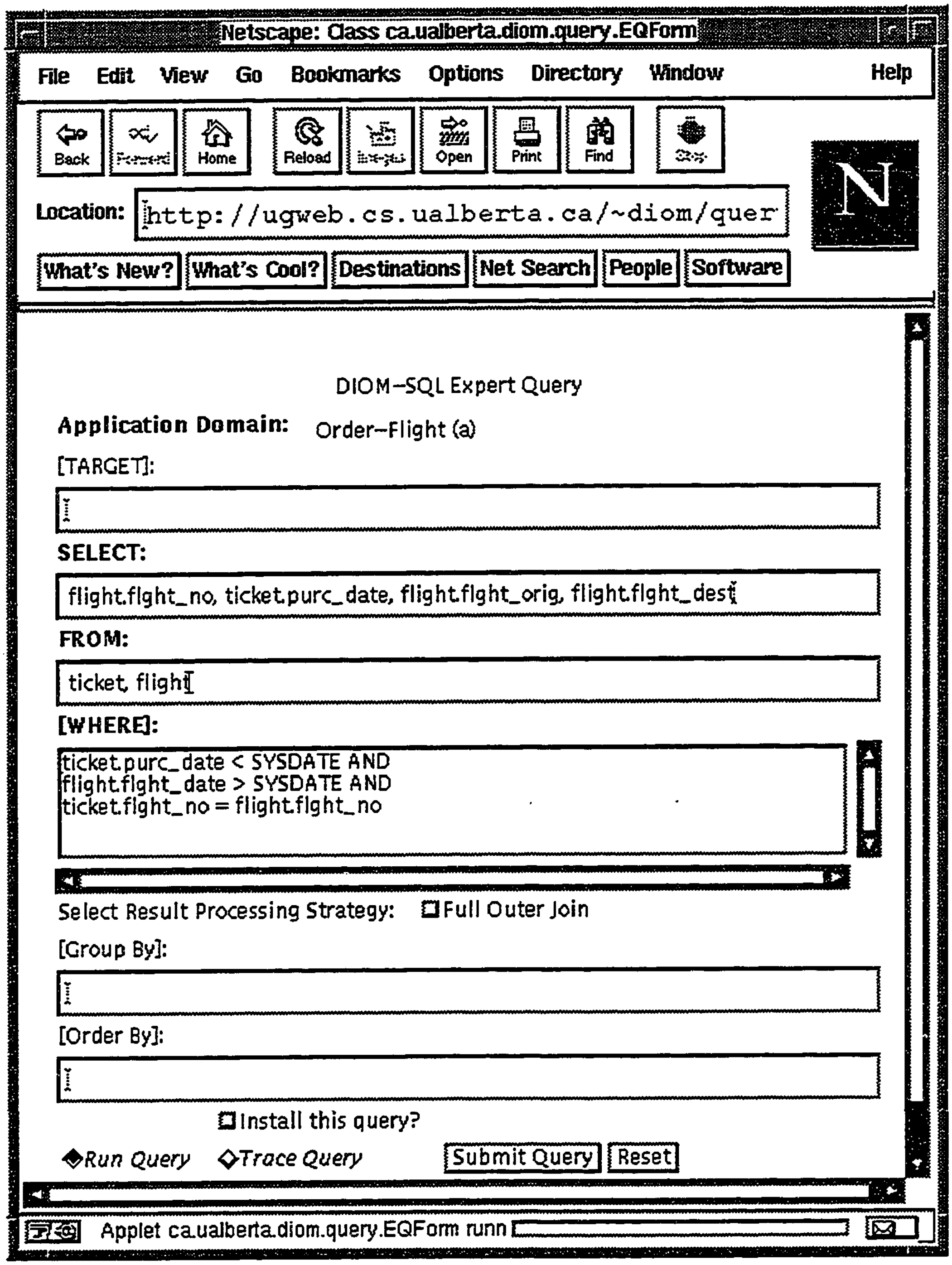
This form supports both mouse-driven and keyboard-driven navigation. The user may navigate through the GUI elements by pressing the TAB key, which, combined with Shift key, reverses the navigation, and launch the query manager by pressing the Return key.

4.5.2 Query Manager Screen

The query manager screen consists of the following components located at the bottom of each query manager window:

Cancel button terminates the current query scheduling process and closes the query manager window and all of its child windows;

<< Start button is enabled at all steps of query scheduling except the first. It brings the query manager into the first state and displays the router panel (shown in Figure 4-8;



Location: http://ugweb.cs.ualberta.ca/~diom/quer

DIOM-SQL Expert Query

Application Domain: Order-Flight (a)

[TARGET]:

SELECT: flight.flght_no, ticket.purc_date, flight.flght_orig, flight.flght_dest

FROM: ticket, flight

[WHERE]:

```
ticket.purc_date < SYSDATE AND
flight.flght_date > SYSDATE AND
ticket.flght_no = flight.flght_no
```

Select Result Processing Strategy: □ Full Outer Join

[Group By]:

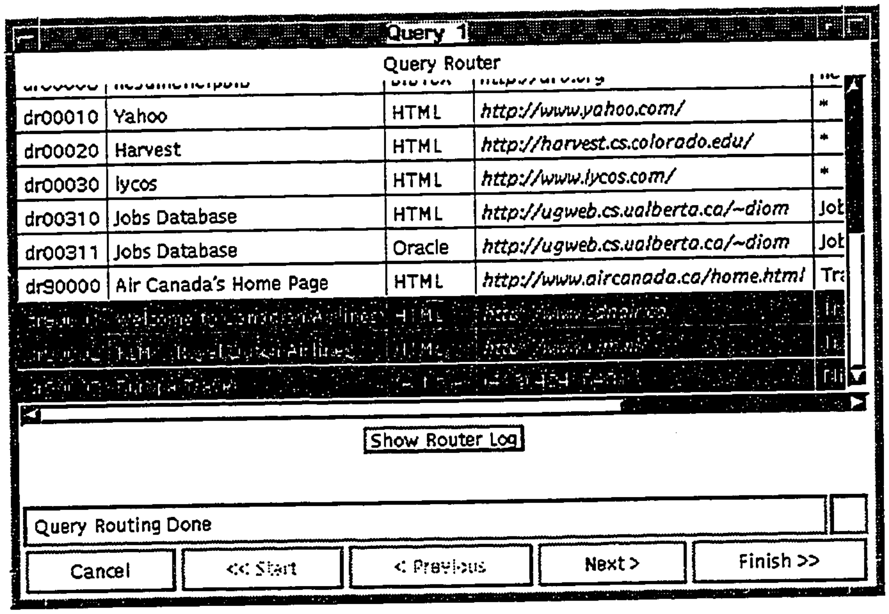
[Order By]:

□ Install this query?

