Distributed Query Scheduling Service: An Architecture and Its Implementation

Ling Liu and Calton Pu
Oregon Graduate Institute
Department of Computer Science & Engineering
P.O.Box 91000 Portland
Oregon 97291-1000 USA
{lingliu,calton}@cse.ogi.edu

Kirill Richine
University of Alberta
Department of Computer Science
GSB615, Edmonton
T6G2H1 AB, Canada
kirill@cs.ualberta.ca

Abstract

We present the systematic design and development of a distributed query scheduling service (DQS) in the context of DIOM, a distributed and interoperable query mediation system [26]. DQS consists of an extensible architecture for distributed query processing, a three-phase optimization algorithm for generating efficient query execution schedules, and a prototype implementation. Functionally, two important execution models of distributed queries, namely *moving query to data* or *moving data to query*, are supported and combined into a unified framework, allowing the data sources with limited search and filtering capabilities to be incorporated through wrappers into the distributed query scheduling process. Algorithmically, conventional optimization factors (such as join order) are considered separately from and refined by distributed system factors (such as data distribution, execution location, heterogeneous host capabilities), allowing for stepwise refinement through three optimization phases: compilation, parallelization, site selection and execution. A subset of DQS algorithms has been implemented in Java to demonstrate the practicality of the architecture and the usefulness of the distributed query scheduling algorithm in optimizing execution schedules for inter-site queries.

Keywords: Distributed and Cooperative Information System, Distributed Query Scheduling, Distributed Heterogeneous Information service.

1 Introduction

Over the last few years the explosive growth of Internet made a vast amount of diverse information available, but increasingly difficult to access. Referral services (e.g., Yahoo) provide static, expert-selected pointers to the most useful sites, and search engines (e.g., AltaVista and InfoSeek) provide dynamic, user-selected pointers to the most likely web pages of interest. However, our ability to obtain the relevant information is limited by the time we spend *browsing*, which remains the dominant interaction mode with the Internet. Furthermore, browsing with the current generation of search engines is limited to keyword-based search, i.e., user supplies a set of representative keywords and the search engines apply partial string matching techniques on web page title or some content description fields. Despite its success in Information Retrieval applications, keyword-based search has its limitations; for example, it lacks crucial database query facilities such as inter-site joins. Such facilities can be extremely useful for fusion of information over multiple structured or semi-structured data sources. Suppose we are interested in purchasing a 1998 Toyota car. We need to combine the information on model reliability (e.g., from consumer evaluation or expert reviews), average car prices (e.g., the NADA Blue Book), with actual car availability and prices from car dealers. While the information could be joined in the user's mind (as it is done today), it is desirable to design services capable of bringing inter-site join results instead of raw information.

Traditional query planning and optimization techniques cannot be applied naively, since the maintenance of a global schema is impractical in an open environment, where multiple disparate information sources are changing constantly in numbers, volume, contents and query capabilities. To build an efficient query execution schedule for queries in this environment, there are several issues to be resolved in the query scheduling problem for inter-site query executions:

1. what architecture should one use to process and partition user queries over distributed and possibly heterogeneous information sources, while meeting the demands for site autonomy and fully distributed computation;

2. how do conventional optimization factors (such as join order) influence or relate to distributed system factors (such as data distribution layout, execution location, autonomy and heterogeneity of the remote data sources);

3. which approaches are effective and yet practical for generating an optimized query execution schedule that minimizes the response time or the total processing cost (according to some measure).

The first contribution of the paper is the systematic development of a distributed query scheduling service that addresses these issues in the context of DIOM, a distributed and interoperable query mediation system [26]. The novelty of such a service is two-fold. First, it provides an extensible architecture for distributed query processing, which supports and coordinates between the two possible execution models of distributed queries[1]: (1) *moving query to data* and (2) *moving data to query*. By unifying these two strategies into one framework, those data sources that are lacking of the required search or filtering capabilities can be seamlessly incorporated into the distributed query plans, provided that each of these sources are wrapped by a system-compatible wrapper. Second, such service implements a three-phase optimization algorithm that separates conventional query optimization factors from distributed system factors. At query compilation phase, only conventional optimization factors are considered. At query parallelization phase, the data distribution factor is taken into account and the mechanism by which

[1] A distributed query is a query that is posed (appears) at site A which requires data at site B.

the system-controlled parallel execution is introduced to parallelize the result of the first phase. At the third phase, mechanisms for site selection and execution are applied and an optimized query execution plan is generated. The parallelization phase and the site selection and execution phase can be seen as stepwise refinements of the compilation phase result using distributed system factors.

The second contribution of this paper is a concrete implementation to demonstrate the feasibility of the three-phase distributed query scheduling algorithm that optimizes execution schedules for inter-site queries. The prototype system implements a subset of the proposed query scheduling algorithm in Java, accessible from browsers such as Netscape 4 (`www.cse.ogi.edu/DISC/DIOM/query/EQ.html`). An interesting and useful feature of our Java implementation is the tracing facility that supports user observation of the query scheduling process through trace logs.

The rest of the paper is organized as follows: We discuss the related work in Section 2. Section 3 presents an overview of the DIOM system and its query processor. A running example is used to walk through the three-phase query scheduling process. Section 4 describes the three-phase optimization algorithm. Section 5 describes the prototype 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 Section 4. We conclude the paper in Section 6.

2 Related Work

There have been three approaches to the management of distributed data that have been extensively investigated by the research community or commercial products, namely distributed database systems, distributed information mediation systems, and client-server distributed file systems. Each of the approaches presents some architectural characteristics for processing distributed queries. In this section we first discuss each of the three approaches and summarize their architectural difference, and then discuss the state of art technology in distributed query optimization research that are related to our work.

2.1 Distributed Database Systems

We first consider the approach taken by distributed database management systems [32]. Several distributed DBMS prototypes were developed during early 1980s, such as R* [23, 38], SDD-1 [3], and distributed INGRES [43]. All extended single-site DBMSs to manage relations that were distributed over the sites in a computer network. Various techniques were developed for handling distributed query optimization [3, 38, 8] and distributed transactions [3, 23]. Commercial systems based on these techniques are now available from several relational DBMS vendors.

Architecturally, each distributed DBMS assumes a "share nothing" architecture [42] and supports the basic distributed processing model of "*moving query to data*". Thus, these systems allocate data to the sites in a computer network and the data allocation is managed by a database administrator. Another distinct feature of distributed database systems is to have a dedicated central site that keeps the schema of the database, including the fragmentation and location information. This site coordinates the processing of all queries coming from the users. Due to the critical role of communication parameter in the traditional distributed query processing cost, the semijoin operator was introduced to reduce communication cost of inter-site joins [32]. The main idea of the semi-join operator is to move the key portion of the data at one remote data site to another before performing the actual inter-site join operation.

2.2 Distributed Information Mediation Systems

Distributed information mediation systems are evolved from federated database systems [40]. A main difference between distributed database systems, federated database system and distributed information mediation systems is the open world assumption, namely, the distributed information mediation systems must deal with the dynamics of an open environment, where information sources available on-line are constantly changing in number, volume, content and capability. More concretely, the distributed information mediation systems must deal with the dynamic increase of the solution space from which to choose the distributed execution schedule. Furthermore the statistical information that is accessible in distributed database systems may not be available in open environments due to the autonomy of individual information sources. Therefore, the effectiveness of the distributed query processing 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 the total processing cost or the response time of the query (i.e., the time elapsed for executing the query) or a weighted combination of cost components (see Section 4.1).

Several distributed information mediation systems are being built for providing uniform access over multiple information sources based on the mediator architecture [48]. DIOM [26] is a prototype of the distributed information mediator systems. Other prototype systems include TSIMMIS [12], Garlic [4], CARNOT [6], SIMS [2], DISCO [5]), and Information Manifold [21]. The key aspect distinguishing DIOM from the other systems is its emphasis on scalability and extensibility of the query services by promoting source-independent query processing at the mediator level and by describing the content and capability description of each source independently of description of other sources and of the way how queries are posed. Thus, users may pose queries on the fly and the new sources can be incorporated into the query scheduling process dynamically and seamlessly. For example, SIMS and Information Manifold are the two systems that also describe information sources independently of the user queries. However, neither SIMS nor Information uses an optimized execution schedule to submit the set of subquery plans to the respective data sources (via wrappers). TSIMMIS and DISCO both use a set of sophisticated query rewriting patterns to mediate the queries and the multiple information sources. TSIMMIS provides a rich set of pre-defined query patterns and map each query into one of the pre-defined patterns and then apply the rewriting rules for each pre-defined pattern to generate an executable plan. It is not clear whether the set of pre-defined patterns is extensible and the new patterns can be added into their rewriting system seamlessly. DISCO uses multiple F-logic object schema to interoperate among multiple sources via the KIF knowledge interchange logic, but it is unclear how their logic-based transformation system scales when changes occur in the number of the sources or the content of individual sources.

2.3 Client-Server File Systems

In client-server distributed file systems such as Andrew [14] and the NFS-oriented commercial offerings, a file or a collection of files is the unit of storage allocation and has unique home on some server. Architecturally, unlike the previous two types of systems, client-server file systems implement the strategy of "moving the data to the query". Upon a request, relatively small, fixed size blocks of the file are brought to a client. Blocks are cached on a client site until they are no longer needed. Most file systems implement a hard-coded cache manager, typically utilizing a least-recent-used (LRU) eviction strategy.

In contrast, DIOM processes a distributed query using either the mode of moving the query to the data, or the mode of moving the data to the query (similar to client-server file systems in this case), or a combination of both modes, depending on the capabilities of the data sources involved in the query (see Section 4.7.1 for detail).

2.4 Related Query Optimization Research in Distributed Systems

The problem of distributed query optimization has been extensively studied in the literature, and many solutions have been proposed. Some solutions perform a rather exhaustive enumeration of query plans, hence do not scale well [1, 3, 9, 13, 22, 31, 34, 35, 38, 47]. Especially for the Internet-scale application scenarios with large number of data sources, these schemes are too expensive. Some other solutions reduce the search space using techniques like simulated annealing, random probes, or other heuristics [10, 18, 17, 30, 33, 41, 45, 44]. While these approaches may generate efficient plans in the intended cases, they do not have good performance guarantees in terms of the quality of the plans generated. Many of these techniques may generate plans that is arbitrarily far from the optimized one, or even fail to generate a feasible plan, when the user query does have a feasible plan.

Other interesting solutions proposed so far are those [15, 37] which use specific cost models and clever techniques to produce optimized join orders efficiently. However, it is unclear how such solutions, which rely heavily on sophisticated statistics about content of each remote data source, can be applied effectively to situations in an open environment.

3 Background and Terminology

3.1 An Overview of the DIOM system

The DIOM [26] system has two-tier architecture and offers services at both the mediator tier and the wrapper tier (see Figure 1). Mediators are software modules that handle application-specific service requests [25]. One of the main tasks of the mediator sub-system is to utilize the metadata obtained from both information consumers (i.e., user profiles) and information producers (i.e., source capability descriptions) for efficient processing of distributed queries [24].

Wrappers are software modules that need to be built around the external Information sources in order to make them accessible from the network of DIOM mediators. Each wrapper serves one individual information source. The main task of a wrapper is to control and facilitate external access to the information source by utilizing the wrapper functions and the local metadata maintained in the DIOM implementation repository. Services provided by a wrapper include: (1) translating a subquery in consumer's query expression into an information producer's query language expression, (2) submitting the translated subquery to the target information source, and (3) packaging the result of a subquery obtained from the source in terms of the objects understandable by the corresponding mediator. Building a wrapper around an external information system turns the remote System into a cooperative information agent.

The information sources in DIOM 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 or XML files, bibliographical record files, other text-based records, or *non-structured*: 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 consent from the mediators. If, however, an information source makes a change in its export schema (so called the source content and capability description in DIOM), including logical structure, naming, or semantic constraints, then either it notifies the DIOM server or the DIOM wrapper periodically refreshes the metadata maintained at the DIOM server (and the corresponding DIOM implementation repository) according to the state of the export schema.

3.2 DIOM Query Processor

In order to meet our demands for site autonomy and distributed computation, we take a new approach to query optimization and query processing. In this section we briefly describe the mechanisms by which DIOM handles query parsing (initial pre-processing), query optimization and query execution.

Figure 1 presents a sketch of how a query is processed inside of the DIOM system. The main task of the query mediation manager is to coordinate the communication and distribution of the processing of information consumer's query requests among the network of mediators and wrappers [26]. The general procedure of distributed query processing in DIOM primarily consists of the following steps: query compilation, query parallelization, site selection and execution, and query result assembly at the mediator tier, and subquery translation, subquery execution, and local result packaging at the wrapper tier.

