# Capabilities Aware Planner/Optimizer/Executioner for COntext INterchange Project

**by**

**Tarik Alatovic**

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute Of Technology

February 04, 2002

Author ______________________________________________
Department of Electrical Engineering and Computer Science
February 04, 2002

Certified By ______________________________________________
Dr. Stuart Madnick
John Norris Maguire Professor of Information Technologies
Thesis Supervisor

Accepted By ______________________________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Capabilities Aware Planner/Optimizer/Executioner
for COntext INterchange Project
by
Tarik Alatovic

Submitted to the
Department of Electrical Engineering and Computer Science

February 04, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

I present the design and implementation of a Planner, Optimizer, and Executioner (POE) for the COntext INterchange System. POE takes a context-mediated datalog query from the mediation engine as an input and returns query answer results assembled from data from remote data sources. Context-mediated queries are composed of component subqueries (CSQs) to multiple data sources. The Planner creates a query execution plan, which specifies the execution order of the CSQs. The Executioner executes each CSQ remotely, merges the results, and returns an answer to the original query.

In this thesis, I present novel approaches to three important query answering issues: (1) handling data sources with varying processing capabilities, (2) optimizing queries in distributed environments where costs statistics are not readily available, and (3) integrating with non-relational data sources.

Thesis Supervisor: Dr. Stuart Madnick
Title: John Norris Maguire Professor of Information Technologies

## Contents

## List of Figures

# 1 Introduction

Context Interchange is a novel approach to the integration of heterogeneous data sources, which was developed at the MIT Sloan School of Management. It seeks to integrate a wide variety of data sources in a seamless manner, not only at the physical level, but also at the semantic level. The Context Interchange approach attacks the problem of semantic integration by proposing the idea of contexts associated with the sources and users of data. A context can be defined as “the assumptions underlying the way an agent represents or interprets data” [4]. The rationale behind this approach is that once all the assumptions about the data have been explicated and clarified, it would be easy to resolve the conflicts that arise because of the discrepancies in these assumptions.

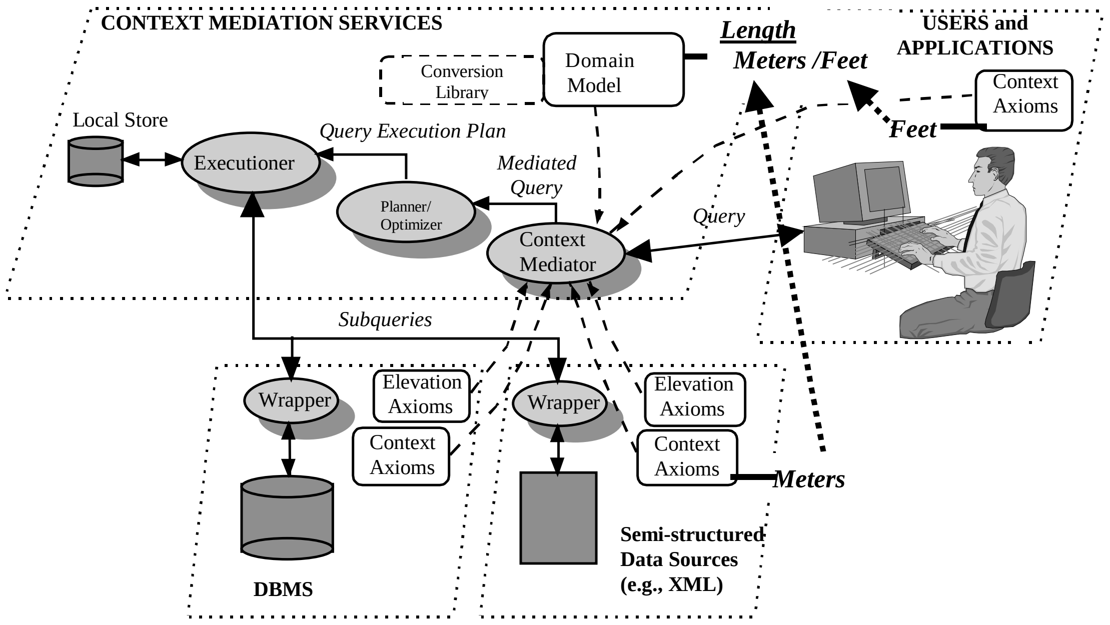


**Figure 1-1 Architectural Overview of the COntext INterchange System**

The COntext INterchange (COIN) project proposes the use of a context mediator [1], which is a component that sits between the users of data and the sources of data. When a user sends a query on the set of data sources, the context mediator analyzes the query to identify and resolve any semantic conflicts, which may exist in it. It resolves the conflicts by rewriting the query with the necessary conversions to map data from one context to another. The rewritten query is referred to as a context mediated query. This idea has been implemented in the COIN system, which integrates standard relational databases and wrapped non-relational data sources. Examples of wrapped data sources are web sources and user defined functions, which provide arithmetic and string manipulation features.

The diagram in Figure 1-1 shows the architecture of the COIN system. It is a three-tier architecture. Users interact with the client processes, which route all requests to the context mediator. The second set of processes in the COIN system are the mediator processes which consist of the context mediator and Planner/Optimizer/Executioner

(POE). These processes provide all the mediation services. Together, they resolve the semantic conflicts, which exist in the query and also generate and execute a query execution plan (QEP). Finally, the server processes provide the physical connectivity to the data sources. They provide a uniform interface for accessing both relational databases and wrapped non-relational sources.

In this thesis, I present the design and implementation of the POE component of the COntext INterchange System. The input to POE is a context-mediated datalog query from the mediation engine and the output is a query answer assembled from the data fetched from relational databases and non-relational data sources. My work focuses on resolving three important problems: (1) handling data sources with varying processing capabilities, (2) optimizing queries in distributed environments where costs statistics are not readily available, and (3) integration with non-relational data sources.

## 1.1 Motivational Example

We start with a motivational example that is used throughout the thesis to explain the issues involved in planning and execution of context mediated queries.

```
context=c_ws
select DStreamAF.NAME, DStreamAF.TOTAL_SALES,
       WorldcAF.LATEST_ANNUAL_FINANCIAL_DATE, WorldcAF.TOTAL_ASSETS
from   DStreamAF, WorldcAF
where DStreamAF.AS_OF_DATE = '01/05/94'
      and WorldcAF.COMPANY_NAME = DStreamAF.NAME
      and DStreamAF.TOTAL_SALES > 10,000,000;
```

**Figure 1-2 User Query**

Joan poses the query shown in Figure 1-2. She is interested in the names, total sales, and total assets of all companies with total sales greater than 10,000,000. This information is provided by two relations: DStreamAF and WorldcAF. Both DStreamAF and WorldcAF provide historical financial data on a variety of international companies. Joan is used to working with Worldscope data source (providing WorldcAF relation), but it now also needs to query DStreamAF relation from Datastream source because only DStreamAF contains data on the total assets that she is interested in. Schemas and contexts[1] for these two relations are shown in Figures 1-3 and 1-4 below.

| **WorldcAF** | |
|---|---|
| COMPANY_NAME | string |
| LATEST_ANNUAL_FINANCIAL_DATE | string |
| CURRENT_OUTSTANDING_SHARES | number |
| NET_INCOME | number |
| SALES | number |
| TOTAL_ASSETS | string |

| **DStreamAF** |
|---|

[1] Complete schemas and contexts for all working examples are given in Appendices A and B

|  |  |
|---|---|
| AS_OF_DATE | string |
| NAME | string |
| TOTAL_SALES | number |
| TOTAL_EXTRAORD_ITEMS_PRE_TAX | number |
| EARNED_FOR_ORDINARY | Number |
| CURRENCY | String |

**Figure 1-3 Schema for WorldcAF and DStreamAF Relations**

| Context | Currency | Scale Factor | Currency Type | Date Format |
|---|---|---|---|---|
| Worldscope, c_ws | USD | 1000 | 3char | American Style / |
| Datastream, c_dt | Country of Incorporation | 1000 | 2char | European Style - |

**Figure 1-4 Worldscope and Datastream Contexts**

Because Joan is used to working with Worldscope, she wants results reported in Worldscope contexts where all financial figures have a scale factor of a 1000 and are reported in American dollars. She is interested in financial results as of January, 5th, 1994 and she specifies this date in a Worldscope format as '01/05/94'.

The query Joan entered is then sent for processing to the meditation engine. The context mediator resolves semantic conflicts by rewriting the query with the necessary conversions to map data from Datastream context to Worldscope context. Figure 1-5 below shows the resulting mediated datalog query and Figure 1-6 its SQL translation generated using POE's SQL Translator component described in section 4.3.

```
answer('V31', 'V30', 'V29', 'V28') :-
        datexform('V27', "European Style -", "01/05/94", "American Style /"),
        'Name_map_Dt_Ws'('V26', 'V31'),
        10000000 < 'V30',
        'DStreamAF'('V27', 'V26', 'V30', 'V25', 'V24', 'V23'),
        'currency_map'('USD', 'V23'),
        'WorldcAF'('V31', 'V29', 'V22', 'V21', 'V20', 'V28', 'V19').

answer('V18', 'V17', 'V16', 'V15') :-
        datexform('V14', "European Style -", "01/05/94", "American Style /"),
        10000000 < 'V13',
        'Name_map_Dt_Ws'('V12', 'V18'),
        'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
        'currency_map'('V7', 'V8'),
        <>('V7', 'USD'),
        'V13' is 'V11' * 'V5',
        olsen('V7', 'USD', 'V5', "01/05/94"),
        'V17' is 'V11' * 'V5',
        'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

**Figure 1-5 Mediated Datalog Query for Motivational Example**