◆ *Run Query* ◇ *Trace Query* [Submit Query] [Reset]

Applet ca.ualberta.diom.query.EQForm runn

Figure 4-7: Query Entry Form Screen



dr00010	Yahoo	HTML	http://www.yahoo.com/	*
dr00020	Harvest	HTML	http://harvest.cs.colorado.edu/	*
dr00030	lycos	HTML	http://www.lycos.com/	*
dr00310	Jobs Database	HTML	http://ugweb.cs.ualberta.ca/~diom	Jot
dr00311	Jobs Database	Oracle	http://ugweb.cs.ualberta.ca/~diom	Jot
dr90000	Air Canada's Home Page	HTML	http://www.aircanada.ca/home.html	Tr
[illegible]	Welcome to Canadian Airlines	HTML	[illegible]	[illegible]
[illegible]	KLM - Royal Dutch Airlines	HTML	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Query Routing Done

Figure 4-8: Router Screen

< Previous button is enabled at all steps of query scheduling except the first. It brings the query manager into the previous state and displays the result of the corresponding query scheduling step;

Next > button is enabled at all steps of query scheduling except the last. It brings the query manager into the next state and displays the result of the corresponding query scheduling step. The user modifications done at the current and any of the previous steps of query scheduling take effect, e.g., if the user excludes one of the automatically selected information sources, and then presses this button, the decomposition step will exclude the source from the new query tree;

Finish >> button is enabled at all steps of query scheduling except the last. It brings the query manager into the last state and displays the query execution result. The user modifications done at the current and any of the previous steps of query scheduling take effect;

The status text field is used to display the important user messages concerning the status of the query manager and its main components;

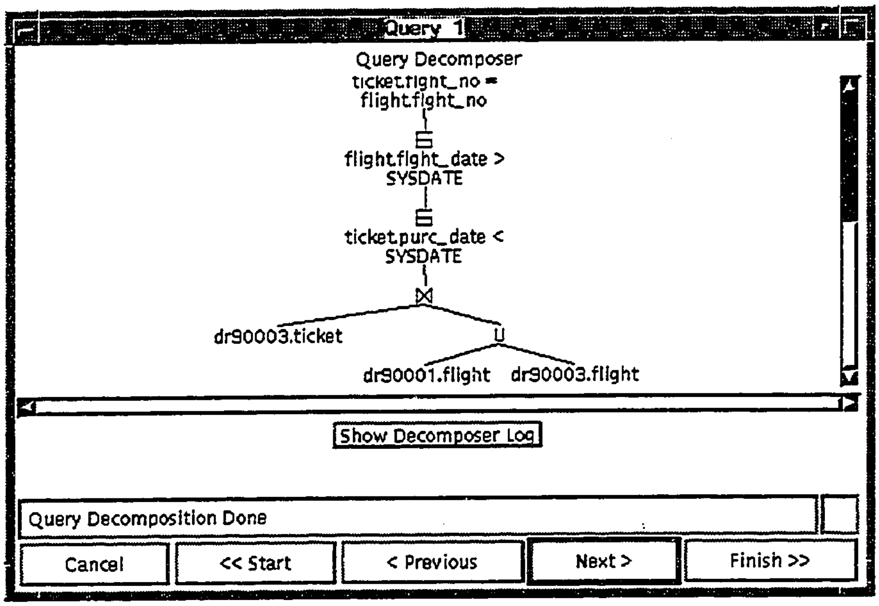
The progress display component (to the right of status field), when animated, indicates that the query manager is in action.

An example of query manager screen containing the router panel is shown in Figure 4-8.

4.5.2.1 Router Panel

The router panel displays the result of the query routing step and is also used for user selection and deselection of information sources, its sample is shown in Figure 4-8. The main part of the router panel is the scrollable GUI component with the canvas that displays the visible portion of the router table. Each line in the table corresponds to one information repository currently registered with DIOM. The user can select or deselect each source by clicking the mouse in the corresponding line. The automatically selected sources are highlighted with the reversed foreground color, and the user-selected sources are highlighted with the reversed brighter color in the same palette.

The other component of this panel is Show Router Log button that brings up a Log View Window that contains the router log. The GUI of the Log View Window is covered in detail in Section 4.5.3.



Query Decomposition Done

Figure 4-9: Query Tree Screen displaying decomposition tree for query in Figure 4-7.

4.5.2.2 Decomposer and Heuristic Processor Panels

A sample of Decomposer/Heuristic Processor Panels is shown in Figure 4-9. This panel consists of the following GUI components:

main display area displays the visible portion of the query tree. By resizing the window or manipulating the scrollbars the user may view the entire tree;

Another Heuristic button is present on the Heuristic Processor Panel and is used to display the query trees corresponding to each of the heuristic processors installed in the package; The name of the ID number of the current heuristic is displayed in the status text field of the query manager;

Show * Log button brings up a Log View Window that contains the log of the current processor, where * can be one of Router, Decomposer, Heuristic, or Cost. The GUI of the Log View Window is covered in detail in Section 4.5.3. A sample log for the *Move Joins Heuristic Processor* is shown in Figure 4-12.

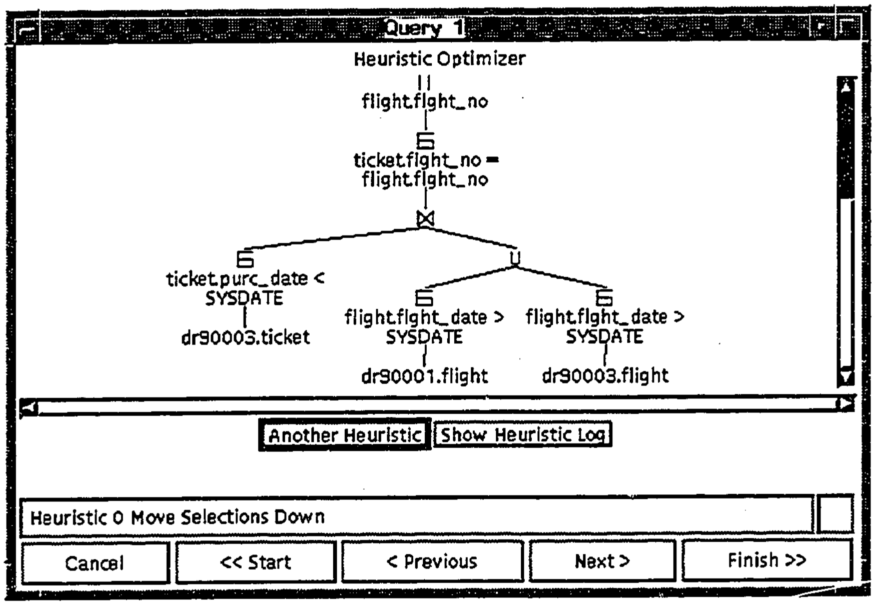
If the total height or width of the tree is less than the height or width of the visible area, the user can use the scrollbars to see the desired are of the tree, or resize the query manager window to see the entire tree. Otherwise the tree is centered horizontally and top-adjusted vertically in the display area.