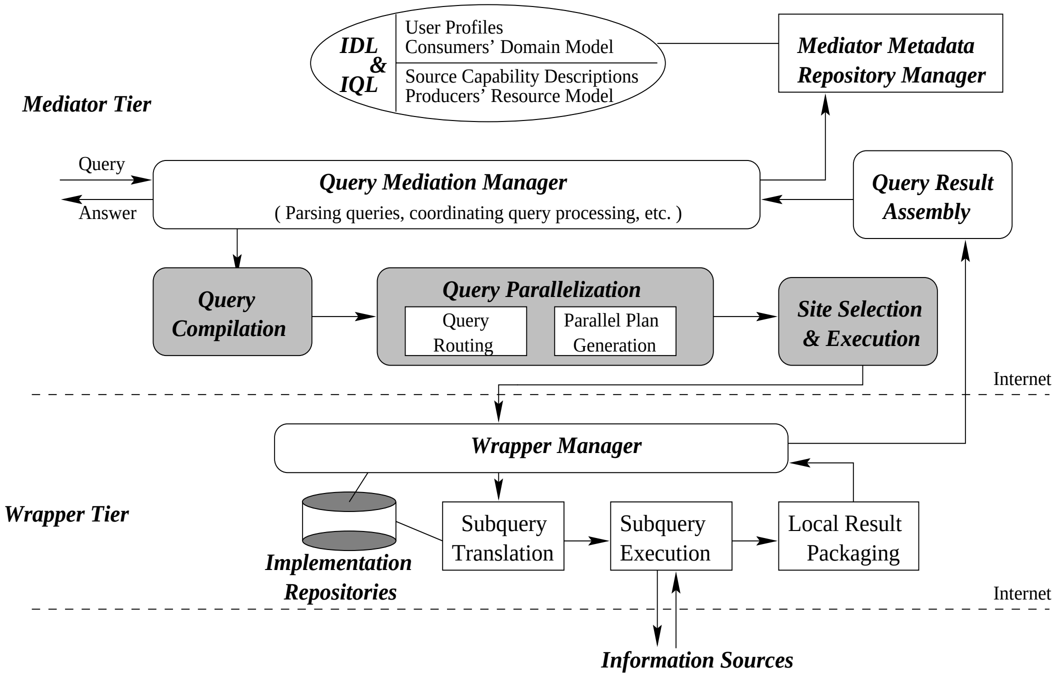


Figure 1: A Sketch of the DIOM distributed query scheduling framework

Query parsing is the initial query processing step, handled by the query mediation manager. We assume the following model for query parsing and initial query processing. A query is delivered to DIOM by a user at some site. This site becomes the home site for the query, responsible for both parsing the query and delivering the answer to the user. In order to accept a query that is posed on the fly, the parser must know that there exists some source who is capable of handling the query. Thus, the query processor at the parsing site issues the following system service request `who-is-capable(C)` to the DIOM metadata manager for each class C referenced in the query. The metadata manager maintains a set of source capability descriptions for each source that is accessible to the DIOM server. Upon a non-empty response from the metadata manager, the parser generates a query parse tree to be passed to the distributed query scheduling algorithm for query optimization and execution.

Query compilation is the first step of distributed query scheduling algorithm. This step takes a (unordered) query parse tree as its input and produces an ordered binary query plan tree by completely disregarding the distribution of the data. Thus, this step can use any optimization method proposed for single-site database management systems that produces a binary query operator tree.

Query parallelization follows the query compilation step. It consists of two tasks: *query routing* and *parallel query plan generation.* The main task of query routing is to select relevant information sources from available ones for answering the given query. This task is performed by mapping the consumers' domain model terminology to the information producers' resource model terminology, by eliminating null queries, which return empty results, and by transforming ambiguous queries into semantic-clean queries [24]. The consumers' user query profiles and producers' source capability descriptions play an important role in establishing the interconnection between a consumer's query request and the relevant information sources. The query routing step takes the binary query operator tree as input and produces an augmented binary query operator tree where the list of relevant information sources discovered is placed at the root of the binary query operator tree as the target of the query. The main task of the parallel query plan generation component is to determine the degree of intra-operator parallelism used throughout the query plan tree and introduces the mechanisms (union collector nodes), to be discussed in Section 3.5, by which DIOM controls the parallel execution of the given query.

The next step is *site selection and execution*. This is the third and the last phase of the distributed query scheduling service. This step takes a parallelized (non-binary) query plan tree, distributes the tree nodes among various DIOM accessible sites and executes it. The main objective of this phase is to generate a distributed query execution plan that will minimize the overall response time and reduce the total query processing cost. The actual mechanisms by which the query operator nodes are distributed will be described later in Section 4.

We refer to the query compilation, query parallelization, and site selection and execution as three phases of the DIOM distributed query scheduling service. We will discuss each phase in detail in Section 4.

Once an optimized query execution plan is generated, its execution will be performed in cooperation with the *Subquery Translation* and *Subquery Execution* modules. The translation process basically converts each subquery expressed in the mediator interface query language into the corresponding information producer's query language expression, and adds the necessary join conditions required by the information source system. For each subquery, the subquery execution module will invoke the corresponding wrapper function to execute the subquery. The issue of subquery translation and subquery execution is beyond the scope of this paper and is covered in detail in [20, 26].

The results returned by each subquery will be first packaged through the local result packaging module at the corresponding wrapper and then sent to the query result assembly step at the mediator tier to combine with other subquery results, before delivering the final answer to the user. The semantic attachment operations and the consumers' query profiles are the main mechanisms that we use for resolving semantics heterogeneity implied in the subquery results. The specifics of query result assembly are also beyond the scope of this paper.

3.3 The Running Application Scenario: Airline Reservation Example

In this section we introduce an application scenario taken from an Airline Reservation application domain to illustrate the three phases of the DIOM distributed query scheduling service. This application scenario will also be used as the running example throughout the paper.

An example query in this application domain is a query asking for reservation information about all customers who flew to Europe with Canadian Airlines in 1997:

```
SELECT *
FROM   Customer, Flight, Ticket Order
WHERE  Flight->destination CONTAINS 'Europe' AND
       Flight->date BETWEEN '01-JAN-97' AND '31-DEC-97' AND
       Flight->airline = 'Canadian Airlines International'
```

Now we use this example query to illustrate the three-phase optimization process.

The first optimization phase is query compilation. This phase *compiles* the query string into a locally optimized query plan without considering distribution of data. By local optimized we mean that the plan is generated based on the assumption that all data involved in the query is available at the same site as the query processing manager. Figure 2(a) shows the query parse tree. Figure 2(b) shows the binary query operator tree produced by this first optimization pass, where the most selective operand `Flight` is placed as the left most leaf node. The main tactics used in this initial phase are those that are commonly used in commercial relational DBMS products. Most of the tactics are also covered in many of the database textbooks (e.g., [46, 32]).

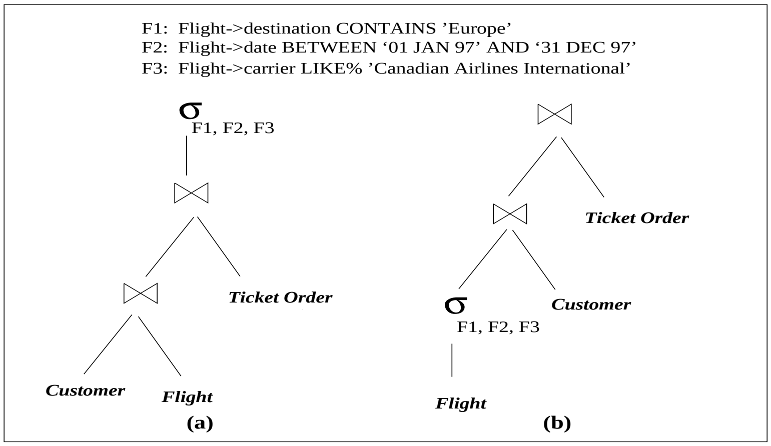

F1: Flight->destination CONTAINS 'Europe'
F2: Flight->date BETWEEN ‘01 JAN 97’ AND ‘31 DEC 97’
F3: Flight->carrier LIKE% 'Canadian Airlines International'
σ F1, F2, F3
⋈
⋈
Ticket Order
Customer
Flight
(a)
⋈
⋈
Ticket Order
σ F1, F2, F3
Customer
Flight
(b)

Figure 2: Query Compilation Step

The second optimization phase is query parallelization. This phase determines the degree of parallelism used throughout the distributed processing of the query plan tree generated from the first pass. It consists of two steps: query routing and parallel query plan generation. To answer a query in *Airline Reservation* domain, the potential number of available data sources may be huge. There may be many airlines and travel agencies accessible to the DIOM server. The main task of the query routing step is to discover and select the data sources that can actually contribute to the answer of the query. Since query routing algorithms are not the theme of this paper, for simplicity in this example we assume that the DIOM query router identifies the following three information sources to answer the given query:

- `dr90003` information source, accessible to DIOM as *Europa Travel* travel agency on-line reservation system, contains objects of type *Customer*, *Flight*, *Ticket Order*;
- `dr90001` information source, accessible to DIOM as *Canadian Airlines Flight Schedule* on-line database, contains objects of type *Flight*;
- `dr89904` information source, accessible to DIOM as *CA connection* – Canadian Airlines online reservation system, contains objects of type *Customer* and *Flight*.

As a result of query routing, the query tree is augmented by inserting a new target node as the root of the tree where these three information sources are explicitly identified (see Figure 3(a)).

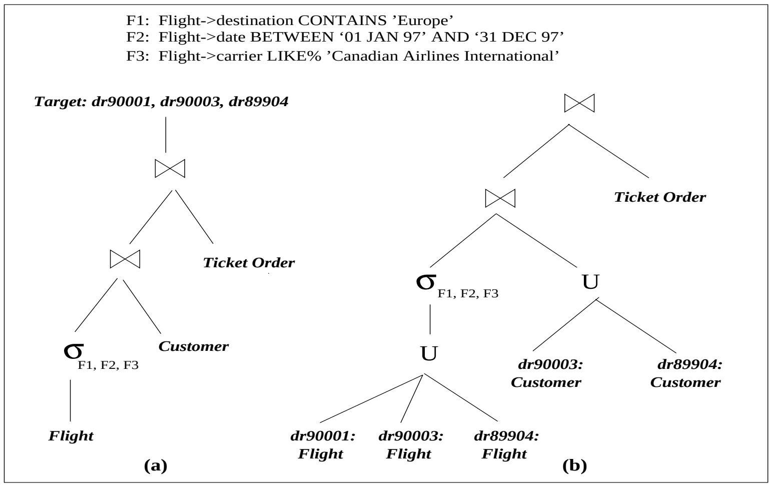

F1: Flight->destination CONTAINS 'Europe'
F2: Flight->date BETWEEN '01 JAN 97' AND '31 DEC 97'
F3: Flight->carrier LIKE% 'Canadian Airlines International'
Target: dr90001, dr90003, dr89904
⋈
⋈
Ticket Order
σ F1, F2, F3
Customer
Flight
(a)
⋈
⋈
Ticket Order
σ F1, F2, F3
U
U
dr90003: Customer
dr89904: Customer
dr90001: Flight
dr90003: Flight
dr89904: Flight
(b)

Figure 3: Query Parallization Step

The parallel query plan generation module incorporates the query routing result by inserting union collector nodes throughout the plan as shown in Figure 3(b). It also optimizes the order of the query operators to be executed as shown in Figure 4. The mechanism used for inserting union-collector nodes and the mechanism used for optimizing the order of join and union collector will be discussed later in Section 4.

The objective of the third optimization phase – site selection and execution is to select a site for each of the query operators, in which it can be executed most efficiently. We formally describe this problem in Section 4. In Section 5 we report our experimental design and development of the *DIOM Query Scheduling Utility* that implements the query scheduling algorithm proposed in Section 4.

So far we have briefly described the main steps of query processing in DIOM and introduced the running application scenario to be used throughout the rest of the paper. The purpose of this section is to present a general picture, in which our work takes place. The remaining sections present a concrete solution for the query scheduling components and its implementation in the context of this general picture.