```
select name_map_dt_ws.ws_names, DStreamAF.total_sales, worldcaf.latest_annual_financial_date,
         worldcaf.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
          from   datexform          where  format1='European Style -'
          and    date2='01/05/94'
          and    format2='American Style /') datexform,
         (select dt_names, ws_names
          from   name_map_dt_ws) name_map_dt_ws,
         (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
          from   DStreamAF
          where  10000000 < total_sales) DStreamAF,
        (select 'USD', char2_currency
         from   currency_map
         where  char3_currency='USD') currency_map,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
                  net_income, sales, total_assets, country_of_incorp
       from   worldcaf) worldcaf
where  DStreamAF.currency = currency_map.char2_currency
and    name_map_dt_ws.ws_names = worldcaf.company_name
and    datexform.date1 = DStreamAF.as_of_date
and    name_map_dt_ws.dt_names = DStreamAF.name
and    10000000 < DStreamAF.total_sales
union
select name_map_dt_ws2.ws_names, DStreamAF2.total_sales*olsen.rate,
          worldcaf2.latest_annual_financial_date, worldcaf2.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
          from   datexform
          where  format1='European Style -'
          and    date2='01/05/94'
          and    format2='American Style /') datexform2,
         (select dt_names, ws_names
          from   name_map_dt_ws) name_map_dt_ws2,
         (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
          from   DStreamAF) DStreamAF2,
         (select char3_currency, char2_currency
          from   currency_map
          where  char3_currency <> 'USD') currency_map2,
         (select exchanged, 'USD', rate, '01/05/94'
          from   olsen
          where  expressed='USD'
          and    date='01/05/94') olsen,
         (select company_name, latest_annual_financial_date, current_outstanding_shares,
                net_income, sales, total_assets, country_of_incorp
          from   worldcaf) worldcaf2
where  name_map_dt_ws2.ws_names = worldcaf2.company_name
and    datexform2.date1 = DStreamAF2.as_of_date
and    name_map_dt_ws2.dt_names = DStreamAF2.name
and    DStreamAF2.currency = currency_map2.char2_currency
and    currency_map2.char3_currency = olsen.exchanged
and    10000000 < DStreamAF2.total_sales*olsen.rate
and    currency_map2.char3_currency <> 'USD'
```

**Figure 1-6 SQL Translation of Mediated Datalog of Motivational Example**

A datalog query is a logical rule of the form A:-B, where A is the head of the rule and B is the body. The procedural reading for the rule A:-B is: “To answer the query A, we must first answer B.” In Figure 1-5, the head of each query is an answer() predicate. This predicate represents the result of the query. Each fact in B is either a relational predicate, a boolean statement, or a conversion function. We refer to all non-relational functions as conversion functions. Typically, these functions are introduced into the query to map data from one context to another[2].

Relational predicates represent queries to data sources. For example, the predicate 'DStreamAF'('V27', 'V26', 'V30', 'V25', 'V24', 'V23') represents the relation DStreamAF in the Datastream source. Each argument in the predicate represents an attribute of DStreamAF relation. The order of the arguments corresponds directly to the order of the attributes in the relation. Thus, V27 represents DStreamAF.AS_OF_DATE, V26 represents DStreamAF.NAME , V30 represents DStreamAF.TOTAL_SALES , V25 represents DStreamAF.TOTAL_EXTRAORD_ITEMS_PRE_TAX , V24 represents DStreamAF.EARNED_FOR_ORDINARY , and V23 represents DStreamAF.CURRENCY.

Variables shared by different facts must instantiate to the same value. In our example, the first attribute of Name_map_Dt_Ws and the second attribute of DStreamAF must instantiate to the same value because they are both represented by the variable V26. Thus, shared variables in the relations represent joins among the relations in the query. Shared variables are also used to show which attributes must be returned to the user as part of the answer. In our example, V30 is shared by DStreamAF and answer(). Therefore, the third attribute of DStreamAF (e.g. TOTAL_SALES) would have to be returned as part of the answer to the query.

Boolean statements represent selection conditions on attributes in the query. A statement like <>('V7', 'USD') restricts the value of currency_map.char3_currency to be other than 'USD'. Conversion functions represent the operations needed for context conversion. In our example, the second datalog query has the conversion function V17 is V11 * V5. V17 is the second argument in the answer() predicate, V11 is the third attribute of DStreamAF (e.g. TOTAL_SALES), and V5 is the third argument of olsen (e.g. Rate). This conversion function represents the fact that in order for the DStreamAF.TOTAL_SALES to be returned to the user, its value needs to be multiplied by currency exchange rate to account for context differences.

Each datalog query in the output of the context mediator resolves a possible semantic conflict in the original query. In our example, the two queries represent the two kinds of conflicts that could occur. The first datalog query handles the conflicts, which arise when financial figures in DStreamAF relation are reported in American dollars, while the second handles the conflicts, which occur when the financial figures in DStreamAF are reported in currency other than American dollars. In the first datalog query, fact 'currency_map'('USD', 'V23') ensures that DStreamAF financial figures are reported in American dollars. There is no need for a conversion of the currency of financial figures once this condition is satisfied. The only conversions that need to be performed are a translation of the name of the company from the Worldscope format to Datastream format and a date format conversion. In the second datalog query, facts 'currency_map'('V7', 'V8') and <>('V7', 'USD') ensure that DStreamAF financial figures are not reported in American

[2] We will see later that the conversion function datexform is used for date conversions, such as Datastream context date in 'European Style -' format to Worldscope context date format in 'American Style /'.

dollars. In this case a currency conversion using olsen relation is needed in addition to the previously mentioned conversions.

The relational predicates DStreamAF and WorldcAF represent the relations that were identified by the user in the original SQL query. Name_map_Dt_Ws, datexform, currency_map and olsen are auxiliary relations that provide information for mediation. name_map_dt_ws provides a translation of company names between Datastream and Worldscope formats. datexform provides date format conversions. currency_map provides translation between 3 letter currency codes and 2 letter currency codes. olsen provides historical exchange rates for a given date.

Execution of this query involves accesses to Datastream and Worldcaf relations to retrieve the information Joan needs. Because the user expects all financial data in US Dollars, each financial figure requested by the user needs to be converted to US Dollars. This conversion requires the exchange rate between the currency of financial figures and US Dollars. Relation olsen provides this conversion. However, it imposes the restriction that the olsen.Exchanged, olsen.Expressed, and olsen.Fate attributes be bound. Bindings for olsen.Expressed and olsen.Date are 'USD' and '01/05/94' respectively. To get around binding restriction[3] for olsen.Exchanged we use the CURRENCY attribute of DStreamAF relation (converted from 2char to 3char representation using currency_map). Using these olsen bindings we can retrieve the exchange rate and use it to convert DStreamAF financial figures to US Dollars.

## 1.2 Organization of the Thesis

The motivational example provided us with insight into steps necessary for retrieving the query answer. The rest of the thesis describes design and implementation of Planner, Optimizer, and Executioner components.

In chapter 2, I present related work in areas of source capabilities and distributed query optimization. In chapter 3, I present an overview of the POE architecture and discuss important implementation choices.

In chapter 4, I develop internal query representation, show how input datalog query is parsed into internal representation, and describe SQL Translator component that generates SQL corresponding to the internal query representation.

In chapter 5, I describe Executioner component. I show how data is fetched from remote sources, how it is stored locally, and how it is merged into a query answer. I also discuss how query execution is parallelized.

In Chapter 6, the richest chapter, I describe Planner and Optimizer components. I start the chapter with a discussion on query restrictions and present capability records – the language for capturing query restrictions. I continue with describing the format of query execution plan (QEP). The rest of the chapter discusses QEP generation algorithm.

[3] Binding restrictions are discussed in detail in section 6.1 on query restrictions.

Because of the complexity of the algorithm, I describe it in several stages starting of with a trivial version of the algorithm and ending with a cost optimized version.  I also discuss query planning issues arising from restrictions on the queries that we can execute on the remote sources.

In Chapter 7, I discuss issues arising from integration with non-relational data sources.

I conclude with chapter 8, which presents future directions for improvements of COIN Planner/Optimizer/Executioner.

# 2 Related Work

An earlier version of Planner/Optimizer/Executioner for COIN system was built by Koffi Fynn [16]. My POE offers improvements over Koffi Fynn's in the following areas:

- handling data sources with varying processing capabilities
- optimizing queries in distributed environment where costs statistics are not available
- integration with non-relational data sources

In the area of handling data source with varying processing capabilities, I improve upon Koffi Fynn by handling key-at-a-time restriction common among web sources.

In the area of distributed query optimization, I improve significantly over the Koffi Fynn POE. I generate costs statistics for database queries and use cost-optimization algorithm to minimize transfer of tuples over the network – the key factor of query execution time. I also introduce parallel execution of component subqueries and key-at-a-time queries.

In the area of integration with non-relational data sources, I improve over Koffi Fynn by integrating with Cameleon web wrapper engine through CGI interface and retrieving query results in XML format. I also introduce Function Container Servlet used for exporting conversion functions.

In addition, my POE has features not present in Koffi Fynn Executioner. Most notably, SQL Translator component is used for translating mediation datalog into more readable SQL notation. Also, I support a query execution trace feature, which shows in detail operations that Executioner performs in answering the query.

The biggest difference between my and Koffi Fynn Executioner is in the underlying technology. Koffi Fynn POE implementation uses Prolog and my POE implementation uses Java and is built upon Oracle RDBMS engine (see section 3.2 for detailed discussion of implementation choices).

In the rest of this section, I present a short survey of research work done in the areas of query capabilities and distributed query optimization.

## 2.1 Source Capabilities

There are a number of efforts to develop languages for describing the capability restrictions imposed by sources. Examples of research projects, which are tackling these problems, are the Garlic Project at IBM, TSIMMIS Project at Stanford, Information Manifold, and Disco.

IBM's Garlic project incorporates capability based query rewriter based on Relational Query Description Language (RQDL) [13]. RQDL developed by researchers at Stanford, University of California, and University of Maryland is an extension of Datalog and can describe large and infinite sets of supported queries as well as schema independent queries. However, the algorithms used in Garlic for integrating query capability descriptions in the query planning phase only works among the different components of a single subquery and not among different subqueries. For example, Garlic does not consider the possibility of using information from one source to provide the information required by other sources. It only uses the information to remove restrictions within a single source.

Information Manifold uses the notion of a capability record [10] for representing query capability restrictions where capability descriptions are attached to the schema exported by the wrapper. The description states which and how many conditions may be applied on each attribute.

TSIMMIS uses a language called the Query Description and Translation Language (QDTL) [12]. It suggests an explicit description of the wrapper's query capabilities, using the context-free grammar approach. However, TSIMMIS considers a restricted form of the problem wherein descriptions consider relations of prespecified arities and the mediator can only select or project the results of a single CSQ.

DISCO [17] describes the set of supported queries using context-free grammars. This technique reduces the efficiency of capabilities-based rewriting because it treats queries as "strings."

## 2.2 Distributed Query Optimization

There has been a significant amount of work in building optimizers for distributed homogenous databases like system R*. However, only a limited amount of work has been done in optimization of queries in distributed heterogeneous environments. Moreover, the work that has been done in distributed heterogeneous environments is not universally applicable because different sets of assumptions are made in proposed optimization algorithms. For example, IBM’s Garlic system makes an assumption that each of the data sources provides reliable cost estimates in a format specified by Garlic system [18]. Another proposed optimization strategy is collecting statistical data on the cost of previously executed queries [19]. This strategy assumes that the data at sources is not changing rapidly and that queries performed are often repeated.

# 3 Design Overview

I first present the high-level architecture of the POE and then discuss the implementation choices made. I finish with an overview of software components.

## 3.1 POE Architecture

As its name suggests, POE consists of three main components: Planner, Optimizer, and Executioner (see Figure 3-1).

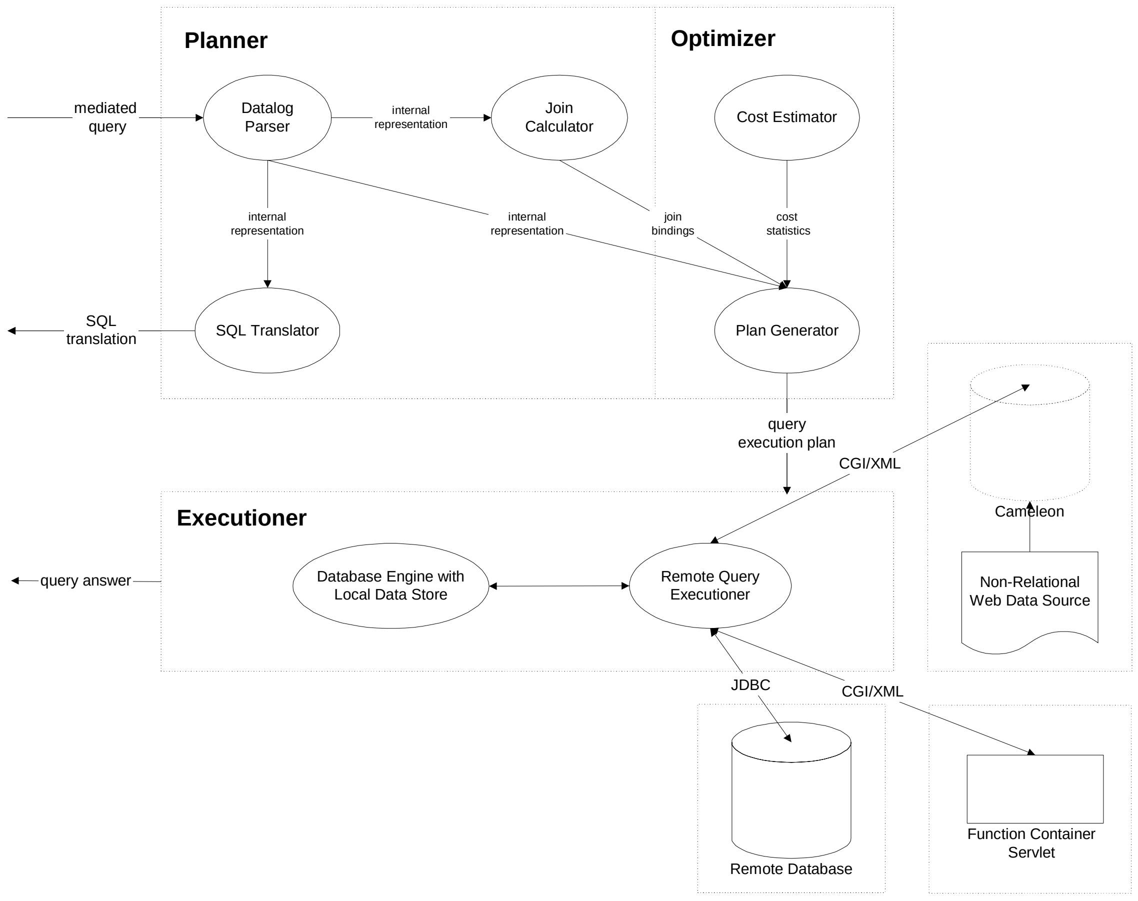


**Figure 3-1 POE Architecture**

Executioner executes the query plan. It dispatches the component subqueries to the remote sources and combines the returned results. It also performs joins and condition filtering that could not have been done at the remote sources. Intermediate results are stored in the local data store.

In my implementation, Executioner is built on top of the conventional RDBMS. The major challenge with this approach is that RDBMS can only perform relational operations on the locally stored data because it does not have ability to access data in remote sources. The task of efficiently bringing the data from remote sources into the local data store of RDBMS is in fact the major task of Planner and Optimizer components. Once the data from remote sources relevant for answering the query is brought into the local data store, the RDBMS can perform relational operations on it and answer the original query.

Planner takes a datalog query as an input and produces a query execution plan (QEP). QEP in my implementation differs from query execution plans used for single database executioners in that it operates at a higher level and does not specify complete ordering of relational operations. Instead, it specifies constraints that need to be satisfied in the execution of component subqueries (CSQs). The reasons why I chose a different role for QEP are twofold: (1) I rely on powerful Executioner component that can execute queries on locally stored data, and (2) CSQs may be executed in parallel, which means that complete ordering of CSQ execution may not be known until the run-time.

Planner ensures that CSQs in QEP can be executed at the remote sources. This is necessary because some data sources impose restrictions on the kinds of queries they can answer. Typically, some sources require that bindings be provided for some of their attributes or that only a specified number of attributes can be bound in a query. Additionally, sources can have restrictions on the types of relational operators, which they can handle.

Planner also includes a SQL Translator component. While SQL Translator is not an essential component of the Planner, it is useful for understanding of mediated queries because SQL tends to be more readable than the Datalog produced by the mediation engine. Moreover, SQL Translator proved to be useful as a debugging tool. SQL Translator is explained in the section 4.3.

Optimizer uses cost estimates to improve planner's QEP by searching for an optimal execution path – ordering of CSQs that minimizes transfer of tuples across the network. Optimizer is a critical component for the performance of the system because the order in which component subqueries are executed can result in orders of magnitude difference in the amount of network communication.

For example, a relation in source A with 1,000,000 tuples and a relation in source B with 5 tuples can be joined at primary keys in two ways. In the first way, 1,000,000 tuples from source A and 5 tuples from source B are shipped over the network to the local data store L, and then the join is performed in the local data store L. A total of 1,000,005 rows is transferred over the network. In the second way, we perform a remote join[4]. The local data store L first imports 5 tuples from source B and then forwards the join attribute values to source A. Source A uses join attribute values to identify tuples that need to be returned for performing join at the local data store L. Because A and B are joined at primary keys and B contains only 5 tuples, the source A will also return only 5 tuples after restricting tuples to those containing the join attribute values from source B. A total of 15 rows is transferred over the network. In conclusion, a small difference in the

[4] In section 5.4, I explain details of how remote joins are performed, and in section 6.3.5, I present a cost-optimization algorithm using remote joins.

ordering of operations may result in orders of magnitude difference in network communication – a major cost in execution of distributed queries.

## 3.2 Implementation Decisions

My implementation choices were guided by three important design goals:

- ease of integration
- ease of implementation
- robustness

Ease of integration is important because POE is just a component of a larger COIN project. Moreover, POE needs to communicate with a variety of data sources. Ease of implementation is important because the project is maintained by graduate students who work on this project for only a portion of the overall project lifecycle. Finally, robustness is important because data sources POE uses for fetching data are not reliable. For example, web sources may never return an answer or may take an unacceptable amount of time to return one.

In line with these design goals, I made the following implementation choices:

- implementing Planner/Optimizer/Executioner in Java
- using Oracle RDBMS as a basis of Executioner component
- using JDBC for database connectivity

### 3.2.1 Implementation Language

Using Java to implement POE meets both goals of ease of integration and ease of implementation. Goal of ease of integration is met because Java is used in COIN project as a glue between system components. Java also makes implementation easier than alternative languages for the following reasons:

- Java naturally supports database connectivity through JDBC interface
- Planner, Optimizer, and Executioner contain many algorithms that would be hard to code efficiently in the logical programming language that Mediation Engine is coded in
- Java provides a large number of libraries with pre-built functionality

### 3.2.2 RDBMS as Local Data Store

I chose to use Oracle RDBMS for local data storage and relational operation processing instead of building a multi-database executioner from ground up because this approach provides three important benefits:

- robust implementation of local data storage,
- efficent relational operations (e.g. joins and arithmetic calculations),
- shortened implementation time

### 3.2.3 Database Connectivity

Database connectivity is accomplished through JDBC interface. In line with my design goals, I chose JDBC instead of a more commonly used ODBC interface. The main reason is that ODBC is a C interface and is not appropriate for direct use from Java as calls from Java to C have drawbacks in the security and portability. Additionally, most databases provide JDBC interface. For databases that do not provide JDBC interface it is possible to use Sun's generic ODBC-JDBC bridge.

## 3.3 Software Components

The POE implementation consists of 40 Java modules grouped into 9 packages totaling 5,000 lines of code (see Appendix F). query, metadata, term, and operators packages implement abstract data types for internal data representation (explained in the next section). executioner package contains modules for remote execution of component subqueries and modules for implementing optimization algorithms.

# 4 Representations

Before we can start discussion of query execution algorithms, we need to understand the details of internal query representation. In this section, I describe internal query representation and show how input datalog query is parsed into the internal representation. I finish off the section with the description of the SQL Translator component, which generates SQL from the internal query representation. Additionally, in this section I will introduce the terminology used in the rest of the thesis.

## 4.1 Internal Query Representation

Figure 4-1 below shows components of the internal query representation. I explain in turn each of the query components.

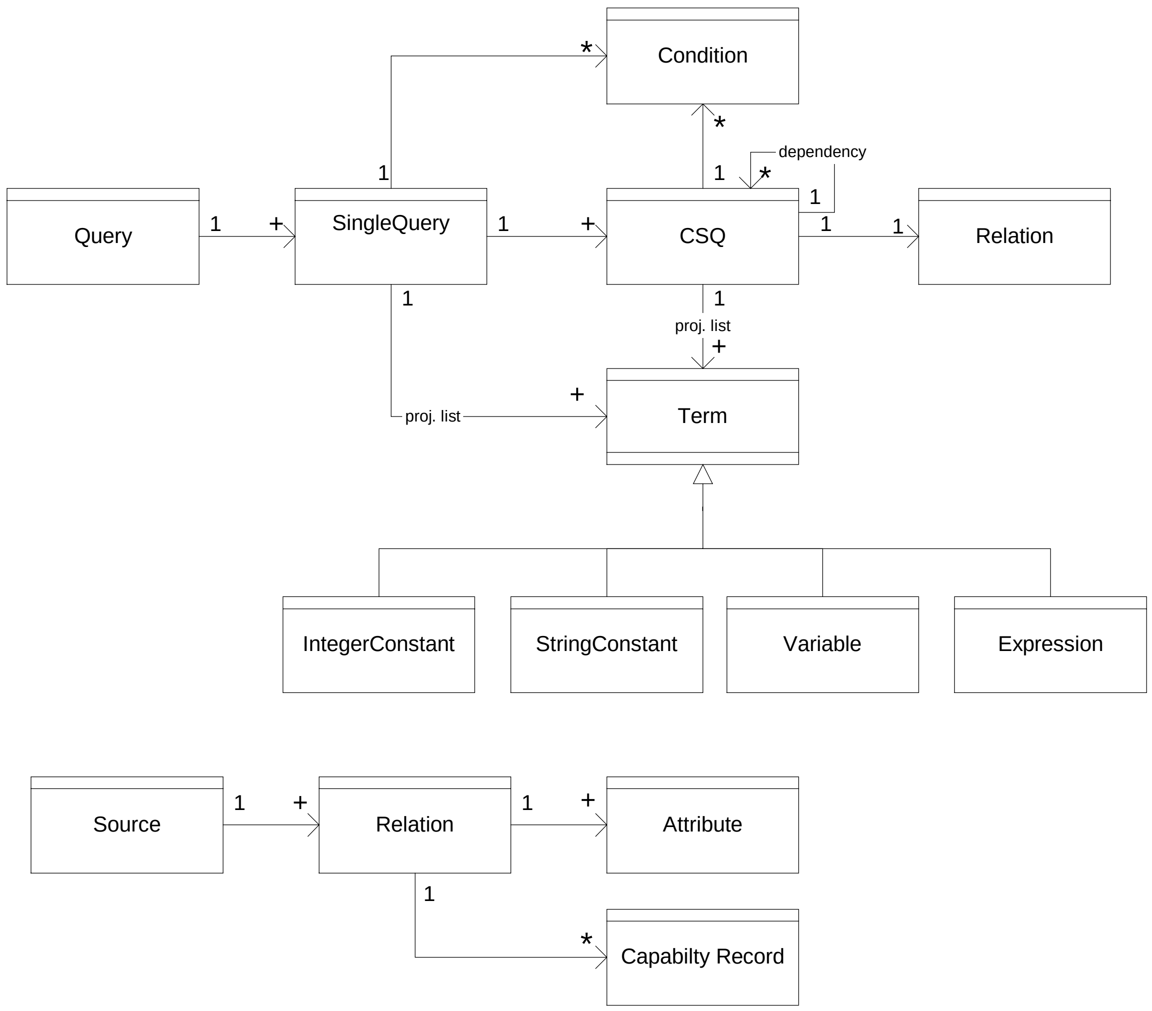


**Figure 4-1 Internal Query Representation**

### 4.1.1 Query

Query component represents the whole query. It is a union of Single Query components.

```
Query:
Single Query 1: answer('V31', 'V30', 'V29', 'V28') :-
                datexform('V27', "European Style -", "01/05/94", "American Style /"),
                'Name_map_Dt_Ws'('V26', 'V31'),
                10000000 < 'V30',
                'DStreamAF'('V27', 'V26', 'V30', 'V25', 'V24', 'V23'),
                'currency_map'('USD', 'V23'),
                'WorldcAF'('V31', 'V29', 'V22', 'V21', 'V20', 'V28', 'V19').

Single Query 2: answer('V18', 'V17', 'V16', 'V15') :-
                datexform('V14', "European Style -", "01/05/94", "American Style /"),
                10000000 < 'V13',
                'Name_map_Dt_Ws'('V12', 'V18'),
                'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
                'currency_map'('V7', 'V8'),
                <>('V7', 'USD'),
                'V13' is 'V11' * 'V5',
                olsen('V7', 'USD', 'V5', "01/05/94"),
                'V17' is 'V11' * 'V5',
                'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

In the datalog of the motivational example shown above, each of the two datalog queries maps to a Single Query component, and a complete datalog including both datalog queries maps to a Query component.

### 4.1.2 Single Query

Single Query represents a single datalog query and it consists of a projection list, component subqueries and query conditions. Let us examine second Single Query from our motivational example and identify each of the components:

```
Single Query 2: answer('V18', 'V17', 'V16', 'V15') :-
                datexform('V14', "European Style -", "01/05/94", "American Style /"),
                10000000 < 'V13',
                'Name_map_Dt_Ws'('V12', 'V18'),
                'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
                'currency_map'('V7', 'V8'),
                <>('V7', 'USD'),
                'V13' is 'V11' * 'V5',
                olsen('V7', 'USD', 'V5', "01/05/94"),
                'V17' is 'V11' * 'V5',
                'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

Projection list is a list of attributes returned by the query. Line (answer('V18', 'V17', 'V16', 'V15')) of Single Query 2 projects attribute ws_names of Name_map_Dt_Ws relation, attribute total_sales of DStreamAF relation multiplied by exchange rate obtained from olsen relation ('V17' is 'V11' * 'V5'), attribute latest_annual_financial_date of WorldcAF relation, and attribute total_assets of Worldcaf relation[5]. Note that items of projection list are Terms, where each term resolves to a relation attribute, a constant, or an expression (e.g. DStreamAF.total_sales*olsen.rate).

Component subqueries (CSQs) are subqueries accessing relations at remote data sources. Component subqueries also include conditions that can be executed on those relations. For example, the following part of Single Query2 constitutes a CSQ:

```
'currency_map'('V7', 'V8'),
<>('V7', 'USD'),
```

Remember that the order of the variables in relation corresponds directly to the order of the attributes in the relation. Thus, each variable in the predicate 'currency_map'('V7', 'V8') represents a corresponding attribute of currency_map relation. V7 is the first predicate, and it matches the first attribute of currency_map relation – char3_currency. V8 is the second predicate, and it matches the second attribute of currency_map relation – char2_currency. Note, also, that the currency_map CSQ has an associated condition that char3_currency differs from 'USD'.

Finally, Single Query consists of conditions that could not be performed at the level of component subqueries. These conditions can be classified into join conditions and explicit conditions. Join conditions appear between shared CSQ variables. For example, variable V14 is shared between CSQs datexform and DStreamAF. This means that attribute date1 of datexform is joined with attribute as_of_date of DStreamAF (datexform.date1=DStreamAF.as_of_date). If a join variable is shared by more than 2 CSQs then we compute a transitive closure finding all possible join condition between CSQs sharing the join variable.

Explicit conditions are either CSQ conditions that could not have been performed at the CSQ level because of query restrictions (e.g. no conditions can be specified on Cameleon's olsen relation) or non-join conditions between variables in different CSQs. An example of a non-join condition between variables in different CSQs is 10000000 < 'V13'. Variable V13 is an alias for V11*V5 where V11 is DStreamAF.total_sales and V5 is olsen.rate. Thus, this condition translates to 1000000<DStreamAF.total_sales*olsen.rate, which is a non-join condition involving CSQs DStreamAF and olsen.

### 4.1.3 Component Subquery (CSQ)

CSQ is a component query of a larger Single Query. It retrieves data from a single relation relevant in answering of Single Query. CSQ has a structure similar to Single Query and it consists of underlying relation, projection list, and CSQ conditions.

Let us examine CSQ currency_map in more detail:

[5] Appendix A shows the metadata file with schema for all relations in the motivational example.

```
'currency_map'('V7', 'V8'),
<>('V7', 'USD')
```

or in more friendly SQL notation:

```
select char3_currency, char2_currency
from   currency_map
where  char3_currency <> 'USD'
```

The underlying relation of this CSQ is currency_map relation. Indeed, we name CSQs after their underlying relations[6]. The projection list of currency_map CSQ are variables V7 and V8 corresponding in turn to attributes char3_currency and char2_currency of underlying currency_map relation.

Condition <>('V7', 'USD') belongs to this CSQ because all variables in this condition belong to this CSQ (in this case this is variable V7). Moreover, currency_map relation comes from a relational data source and is capable of executing condition <>('V7', 'USD').

As in the case of Single Query, the items in a projection list are Terms and can be constants, variables, or expressions. For example, olsen CSQ uses constants in its projection list (olsen('V7', 'USD', 'V5', "01/05/94") and translates to the following SQL:

```
select  exchanged, 'USD', rate, '01/05/94'
from    olsen
where   expressed='USD'
and     date='01/05/94'
```

### 4.1.4 Term

Term is a generic item that can appear in a query's projection list or its conditions. In our Java implementation, Term is an interface implemented by four classes: String Constant, Number Constant, Variable, and Expression.

Let us examine Single Query2 once again and identify some terms in it:

```
Single Query 2: answer('V18', 'V17', 'V16', 'V15') :-
                datexform('V14', "European Style -", "01/05/94", "American Style /"),
                10000000 < 'V13',
                'Name_map_Dt_Ws'('V12', 'V18'),
                'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
                'currency_map'('V7', 'V8'),
                <>('V7', 'USD'),
                'V13' is 'V11' * 'V5',
                olsen('V7', 'USD', 'V5', "01/05/94"),
                'V17' is 'V11' * 'V5',
                'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

[6] As we will see in the Datalog Parsing section, if several CSQs are present in the same query we number them sequentially.

‘V18’, ‘V17’, ‘V16’, and ‘V15’ in the answer clause are all Variable Terms.
“European Style -”, “American Style /”, and “01/05/94” in datexform CSQ are all String Constants.
1000000 in Condition 10000000 < 'V13' is a Number Constant.
‘V11’*’V5’ in Alias 'V13' is 'V11' * 'V5' is an Expression.

An Expression consists of two Term operands and an operator. Supported operators are +, -. *, /. Notice the recursive structure of the Expression. This recursive structure allows us to represent expressions of arbitrary complexity. For example, (‘V11’*’V5’)/’V3’ is an expression with first operand equal to expression ‘V11’*’V5’, second operand equal to expression V3, and operator equal to ‘/’.

Aliases are special types of variables that resolve to other terms. In our example, 'V13' is 'V11' * 'V5' is an alias clause. Alias V13 resolves to Expression 'V11' * 'V5'. In general, Aliases can resolve to any Term.

### 4.1.5 Condition

Condition component represents a SQL condition. It has a structure similar to Expression component and it consists of two Terms and a comparator. Supported comparators are =, <>, >, <, >=, and <=.

Let us examine condition 10000000 < 'V13'. V13 is an alias and it resolves to 'V11' * 'V5'. We can rewrite this condition as 10000000 < 'V11' * 'V5'. In this condition, the first Term is a Number Constant 10000000. The second Term is an Expression, 'V11' * 'V5'. The comparator is <.

### 4.1.6 Relation

The relation component captures schema information for a given relation. Every relation has a name unique within its data source and consists of one or more attributes. In addition to traditional schema information, relation component also contains a capability record, which specifies kinds of queries that can be executed on the relation.

```
relation(cameleon,
            olsen,
            [['Exchanged',string],['Expressed',string],['Rate',number],
             ['Date',string]],
            cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>='])).
```

**Figure 4-2 Relation Record for olsen Relation**

Above figure shows record for olsen relation[7]. It specifies Cameleon as the data source of olsen and specifies 4 attributes of olsen relation: Exchanged, Expressed, Rate, and Date. The last clause specifies the capability record.

[7] Appendix A shows the metadata file with schema for all relations in the motivational example.

### 4.1.7 Attribute

The attribute component represents a relation attribute. It is a simple structure consisting of attribute name and its type. In order to support schemas across a variety of databases, I decided to make attribute types as generic as possible. Therefore, only two attribute types are String and Number where arithmetic operations can be performed only on attributes of type Number. Figure 4-2 shows that olsen has Rate attribute of type Number and three attributes of type String: Exchanged, Expressed, and Date.

### 4.1.8 Source

Source component captures two types of information about the data source:

- How to physically connect to the data source
- What mechanism to use to query data in the remote source

```
source(oracle, database, ‘jdbc:oracle:oci8:@coin&system&manager’).
source(cameleon, cameleon, ‘http://context2.mit.edu:8081/servlet/camserv’).
```

**Figure 4-3 Source Records for Oracle and Cameleon Sources**

Figure 4-3 shows source records for remote sources Oracle and Cameleon. The first element in a source record is the source name. The second element tells us what mechanism to use to query the data in the remote source. Oracle source is treated as a database and SQL queries are sent to it through Relational Access interface[8]. Cameleon source can only accept SQL queries in functional form and is accessed through Functional Access interface. The third element contains physical connection properties for the source. Oracle source is physically accessed through JDBC interface with connection string jdbc:oracle:oci8:@coin&system&manager. Cameleon source is physically accessed through CGI interface at URL http://context2.mit.edu:8081/servlet/camserv.

