A Planner/Optimizer/Executioner for Context Mediated Queries.

by

Kofi Duodu Fynn

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

© Kofi Duodu Fynn, MCMXCVII. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by. . . . .
Dr. Michael Siegel
Principal Research Scientist, Sloan School of Management
Thesis Supervisor

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

A Planner/Optimizer/Executioner for Context Mediated Queries.

by

Kofi Duodu Fynn

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and implementation of a planner/optimizer/executioner (POE) for the COntext INterchange (*COIN*) system, . The POE takes a context-mediated query from the Mediation Engine of the system and returns the results of the query to the user of the system. Since context-mediated queries are composed of sub-queries to multiple data-sources, the POE plans the execution order of the sub-queries , executes each sub-query, composes the results and returns this as the answer of the mediated query. The thesis also describes the design of helper components of the POE such as the Capability Checker. Finally, the thesis describes a preliminary performance analysis study which was used to determine the allocation of costs in the *COIN* system.

Thesis Supervisor: Dr. Michael Siegel
Title: Principal Research Scientist, Sloan School of Management

Acknowledgments

I am African. Africans always write long acknowledgements and so please bear with the length of this one.

First of all, I would like to thank Professor Stuart Madnick and Dr. Michael Siegel for giving me the opportunity to participate in a great project, and for their invaluable advice and support. I would also like to thank the whole COntext INterchange team for providing a great working environment for me. In particular, I would like to thank Dr. Stéphane Bressane for his wonderful insights and suggestions. My gratitude also goes to Tom Lee and Fortunato Peña for the support and encouragement. My thanks also to Dr. Cheng Goh for making a lot of my thesis work possible.

The journey has been long and hard, and I could not have made it without the support of numerous friends and teachers and relatives. My gratitude goes out to Mrs. Comfort Engmann and Mr. Quarcoo-Tchire, and Dr. K. Oppong, three individuals whose influence is still visible in my life.

To my all my friends. Thank you for putting up with me, and encouraging me to stay focused. Dudzai, Wasi, Lee, Kamel, Folusho and Ebow. I could not have done this without you guys. Poks, thanks for being there for me. Your presense kept me from packing up and returning to Ghana. Benjy, Steve, Nana Benneh, Kafui and Kwame. I hope I have been a better thesis writer than I am a letter writer. I will also like to thank all the friends of the Fynn family.

To my families, the Arkhursts and the Fynns, thank you. The combined efforts of twenty uncles and aunts and their families paved the path for my success. Thank you for your prayers and support. Especially Sister Efuwa, Auntie Adwoa, Auntie Ama and Auntie Nana Johnson.

I will like to thank my siblings Kojo Ennin, Kofi Nsarko, Ekuwa, Esi Danquah, Ama Manfoa and Kwesi Debrah. Your love was given unconditionally, and you always provided a warm place of rest. Thanks Danny boy and Kwesi.

Thank you Elisa, for your love, encouragement and support. Your efforts during the past two months provided me with the peace of mind required to complete this work.

Finally, I would like to acknowledge my parents Kodwo Debrah and Maenoba Rose. We have done it. Without you none of this would be possible, and to you I dedicate this thesis.

Contents

List of Figures

6

List of Tables

Chapter 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.

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 to any of 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 other sources which have been wrapped to give a relational interface. 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. An example of a client process is the multi-database browser [7]. The second set of process in the *COIN* system are the mediator processes which consist of the context mediator, the planning components, and the execution engine. These processes provide all the mediation services. Together, they resolve the semantic conflicts which exist in the query and also generate and execute a plan for the query. Finally, the server processes provide the physical connectivity to the data-sources. They provide a uniform interface for accessing both relational databases and wrapped web-sources. Special components called wrappers provide the relational interface to the web-sources. The mediator processes are therefore insulated from the idiosyncrasies of different database management systems.
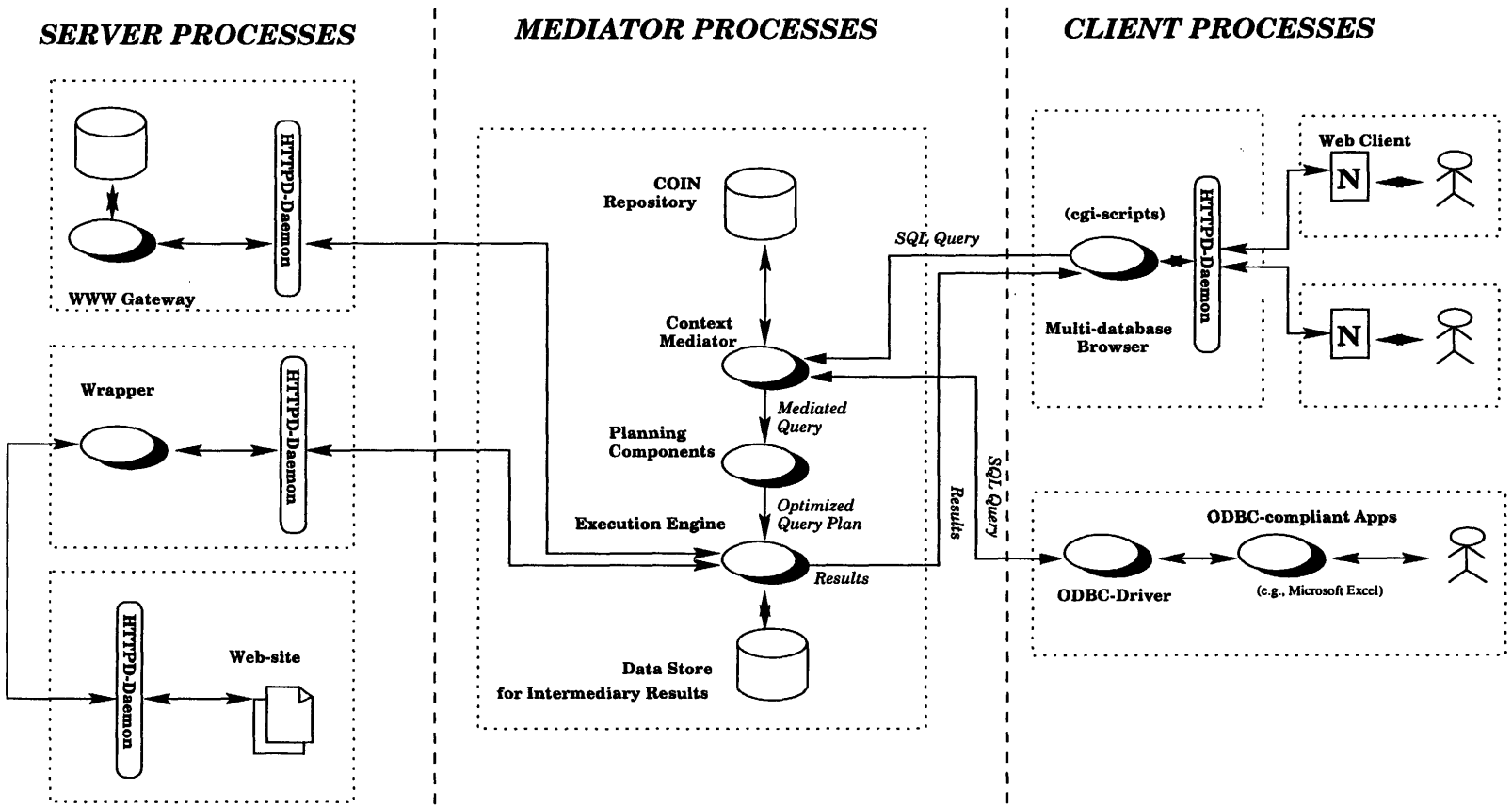
Figure 1-1: Architectural Overview of the Context Interchange Prototype

SERVER PROCESSES

- HTTPD-Daemon
- WWW Gateway
- Wrapper
- HTTPD-Daemon
- HTTPD-Daemon
- Web-site

MEDIATOR PROCESSES

- COIN Repository
- Context Mediator
- Planning Components
- *Mediated Query*
- Execution Engine
- *Optimized Query Plan*
- *Results*
- Data Store for Intermediary Results

CLIENT PROCESSES

- *SQL Query*
- (cgi-scripts)
- HTTPD-Daemon
- Web Client
- N
- Multi-database Browser
- N
- *SQL Query*
- *Results*
- ODBC-compliant Apps
- ODBC-Driver
- (e.g., Microsoft Excel)

9

This thesis describes the items labelled *Planning Components* and *Executioner* in Figure 1-1. Together, they form the Planner/Optimizer/Executioner (POE). The POE is responsible for decomposing the mediated query into its sub-queries, and for planning the order of execution of these sub-queries. It is also responsible for executing the sub-queries to retrieve the result of the mediated query. The POE does not perform context mediation per se. It is a standard multi-database back-end with the added functionality of handling non-relational query operators, and access to non-standard data-sources.

1.1 Background

A lot of work has already been done in the area of ordering, decomposing, optimizing and executing queries to multiple data-sources. Significant among these are the work in [9] which proposes an algorithm for ordering the subgoals of a query and [3] which discusses query optimization in heterogenous database systems. There are also a number of current efforts to develop languages for describing the capability restrictions imposed by sources. Examples of research projects which are tackling these problems are the Information Manifold (IM) Project at AT & T, the Tsimmis Project at Stanford, and the Garlic Project at IBM. IM uses the notion of a capability record [8] for representing source capability restrictions. Tsimmis uses a language called the Query Description and Translation Language (QDTL) [10] while Garlic uses a language called the Relational Query Description Language (RQDL) [11]. Both QDTL and RQDL are capable of describing arbitrary schemas. However, to our knowledge, the algorithms they use for integrating source capability descriptions in the query planning phase only plan among the different components of a single sub-query and not among different sub-queries. For example, they do not consider the possibility of using information from one source to provide the information required by other sources. They only use information to remove restrictions within a single source.

1.2 Motivational Example

An example will help us clarify some of the issues involved in planning and executing context mediated queries.

In this example, a hypothetical user wants the names, net incomes and last selling prices of all companies with net income greater than 2,500,000. This information is provided by the two relations DiscAF and secapl. DiscAF provides historical data on a variety of international companies, while secapl provides market information on companies. DiscAF is in the context c_ds. In this context, all financial figures have a scale-factor of 1, and are published in the currency of the country where the company is located. secapl is in the context c_ws, in which all financial figures have a scale-factor of 1,000 and are published in US Dollars. The hypothetical user is also in the context c_ws. The export schemas and the contexts of all the relations used in our example are listed in Appendix B. [5] gives a more complete discussion of contexts and mediation.