Figure 4-9 on page 73 displays the lower part of the resulting query tree from query decomposition. Figure 4-10 on page 74 displays the lower part of the result tree at heuristic-driven query optimization phase by applying heuristic *Move Selections Down.*

4.5.2.3 Cost Processor Panel

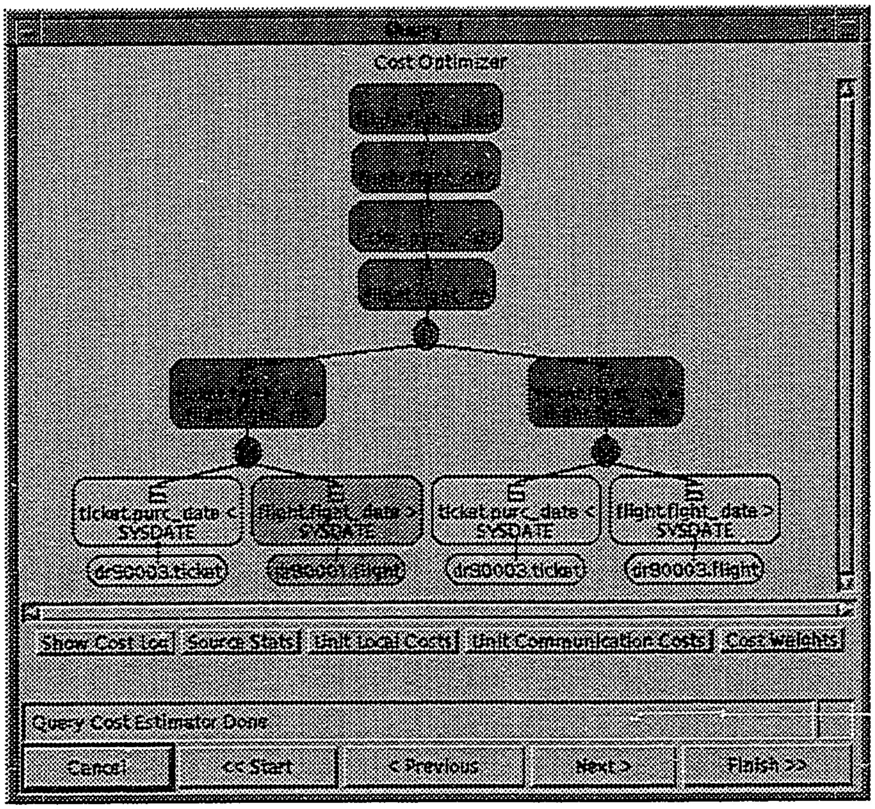
The Cost Processor Panel is shown in Figure 4-11 on page 74. This panel consists of the following GUI components:

main display area displays the visible portion of the cost query tree (see section 4.2.3) where each node is highlighted with a site-specific color. Mouse click in the highlighted area of a node brings up a Log View Window that contains the site and cost-related information about this node and the detailed statistical information of the objects at this node and their attributes;



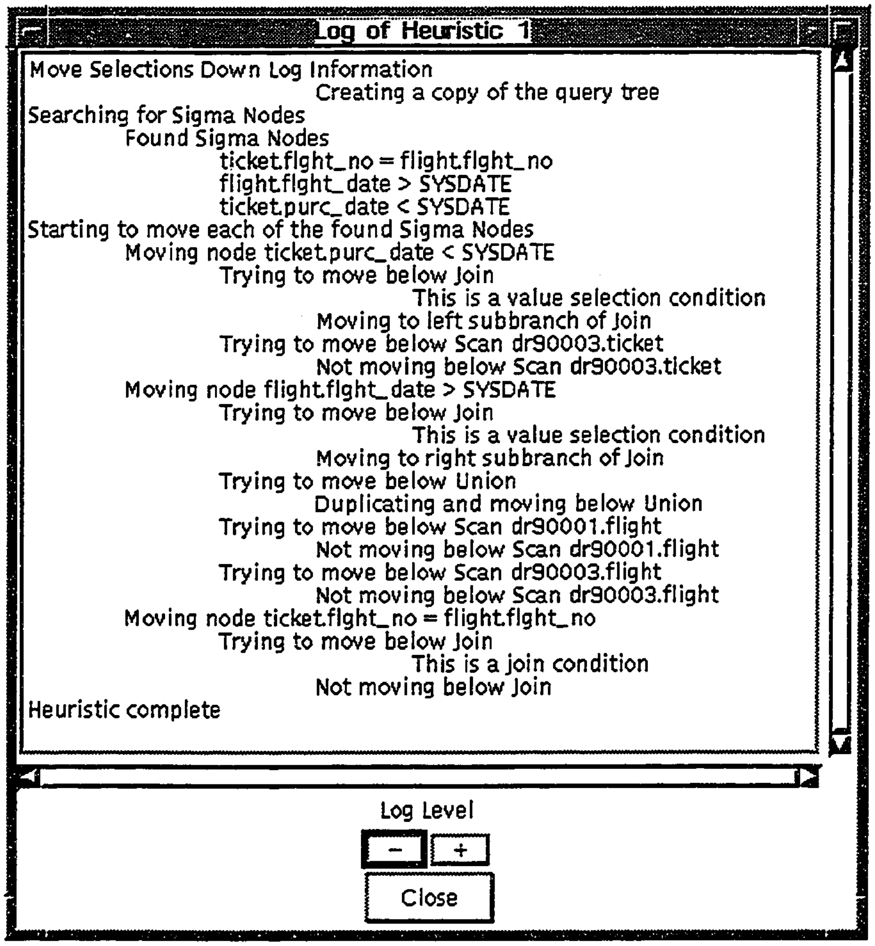
Heuristic 0 Move Selections Down

Figure 4-10: Query Tree Screen displaying the result of applying *Move Selections Down* heuristic to the decomposition tree shown in Figure 4-9.



Query Cost Estimator Done

Figure 4-11: Cost Processor Panel Screen



```
Move Selections Down Log Information
                              Creating a copy of the query tree
Searching for Sigma Nodes
          Found Sigma Nodes
                    ticket.flght_no = flight.flght_no
                    flight.flght_date > SYSDATE
                    ticket.purc_date < SYSDATE
Starting to move each of the found Sigma Nodes
          Moving node ticket.purc_date < SYSDATE
                    Trying to move below Join
                                        This is a value selection condition
                              Moving to left subbranch of Join
                    Trying to move below Scan dr90003.ticket
                              Not moving below Scan dr90003.ticket
          Moving node flight.flght_date > SYSDATE
                    Trying to move below Join
                                        This is a value selection condition
                              Moving to right subbranch of Join
                    Trying to move below Union
                              Duplicating and moving below Union
                    Trying to move below Scan dr90001.flight
                              Not moving below Scan dr90001.flight
                    Trying to move below Scan dr90003.flight
                              Not moving below Scan dr90003.flight
          Moving node ticket.flght_no = flight.flght_no
                    Trying to move below Join
                                        This is a join condition
                              Not moving below Join
Heuristic complete
```

Figure 4-12: Log View Window Screen

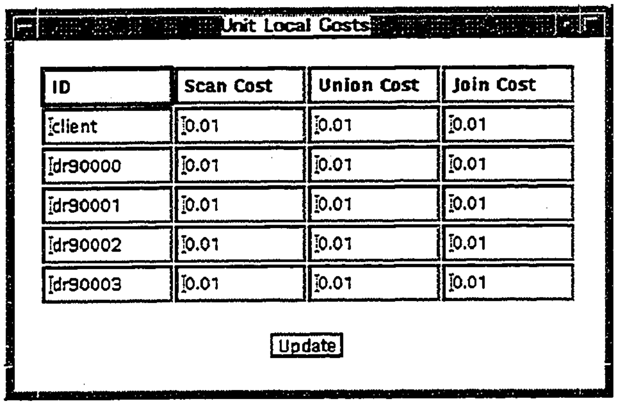
Show Log button button brings up a Log View Window that contains the log of the cost processor;

Source Stats button, Unit Local Costs button, Unit Communication Costs button, and Cost Weights button each bring up a Parameter Edit Window filled with statistical parameters, unit local cost parameters, unit communication cost parameters, and cost weights parameters respectively. If the user updates the parameters and reruns the *Cost Processor*, then the new query tree will reflect the update. The GUI components of the Log View Window and the Parameter Edit Window are covered in detail in Section 4.5.3. An example of the Parameter Edit Window with unit local cost parameters is shown in Figure 4-13.