### 4.1.9 Capability Record

The representation for Capability Records is given in section 6.1 on query restrictions.

## 4.2 Datalog Parser

Because datalog is a fairly simple language, I decided to use Java’s standard StringTokenizer class instead of more sophisticated grammar-based parsers, such as javacc. This proved to be a good choice because in only a few days I had a working datalog parser that converted input datalog into a list of tokens and then into internal query representation.

Conversion of datalog to internal representation is done in the following stages:

- Separation in datalog clauses
- Parsing individual clauses

[8] Remote Access Interfaces are discussed in detail in section 5.1.

- Post parsing
- Resolving aliases
- Naming CSQs
- Associating conditions with CSQs

In the first pass through the datalog query, parsing algorithm separates it into clauses and identifies the type of each clause. A datalog clause can be one of the following types:

- an answer clause
- a CSQ clause
- a prefix condition clause
- an infix condition clause
- an alias

```
answer('V31', 'V30', 'V29', 'V28') :-
        datexform('V27', "European Style -", "01/05/94", "American Style /"),
        'Name_map_Dt_Ws'('V26', 'V31'),
        10000000 < 'V30',
        'DStreamAF'('V27', 'V26', 'V30', 'V25', 'V24', 'V23'),
        'currency_map'('USD', 'V23'),
        'WorldcAF'('V31', 'V29', 'V22', 'V21', 'V20', 'V28', 'V19').

answer('V18', 'V17', 'V16', 'V15') :-
        datexform('V14', "European Style -", "01/05/94", "American Style /"),
        10000000 < 'V13',
        'Name_map_Dt_Ws'('V12', 'V18'),
        'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
        'currency_map'('V7', 'V8'),
        <>('V7', 'USD'),
        'V13' is 'V11' * 'V5',
        olsen('V7', 'USD', 'V5', "01/05/94"),
        'V17' is 'V11' * 'V5',
        'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

**Figure 4-4 Mediated Datalog Query for Motivational Example**

In the second stage, each of the datalog clauses is parsed into internal representation equivalents.

An answer clause (e.g. answer('V18', 'V17', 'V16', 'V15')) is mapped to a projection list of a Single Query. Each variable in the answer clause is mapped to a term, which resolves to a relation attribute, a constant, or an expression.

A CSQ clause (e.g. 'currency_map'('V7', 'V8')) is mapped to an underlying relation of a CSQ. currency_map CSQ is associated with the underlying relation currency_map. Variables in CSQ clause are mapped to attributes of a relation. Order is important in parsing of CSQ queries because we can determine the attribute of relation by its position in the relation clause. Thus, in 'currency_map'('V7', 'V8'), we know that V7 maps to attribute char3_currency and V8 maps to attribute char2_currency.

A prefix condition clause (e.g. <>('V7', 'USD')) specifies query condition in a prefix notation where operator comes before operands. It is mapped to a condition component of internal representation. In the third stage of conversion from datalog to internal representation, the condition is associated with either a CSQ or a Single Query.

A infix condition clause (e.g. 10000000 < 'V30') specifies query condition in an infix notation where operator comes between operands. It is mapped to a condition component of internal representation. In the third stage of conversion from datalog to internal representation, the condition is associated with either a CSQ or a Single Query.

An alias clause (e.g. 'V17' is 'V11' * 'V5') specifies variable as an alias for a Term component. In 'V17' is 'V11' * 'V5', V17 is an alias for expression 'V11' * 'V5'. Aliases are resolved in the third stage of conversion from datalog to internal representation.

After the second stage is complete, we are left with a projection list, a list of CSQs, a list of aliases, and a list of conditions. In the final post parsing stage, we resolve aliases, give unique name to CSQs and associate conditions with corresponding CSQs.

Aliases are resolved by substituting alias variables wherever they appear with the alias Term they represent.

We name CSQs with the names of their underlying relations. Since relation names are unique, the issue of unique CSQ names only comes into play when more than one relation is accessed in a datalog query. In this case, the CSQs are named sequentially. In the datalog of the motivational example, relation datexform is accessed in two CSQ clauses. As a result, the CSQ from first CSQ clause is named datexform, and the CSQ from second datexform clause is named datexform2.

Finally, conditions are associated with CSQs or Single Queries. I present the algorithm for associating conditions in Figure 4-5 below.

```
Associate Query Conditions:
1. for every condition cond in Single Query q
2.    let S be a set of all variables referenced in cond
3.        if all variables in S belong to a single CSQ c
4.            associate condition cond with a CSQ c
5.        else
6.            associate condition cond with a Single Query q
```

**Figure 4-5 Algorithm for Associating Query Conditions**

I will demonstrate this algorithm on the Single Query 2 of the motivational example:

```
answer('V18', 'V17', 'V16', 'V15') :-
        datexform('V14', "European Style -", "01/05/94", "American Style /"),
        10000000 < 'V13',
        'Name_map_Dt_Ws'('V12', 'V18'),
        'DStreamAF'('V14', 'V12', 'V11', 'V10', 'V9', 'V8'),
        'currency_map'('V7', 'V8'),
        <>('V7', 'USD'),
        'V13' is 'V11' * 'V5',
        olsen('V7', 'USD', 'V5', "01/05/94"),
        'V17' is 'V11' * 'V5',
        'WorldcAF'('V18', 'V16', 'V4', 'V3', 'V2', 'V15', 'V1').
```

Condition 10000000 < 'V13' contains alias variable V13, which resolves to 'V11' * 'V5'. Variable V11 belongs only to DStreamAF CSQ and variable V5 only to olsen CSQ. Thus, condition 10000000 < 'V13' does not reference variables within the same CSQ and is consequently associated with the Single Query 2.

Condition, <>('V7', 'USD') references variable V7, which belongs to currency_map CSQ. Thus, the condition is associated with the currency_map CSQ.

## 4.3 SQL Translation

Once we have datalog parsed into internal representation generating SQL representation of a query is fairly straightforward process as we now show. Figure 4-6 below shows generated SQL corresponding to the datalog of our motivational example.

```
select name_map_dt_ws.ws_names, DStreamAF.total_sales, worldcaf.latest_annual_financial_date,
        worldcaf.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
         from   datexform
         where  format1='European Style -'
         and    date2='01/05/94'
         and    format2='American Style /') datexform,
        (select dt_names, ws_names
         from   name_map_dt_ws) name_map_dt_ws,
        (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
         from   DStreamAF
         where  10000000 < total_sales) DStreamAF,
        (select 'USD', char2_currency
        from   currency_map
        where  char3_currency='USD') currency_map,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
                net_income, sales, total_assets, country_of_incorp
       from   worldcaf) worldcaf
where  DStreamAF.currency = currency_map.char2_currency
and    name_map_dt_ws.ws_names = worldcaf.company_name
and    datexform.date1 = DStreamAF.as_of_date
and    name_map_dt_ws.dt_names = DStreamAF.name
and    10000000 < DStreamAF.total_sales
union
select name_map_dt_ws2.ws_names, DStreamAF2.total_sales*olsen.rate,
         worldcaf2.latest_annual_financial_date, worldcaf2.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
         from   datexform
         where  format1='European Style -'
         and    date2='01/05/94'
         and    format2='American Style /') datexform2,
        (select dt_names, ws_names
         from   name_map_dt_ws) name_map_dt_ws2,
        (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
         from   DStreamAF) DStreamAF2,
        (select char3_currency, char2_currency
         from   currency_map
         where  char3_currency <> 'USD') currency_map2,
        (select exchanged, 'USD', rate, '01/05/94'
         from   olsen
         where  expressed='USD'
         and    date='01/05/94') olsen,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
              net_income, sales, total_assets, country_of_incorp
         from   worldcaf) worldcaf2
where  name_map_dt_ws2.ws_names = worldcaf2.company_name
and    datexform2.date1 = DStreamAF2.as_of_date
and    name_map_dt_ws2.dt_names = DStreamAF2.name
and    DStreamAF2.currency = currency_map2.char2_currency
and    currency_map2.char3_currency = olsen.exchanged
and    10000000 < DStreamAF2.total_sales*olsen.rate
and    currency_map2.char3_currency <> 'USD'
```

**Figure 4-6 SQL Translation of Mediated Datalog of Motivational Example**

While the generated SQL may seem awkward at first, it is in fact fairly easy to understand. First, notice that the above SQL is a union of two queries corresponding to Single Queries 1 and 2 of internal representation. Columns in select lists of the two queries are the Terms of Single Query projection list.

From statements of two queries consist of component subqueries accessing data at the remote sources. Consider the currency_map2 CSQ in Single Query 2:

```
(select char3_currency, char2_currency
 from   currency_map
 where  char3_currency <> 'USD') currency_map2
```

The CSQ SQL is generated directly from the internal representation. The select statement of CSQ SQL (select char3_currency, char2_currency) contains attributes of underlying relation currency_map. The where statement contains all the conditions associated with the CSQ. As we discovered at the end of section 4.2, the condition char3_currency <> 'USD' is associated with the currency_map2 CSQ.

The where statement of Single Query consists of two kinds of conditions: (1) join conditions between CSQ attributes and (2) conditions that could not have been associated with individual CSQs because they involve attributes from multiple CSQs.

In Single Query 1, condition DStreamAF.currency = currency_map.char2_currency in where statement is a join condition between currency attribute of DStreamAF CSQ and char2_currency attribute of currency_map relation.

In Single Query 2, condition 10000000 < DStreamAF2.total_sales*olsen.rate in where statement could not have been performed at the CSQ level because it references attributes from two distinct CSQs: DStreamAF2 and olsen.

# 5 Executioner

I describe the Executioner component before the Planner and Optimizer because strong capabilities of the Executioner were drivers behind the design of the Planner and Optimizer components. I build the Executioner on top of the conventional RDBMDS providing it with ability to answer queries on the relations stored in the local data store (see Figure 5-1 below).

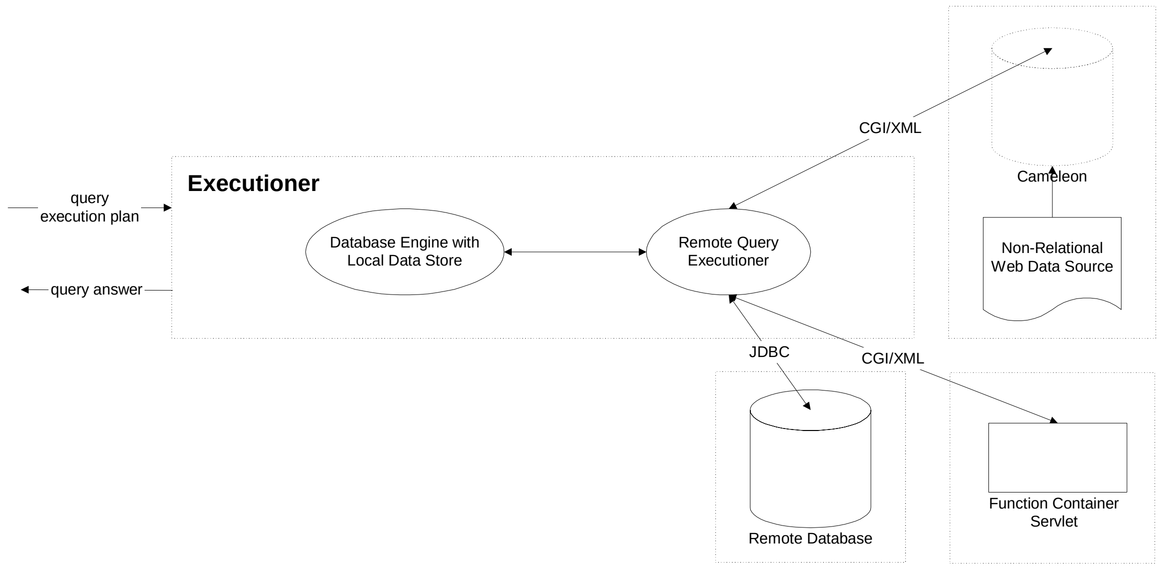


**Figure 5-1 Executioner Architecture**

I chose to use the Oracle RDBMS for local data storage and relational operation processing instead of building a multi-database executioner from ground up. This approach provides three important benefits:

- robust implementation of local data storage,
- efficient relational operations (e.g. joins and arithmetic calculations),
- shortened implementation time

The major challenge with using an RDBMS for this purpose is that an RDBMS can only perform relational operations on the locally stored data[9] because it does not have the ability to access data in remote data sources. The task of efficiently bringing the data from remote sources into the local data store of RDBMS is in fact the major task of Planner and Optimizer components. Once the data from remote sources relevant for answering the query is brought into the local data store the RDBMS can perform relational operations on it and answer the original query.

[9] Oracle can in fact access data stored in other Oracle database through database links. However, database links are limited to Oracle databases only and do not provide general solution to the problem of accessing data from remote sources.

In COIN POE, the Executioner assumes the following tasks: (1) fetching data from remote sources, (2) storing data from remote source locally, and (3) executing queries in parallel. I discuss in turn each of this tasks in the following three sections. In the last section, I show the generated query execution trace for motivational example.

## 5.1 Fetching Data From Remote Sources

As shown in the Figure 5-2, data sources are accessed through the Remote Access interface[10]. Remote Access interface is extended by two specific data access interfaces: Relational Access and Functional Access. Relational access interface is used for accessing data in relational databases supporting full SQL specification. Functional access is used for accessing data sources conforming to parameters-in, results-out model used in functional conversions and non-relational wrappers.

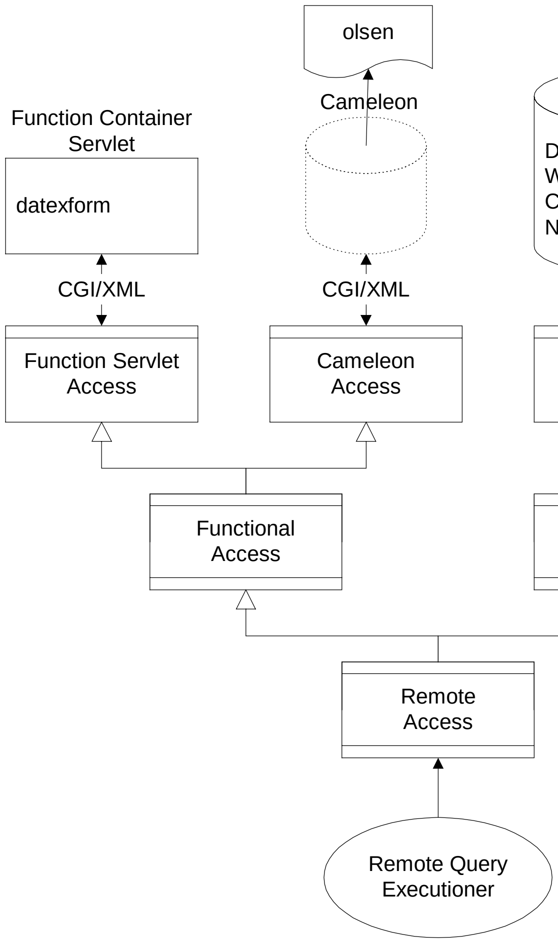


**Figure 5-2 Remote Access Hierarchy for Query in Motivational Example**

Each of the remote interfaces is implemented by one or more classes that encapsulate drivers for specific data sources. Relational Access interface is implemented by DB

[10] Definitions of remote access interfaces are given in Appendix C

Access class that uses JDBC to provide connectivity to relational databases. Functional Access interface is implemented by Cameleon Access class and Servlet Access class. Cameleon Access is used for accessing wrapped web sources. It uses CGI to send SQL to Cameleon engine and XML to get back the query results. Servlet Access class is used to access conversion functions. It uses CGI interface to send the function name and query parameters and it uses XML to get back the query results.

Figure 5-2 shows six relations accessing three distinct remote data sources in answering the motivational example query. Relations DStreamAF, worldcaf, name_map_dt_ws, currency_map stored in Oracle database are accessed through Relational Access interface because Oracle supports full SQL specification. olsen relation provides historical currency conversions and is accessible through Cameleon web wrapper. Even though Cameleon web wrapper accepts SQL syntax it is inherently non-relational because it only accepts functional queries subscribing to parameters-in, results-out model. Therefore, data in olsen relation is accessed through functional access interface. datexform relation performs data format conversions and is accessible through Function Container Servlet using functional access interface.

### 5.1.1 Relational Access

```
public interface RelationalAccess extends RemoteAccess {
   public OrderedTable executeQuery(String sql) throws Exception;
   public int type();
}
```

Relational access interface provides executeQuery method that takes SQL as input and returns results as an ordered list of tuples using OrderedTable structure (see Appendix D). In our example, data from DStreamAF relation is fetched through Relational Access interface by executing following SQL:

```
select as_of_date, name, total_sales, total_extraord_items_pre_tax,
         earned_for_ordinary, currency
from   DStreamAF
where  10000000 < total_sales
```

The tuples returned by executing this query are stored in Ordered Table structure. Data from other Oracle relations is fetched in the same way.

### 5.1.2 Functional Access

Data from olsen relation is fetched through Functional Access interface. Cameleon Access class implements Functional Access interface by constructing simple SQL queries that Cameleon web wrapper can handle and then sending them to Cameleon engine. Here is one of the queries executed remotely through Cameleon engine:

```
select exchanged, expressed, rate, date
from olsen
where exchanged="DEM"
and expressed="USD"
and date="01/05/94"
```

Note the functional form of the query with input parameters (exchanged, expressed and date) and output parameters (exchanged, expressed, rate, and date). Above SQL query is easily constructed from arguments in executeQuery method of functional access interface:

```
public interface FunctionalAccess extends RemoteAccess {
        public OrderedTable executeQuery(Relation r, boolean[] in, boolean[] out, String[] inBinding);
        public OrderedTable executeQuery(Relation r, boolean[] in, boolean[] out, OrderedTable inBindings);
}
```

inBinding parameter and boolean array in specify input parameters to the function. Boolean array out specifies parameters we want to get back from the relation r. Like in the case of relational access, results are returned in Ordered Table structure.

Functional Access interface is also used to fetch the data from datexform relation. datexform is a conversion function providing date format conversions. Class Servlet Access implements Functional Access interface. Servlet Access uses CGI interface to supply function name and input parameters to functions residing in function container servlet. The output results are returned in XML format.

Following is the format of request sent to function container servlet for conversion of date '01/05/94' from American Style with '/' used as delimiter to European Style with '-' used as delimiter:

```
http://hostname/servlet/edu.mit.gcms.functionservlet?function=datexform&
format1=European Style -&date2=01/05/94&format2=American Style /
```

The result of this conversion returned in XML format is:

```
<?>>>>>>>>>>>>>>>>>>>>?>>>
  <>>>>>>>>>   >>
  <>>>>>>>>> >>
 ><>>>>>>>05/0////  <>>>>>>>>> >>
 ><>>>>>>>>>////////////////    <>>>>>>>>>>>> >>
 ><>>>>>>>0//05///   <>>>>>>>>> >>
 ><>>>>>>>>>////////////////     <>>>>>>>>>>>> >>
 ><>>>>>>>>>> >>
 ><>>>>>>>>>>>   >>
```

## 5.2 Storing Data From the Remote Sources Locally

As explained at the opening of this section, the data from remote sources needs to be stored locally so that RDBMS engine can perform relational operations on it. However, the task of bringing the data into RDBMS at run-time is more challenging than it at first appears. The reason why this task is challenging is that inserts in relational databases involve hard disk operations[11]. Performing large inserts on regular RDMBS tables is a prohibitively expensive operation.

Oracle and some other commercial databases provide a solution for this problem through the facility of temporary tables designed for storage of temporary data. Temporary tables take much less time to create and do not involve hard disk operations. In addition, Oracle temporary tables have support for session and transaction specific data. We use this feature to empty the temporary table once the query is answered and temporary data is no longer needed.

I create temporary tables at the Executioner initialization time. The reason why I do not create them at the run time is that creating temporary table takes a long time – on the order of 50ms. However, creating temporary tables at the Executioner initialization time has its own problems. Because several CSQs can access the same relation in the query, we need to create several temporary relations for each CSQ and this is a clear scalability drawback[12]. In section 8.2.3, I present a better approach for managing temporary tables that solves both scalability and performance issues.

I name the temporary tables by the names of CSQs for which they are storing the data. Only minor nuisances with using Oracle temporary tables is the restriction on column names. These restrictions make it impossible to have one-to-one mapping between relation attribute names and temporary table column names.[13]

Temporary table for storing data fetched from olsen relation is created as follows:

```
create global temporary table TTP_olsen (
        DAP_exchanged       varchar,
        DAP_expressed       varchar,
        DAP_rate            number,
        DAP_date            varchar) on commit delete rows;
```

Few points are worth mentioning about the above temporary table definition. First, 'global temporary' in create table statement is the Oracle specific feature and designates the table as a temporary table. Second, TTP_ is prefix used to avoid naming collisions with system tables present in the database, and DAP_ is prefix used to avoid naming collisions with RDBMS reserved key words such as date. Both TTP_ and DAP_ are

[11] This is because RDBMS were designed for permanent data storage. Inserts and updates are not cached in the memory.
[12] Number of temporary tables to be created is specified in NUM_TEMP_TABLES_PER_RELATION constant in module edu.mit.gcms.poe.executioner.Executioner.java
[13] Column names are restricted to length of 32 characters and cannot feature any of Oracle's reserved keyword, such as, date.

customizable parameters and can be changed if the underlying RDBMS used for executioner is changed. Second, 'on commit delete rows' specifies that data stored in this temporary table is transaction specific and is automatically deleted once the transaction is committed. Executioner commits transaction once the results to original user query are returned and data stored in temporary table is no longer needed. Finally, we use two datatypes: varchar matching Java's String and number matching Java's double to store all the data in temporary tables. We need RDBMS's number type in order to be able to perform arithmetic operations specified in the original query datalog (e.g. olsen.rate*1000).

Once we create a temporary table, we need to populate it with data fetched from remote sources. The data from a remote source is returned in the Ordered Table structure and we need to insert this data into a temporary table. The most straightforward way of feeding data into table with the JDBC interface is by doing one insert at a time for each row or Ordered Table structure. However, this straightforward approach is inefficient because it makes unnecessary round trips between Executioner and underlying database engine. To avoid this performance problem, I used a new feature of JDBC 2.0 specification supported by Oracle driver, which allows for batch inserts of large amounts of data into the table.

## 5.3 Executing Queries in Parallel

The Executioner supports two kinds of parallel processing:

- parallel execution of CSQs
- parallel execution of key-at-a-time queries within CSQs.

The Executioner schedules in parallel a Java thread for fetching data for each of the CSQs. Since some CSQs depend on the data obtained from other CSQs – as specified in the query execution plan – the dependent CSQs need to wait until the thread fetching the data for the CSQ they depend on is executed. The mechanism for doing this is Java thread's join method which delays execution of a thread until other specified threads have completed their execution. Therefore, when all threads that a particular thread depends on are complete, the dependent thread is allowed to proceed. Once all Java threads complete – that is once the data from all CSQs is fetched – the Executioner proceeds to the next stage of merging CSQ data obtained from the remote data sources.

Parallel execution for key-at-time queries is simpler because key-at-a-time queries are independent of each other and can all be executed in parallel[14]. Figure 5-3 is a portion of query execution trace for motivational example query showing key-at-a-time queries for olsen CSQ sent to the Cameleon source. Execution of 4 queries took 877ms to complete because the longest running queries took 877ms and all four queries were executed in parallel at the same time.

[14] Executioner can define a maximum number of queries that can be sent to each source for processing. This is important because some servers may only handle only a few concurrent connections at any given time.

| | |
|---|---|
| 877ms | Parallel Execution |
| 290ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="DEM" and expressed="USD" and date="01/05/94" |
| 470ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="FRF" and expressed="USD" and date="01/05/94" |
| 480ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="JPY" and expressed="USD" and date="01/05/94" |
| 877ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="GBP" and expressed="USD" and date="01/05/94" |

**Figure 5-3 Parallel Execution of Key-at-a-Time Queries**

## 5.4 Executing Query Execution Plan

In this section, I explain how query execution plan (QEP) is executed[15]. QEP for motivational example query is shown in Figure 5-4 below.

| | |
|---|---|
| 0: | currency_map, datexform, currency_map2, datexform2 |
| currency_map: | DStreamAF ((currency currency_map.char2_currency)) |
| DStreamAF: | name_map_dt_ws ((dt_names DStreamAF.name)) |
| Name_map_dt_ws | worldcaf ((company_name name_map_dt_ws.ws_names)) |
| currency_map2: | DStreamAF2 ((currency currency_map2.char2_currency)),<br>olsen ((exchanged (currency_map2.char3_currency)) |
| DStreamAF2: | name_map_dt_ws2 ((dt_names DStreamAF2. name)) |
| Name_map_dt_ws2: | worldcaf2 ((company_name name_map_dt_ws2.ws_names)) |

**Figure 5-4 Query Execution Plan for Motivational Example Query**

As explained in the previous section, a Java thread is scheduled for execution of each CSQ. It follows that all independently executable CSQs in the first row of the QEP are executed in parallel. All other CSQs are executed once the CSQs they depend on have been executed. As soon as the execution of a CSQ c completes, all CSQs depending on the execution of CSQ c are notified and are then executed[16]. In our example, DStreamAF CSQ is dependent on currency_map CSQ. Once execution of currency_map is completed, thread executing DStreamAF is notified and may proceed with the execution. Similarly, thread executing Name_map_dt_ws blocks until the execution of DStreamAF has been completed.

Now, let us examine how remote joins are performed. In the second row of QEP shown in Figure 5-4, we see that DStreamAF's attribute currency is bound to the currency_map's

[15] QEP is explained in detail in section 6.2.
[16] Note that a CSQ may depend on the execution of more than one CSQ. In that case CSQ is executed once all the CSQs it depends on have been executed.

attribute char2_currency. I perform the remote join by using SQL's 'in' construct. Figure 5-5 shows remote join performed on DStreamAF relation using bindings from currency_map relation. We can now clearly see how DStreamAF CSQ depends on the execution of currency_map CSQ. The query in Figure 5-5 can only be executed once the data from currency_map CSQ has been fetched.

```
select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary,
currency
from   dstreamaf
where  10000000 < total_sales
and    currency in (select char2_currency from currency_map)
```

**Figure 5-5 Remote Join on DStreamAF CSQ Using Bindings from currency_map CSQ**

After we store the data from all CSQs in the temporary tables of local data store, we can use RDBMS's relational processing facilities to answer the original query. For our working example, the final relational query is:

```
select TTP_name_map_dt_ws.ws_names, TTP_DStreamAF.total_sales,
       TTP_worldcaf.latest_annual_financial_date, TTP_worldcaf.total_assets
from   TTP_datexform, TTP_name_map_dt_ws, TTP_DStreamAF, TTP_currency_map,
       TTP_worldcaf
where  TTP_DStreamAF.currency = TTP_currency_map.char2_currency
and    TTP_name_map_dt_ws.ws_names = TTP_worldcaf.company_name
and    TTP_datexform.date1 = TTP_DStreamAF.as_of_date
and    TTP_name_map_dt_ws.dt_names = TTP_DStreamAF.name
union
select TTP_name_map_dt_ws2.ws_names, TTP_DStreamAF2.total_sales*TTP_olsen.rate,
       TTP_worldcaf2.latest_annual_financial_date, TTP_worldcaf2.total_assets
from   TTP_datexform2, TTP_name_map_dt_ws2, TTP_DStreamAF2, TTP_currency_map2,
       TTP_olsen, TTP_worldcaf2
where  TTP_name_map_dt_ws2.ws_names = TTP_worldcaf2.company_name
and    TTP_datexform2.date1 = TTP_DStreamAF2.as_of_date
and    TTP_name_map_dt_ws2.dt_names = TTP_DStreamAF2.name
and    TTP_DStreamAF2.currency = TTP_currency_map2.char2_currency
and    TTP_currency_map2.char3_currency = TTP_olsen.exchanged
and    10000000 < TTP_DStreamAF2.total_sales*TTP_olsen.rate
```

Notice that the above query contains all join conditions of the original query and all conditions that could not have been applied at the CSQ level. The query also contains arithmetic operations (e.g. TTP_DStreamAF2.total_sales*TTP_olsen.rate) that are efficiently processed by the RDBMS engine.

## 5.5 Query Execution Trace

I developed query execution trace tool as a component of the Executioner. Query execution trace proved to be invaluable for debugging the Executioner as well as for getting the insight in the way the POE is answering the queries. Furthermore, timing information provided in the trace allowed me to identify query execution bottlenecks and focus on the most important planning and optimization issues.

Appendix G shows query execution trace for the motivational example query. Query execution trace is a hierarchical log with timing information and it consists of four main stages: datalog parsing, planning & optimization, fetching remote data, and execution of final database query. Datalog parsing stage only provides the time elapsed in parsing the datalog into the internal query representation. Planning & Optimization shows intermediate steps of calculating the query execution plan and the time elapsed for each of the intermediate stages.

Fetching remote data stage shows component subqueries sent to remote sources. The component subqueries are displayed in the format specific to the data source they are sent to. Thus, database and cameleon CSQs are displayed in SQL notation (see Figures 5-6 and 5-7), and queries to Function Container Servlet are displayed as a parameterized query URL (see Figure 5-8).

```
        Executing at remote database oracle:
        select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
20ms    from   dstreamaf
        where  10000000 < total_sales
        and    currency in (US)
```

**Figure 5-6 Query Execution Trace for Executing Database CSQ**

```
         Executing Remote Cameleon Query:
290ms    select exchanged, expressed, rate, date from olsen where exchanged="DEM" and expressed="USD"
         and date="01/05/94"
```

**Figure 5-7 Query Execution Trace for Executing Cameleon CSQ**

```
        Executing Remote Functional Query:
10ms    http://avocado.mit.edu/servlet/edu.mit.gcms.demo.servlets.datexform?function=datexform&
        format1=European+Style+-&date2=01%2F05%2F94&format2=American+Style+%2F
```

**Figure 5-8 Query Execution Trace for Exeucting Functional CSQ**

The last stage of the query execution trace shows the final query sent to the database for execution. The final query performs relational operations on the temporary tables containing data fetched from remote sources. Figure 5-9 below shows the final query in the execution of motivational example query.

```
        Database Execution of Final Query:
        select TTP_name_map_dt_ws.ws_names, TTP_dstreamaf.total_sales,
        TTP_worldcaf.latest_annual_financial_date, TTP_worldcaf.total_assets
        from   TTP_datexform, TTP_name_map_dt_ws, TTP_dstreamaf, TTP_currency_map, TTP_worldcaf
        where  TTP_dstreamaf.currency = TTP_currency_map.char2_currency
        and    TTP_name_map_dt_ws.ws_names = TTP_worldcaf.company_name
30ms    and    TTP_datexform.date1 = TTP_dstreamaf.as_of_date
        and    TTP_name_map_dt_ws.dt_names = TTP_dstreamaf.name
        union
        select TTP_name_map_dt_ws2.ws_names, TTP_dstreamaf2.total_sales*TTP_olsen.rate,
        TTP_worldcaf2.latest_annual_financial_date, TTP_worldcaf2.total_assets
        from   TTP_datexform2, TTP_name_map_dt_ws2, TTP_dstreamaf2, TTP_currency_map2, TTP_olsen,
        TTP_worldcaf2
        where  TTP_name_map_dt_ws2.ws_names = TTP_worldcaf2.company_name
        and    TTP_datexform2.date1 = TTP_dstreamaf2.as_of_date
        and    TTP_name_map_dt_ws2.dt_names = TTP_dstreamaf2.name
        and    TTP_dstreamaf2.currency = TTP_currency_map2.char2_currency
        and    TTP_currency_map2.char3_currency = TTP_olsen.exchanged
        and    10000000 < TTP_dstreamaf2.total_sales*TTP_olsen.rate
```

**Figure 5-9 Query Execution Trace for the Final Query of Motivational Example**

# 6 Planner & Optimizer

The challenge for the Planner and Optimizer components is to create a plan for efficiently bringing data from remote data sources into the local data store while satisfying restrictions on the kinds of queries that the remote sources can answer. The Planner and Optimizer generate a query execution plan (QEP), which specifies CSQ dependencies and joins that need to be performed. Additionally, the running time of QEP generation algorithm must not be exponential because of the complexity of queries involved.

I present Planner and Optimizer components together because of my approach to the query optimization. I do not first build a feasible plan and then optimize it by making changes to feasible plan. Instead, I build optimized QEP recursively from scratch by using greedy algorithm that relies on cost statistics.

I start of the chapter with the discussion of query restrictions. In section 6.2, I explain the format of query execution plan, and in section 6.3 I develop the QEP generation algorithm.

## 6.1 Query Restrictions

Due to the nature of some of the data sources in the COIN system (e.g. non-relational web sources) the wrappers cannot provide the full relational capability. Therefore, restrictions are placed on the nature of the queries, which can be dispatched to the wrappers. For example, the relation olsen requires that the attributes Expressed, Exchanged, and Date be bound whenever a query is sent to it. It also requires that the attribute Rate be free. A capability record is used to describe query restrictions for a relation. My capability records are very similar to those described in [16] by Koffi Fynn in that I create separate records for the binding restrictions and the operator restrictions of the source.

Figure 6-1 below shows the schema of olsen relation together with its capability record. Data in olsen relation is accessible through Cameleon web wrapper.

```
relation(cameleon,
        olsen,
        [['Exchanged',string],['Expressed',string],['Rate',number], ['Date',string]],
        cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>=']).
```

**Figure 6-1 Capability Record for olsen Relation**

Let us examine the the capability record of olsen relation cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>=']). The first part of the capability record [[b(1),b(1),f,b(1)]] specifies binding restrictions and the second part ['<','>','<>','<=','>='] specifies operator restrictions.

Binding restrictions is a list of all possible binding combinations of the attributes in the relation. A binding combination specifies attributes that need to be bound, attributes that need to be free, and attributes that can be either free or bound. It is represented with a list of binding specifiers for each of the attributes in the relation. A binding specifier can be one of the following: b, b(N), f, and ?. b indicates that the attribute has to be bound. b(N) indicates that the attribute has to be bound with N keys-at-a-time binding restriction. f indicates that the attribute must be free. ? indicates that the attribute can be either bound or free. The record for operator restrictions is a list of the operators, which cannot be used in queries on the relation.

Note that key-at-a time restrictions are quite common among the web wrapped relations. The olsen query I already described can only bind one key at a time for its attributes Exchanged, Expressed, and Date. Key-at-a-time restrictions that can bind more than key at a time (N>1) are also common. A good example of this is a stock quote server like finance.yahoo.com, which allows up to 50 stock quote symbols to be entered at one time.

The capability record in Figure 6-1 can be read as follows:

- attributes Exchanged, Expressed, and Date have 1 key-at-time binding restriction
- attribute Date needs to be free
- comparators '<', '>', '<>', '<=', and '>=' cannot be used in the query

## 6.2 Query Execution Plan

In this section, I explain the format of the query execution plan (QEP). My QEP differs from conventional ones for single databases. The main difference is that it operates at a higher level and does not specify complete ordering of relational operations. Instead, it specifies the order in which CSQs need to be executed. The reason why my QEP is different is the unique features of the Executioner, which is capable of performing relational operations on locally stored data. Because the Executioner can perform relational operation locally on the data from CSQs, the information we need from QEP is not how to perform relational operations but how to bring CSQ data from remote sources into the local data storage.

| **Necessary CSQs** | **Dependent CSQs** |
|---|---|
| 0: | currency_map, datexform, name_map_dt_ws2, datexform2 |
| currency_map: | DStreamAF ((currency currency_map.char2_currency)) |
| DStreamAF: | name_map_dt_ws ((dt_names DStreamAF.name)) |
| name_map_dt_ws | worldcaf ((company_name name_map_dt_ws.ws_names)) |
| name_map_dt_ws2: | worldcaf2 ((company_name name_map_dt_ws2.ws_names)), DStreamAF2 ((name name_map_dt_ws2.dt_names)) |
| DStreamAF2: | currency_map2 ((char2_currency DStreamAF2.currency)) |
| currency_map2: | olsen ((exchanged (currency_map2 char3_currency)) |

**Figure 6-2 Query Execution Plan for Motivational Example Query**

Let us examine the format of the query execution plan. Figure 6-2 above shows the QEP generated for the query in our motivational example. QEP has two columns: necessary CSQs and dependent CSQs. The CSQs in the Dependent column depend on CSQs in the Necessary column. For example, the second row of QEP tells us that DStreamAF CSQ depends on the execution of currency_map CSQ. This means that DStreamAF CSQ can only be executed once the execution of Currency_Map CSQ is completed. In addition, DStreamAF ((currency currency_map.char2_currency)) specifies that currency attribute of DStreamAF needs to be joined remotely with char2_currency of currency_map[17].

The first row of the QEP is different from others because it contains CSQs that do not depend on any other CSQs. 0: currency_map, datexform, currency_map2, datexform2 means that CSQs currency_map, datexform, currency_map2, and datexform2 do not have to wait for execution of any CSQs to complete. At run time, these four CSQs are executed in parallel.

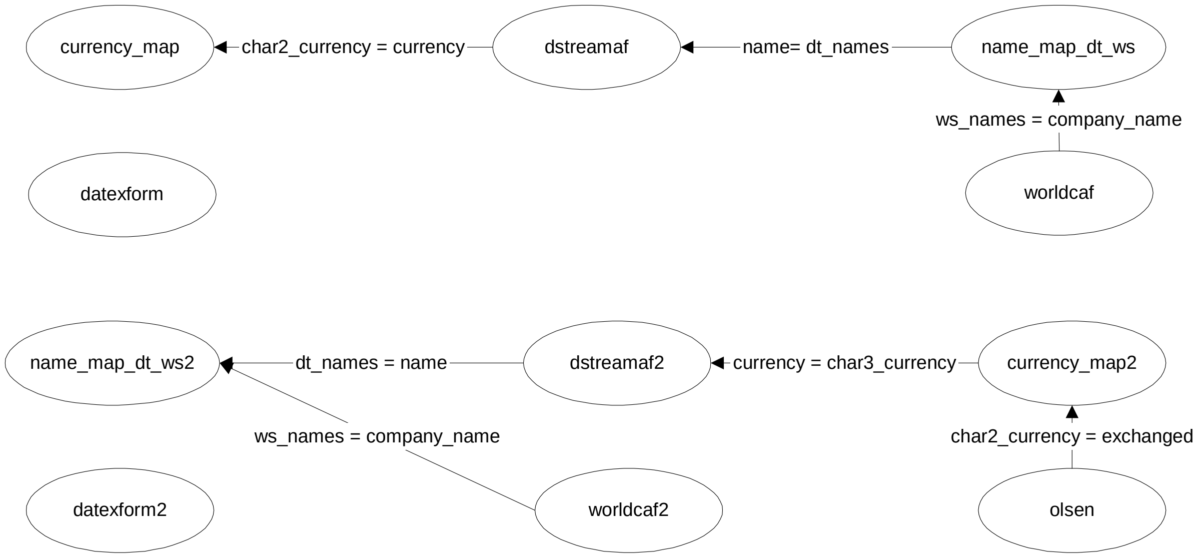


**Figure 6-3 Graph Representation of QEP**

Notice that the QEP has a form of a graph of CSQ dependencies. Figure 6-3 shows a graph representation of QEP where nodes represent CSQs, and arrows dependency conditions between CSQs. Names on the arrows are names of attributes of two CSQs that are being joined. We read the graph as follows:

- DStreamAF CSQ is joined remotely with currency_map CSQ. char2_currency attribute of currency_map supplies join binding for currency attribute of DStreamAF
- Name_map_dt_ws CSQ is joined remotely with DStreamAF CSQ. name attribute of DStreamAF supplies join bindings for dt_names attribute of Name_map_dt_ws
- Wolrdcaf CSQ is joined remotely with Name_map_dt_ws CSQ. ws_names attribute of Name_map_dt_ws supplies join bindings for company_name attribute of Wolrdcaf

The reason why QEP only specifies dependencies between CSQs and not the complete sequential ordering of CSQs like in Koffi Fynn POE [16] is that the Executioner parallelizes execution of CSQs whenever possible. The Executioner spins off a separate thread for execution of each CSQ[18] and because the running time of CSQ threads cannot be known until the run-time, it is not possible to determine what CSQs are executed in

[17] Section 5.4 shows how remote joins are performed.
[18] See Section 5.3.

parallel prior to run-time. Therefore, QEP only specifies CSQ dependencies and Executioner starts execution of a CSQ once all the CSQs it depends on have been executed.

## 6.3 QEP Generation Algorithm

In this section, I present the algorithm for generating the query execution plan. As mentioned in previous sections, the main task of Planner and Optimizer is to find a way to bring relevant data from the CSQs into the local RDBMS engine in the least amount of time.

In order to get a feel for the quality of optimizations, I assume a cost per retrieved tuple of 10ms for database relations in the motivational example query (name_map_dt_ws, DStreamAF, currency_map, worldcaf)[19], cost of 100ms per retrieved tuple for remote functional conversion datexform, and cost of 300ms per retrieved tuple for Cameleon relation olsen. These are approximate costs observed while executing above query on a sample setup of COIN system (see Appendix H).

I use the motivational example query to illustrate steps of QEP generation algorithm and demonstrate where optimizations take place. Since the optimization algorithm is complex, I first consider the trivial version of the algorithm in section 6.3.1. I then add parallelization in section 6.3.2, handling query restrictions in sections 6.3.3 and 6.3.4, and cost-based optimizations in section 6.3.5. Finally, section 6.3.6 shows how to handle key-at-a-time restrictions at run-time.

### 6.3.1 Trivial Algorithm

Let us take a look once again at Single Query 1 of our motivational example:

[19] Database in the experimental setup runs on the same machine as COIN system.