To obtain his/her answer, the user would input the following query to the *COIN* system.

context = c_ws
select DiscAF.COMPANY_NAME, DiscAF.NET_INCOME,
secapl.Last
from Ticker_Lookup, DiscAF, secapl
where Ticker_Lookup.TICKER = secapl.Ticker
and Ticker_Lookup.COMP_NAME = DiscAF.COMPANY_NAME
and DiscAF.NET_INCOME > 2,500,000;

After mediation, the Context Mediator will output a list containing the datalog [2] queries shown on in Table 1.1.

answer(V20, V13, V8):-
    Ticker_Lookup(V6, V5, V4),
    Name_Map(V6, V20),
    DiscAF(V6,V3, V10, V2, V9,V18,V12),
    '2,500,000' < V13,
    V13 is V2 * 0.001,
    secapl(V5, V8, V1, V7),
    V12 = 'USA'.

answer(V20, V17, V8):-
    Ticker_Lookup(V6, V5, V4),
    Name_Map(V6, V20),
    DiscAF(V6, V3, V10, V2, V9, V18, V12),
    '2,500,000' < V17,
    secapl(V5, V8, V1, V7),
    V12 <> 'USA',
    Currencytypes(V12,V14),
    olsen(V14,V15,V13,V19),
    V15 = 'USD',
    V16 is V13 * V2,
    V17 is V16 * 0.001.

Table 1.1: Mediator Output for 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. B is usually a conjunction of facts. The procedural reading for the rule A:-B is: "To answer the query A, we must first answer B." In Table 1.1, 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. However, the user can also add such functions into the initial query to the *COIN* system. Since B is a conjunction, we need to answer all the facts in B

11

before we can get the result of the mediated query. Once we have an answer for B, A will have an answer and therefore the mediated query would have an answer.

Relational predicates represent queries to data-sources. For example, the predicate Ticker_Lookup(V6,V5,V4), represents the relation Ticker_Lookup in the source Disclosure. Each argument in the predicate represents an attribute of Ticker_Lookup. The order of the arguments correspond directly to the order of the attributes in the relation, as shown in Appendix B. Thus, V6 represents Ticker_Lookup.COMP_NAME, V5 represents Ticker_Lookup.TICKER and V4 represents Ticker_Lookup.EXCHANGE. Variables which are shared by different facts must instantiate to the same value. In our example, the first attribute of Ticker_Lookup and the first attribute of Name_Map must instantiate to the same value because they are both represented by the variable V6. Thus, shared variables in the relations represent equijoins 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, V8 is shared by secapl() and answer(). Therefore, the second attribute in secapl() would have to be returned as part of the answer to the query. Finally, shared variables are used to show which attributes in the query are arguments to boolean statements and conversion functions.

Boolean statements represent selection conditions on attributes in the query. A statement like V15 = 'USD' restricts the value of the second attribute of the relation olsen() to 'USD'. Conversion functions represent the operations needed for context conversion. In our example, the first datalog query has the conversion function V13 is V2 * 0.001. V2 is the fourth attribute of the relation DiscAF(). V13 is the third argument in the answer() predicate. The conversion function represents the fact that in order for the fourth attribute of DiscAF() to be returned to the user, its value would have to be multiplied by a scale-factor of 0.001.

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 the company is incorporated in the USA, while the second handles the conflicts which occur when the company is incorporated in another country other than the USA. In the first datalog query, the boolean statement ensures that the value of DiscAF.LOCATION_OF_INCORPORATION is 'USA'. There is no need for a conversion of the currency of financial figures once this condition is satisfied. The only conversions that needs to be performed are a translation of the name of the company from the format of c_ds to the format of c_ws, and a conversion of the scale factor. In the second datalog query, the boolean statement ensures the value of DiscAF.LOCATION_OF_INCORPORATION is not 'USA'. In that case a currency conversion is needed in addition to the conversions mentioned previously.

We will use the second datalog query in the list to outline the issues involved in retrieving an answer to a query. The reader can extend the explanations we provide to the first datalog query.

The relational predicates Ticker_Lookup(), DiscAF(), and secapl() represent the relations which were identified by the user in the original SQL query. Name_Map(), olsen() and Currencytypes() are auxiliary relations that provide information for mediation. Since the user in in the c_ws context, she expects all financial figures to have

a currency of US Dollars and a scale-factor of 1. olsen() provides the current exchange rate from the currency of the location of incorporation of the company to US Dollars. Name_Map() provides a translation of the company name from the format of the DiscAF() context to the user context, and Currencytypes() provides a mapping between countries and their currencies. The query also consists of a boolean statement and two conversion function.

Any execution of this query would involve an access to DiscAF() to retrieve the information which the user needs. Here, we can take advantage of the fact that we do not require all the attributes from DiscAF. We can just project out the attributes which we need from this relation. Since the query also asks for information from secapl(), we would also have to access that relation. Unfortunately, secapl() cannot be accessed in the same way as DiscAF because it imposes a restriction on queries sent to it. Every query to secapl() must have the attribute secapl.Ticker bound to a constant. The only place where we can get these bindings is from the relation Ticker_Lookup(). We first have to access Ticker_Lookup() and for each ticker symbol, retrieve the required data from secapl(). Alternatively, we could restrict the number of tickers by using only the tickers of companies which were returned from DiscAF.

Because the user expects all financial data in US Dollars, each financial figure requested by the user will have to be converted to US Dollars. This conversion requires the exchange rate between the currency of the financial figure and US Dollars. The relation olsen() provides this information. However, it imposes the restriction that the olsen.Exchanged and olsen.Expressed attributes have to be bound. To get around this restriction, we use the LOCATION_OF_INCORP attribute of DiscAF to retrieve the currency from Currencytypes(). Once we have the currency, we can retrieve the exchange rate and use it to convert the DiscAF financial figures to US Dollars. We can then apply the scale-factor conversion function to the result. Finally, we can apply the user-imposed conditions on the converted results. This marks the end of execution for the query, and the results can be returned to the user.

This relatively simple example provides us with some insight into some of the steps necessary for retrieving an answer for the user. To answer a context mediated query, the POE needs to ensure that queries that are sent to sources can be handled by the sources. The POE also needs to make the implicit order of execution of sub-queries explicit. This involves the transformation of the logical description provided by the context mediation engine into an operational description which describes the order of retrieval of data from the remote source. The operational description would also have to describe the manner in which data are combined to provide the answer that the user expects. Finally, the operations specified by the operational description of the context mediated query have to be executed. An execution of the operations involves accessing the remote data-sources and retrieving the information required for the query. The information then has to be composed into the answer requested by the user.

1.3 Organization of the Thesis

The remainder of this thesis provides a more in-depth look at the POE components of the *COIN* architecture. Chapter 2 presents an overview of the design and implementation of whole the POE. The next two chapters present a description of the two major components of the POE. In Chapter 3 the Execution Engine is described while Chapter 4 provides a description of the Planner. Finally, Chapter 5 describes a framework for performance analysis studies in the *COIN* back-end and presents the results of a preliminary study. Chapter 6 concludes the thesis with recommendations for future work in this area.

Chapter 2

The *COIN* system Backend

This chapter presents an overview of the design and implementation of the whole POE.

2.1   Architectural Design
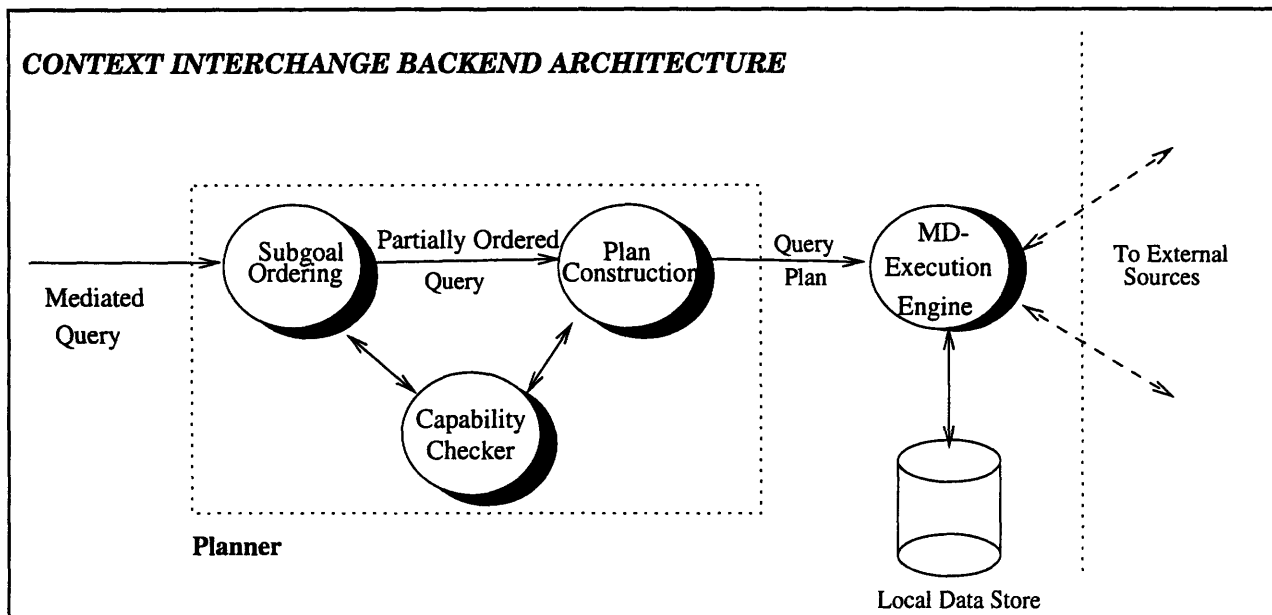
Figure 2-1: The Context Interchange Backend Architecture

The backend is divided into two main modules: the Query Planner and the Multi-Database Executioner. Together, they ensure that the answer to the mediated query is retrieved from the remote data-sources and returned to the user.

The Planner checks the query to ensure that a plan which will produce an answer to the initial query exists. Once the planner determines this, it generates a set of constraints on the order in which the different sub-queries can be executed. These

constraints are sometimes necessary because some data-sources in the *COIN* system impose restrictions on the kinds of queries which they accept. Typically, some sources require that bindings must be provided for some of their attributes. Sources might also have restrictions on the types of relational operators which they can handle. Under these constraints, standard optimization heuristics are applied to generate the query execution plan. The planner is subdivided into three separate submodules. These are the capability checker, the subgoal ordering submodule, and the plan construction submodule. These submodules will be discussed in Chapter 4.

The execution engine executes the query plan. It dispatches the sub-queries to the remote sources and combines the results obtained from the external data-sources. Intermediate results are stored in a local data store.

The decision to separate the planning issues from the issues involved with retrieving and composing the of answer to queries increases the modularity of the system and ensures that our infrastructure is extensible. For example, plans can be stored and reused by the execution engine without the need for intervention from the planner. Also, changes can be introduced into either the planner or the execution engine without having adverse effects on other parts of the system. More importantly, the execution engine is isolated from the context mediator itself, ensuring that changes in the output language of the mediator will have no effect on it. Thus, we can design and implement the execution engine, keeping only the actual relational operations in mind.

2.2   Implementation Decisions

2.2.1   Implementation Language

The *COIN* system was developed on the *ECLiPSe*[1] platform which is a parallel constraint logic programming environment with database capabilities. The kernel of *ECLiPSe* is an implementation of Standard Prolog [13]. The main reason for choosing Prolog as our implementation language was because of the adequacy of Prolog for implementing problems which involve a significant amount of symbolic manipulation. Moreover, this specific Prolog implementation gave us the added bonus of a multi-indexing temporary data store called BANG. BANG provided the system with an efficient way to access intermediate query results of ad hoc queries.