4.5.3 Common GUI Components

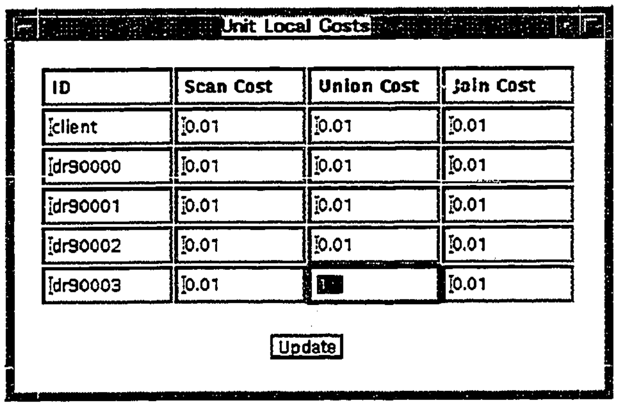
4.5.3.1 Log View Window

An example of Log View Window is shown in Figure 4-12 on page 75. It displays the log of *Move Selections Heuristic Processor* The log shown in Figure 4-12 illustrates how the first heuristic *Move Selections Down* is applied to the query tree shown in Figure 4-9 on page 73. It results in the query tree shown in Figure 4-10 on page 74.



ID	Scan Cost	Union Cost	Join Cost
client	0.01	0.01	0.01
dr90000	0.01	0.01	0.01
dr90001	0.01	0.01	0.01
dr90002	0.01	0.01	0.01
dr90003	0.01	0.01	0.01

Figure 4-13: Parameter Edit Window Screen with *Unit Local Cost* parameters.



ID	Scan Cost	Union Cost	Join Cost
client	0.01	0.01	0.01
dr90000	0.01	0.01	0.01
dr90001	0.01	0.01	0.01
dr90002	0.01	0.01	0.01
dr90003	0.01	1.0	0.01

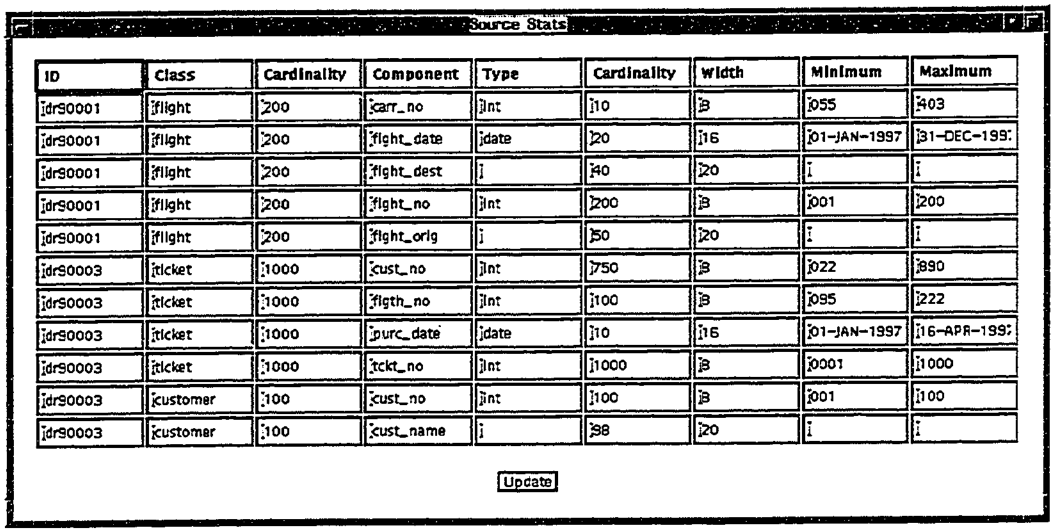
Figure 4-14: Parameter Edit Window Screen displaying *Unit Local Cost* parameters after user update.

The main component of this window is the scrollable text area that displays the multi-line log information. In the bottom of the window is the control panel that contains the following buttons:

- + and − buttons allow to change the level of log detail. Figure 4-10 shows the full log, with maximum log level. The user can change the appearance of the log by pressing on one of these buttons;
- Close button disposes of this window and frees the system resources.

4.5.3.2 Parameter Edit Window

Figures 4-13 through 4-15 show examples of Cost Parameter Edit Window. The main area of this window is a collection of text fields where the user may modify the cost parameters, e.g. Figure 4-14 shows how the parameter window looks after the user updates one of the *Unit Local Cost* parameters. See the change of unit local processing cost at dr90003 from 0.01 to 1.0. Similar change operation may apply to statistical information shown in Figure 4-15 on page 77. Each cost parameter edit window also contains an Update button that copies the parameters from the window to the current query manager. The query manager will automatically incorporate the updated parameters into the cost processor component if the user repeats the query cost evaluation step.



ID	Class	Cardinality	Component	Type	Cardinality	Width	Minimum	Maximum
dr90001	flight	200	carr_no	int	10	8	055	403
dr90001	flight	200	flight_date	date	20	16	01-JAN-1997	31-DEC-199
dr90001	flight	200	flight_dest		40	20		
dr90001	flight	200	flight_no	int	200	8	001	200
dr90001	flight	200	flight_orig		50	20		
dr90003	ticket	1000	cust_no	int	750	8	022	890
dr90003	ticket	1000	fligth_no	int	100	8	095	222
dr90003	ticket	1000	purc_date	date	10	16	01-JAN-1997	16-APR-199
dr90003	ticket	1000	tckt_no	int	1000	8	0001	1000
dr90003	customer	100	cust_no	int	100	8	001	100
dr90003	customer	100	cust_name		98	20		

Figure 4-15: Parameter Edit Window Screen displaying *Source Statistics* parameters.

4.6 Query Processing Demonstration Experiments

This section provides two case examples that show the *DIOM Query Scheduling Utility* in action. The first example shows how the *Move Joins Heuristic Processor* decides to apply the heuristic based on the changed query tree, and the second example shows how the *Cost Processor* assigns the query operators to different sites based on the changed statistical and cost parameters.

4.6.1 Demonstration of Heuristic Processors

Consider the running example query in the Flight-Ticket-Order Domain (see Figure 4-7 on page 71). The router selects three information sources as relevant to this query (see Figure 4-8). The decomposer step results in a query decomposition tree with four scan nodes, two of which are done at site dr90003 (Figure 4-16).

