(4)

A Lightweight Multi-Database Execution Engine

by

Ricardo S. Ambrose

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

at the

Massachusetts Institute of Technology

June 1998

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
May 20, 1998

Certified by [illegible]
Prof. Stuart Madnick
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Lightweight Multi-Database Execution Engine

Development of an Database Execution engine to deal with structural context differences between data sources

by

Ricardo S. Ambrose

Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1998 in partial fulfillment of the
requirements for the Degree of Master of Science in
Computer Science

Abstract

In this thesis, I describe the design and implementation of a lightweight multi database engine for data intensive Web applications. The engine was designed to provide structured querying (relational database access) to data from online data sources ranging from databases to semi-structured documents. The implementation focused on allowing users to write queries, which would interface with distributed data, sources on the World Wide Web. One of the major problems dealt with was the automatic extraction and manipulation of data from distributed sources, taking into consideration the structural differences and data stored in semi-structured manner. The engine was implemented using an concurrent execution model that allowed much higher network parallelism compared to earlier versions of the system.

Acknowledgements

My final year at MIT has by far been the toughest, most enjoyable and edifying year. Writing this thesis allowed me to focus my skills on this project and realize that there are so many who support me.

First of all I would like to thank the members of COIN research group. Professor Stuart Madnick and Dr. Michael Siegel gave me the opportunity to join this group and were a real support throughout. Dr. Stéphane Bressan guided me from beginning to end, and was there to help with any problem that I had. I also thank the other, current and past members, of the Team such as Ahmed Ahzar, Tom Lee and Ozlem Ouzuner for being great a support.

My family, although many thousands of miles away, would always magnify the good and downplay the bad. They prepared me to face these challenges and for that I give them all my thanks.

My MIT friends were always there to help relax my mind, even when everyone was loaded with work. I thank them for that, and also for helping me to focus when it was so easy to stray.

To my fiancée, Winnette McIntosh, through every day and night she has been there for me. Supporting me, encouraging and caring for me have been her tasks throughout, and she has been very successful at them all. I thank her for being here for me throughout this project and choosing to be with me forever.

Finally and most importantly I thank God for everything, the opportunities that I have been given, the ability to make good on those opportunities, the grace God has given me to live a life with reference to him, the people he has put into my life and of course the strength to complete this project.

Table of Contents

List of Figures

Chapter 1

Introduction

The amount of data becoming available over the past few years has grown immensely and will continue to do such in the future. One of the major factors behind this trend is the advance in networking and telecommunication services, which allows the transfer of data between points to be very fast. These advancements have incited a general use of the World Wide Web to transfer information.

The data is readily available in many different forms from a variety of sources. Although the data may exist there are often many difficulties when attempting to utilize the data stored in the heterogeneous sources. How does one collate large amounts of data from a couple different sources or small amounts of data from many sources? To answer this question possible data sources must be identified. Web pages, databases (relational or not) and flat text files can all be used as a store for data[1]. In many of these cases the data is in a semi-structured format, hence it is not immediately clear how to automate the extraction of such data. The manipulation[2] of the data when extracted from the sources is equally important to the actual data extraction requirement in such a process.

Unfortunately, the ability to exchange meaningful information has actually become more difficult as the number of sources continue to steadily increase. Providing logical connectivity between disparate sources is the base of the problems that we seek to address in this thesis. There is an inherent bias in the implementation and design to accommodate for World Wide Web access. This was because the hard problem lies within the attempt to extract the data from the semi-structured sources as stored on the web combined with maintaining the logical connectivity between the sources accessed. There is also the problem of time to access the data. When dealing with large amounts of data from the WEB the issue of network delays becomes much more of an issue.

In this thesis I seek to identify the major design decisions taken for the implementation of a query execution engine. This takes a ordered list of sources to access with the and executes the sub-queries to those sources and combines he result into a table to be presented to the user. Most of

[1] This list is not complete, as there are types of sources such as music files, news groups etc.
[2] Querying the data to produce a table with the required result data.

the inovation in design and implementation went into he execution engine which was built to be more robust, easy to migrate and which uses an execution model that takes advantage of the network communication delays that are inherent in a web intensive application. In order to demonstrate the effectiveness of the execution engine it was necessary to build a planner/optimizer that is responsible for decomposing the user's SQL query into sub-queries and for planning the order of execution for those queries.