2.3   Communication Protocol

We relied on the Hypertext Transfer Protocol (HTTP) for the physical connectivity between the remote data-sources and the execution engine. This protocol provided a general, widely used communication medium which facilitated the integration of new data-sources into the *COIN* system. Moreover, because our querying paradigm is

---

[1]ECLiPSe: The ECRC Constraint Logic Parallel System. More information can be obtained at http://www.ecrc.de/eclipse/.

very similar to the stateless design of the HTTP protocol, this protocol meshed well with our system, and did not have to be customized.

Chapter 3

The Execution Engine

Let us consider the first datalog query presented in the motivational example in Section 1.2. To answer this query, we would need to execute all the operations specified in the body of the query. These operations can be described by a Query Execution Plan (QEP). For our example, the QEP would have to specify how each of the sources would be accessed, and what data would need to be retrieved from each data-source. Moreover, the QEP would have to specify how data retrieved form each of the three data-sources would be combined. Finally, it would need to show when the conversion function and the boolean conditions would be applied, and which attributes they would have to be applied to. The QEP also specifies which attributes would be returned to user as the answer to the query.

The multi-database execution engine implements all the operations specified by the QEP. Figure 3-1 shows a diagram of the engine. It takes a QEP as its input. The engine executes all the operations that are specified in the QEP. An example of a QEP is shown in Figure 3-2. It retrieves data from the data-sources, processes the data, and returns the answer to the user. In addition to the standard **select, project, join**, and **union** relational operators, it implements a **cvt** operator for conversion functions, and a **join-sfw** operator for access to sources whose input binding restrictions have to be satisfied by data from other relations in the query plan. These bindings can be provided only at run-time.

Apart from the **project** operator, each relational operator corresponds to a node in the QEP. The **project** operator is implicitly provided in each node as we will show in the discussion below.

3.1   The Query Execution Plan

The Query Execution Plan (QEP) is a data structure that specifies execution of the operations in the query. It is in the form of an algebraic operator tree in which each operation is represented by a node. We classify the nodes into two groups. These are access nodes and local nodes. Access nodes are nodes which represent access to some data-source, while local nodes represent operations in the local execution engine.
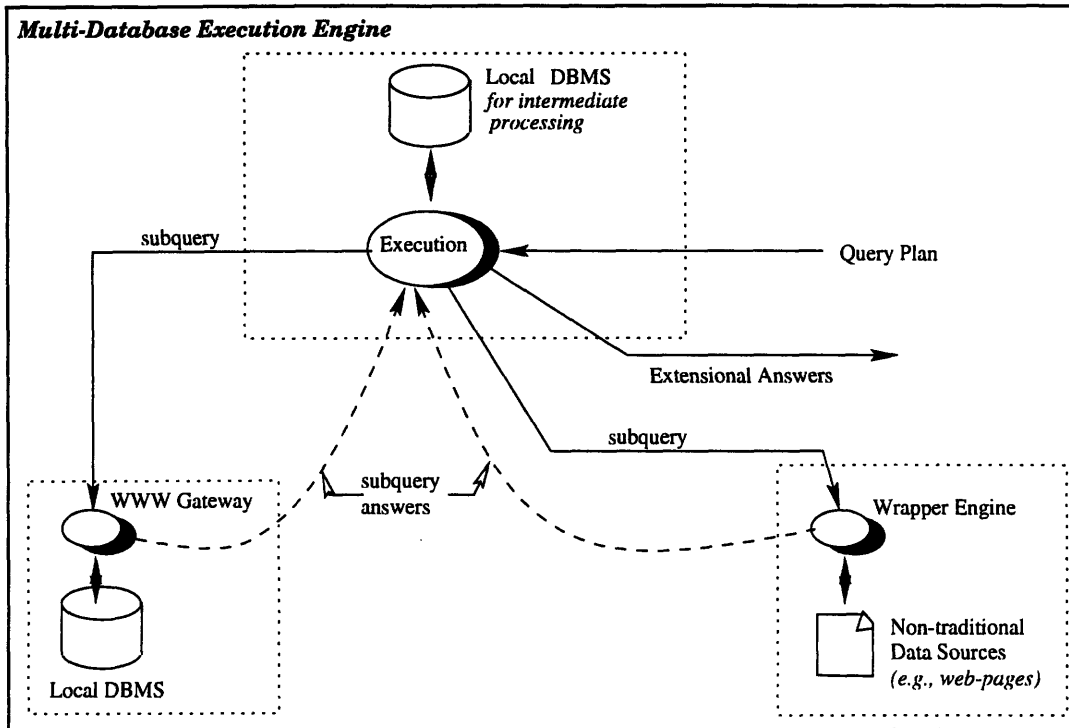
Figure 3-1: The Multi-Database Execution Engine

3.1.1 Access Nodes

There are two access nodes. These are the **sfw** node and the **join-sfw** node. The sfw node represents access to data-sources that do not require input bindings from other sources in the query.[1] **join-sfw** nodes on the other hand represent access to data-sources which require bindings from other data-sources in the query.

The sfw Node

The data-structure for the **sfw** node has the following form:

sfw(Relation,Sourcename,Maplist,Projectionlist,Conditionlist)

Each component of this data structure is described below.

Relation - The name of the relation.

Sourcename - The name of the source containing the relation.

---

[1] **sfw** nodes do not always adhere to this access/local categorization of nodes. This is because in queries which have constants that are not grounded in any relation, a temporary relation is created during execution and the constants are inserted into this relation. In that situation, the **sfw** node associated with this temporary relation represents a local operation. We will clarify this issue in the last section.

19

Maplist - This is a list of the actual names of the attributes in the relation. The Maplist is required by the multidb executioner when it is composing the SQL query to access the relation.

Conditionlist - This is the list of conditions that can or must be pushed to the data-source. We need to remember that not all conditions relevant to the data-source can be pushed to it. Sometimes, relevant queries have to be applied at the local database because they cannot be handled at the actual data-source.

Projectionlist - This is the list of variables which have to be projected from the relation. The membership of this list depends on attributes involved in the the implicit join conditions in the query, attributes which are arguments of conversion functions, and attributes in the final projection list of the query. It also depends on attributes which are arguments in the relevant conditions that could not be pushed to the data-source. All such attributes have to be projected out of the data-source.

Each attribute in the Conditionlist and the Projectionlist is replaced by an index into the Maplist. An index is of the form att(X,Y) where X is the number of the child data-structure and Y is the position of the attribute in the Maplist of that data-structure. In the case of the **sfw** node all indices are of the form att(1,Y). In nodes which have two child data-structures, the indices are att(1,Y) and att(2,X) depending on which child data-structure the attribute belongs to.

The join-sfw Node

The data-structure for the **join-sfw** node is shown below.

join-sfw(Relation,Sourcename,Tree,Maplist,Projectionlist,Conditionlist)

This node can be thought of as representing a nested-loops join. The values for the attributes that require bindings are projected out of the data-structure represented by Tree and incorporated into the condition list of the SQL query to the data-source. The Maplist is the same as the Maplist for an **sfw** node. Attributes in the Projectionlist and the Conditionlist are replaced by indices into the Maplist and the Projectionlist of Tree. An index into Maplist is of the form att(1,X) while an index into the Tree is of the form att(2,Y).

3.1.2 Local Nodes

There are four local nodes. These are the **join** node for joining intermediate query results, the **cvt** node for applying conversion functions to map data from one context to another, the **select** node for applying conditions to intermediate results and the **union** node for taking the union of the results of queries.

The join Node

The data-structure for a **join** node is

$$\text{join}(\text{Tree1}, \text{Tree2}, \text{Projectionlist}, \text{Conditionlist})$$

Tree$x$ - The data-structure for a subtree being joined in.

Conditionlist - A list of the join conditions. Each attribute is replaced by an index into the Projectionlist of Tree1 or Tree2.

Projectionlist - This is similar to the projection list of an access node. The only difference is that attributes are replaced by indices into the Projectionlists of Tree1 and Tree2.

The select Node

$$\text{select}(\text{Projectionlist}, \text{Tree}, \text{Conditionlist})$$

The **select** node is used to apply conditions to intermediate results. Usually, these are usually conditions that could not be applied at the remote data-source, or conditions which act on the output of conversion functions. The Projectionlist and Conditionlist are similar to those for the **join** node. Each attribute is replaced by an index into the Maplist of Tree.

The cvt Node

$$\text{cvt}(\text{Predicate}, \text{Tree}, \text{Projectionlist}, \text{SpecialConditionlist})$$

The **cvt** node is just used to apply conversion functions to intermediate query results. It has two components which make it different from the other nodes.

Predicate - This represents the conversion function to be applied. [2]

SpecialConditionlist - This is a list which maps the arguments of the conversion functions to indices into the projection list of the subtree.

The union Node

$$\text{union}(\text{Querylist})$$

The **union** node represents a union of the results obtained by executing the members of the list Querylist.

21

JOIN-SFW:Secapl
Source: Secapl
MapL: [(secapl, Ticker), (secapl, Last), (secapl, Date), (secapl, Shareout)]
ProjL:[att(2, 2), att(2, 3), att(1, 2),]
CondL: [att(1, 1) = att(2, 1)]

    SELECT
    ProjL:[att(2,2), att(2,3), att(2,1)]
    CondL: [att(2,1) > 2,500,000]

        CVT
        ProjL:[att(1,1), att(2,1),att(2,2)]
        SpecialCondL:[att(1,1) is att(2,3) * 0.001]

            JOIN
            ProjL:[att(2, 1), att(1, 1), att(1,2)]
            CondL: [att(1, 1) = att(2, 2)]

                SFW:DiscAF
                Source: Disclosure
                MapL:[('DiscAF', 'COMPANY_NAME'),
                    ('DiscAF', 'LATEST_ANNUAL_DATA'),
                    ('DiscAF', 'CURRENT_SHARES_OUTSTANDING'),
                    ('DiscAF', 'NET_INCOME'),
                    ('DiscAF', 'NET_SALES'), ('DiscAF', 'TOTAL_ASSETS'),
                    ('DiscAF', 'LOCATION_OF_INCORP')]
                ProjL: [att(1, 1),att(1, 4)]
                CondL: [('USA' = att(1, 7))]
                JOIN
                ProjL: [att(2, 2), att(1, 2)]
                CondL: [att(1, 1) = att(2, 1)]

                    SFW Name_map
                    Source: Disclosure
                    MapL: [(Name_map, DS_NAMES), (Name_map, WS_NAMES)]
                    ProjL:[att(1, 1), att(1, 2)]
                    CondL: [ ]
                    SFW:Ticker_Lookup
                    Source:Disclosure
                    MapL: [(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,
                      TICKER), (Ticker_Lookup, EXCHANGE)]
                    ProjL: [att(1, 1), att(1, 2)]
                    CondL: [ ]