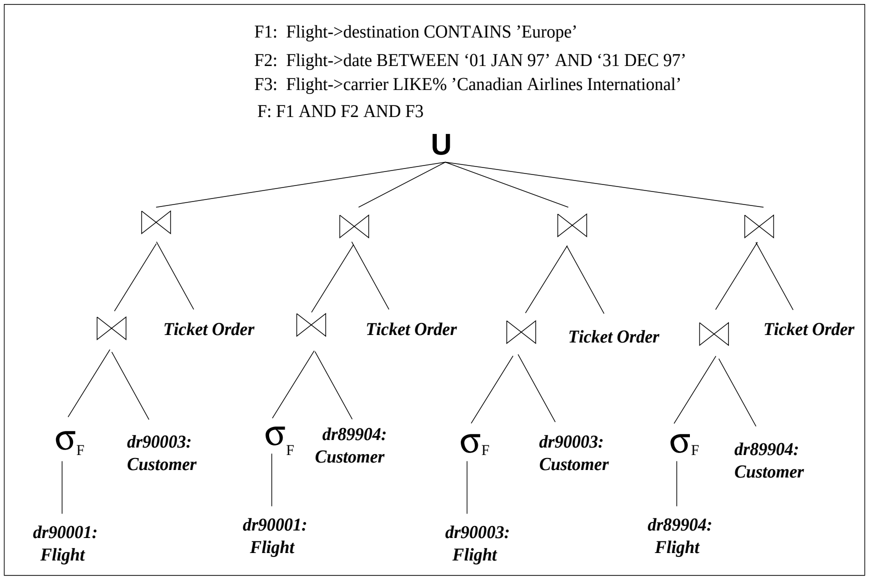


Figure 4: The optimized order of the query operators produced in the query parallelization phase

4 Distributed Query Scheduling Service

A key task of a distributed query scheduling service is query optimization. In addition to the standard problems associated with query optimization, the DIOM query scheduler must deal with some additional problems and restrictions caused by the distribution and full autonomy of data sources, including: (a) deciding on moving query to the data or moving data to the query, (b) load/capacity heterogeneity between computers, (c) type/method heterogeneity (extension library inconsistency), (d) harnessing the parallelism made possible by data distribution, (f) permitting maximal distribution and autonomy during assignment of work to sites. We discuss our solution to these problems in the following sections.

4.1 Global Optimization Criteria

Traditional 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 information consumers, 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 the following three main factors: communication cost, local processing cost, and total response time cost. They may be combined additively into a generic goal formula shown in Equation 1.

$$Cost = a_{cc} \cdot C + a_{lqp} \cdot L + a_{rt} \cdot R = A^T \cdot \begin{pmatrix} C \\ L \\ R \end{pmatrix}, \text{ where} \tag{1}$$

- 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.

Note that the total local processing cost L is measured by the sum of all local processing costs, namely SUM($\{L_1, ..., L_i, ..., L_n\}$) where each L_i $(i = 1, ..., n)$ denotes the local processing cost at one specific data source, whereas the total response time R is measured by the maximum communication cost and the maximum local processing cost, namely MAX($\{C_1, ..., C_i, ..., C_n\}$) + MAX($\{L_1, ..., L_i, ..., L_n\}$). C_i $(i = 1, ..., n)$ denotes the cost of connecting to a specific data source and the total communication cost C is measured by SUM($\{C_1, ..., C_i, ..., C_n\}$).

The coefficients associated with each of the three cost components are the indicators of the desired optimization profile. They can be controlled by the user of the system by setting the profile via the components of vector A^T. For example, if the user's primary concern in optimizing a 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.

4.2 Query Scheduling Strategy

The DIOM query scheduling service must operate within the constraints described previously. We argue that the straightforward use of traditional approaches to query processing is not adequate for large-scale and extensible distributed information systems. We present our own query processing strategy that addresses this problem and describes how our approach uses the standard optimizer technology as a basis for subsequent refinement whenever adequate.

Traditional cost-based query optimizers, including nearly all optimizers found in commercial products, have been based on resource-usage models and exponential-complexity dynamic programming search algorithms similar to those developed in System R [39]. Extensions of these traditional optimizers to handle distributed database systems, as in R* [38, 29], is straightforward and produces optimized plans. However this approach has a key disadvantage. The exponential complexity of the search space makes the use of such optimizers impractical in very large distributed systems.

The strategy that the DIOM query scheduling service promotes is to consider conventional optimization factors (such as join order) separately from distributed system factors (such as data distribution and execution location). Hence, the DIOM query scheduling algorithm is a multi-pass process. We now describe our algorithm.

4.3 Three-Phase Optimization

The basic idea of the three-phase optimization approach is to generate query plans at compile time that completely disregard the current location of the data and the type of structures of the actual data sources (structured, semi-structured, or unstructured), and make decisions with respect to the degree of parallel execution, data movement, and query execution sites at execution time. Although the initial use of "single-site" optimization may often generate sub-optimal plans, as pointed out in [29, 19], we believe that other strategies require an arbitrarily large amount of current global knowledge regarding integrated schema and data.

In the rest of the paper the term *query plan* and *query tree* are used interchangeably. As with many query processing systems, DIOM implements query execution plans as trees of nodes corresponding to the set of query processing operations (scans, joins, etc.). Object-based dataflow occurs along the branches. Such trees can be trivially decomposed into subtrees, each of which describes the execution of a portion of the original query (subquery). These trees can be encapsulated for transmission between hosts.

A sketch of the proposed three-phase query scheduling algorithm is shown in Figure 5.

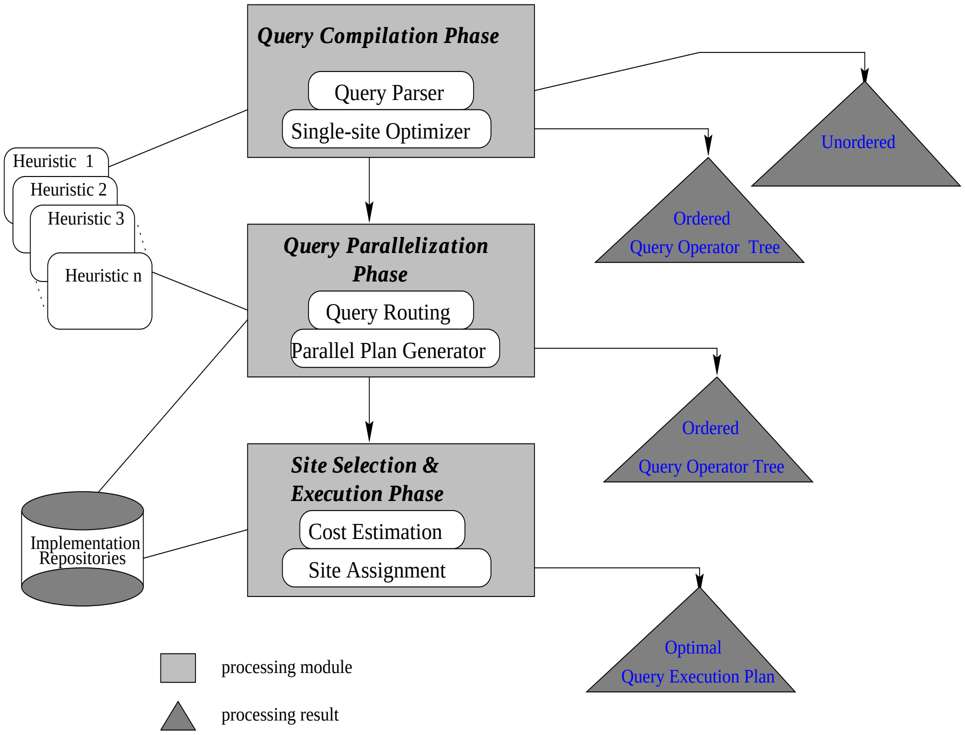


Figure 5: Three-phase query scheduling framework

Briefly, the DIOM query scheduler starts by conducting the first optimization pass that *compiles* the query string into an initial query plan without considering distribution. Immediately before execution, the scheduler *parallelizes* the result of this first pass. At the execution time, the DIOM scheduler *selects sites* on which to run each query processing operator (scans, joins, etc.) in a distributed, top-down manner. Sites make decisions to accept or reject responsibility for handling portions of a query. As sites accept responsibility for handling portions of a query, they also assume responsibility for selecting sites to run the portions of their own subqueries.

A slightly more detailed sketch of the algorithm is presented in the next three sections, each focuses on one phase of the query scheduling algorithm.

4.4 Query Compilation Phase

As shown in Figure 5, this phase takes a unordered query parse tree as its input and produces an ordered binary query operator tree without considering distribution of data. The parse tree represents the result of compiling the query string into an initial query plan tree. The ordered binary query operator tree is constructed as a locally optimized query plan by running a local (single-site) optimizer over the query, assuming in its cost calculations that all data is local to the home site of the query. At this point,

the query execution plan tree is simply a standard binary operator tree. This phase determines such items as locally optimized join order and the application of well-known tactics such as performing selection before join, performing *Cartesian product* last. For clarity, we will not consider such issues as the indexes or sorting of results in this paper.

We summarize the common tactics used in the standard optimizer technology in Heuristic 1.

Heuristic 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 [7, 46];*
- *(b) considering only such join orderings that do not result in Cartesian product between relations [39];*
- *(c) performing the joins whose estimated result is smaller before the joins that are expected to have larger intermediate result [3, 7].*

The heuristic-based optimization involves the use of query rewriting rules to generate the optimized query expression that is equivalent to the original query. An optimize query expression here refers to the one that minimizes the response time or the overall processing cost. The main manipulating factors that affect cost and response time of a query execution plan is the order of the query operators and the site assignment of the binary operators of the query. In DIOM the order of query operators is first considered in the query compilation pass and then refined in the query parallelization pass. The site selection for each query operator is determined in the third optimization pass.

It is worth noting that the local optimizer builds query execution plans using the metadata provided by the system. Inaccurate class statistics (e.g., the number of objects per class, overall size, selectivity information) will have exactly the same effect as they would on a non-distributed optimization – the plan produced may deviate from the optimized plan. In this case, however, the plan is still usable. Furthermore, this first phase can consist of any optimization method proposed for single-site database management systems, which produces (or can be modified to produce) a binary query operator tree. Recent research demonstrates that bushy query plan trees permit substantial gains in performance in both serial (e.g., [16]) and parallel (e.g., [11]) database systems. We intend to experiment with different optimization strategies proposed to determine which method provides the best results in the DIOM system.

4.5 Query Parallelization Phase

This phase takes a binary query operator tree as input and produces a non-binary query plan tree. The main objective of this second optimization pass is to determine the degree of intra-operator parallelism required for various subtrees of the plan tree and insert union collector nodes throughout the plan.

A union collector node, denoted by $\cup_{collector}(R_1, ..., R_n)$, is defined as an operator over n intermediate subquery results $R_1, ..., R_n$, each returned from one of the collector's subtrees. The result of a union collector $\cup_{collector}(R_1, ..., R_n)$ is equivalent to the union of $R_1, ..., R_n$, assuming $R_1, ..., R_n$ are union-compatible. In other words the following equation holds:

$$\cup_{collector}(R_1, ..., R_n) = \cup(... \cup (\cup(R_1, R_2), R_3) ..., R_n) \qquad (2)$$

This equation says that a n-way union collector can be rewritten into a n-way union. Thus, in the rest of the paper we use union collector and union interchangeably.

A query parallelization pass consists of two steps: query routing and parallel query plan generation.

4.5.1 Query Routing

For any user query posed on the fly, the query routing module is used to discover the number of data sources that are capable of answering the query by sending a message `who-is-capable(Q)` (`Q` is the original user query) to the DIOM metadata manager. The result of the query routing process will be used by the parallel query plan generation module to determine the number of union collector nodes to be inserted throughout the query plan tree.

4.5.2 Parallel Query Plan Generation

There are two main tasks for the parallel query plan generation module. First, it determines where and what union collector nodes need to be inserted throughout the plan. The mechanism for inserting the union collector nodes is to associate the list of information sources discovered in the query router step to each leaf node of the query plan tree, and invoke the wrapper function `discover-source-structure(C)` for each object class `C` referenced in the original user query to identify the actual number of source classes that correspond to `C`. Union collector nodes serve two purposes. First, they coordinate the execution of the portion of the query plan tree that lies immediately below them. Second, they consolidate the results produced by the query plan tree nodes immediately below them. For example, a union collector node may have several child scan nodes. The results from those scans are fed into the union collector node and piped up to the next level in the query plan tree as a single stream. The method by which union collector nodes are generated will be described later. For now, it suffices to say that this phase determines the degree of parallelism used throughout the plan tree and introduces the mechanisms (union collector nodes) by which DIOM controls parallel execution.

Second, the query parallelization phase refines and finalizes the optimization (efficiency) plan for the order of the query operators to be executed during the parallel query plan generation process. The decision to optimize the order of the query operators involves the use of a heuristic-based approach to refine the join order produced from the first optimization phase and determine the order of join nodes and union collector nodes. We describe the collection of heuristics we use in the next subsection.