1.1 Organization of Thesis

The thesis is divided into seven chapters and a set of three appendices. The second chapter gives a detailed description of the motivation behind this thesis, specifically with a stock portfolio application that gets it's information from various web sites. The third chapter gives an overview of the different pieces in the system architecture for the execution engine and planner/optimizer. Chapter four goes more into the details of the execution engine implementation, the reasons for the implementation decisions and a discussion of the execution model being used. The fifth chapter summarizes the planning algorithm that was implemented in the system. Details of all the implementations decisions made to build the various components will be described in the sixth chapter. The conclusion of this thesis and a number of ideas of how this system can be improved and used to build other applications can be found in the final chapter. The appendices hold various technical annotations; such as a simple installation guide, example specification files, code statistics (size, use) and some results from a demo of the system, which can be referenced to fully describe a few of the factors used in the actual implementation.

Chapter 2

Motivational Scenario

As an example consider a businessman who regularly updates his stock portfolio. Current and historical information on the movement of the stock in the portfolio can be visualized in a spreadsheet. To keep this up to date, the businessman must collect the relevant data for the stock on a daily basis and then update the spreadsheet. The New York Stock Exchange (NYSE) provides a very useful web site with pages for each of the companies trading on the exchange. These pages hold information such as the last stock price and the company ticker. Zacks web site provides, for most listed companies, the recommendation[3] (whether to buy, sell or hold) for the companies' stock. Yahoo stock pages show the price, percentage and point changes, the volume sold for the trading period and news headlines for each company. An example of the information provided by all three sites can be seen below (fig 2.1 and 2.2) for Oracle Corporation.

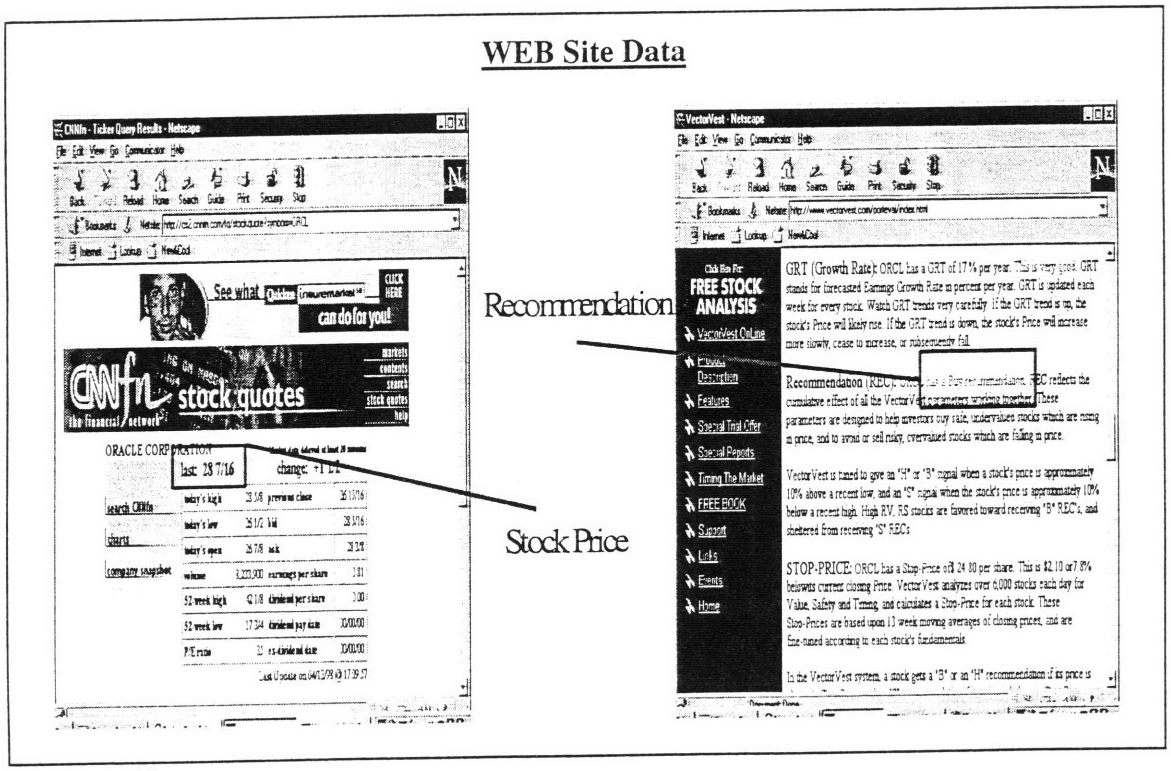
Although the information is very easily accessible, the only way to get to this information is to manually browse & click to load up the required pages. For a single company this may not be very difficult but when dealing with a large portfolio, the manual gathering of the data becomes a bit more tedious. An even graver problem occurs if the search set is not predefined to certain companies. Below, two web-intensive data-extraction situations will be described and the procedures, which can be used to automate this extraction, will be highlighted.