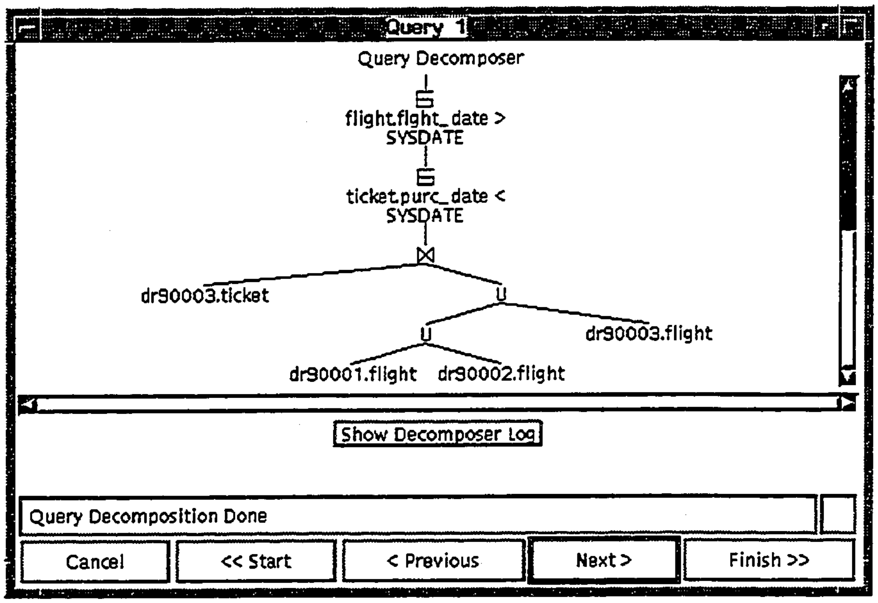
The *Move Joins Heuristic Processor* takes this tree as an input and decides not to move the join down. This decision was based on the Heuristic 3.4 given on page 20. The *Move Joins Heuristic Processor* provides the user with the log information shown in Figure 4-17, which explains how this decision was made.

The *Move Joins Heuristic Processor* locates all the scan nodes present in the tree and identifies the information source associated with each of the scan nodes. For each information source, it counts the number of scan nodes that are assigned to it. If fewer than half of all scans are done at the same source, it will decide not to move the join nodes, thus leaving the tree unchanged.

Now, let us try to see if this heuristic will work on a slightly modified tree. Return to the router panel, deselect source dr90002 by clicking the pointing device on its line in the router table, and run the heuristic processors again by using Next button. Note that the decomposition tree has changed and now the decomposition tree does not include the scan node at dr90002 site (see Figure 4-9 on page 73). The *Move Joins Heuristic Processor* has now produced a query tree with the join operators moved below the union (see Figure 4-18 on page 79). The new log of this processor is shown in Figure 4-19 on page 79.

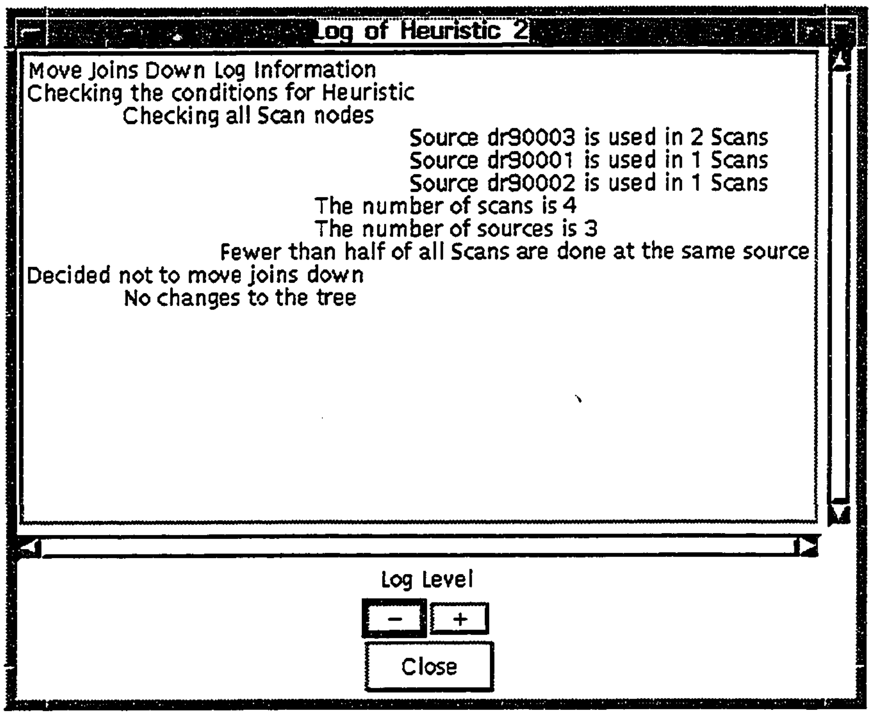
4.6.2 Demonstration of Cost Processor

Execute the *Cost Processor* step for the modified query tree in the last example (see Figure 4-18 on page 79). Note that the scheduler assigns all three binary operators, i.e., two joins and one union in



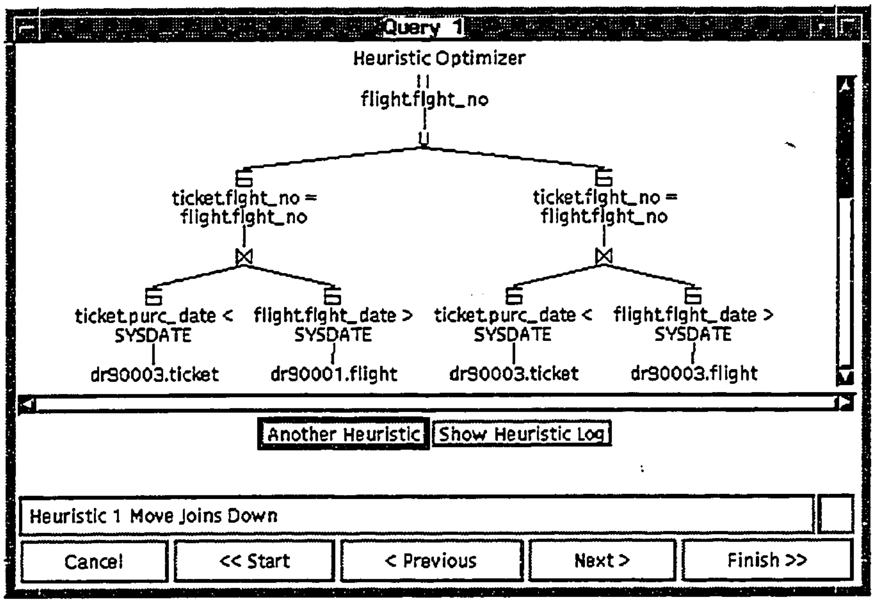
Query Decomposition Done

Figure 4-16: Query Decomposition Tree for query shown in Figure 4-7, which corresponds to the Router results shown in Figure 4-8.



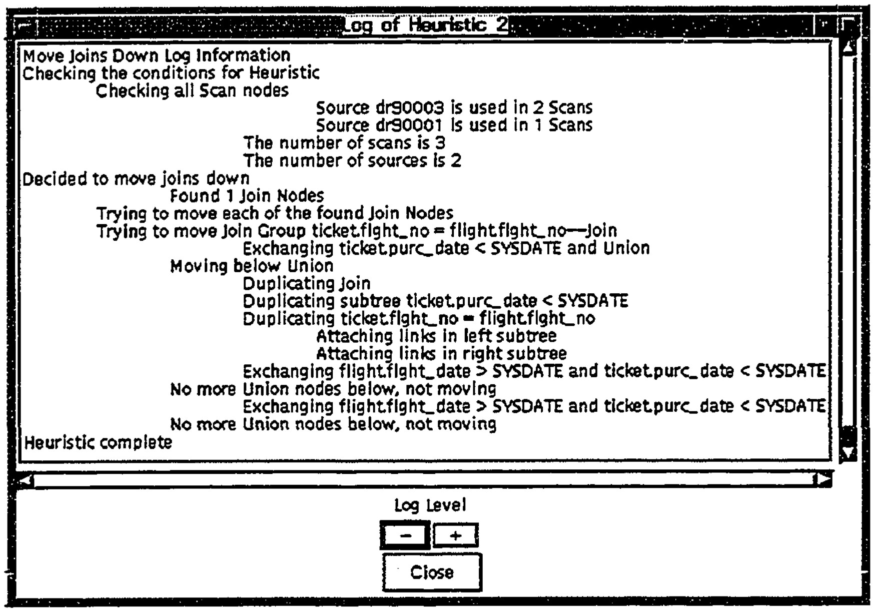
```
Move Joins Down Log Information
Checking the conditions for Heuristic
        Checking all Scan nodes
                                Source dr90003 is used in 2 Scans
                                Source dr90001 is used in 1 Scans
                                Source dr90002 is used in 1 Scans
                        The number of scans is 4
                        The number of sources is 3
                Fewer than half of all Scans are done at the same source
Decided not to move joins down
        No changes to the tree
```

Figure 4-17: Screen showing the log information of *Move Joins Down* heuristic processor when it decided not to move the join down.