Figure 3-2: Plan for first query from Section 1.2

22

JOIN-SFW:Secapl
Source: Secapl
MapL: [(secapl, Ticker), (secapl, Last), (secapl, Date), (secapl, Shareout)]
ProjL:[att(2, 2), att(2, 3), att(1, 2),]
CondL: [att(1, 1) = att(2, 1)]
     SELECT
     ProjL:[att(2,2), att(2,3), att(2,1)]
     CondL: [att(2,1) > 2,500,000]
         CVT
         ProjL:[att(1,1), att(2,1),att(2,2)]
         SpecialCondL:[att(1,1) is att(2,1) * 0.001]
            CVT
            ProjL:[att(1,1), att(2,1),att(2,2)]
            SpecialCondL:[att(1,1) is att(2,3) * att(2,4)]
                JOIN-SFW:olsen
                Source: Olsen
                MapL: [(olsen,Exc),(olsen,Exp),(olsen,Rate),(secapl, Date)]
                ProjL:[att(2, 1), att(2, 2), att(2,3), att(1,3)]
                CondL: [att(1, 1) = att(2, 4), att(1,2) = 'USD']
                    JOIN
                  ProjL:[att(2, 1), att(2,2), att(2,3), att(1,2)]
                  CondL: [att(1, 1) = att(2, 2)]
                      SFW:Currencytypes
                      Source: Currencytypes
                      MapL:[('Currencytypes', 'COUNTRY'),
                            ('Currencytypes', 'CURRENCY')]
                      ProjL: [att(1, 1),att(1, 2)]
                      CondL: [ ]
                  JOIN
                  ProjL: [att(2, 1), att(1, 1), att(1,2)]
                  CondL: [att(1, 1) = att(2, 2)]
                      SFW:DiscAF
                      Source: Disclosure
                      MapL: [('DiscAF', 'COMPANY_NAME'),
                            ('DiscAF', 'LATEST_ANNUAL_DATA'),
                            ('DiscAF', 'CURRENT_SHARES_OUTSTANDING'),
                            ('DiscAF', 'NET_INCOME'),
                            ('DiscAF', 'NET_SALES'), ('DiscAF', 'TOTAL_ASSETS'),
                            ('DiscAF', 'LOCATION_OF_INCORP')]
                      ProjL: [att(1, 1),att(1, 4),att(1,7)]
                      CondL: [('USA' <> att(1, 7))]
                  JOIN
                  ProjL: [att(2, 2), att(1, 2)]
                  CondL: [att(1, 1) = att(2, 1)]
                      SFW:Name_map
                      Source: Disclosure
                      MapL: [(Name_map, DS_NAMES), (Name_map, WS_NAMES)]
                      ProjL:[att(1, 1), att(1, 2)]
                      CondL: [ ]
                      SFW:Ticker_Lookup
                      MapL: [(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,
                          TICKER), (Ticker_Lookup, EXCHANGE)]
                      ProjL: [att(1, 1), att(1, 2)]
                      CondL: [ ]

23

Figure 3-3: Plan for second query from Section 1.2

3.2 Executing a Query Plan

We will ground our discussion of how the execution engine executes a plan with the example presented in Section 1.2.

Figure 3-2 shows the query plan for the first datalog query while Figure 3-3 shows the query plan for the second query. After executing both of these query plans, their results would be combined together with a **union** node which is not shown. I will only consider the execution of the plan shown in Figure 3-2 for the following discussion.

Starting at the root node of the query plan, the execution engine begins the execution of the operation associated with each node. It first executes the operations associated with the subtree or subtrees of each node. Then it applies the operation for the node on the result of executing the subtree. The execution engine descends the query plan in a depth-first manner until it encounter an access node. For each access node in the plan, an SQL query is composed. This query is sent to the remote data-source which the access node represents. Any results returned by the query are inserted into temporary relations in the local data store. Once all the sub-trees of a local node have been executed, the operation associated with the local node is applied to the temporary relations which were created from the results of the sub-trees. This procedure continues until the root of the query plan is executed. The execution engine then takes the next query in the **union** node and executes it in the manner just described. Once all the results from the queries in the **union** node have been inserted into temporary relations, a union is done of all the relations. The result of the union is the answer to the query, and this is sent back to the user.

For our example query, the execution engine begins at the **join-sfw** node secapl(). This has only one sub-tree. The sub-tree also executes its child sub-tree. Subtrees are executed recursiverly until a leaf is reached. In our example, the first leaf node is the **sfw** node for Ticker_Lookup(). The node is translated into an SQL query which is then submitted to the source which contains Ticker_Lookup. The query to Ticker_Lookup is

> **select** Ticker_Lookup.COMP_NAME, Ticker_Lookup.TICKER
> **from** Ticker_Lookup;

The next node is the **sfw** node for Name_Map(). This results in the following query:

> **select** Name_Map.DS_NAMES, Name_Map.WS_NAMES
> **from** Name_Map;

These two nodes are sub-trees of a **join** node, and once they have been executed, the **join** node itself is executed. The **join** node represents an equi-join Name_Map.DS_NAMES and Ticker_Lookup.COMP_NAME. Name_Map.WS_NAMES and Ticker_Lookup.TICKER are projected out of the result of this join as the result of the executing the **join** node.

---

[2]An example of a conversion function would be X is $Y * Z$

The next node to be executed is the **sfw** node which represents a query to the relation DiscAF(). The resulting SQL query from this node is

> **select** DiscAF.COMPANY_NAME DiscAF.NET_INCOME
> **from** DiscAF
> **where** DiscAF.LOCATION_OF_INCORP = 'USA';

The results of this query are joined in with the results of the join mentioned previously. As the result of each subsequent sub-tree becomes available, the operation associated with the node which covers it is executed. Thus, the conversion function is executed on the result of the **join** node. A selection is made on the result of executing this **cvt** node, and finally the **join-sfw** node is executed. The execution of the **join-sfw** node needs special mention. All through the execution of the previous nodes, it can be seen from examination of the query plan, that the attribute Ticker_Lookup.Ticker is always projected up. In the **join-sfw** node, this attribute is joined to the attribute secapl.Ticker. This node represents an operation similar to a nested loops join, in that each Ticker_Lookup.Ticker which is projected up is incorporated into a query to secapl(). Thus, the query to secapl() would be of the form

> **select** secapl.Ticker
> **from** secapl
> **where** secapl.Ticker = *Ticker_Lookup.Ticker value*;

and *Ticker_Lookup.Ticker value* will be replaced by one value of Ticker_Lookup.Ticker. The query will be executed for as many values of Ticker_Lookup.Ticker as there are in the result of the **select** node.

Finally, attributes are projected out of both the result of the **select** node, and the result of this "nested-loops join". The result of the projection is the result of the query.

Once all the queries in the list of the **union** node have been executed, their results are unioned together to give the result of the mediated query. This result is then returned to the user.

3.3   Constants in the Head and Existential Queries

In the previous section, we described the standard way that the execution engine implements the relational operators. However, there are two instances when the execution engine uses alternate interpretations for some of the algebraic operators. These situations arise when there are constants in the head of the datalog query, or when some of the facts in the body of the datalog query are purely existential.

3.3.1   Constants in the Head

Let us imagine that our hypothetical user from our example in Section 1.2 decides to modify his/her query. She now wants to know the country of incorporation of the

companies that are returned in the answer to his original query. She would input the following query to the *COIN* system.

> **context** = c_ws
> **select** DiscAF.COMPANY_NAME, DiscAF.NET_INCOME,
> secapl.Last, DiscAF.LOCATION_OF_INCORP
> **from** Ticker_Lookup, DiscAF, secapl
> **where** Ticker_Lookup.TICKER = secapl.Ticker
> and Ticker_Lookup.COMP_NAME = DiscAF.COMPANY_NAME
> and DiscAF.NET_INCOME > 2,500,000;

The new output from the mediation engine will be a list containing the following two queries.

> answer(V20, V13, V8, 'USA'):-
>   Ticker_Lookup(V6, V5, V4),
>   Name_Map(V6, V20),
>   DiscAF(V6,V3, V10, V2, V9,V18,V12),
>   '2,500,000' < V13,
>   V13 is V2 * 0.001,
>   secapl(V5, V8, V1, V7),
>   V12 = 'USA'.

> answer(V20, V17, V8, V12):-
>   Ticker_Lookup(V6, V5, V4),
>   Name_Map(V6, V20),
>   DiscAF(V6, V3, V10, V2, V9, V18, V12),
>   '2,500,000' < V17,
>   secapl(V5, V8, V1, V7),
>   V12 <> 'USA',
>   Currencytypes(V12,V14),
>   olsen(V14,V15,V13,V19),
>   V15 = 'USD',
>   V16 is V13 * V2,
>   V17 is V16 * 0.001.

Table 3.1: Mediator Output for Constants in the Head Example.

The change in the second datalog query in the list is not significant. It only reflects the fact that one more attribute has been requested by the user. However, the change in the first datalog has some serious implications because the value of the new attribute requested by the user is already known. Instead of using a variable in the answer() predicate to represent the attribute, the mediation engine inserts the actual value of the attribute into the predicate. Thus, the head of the query becomes answer(V20, V13, V8, 'USA'). We refer to 'USA' as a constant in the head.

None of the nodes presented in the the QEP can be used to express the constant in the head. This is because the Projectionlists of the nodes are written in terms of

indices into the Maplists of access nodes, or indices into the Projectionlists of local nodes. Because the constant in the head is not ground in any relation, it cannot be found in any Maplist or Projectionlist, and therefore cannot be projected out by any of the standard nodes of the QEP.

To circumvent this problem a new kind of node was introduced into the QEP. This node represents a relation which contains all the constants in the head of the query.

$$\textbf{sfw}(\text{Maplist})$$

The standard nodes of the QEP can now refer to these constants using indices into the relation which the node represents. We refer to this node as an **sfw** node because the operation associated with it is very similar to the operation of a standard **sfw** node. When the execution engine encounters this node in a query plan, it creates a relation in the local data-store and inserts the constant values into this relation. Standard relational operations such as joins and projections can then be applied to this relation in order to get the answer of the datalog query. The only difference between this node and a standard **sfw** node is that the values that are inserted into the relation are provided in the QEP and not retrieved from a remote data-source. The Maplist in the node contains the type and value of each constant in the head and is used to create the relation in the local data-store.

3.3.2 Existential Queries

To motivate our discussion on the problems associated with executing existential queries, we will need to present a completely new example. Let us suppose that the hypothetical user wants to sell off Microsoft stocks in order to invest in German companies which have a net income greater than 2,500,000, but will do so only if the selling price for Microsoft stock is favorable. At this point, the user is not interested in specific companies. She only wants to know if an investment opportunity with the desired characteristics exists. With regard to the *COIN* system, the user would like to know if the relation DiscAF has entries for German companies with net income greater than 2,500,000. If such companies exist, then the user would like to know the last selling price for Microsoft stock. To obtain this answer, the user would input the following query to the *COIN* system.

> **context** = c_ws
> **select** secapl.Last
> **from** DiscAF, secapl, Ticker_Lookup
> **where** DiscAF.LOCATION_OF_INCORP = 'GERMANY'
> and DiscAF.NET_INCOME > 2,500,000
> and Ticker_Lookup.COMPANY_NAME = 'MICROSOFT'
> and Ticker_Lookup.TICKER = secapl.Ticker;

The output from the mediation engine will be a list containing the datalog query shown in Table 3.2.

```
answer(V20):-
      secapl(V1, V20, V3, V4),
      Ticker_Lookup(V5,V1,V13),
      DiscAF(V6,V13, V10, V2, V9,V18,V12),
      '2,500,000' < V13,
      V13 is V2 * 0.001,
      V12 = 'GERMANY',
      V5 = 'MICROSOFT'.
```

Table 3.2: Mediator Output for Existential Queries Example.