4.5.3 Heuristics on Union and Join Ordering

A commonly accepted query optimization principle is to perform the most expensive and least effective operators last in the query. In other words, the goal of the query optimization is to reduce the scope of the search as early as possible. Thus we place 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 the Cartesian product between two collections of objects because the output size of this operator is the product of sizes of the operands. Join operator can be quite costly when it has a non-equal join predicate. Another one is the binary union operator. It results in a collection of objects that is at least the size of the largest of the operands. In a word all three binary operators are expensive.

By properly ordering the operands or the execution sequence of these operators, we may obtain performance improvement. The following heuristic is derived based on the observation that both union and Cartesian product should always be performed at the site where the result is expected because the communication cost of transferring the result is greater than the communication cost of transferring any one of the operands (see section 4.7.1 on page 20). However, the union is still a better reducer than Cartesian product because the processing cost of a union operator is less than a sum of the processing cost of each operand, i.e., $l(\cup(Q_i, Q_j)) \leq l(Q_i) + l(Q_j)$, whereas the processing cost of a Cartesian product is equal to the product of the size of two operands, i.e., $l(\times(Q_i, Q_j)) = l(Q_i) \times l(Q_j)$, (see page 23 for definition of $l(Q)$). Thus it would be more beneficial to perform Cartesian product after the union, except for the cases when one of the operands is empty.

Heuristic 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 a Cartesian product of* flight *and* order*, i.e., find all the possible flight-order combinations* Assume that three data sources have been selected to answer this query, Two of which contain *Flight* objects, and the third contains *Order* objects. Let $flight_1$ denote $Flight@Source_1$, $flight_1$ denote $Flight@Source_1$ and *order* denote $Order@Source_3$. Then the result of the query parallelization can be expressed as follows:

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

Based on Heuristic 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 the query parallelization phase 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 performing 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. The first factor is that the join itself has to be a good reducer (Heuristic 3).

Heuristic 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 the join result is smaller than any of its inputs.

Consider an example query: *select flight and ticket order information about all ticket orders that were made on the same day as the date on which the flights were booked to depart.* Assume the same three sources have been selected, two with *flight* objects, and one with *order* objects. The query plan produced by the query parallelization for this query can be expressed as follows:

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

It is beneficial to rewrite this query plan as follows if the expected size (estimation) of the join result is smaller than both of the size of $\cup(flight_1, flight_2)$ and the size of *order*.

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

because the join operator may substantially reduce the size of the query result. Otherwise, if the estimated size of the join result is large, then no change on the order of join and union is necessary and the old expression is kept.

The second factor used to estimate the required performance gain when deciding the order of join and union operators is to make sure that the interchange, if done, may result in fewer site accesses (see Heuristic 3). In other words, if the interchange of join and union does not lead to an increase in the number of inter-site joins, then such interchange is performed, otherwise, the order of operations is kept unchanged.

Heuristic 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.

This heuristic is basically saying that it is beneficial to interchange the order of join and union if the number of inter-site joins are not increased. Put differently, there are more than 50% of joins resulting from the interchange are the intra-site joins in the sense that both operands of each such join are located at the same site.

Consider the same query example but assume now that one of the data sources of the *Flight* objects is the same as the source of the *Order* objects (say $Source_1$). Then by applying Heuristic 4 to the query plan produced at the early stage of the query parallelization: $\bowtie (\cup(flight_1, flight_2), order_1))$, we have

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

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

Note, however, that Heuristic 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.

Heuristic 3 and Heuristic 4 are alternative rewriting rules. Either one of them will trigger the transformation of the query expression. If one of them triggers the transformation that moves a particular join operator down, the other heuristic does not need to be applied.

Recall the example query given in Section 2.3 and Figure 3(b) where $n = 3$ (three *Flight* sources) and $m = 2$ (two *Customer* sources). In the situation where there is not enough statistical information about the selectivity of join attributes, it will be difficult to see if the expected result size of the join in Figure 3(b) is smaller than any of the join operands. Thus Heuristic 3 can not be applied. To optimize the order of join and union, we can apply Heuristic 4. The query tree shown in Figure 4 is the result of applying Heuristic 4 to the query tree in Figure 3(b). More conceretly, based on the domain knowledge that each source has its own customer set, the join between *Flight* at source `dr90001` and *Customer* at source `dr90003` as well as the join between *Flight* at source `dr90001` and *Customer* at source `dr89004` are duplicates and thus can be removed. Hence, the parallelized query plan shown in Figure 4 has two inter-site joins and two intra-site joins.

4.5.4 Discussion

We have described the tasks of query parallelization and the mechanisms used for accomplishing these tasks. To elaborate on the key ideas used in the query parallelization phase, in this section we walk through the query parallelization process using single-class scans and two-class joins.

The most basic query processing operation is the single-variable query (single-class scan). However, in DIOM a simple scan over a single class may become very complex because the data may distributed over several sites and the plan may be executed using several parallel threads. DIOM uses union-collector nodes to control both the flow of results and thread execution. Typically, during the query parallelization phase, the query scheduler turns the single scan node into a collection of parallel scans whose execution is coordinated by a union collector node. In general, a scan on n different data sources will have n parallel threads of execution of the scan, one per data source. Hence, the query plan tree changes from a single node to a n-way tree – one union-collector node with n child scan nodes. All of the scan results must be collected on the site on which the union-collector node resides. However the location at which DIOM performs the actual source-class scans may or may not be that union collector site. Furthermore, the union collector node must make the following decision for each scan thread: the union collector can either fetch (import) the corresponding data source to the site where the union collector locates and executing the scan (and filtering) locally (moving the data to the query), or it can execute scan on the site currently storing the data (moving the query to the data). The decision in the current DIOM implementation primarily depends on the type of data sources where the data is located. Typically, for semi-structured data sources where no search or filtering capability is provided at the source (such as Web pages), the policy of moving data to query is applied; and for structured data sources such as relational databases, the policy of moving query to data is used, since most of the relational DBMS products have very powerful search and filtering capability and API. The basic join operation is the simple two-class join. DIOM implements both nested-loop and hashed join methods. In both cases, the result of the query compilation phase produces a standard two-way join plan: a join node with two single-class scan nodes as children. The query parallelization phase inserts the union-collector nodes to reflect the number of relevant data sources into which each class is divided, producing a multi-way query tree. This tree will look different for the two join methods, as described below.

- **Nested-loop Join.** A nested-loop join can be naturally decomposed into n_{outer} parallel joins, where n_{outer} is the number of sources in the outer join class. This essentially turns into n_{outer} separate scans of the inner join class. The union-collector node for the inner scans is merged with the nested-loop query plan node and performs the join with the objects in each of the outer source-class scan nodes. Hence there will be n_{outer} results at the union-collector node of the outer join class, which are in turn collected into a final result at the home site of the user query. Two types of implementation choices are considered: The first one is to implement nested-loop join using a single parallel inner scan (with a single union collector node), the unit of import would then be the entire inner scan result. The second choice is to replicate the parallel inner scans, the union-collector node of the outer join class can choose to import individual data sources of the inner class. Figure 4 presents an concrete implementation of this second choice.

- **Hash Join.** Since it is not common to assume that any given hashed access method will exist on all of the data sources, we assume that hash join executes as two separate stages: First, hashing the object classes into b buckets (using split object classes to route objects to the correct bucket site). For the purpose of discussion, let us assume $b = n_{outer}$. Then performing joins of corresponding buckets in parallel. This requires b collector nodes, which are again merged with the hash-join query plan node, for the results of the b subjoins.

This method of producing query plans for parallel two-way joins can be naturally extended for producing plans for parallel mutli-class joins.

4.6 The Third Optimization Phase – Site Selection and Execution

This third optimization phase takes a parallelized query plan tree, distributes the tree nodes among the various DIOM sites and execute it. The actual mechanism by which the query operator nodes are distributed will be described later. Conceptually, the criteria for selection of a site to execute a particular node depends on the type of operator node and the potential collection of distribution sites to be considered. The following site selection rules are used in the DIOM query scheduler.

- To handle a particular *scan node*, the initial selection of a site will be determined by the location of the required data objects, though other sites may be considered.

- To handle a particular *unary operator node* (such as selection σ, projection π)), the default selection of a site will be determined by the location of the scan node below it. Put differently, a unary operator node is distributed to the same site as the one assigned to the scan node associated with it. However, if the site where the data locates is a semi-structured or unstructured data source with littler search or filter capability, such as Web pages of many corporate or organization sites, then a DIOM server site will be chosen, which is normally the site where the respective wrapper of the data source locates. In this case, the policy of moving data to query will be chosen to replace the default policy of moving query to the data.

- To handle a particular join node, union node or other type of binary operator nodes, the possible sites to perform the join or the union can be either one of the operand sites (the site where one of its operands is located) or a DIOM server site where the intermediate results are recorded.

For a scan operator or a unary operator, the selection of a site is simple in concept. However, for a binary operator node such as join or union, the selection process becomes complicated when several sites are capable of handling the operator node. The key issue is to decide which of the site choices is the best (cheapest w.r.t. the overall processing cost and/or the response time) for each binary operator. In DIOM a cost estimation based approach is used to handle site selection task for binary operator nodes, which is the topic of the next section.

The site selection process also assumes that any site considered must be able to conform to a number of constraints. As discussed previously, these constraints include both load balance considerations, capacity and capability restrictions as well as data dependencies. If a constraint is not met, then the site refuses the subquery (subtree) and another site must be found. If no site will accept the subquery, then the original query must be aborted.

Since site-selection occurs in a top down fashion, execution can begin immediately after sites have been selected for the bottom-level operator nodes.

4.7 Cost-based Site Selection for Inter-site Queries

A truly distributed query is an inter-site query which require either inter-site joins or inter-site set operations such as unions, differences. The cost estimation of inter-site queries involves not only local processing cost but also data shipping and other communication cost. Given an inter-site query tree with its selection and projection pushed down and close to its leaf node, we need to consider the following factors in order to optimize site selection and execution plans:

- the hypothetical possibility to assign the binary operation to a site;
- the collection of all sites that are able to take the binary operation in terms of their source capability descriptions – such as the appropriate processing power, temporary disk space, as well as query and data definition compatibility;
- the capability of the server site where the client request is received and processed.

To simplify the discussion, we first consider the inter-site query trees that contain either a single union operator (Section 4.7.1) or a single join operator (Section 4.7.2). Then we discuss the site-selection cost functions for inter-site queries in general.

4.7.1 Site Selection for Single Union Inter-Site Queries

Consider the following example query asking for all customer names and addresses who live in Edmonton, Canada:

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

`!=` is a substring matching operator introduced in DIOM Interface query language (IQL) [26]. Let us assume that after the query is submitted to the query processor, and undergone query compilation and query parallelization phases, only two data sources are found, which are able to answer this query, and the original query is transformed into the parallelized query plan tree as shown in Figure 6(a), where Q_1 denotes `Scan:customer@site1` and Q_2 denotes `Scan:customer@site2`. Figure 6(b) is the final result of the query parallelization after applying Heuristic 1 to the query plan in Figure 6(a).

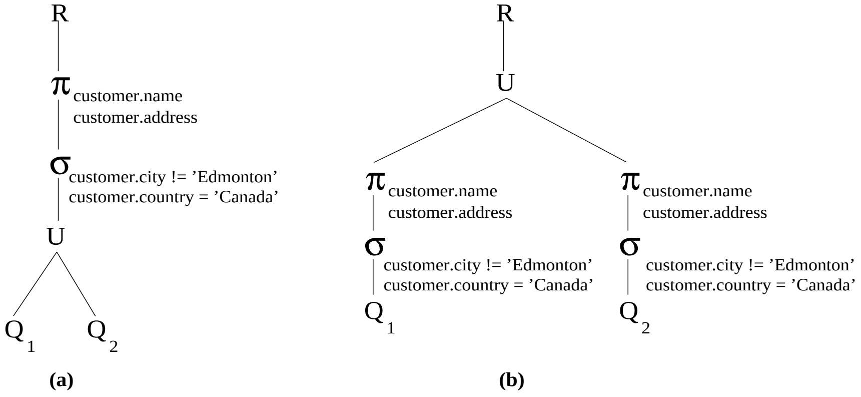

R
π customer.name customer.address
σ customer.city != ’Edmonton’ customer.country = ’Canada’
U
Q1
Q2
(a)
R
U
π customer.name customer.address
σ customer.city != ’Edmonton’ customer.country = ’Canada’
Q1
π customer.name customer.address
σ customer.city != ’Edmonton’ customer.country = ’Canada’
Q2
(b)

Figure 6: The query tree produced after query compilation and query parallelization

As mentioned earlier, information sources in DIOM may have heterogeneous representations for the same real world objects and their properties. 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 local result packaging and result assembly stage. Therefore, at query optimization stage we assume all subqueries are expressed uniformly in terms of the terminology used in the user query.