```
select name_map_dt_ws.ws_names, DStreamAF.total_sales, worldcaf.latest_annual_financial_date,
         worldcaf.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
          from   datexform
          where  format1='European Style -'
          and    date2='01/05/94'
          and    format2='American Style /') datexform,
         (select dt_names, ws_names
          from   name_map_dt_ws) name_map_dt_ws,
         (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
          from   DStreamAF
          where  10000000 < total_sales) DStreamAF,
        (select 'USD', char2_currency
         from   currency_map
         where  char3_currency='USD') currency_map,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
                  net_income, sales, total_assets, country_of_incorp
       from   worldcaf) worldcaf
where  DStreamAF.currency = currency_map.char2_currency
and    name_map_dt_ws.ws_names = worldcaf.company_name
and    datexform.date1 = DStreamAF.as_of_date
and    name_map_dt_ws.dt_names = DStreamAF.name
and    10000000 < DStreamAF.total_sales
```

**Figure 6-4 SQL Corresponding to Single Query 1 of Motivational Example**

The above query consists of five distinct CSQs: datexform, name_map_dt_ws, DStreamAF, currency_map, and worldcaf. The most straightforward way for fetching the data from remote CSQs is to simply execute all five CSQs in sequence and bring the data into local data storage. This approach works for this query because all CSQs except datexform have no capability restrictions. Moreover, binding restrictions for datexform are satisfied because format1, date2, and format2 attributes are bound and thus satisfy restrictions of datexform's capability record [f, b(1), b(1), b(1)].

Since all CSQs are independently executable – that is they do not depend on the execution of each other – the QEP for execution of the above query can be written as follows:

| **Necessary CSQs** | **Dependent CSQs** |
|---|---|
| 0: | datexform, name_map_dt_ws, DStreamAF, currency_map, worldcaf |

Notice, however, that the above QEP is inefficient. The reason is that Executioner is transferring total of 623 tuples across the network as shown in the Figure 6-5 below.

```
    (select date1, 'European Style -', '01/05/94', 'American Style /'
      from   datexform
      where  format1='European Style -'
      and    date2='01/05/94'
      and    format2='American Style /') datexform
      NUMBER OF TUPLES: 1

     (select dt_names, ws_names
      from   name_map_dt_ws) name_map_dt_ws
      NUMBER OF TUPLES: 7


     (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary,
     currency
     from   DStreamAF
     where  10000000 < total_sales)
     NUMBER OF TUPLES: 266

     (select 'USD', char2_currency
     from   currency_map
     where  char3_currency='USD') currency_map
     NUMBER OF TUPLES: 1

     (select company_name, latest_annual_financial_date, current_outstanding_shares,
             net_income, sales, total_assets, country_of_incorp
     from   worldcaf) worldcaf
     NUMBER OF TUPLES: 354
```

**Figure 6-5 Number of Tuples in Single Query 1 of Motivational Example**

As Figure 6-6 shows, the trivial version of the algorithm takes a total of 622x10ms + 100ms= 6,320 ms.

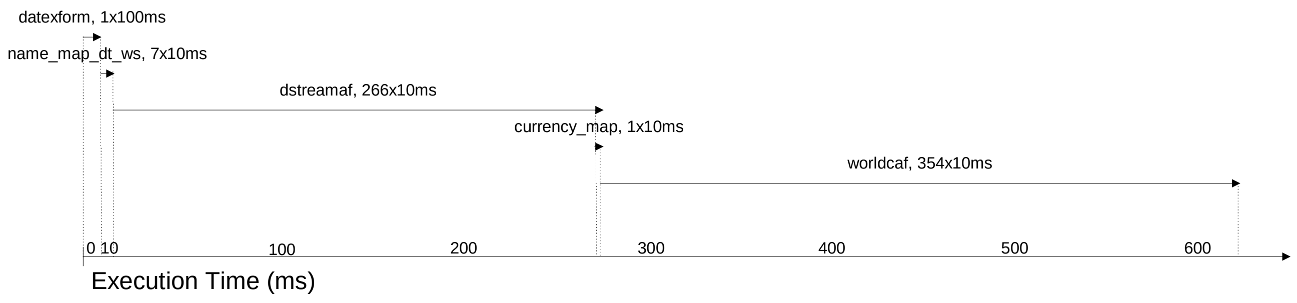


**Figure 6-6 Time Graph of Execution of Single Query 1 Using Trivial Algorithm**

## 6.3.2 Parallel Query Execution

A significant improvement to the trivial algorithm can be achieved by executing the queries in the parallel. This is possible for the QEP in the previous section of Single Query 1 because all 5 component subqueries are independently executable. Adding parallelization at the execution stage does not change QEP because QEP only captures the dependencies between CSQs. However, the execution time of query changes because all five CSQs are executed in parallel.

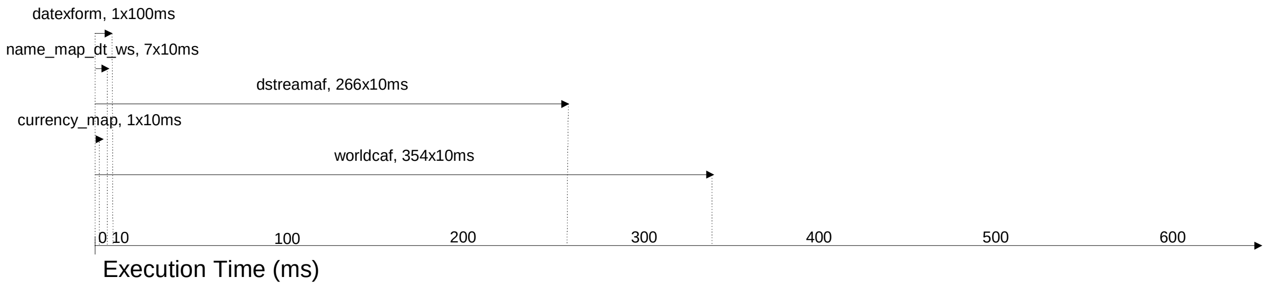


**Figure 6-7 Time Graph of Execution of Single Query 1 Using Trivial Algorithm with Parallelization**

As Figure 6-7 shows, parallelization cuts query execution time almost in half. Calculation is shown below:

$$\boxed{\begin{aligned} T &= \text{Max}(\text{datexform}_t, \text{name\_map\_dt\_ws}_t, \text{DStreamAF}_t, \text{currency\_map}_t, \text{worldcaf}_t) \\ &= \text{Max}(1\text{x}100\text{ms}, 1\text{x}10\text{ms}, 255\text{x}10\text{ms}, 1\text{x}10\text{ms}, 354\text{x}10\text{ms}) \\ &= 354\text{x}10\text{ms} \\ &= 3{,}540 \text{ ms} \end{aligned}}$$

### 6.3.3 Handling Query Binding Restrictions

Before we consider further optimization improvements, we need to make sure that queries can execute correctly with respect to query restrictions of data sources. As I mentioned in the explanation of the trivial algorithm, the reason why the trivial algorithm works in the first place is that all CSQs in Single Query 1 are independently executable. This, however, is not the case for Single Query 2 shown in Figure 6-8.

```
select name_map_dt_ws2.ws_names, DStreamAF2.total_sales*olsen.rate,
         worldcaf2.latest_annual_financial_date, worldcaf2.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
         from   datexform
         where  format1='European Style -'
         and    date2='01/05/94'
         and    format2='American Style /') datexform2,
        (select dt_names, ws_names
         from   name_map_dt_ws) name_map_dt_ws2,
        (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
         from   DStreamAF) DStreamAF2,
        (select char3_currency, char2_currency
         from   currency_map
         where  char3_currency <> 'USD') currency_map2,
        (select exchanged, 'USD', rate, '01/05/94'
         from   olsen
         where  expressed='USD'
         and    date='01/05/94') olsen,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
               net_income, sales, total_assets, country_of_incorp
         from   worldcaf) worldcaf2
where  name_map_dt_ws2.ws_names = worldcaf2.company_name
and    datexform2.date1 = DStreamAF2.as_of_date
and    name_map_dt_ws2.dt_names = DStreamAF2.name
and    DStreamAF2.currency = currency_map2.char2_currency
and    currency_map2.char3_currency = olsen.exchanged
and    10000000 < DStreamAF2.total_sales*olsen.rate
and    currency_map2.char3_currency <> 'USD'
```

**Figure 6-8 SQL Corresponding to Single Query 2 of Motivational Example**

In Single Query 2, CSQ olsen is not independently executable. As shown in Figure 6-9, olsen's attribute exchanged is not bound and consequently the binding restriction [b(1), b(1), f, b(1)] is not satisfied. The trivial algorithm will not work for Single Query 2 because a required binding for olsen is not provided.

```
(select exchanged, 'USD', rate, '01/05/94'
 from    olsen
 where expressed='USD'
 and     date='01/05/94') olsen
```

**Figure 6-9 Binding for exchanged Attribute is olsen CSQ is Missing**

The query can still be executed by using join condition currency_map2.char3_currency = olsen.exchanged and supplying binding for olsen.exchanged from CSQ currency_map2. Figure 6-10 shows olsen CSQ modified so that it satisfies the binding requirement:

```
(select exchanged, 'USD', rate, '01/05/94'
 from   olsen
 where  expressed='USD'
 and    date='01/05/94'
 and exchanged in (select char3_currency from currency_map2)) olsen
```

**Figure 6-10 Olsen CSQ with Added Binding from Join Relation with currency_map2**

Notice, however, that CSQ currency_map2 now needs to be executed before olsen is executed. In other words, CSQ olsen depends on CSQ currency_map2. Figure 6-11 below shows QEP for Single Query 2 that handles query restrictions on olsen relation.

| Necessary CSQs | Dependent CSQs |
|---|---|
| 0: | currency_map2, datexform2, DStreamAF2, name_map_dt_ws2, worldcaf2 |
| currency_map2: | olsen ((exchanged (currency_map2.char3_currency)) |

**Figure 6-11 Olsen CSQ with Added Binding from Join Relation with currency_map2**

The first row tells us that CSQs currency_map2, datexform2, DStreamAF2, name_map_dt_ws2, worldcaf2 are independently executable. Second row tells us that CSQ olsen needs to wait for the execution of CSQ currency_map2 to complete. It also specifies that olsen needs to bind its attribute exchanged to the attribute char3_currency of currency_map2.

I now present the general QEP algorithm that handles binding conditions on CSQs.

```
Input: Single Query q
Output: Query Execution Plan (QEP)

QEP Generation Algorithm:
1.  initialize set S to an empty set
2.  for all CSQs c in S
3.      if c is independently executable
4.      add c to set S
5.      add entry 0: c to QEP
6.  repeat until no more CSQs are added to S
7.      for all CSQs c outside of S
8.          if CSQ c can be executed using bindings from CSQs in S
9.              add an entry for c to QEP including all join bindings of c
10.             add CSQ c to set S
11.  if S does not contain all CSQs in a query
12.      throw exception "query cannot be executed"
13. return QEP
```

**Figure 6-12 QEP Generation Algorithm Supporting Binding Query Restrictions**

There are two non-trivial steps in the above algorithm: step 3 and step 8. In step 3 we compute all independently executable CSQs, and in step 8 we determine whether a CSQ c can be executed using join bindings from CSQs in set S. I will present detailed algorithms for steps 3 and 8 shortly, but let us first walk through the execution of QEP generation algorithm for Single Query 2 and convince ourselves that it generates a feasible query execution plan.

Steps 1-5: Finds all independently executable CSQs in Single Query 2: currency_map2, datexform2, DStreamAF2, name_map_dt_ws2, worldcaf2 and adds them to the first row of QEP

Steps 6-10: CSQ olsen is the only CSQ in Single Query 2 that is not independently executable. In step 8, algorithm discovers that olsen CSQ can be executed by binding olsen's exchanged attribute to char3_currency attribute of independently executable CSQ currency_map2.

Steps 11-13: Algorithm finds that all CSQs in Single Query 2 have been covered and returns the resulting QEP.

Now, let us analyze the efficiency of generated QEP. As Figure 6-13 below shows, CSQs currency_map2, datexform2, DStreamAF2, name_map_dt_ws2, worldcaf2 are executed in parallel. CSQ olsen is dependent on CSQ currency_map2 and needs to wait until execution of currency_map2 to complete. Total running time is 3,540 ms dominated by 354 tuples of CSQ worldcaf.

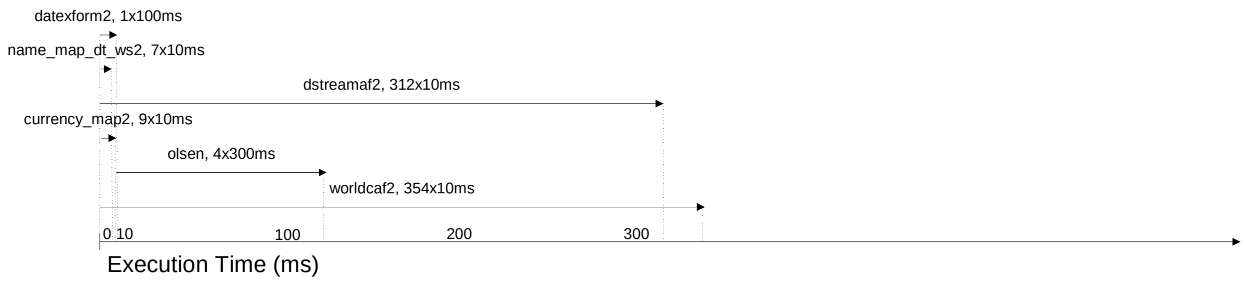


**Figure 6-13 Time Graph of Execution of Single Query 2 Using QEP Generation Algorithm Supporting Binding Restrictions**

For comparison, I also present a time graph for execution of Single Query 2 where CSQs are executed sequentially. Figure 6-14 shows that sequential version of the algorithm takes almost three time as much time as the paralellized version of the algorithm supporting binding restrictions.

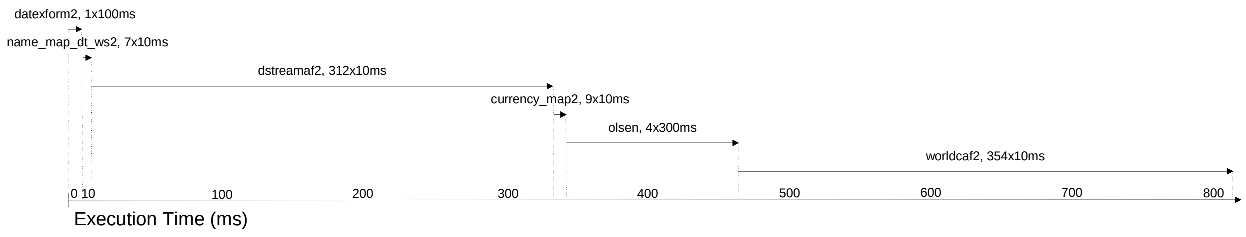


**Figure 6-14 Time Graph of Execution of Single Query 2 Where CSQs are Executed Sequentially**

Now, we address the non-trivial steps of QEP generation algorithm. I first present the algorithm for finding independently executable CSQs and then the algorithm for determining whether a CSQ c can be executed using join bindings from a set of CSQs S.

### 6.3.3.1 Determining whether CSQ is Independently Executable

Figure 6-15 shows the algorithm for determining whether a CSQ c is independently executable.

```
Input: CSQ c
Output: true if CSQ c is independently executable
          false otherwise

Independently Executable CSQ:
1. for all binding specifiers bs of c's underlying relation r
2.     for all attribute specifiers as of bs
3.          if as is of type bound and there is no binding in CSQ c for corresponding attribute
4.               continue 1
5.          else
6.               continue 2
7.      return true
8.  return false
```

**Figure 6-15 Algorithm for Determining Whether a CSQ is Independently Executable**

I now show how the algorithm works for independently executable CSQs currency_map2 and olsen of Single Query 2. Schemas and capability records for relations of these two CSQs is shown in Figure 6-16 below.

```
relation(oracle,
           'currency_map',
           [['CHAR3_CURRENCY',string],['CHAR2_CURRENCY',string]],
            cap([[?,?]], [])).


relation(cameleon,
           olsen,
           [['Exchanged',string],['Expressed',string],['Rate',number],
            ['Date',string]],
           cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>=']).
```

**Figure 6-16 Schemas and Capability Records for Relations olsen and currency_map**

The currrency_map has only one binding specifier [?, ?]. Since the ? attribute specifier is not binding, the if condition in step 3 leads to execution of else statement in step 6 for both attribute specifiers. It follows, that algorithm reaches step 7 and returns true. Therefore, the algorithm correctly determined that currency_map2 CSQ is independently executable.

```
 (select exchanged, 'USD', rate, '01/05/94'
  from   olsen
  where  expressed='USD'
  and    date='01/05/94') olsen
```

**Figure 6-17 Binding for exchanged Attribute in olsen CSQ is missing**

olsen also has only one binding specifier [b(1),b(1),f,b(1)]. exchanged attribute is the first attribute of olsen relations and according to capability record it needs to be bound.

However, as Figure 6-17 shows, attribute exchanged is not bound. The algorithm detects missing binding in step 3 and looks whether binding conditions can be satisfied for another binding specifier. However, since [b(1),b(1),f,b(1)] was the only binding specifier for olsen, the algorithm exits the for loop and returns false. Therefore, the algorithm correctly determined that olsen CSQ is not independently executable.

### 6.3.3.2 Determining whether CSQ is executable given a set of executed CSQs

Figure 6-18 shows the algorithm for determining whether a CSQ c is executable given a set of already executed CSQs S.

```
Input:    set of executed CSQs S, new CSQ n
Output:  if n cannot be executed given join bindings from CSQs in S
              returns null
           else
              returns list of join bindings for CSQ n

CSQ Executable:
1.  for all binding specifiers bs of CSQ n
2.      initialize list of join bindings to an empty list jbl
3.      for all attribute specifiers as of bs
4.          if as is of type bound and CSQ n does not contain binding for attribute matching as
5.              if there is a join binding jb from n's attribute matching as to one of CSQs in S
6.              add jb to jbl
7.              continue 3
8.          else
9.              continue 1
10.     return jbl
11. return null
```

**Figure 6-18 Algorithm for Determining Whether a CSQ is Executable Given a Set of Executed CSQs**

olsen also has only one binding specifier [b(1),b(1),f,b(1)]. In the first iteration of the for loop in step 3, the algorithm discovers that attribute exchanged has a join binding to attribute char3_currency of currency_map2 (see Figure 6-19 showing all join bindings in Single Query 2). In the next iterations of for loop in step 3, the algorithm finds no new join bindings because the attribute rate is free and the attributes expressed and date are already bound within olsen CSQ. Since CSQ olsen can be executed given join binding for attribute exchanged, the for loop in step 3 finishes execution and returns join binding list in step 10. The join binding list contains only one join binding – the join binding between olsen's attribute exchanged and currency_map2's attribute char3_currency.

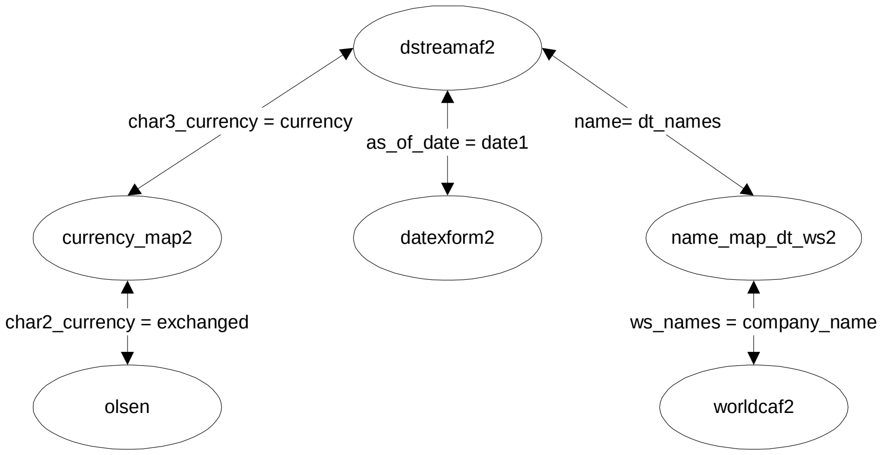


**Figure 6-19 Join Bindings in Single Query 2**

### 6.3.4 Handling Query Operator Restrictions

In this section, we discuss how to handle query operator restrictions. Figure 6-20 shows olsen CSQ of Single Query 2 with added binding from join relation with currency_map2 CSQ.

```
(select exchanged, 'USD', rate, '01/05/94'
 from   olsen
 where  expressed='USD'
 and    date='01/05/94'
 and exchanged in (select char3_currency from currency_map2)) olsen
```

**Figure 6-20 Olsen CSQ with Added Binding from Join Relation with currency_map2**

As capability record for olsen cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>=']) indicates, the olsen CSQ cannot handle operators '<', '>', '<>', '<=', and '>='. Since CSQ olsen in Figure 6-20 does not contain any of the restricted operators the query needs not be modified. Now, imagine that we are only interested in exchange rates greater than 1.0. Modified olsen CSQ (see Figure 6-21) cannot be executed. Condition rate>1.0 violates the operator query restriction.

```
(select exchanged, 'USD', rate, '01/05/94'
 from   olsen
 where  expressed='USD'
 and    date='01/05/94'
 and exchanged in (select char3_currency from currency_map2)
 and rate>1.0) olsen
```

**Figure 6-21 Olsen CSQ with Added Condition rate>1.0**

The solution to the problem of operator restrictions is straightforward. We execute the CSQ without violating condition and add the violating condition to the final query sent to Executioner's database engine. In our case, the violating condition olsen.rate>1.0 would be removed from olsen CSQ and added to the final where clause of Single Query. The condition would be performed because the final where clause is carried out in the local data store using RDBMS engine (see Figure 6-22).