Heuristic 1 Move Joins Down

Figure 4-18: Screen showing the result of applying *Move Joins Down* heuristic corresponding to decomposition tree shown in Figure 4-9.



```
Move Joins Down Log Information
Checking the conditions for Heuristic
          Checking all Scan nodes
                              Source dr90003 is used in 2 Scans
                              Source dr90001 is used in 1 Scans
                    The number of scans is 3
                    The number of sources is 2
Decided to move joins down
               Found 1 Join Nodes
          Trying to move each of the found Join Nodes
          Trying to move Join Group ticket.flight_no = flight.flight_no—Join
                    Exchanging ticket.purc_date < SYSDATE and Union
               Moving below Union
                    Duplicating Join
                    Duplicating subtree ticket.purc_date < SYSDATE
                    Duplicating ticket.flight_no = flight.flight_no
                              Attaching links in left subtree
                              Attaching links in right subtree
                    Exchanging flight.flight_date > SYSDATE and ticket.purc_date < SYSDATE
               No more Union nodes below, not moving
                    Exchanging flight.flight_date > SYSDATE and ticket.purc_date < SYSDATE
               No more Union nodes below, not moving
Heuristic complete
```

Figure 4-19: Screen showing the log information of *Move Joins Down* heuristic processor when it decided to move the join down.

```
Union
Total Cost = 197559
Local Processing Cost = 95116.6
        Communication Cost = 27840
        Response Time Cost = 74602.2
Operator Assigned to Site dr90003
        Total Extent Size = 40144
        Number of Aggregate Components = 8
```

(a) before updating the unit local cost

```
Union
Total Cost = 389015
        Local Processing Cost = 95116.6
        Communication Cost = 163712
        Response Time Cost = 130186
Operator Assigned to Site client
        Total Extent Size = 40144
        Number of Aggregate Components = 8
```

(b) after updating the unit local cost

Figure 4-20: Comparison of cost information for the *Union* node.

Figure 4-11 on page 74, to site dr90003, which is expected because most of the relevant information for this query is found at that site. Consider the log information of the query node containing the *Union*. To open the log view window click on the union node.

Now, consider modifying the unit local processing cost at that site. Open the Unit Local Costs window by pressing the button in the Cost Processor Panel, in column Union Cost of site dr90003, modify the text field so that it contains 1 (as shown in Figure 4-14). Re-run the *Cost Optimizer* by pressing Previous and Next buttons on the query manager window. Note that the site and the cost of the node containing the *Union* have changed (note highlighting color used for this node). Again, open the log of this node and compare it with the old log. The two logs are shown in Figure 4-20 on page 80. Note that not only the site but the cost components of this node has changed. The communication cost is six times higher.

Although the change was only made to the unit local processing cost of site dr90003, the overall local cost has not been changed because the union is now scheduled at the client site. The scheduler has decided to assign the union to the client site and thus compensate for the 100-time increase in the local processing costs of the union at site dr90003. As a result, the total cost has almost doubled.

Chapter 5

Related Work and Conclusion

This Chapter contains a section that outlines some of the related work that is most representative in the area of distributed query processing and optimization, and a summary of the work done in the present thesis, it also outlines the contributions of this thesis, and comments on its possible future developments and improvements.

5.1 Overview of Related Work

Most theoretical research of the present thesis is based on the results obtained in [18, 1, 5]. In this section we provide a brief summary of the main ideas proposed in each of these papers.

5.1.1 Query Optimization in System R

The paper describes query optimization techniques used in *System R*, a centralized experimental database management system based on the relational model [18].

Query processing in this system consists of four phases: parsing, optimization, code generation, and execution. The main focus of the paper is on the optimization phase.

The optimizer in *System R* performs global query optimization by evaluating the cost of each possible query execution plan and selecting the plan with the minimum cost. Statistical information, such as estimated relation cardinality, and index cardinality, along with the selectivity factors for each of the relational operators, are used to evaluate the cost of each query execution plan.

Two techniques are used for executing the join operator: method of nested loops, and method of merging scans. The optimizer presented in the paper selects the most beneficial of the two methods to execute each of the joins in the query being processed. For queries with multiple join operators the optimizer also chooses the optimal join order that results in the minimum execution cost.

To reduce the number of join order permutations the authors propose to use a heuristic by which the joins between relations that do not have a common join column, i.e., the ones requiring Cartesian product, are performed as late in the join sequence as possible [18]. To find the optimal join order among the remaining join permutations the following search is performed:

1. the optimizer finds the best way to access each single relation found in the FROM clause of the query;

2. the optimizer finds the best way to join any of the remaining relations to any of the relations optimized at step 1, thus forming the optimized sequences of two relations;

3. the optimizer finds the best way to join any of the remaining relations to any of the two-relation sequences obtained at step 2, thus forming the optimized sequences of three relations, etc;

4. when all relations are included in the optimized sequence, the optimizer chooses the sequence with the minimum cost.

The cost of joins is estimated based on the selectivity factors and the available statistical information about the arguments of the join operators.

The contributions of the query optimizer are the following [18]:

- extensive use of statistical information in estimating the cost of the query execution plan;
- global query optimization, which is achieved by exhaustive search of all possible join orders;
- inclusion of ORDER BY and GROUP BY clauses into the process of optimization, which results in avoiding the need to sort the intermediate query results;
- use of heuristics that allow to prune the search of the optimal query plan;

As authors point out, their optimizer makes the query performance of the database systems that support non-procedural query languages comparable to the query performance of DBMS that are based on procedural query languages.

Unlike DIOM, which is a distributed system, the work presented in this paper is targeted at a centralized database management system. This fundamental difference does not allow us to employ most of the query optimization techniques described in this paper [16]. However, this paper has proven to be a valuable source for our work in the present thesis in the following aspects:

- In our *Distributed Query Scheduling* prototype we propose to use the same statistical model for evaluating the cardinalities of intermediate query results as the model used in this paper;
- We utilize the global approach to the optimization problem, i.e., we consider all possible site assignments to select the optimal query execution plan. However, we have not applied this technique for join ordering problem. We anticipate that the application of our heuristic-based query processor will reduce the complexity introduced by the presence of inter-site joins;
- We use the heuristic-based approach to reduce the solution space of the optimization problem prior to evaluating the cost of each of the query plans, which significantly reduces the complexity of the problem.

5.1.2 Query Processing in SDD-1

The main objective of [1] is to describe a technique called *semijoin* and the application of this technique to distributed query optimization in the distributed version of *System R SDD-1.* Semijoin is an operation that allows to reduce the quantity of inter-site data transfer needed to execute an inter-site join operation.