For the query tree shown in Figure 6(b), the site selection process will proceed in a top-down fashion. We start with the union operator node. In order to optimize the site selection plan for this union operator (i.e., the site where this union operator can be processed most efficiently), let us assume that the collection of sites involved in answering the query are three sites, namely two data source sites: site 1, site 2, and a result delivery site: site 3. Q_1 and Q_2 represent the *customer* subqueries on site 1 and site 2 respectively, and the result R of the query is expected at a remote third site. We also assume that there are no constraints on any of the hypothetical site to take the union. Then there are three possible site selection and execution plans for this query:

1. placing union at the site of Q_1 (see Figure 7(a)),

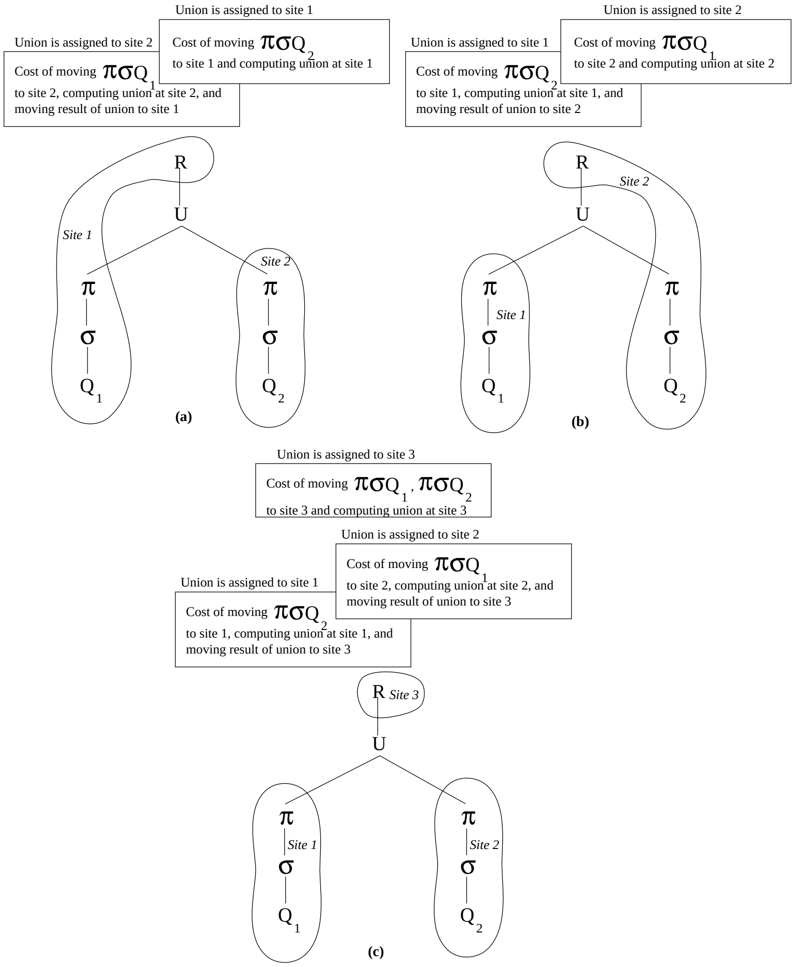

Union is assigned to site 2
Cost of moving πσQ1
to site 2, computing union at site 2, and
moving result of union to site 1
Union is assigned to site 1
Cost of moving πσQ2
to site 1 and computing union at site 1
R
U
Site 1
Site 2
π
σ
Q1
π
σ
Q2
(a)
Union is assigned to site 1
Cost of moving πσQ2
to site 1, computing union at site 1, and
moving result of union to site 2
Union is assigned to site 2
Cost of moving πσQ1
to site 2 and computing union at site 2
R
Site 2
U
π
Site 1
σ
Q1
π
σ
Q2
(b)
Union is assigned to site 3
Cost of moving πσQ1, πσQ2
to site 3 and computing union at site 3
Union is assigned to site 2
Cost of moving πσQ1
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 πσQ2
to site 1, computing union at site 1, and
moving result of union to site 3
R
Site 3
U
π
Site 1
σ
Q1
π
Site 2
σ
Q2
(c)

Figure 7: 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.

- 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 3 where result is expected;

2. placing union at the site of Q_2 (see Figure 7(b)),
   - materialize Q_1 at site 1, perform selection and projection there, and ship the result to the second site;
   - materialize Q_2 at site 2, perform selection and projection there;
   - perform union at the second site, ship the result to the site 3 where result is expected;

3. placing union at the result delivery site (see Figure 7(c)),
   - materialize Q_1 at site 1, perform selection and projection there, and ship the result to the site 3;

- materialize Q_2 at site 2, perform selection and projection there, and ship the result to the site 3;
- perform union at the site 3;

The cost-based site selection pass will decide which site the union operation should be assigned to, based on the costs associated with each of the possible decisions. The optimized site selection and execution plan would be the one that offers the least cost and the fastest response time. According to Equation 1, the formulae for deriving the component costs for the scenarios shown in Figure 7 are given in Equation 3.

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

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. When Q_i denotes a union operator, $l(\cup(Q_k, Q_j)) = \max(l(Q_k), l(Q_j))$;

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

Note that the response time cost is computed, according to the intended parallelization of the tasks, as a maximum of the the concurrent subplans, each executing at a different site. Also note that the formulae in Equation 3 provides a general means for computing the comprehensive cost of each of the possible query plans for the case of two-way inter-site union.

In the initial implementation of the DIOM distributed query scheduling service, we assume that all network connections have equivalent bandwidth and latency. This is not to say that all hosts must be on the same local network work or a set of completely homogeneous networks, but rather that the networks are all "fast" in the sense discussed in [29], such that network bandwidth does not overwhelmingly dominate query processing costs. More concretely, we make following assumptions concerning the inter-site communication and local on-site processing:

- 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, i.e., $\forall\ i, j\ cc_{ij} = cc$, and
- 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, i.e., $\forall\ i\ cl_{opi} = cl_{op}$.

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 dynamically accommodate the changing system parameters. However, when the actual cost parameters are not available, it is useful to make these assumptions.

Based on the assumption of equal unit communication and local processing cost, the rules given in the following Equation 4 can be derived from the cost formulae in Equation 3.

$$\begin{array}{lll} C_{11} \leq C_{12} & L_{11} = L_{12} = L & R_{11} \leq R_{12} \\ C_{22} \leq C_{21} & L_{22} = L_{21} = L & R_{22} \leq R_{21} \\ C_{33} \leq C_{31}, C_{32} & L_{33} = L_{31} = L_{32} = L & R_{33} \leq R_{31}, R_{32}, \end{array} \tag{4}$$

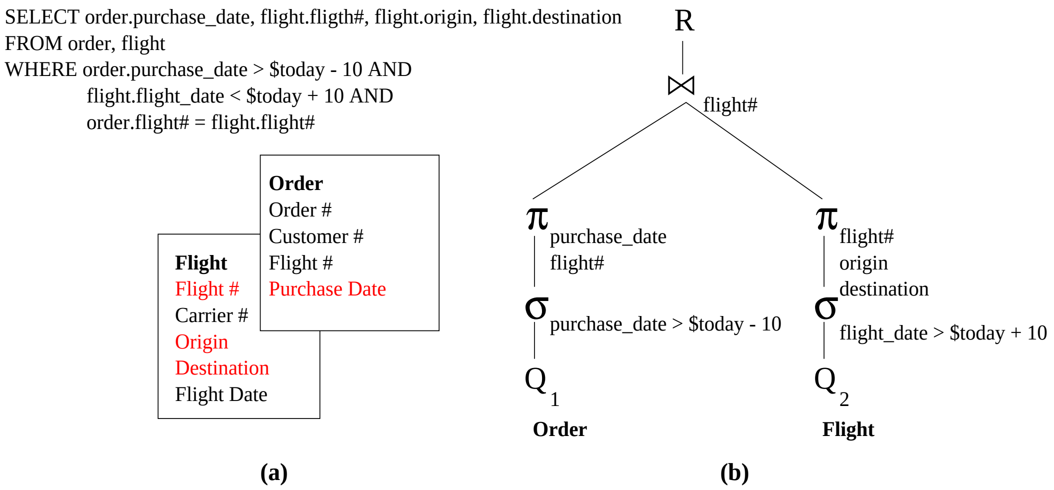
These rules are used to determine which of the three potential sites is the optimized site choice for a union operator. They amount to say that it is always beneficial if the union is assigned to the same site as the site where the result of the query is expected. We can prove that the best query plans for the scenarios of Figure 7(a), (b), and (c) are 11, 22, and 33 respectively. In other words, as a rule, for queries that involve a two-way union, the optimized site selection plan is to perform the union at the site where the result is expected. In DIOM, this observation is used as a heuristic for union operators in inter-site queries. We omit the formal proof here due to space limitation. Readers who are interested in further details may refer to the technical report [36].

Note that, however, the rules in Equation 4 may not be true if the assumptions of equal unit communication and equal local processing costs are broken. That is, there may be cases where assigning union operator to the result delivery site will not lead to the minimum cost [36].

4.7.2 Site Selection for Single Join Inter-site Queries

We have described the cost function formulae and the rules that optimize the site selection plans for single union inter-site queries. In this section we discuss the cost function formulae used in site selection for single join inter-site queries.