The operational translatation for this query would be the QEP shown in Figure 3-4.

An execution of this query in the manner described in Section 3.2 will sometimes return the wrong result. To clarify the issue we will rewrite the datalog query in Table 3.2. The rewritten query is shown in Table 3.3. We have split the original query into two parts. These are the existential query which we refer to with the predicate test(), and the query to secapl() and Ticker_Lookup().

```
answer(V20):-
          secapl(V1, V20, V3, V4),
          Ticker_Lookup(V5,V1,V13),
          V5 = 'MICROSOFT'.

test:-
          DiscAF(V6,V13, V10, V2, V9,V18,V12),
          '2,500,000' < V13,
          V13 is V2 * 0.001,
          V12 = 'GERMANY'.
```

Table 3.3: Mediator Output for Existential Queries Example.

We can see from Table 3.3 that the original datalog query is actually composed of two disjoint queries which do not share variables. test() only ensures that there exist tuples which satisfy the conditions in the query. Since the user does not require answer from the relations in test() and none of the attributes in test are required in the rest of the query, the projection from test() should be either the empty relation Φ, or $\{\epsilon\}$, the relation which contains no tuples. The former result should be returned if no tuples satisfy test(), while the latter should be returned if some tuples satisfy test().

We run into a problem with existential queries because BANG does not distinguish between Φ and $\{\epsilon\}$. It interpretes both relations as Φ. Thus, when the execution engine tries to perform a join between the relation created from accessing Ticker_Lookup() and the result of test() the result of the join would always be Φ. Obviously, that should not be the answer if the existential query is non-empty. If

JOIN-SFW:Secapl
Source: Secapl
MapL: [(secapl, Ticker), (secapl, Last), (secapl, Date), (secapl, Shareout)]
ProjL:[att(1, 2)]
CondL: [att(1, 1) = att(2, 1)]
    JOIN
    ProjL:[att(1,1)]
    CondL: [ ]
        SFW:Ticker_Lookup
        Source:Disclosure
        MapL:[(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,
            TICKER), (Ticker_Lookup, EXCHANGE)]
        ProjL: [att(1, 2)]
        CondL: [att(1,1) = 'MICROSOFT']
        SELECT
        ProjL: [ ]
        CondL: [att(1,1) > 2,500,000]
            CVT
            ProjL:[att(1,1)]
            SpecialCondL:[att(1,1) is att(2,1) * 0.001]
                SFW:DiscAF
                Source: Disclosure
                MapL:[('DiscAF', 'COMPANY_NAME'),
                    ('DiscAF', 'LATEST_ANNUAL_DATA'),
                    ('DiscAF', 'CURRENT_SHARES_OUTSTANDING'),
                    ('DiscAF', 'NET_INCOME'),
                    ('DiscAF', 'NET_SALES'), ('DiscAF', 'TOTAL_ASSETS'),
                    ('DiscAF', 'LOCATION_OF_INCORP')]
                ProjL: [att(1, 4)]
                CondL: [('GERMANY' = att(1, 7))]

Figure 3-4: Plan for Existential Query Example

29

the existential query is non-empty, then we would want the rest of the query to be executed as usual. A join between any non-empty relation $\mathcal{A}$ and $\{\epsilon\}$ should result in $\mathcal{A}$ and not in the empty relation Φ.

We solve this problem by extending the execution engine to handle the existential queries. The engine can identify existential queries from the fact that their projection lists are empty. When the engine discovers an existential query, it executes the query to determine if the answer to the query is empty. The execution engine creates a dummy relation $\mathcal{R}$ to simulate the result of the existential query. If the answer to the existential query is the empty relation, then $\mathcal{R}$ is empty. The result of the mediated query would therefore be Φ as discussed above. However, if the answer to the existential query is non-empty, then the execution engine inserts a dummy tupple into $\mathcal{R}$ to signify the fact that the existential query succeeded. Because no attributes are projected out of the existential query, we can insert any values into the dummy relation without fear of contaminating the result of the query. We can therefore join this dummy relation to other relations in the query. Even though no attribute is projected out of the dummy relation, BANG can join it to other temporary relations because it is non-empty.

Chapter 4

Planner

The Planner takes a union of datalog queries as input and produces a query plan
for the execution engine to execute. It consists of three submodules. The Capability
Checker, The Subgoal Ordering Submodule, and the Plan Construction Submodule.
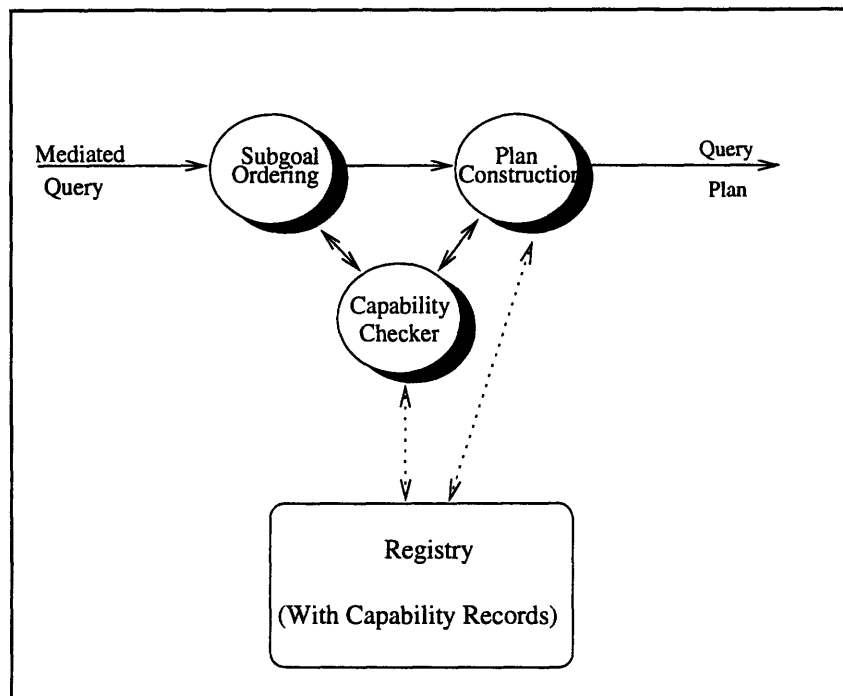
PLANNER

Figure 4-1: The Planner

4.1 The Capability Checker

The *COIN* system mediates queries to a variety of data-sources, including relational
databases and web sources which have been wrapped to provide a relational interface.

31

Due to the nature of some of the data-sources in the *COIN* system the wrappers cannot provide the full relational capability. Therefore, restrictions are placed on the nature of the SQL query which can be dispatched to the wrappers. For example, the relation olsen() in the source **Olsen** has the schema olsen(Expressed,Exchanged,Rate,Date). However, olsen() requires that the attributes Expressed and Exchanged need to be bound whenever a query is sent to it. It also requires that the attribute Rate has to be free. These restrictions cannot be inferred from the export schema. The Capability Checker is a module which takes a description of the capability restrictions imposed by a data-source, and determines if the sub-query destined for the data-source satisfies the restrictions imposed by it. The description of a source's capability restriction is called a capability record.

4.1.1 Capability Records

Our capability records are very similar to those described in [8] in that we create separate records for the binding restrictions, and the operator restrictions of the source. Each record of binding restrictions consists of a list of all the possible binding combinations of the attributes in the source. The combinations are represented as bit fields with an entry for each attributes in the source. The bits have a value of one for attributes which must be bound, and a value of zero for attributes which must be free. The records for operator restrictions are a list of the operators which the source cannot handle.[1]We show two examples of capability records below. The first is for olsen(), which has only one possible combination of binding restrictions, and the second is for dateXform(), which has two.

Example 1
olsen: cap([[1, 1, 0, 1]], ['<',' >',' =',' <>',' =<',' >=']).

Example 2
dateXform: cap([[1, 1, 0, 1], [0, 1, 1, 1]], ['<',' >',' =',' <>',' =<',' >=']).

For olsen(), the first, second and fourth attributes have to be bound and the third has to be free. For dateXform(), the first and second attributes always have to be bound, and either the third attribute has to be bound and the fourth has to be free, or the third has to be free and the fourth has to be bound. Neither of these sources accepts the relational operators listed in the capability record.[2]

4.1.2 Capability Checking

The capability checker takes a component sub-query (CSQ) as its input. A component sub-query is a query destined for a single data-source. It consists of the predicate

---

[1]We only handle the following operators in our system. $=, <, >, <>, >=, =<$. The list of operators which a source cannot handle is taken from this set.

[2]In an SQL query to the data-sources, attributes would be bound to constants in an '=' conditional statement. However, this will be the only situation where both of these sources would accept the '=' operator. Therefore, we cannot really say that the sources accept the '=' operator'

for the data-source, and all the selection conditions which need to be applied to attributes in that source. The capability checker uses the capability record of the source to determine if the CSQ can be handled by the source. Let us look at the situation where Q, a CSQ, is input to the capability checker. The capability checker will return one of the following answers:

1. Q is directly supported by the data-source.

2. Q is indirectly supported by the data-source. Therefore, it must be decomposed into a directly supported query Q' and a filter query f.

3. Q is not supported in the current form and needs extra conditions C in order for it to be supported. Presently, all such conditions are conditions which satisfy binding requirements.

The three CSQs shown below illustrate the three possible responses from the capability checker.

1. olsen(V1, V2, V3, V4),
   V1 = 'USD'

2. olsen(V1, V2, V3, V4),
   V1 = 'USD',
   V2 = 'DEM',
   V3 = 1.1

3. olsen(V1, V2, V3, V4),
   V2 = 'DEM',
   V3 = 1.1

The CSQ shown in the first example is directly supported by olsen(). The two boolean statements provide the required bindings for the attributes olsen.Expressed and olsen.Exchanged. The second CSQ is indirectly supported by olsen(). The condition V3 = 1.1 cannot be handled at the data-source and would have to be executed locally in the execution engine. This is because olsen() requires that the attribute olsen.Rate which is represented by V3 must be free. The selection V3 = 1.1 filters the result of the query obtained by accessing the relation olsen() and is therefore called a filter query. Finally, the third CSQ shows an unsupported query. This query is not supported by olsen() because it violates the binding restrictions which the source imposes on all queries to it. Specifically, the attribute olsen.Expressed is not bound in the query. To access the relation we would have to provide the binding for olsen.Expressed at run-time, using the result obtained by querying another source in the plan. In the event that no source can provide the required binding, the query would fail. The issue of planning the query to ensure that bindings are provided for all queries which require them is discussed in Section 4.2.

4.2 Subgoal Ordering Submodule

This submodule determines if an execution order should be imposed on the sub-queries in the input query. The order is determined by the capabilities of the sources accessed in the query. If the subgoal ordering submodule cannot find an order that satisfies the restrictions imposed by the sources, then the query cannot be answered and an error is generated. This submodule first decomposes the input query into a set of component sub-queries (CSQ). Each CSQ is then sent to the Capability Checker which determines if the source can accept the query it represents.