The algorithm proposed in this paper consists of three main steps [1]:

1. mapping the user query into relational calculus form (referred to as *envelope*);
2. evaluating the envelope. Evaluation is accomplished by translating the relational calculus query into a program that contains relational operations (referred to as *reducers*) and the *move* commands that move the results of *reducers*. The goal of this step is to construct such a program that the cost of computing *reducer* and moving its results to the site where the result is assembled is minimum over all reducers and sites;
3. executing the user query at the result assembly site using the data assembled by step 2.

The paper is focused on the optimization problem posed by step 2 of the query processing algorithm.

The paper defines the notions of *envelope* and *reducer* in terms of relational algebra. Also, it defines the *benefit* of a *reducer* as the amount of data it eliminates, and the *cost* of a *reducer* as the amount of inter-site data transfer required to compute this *reducer*.

Relational *selection* and *projection* operators are one type of *reducers*, they have nonnegative benefit and zero communication cost (because they do not require any inter-site data transfer). Relational *join* operator is another type of *reducer*, its cost-to-benefit ratio may vary depending on the size of its arguments and on the size of its result.

The authors propose to use operator called *semijoin* instead of *join* to achieve the same data reduction. The cost of *semijoin* is lower than the cost of *join*. Although the *benefit* of a *semijoin* may be lower than the *benefit* of *join*, the combination of two symmetrical *semijoins* results in the same *benefit* as *join*. See [1] for full definition of the *semijoin* operator.

The authors propose a method to estimate the cost and the benefit of reducers. They first introduce a statistical model of the database, then they provide an algorithm that evaluates the *selectivity* of relational *selection* operator and the *selectivity* of the *semijoin* operator. The selectivity allows to estimate the *cardinality* of the result of the operator based on the *cardinalities* of its operands.

The authors present the optimization algorithm for performing step 2 of query processing. This algorithm performs an iterative hill-climbing procedure that at each iteration selects the most profitable semijoin and appends it to the resulting query execution program until all profitable semijoins have been exhausted [1]. As the authors concede, the proposed algorithm is an example of a *greedy* optimization algorithm. Such algorithm breaks down the complex optimization problem into simple subproblems and then solves each of these subproblems separately. The advantage of this approach is that the complexity is reduced significantly, while the disadvantage is that the solution found by such algorithm is not guaranteed to be optimal. There are two enhancements to this algorithm proposed in the paper, which we will not discuss here as they do not change the nature of the optimization approach taken by the authors. For details, see [1].

The *Distributed Query Scheduling* prototype that we propose uses many of the ideas presented in this paper. For instance, we propose to employ the same statistical model and the cost estimation techniques for generating the optimal query execution plan.

However, there are a number of fundamental differences between **SDD-1** and the DIOM system that prevent us from using the semijoin technique:

- **SDD-1** is a classical distributed database management system, therefore it does not address the heterogeneity issue, which affects query processing significantly, e.g., there may be constraints imposed by the information sources that limit the optimizer's ability to assign certain query operators to it, therefore the optimization algorithm presented in the paper does not suit the requirements of DIOM query processing;

- the main assumption of the authors is that the communication cost is the dominant factor in the total cost of query evaluation. At the time the paper was written this assumption was legitimate as the performance bottleneck of the distributed database systems was the network connection. In our prototype of *DQS* we propose to use a more flexible cost model that takes into account not only the communication cost but also the local processing cost and response time parameters.

5.1.3 Query Processing in a Multidatabase System

This paper was written to summarize the query processing techniques used in the *Multibase* system [5]. *Multibase* is a system that integrates a collection of distributed databases that are heterogeneous and possibly inconsistent with one another. It keeps a global schema that facilitates a uniform access to these databases.

The global query posed in such multidatabase system is processed in four steps:

1. query modification, in which the global query is modified into a query over local schemas;

2. global query optimization, in which a global execution plan is constructed. The plan consists of a set of local queries, a set of move instructions, and a set of post-processing instructions;

3. local query optimization;

4. translation of the local query plans into the language of the local schemas for execution.

The global query optimization step is the main focus of the paper. The following tactics are used in global query optimization:

- moving selections and projections below inter-site joins and unions, if possible;
- performing the semiouterjoin operation – unlike semijoin, this operation addresses the issue of possible inconsistency among the local databases. Although semiouterjoin does not have an immediate gain in the cost of query execution, it facilitates the subsequent distribution of relational selections and joins;
- distributing inter-site join operations to the local databases, if it is possible and beneficial;
- using semijoins if the corresponding joins are distributed and if it is beneficial.

The paper describes the cost estimation model used to determine the benefit of the considered query optimization decisions. Based on the assumption of equal unit costs throughout the distributed database, it states that the cost of a query operation is defined by the size of its operands. The information about the size of the intermediate results can then be statistically estimated and used for subsequent query operators.

The paper also provides a summary of the available heuristic-based optimization techniques, such as applying semijoins "greedily", as done in [1], or enumerating all possible join orders, as done in [18]. As the middle ground, between these two heuristics, the paper proposes to use a "parametric hill climbing" heuristic that examines join sequences of length k at a time, where k is the heuristic parameter.

Another issue the paper addresses is the effect of aggregation functions on the distributed query processing. Due to the fact that the data can be inconsistent across local databases, relational selections and projections on a distributed relation may not always be performed before the relation is materialized at the central site. The authors present a detailed analysis of such cases (see [5] for details on this issue).

The main architectural difference between the system presented in [5] and DIOM is that DIOM uses query mediator-wrapper approach to solve the problem of heterogeneity of the information producers [14], while the multidatabase model of [5] uses the global schema that logically integrates the local databases through the global view mechanism. Due to this difference query processing in DIOM is a more complex problem than in the multidatabase model described in [5]. For instance, such optimizing techniques as semijoins and semiouterjoins can not be used directly. Moreover, the aggregation query functions pose an even greater problem in DIOM environment than in the multidatabase environment.

Our version of *Distributed Query Scheduling* prototype does not address these issues, leaving them for future development. At the same time our prototype does employ such techniques as query cost estimation and moving the joins down, presented in [5]. Unlike the system described in [5], *DQS* prototype uses the global optimization criteria for evaluating the cost of a distributed query (see Chapter 3), which allows to profile the user's preferences. We also use the site-specific unit cost information, which allows to calibrate each of the information sources registered with DIOM.