Consider a query: *find the date of purchase, flight number, origin, and destination for all flights booked no later than 10 days ago and flying within the next 10 days.* The query is expressed in Figure 8(a) Assume that two data sources are found after the query compilation and query parallelization passes. All `Order` objects are accessible at site 1 and all `Flight` objects are accessible at site 2. Figure 8(b) shows the parallelized query plan tree produced by the query parallelization phase. It contains a single join operator. Q_1 stands for `scan:Order@site1` and Q_2 denotes `scan:Flight@site2`.



```
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 8: The single join query example.

Similar to the single union case, we present the possible site distributions for processing an inter-join query (e.g., the example query) in Figure 9.

The formulae for computing the cost of single join inter-site queries is given in Euqation 5.

From the formulae in Equation 5 and the assumption that the unit communication cost and local processing cost are the same for all sites and all communication links in the network, we can easily prove that *if the result of a join has the size that is greater than the sum of the sizes of its operands, then the site selection plan that has the lowest cost is to place the join operator on the server site where the result of the client query is to be collected and returned.*

However, unlike the single union, we may not generalize this observation to the level of a universally applicable heuristic rule for selecting the best site for join site assignment, even when assuming the equal communication unit cost and local processing cost among sites. The difference is that the estimated result size of a 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. Indeed, if the result of a join has the size that is less than the sum of the sizes of its operand, then the outcome would be quite different [36]. The cost functions in Equation 5 will be used to make the decision about the best site selection plan. Due to the space limitation, we omit the concrete examples that illustrate the cost functions for inter-site joins. Readers who are interested in further detail may refer to [36].

$$
\begin{array}{l}
C_{11} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) \\
C_{12} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{21} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{12} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{22} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) \\
C_{31} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{21} \cdot l(Q_2) + cc_{13} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{32} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{12} \cdot l(Q_1) + cc_{23} \cdot l(\bowtie (Q_1, Q_2)) \\
C_{33} = C_{1\star}(Q_1) + C_{2\star}(Q_2) + cc_{13} \cdot l(Q_1) + cc_{23} \cdot l(Q_2) \\
\\
L_{ij} = L(Q_1) + L(Q_2) + cl_{\bowtie j} \cdot (l(Q_1) \cdot l(Q_2)),\ i = 1, 2, 3,\ j = 1, 2, 3 \\
\\
R_{1\star}(Q_1) = L(Q_1) + C_{1\star}(Q_1) \\
R_{2\star}(Q_2) = L(Q_2) + C_{2\star}(Q_2) \\
R_{11} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{12} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{21} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{21} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{12} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{22} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) \\
R_{31} = \max\{R_{1\star}(Q_1),\ R_{2\star}(Q_2) + cc_{21} \cdot l(Q_2)\} + cl_{\bowtie 1} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{13} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{32} = \max\{R_{1\star}(Q_1) + cc_{12} \cdot l(Q_1), R_{2\star}(Q_2)\} + cl_{\bowtie 2} \cdot (l(Q_1) \cdot l(Q_2)) + cc_{23} \cdot l(\bowtie (Q_1, Q_2)) \\
R_{33} = max\,\{R_{1\star}(Q_1) + cc_{13} \cdot l(Q_1),\ R_{2\star}(Q_2) + cc_{23} \cdot l(Q_2)\} + cl_{\bowtie 3} \cdot (l(Q_1) \cdot l(Q_2))
\end{array}
\tag{5}
$$

4.7.3 Site-Selection Cost Functions for Inter-site Queries

The cost functions for inter-site queries can be considered as recursive functions that start from the root of the parallelized query plan tree of a given query. The total cost of the query depends on the costs of obtaining the left and right subquery inputs. Similarly, the cost of the left (or right) subquery node is again computed based on the cost of obtaining its subsequent left and right children nodes. Thus the process of optimization may be organized as a downward traversal of the entire query tree by the cost optimizer. At each step of the traversal the optimizer analyses the leaf nodes of the current subtree. If one of the nodes is by itself a subtree, then the optimizer recursively invokes another optimization process, and passes the subtree to it as the optimization task. As soon as an invoked optimization process ends, the optimizer may assemble the result and pass it back, one level up, to the process that invoked it. Figure 10 presents an intuitive illustration of this recursive approach to formulate cost functions.

We abstract this recursive process of the cost functions in the following equations:

$$
\begin{array}{ll}
C(Q) & = C(Q_{left}) + C(Q_{right}) + \mathit{comm\_cost}(Q_{this}); \\
L(Q) & = L(Q_{left}) + L(Q_{right}) + \mathit{loc\_cost}(Q_{this}); \\
R(Q) & = \max[R(Q_{left}), R(Q_{right})] + \mathit{comm\_cost}(Q_{this}) + \mathit{loc\_cost}(Q_{this})
\end{array}
\tag{6}
$$

Due to the space limitation, we omit the examples here. Readers who are interested in further detail may refer to the technical report [36].

So far we have described the theoretical foundations and the design framework for distributed query scheduling service in DIOM. The next section is dedicated to the system design and implementation specifics of our proposed approach.

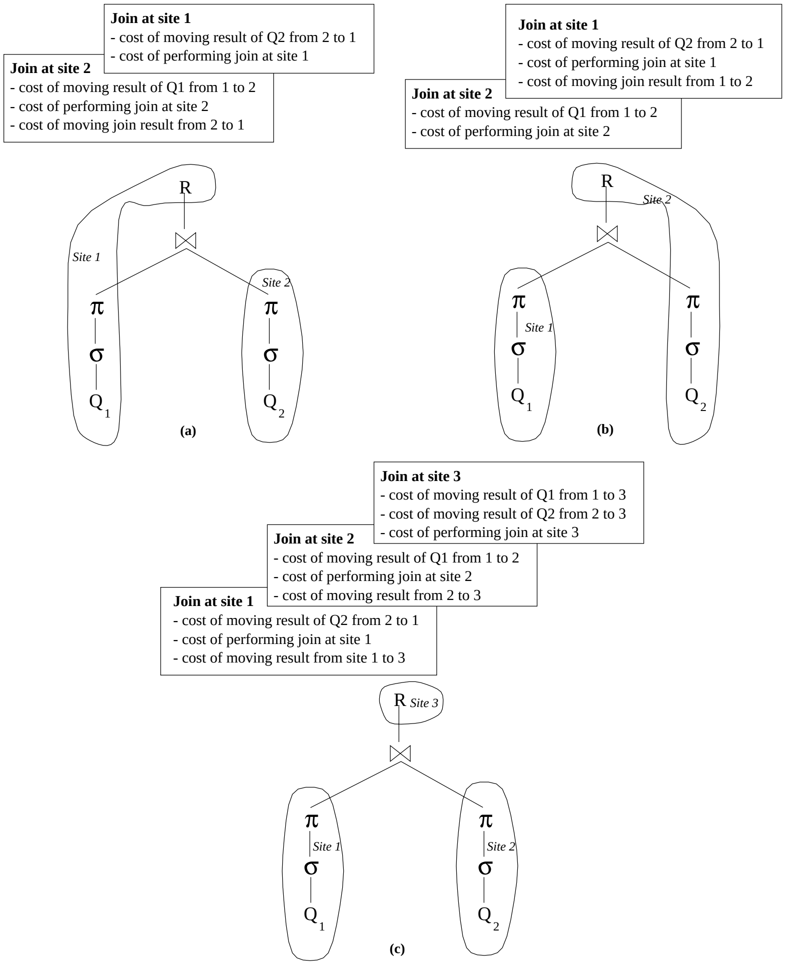

Join at site 1
- cost of moving result of Q2 from 2 to 1
- cost of performing join at site 1
Join at site 2
- cost of moving result of Q1 from 1 to 2
- cost of performing join at site 2
- cost of moving join result from 2 to 1
R
Site 1
Site 2
π
σ
Q1
π
σ
Q2
(a)
Join at site 1
- cost of moving result of 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
R
Site 2
Site 1
π
σ
Q1
π
σ
Q2
(b)
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
R Site 3
Site 1
Site 2
π
σ
Q1
π
σ
Q2
(c)

Figure 9: 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.

5 Prototype Design and Implementation

In this section we will discuss the issues involved in the design and implementation of the *DIOM Query Scheduling Utility*. The task of implementing the *DIOM Query Scheduling Utility* can be seen as a demonstration of viability of the ideas and principles presented in Section 3. On the other hand, the theoretical results described in Section 3 are served as the baselines for the software development covered in this section.

5.1 Implementation Architecture

To cover the functionality of the *Distributed Query Scheduling* software utility we use the data flow diagrams 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 11.

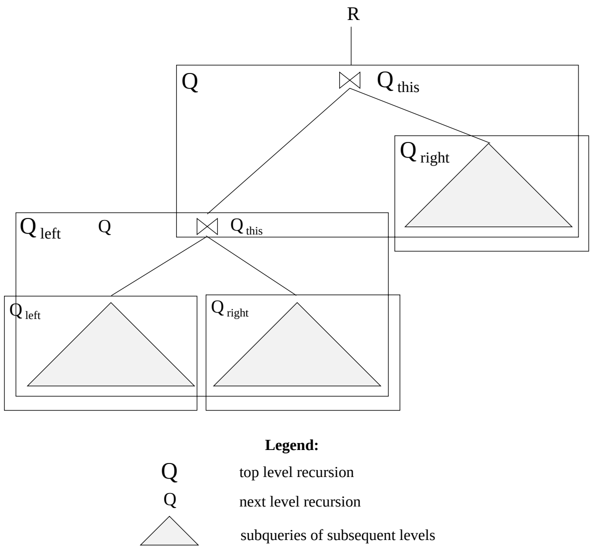


Figure 10: Cost Estimation: The Recursive Approach

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

- **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. The front-end of the *DIOM Query Scheduling Utility*, the query entry form, is shown in Figure 12.

- **Input/Output Processing Components (IOP)**
  The objects of this component are responsible for input and output. Some of the main 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.

- **Distributed Query Processing Components (DQP)**
  These are the main functional components in this application. The query manager, query router, decomposer (for query parallelization), heuristic-based and cost-based query processors for site selection and execution are the main objects in this group of components. They must be able to communicate with the query manager that coordinates their operation and ensures that the necessary objects are passed to and from the user interface components, as well as to and from each of the query processing components. Section 5.2 contains the detailed specification for each of the distributed query scheduling components.

- **Maintenance and Testing Components**
  The components that allow the user to test and diagnose all other functional components. This

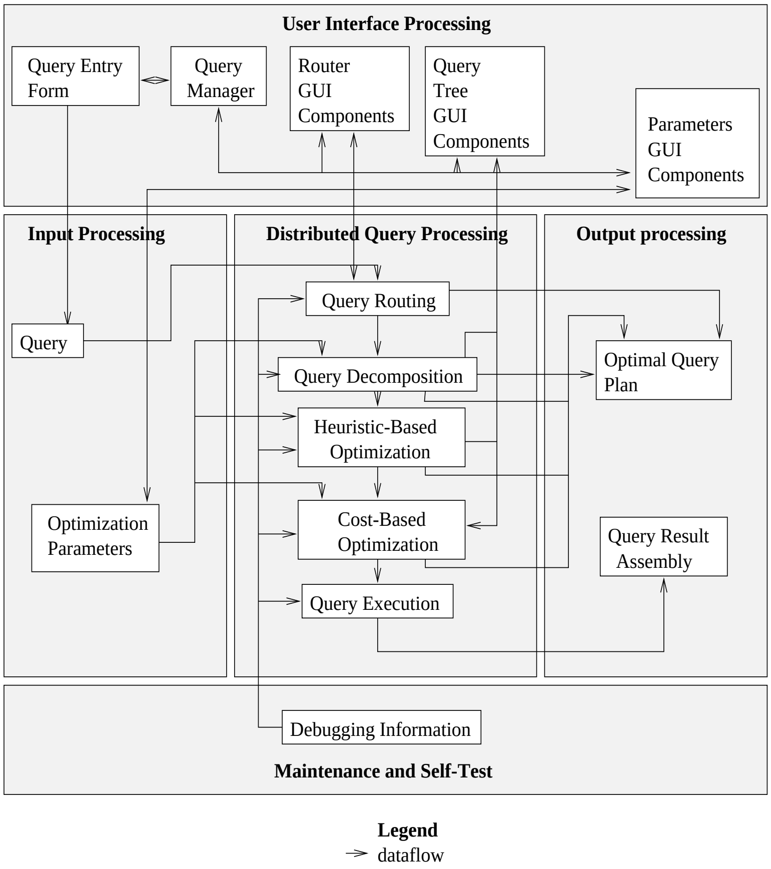