```
select name_map_dt_ws2.ws_names, DStreamAF2.total_sales*olsen.rate,
         worldcaf2.latest_annual_financial_date, worldcaf2.total_assets
from   (select date1, 'European Style -', '01/05/94', 'American Style /'
         from   datexform
         where  format1='European Style -'
         and    date2='01/05/94'
         and    format2='American Style /') datexform2,
        (select dt_names, ws_names
         from   name_map_dt_ws) name_map_dt_ws2,
        (select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency
         from   DStreamAF) DStreamAF2,
        (select char3_currency, char2_currency
         from   currency_map
         where  char3_currency <> 'USD') currency_map2,
        (select exchanged, 'USD', rate, '01/05/94'
         from   olsen
         where  expressed='USD'
         and    date='01/05/94'
         and exchanged in (select char3_currency from currency_map2)) olsen
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
              net_income, sales, total_assets, country_of_incorp
         from   worldcaf) worldcaf2
where name_map_dt_ws2.ws_names = worldcaf2.company_name
and    datexform2.date1 = DStreamAF2.as_of_date
and    name_map_dt_ws2.dt_names = DStreamAF2.name
and    DStreamAF2.currency = currency_map2.char2_currency
and    currency_map2.char3_currency = olsen.exchanged
and    10000000 < DStreamAF2.total_sales*olsen.rate
and    currency_map2.char3_currency <> 'USD'
and    olsen.rate>1.0
```

**Figure 6-22 SQL of Modified Single Query 2 Illustrating Solution for Operator Restrictions Problem**

### 6.3.5 Cost Optimized QEP Generation Algorithm

Query optimizations presented in this section do not rely on source provided cost statistics for producing reasonable plans. However, cost statistics provided by sources can be used to improve on reasonable query execution plans.

The reason why I decided not to rely on the existence of reliable costs statistics from the sources are based on the following observations in the COIN system:

- new date sources may be added at any time
- sources of information are heterogeneous and do not subscribe to any particular cost estimation model
- administrators of data sources cannot be mandated to provide cost statistics for an arbitrarily chosen cost model

I start the following section with discussion of cost model, continue with describing methods for gathering cost statistics, and finish with presenting final cost-optimized QEP generation algorithm that uses remote joins.

#### 6.3.5.1 Cost Model

Total Cost= Query processing time + Communication time

Query processing time depends on the performance of the source node and its current load. Communication time depends on the location of source node and load on the network. Notice that both cost components are dynamic and change with time. Therefore, we need a flexible cost model that can estimate cost in real time.

Query processing time is hard to estimate in heterogeneous settings because of diversity of data sources. It depends on performance of the source node and its load. To get around the problem of estimating query processing cost, a commonly made assumption in optimizers of distributed heterogeneous sources is that communication cost dominates the overall cost of the query. This assumption is justified in the case of COIN because data sources are widely dispersed.

Communication time = Number of CSQ Tuples Transmitted * Average Time per CSQ Tuple

Communication time is equal to product of number of tuples transmitted over the network and the average time per transmitted tuple. Average Time per CSQ Tuple captures physical constraints, such as, round-trip time and load on the network.
In the next section on cost statistics generation, I show how to estimate both number of tuples transmitted and average time per tuple.

Using our assumption that communication time dominates query processing time, we arrive at the following simplified cost model:

Total Cost = Number of CSQ Tuples Transmitted * Average Time per CSQ Tuple

Since average time per CSQ tuple captures physical constraints that we do not have control of, it follows that we can minimize the total cost of executing the query by minimizing the number of tuples transmitted over the network. We show how to do that in section 6.3.5.3.

#### 6.3.5.2 Cost Statistics Generation

We need to generate two kinds of cost statistics: (1) average time per retrieved CSQ tuple, and (2) estimated number of tuples returned by execution of a CSQ.

Average time per retrieved CSQ can be estimated by keeping time statistics of previously executed CSQs on the same underlying relation. The estimation process can be initiated by starting with a conservative default time estimate and then improving on it exponentially using time statistics on recently executed CSQs on the same underlying relation.

Number of tuples returned by execution of a CSQ is harder to estimate because number of tuples varies dramatically by changing conditions on CSQ. For example, query select * from DStreamAF where name= 'DAIMLER-BENZ' returns one tuple and query select * from DStreamAF returns 312 tuples. One possible approach to this problem is to require sources to provide cost statistics as is done in IBM's Garlic system. However, as stated in the design goals, COIN cannot rely on source to provide cost statistics.

In order to estimate number of tuples in CSQs of database sources, I use the SQL supported count operator. We can obtain cardinalities of two queries mentioned in previous paragraph by performing following single tuple queries: select count(*) from DStreamAF where name= 'DAIMLER-BENZ' and select count(*) from DStreamAF. Notice that cardinalities obtained by these queries can be cached and thus save one round-trip time.

In general case, number of tuples in CSQs of non-relational sources cannot be obtained because standard count(*) operator is not supported and we can only obtain source provided cost statistics[20]. However, in line with my assumption of non-reliance on source cost statistics, I do not require sources to provide cost statistics; the QEP generation algorithm presented in the next section still produces reasonable albeit not optimal plans. Nevertheless, non-relational sources may optionally provide information on cardinalities of their relations. Cardinalities of relations can then be used to estimate number of tuples in CSQs using query sampling method [20].

#### 6.3.5.3 Cost Optimized QEP Generation Algorithm Using Remote Joins

The optimization in this section is made possible by the following key observation: Joining two relations with their primary keys equal results in a relation with number of tuples of at most the number of tuples in a smaller of two relations. I illustrate this observation on a concrete example by joining relations worldcaf and name_map_dt_ws (see Figure 6-23).

```
select  *
from   (select dt_names, ws_names
        from   name_map_dt_ws) name_map_dt_ws,
        (select company_name, latest_annual_financial_date, current_outstanding_shares,
                net_income, sales, total_assets, country_of_incorp
         from   worldcaf) worldcaf
where  name_map_dt_ws.ws_names = worldcaf.company_name;
```

**Figure 6-23 Join of worldcaf and name_map_dt_ws Relations at their Primary Keys**

[20] Note that in certain cases cardinalities of CSQ result sets can be calculated for non-relational sources using binding source restrictions and integrity constraints. For example, if Cameleon query has a key-at-a-time binding restriction on a unique key of a relation, we can infer that exactly one tuple will be returned by the query.

As shown in Figures 6-24 and 6-25, relation worldcaf contains 354 tuples and relation name_map_dt_ws 7 tuples.

| **worldcaf.company_name** | **…** |
|---|---|
| ABBOTT LABORATORIES | … |
| ACKLANDS LIMITED | … |
| ADVANCED MICRO DEVICES, INC. | … |
| AETNA LIFE & CASUALTY COMPANY | … |
| AIR EXPRESS INTERNATIONAL CORP | … |
| AIR PRODUCTS AND CHEMICALS, IN | … |
| AIRBORNE FREIGHT CORPORATION | … |
| ALASKA AIR GROUP, INC. | … |
| ALBERTSON'S, INCORPORATED | … |
| … (**TOTAL of 354 tuples**) | … |

**Figure 6-24 Tuples of worldcaf Relation**

| **name_map_dt_ws.ws_names** | **…** |
|---|---|
| BRITISH TELECOMMUNICATIONS PLC | … |
| DAIMLER-BENZ AG | … |
| ITOCHU CORPORATION | … |
| LYONNAISE DES EAUX SA | … |
| NIPPON TELEGRAPH & TELEPHONE C | … |
| SUMITOMO CORPORATION | … |
| TOMEN CORPORATION | … |

**Figure 6-25 Tuples of** name_map_dt_ws **Relation**

| **name_map_dt_ws.ws_names=worldcaf.company_name** | **…** |
|---|---|
| BRITISH TELECOMMUNICATIONS PLC | … |
| DAIMLER-BENZ AG | … |
| ITOCHU CORPORATION | … |
| LYONNAISE DES EAUX SA | … |
| NIPPON TELEGRAPH & TELEPHONE C | … |
| SUMITOMO CORPORATION | … |
| TOMEN CORPORATION | … |

**Figure 6-26 Tuples of Join of worldcaf and name_map_dt_ws Relations at their Primary Keys**

As shown in Figure 6.26, the join of two relations at their primary keys (name_map_dt_ws.ws_names = worldcaf.company_name), results in only 7 tuples – the number of tuples in a smaller of two relations. This is an important property because it allows us to reduce the number of tuples transferred over the network by performing joins remotely. Instead of brining 354 tuples of worldcaf and 7 tuples of name_map_dt_ws into the local datastore, we can reduce the number of transferred tuples to only 21 as I now show.

Step 1 – Bring tuples from name_map_dt_ws into the local store by running following query:

```
select *
from   name_map_dt_ws;
 (7 tuples transferred over the network to retrieve the query results)
```

Step 2 – Do remote join of name_map_dt_ws and worldcaf at the source of worldcaf by running following query:

```
select  *
from   worldcaf
where company_name in ('BRITISH TELECOMMUNICATIONS PLC', 'DAIMLER-BENZ AG',
                       'ITOCHU CORPORATION', 'LYONNAISE DES EAUX SA',
                       'NIPPON TELEGRAPH & TELEPHONE C',
                       'SUMITOMO CORPORATION', 'TOMEN CORPORATION');
(7 tuples transferred over the network to send the join attribute values of name_map_dt_ws)
(7 tuples transferred over the network to retrieve the query results)
```

Note that approach scales well as number of tuples in relations increase. If worldcaf relation had 1,000,000 instead of 354 rows, which is typical of many databases, the number of tuples transported over the network would reduce from 1,000,007 to only 21.

I generalized remote joins optimization approach and developed the following improved QEP generation algorithm:

```
Input: Single Query q
Output: Query Execution Plan (QEP)

QEP Generation Algorithm:
1.  initialize set A to an empty set
2.  for all CSQs c in A
3.      if c is independently executable
4.      add c to set A
5.  sort CSQs in set A by increasing estimated cardinality of tuples
6.  initialize set S to an empty set
7.  while A is not empty
8.      initialize set B to an empty set
9.      move first element of set A to set B
10.     repeat until no more CSQs are added to B
11.         for all CSQs c outside of B and S
12.             if CSQ c can be executed using bindings from CSQs in B
13.                 add an entry for c to QEP including all join bindings of c
14.                 add CSQ c to set B
15.     add all elements of set B to set S
16. if S does not contain all CSQs in a query
17.       throw exception "query cannot be executed"
18. return QEP
```

**Figure 6-27 Cost Optimized QEP Generation Algorithm Using Remote Joins**

The algorithm above looks complicated but it is not much different from the algorithm in section 6.3.3 where we already discussed steps 3 and 12 of the algorithm: determining whether CSQ is independently executable and determining whether CSQ is executable given a set of already executed CSQs.

There are two notable differences from the non-optimized version of the algorithm. First, the algorithm performs remote joins as soon as possible in order to minimize the transfer of tuples across the network. Second, in step 5, we use cost statistics to sort independently executable CSQs by increasing estimated cardinality of tuples. The algorithm produces QEP for Single Query 2 shown in Figure 6-28 in table form and Figure 6-29 in graph form.

| **Necessary CSQs** | **Dependent CSQs** |
|---|---|
| 0: | name_map_dt_ws2, datexform2 |
| name_map_dt_ws2: | worldcaf2 ((company_name name_map_dt_ws2.ws_names)), DStreamAF2 ((name name_map_dt_ws2.dt_names)) |
| DStreamAF2: | currency_map2 ((char2_currency DStreamAF2.currency)) |
| currency_map2: | olsen ((exchanged (currency_map2 char3_currency)) |

**Figure 6-28 Query Execution Plan for Single Query 2**

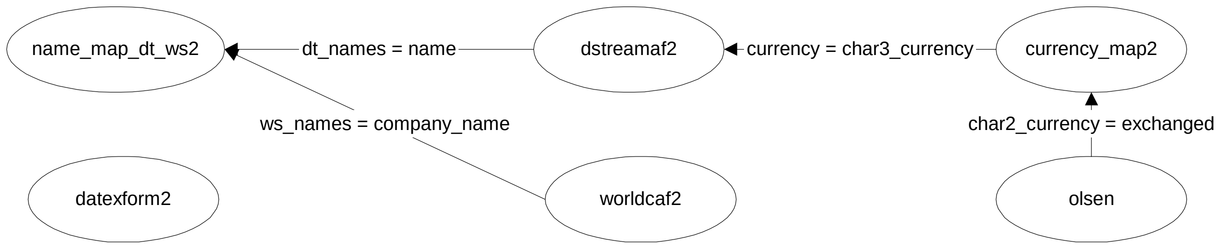
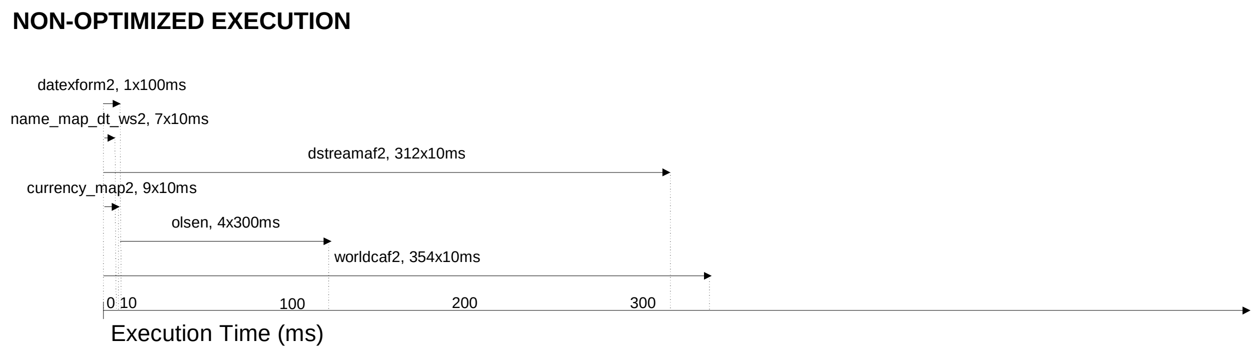


**Figure 6-29 Graph of QEP for Single Query 2**

Now, let us compare the execution of cost-optimized version of the algorithm presented in this section with the non-optimized version of section 6.3.3. As Figure 6-30 shows, cost optimization cuts the execution time by approximately a half. Note, however, that cost-optimization algorithm scales very well and with increasing number of tuples in relations, reductions in query execution time may be much larger. Now, let us examine the timing graphs of two algorithms in more detail. In non-optimized version of the algorithm, the query execution time is determined by the time it takes to retrieve tuples of two large CSQs – DStreamAF2 and WorldcAF2. Executioner retrieves 312 tuples from DStreamAF relation and 354 tuples from WolrdcAF relation. In the cost optimized version of the algorithm remote joins cut down the number of tuples from DStreamAF relation to 29 and number of tuples from WorldcAF relation to only 7. The limiting factor in the query execution time now becomes olsen relation which takes 1200ms to fetch 4 tuples.



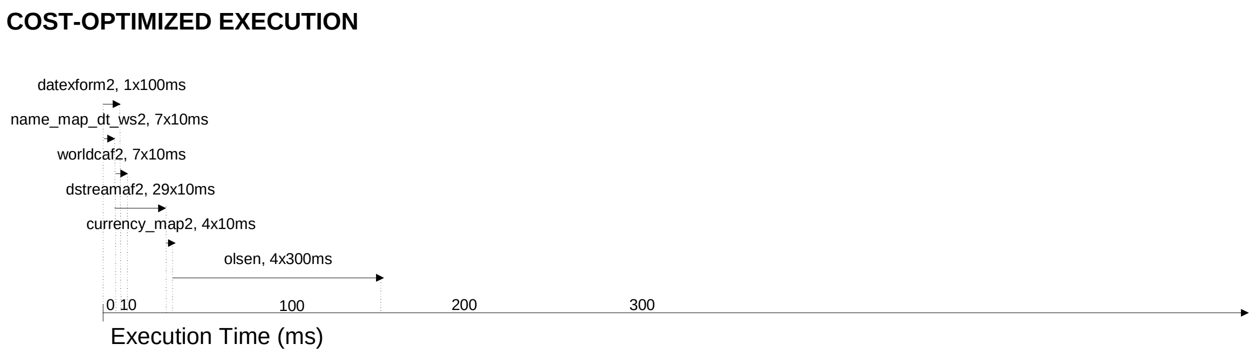


**Figure 6-30 Head to Head Timing Comparison Between Cost-Optimized Algorithm and Non-Optimized Algorithm for Execution of Single Query 2**

### 6.3.6 Handling Key-at-a-Time Query Restriction

The last hurdle we have to overcome in query execution is dealing with key-at-a-time query restriction. In our motivational example, both cameleon and functional container servlet source impose key-at-a-time restriction for its bindings.

Let us examine the challenge of key-at-a-time query restriction on a concrete example:

```
select price
from secapl
where ticker in (‘GE’, ‘IBM’, ‘MSFT’);
```

secapl is web-wrapped relation provided through cameleon interface and returns latest stock price for a company specified by ticker. Capability record for secapl relation is shown below:

```
relation(cameleon,
            ‘secapl’,
            [[‘TICKER’, string], [‘PRICE’, number]],
            cap([[b(1), f]], [‘<’, ‘>’, ‘<>’, ‘<=’, ‘>=’])
```

**Figure 6-31 Capability Record for secapl Relation**

Cameleon wrapper cannot handle this query directly because secapl relation has key-at-a-time query restrictions for the attribute ticker and cannot handle queries with 'in' operator[21]. In order to answer the query, we need to rewrite it as union of three key-at-a-time queries, and perform the union operation locally.

Our query becomes:

```
(select price
 from secapl
 where ticker= 'GE')
union
(select price
 from secapl
 where ticker= 'IBM')
union
(select price
 from secapl
 where ticker= 'MSFT')
```

Now, let us examine how to automate the task of query rewriting for handling key-at-a-time restriction. The input to the algorithm is CSQ c with some key-at-a-time restricted attribute. The output is union of simpler key-at-a-time queries whose union returns same result as original CSQ c.

```
KAT (key-at-a-time) rewriting algorithm:
1.  for every key-at-a-time-restricted attribute in CSQ
2.      if attribute is bound to more values than allowed by query restriction then
3.          rewrite the CSQ as union of 2 KAT queries:
4.              first KAT query with the attribute bound to the maximum allowed number of values
5.              second KAT query with the attribute bound to the remaining values
6.          call KAT algorithm recursively on both KAT queries
```

Let us examine in detail how KAT algorithm processes our original CSQ:

Step 1 – Original CSQ

```
select price
from secapl
where ticker in ('GE', 'IBM', 'MSFT');
```

Step 2 – CSQ gets decomposed into a union of two simpler queries

[21] Note that queries containing 'in' operator are common because when CSQ c1 is joined remotely with CSQ c2, executioner feeds results of CSQ c1 into CSQ c2 through 'in' operator (e.g. ticker in (select ticker from Ticker_Lookup)).

```
KAT Query 1: select price
                    from secapl
                    where ticker= (‘GE’)
                    union
KAT Query 2: select price
                    from secapl
                    where ticker in (‘IBM’, ‘MSFT’);
```

Step 3 – KAT Query 1 satisfies key-at-a-time restriction

Step 4 – KAT Query 2 gets decomposed into a union of two simpler queries

```
KAT Query 1: select price
                    from secapl
                    where ticker= (‘GE’)
                    union
KAT Query 2: select price
                    from secapl
                    where ticker= (‘IBM’)
                    union
KAT Query 3: select price
                    from secapl
                    where ticker= (‘MSFT’);
```

Step 5 – KAT Query 2 satisfies key-at-a-time restriction
Step 6 – KAT Query 3 satisfies key-at-a-time restriction

# 7 Integration with Non-Relational Data Sources

In this section, I discuss integration issues with non-relational data sources. In sections 7.1 and 7.2 I present the two most commonly encountered non-relational data sources: web wrappers and functional relations. I end the section by presenting a generic framework for classifying non-relational sources.

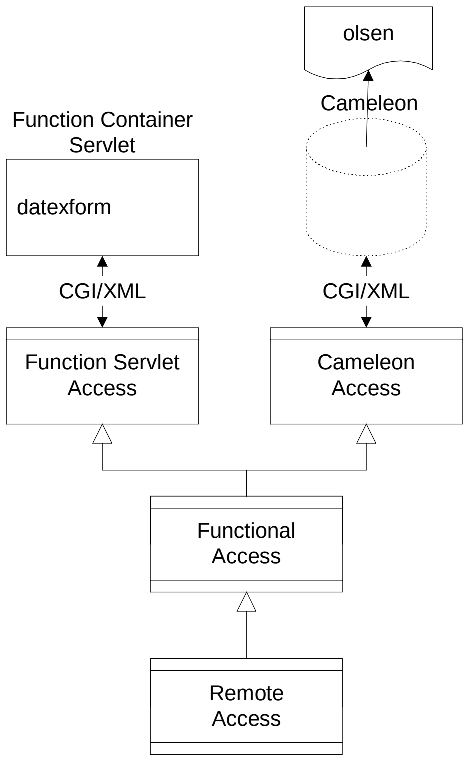


**Figure 7-1 Remote Access to Non-Relational Sources**

## 7.1 Web Wrapped Sources

The COIN system includes a web wrapper component – the Cameleon engine. Cameleon integrates data sources from multiple web sites under a unified SQL interface. It extracts data from web pages using declarative specification files that define extraction rules and it lets us wrap web sources and execute simple SQL queries against them. Query results by Cameleon are returned in several formats including XML.

As discussed in section 5.1, remote access to Cameleon is achieved through the Cameleon Access class, which uses CGI to send query parameters to the Cameleon engine and to get back the query results as XML. Figure 7-1 shows the Cameleon wrapped relation olsen, which provides historical currency conversion rates. Here is one of the queries executed remotely through Cameleon engine:

```
select  exchanged, expressed, rate, date
from    olsen
where exchanged="DEM"
and     expressed="USD"
and     date="01/05/94"
```

Even though the Cameleon web wrapper accepts the SQL syntax it is inherently non-relational because it only accepts functional queries subscribing to the parameters-in, results-out model. Note the functional form of the query with input parameters (exchanged, expressed and date) and output parameters (exchanged, expressed, rate, and date). Consequently, data in the olsen relation is accessed through functional access interface.

Cameleon returns query results in XML format.  Here is the XML returned when executing the above query on olsen relation:

```
<?>>>>>>>>>>>>>>>>>>>>?>>>
>><>>>>>>>>   >>
>><>>>>>>>> >>
 ><>>>> >>>>>  >///  <>>>>> >>>>>  >>>
 ><>>>>>>>>>>>///  <>>>>>>>>>>>>>
 ><>>>>>0/5//00 <>>>>>>>>
 ><>> >>>0//05///  <>>> >>>>>
 ><>>>>>>>>> >>
 ><>>>>>>>>>>   >>
```

## 7.2 Functional Relations

Functional relations are commonly encountered in mediation systems because they provide conversion functions between different contexts.  In our motivational example, we use the datexform source, which converts dates between different formats.  Functional relations reside inside a Function Ccontainer Servlet.  Class Servlet Access implements Functional Access interface and makes calls on functions residing inside a Function Container Servlet.  Servlet Access uses the CGI interface to supply the function name and input parameters to functions residing in the Function Container Servlet.  The output results are returned in XML format.  Appendix E shows the code implementing the Function Container Servlet containing a single conversion function – datexform.  Note that more general Function Containers may be implemented containing many conversion functions.

The following is the format of the request sent to the Function Container Servlet for conversion of date '01/05/94' from American Style with '/' used as delimiter to European Style with '-' used as delimiter:

```
http://hostname/servlet/edu.mit.gcms.functionservlet?function=datexform&
format1=European Style -& date2=01/05/94&format2=American Style /
```

The result of this conversion returned in XML format is:

```
<?>>>>>>>>>>>>>>>>>>?>>>
  <>>>>>>>>   >>
  <>>>>>>>> >>
 ><>>>>>>05/0////  <>>>>>>>>>
 ><>>>>>>>>////////////////   <>>>>>>>>>>>
 ><>>>>>>0//05///   <>>>>>>>>>
 ><>>>>>>>>////////////////    <>>>>>>>>>>>
 ><>>>>>>>> >>
 ><>>>>>>>>>   >
```

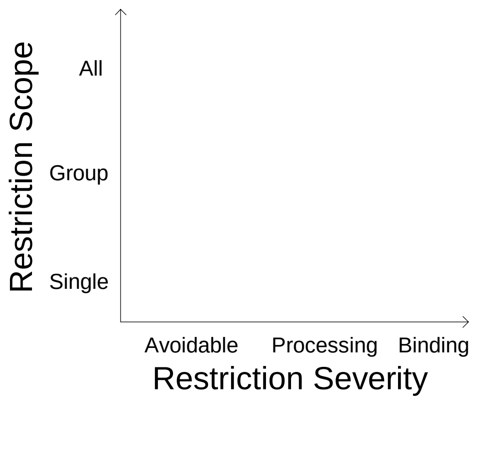
An alternative to sending function parameters as URL query strings is to use SQL like in the case of the Cameleon queries.  Here is the resulting SQL for datexform CSQ:

```
select  date1, format1, date2, format2
from    datexform
where date1="01/05/94"
and     format1= "European Style -"
and     format2="American Style /"
```

Advantage of this approach is that it provides a common SQL interface and disadvantage that it is harder to implement because a SQL parser is needed to parse the SQL query.

## 7.3 Classification of Source Restrictions for Non-Relational Sources

I classify query restrictions along two orthogonal dimensions: (1) severity of restriction, and (2) scope of restriction.



### 7.3.1 Restriction Severity

With regard to severity, I classify source restrictions in the following three categories:

- Binding restrictions
- Processing restrictions
- Avoidable query restrictions

Binding restrictions make it impossible to retrieve all information present in a data source. For example, finance.yahoo.com asks for a ticker symbol in order to retrieve quote data. We cannot issue a query or series of queries (without the knowledge of all the tickers) that would retrieve all the data present in finance.yahoo.com information source. This data restriction prevents us from issuing a relational query that would retrieve tickers of all stocks with trading price above 100. Another example of binding restrictions comes from our motivational example. A query on relation olsen cannot return an answer unless attributes Expressed, Exchanged, and Date are bound.

Processing restrictions are those restrictions that prevent us from doing transformations of the data at the information source. Typical RDBMS, such as Oracle, impose no restrictions on transformations that can be performed. Most of the web sources, however, have little or no processing capabilities, as they cannot perform joins, condition, subqueries, or even simple arithmetic operations on attributes, such as addition. Operator restrictions discussed in section 6.3.4 are a type of processing restrictions.

Avoidable query restrictions are those restrictions that make it impossible to retrieve required data in a single query. However, it is possible to execute a series of queries on this information source that would yield the required data. An example of avoidable query restriction is a batch-of-tuples-at-a-time restriction, where the information source allows only a few records to be retrieved at a time (see section 8.1.1). To avoid batch-of-tuples-at-a-time restriction, the Executioner can keep querying the source until all the required data is retrieved.

### 7.3.2 Restriction Scope

With regard to scope, I classify source restrictions in the following three categories:

- Single relation
- Group of relations
- All relations

The lowest level of granularity of restriction severity is a single relation. Capability records used in my Planner/Optimizer/Executioner specify restrictions on single relations only. Note, however, that not all restrictions can be expressed at the granularity of a single relation. For example, restriction that the JOIN between the two specific relations in a source is prohibited requires restriction scope to contain at least those two relations. Additional benefit of ‘group of relations’ and ‘all relations’ restriction scopes is that they make it easier for the user to define source restrictions. Instead of defining a restriction for every relation in a source, the user can define a single restriction shared by a group of relations or all relations (e.g. schema independent query restrictions). For example, this is useful in a web source where all relations typically have a restriction of not being able to perform a JOIN operation.

# 8 Conclusions and Future Work

In general, the POE has achieved its aim of providing a stable execution environment for context mediated queries. Nevertheless, there are a number of improvements which can be made in the POE to improve its execution. These improvements are in the areas of handling query restrictions, optimizing query execution, and integration with non-relational sources.

## 8.1 Handling Query Restrictions

While capability records in POE handle the most common cases of query restrictions, such as binding restrictions, operator restrictions, and key-at-a-time restrictions, improvements can be made in handling less commonly occurring query restrictions. In section 8.1.1, I describe how to handle bach-of-tuples-at-a-time restriction, and in section 8.1.2, I present a generic grammar-based approach for describing query capabilities of arbitrary complexity.

### 8.1.1 Batch of Tuples at a Time Retrieval

Some web sources impose a restriction on the number of tuples that can be returned when querying a relation. Search engines typically exhibit this behavior – they only return a pre-set number of results at the time for each query. For example, querying www.google.com for 'context mediation' results in 280 records but only 10 records are returned per page. If we wished to obtain all the results, we would need to query www.google.com 28 times fetching 10 tuples at a time.

Handling batch-of-tuples-at-a-time restriction involves extending capability records to carry the information on how many tuples can be returned by a query. In addition, Functional Access interface needs to be modified in order to keep track of the execution state. In our example, the execution state is the record number from which to start counting next batch of tuples to be retrieved.

Bathch-of-tuples-at-time-restriction can only be handled by wrappers that take starting record number as an input parameter. In the case of Cameleon, we can extend SQL with startat keyword specifying the record number from which to start counting the batch of tuples. Then, retrieving records 31-40 from the google relation can be achieve with the following query:

```
select search_results
from google
where search_keyword='context mediation'
and startat= 31;
```

### 8.1.2 Grammar-Based Approach for Specifying Query Capabilities

As we have already seen in the previous section, capability records are limited in their ability to express query capabilities. An improvement could be made in integrating sources with complex query restrictions by using Relational Query Description Language (RQDL) [13]. RQDL is a grammar-based approach for specifying query capabilities and it can describe large and infinite sets of supported queries as well as schema independent queries.

However, RQDL also has disadvantages over capability records. While a capability record is easy to understand and write, RQDL may not be so. An effective solution may be to use capability records at the front end for typical usage and RQDL at the backend. For advance uses when the power of RQDL is necessary, administrators can be given an option to specify query capabilities using RQDL.

## 8.2 Optimizing Query Execution

In this section, I present practical approaches for optimizing query execution. I first discuss semantic query optimization, an approach applicable to all queries and showing a great promise in reducing the query execution time. I continue with the discussion of performing joins locally in the queries involving joined relations from the same source. I finish with presenting a better strategy for managing temporary relation in the RDBMS engine than the one described in section 5.2.

### 8.2.1 Semantic Query Optimization

Let us examine the query shown in Figure 8-1 and the mediated datalog produced by mediation engine shown in Figure 8-2.

```
context=c_dt
select COMPANY_NAME
from DiscAF
where COMPANY_NAME='DAIMLER-BENZ';
```

**Figure 8-1 SQL for Semantic Query Optimization Example**

```
answer("DAIMLER-BENZ") :-
        'DiscAF'('V7', 'V6', 'V5', 'V4', 'V3', 'V2', 'V1'),
        'Name_map_Dt_Ds'("DAIMLER-BENZ", 'V7').
```

**Figure 8-2 Datalog for Semantic Query Optimization Example**

Notice that mediated datalog in Figure 8-2 does not need to access relations DiscAF and Name_map_Dt_Ds relations. answer("DAIMLER-BENZ") contains a constant "DAIMLER-BENZ" in its projection list and it does not need any additional information to answer the original user query. In general, semantic optimization can be performed on a datalog query before it is sent to the Executioner for processing. The straightforward algorithm for detecting unused variables unused in answering the query is presented in Figure 8-3. Unused relations can then be easily identified because they only contain unused variables.

```
Input:    Datalog Query d
Output: Set of variables not used in answering query d

Find Unused Variables:
1.  initialize set UNUSED to all variables occurring in a query d
2.  initialize set USED to an empty set
3.  move all variables occurring in answer() predicate from UNUSED to USED
4.  repeat until no new variables are added to the UNUSED set
5.      for all USED variables v
6.          for all relations r containing variable v
7.              if relation r contains some variable u in UNUSED set
8.                  move variable u from UNUSED to USED
9.                  continue 4
10. return UNUSED
```

**Figure 8-3 Algorithm for Finding Unused Variables in a Datalog Query**

The semantic query optimization done in the POE should be closely coordinated with the work done on mediation engine. The reason is that the mediation engine already performs some semantic query optimization as a part of the mediation process. A clear-cut demarcation needs to be made between semantic optimizations in the mediation engine and semantic optimizations in the POE

### 8.2.2 Performing Joins Locally on the Relations From the Same Source

A significant reduction in query execution time can be achieved by performing joins locally on all relations coming from the same source. This approach is applicable to any query containing two joined relations from the same source. In our motivational example, relations DStreamAF, WolrdcAF and Name_map_Dt_Ws all reside in the same database. Instead of accessing each of these three relations separately by sending three SQL queries to the database to fetch the data, we can send a single query that performs join on these three data sources and fetches results into the local data store. This way the database source is accessed only once and the number of tuples transferred over the network is reduced.

### 8.2.3 Pooling Temporary Relations

In section 5.2, I explained how temporary relations are created at the Executioner initialization time. However, this approach does not scale to a global system with a large number of relations. The reason is that creating temporary relations would take too long at the initialization time and that underlying RDBMS engine may have limitations on the number of temporary relations it can support.

Creating temporary relations on the fly solves scalability problem. However, this approach lengthens the query execution time because creating temporary tables at a run-time takes a long time (on the order of 50ms). An alternative hybrid solution that solves both scalability and performance problems is to keep a pool of temporary relations for frequently accessed relations. Every time a query is executed, the Executioner first checks if the temporary relation needed for answering the query is already in the pool of temporary relations. If the temporary relation is already in the pool, then the Executioner uses the pooled temporary relation and if its not then the Executioner creates a new temporary table on the fly and adds it to the pool of temporary relations.

## 8.3 Integration with Non-Relational Sources

In the current implementation of the POE, I use a CGI interface to send function parameters to Function Container Servlet and retrieve the results in XML format. While this approach works well, it can be improved by using a recently announced XML-RPC protocol.

In the XML based RPC, a remote procedure call is represented using an XML based protocol such as the SOAP 1.1 specification. Using XML-RPC, a Function Container Servlet can define, describe and export conversion functions as RPC based services. Executioner can then make remote procedure calls to Function Container Servlet to execute the conversion functions. The advantage of XML-RPC over existing CGI/XML approach is that XML-RPC is a standard, it is more robust, and it makes defining and exporting conversion functions easier.

## 8.4 Conclusion

The Planner/Optimizer/Executioner I implemented has been successfully deployed on the financial application TASC built on top of the COIN system. It has proved to be both versatile and effective in answering the queries from a variety of data sources. In addition, I developed a fully functional datalog to SQL translator and demonstrated a feasibility of using RDBMS engine of a single database as a basis for the multi database Executioner. On the theoretical side, I showed how to handle data sources with varying processing capabilities, showed how to optimize queries in distributed environments where costs statistics are not readily available, and presented the solution to a problem of integration with non-relational data sources.

## 9 References

[1] S. Bressan, K. Fynn, T. Pena, C. Goh, and et al. Demonstration of the context interchange mediator prototype. In Proceedings of ACM SIGMOD/PODS Conference on Management of Data, Tucson, AZ, May 1997.

[2] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). IEEE Transactions On Knowledge And Data Engineering, 1(1), March 1989.

[3] W. Du, R. Krishnamurthy, and M. C. Shan. Query optimization in heterogeneous dbms. In International Conference on VLDB, Vancouver, Canada, September 1992.

[4] C. H. Goh, S. Madnick, and M. Siegel. Context interchange: Overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In Proceedings of the Third Int'l Conf on Information and Knowledge Management, Gaithersburg, MD, November 1994.

[5] Cheng Hian Goh. Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems. PhD dissertation, Massachusetts Institute of Technology, Sloan School of Management, December 1996.

[6] C.H. Goh, S. Bressan, S. Madnick, M. Siegel, "Context Interchange: New Features and Formalisms for the Intelligent Integration of Information", ACM Transactions on Office Information Systems, July 1999 [SWP #3941, CISL #97-03].

[7] S. Bressan, C. Goh, N. Levina, A. Shah, S. Madnick, M. Siegel, “Context Knowledge Representation and Reasoning in the Context Interchange System”, Applied Intelligence: The International Journal of Artificial Intelligence, Neutral Networks, and Complex Problem-Solving Technologies, Volume 12, Number 2, September 2000, pp. 165-179, [SWP #4133, CISL #00-04].

[8] R. Jain. The Art of Computer Systems Performance Analysis. John Wiley and Sons, Inc., 1991.

[9] Marta Jakobisiak. Programming the web – design and implementation of a multidatabase browser. Technical Report CISL WP#96-04, Sloan School of Management, Massachusetts Institute of Technology, May 1996.

[10] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogenous information sources using source descriptions. In Proceedings of the 22nd VLDB Conference., Mumbai (Bombay), India, September 1996.

[11] K. A. Morris. An algorithm for ordering subgoals in nail. In Proceedings of the 7th ACM Symposium on Principles of Database Systems, Autin, TX, March 1988.

[12] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In Deductive and Object-Oriented Databases, Fourth International Conference., pages 161-186, Singapore, December 1995.

[13] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In to appear in Fourth International Conference on Paralled and Distributed Information Systems, Miami Beach, Florida, December 1996.

[14] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD International

Conference on Management of Data, pages 23–34, Boston, May 1979.
[15] Leon Sterling and Ehud Shapiro. The Art of Prolog : Advanced Programming Techniques. The MIT Press, 1994.
[16] Koffi Fynn, A Planner/Optimizer/Executioner for Context Mediated Queries, MIT Masters Thesis, Electrical Engineering and Computer Science, 1997.
[17] A. Tomasic, L. Raschid, and P. Valduriez. A data model and query processing techniques for scaling acccess to distributed heterogeneous databases in DISCO, 1996. Submitted for publication.
[18] Laura M. Hass, Donald Kossmann, Edward L. Wimmers, Jun Yang. Optimizing Queries across Diverse Data Sources. Proceedings of the 23rd VLDB Conference. Athens, Greece, 1997.
[19] S. Adali, K.S. Candan, et al. Query Caching and Optimization in Distributed Mediator Systems. SIGMOD Conference 1996.
[20] Qiang Zhu, and P. A. Larson. A query sampling method of estimating local cost parameters in a multidatabase system. Technical Report CS93-42, University of Waterloo, Canada, 1993.

## Appendix A Relation Sources, Schemas, and Capabilities

```
%%
%% Sources
%%

source(oracle, database, ‘jdbc:oracle:oci8:@coin&system&manager’).
source(cameleon, cameleon, ‘http://context2.mit.edu:8081/servlet/camserv’).
source(servlet, functionservlet, 'http://context2.mit.edu/servlet/edu.mit.gcms.functionservlet ').

%%
%% Relations
%%

relation(cameleon,
        ‘secapl’,
         [[‘TICKER’, string], [‘PRICE’, number]],
         cap([[b(1), f]], [‘<’, ‘>’, ‘<>’, ‘<=’, ‘>=’])

relation(oracle,
        'Currencytypes',
        [['COUNTRY',string],['CURRENCY',string]],
         cap([[?, ?]], [])).

relation(oracle,
        'currency_map',
        [['CHAR3_CURRENCY',string],['CHAR2_CURRENCY',string]],
         cap([[?,?]], [])).

relation(servlet,
        datexform,
        [['Date1', string], ['Format1', string], ['Date2', string],
         ['Format2', string]],
        cap([[b(1),b(1),f,b(1)], [f,b(1),b(1),b(1)]], ['<','>','<>','<=','>='])).

relation(cameleon,
        olsen,
        [['Exchanged',string],['Expressed',string],['Rate',number],
         ['Date',string]],
        cap([[b(1),b(1),f,b(1)]], ['<','>','<>','<=','>='])).

relation(cameleon,
        quotes,
        [['Cname',string],['Last', string]],
        cap([[b(1),f]], ['<','>','<>','<=','>='] )).


relation(oracle,
        'DiscAF',
        [['COMPANY_NAME',string],['LATEST_ANNUAL_DATA',string],
         ['CURRENT_SHARES_OUTSTANDING',number],['NET_INCOME',number],
         ['NET_SALES',number],['TOTAL_ASSETS',number],
         ['LOCATION_OF_INCORP',string]],
        cap([[?,?,?,?,?,?,?]], [])).
```

```
relation(oracle,
            'WorldcAF',
            [['COMPANY_NAME',string],['LATEST_ANNUAL_FINANCIAL_DATE',string],
             ['CURRENT_OUTSTANDING_SHARES',number],
             ['NET_INCOME',number],['SALES',number],['TOTAL_ASSETS',number],
             ['COUNTRY_OF_INCORP', string]],
            cap([[?,?,?,?,?,?,?]], [])).

relation(oracle,
            'WorldcAFT',
            [['COMPANY_NAME',string],['LATEST_ANNUAL_FINANCIAL_DATE',string],
             ['CURRENT_OUTSTANDING_SHARES',number],
             ['NET_INCOME',number],['SALES',number],['TOTAL_ASSETS',number],
             ['COUNTRY_OF_INCORP', string]],
            cap([[?,?,?,?,?,?,?]], [])).

relation(oracle,
            'Name_map_Ds_Ws',
            [['DS_NAMES',string],['WS_NAMES',string]],
            cap([[?,?]], [])).

relation(oracle,
            'Ticker_Lookup',
            [['COMP_NAME',string],['TICKER',string],
            ['EXC', string]],
            cap([[?,?,?]], [])).


relation(oracle,
            'Name_map_Dt_Ds',
            [['DT_NAMES',string],['DS_NAMES',string]],
            cap([[?,?]], [])).

relation(oracle,
            'Name_map_Dt_Ws',
            [['DT_NAMES',string],['WS_NAMES',string]],
            cap([[?,?]], [])).

relation(oracle,
            'DStreamAF',
            [['AS_OF_DATE',string],['NAME',string],
             ['TOTAL_SALES',number],
             ['TOTAL_EXTRAORD_ITEMS_PRE_TAX',number],
             ['EARNED_FOR_ORDINARY',number],
             ['CURRENCY',string]],
            cap([[?,?,?,?,?,?]], [])).

relation(cameleon,
            quicken,
            [['Ticker',string],
             ['Headlines',string],
             ['LastTrade',number],
            cap([[b(1), f, f]], [<, >, <>, <=, >=])).

relation(cameleon,
            cia,
            [['Country',string],
             ['capital',string],
             ['economy',string],
             ['location',string],
             ['coordinates',string],
             ['totalarea',string],
             ['climate',string],
             ['population',string],
             ['telephone',string],
             ['GDP',string],
             ['Background',string],
             ['Link',string],
            cap([[b(1), f, f, f, f, f, f, f, f, f, f, f]], [<, >, <>, <=, >=])).

relation(cameleon,
            moneyrates,
            [['bankname',string],
             ['rate',number],
             ['yield',number],
             ['minbalance',number],
            cap([[f, f, f, f]], [<, >, <>, <=, >=])).
```

## Appendix B Specification of Contexts

| Context | Currency | Scale Factor | Currency Type | Date Format |
|---|---|---|---|---|
| Worldscope, c_ws | USD | 1000 | 3char | American Style / |
| Datastream, c_dt | Country of Incorporation | 1000 | 2char | European Style - |
| Disclosure, c_ds | Country of Incorporation | 1 | 3char | American Style / |
| Olsen, c_ol |  | 1 | 3char | American Style / |

# Appendix C Remote Access Interfaces