When the capability checker determines that a CSQ cannot be executed at the data-source because of the violation of a binding restriction, the subgoal ordering submodule attempts to create an executable plan which would obey the restriction. It tries to determine if some subset S of the other CSQs in the query can provide the required binding. If it can make this determination, then it restricts the order of the CSQs. If Q is the CSQ which violates the binding restriction, then the subgoal ordering submodule would ensure that Q cannot be executed before any element of S, by imposing an order on the CSQs. We will use a section of the example from Section 1.2 to clarify this point. Suppose the mediated query consisted only of the predicates secapl(V5, V8, V1, V7), DiscAF(V6,V3, V10, V2,V9,V18,V12) and Ticker_Lookup(V6, V5, V4) as shown in Example 3. We already know that secapl requires that all queries to it should have the attribute secapl.Ticker bound to a constant.

Example 3
answer(V6, V5, V7) :-
  DiscAF(V6,V3, V10, V2, V9,V18,V12), Ticker_Lookup(V6, V5, V4),
  secapl(V5, V8, V1, V7)

In the example, the binding can be provided by Ticker_Lookup because of the implicit join condition on V5. Thus, the Ticker_Lookup predicate would have to precede the secapl predicate. Valid orderings of the body are:

1. DiscAF(V6,V3, V10, V2, V9,V18,V12), Ticker_Lookup(V6, V5, V4),
   secapl(V5, V8, V1, V7)

2. Ticker_Lookup(V6, V5, V4), secapl(V5, V8, V1, V7),
   DiscAF(V6,V3, V10, V2, V9,V18,V12)

3. Ticker_Lookup(V6, V5, V4), DiscAF(V6,V3, V10, V2, V9,V18,V12),
   secapl(V5, V8, V1, V7)

4.2.1 Algorithm for Ordering

The algorithm for ordering the CSQs is shown in Algorithm 4.2.1. When we feed the input of Example 3 to this algorithm, we get the following behavior. The algorithm starts our with U and O in the following state:

```
Input :
    U, The unordered portion of the body (minus conditions). Initially the whole
    body minus conditions L.
    O, The ordered part of the query. Initially [ ].

Result :An ordered combination of U and O that constitutes a valid ordering for
    the query.

Method :

    order(U,O){
    For each predicate P in list U
        If P requires bindings
            get list R of attributes requiring bindings from registry
            If ∃ E ∈ R such that
            1. Either E is not bound in query or
            2. E is not in some predicate in O
                then append P to NewU
            else append P to O
            If NewU is empty
                then return O
            If length(U) == length(NewU)
                then return 'Error'
            else return(order(NewU,O))
    }
```

Algorithm 4.2.1: Algorithm for ordering CSQs

U = [DiscAF(V6,V3, V10, V2, V9,V18,V12),secapl(V5, V8, V1, V7),
    Ticker_Lookup(V6, V5, V4)]
O = [ ];

Initially, DiscAF has no binding restrictions and so is inserted into O. At this point,
the bound variables are [V6,V3, V10, V2, V9,V18,V12]. secapl has a restriction that
V5 be bound. Since this restriction is not satisfied, secapl gets appended to *NewU*.
Ticker_Lookup has no restrictions and so can also be appended to O. This ends the
first iteration of the algorithm.

NewU = [secapl(V5, V8, V1, V7)].
O = [DiscAF(V6,V3, V10, V2, V9,V18,V12),Ticker_Lookup(V6, V5, V4)]

Because *NewU* is neither empty nor equal to U, we must go through a second
iteration of the loop. In this iteration, all the attributes needed to access secapl()
have been bound in O. We can therefore add the secapl predicate to O. At the end

of this iteration *NewU* is empty. The algorithm returns the predicates in the order shown in *O*.

NewU = [ ].
O = [DiscAF(V6,V3, V10, V2, V9,V18,V12),Ticker_Lookup(V6, V5, V4),
        secapl(V5, V8, V1, V7)].

4.3   Plan Construction Submodule

This submodule takes the output of the Subgoal Ordering Submodule, which is a partially ordered list of relations and conversion functions and creates a full algebraic query plan from it. It traverses the ordered list, folding in each predicate into a query plan. Again, I will use the first mediated query from the example in Section 1.2 to explicate the process of constructing the query plan from the output of the ordering submodule.

4.3.1   Example Plan

When the ordering submodule is given a query such as the following:
it orders the body in a way that satisfies all the restrictions imposed by the sources.

> answer(V20, V13, V8):-
>   Ticker_Lookup(V6, V5, V4),
>   Name_Map(V6, V20),
>   DiscAF(V6,V3, V10, V2, V9,V18,V12),
>   '2,500,000' < V13,
>   V13 is V2 * 0.001,
>   secapl(V5, V8, V1, V7),
>   V12 = 'USA'.

One such order could be the one shown below.

> Ticker_Lookup(V6, V5, V4),
> Name_Map(V6, V20),
> V12 = 'USA',
> DiscAF(V6,V3, V10,V2, V9,V18,V12),
> 2,500,000 < V13,
> V13 is V2 * 0.001,
> secapl(V5,V8, V1, V7)]

The plan construction submodule traverses the list provided by the subgoal ordering submodule and produces the query plan. The first entry in the list gives us an **sfw** node.

SFW:Ticker_Lookup
MapL:[(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,
     TICKER), (Ticker_Lookup, EXCHANGE)]
ProjL: [att(1, 1), att(1, 2)]
CondL: [ ]

This **sfw** node tells the execution engine to send a query to the relation Ticker_Lookup
to retrieve the attributes COMP_NAME and TICKER. Even though the original
datalog query only asks for the COMP_NAME attribute, the plan projects out the
TICKER attribute because it is used in a condition. There are no conditions on
the query to Ticker_Lookup. A similar **sfw** node is created for the next predicate
Name_Map(V6, V20) and this is folded into the sub-tree which consists only of the
sfw node for Ticker_Lookup. A **join** node is used for folding in sub-trees.

JOIN
ProjL:[att(2, 2), att(1, 2)]
CondL: [att(1, 1) = att(2, 1)]
       SFW:Name_Map
       Source: Disclosure
       MapL: [(Name_Map, DS_NAMES), (Name_Map, WS_NAMES)]
       ProjL:[att(1, 1), att(1, 2)]
       CondL: [ ]
       SFW: Ticker_Lookup
       MapL: [(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,
            TICKER), (Ticker_Lookup, EXCHANGE)]
       ProjL: [att(1, 1), att(1, 2)]
       CondL: [ ]

The **sfw** node for Name_Map tells the execution engine to retrieve from Name_Map
the attributes DS_NAMES and WS_NAMES. The **join** node then tells the exe-
cution engine to join the two temporary relations retrieved from Name_Map and
Ticker_Lookup on the condition that Ticker_Lookup.COMP_NAMES = Name_Map.DS_NAMES.
It also tells the engine to project (Ticker_Lookup, TICKER) and (Name_Map, WS_NAMES)
out of the result of this join. We will skip an explanation of the processing for the
entry for the DiscAF predicate since we already discussed the construction of an **sfw**
node. The only difference between the node for DiscAF and those of the previous
two predicates is that the condition V12 = 'USA' can be sent as part of the query to
DiscAF. It is therefore put in the condition list of the DiscAF node. After processing
the DiscAF predicate, the conversion function is folded in.
    This converts the NET_INCOME attribute of DiscAF from the DiscAF context
to c_ws by applying the right scale-factor conversion. The result of this conversion

CVT

ProjL:[att(1,1), att(2,1),att(2,2)]

SpecialCondL:[att(1,1) is att(2,3) * 0.001]

    JOIN

    ProjL:[att(2, 1), att(1, 1), att(1,2)]

    CondL: [att(1, 1) = att(2, 2)]

        SFW:DiscAF

        Source: Disclosure

        MapL:[('DiscAF', 'COMPANY_NAME'),

            ('DiscAF', 'LATEST_ANNUAL_DATA'),

            ('DiscAF', 'CURRENT_SHARES_OUTSTANDING'),

            ('DiscAF', 'NET_INCOME'),

            ('DiscAF', 'NET_SALES'), ('DiscAF', 'TOTAL_ASSETS'),

            ('DiscAF', 'LOCATION_OF_INCORP')]

        ProjL: [att(1, 1),att(1, 4)]

        CondL: [('USA' = att(1, 7))]

        JOIN

        ProjL: [att(2, 2), att(1, 2)]

        CondL: [att(1, 1) = att(2, 1)]

            SFW: Name_Map

            Source: Disclosure

            MapL: [(Name_Map, DS_NAMES), (Name_Map, WS_NAMES)]

            ProjL:[att(1, 1), att(1, 2)]

            CondL: [ ]

            SFW: Ticker_Lookup

            MapL: [(Ticker_Lookup, COMP_NAME), (Ticker_Lookup,

                TICKER), (Ticker_Lookup, EXCHANGE)]

            ProjL: [att(1, 1), att(1, 2)]

            CondL: [ ]

is projected out with (Ticker_Lookup, TICKER) and (DiscAF, COMPANY_NAME). (Ticker_Lookup,TICKER) is kept because it is required in a join condition. At this point, the single selection condition in the query can be applied to the correct value of NET_INCOME. Once this is done, the access to olsen can be made and the final result can be returned to the user. The order of these last two nodes does not affect the answer we get for the query. However, we try to plan so that the sizes of intermediate results are reduced as much as possible. Therefore, we push selections as far from the root node of the plan as is possible. The final query plan is as shown in Figure 3-2.

4.4　Optimizations in the Planner

The planner applies some standard optimization heuristics to the query plan. For example, it plans for the executioner to send as many relevant conditions to the remote data-sources as their capabilities would allow. In the event that the relevant conditions cannot be executed at the data-source, the planner ensures that the conditions are applied at the earliest possible point in the execution. Furthermore, it ensures that only attributes that have been requested by the user as part of the answer to the query, or which are required as inputs to conversion functions and boolean operators are projected out of data-sources and intermediate relations. These optimization efforts are based on the assumptions which we made about the time-cost of sending queries to the data-sources. The next chapter presents a discussion of these assumptions, and presents the details of a performance analysis study which was used for a preliminary investigation of the validity of the assumptions.

Chapter 5

Performance Analysis

5.1 Motivations for Performance Analysis

Our attempts at optimizing the query execution and our ideas for future optimizations in the system have been based on some assumptions we made about the distribution of the time involved in executing queries. The query response time is composed of the retrieval time and the insertion time. The retrieval time is the difference between the time the execution engine opens a connection for a query request to the data-source, and the time when that connection is closed. The insertion time is the difference between the time when the connection to the data-source is closed, and the time when all the tuples in the result have been inserted in the local store. The assumptions we made are:

1. Retrieval time consists of the communication time between the executioner and the data-source, and the execution time of the query at the data-source.