Figure 11: Architecture Flow Diagram of The *DIOM Query Scheduling Utility* Application.

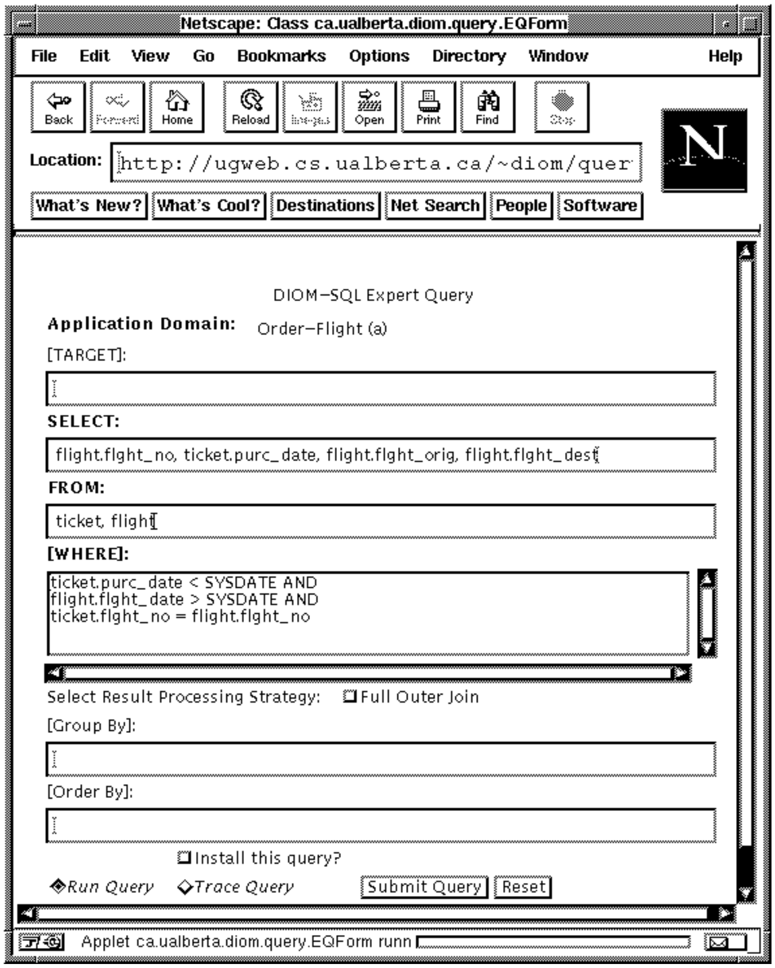
may include query processing scenarios that test for certain features in the processing components. See Section 5.3 for more detail.

Since the focus of this paper is on the design and implementation of the DIOM distributed query scheduling service, in the remaining sections, we concentrate more on the DQP components and omit any further discussion on the UIP and IOP. Readers who are interested in further detail may refer to [36].

5.2 DQP Components

5.2.1 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 the DQP application can impose on the system.



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, fligh

[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 12: Query Entry Form Screen

The 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* (see Figure 13). 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 the *Query Manager* implements the runnable interface, this method must be defined in the *Query Manager* class and is called whenever a 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 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 13).

< 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 removes 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.

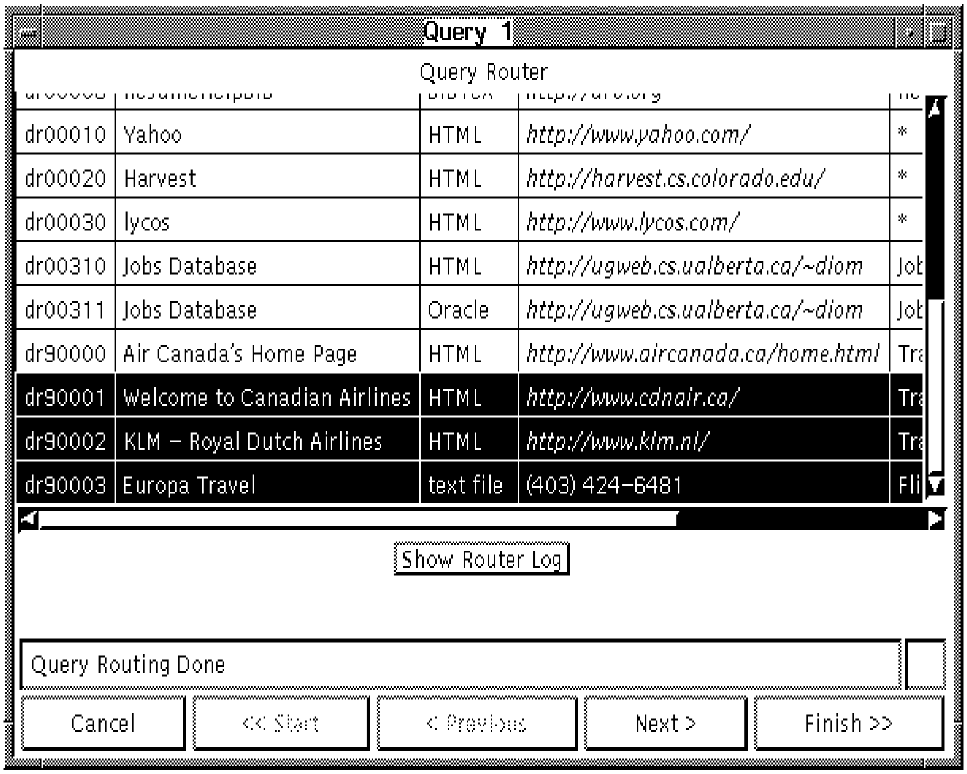
The status text field is used to display the important user messages concerning the status of the query manager and its main components.

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 13.

5.2.2 Router Object

Router DQP component is responsible for the query routing step of the distributed query scheduling, it is instantiated by the *Query Manager*. The router screen (Figure 13) displays the result of the query routing step. The main part of the router screen 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 or accessible to DIOM. Users may select or deselect each source by clicking the mouse in the corresponding line. The automatically selected sources by the system 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 Router screen is Show Router Log button that brings up a Log View Window that contains the router log.



[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
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*	Job
dr00311	Jobs Database	Oracle	*http://ugweb.cs.ualberta.ca/~diom*	Job
dr90000	Air Canada's Home Page	HTML	*http://www.aircanada.ca/home.html*	Tra
dr90001	Welcome to Canadian Airlines	HTML	*http://www.cdnair.ca/*	Tra
dr90002	KLM – Royal Dutch Airlines	HTML	*http://www.klm.nl/*	Tra
dr90003	Europa Travel	text file	(403) 424-6481	Fli

Query Routing Done

Figure 13: Query Manager Control Panel showing Router Screen

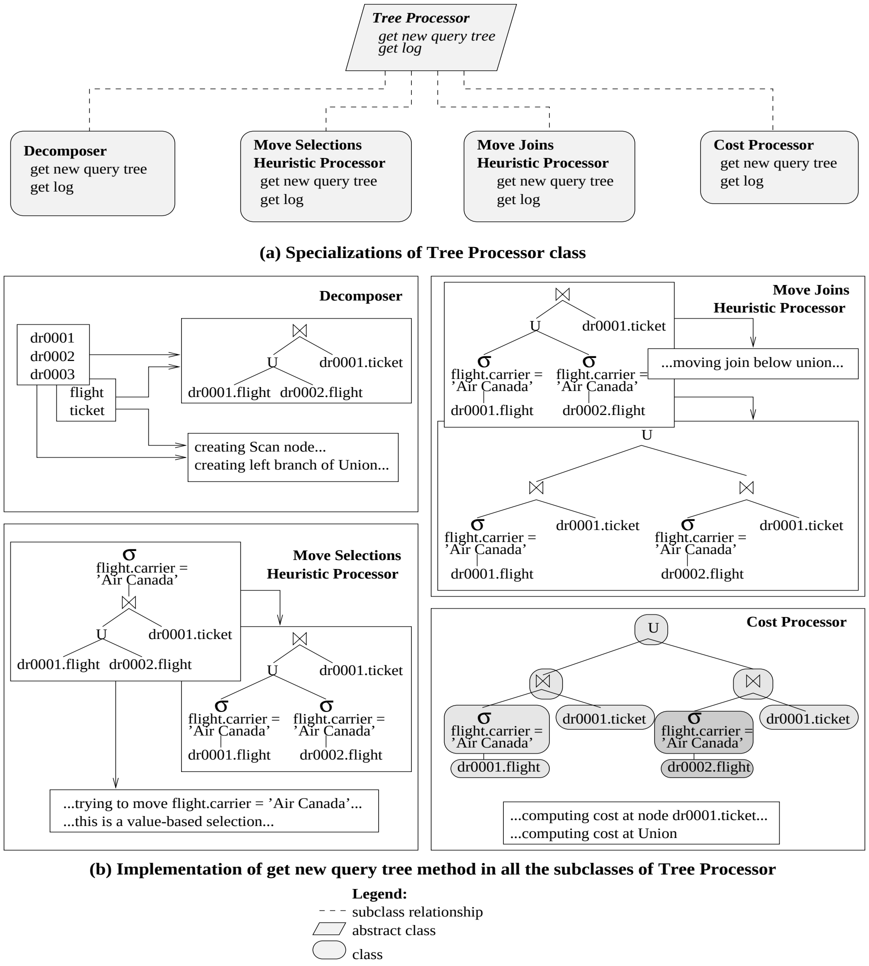
5.2.3 Query Tree Processor Object

This is an abstract component whose main function is to process a binary tree of *Query Tree Node*. *Query Tree Processor* 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*. It contains the intrinsics of the current component's query processing. Given a query tree, the tree processor object processes it and generates a new query tree;


Tree Processor
get new query tree
get log
Decomposer
get new query tree
get log
Move Selections Heuristic Processor
get new query tree
get log
Move Joins Heuristic Processor
get new query tree
get log
Cost Processor
get new query tree
get log
Decomposer
dr0001
dr0002
dr0003
flight
ticket
U
dr0001.ticket
dr0001.flight
dr0002.flight
creating Scan node...
creating left branch of Union...
Move Joins Heuristic Processor
...moving join below union...
flight.carrier = 'Air Canada'
Move Selections Heuristic Processor
...trying to move flight.carrier = 'Air Canada'...
...this is a value-based selection...
Cost Processor
...computing cost at node dr0001.ticket...
...computing cost at Union
Legend:
subclass relationship
abstract class
class

(a) Specializations of Tree Processor class

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

Figure 14: Tree Processor Classification.

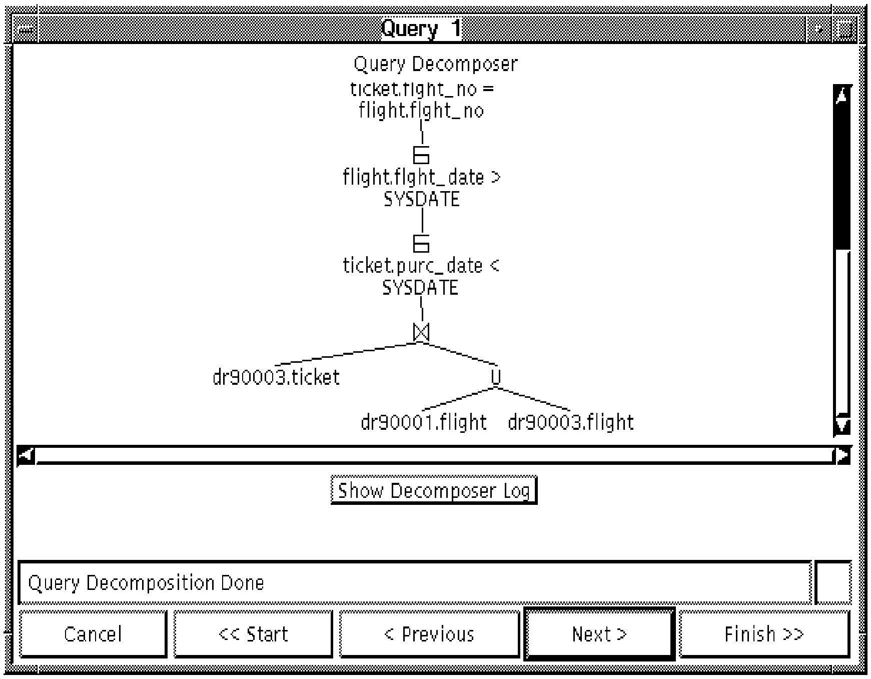
Figure 14 illustrates how the subclasses of *Query Tree Processor* implement the *get new query tree* method.

5.2.4 Decomposer Object

Decomposer is responsible for inserting the union collector nodes into the query tree produced from the query routing step in the distributed query processing session of the *DIOM Query Scheduling Utility*. An example of the Decomposer Processor Panel is shown in Figure 15.

5.2.5 Heuristic Processor Object

The current version of *DIOM Query Scheduling Utility* supports two of the heuristics covered in Section 3, moving selections down and moving joins below unions. Their design specifications are presented below. Figure 16 displays the lower part of the result tree produced from the query parallelization phase by applying heuristic *Move Selections Down* after inserting the union collector nodes.



Query Decomposition Done

Figure 15: Query Tree Screen displaying the parallelized query plan tree for the query in Figure 12.

5.2.6 Cost Processor Object

This component is responsible for the final step in distributed query Scheduling – site selection and execution. It is also the most complex component and consists of two steps 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 each of the possible site selection and execution plans that are considered as potential candidates for the selection of an optimized query execution schedule. The theoretical model for this step is covered in sections 4.1 and 4.7.

The Cost Processor Panel is shown in Figure 17. This panel consists of the following GUI components:

Main display area displays the visible portion of the cost query tree 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;

Show Log 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 brings 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.

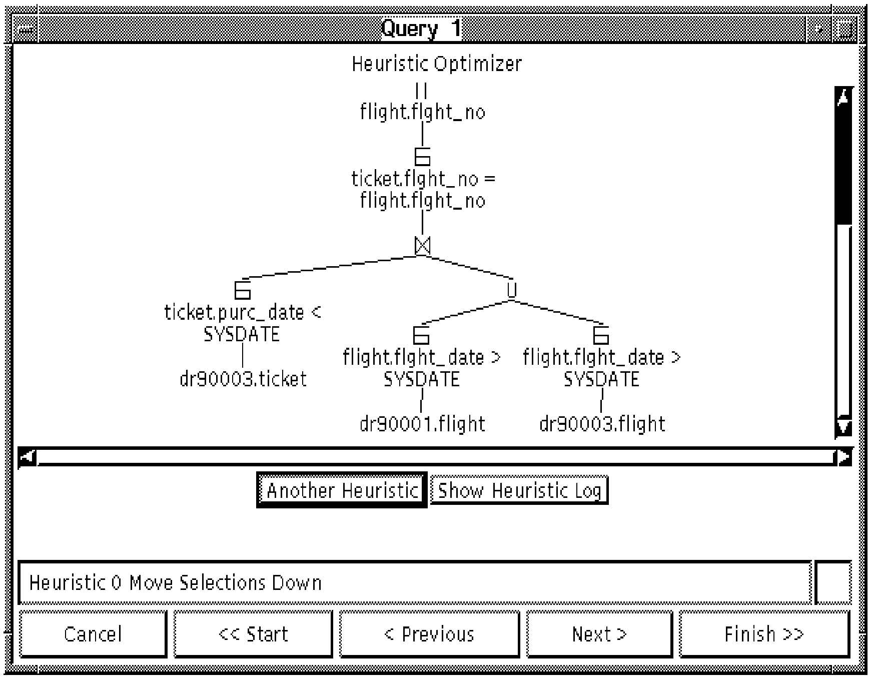


Figure 16: Query Tree Screen displaying the result of applying *Move Selections Down* heuristic to the parallelized query tree shown in Figure 15.

5.3 Maintenance and Diagnostic Components Specification

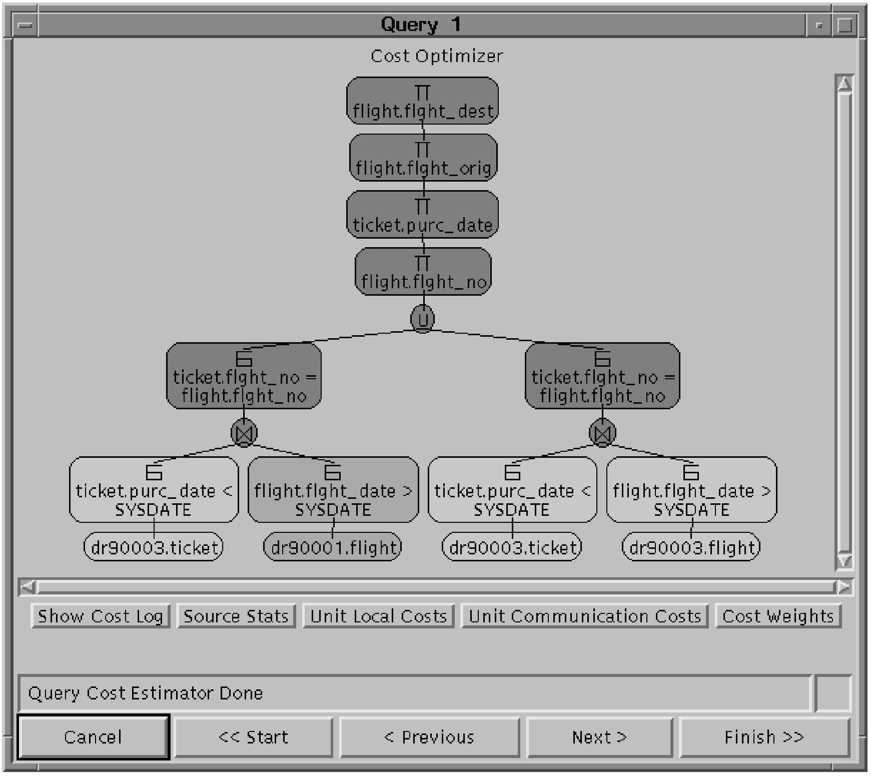
To develop bug-free, robust software, there needs to be a facility that allows to test and diagnose each of its design components. Each of the components of *DIOM Query Scheduling Utility* has been equipped with uniform diagnostic and bug detection methods.

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. One distinct feature of the DQS prototype is the provision of log view window which annotates the process algorithm. An example of Log View Window is shown in Figure 18. It displays the log of *Move Selections Heuristic Processor* for the running example given in Figure 12.

5.4 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. The DQS 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;



Query Cost Estimator Done

Figure 17: Cost Processor Panel Screen

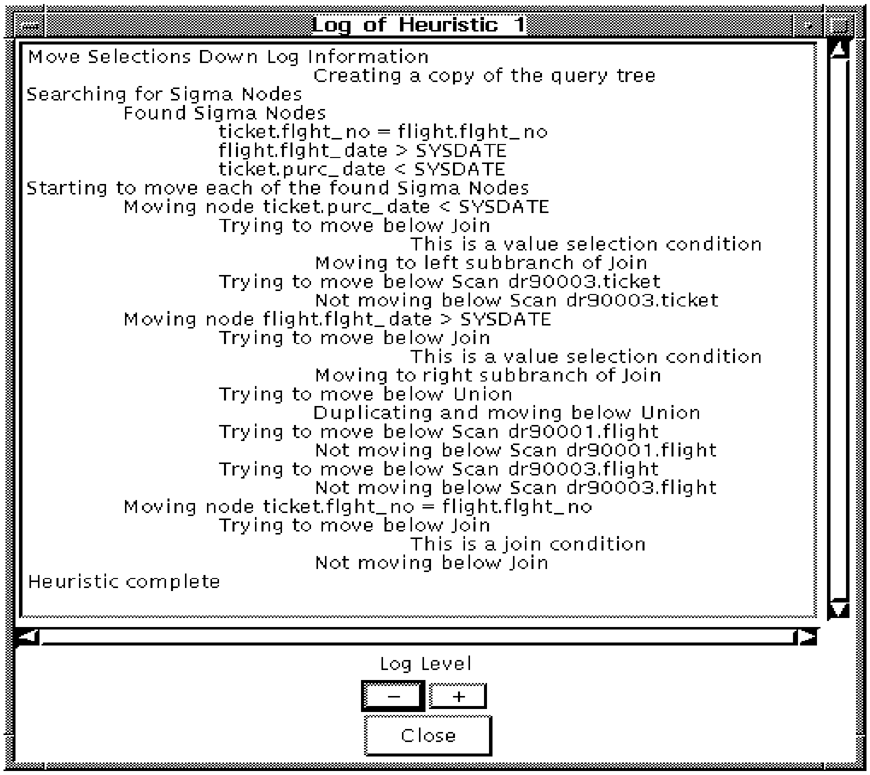
- Sun OS v. 4.1.4, using Netscape Navigator v. 3.0.

In addition to JDK 1.1, the main technologies used in the prototype implementation of the DQS utility also include Perl, Oraperl, SQL/Plus, MicroSoft SQL server, and Oracle 8.0.

6 Conclusion

We have presented the design framework, the algorithm and a prototype implementation of the DIOM *Distributed Query Scheduling* (DQS) service. The DQS is an example of distributed query scheduling services in the Internet that provide higher levels of query capability compared to keyword-based search. Specifically, DQS supports efficient inter-site joins over heterogeneous and evolving data sources. We believe that middleware level services such as DQS are an important step towards the next generation of Internet software that support information flow beyond browsing.

The paper contains two main technical contributions. First, we develop an extensible and scalable architecture for building a distributed query scheduling service based on the three-phase optimization model that separates conventional query optimization factors from distributed system factors. Concretely, Phase one is concerned with query structuring issues such as join order and applying selection before join. Phase two refines the query schedule by introducing intra-operator parallelism throughout the plan based on the data distribution factor. Phase three determines the best site selection plan by