```
public interface RemoteAccess {
    public static final int RELATIONAL_ACCESS=1;
    public static final int FUNCTIONAL_ACCESS=2;

    public int type();
}


public interface RelationalAccess extends RemoteAccess {
    public OrderedTable executeQuery(String sql) throws Exception;
    public int type();
}


public interface FunctionalAccess extends RemoteAccess {
public OrderedTable executeQuery(Relation r, boolean[] in, boolean[] out, String[] inBinding);
public OrderedTable executeQuery(Relation r, boolean[] in, boolean[] out, OrderedTable inBindings);
    public int type();
}
```

## Appendix D Ordered Table Class

```
public class OrderedTable {

   public void addRow(List row);
   public List getRow(int rowNum);
   public int getNumRows() ;
   public int getNumColumns();
   public Object getElement(int rowNum, int colNum);
   public void merge(OrderedTable o);
}
```

## Appendix E Datexform Servlet

```
package edu.mit.gcms.demo.servlets;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import edu.mit.gcms.util.StringUtils;

public class datexform extends HttpServlet {
  private static final String CONTENT_TYPE = "text/xml";
  public static final int EUROPEAN= 1;
  public static final int AMERICAN= 2;

  /**Initialize global variables*/
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
  }

  public String getParamValue(HttpServletRequest request, String paramName) {
      String[] paramValues= request.getParameterValues(paramName);
      if (paramValues==null) {
          return "";
      } else {
          return paramValues[0];
      }
  }

  boolean bound(String paramName) {
      return !paramName.equals("");
  }

  /**Process the HTTP Get request*/
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();

    String function= getParamValue(request, "function");
    String date1= getParamValue(request, "date1");
    String date2= getParamValue(request, "date2");
    String format1= getParamValue(request, "format1");
    String format2= getParamValue(request, "format2");

    if (!function.equals("datexform")) {
        throw new ServletException("unknown function");
    }

    if (bound(date1) && bound(format1) && bound(format2) && !bound(date2)) {
        date2= convertDateFormat(date1, format1, format2);
    } else if (bound(date2) && bound(format1) && bound(format2) && !bound(date1)) {
        date1= convertDateFormat(date2, format2, format1);
    } else {
        throw new ServletException("unsupported function bindings");
    }

    String xml= "";

    xml+= "<?xml version='1.0'?>\n";
    xml+= "<DOCUMENT>\n";
    xml+= "<ELEMENT>\n";
    xml+= "<date1>"+date1+"</date1>\n";
    xml+= "<format1>"+format1+"</format1>\n";
    xml+= "<date2>"+date2+"</date2>\n";
    xml+= "<format2>"+format2+"</format2>\n";
    xml+= "</ELEMENT>\n";
    xml+= "</DOCUMENT>\n";

    out.print(xml);
  }
  /**Process the HTTP Post request*/
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    doGet(request, response);
  }

  /**Clean up resources*/
  public void destroy() {
  }

  // european style= dd-mm-yy
  // american style= mm/dd/yy
  public String convertDateFormat(String date, String fromFormat, String toFormat) {
    int fromFormatType, toFormatType;
    char fromDelimiter, toDelimiter;
    String day, month, year;

    if (fromFormat.startsWith("European Style")) {
      fromFormatType= datexform.EUROPEAN;
      fromDelimiter= fromFormat.charAt(15);
    } else if (fromFormat.startsWith("American Style")) {
      fromFormatType= datexform.AMERICAN;
      fromDelimiter= fromFormat.charAt(15);
    } else {
      throw new RuntimeException(fromFormat + " is unrecognized date format");
    }

    if (toFormat.startsWith("European Style")) {
      toFormatType= datexform.EUROPEAN;
      toDelimiter= toFormat.charAt(15);
    } else if (toFormat.startsWith("American Style")) {
      toFormatType= datexform.AMERICAN;
      toDelimiter= toFormat.charAt(15);
    } else {
      throw new RuntimeException(toFormat + " is unrecognized date format");
    }

    String[] elements= StringUtils.split(date, ""+fromDelimiter);

    if (elements.length!=3) {
            throw new RuntimeException("invalid date format");
        }

        switch (fromFormatType) {
            case datexform.EUROPEAN:
                day= elements[0];
                month= elements[1];
                year= elements[2];
                break;
            case datexform.AMERICAN:
                day= elements[1];
                month= elements[0];
                year= elements[2];
                break;
            default:
                throw new RuntimeException("invalid date format");
        }

        switch (toFormatType) {
            case datexform.EUROPEAN:
                return day+toDelimiter+month+toDelimiter+year;
            case datexform.AMERICAN:
                return month+toDelimiter+day+toDelimiter+year;
            default:
                throw new RuntimeException("invalid date format");
        }
    }
```

# Appendix F Software Components

package edu.mit.gcms.poe (630 lines)

Decription: Contains modules for working with internal query representation: (1) Datalog Parser for converting datalog into internal query representation, and (2) SQL Generator for generating SQL from internal query representation.

451 lines DatalogParser.java
179 lines SQLGenerator.java

package edu.mit.gcms.poe.executioner (1,448 lines)

Description: Contains modules of Executioner component. This includes remote access interfaces (Relational Access and Functional Access), classes implementing those remote interfaces (DB Access, Cameleon Access, and Servlet Access), parallel functional executioner responsible for queries sent to Cameleon and function container servlet, and main Executioner class, which executes QEP.

897 lines Executioner.java
122 lines CameleonAccess.java
108 lines CSQOrdering.java
96 lines ServletAccess.java
74 lines DBAccess.java
66 lines ParallelFunctionalExecutioner.java
30 lines CSQBinding.java
19 lines RemoteAccess.java (interface)
19 lines FunctionalAccess.java (interface)
17 lines RelationalAccess.java (interface)

package edu.mit.gcms.poe.query (782 lines)

Description: Contains internal query representation structures: Single Query, CSQ, Condition, and Alias.

594 lines Single Query.java
84 lines CSQ.java
42 lines Alias.java
39 lines Condition.java
23 lines CSQAttribute.java

package edu.mit.gcms.poe.metadata (538 lines)

Description: Contains metadata for representing schema information: Sources, Relations, Attributes, and Capability Records.

48 lines Attribute.java
68 lines AttributeSpecifier.java
46 lines BindingSpecifier.java
216 lines CapabilityRecord.java
69 lines Relation.java
48 lines Schema.java
43 lines Source.java

package edu.mit.gcms.poe.term (172 lines)

Description: Contains internal representation of the Term component. Term is a Java interface implemented by four classes: Number Constant, String Constant, Expression, and Variable.

44 lines Expression.java
41 lines Variable.java
32 lines NumberConstant.java
32 lines StringConstant.java
23 lines Term.java (interface)

package edu.mit.gcms.poe.operators (147 lines)

Description: Contains modules representing arithmetic operators and comparators.

59 lines ArithmeticOperator.java
88 lines Comparator.java

package edu.mit.gcms.util (514 lines)

Description: Contains modules with utility functions and structures.
101 lines OrderedTable.java
236 lines StringUtils.java
39 lines Timer.java
41 lines TimingEvent.java
53 lines TimingInfo.java
44 lines Util.java

package edu.mit.gcms.demo.servlets (135 lines)
Description: Contains servlets conforming to CGI-XML functional access interface.
135 lines datexform.java

package edu.mit.gcms.demo (659 lines)
Description: Contains code demonstrating capabilities of Executioner and COIN system in general.
538 lines DemoBean.java
121 lines DemoQueries.java

# Appendix G Query Execution Trace for Motivational Example

| | |
|---|---|
| 1629ms | TOTAL |
| 1629ms | Execution |
| 10ms | Datalog Parsing |
| 20ms | Planning and Optimization |
| 0ms | Calculating Join Groups:<br>((datexform, dstreamaf, currency_map, name_map_dt_ws, worldcaf)) |
| 20ms | Calulating Optimal CSQ Cost Ordering<br>((currency_map ()), (dstreamaf ((currency (currency_map char2_currency)))), (datexform ()), (name_map_dt_ws ((dt_names (dstreamaf name)))), (worldcaf ((company_name (name_map_dt_ws ws_names))))) |
| 20ms | Calulating Independent CSQ Cost Ordering:<br>(currency_map, name_map_dt_ws, dstreamaf, worldcaf, datexform) |
| 10ms | Calculating CSQ Cost:<br>select count(*)<br>from dstreamaf<br>where  10000000 < total_sales<br>CSQ Cost= 266 |
| 0ms | Calculating CSQ Cost:<br>select count(*)<br>from currency_map<br>where  char3_currency='USD'<br>CSQ Cost= 1 |
| 10ms | Calculating CSQ Cost:<br>select count(*)<br>from name_map_dt_ws<br>CSQ Cost= 7 |
| 0ms | Calculating CSQ Cost:<br>select count(*)<br>from worldcaf<br>CSQ Cost= 354 |
| 240ms | Fetching Remote CSQ Data |
| 20ms | Fetching currency_map Data |
| 10ms | Preparing Temporary Table TTP_currency_map |
| 10ms | Executing at remote database oracle:<br>select 'USD', char2_currency<br>from   currency_map<br>where  char3_currency='USD' |
| 0ms | Batch Inserting Remote Data into Temporary Table |
| 30ms | Fetching dstreamaf Data |
| 10ms | Preparing Temporary Table TTP_dstreamaf |
| 20ms | Executing at remote database oracle:<br>select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency<br>from   dstreamaf<br>where  10000000 < total_sales<br>and    currency in (US) |
| 0ms | Batch Inserting Remote Data into Temporary Table |
| 120ms | Fetching datexform Data |
| 30ms | Preparing Temporary Table TTP_datexform |

| | |
|---|---|
| 10ms | Executing Remote Functional Query:<br>http://avocado.mit.edu/servlet/edu.mit.gcms.demo.servlets.datexform?function=datexform&format1=European+Style+-&date2=01%2F05%2F94&format2=American+Style+%2F |
| 80ms | Batch Inserting Remote Data into Temporary Table |
| 40ms | Fetching name_map_dt_ws Data |
| 20ms | Preparing Temporary Table TTP_name_map_dt_ws |
| 20ms | Executing at remote database oracle:<br>select dt_names, ws_names<br>from name_map_dt_ws |
| 0ms | Batch Inserting Remote Data into Temporary Table |
| 30ms | Fetching worldcaf Data |
| 20ms | Preparing Temporary Table TTP_worldcaf |
| 10ms | Executing at remote database oracle:<br>select company_name, latest_annual_financial_date, current_outstanding_shares, net_income, sales, total_assets, country_of_incorp<br>from worldcaf |
| 0ms | Batch Inserting Remote Data into Temporary Table |
| 20ms | Planning and Optimization |
| 0ms | Calculating Join Groups:<br>((datexform2, dstreamaf2, name_map_dt_ws2, currency_map2, worldcaf2, olsen)) |
| 20ms | Calulating Optimal CSQ Cost Ordering<br>((name_map_dt_ws2 ()), (worldcaf2 ((company_name (name_map_dt_ws2 ws_names)))), (dstreamaf2 ((name (name_map_dt_ws2 dt_names)))), (datexform2 ()), (currency_map2 ((char2_currency (dstreamaf2 currency)))), (olsen ((exchanged (currency_map2 char3_currency))))) |
| 20ms | Calulating Independent CSQ Cost Ordering:<br>(name_map_dt_ws2, currency_map2, dstreamaf2, worldcaf2, datexform2) |
| 10ms | Calculating CSQ Cost:<br>select count(*)<br>from dstreamaf<br>CSQ Cost= 312 |
| 0ms | Retrieving Cached CSQ Cost:<br>select count(*)<br>from name_map_dt_ws<br>CSQ Cost= 7 |
| 0ms | Calculating CSQ Cost:<br>select count(*)<br>from currency_map<br>where char3_currency <> 'USD'<br>CSQ Cost= 9 |
| 10ms | Retrieving Cached CSQ Cost:<br>select count(*)<br>from worldcaf<br>CSQ Cost= 354 |
| 1108ms | Fetching Remote CSQ Data |
| 30ms | Fetching name_map_dt_ws2 Data |
| 20ms | Preparing Temporary Table TTP_name_map_dt_ws2 |
| 10ms | Retrieving Cached SQL Query :<br>select dt_names, ws_names<br>from name_map_dt_ws |

| | |
|---|---|
| 0ms | Batch Inserting Remote Data into Temporary Table |
| 40ms | Fetching worldcaf2 Data |
| 10ms | Preparing Temporary Table TTP_worldcaf2 |
| 20ms | Executing at remote database oracle:<br>select company_name, latest_annual_financial_date, current_outstanding_shares, net_income, sales, total_assets, country_of_incorp<br>from worldcaf<br>where company_name in (DAIMLER-BENZ AG, BRITISH TELECOMMUNICATIONS PLC, NIPPON TELEGRAPH & TELEPHONE C, LYONNAISE DES EAUX SA, ITOCHU CORPORATION, SUMITOMO CORPORATION, TOMEN CORPORATION) |
| 10ms | Batch Inserting Remote Data into Temporary Table |
| 70ms | Fetching dstreamaf2 Data |
| 10ms | Preparing Temporary Table TTP_dstreamaf2 |
| 30ms | Executing at remote database oracle:<br>select as_of_date, name, total_sales, total_extraord_items_pre_tax, earned_for_ordinary, currency<br>from dstreamaf<br>where name in (DAIMLER-BENZ, BRITISH TELECOM., NTT, LYONNAISE DES EAUX, ITOCHU CORPORATION, SUMITOMO CORPORATION, TOMEN CORPORATION) |
| 30ms | Batch Inserting Remote Data into Temporary Table |
| 20ms | Fetching datexform2 Data |
| 10ms | Preparing Temporary Table TTP_datexform2 |
| 0ms | Retrieving Cached Functional Functional Query:<br>http://avocado.mit.edu/servlet/edu.mit.gcms.demo.servlets.datexform?function=datexform&format1=European+Style+-&date2=01%2F05%2F94&format2=American+Style+%2F |
| 10ms | Batch Inserting Remote Data into Temporary Table |
| 41ms | Fetching currency_map2 Data |
| 11ms | Preparing Temporary Table TTP_currency_map2 |
| 20ms | Executing at remote database oracle:<br>select char3_currency, char2_currency<br>from currency_map<br>where char3_currency <> 'USD'<br>and char2_currency in (BP, BP, BP, BP, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, JY, FF, FF, FF, FF, DM, DM, DM, DM, DM) |
| 10ms | Batch Inserting Remote Data into Temporary Table |
| 907ms | Fetching olsen Data |
| 10ms | Preparing Temporary Table TTP_olsen |
| 877ms | Parallel Execution |
| 290ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="DEM" and expressed="USD" and date="01/05/94" |
| 470ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="FRF" and expressed="USD" and date="01/05/94" |
| 480ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="JPY" and expressed="USD" and date="01/05/94" |
| 877ms | Executing Remote Cameleon Query:<br>select exchanged, expressed, rate, date from olsen where exchanged="GBP" and expressed="USD" and date="01/05/94" |

| | |
|---|---|
| 10ms | Batch Inserting Remote Data into Temporary Table |
| 30ms | Database Execution of Final Query: select TTP_name_map_dt_ws.ws_names, TTP_dstreamaf.total_sales, TTP_worldcaf.latest_annual_financial_date, TTP_worldcaf.total_assets from TTP_datexform, TTP_name_map_dt_ws, TTP_dstreamaf, TTP_currency_map, TTP_worldcaf where TTP_dstreamaf.currency = TTP_currency_map.char2_currency and TTP_name_map_dt_ws.ws_names = TTP_worldcaf.company_name and TTP_datexform.date1 = TTP_dstreamaf.as_of_date and TTP_name_map_dt_ws.dt_names = TTP_dstreamaf.name union select TTP_name_map_dt_ws2.ws_names, TTP_dstreamaf2.total_sales*TTP_olsen.rate, TTP_worldcaf2.latest_annual_financial_date, TTP_worldcaf2.total_assets from TTP_datexform2, TTP_name_map_dt_ws2, TTP_dstreamaf2, TTP_currency_map2, TTP_olsen, TTP_worldcaf2 where TTP_name_map_dt_ws2.ws_names = TTP_worldcaf2.company_name and TTP_datexform2.date1 = TTP_dstreamaf2.as_of_date and TTP_name_map_dt_ws2.dt_names = TTP_dstreamaf2.name and TTP_dstreamaf2.currency = TTP_currency_map2.char2_currency and TTP_currency_map2.char3_currency = TTP_olsen.exchanged and 10000000 < TTP_dstreamaf2.total_sales*TTP_olsen.rate |

## Appendix H Testing Setup

Live demo is accessible from COIN system demo page at
http://context2.mit.edu/coin/demos/

Machine characteristics:
Pentium III 500MHz
256MB RAM
Windows 2000 Professional
Oracle 8.1.7

# Appendix I Installation Instructions

ALL INSTALLATION FILES ARE LOCATED AT http://avocado.mit.edu/gcms/install

## 1 Eclipse

1. You have administrative privileges.
2. Install Eclipse 5.3 under C:\eclipse, Follow instructions in http://avocado.mit.edu/gcms/install/Eclipse5.3_41/README_WIN.TXT

## 2 Web Server (JRun)

1. Install Jr302.exe [accept full installation, run as a service, skip the license number-will be limited to 3 simultaneous connections-]
2. Quit jrun processes [Kill jrun process from Task Manager]
3. Install jr30sp2 [service pack]
4. Make your JRun installation robust:
   When JRun servers are installed as NT services on Windows NT or 2000, Sun JDK/JRE 1.3 and IBM JDK 1.3 users may find that JRun servers stop when the user is logged off. This bug is fixed in the JDK 1.3.1 beta release. Please see to the JDK 1.3.1 release notes. Workaround for JDK 1.3: Open the *JRun_rootdir*/bin/global.properties file in a text editor.
Under Java VM Settings, set the java.exe property to *JRun_rootdir*/bin/dontdiejava.exe as follows:

    java.exe=*JRun_rootdir*\\bin\\dontdiejava.exe.

Add an extra entry for java.jnipath property for directory containing the jvm.dll file as follows:
java.jnipath=C:\\Sun\\jsdk130\\jre\\bin\\hotspot;{user.jnipath};{ejb.jnipath};{servlet.jnipath}
You cannot set java.jnipath indirectly by setting user.jnipath. You must use java.jnipath directly.

## 3 XML Libraries

1. Download Sun's XML Parser from http://java.sun.com/xml/downloads/javaxmlpack.html
2. Unzip it into any directory.

## 4 Oracle

1. Install Oracle Client if you do not have Oracle database on your machine. Copy Oracle8.1.7 directory from avocado GCMS-Install into your own computer. Start installation and choose Oracle Client, and choose the application user type of installation. Database name is coin, connection TCP. Username is system, password manager [you will get an error since the default username is scott, you have to enter correct username and password]
2. Make sure the connection test is successful
3. (OPTIONAL) If you have Oracle database on your machine, you can upload sample data containing Datastream, Disclosure, Worldscope data from SampleData.dmp file using the following command:

```
imp full=y file=SampleData.dmp system/manager
```

## 5 COIN

1. Copy GCMSStable directory into "GCMS-dir"
2. If you do not want any development environment then you can simply use the compiled class files and skip 3-7
3. Start JBuilder
4. Open the GCMS.jpx from "GCMS-dir"

5. Update the parameters under Project=>Project Properties=>Run-change the directory names make them point to GCMS directory
For example
applicationPath = C:\\JBuilder4\\projects\\GCMS\\src\\edu\\mit\\gcms\\demo\\application
eclipsePath = C:\\eclipse\\
...
6. Update the required libraries, have the followings listed:
Servlet=> JBuilder's Servlet.jar
XML => Point to xerces.jar to be found in the directory used in step 5
eclipse => eclipse.jar under eclipse\lib\
jdbc => oracle/ora81/jdbc/lib/classes12.zip & oracle/ora81/jdbc/lib/nls_charset12.zip
7.The project should compile successfully now.
8. Now update JRun: login to admin page from port 8000. In JRun Default Server

*Create a new application under web applications with the following parameter values

Application Name: GCMS
Application Root Directory: C:\JBuilder4\projects\GCMS\src\edu\mit\gcms [put your GCMS-dir here]
Application Mapping: /gcms

*Application Variables under GCMS should be as follows [use your own GCMS-dir, I provide mine below]
gcmsPath
C:\\JBuilder4\\Projects\\GCMS\\src\\edu\\mit\\gcms

applicationPath
C:\\JBuilder4\\Projects\\GCMS\\src\\edu\\mit\\gcms\\demo\\application

eclipsePath
C:\\eclipse\\

dbName
coin

dbUser
system

eclipseApplicationPath
//C/JBuilder4/projects/GCMS/src/edu/mit/gcms/demo/application/

dbPassword
manager

*MIME Type Mappings Under GCMS should be as follows
MIME Type Extension
MIME Type

.pl
plain/text

*Java Settings under JRun Default Server should be as follows [again use your own variations]

Java Executable

C:\JBUILDER4\jdk1.3\bin\javaw.exe
Path to your JVM executable

System.out Log File
{jrun.rootdir}/logs/{jrun.server.name}-out.log
Location where System.out messages appear

System.err Log File
{jrun.rootdir}/logs/{jrun.server.name}-err.log
Location where System.err messages appear

JRun Control Port
53000
Port used by JRun to send server commands

Classpath
{jrun.rootdir}/servers/lib{jrun.server.rootdir}/lib
C:/eclipse/lib/eclipse.jar
C:/oracle/ora81/jdbc/lib/classes12.zip
C:/oracle/ora81/jdbc/lib/nls_charset12.zip
C:/<GCMSRoot>/classes
Additional classpath entries

Java Arguments

Additional command-line arguments passed to the Java Executable

Library Path
{servlet.jnipath};{ejb.jnipath};C:\eclipse\lib\i386_nt;C:\oracle\ora81\bin
Directory of native JNI

9. The system should now be functional at http://localhost/gcms/demo/Demo.jsp