5.2 Summary and Contribution of The Thesis

The present thesis documents the theoretical research and practical implementation of the *Distributed Query Scheduling (DQS)* prototype for Distributed Interoperable Object Model. Our work on this topic started from research of the related work in the area of distributed query processing and optimization, namely [1, 5, 18]. Based on the analysis of the task of distributed query scheduling in DIOM environment and the previous work in DIOM [14, 12, 10] we have accomplished the following:

- identified the necessary components of the distributed query scheduling prototype, the heuristic-based query processing component, and the cost-based query processing component;
- adapted the the heuristic-based optimization techniques of [5] to the heterogeneous environment of DIOM and proposed a number of modified approaches to heuristic-based query optimization;

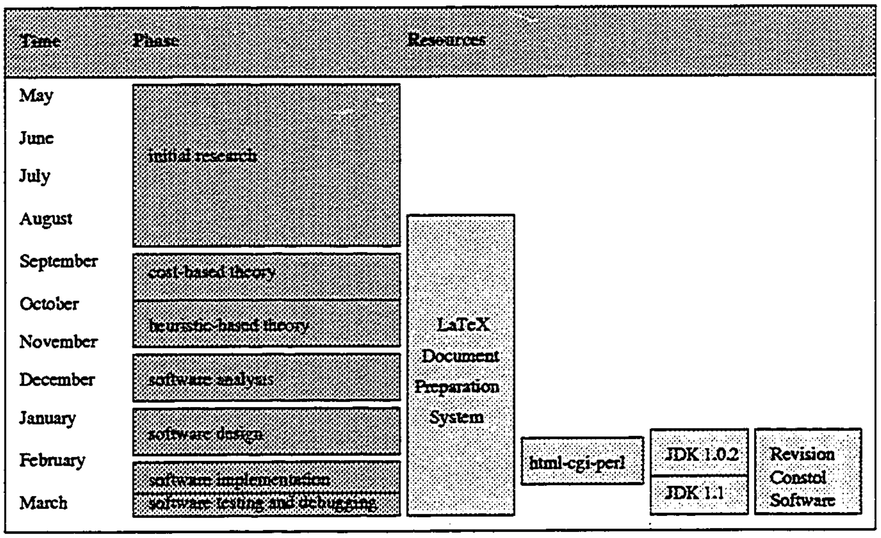


Figure 5-1: Time and Resources Used in The Project.

- unified the techniques for evaluating the statistical information from [1, 18] and generalized them for the prototype of cost-based component of *DQS*;

To provide evidence of viability of the proposed techniques we have carried out a task of implementing these techniques in the form of a *Distributed Query Scheduling* software that demonstrates the intrinsics of the proposed techniques. To accomplish this task we have

- performed a detailed requirements analysis of the *DQS* software and adopted the object-oriented model for the subsequent software development phases;
- designed the main components of the *DQS* software package using object-oriented approach;
- implemented an experimental version of the software package using Java programming language;
- tested and debugged the software and delivered the first prototype of the *DQS* software package that includes the on-line query processing utility, the on-line developer's documentation, and the user's documentation.

5.2.1 Project Statistics

This project commenced in May, 1996 and was completed in April 1997. Figure 5-1 shows the breakdown of time and resources we used. The code design, implementation and testing of the *DQS* software package started in November, 1996 and finished in March, 1997. The resulting software package has approximately 4300 lines of code and 2000 lines of documentation.

5.3 Comments on Future Improvement

The work done in the present thesis is a prototype for distributed query scheduling process in DIOM. Being a prototype, it does not solve all the problems of distributed query processing and will require much more work and research to make it a production software. Below we outline possible future development of this work:

- design and implementation of all heuristics described in Chapter 3 in the *DQS* software package, e.g., the heuristic of moving projections down and the heuristic of ordering the joins in the query tree;
- incorporation of `group by` and `order by` IQL clauses into the query scheduling algorithm and research of their effect on the optimization problem;
- implementation improvements to the cost-based query processor to use a variety of the available statistical parameters, e.g., implementation of statistical parameters for *string* datatype as described in Chapter 3;
- design and implementation of additional query operators, such as aggregate operators (such as SUM, AVERAGE), and analysis of their effect on query processing;
- incorporation of a richer functionality front-end component of the software, e.g., the option to install the query to DIOM server;
- incorporation of the complete query routing and the query result assembly modules into the package;
- support of continual queries [15] as a possible option for distributed query scheduling;
- adaptation of JDBC technology to facilitate more efficient data access between the client and the DIOM server.

Besides the issues specific to the *Distributed Query Scheduling* software, the DIOM team currently works on a number of other issues that are beyond the scope of this thesis but still are relevant to the problem of query processing [14, 12], e.g., the semi-automatic generation of wrappers to ***www-html*** servers, relational database systems, OODB systems, and ASCII file systems (`bibtex` files).

Bibliography

[1] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr., "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, Vol. 6, No. 4, December 1981, pp. 602-626.

[2] M. Betz. "Interoperable Objects: Laying The Foundation for Distributed Object Computing," *Dr. Dobb's Journal: Software Tools for Professional Programmer*, (220), October 1994.

[3] R. Cattell et. al., *The Object Database Standard: ODMG-93 (Release 1.1)* Morgan Kaufmann, 1994.

[4] Wesley W. Chu, and Paul Hurley, "Optimal Query Processing for Distributed Database Systems," *IEEE Transactions on Computers*, Vol. c-31, No. 9, September 1982, pp. 835-850.

[5] Umeshwar Dayal, "Query Processing in a Multidatabase System," *Query Processing: Database Systems, eds. Kim, et al., Springer Verlag, New York*, Vol. 1, pp. 81-108, 1985.

[6] David Flanagan, *Java in a Nutshell,* O'Reilly & Associates, Inc., First Edition, February 1996.

[7] O. R. B. T. Force., "The Common Object Request Broker: Architecture and Specification," *Object Management Group*, 1993.

[8] R. Hull and R. King, "Reference Architecture for The Intelligent Integration of Information (version 1.0.1)," *http://isse.gmu.edu/I3_Arch/index.html*, May 1995.

[9] Henry F. Korth and Abraham Silberschatz, *Database System Concepts,* McGraw-Hill, Inc., Second Edition 1991.

[10] Yoo-Shin Lee, "Prototyping the DIOM Interoperable System (TR96-32)," *Department of Computing Science, University of Alberta*, July 1996.

[11] Ling Liu, "A recursive object algebra based on aggregation abstraction for manipulating complex objects," *Data & Knowledge Engineering,* No. 11, 1993.

[12] Ling Liu and Calton Pu, "An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources," To appear in *Distributed and Parallel Databases: An International Journal* Volume 5, No. 2, 1997.

[13] Ling Liu and Calton Pu, "The Distributed Interoperable Object Model and its Application to Large-Scale Interoperable Database Systems," In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 1995.

[14] Ling Liu, Calton Pu, Yooshin Lee, "Adaptive Approach to Query Mediation across Heterogeneous Information Sources," In *International Conference on Cooperative Information Systems (CoopIS)*, Brussels, Belgium, June 13-19, 1996.

[15] L. Liu, C. Pu, R. Barga, and T. Zhou, "Differential Evaluation of Continual Queries," In *IEEE Proceedings of the* 16^{th} *International Conference on Distributed Computing Systems,* Hong Kong, May, 1996.

[16] M. Tamer Özsu and Patrick Valduriez, *Principles of Distributed Database Systems,* Prentice Hall, 1991.

[17] Roger S. Pressman, *Software Engineering, A Practitioner's Approach,* McGraw-Hill, Third International Edition, 1992.

[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of ACM-SIGMOD*, pp. 82-93, 1979.

[19] A. Sheth and J. Larson. "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Surveys.*, Vol. 22, No.3, pp. 183-236, 1990.

[20] Jeffrey D. Ullman, *Principles of Database Systems,* Computer Science Press, Second Edition 1982.

[21] G. Wiederhold, *Intelligent Information Integration Glossary,* Draft 7, March 16, 1995.

[22] Yao, S.B., "Approximating block accesses in database organizations," *Communications of The ACM*, Vol. 20, 4 (April), pp. 260-261, 1977.

Appendix A

DQS Code Implementation

The code excerpts contained in the draft have been removed because the complete code is available on-line at URL `http://www.cs.ualberta.ca/~diom/query/src/`.