2. The communication time depends on the size of the response to the query.

3. The execution time on the data-source depends on the number of relations selected from.

4. Communication time commands a larger fraction of the retrieval time than the remote execution time does.

5. Insertion time depends on the size of the response to the query.

Based on these assumptions, our optimization efforts have been aimed at reducing communication between the *COIN* backend and the data-sources, and at reducing the size of the results returned from the data-sources. For example, we ensure that any condition which reduces the size of the result of a query is pushed to the data-source. Furthermore, we cache intermediate results so that we do not access the data-sources for multiple results. Some of our future optimization plans are also aimed at dispatching as much of a query to a data-source as we can in a single connection. This would reduce the amount of communication needed between the *COIN* system

and the data-source. In order to include this optimization in the POE, we will need to know whether the reduction in communication time will be offset or overwhelmed by the increase in execution time on the data-source. Moreover, our assumptions about time costs in the system need to be verified. To this end, the performance analysis study described in this chapter was designed. This study intends to lay the framework for future investigations into the allocation of time resources in the POE. The techniques which we used for performing this study were provided by [6].

5.2 Project Plan

This section presents the project plan which was created for the performance analysis study.

System Definition The goal of this study was to determine estimates for the effect of various factors on the retrieval time for queries in the *COIN* system. The response variables for the study are the insertion time, and the response time. Our system under test (SUT) consists of the subset of the *COIN* execution engine which deals with communication and insertion of results into the local data-store, the channels connecting the remote sources to the *COIN* engine, and the database engines on the remote data-sources. The study was designed so that the effects of all other components are minimized.

Services Provided By the System The service provided by the system is information retrieval access to a data-source, and insertion of query results into the local data-store. We assumed that the system is reliable and we do not consider the cases of unreliable data, or unavailable services. Our proposal states that the resources used during the system operation depend on the expected cardinality of the query, and the number of relations accessed in the data-source. The application for this study is "response to queries" and will be classified using the notions of query response time and result size.

Metrics For each service we wanted to know the how much time it took for the service to be rendered. This gave us the following metrics for our study.

1. Remote query response time
2. Insertion time

Parameters The system parameters that affect the performance of a query are:

1. Cardinality of the result.
2. Speed of the network
3. Amount of network traffic
4. Load on the data-source
5. Actual processing power available on the data-source

41

6. The distance of the site from the POE.

Factors The factors of this study are the parameters which were varied in the experiment. These were the expected cardinality of queries to the sources, and the number of relations accessed by the query.

1. Expected Cardinality: We calculate this factor using information from the query itself, and catalog information on the data-sources. A listing of the catalog for the relations used in this study is give in Appendix A. Four levels were chosen for this factor; 0, 2000, 4000, 6000.

2. Number of relations accessed: We only used two levels for this factor, 1 and 2.

Evaluation Techniques Our evaluation technique was a direct measurement of the response variables in the experiment. We chose to use measurement evaluation because we already had a working implementation of the whole system.

Workload The workload was a set of stored queries of different expected cardinalities and which accessed different numbers of relations. These queries were executed by a program which monitored the resources consumed and logged all the results. The unit of measurement for the time resources is milliseconds. To determine the amount of resources used by the monitoring and logging functions, we also monitored and logged null query requests which did no actual work, but served to provide us with a measure of the overhead involved with the measurements and logs.

Experimental Design We used a two-factor full factorial design for the study of the retrieval time. The two factors were the expected cardinality of the query, and the number of relations accessed in the remote data-source. Since we had two levels for the number of relations accessed and four levels for the expected cardinality, the total number of experiments run was eight. The design for the insertion time study was a one-factor design. Here, the factor of interest was the expected cardinality of the query. We replicated the experiment four times at each level of the factor to get a total of sixteen experiments. Our models for both studies were linear regression models. A linear regression model attempts to fit a straight line to the experimental observations so that the residual between the experimental results and the model line are minimized. We used the least-squares criterion for minimizing the residual errors. This criterion states that the best linear model is given by the parameter values which minimize the sum of the squares of the errors. Since the sum of the squares cannot be a negative number, the minimum value for it is zero. This model assumes that the effects of the factors of the experiment add and that the errors are additive. We analyzed the models to estimate the values of the model parameters. These estimates were then used to determine the effect of each factor on the variability in the response variables of the study. Descriptions of the analyses for the studies are given in the next two sections.

5.3 Data Analysis for Retrieval Time Study

We run the experiment in the manner proposed in the plan, and analyzed the data using the model shown below. The model is a multiple linear regression model. In the ensuing discussion, Factor A is the number of data-source relations accessed in the query and Factor B is the expected cardinality of the query result.

5.3.1 Model

We used the following model for our experiment.

$$y_{ij} = \mu + \alpha_j + \beta_i + e_{ij}$$

y_{ij} is the measurement of the retrieval time in the experiment in which Factor A is at a level j and Factor B is at a level i. μ is the mean response. α_j is the effect of Factor A at level j, β_i is the effect of Factor B at level i and e_{ij} is the error term for the experiment. a is the number of levels of Factor A and b is the number of levels for Factor B. For this experiment, a is two and b is four.

5.3.2 Computation of Effects

We compute the values of μ, α_j and β_i such that the following conditions hold:

$$\sum \alpha_j = 0$$

$$\sum \beta_i = 0$$

$$\frac{1}{ab} \sum e_{ij} = 0$$

The first two conditions do not affect the model but only serve to decrease the complexity of analyzing the experimental measurements. The third condition comes from the fact that we use the least-squares criterion to find a line of best fit for the response variable. For the experimental analysis, we insert all the measured responses in a two-dimensional array with four rows and two columns. The columns correspond to the levels of Factor A and the rows correspond to the levels of Factor B. Averaging along the jth column produces

$$\overline{y}_{.j} = \mu + \alpha_j + \frac{1}{b} \sum_i \beta_i + \frac{1}{b} \sum_i e_{ij}$$

where $\overline{y}_{.j}$ is the column mean for column j. The last two terms of this equation are zero, and so we get the following relationship between $\overline{y}_{.j}$, μ and α_j.

$$\overline{y}_{.j} = \mu + \alpha_j$$

A similar analysis along the ith row produces

$$\bar{y}_{i.} = \mu + \beta_i$$

where $\bar{y}_{i.}$ is the row mean for row i. Averaging all the responses should gives us $\bar{y}_{..}$, the grand mean of all our responses.

$$\bar{y}_{..} = \mu$$

We can calculate the parameters for our model using the following formulas.

$$\mu = \bar{y}_{..}$$

$$\alpha_j = \bar{y}_{.j} - \bar{y}_{..}$$

$$\beta_i = \bar{y}_{i.} - \bar{y}_{..}$$

The measured retrieval time in milliseconds, is shown in table 5.3.2. For each row and column, we computed the mean of the measurements. We also computed the grand mean. The difference between the row or column mean and the grand mean is the row or column effect. For example, the effect of the expected cardinality on the retrieval time when the expected cardinality is 2000 is 309.63 milliseconds.

Expected Cardinality	OneRel	TwoRel	RowMean	RowEffect
0	2269	3081	2675	-1645.38
2000	3880	5380	4630	309.63
4000	3881	4950	4415.5	95.13
6000	3821	7301	5561	1240.63
ColumnMean	3462.75	5178	4320.38	
ColumnEffect	-857.63	857.63		

Table 5.1: Computation of Effects for Retrieval Time

5.3.3 Data Interpretation

We can use the results from the previous section to find out what proportions of the retrieval time (our response variable) are due to Factor A, Factor B or to errors. If we use the sum of the squares of each parameter to estimate its contribution to the variation of the response, then we can derive equations for the variation due to each parameter, and from that find out the contribution of each parameter to the variation of the response variable. The total variation in the measurements of the retrieval time is SST and is given by the following equation.

$$SST = SSA + SSB + SSE$$

44

SSA is the sum of squares for Factor A, SSB is the sum of squares for Factor B and SSE is the sum of squared errors. If we can compute SSA, SSB and SSE, then we can find SST. The contribution of each parameter to the variation in the response variable can then be determined by finding its percentage of SST.

Experimental Errors

The sum of squared errors SSE can be computed directly from the model. According to our model the retrieval time can be predicted using the following equation.

$$\hat{y}_{ij} = \mu + \alpha_j + \beta_i$$

where $\hat{y}_{ij}$ is the predicted retrieval time. The difference between the predicted retrieval time and the actual measurement can be attributed to the experimental error. For every i and j the error e_{ij} is

$$e_{ij} = y_{ij} - \hat{y}_{ij} = y_{ij} - \mu - \alpha_j - \beta_i$$

We can then find the SSE from the following formula:

$$SSE = \sum_{i=1}^{b} \sum_{j=1}^{a} e_{ij}^2$$

The SSE for this experiment was 2197087.375 milliseconds^2.

Allocation of Variation

Squaring both sides of the model equation and adding across all the observations gives us the following equation for the sum of squares:

$$\sum_{ij} y_{ij}^2 \quad = \quad ab\mu^2 \quad + \quad b\sum_j \alpha_j^2 \quad + \quad b\sum_i \beta_i^2 \quad + \quad \sum_{ij} e_{ij}^2$$

$$SSY \quad = \quad SSO \quad + \quad SSA \quad + \quad SSB \quad + \quad SSE$$

SSY is the variation in the response variable and SSO is the variation in the means. We can therefore calculate the SSA and SSB directly from the model parameters and a and b. For this experiment the SSA was 4413123.84 milliseconds^2 and SSB was 8702651.37 milliseconds^2. Total variation SST, was therefore 15312862.59 milliseconds^2. The percentage of variation due to Factor A was

$$100 \times \frac{SSA}{SST} = 28.82\%$$

and that due Factor B was

$$100 \times \frac{SSB}{SST} = 56.83\%$$

45

According to this experiment, 56.83% of the retrieval time is due to the expected cardinality of the query, while 28.82% is due to the number of relations accessed by the query. 14.35% of the variation cannot be explained and is due to other factors, or to experimental errors.

5.4 Data Analysis for Insertion Time Study

A similar analysis on a slightly different model yields results for the effect of the expected cardinality of the query on the insertion time of query results. In the ensuing discussion, Factor A is the expected cardinality of the query result. As mentioned in previous discussions, we conducted the study using four levels for the expected cardinality.

5.4.1 Model

$$y_{ij} = \mu + \alpha_j + e_{ij} \tag{5.1}$$

In the ith iteration of the experiment in which Factor A is at level j, y_{ij} is the measurement for the insertion time, μ is the mean response, and α_j is the effect of level j and e_{ij} is the error term. Again, the effects and the error terms are computed so that they add up to zero.

5.4.2 Computation of Effects

We conducted sixteen experiments in this study. This consisted of four replications for the experiment at each of the four levels of the expected cardinality. From the results obtained, the values of μ and α_j were computed in the following manner. Substituting the observations into equation 5.1 and adding up all the equations gives us

$$\sum_{i=1}^{r} \sum_{j=1}^{a} y_{ij} = ar\mu + r \sum_{j=1}^{r} \alpha_j + \sum_{i=1}^{r} \sum_{j=1}^{a} e_{ij}$$

Since the effects of α_j and e_{ij} should add up to zero, we get

$$\mu = \frac{1}{ar} \sum_{i=1}^{r} \sum_{j=1}^{a} y_{ij}$$

where μ is the grand mean of all the measurements.
The mean $\overline{y}_{.j}$ for each column is

$$\overline{y}_{.j} = \frac{1}{r} \sum_{i=1}^{r} y_{ij}$$

Substituting $\mu + \alpha_j + e_{ij}$ for y_{ij}, we get

$$\overline{y}_{.j} = \frac{1}{r}\sum_{i=1}^{r}\mu + \alpha_j + e_{ij}$$

$$= \frac{1}{r}\left(r\mu + r\alpha_j + \sum_{i=1}^{r}e_{ij}\right)$$

$$= \mu + \alpha_j$$

We can therefore calculate the parameter α_j using the following formulas:

$$\alpha_j = \overline{y}_{.j} - \mu$$

The results of this analysis are shown in table 5.4.2. Each column corresponds to four replications of the experiment at one of the four levels of the expected cardinality factor. We calculate the mean for each column, and also calculate the grand mean for all the observations. The difference between a column mean and the grand mean is the column effect.