```
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 18: Log View Window Screen

taking into account distributed system factors, such as execution location, autonomy and heterogeneity of the remote data sources. We have also developed the mechanism (union-collector nodes) by which DQS introduces and controls parallel execution, and a collection of cost function formulae and rules for selection of the best sites to process inter-site joins or inter-site unions. The query execution plan that minimizes the total cost and response time is generated and refined throughout the three-phase optimization process.

The second contribution is an experimental prototype of DQS that implements a subset of the proposed algorithm. The most interesting features of our implementation include (1) the capability for handling changes in source capability descriptions and the location of the data sources that are relevant to a query; and (2) the user-guided query processing performance tuning through a tracing facility, which is able of dynamically incorporating the changes in the unit cost and the local statistical information involved in a query.

The DQS service has limitations in its current state. For example, further research is needed in the large system issues such as system robustness, distributed failure recovery, and performance assessment. Specific issues that have not been addressed include recovery from source unavailability and server crashes, optimality of query schedules under various conditions, and APIs for high level software use of DQS services. Our work on the design and development of scalable and extensible distributed query scheduling service continues. On the development side, we are continuing our effort in the implementation of other heuristics described in Section 4 in the new release of the *DQS* software package. We are also interested in extending distributed query scheduling service to support continual queries [27, 28] in large-scale distributed systems. This includes the implementation methodology for building smart wrappers that can extract or sample useful statistical information and source capability descriptions of available data sources.

Acknowledgement

We would like to thank the DIOM implementation team at University of Alberta and Oregon Graduate Institute, especially David Buttler and Yooshin Lee. Our thanks are also due to the editors of this special issue and the reviewers for their helpful comments and suggestions. This research was partially supported by the ARPA grant MDA972-97-1-0016, the NSERC grants OGP0172859 and STR0181014.

References

[1] P. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *IEEE Trans. Software Engineering*, 9(1), 1993.

[2] Y. Arens and et al. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. J. B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 6(4):602–626, December 1981.

[4] M. Carey, L. Haas, P. Schwarz, and et al. Towards heterogeneous multimedia information systems: the garlic approach. In *IEEE Int. Workshop on Research Issues in Data Engineering*, 1995.

[5] Y. Chang, L. Raschid, and B. Dorr. Transforming queries from a relational schema to an equivalent object schema: a prototype based on f-logic. In *Proceedings of the International Symposium on Methodologies in Information Systems (ISMIS)*, 1994.

[6] C. Collet, M. Huhns, and W. Shen. Obtaining complete answers from incomplete databases. In *IEEE Computer*, 1991.

[7] U. Dayal. Query processing in a multidtatabase system. *Query Processing in Database Systems*, 1985.

[8] R. Epstein, Stonebraker, and E. Wong. Distributed query processing in relational database systems. In *Proc. ACM SIGMOD*, 1978.

[9] R. Epstein and M. Stonebraker. Analysis of distributed database strategies. In *The International Conference on Very Large Data Bases*, 1980.

[10] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Randomized join order selection - why use transformations? In *The International Conference on Very Large Data Bases*, 1994.

[11] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proc. ACM-SIGMOD Conf. on Management of Data*, June 1992.

[12] H. Garcia-Molina and et al. The TSIMMIS approach to mediation: data models and languages (extended abstract). In *NGITS*, 1995.

[13] L. Haas, D. Kossmann, E. Wimmers, and J. Yan. Optimizing queries across diverse data sources. In *The International Conference on Very Large Data Bases*, 1997.

[14] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1), 1988.

[15] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3), 1984.

[16] Y. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. ACM-SIGMOD Conf. on Management of Data*, 1991.

[17] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM-SIGMOD Conf. on Management of Data*, 1987.

[18] Y. Ioannidis and Y.Kang. Randomized algorithms for optimizing large join queries. In *Proc. ACM-SIGMOD Conf. on Management of Data*, 1990.

[19] N. Kabre and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM-SIGMOD Conf. on Management of Data*, 1998.

[20] Y. Lee. Rainbow: A prototype of the diom interoperable system. MSc. Thesis, Department of Computer Science, University of Alberta, July, 1996.

[21] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[22] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M.Valiveti. Capability based mediation in tsimmis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[23] B. G. Lindsay, L. Haas, C. Mohan, P. Wilms, and R. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Trans. Computer Systems*, 2(1), 1984.

[24] L. Liu. Query routing in structured open environments. Technical report, TR97-10, Department of Computing Science, University of Alberta, Edmonton, Alberta, Feb. 1997.

[25] L. Liu and C. 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.

[26] L. Liu and C. Pu. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *DISTRIBUTED AND PARALLEL DATABASES: An International Journal*, 5(2), 1997.

[27] L. Liu, C. Pu, R. Barga, and T. Zhou. Differential evaluation of continual queries. In *IEEE Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 27-30 1996.

[28] L. Liu, C. Pu, W. Tang, J. Biggs, D. Buttler, W. Han, P. Benninghoff, and Fenghua. CQ: A Personalized Update Monitoring Toolkit. In *Proceedings of ACM SIGMOD Conference*, 1998.

[29] L. Mackert and F. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, 1986.

[30] K. Morris. An algorithms for ordering subgoals in nail! In *ACM PODS*, 1988.

[31] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *The International Conference on Very Large Data Bases*, 1990.

[32] M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1991.

[33] C. Papadimitriou and K. Steilitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[34] Y. Papakonstantinou, A. Gupta, and L. Haas. Capability-based query rewriting in mediator systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[35] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *The International Conference on Very Large Data Bases*, 1997.

[36] K. Richine. Distributed query scheduling in the context of diom: An experiment. MSc. Thesis, Department of Computer Science, University of Alberta, April, 1997.

[37] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *ACM PODS'97*, 1997.

[38] P. G. Selinger and M. Adiba. Access path selection in distributed database management systems. In *Int. Conf. on Very Large Data Bases*, 1980.

[39] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. ACM-SIGMOD Conf. on Management of Data*, 1979.

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

[41] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3), 1997.

[42] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering*, 9(1), 1986.

[43] M. Stonebraker. The design and implementation of distributed ingres. *The INGRES Papers*, M. Stonebraker (ed.)(Addison-Wesley, Reading, MA), 1986.

[44] A. Swami. Optimization of large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[45] A. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1989.

[46] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[47] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[48] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer Magazine*, March 1992.