Table 5.2: Computation of Effects for Insertion Time

Iteration	0	2000	4000	6000	
1	0	20	30	101	
2	10	20	31	109	
3	0	21	59	90	
4	0	20	41	111	
ColumnMean	2.5	20.25	40.25	102.75	41.44
ColumnEffect	-38.9375	-21.1875	-1.1875	61.3125	

According to this analysis, the average insertion takes 41.44 milliseconds.

5.4.3 Data Interpretation

We use a similar analysis to the one in the retrieval time study to determine the effect of the expected cardinality on the insertion time of queries. Again, the sum of the squares of each parameter is used to estimate its contribution to the variation of the response. The total variation in the measurements of the retrieval SST is given by

$$SST = SSA + SSE$$

where SSA is the sum of squares for the expected cardinality and SSE is the sum of squared errors. Once SSA and SSE have been computed, the contribution of each parameter to the variation in the response variable can be determined.

Experimental Errors

Our model makes the following prediction for the insertion time of queries.

$$\hat{y}_j = \mu + \alpha_j$$

The difference between the predicted insertion time $\hat{y}_j$, and the actual measurement is the error and can be calculated using the equation

$$e_{ij} = y_{ij} - \hat{y}_{ij} = y_{ij} - \mu - \alpha_j$$

The SSE can then be computed from the following formula:

$$SSE = \sum_{i=1}^{r} \sum_{j=1}^{a} e_{ij}^2$$

The SSE for this study was 891.25 milliseconds^2.

Allocation of Variation

Squaring both sides of the model equation and adding across all the observations gives us the following equation:

$$\sum_{ij} y_{ij}^2 = ar\mu^2 + r\sum_{j}^{A} \alpha_j^2 + \sum_{ij} e_{ij}^2$$

$$SSY = SSO + SSA + SSE$$

SSA for this experiment was 22902.69 milliseconds^2. Therefore, total variation SST was 23793.9375 milliseconds^2. The percentage of variation due to the expected cardinality is therefore

$$100 \times \frac{SSA}{SST} = 96.25\%$$

5.5   Conclusions and Future Work

The results of these two studies look quite promising, and if we were confident of their validity, we could proceed with some of our planned optimizations. For the retrieval time study, we determined that 56.83% of the retrieval time is due to the expected cardinality of the query, while 28.82% is due to the number of relations accessed by the query. Increasing the number of relations accessed in a query by one only increases the retrieval time by 28.82%, while decreasing the expected cardinality by 2000 decreases the retrieval time by 56.83%. We can therefore try to access as many relations on a data-source as we can in one query, and also to push as many conditions to the data-source as possible. According to this study, the benefits of reducing the expected cardinality of query results outweighs the extra processing needed for accessing relations. Furthermore, reducing the expected cardinality of the query reduces the insertion time for the query by quite a significant amount. All

of these results could also be incorporated into a cost model for doing cost-based optimization in the *COIN* system.

Despite these promising results, more investigations must be done before the results can be confidently utilized in the *COIN* system. First, a more exact distribution for the data in the data-sources should be developed. The assumption we used in this study was that the data is uniformly distributed between the max value and the min value of each attribute [12]. Observations of the data-sources have shown that that this assumption does not hold. Secondly, a larger study with a bigger sample size needs to be done in order for us to be able to establish valid confidence intervals for our results.

Chapter 6

Conclusions and Future Work

The POE described in this thesis has been successfully deployed in a variety of implementations of the *COIN* system. It has proved to be both versatile and effective in providing answers to users in a variety of application domains. Two such applications have come from the areas of finance and defense logistics. There are a number of improvements which can be made in the POE to improve its execution. These improvements are mainly in the areas of capability checking, and optimization of the query execution.

6.0.1 Capability Improvements

The capability records in the current POE describe source operator restrictions at the level of the whole source. However, there are situations in which data-sources will restrict the operators on a per attribute basis. For example, olsen() places a binding restriction on two of its four attributes. These are the attributes olsen.Expressed and olsen.Exchanged. These bindings must be sent to olsen() as an '=' condition in an SQL query. However, this is the only situation where olsen() will accept the '=' operator in a query. Thus, olsen() accepts the operator '=', but only on the condition that it binds olsen.Expressed and olsen.Exchanged to constants. Our current capability records cannot express this conditional capability restriction.

6.0.2 Optimization Improvements

There is a lot of room for further improvements in the execution of mediated queries. For example, the planner could aggregate all the queries to relations in the same data-source, so that the source would need to be accessed only once. Secondly, the execution engine could store some of the intermediate query results in main memory rather than in the local data-store. This would improve the query time by reducing access to disk. There are also opportunities for parallelizing the execution of queries. At the moment this can be done for any node which has multiple child data-structures. Examples of these nodes are the **join** node and the **join-sfw** node. However, because the current planner only produces left-deep trees, these opportunities are currently limited. A concerted effort to extend the space of the QEP to bushy trees, and

to parallelize the executioner would probably introduce dramatic improvements in the execution time of queries. Finally, some interesting optimization issues exist in the area of semantic query optimization. However before any semantic query optimization can be done in the POE, there would have to be re-specification of the functions of some of the processes in the *COIN* system. Because the context mediator performs some semantic query optimization as part of the mediation process, a clear cut demarcation will have to be made between optimizations which should occur in the mediation engine itself, and optimizations which should occur in the POE.

Most of these optimizations will involve a coordinated effort among many different components of the *COIN* system. For example, in order to plan for a query which accesses all the relations in a data-store, the registry will have to be redesigned to remove the current notion that relations and data-sources have a one-to-one mapping relationship.

6.0.3 Conclusion

In general, the POE has achieved its aim of providing a stable execution environment for context mediated queries. We demonstrated the abilities of the system at the "ACM SIGMOD/PODS Conference on Management of Data" in May 1997, and presented a paper on the implementation of the system at the "'Fifth International Conference and Exhibition on the Practical Applications of Prolog." in April 1997. It was well received at both conferences.

Appendix A

Selectivity Formulas and Relation Catalogs

Selectivity Formulas.[1]

Condition	Formula	Constant Expression
$Column1 = Const$	$F = \frac{1}{DistinctVals}$	$\frac{1}{10}$
$Column1 = Column2$	$F = \frac{1}{max(NumDiffVal1, NumDiffVal2)}$	$\frac{1}{10}$
$Column1 > Const$	$F = \frac{(MaxVal - Const)}{(MaxVal - MinVal)}$	$\frac{1}{3}$
$Column1 < Const$	$F = \frac{(Const - MinVal)}{(MaxVal - MinVal)}$	$\frac{1}{3}$

The Constant Expression is used in a situation where the formula is not applicable. Weak bounds have the same selectivity values as the above tight bounds. Also, we assume independence of column values that is why we do not have conditional cardinalities.

Using the selectivity of a query, the expected cardinality of a query is computed as follows: If P_s is the product of all the selectivities of the conditions, and P_c is the product of all the cardinalities of the relations accessed in the query, then

$$ExpectedCardinality = P_s \times P_c$$

---

[1]These formulas assume uniform distribution of attributes values between minval and maxval. See [12] for a more complete discussion.

Table A.1: Catalog for Performance Analysis Relations

Relation Name	Number of Tuples
WorldAF	10633
DiscAF	1005
DstreamAF	312

Attribute Name	Rel Name	Type	MaxVal	MinVal	Distinct
COMPANY_NAME	WorldAF	string	-	-	10624
LATEST_ANNUAL _FINANCIAL_DATE	WorldAF	string	-	-	346
CURRENT _OUTSTANDING _SHARES	WorldAF	number	1.0333E+11	0	9951
NET_INCOME	WorldAF	number	5280000	-8101000	9788
SALES	WorldAF	number	171962154	-48878	10520
TOTAL_ASSETS	WorldAF	number	530401911	0	10570
COMPANY_NAME	DiscAF	string	-	-	505
LATEST _ANNUAL_DATA	DiscAF	string	-	-	86
CURRENT_SHARES _OUTSTANDING	DiscAF	number	6211439761	0	482
NET_INCOME	DiscAF	number	6606000000	-878000000	987
NET_SALES	DiscAF	number	3.5798E+12	31594	1004
TOTAL_ASSETS	DiscAF	number	4.0398E+12	0	998
LOCATION _OF_INCORP	DiscAF	string	-	-	37
AS_OF_DATE	DstreamAF	string	-	-	5
NAME	DstreamAF	string	-	-	71
TOTAL_SALES	DstreamAF	number	2.0610E+10	0	310
TOTAL_EXTRAORD _ITEMS_PRE_TAX	DstreamAF	number	-1	-1	1
EARNED _FOR_ORDINARY	DstreamAF	number	431449856	-166054000	307
CURRENCY	DstreamAF	string	-	-	4

Appendix B

Description of Contexts and Relations

Table B.1: Context Descriptions

Context	Currency	Scale Factor
c_ws	US Dollars	1000
c_ds	Currency of Country of Incorp.	1

Table B.2: Relation Schemas and Contexts

Relation	Context	Schema
Currencytypes	-	['COUNTRY',string], ['CURRENCY',string]
Currency_map	-	['CHAR3_CURRENCY',string], ['CHAR2_CURRENCY',string]
dateXform	-	['Date1', string], ['Format1', string], ['Date2', string], ['Format2', string]
olsen	c_ws	['Exchanged',string], ['Expressed',string], ['Rate',real], ['Date',string]
Ticker_Lookup	c_ds	['COMP_NAME', string], ['TICKER', string], ['EXCHANGE' string]
DiscAF	c_ds	['COMPANY_NAME',string], ['LATEST_ANNUAL_DATA',string], ['CURRENT_SHARES_OUTSTANDING',integer], ['NET_INCOME',integer], ['NET_SALES',integer], ['TOTAL_ASSETS',integer], ['LOCATION_OF_INCORP',string]
WorldAF	c_ws	['COMPANY_NAME',string], ['LATEST_ANNUAL_FINANCIAL_DATE',string], ['CURRENT_OUTSTANDING_SHARES',integer], ['NET_INCOME',integer], ['SALES',integer], ['TOTAL_ASSETS',integer], ['COUNTRY_OF_INCORP', string]

Bibliography

[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 Hetergeneous Information Systems*. PhD dissertation, Massachusetts Institute of Technology, Sloan School of Management, December 1996.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Leon Sterling and Ehud Shapiro. *The Art of Prolog : Advanced Programming Techniques*. The MIT Press, 1994.