In the first example a businessman would like to get a list of companies, their tickers, last selling price and any related headlines for which the stock price is less than $50 and with a recommendation of 'buy'. This information is provided by the NYSE and Zacks websites as is highlighted in the diagram below. He would have to go to each of the individual NYSE Company web pages, if the stock price is less than $50 then add the company to a list. Using this list and the Zacks web page determine which companies, on the list, meet the second requirement. Assuming that it takes 30 seconds for a single page to load and the user record the required information, given that there are over 1000 companies on the stock exchange, and two pages must be accessed for each company, the entire process can take as long as a day. Hence although the information can be very useful to a person deciding what stocks to buy, gathering this information carries with it a very high cost. The problem exists not only in the business world but in other application

[3] A figure which is used by traders to determine which stocks are good to buy.

domains. For instance travelers wanting information on flights, car rentals, weather etc. Compiling this information from many different web sites may take a considerable amount of time. In any situation where large amounts of data is being gathered from different sources or many sources are being used to gather data we can find the time consumption problem embedded within the process if done manually.

The second example is very similar to the first as it deals with stock information web sites, but in this case the businessman has an investment portfolio with Tickers of a few companies along with the last selling price, recommendation and any news headlines all stored in a text file formatted in a manner which can be imported into a spreadsheet application. Manually updating this information requires going to a couple of web sites and getting the pages specific to the companies in the portfolio file. Although this is much more manageable than the example above, it still can be automated as it is a monotonous everyday activity.



ORACLE CORPORATION

last: 28 7/16   change: +1 1/2

today's high	28 5/8	previous close	26 15/16
today's low	26 1/2	bid	28 5/16
today's open	26 7/8	ask	28 3/8
volume	3,233,900	earnings per share	0.81
52-week high	42 1/8	dividend per share	0.00
52-week low	17 3/4	dividend pay date	00/00/00
P/E ratio	35	ex-dividend date	00/00/00

Last Update on 04/13/98 @ 17:39:57

http://www.vectorvest.com/porteval/index.html

GRT (Growth Rate): ORCL has a GRT of 17% per year. This is very good. GRT stands for forecasted Earnings Growth Rate in percent per year. GRT is updated each week for every stock. Watch GRT trends very carefully. If the GRT trend is up, the stock's Price will likely rise. If the GRT trend is down, the stock's Price will increase more slowly, cease to increase, or subsequently fall.

Recommendation (REC): ORCL has a Buy recommendation. REC reflects the cumulative effect of all the VectorVest parameters working together. These parameters are designed to help investors buy safe, undervalued stocks which are rising in price, and to avoid or sell risky, overvalued stocks which are falling in price.

VectorVest is tuned to give an "H" or "B" signal when a stock's price is approximately 10% above a recent low, and an "S" signal when the stock's price is approximately 10% below a recent high. High RV, RS stocks are favored toward receiving "B" REC's, and sheltered from receiving "S" RECs.

STOP-PRICE: ORCL has a Stop-Price of $24.80 per share. This is $2.10 or 7.8% below its current closing Price. VectorVest analyzes over 6,000 stocks each day for Value, Safety and Timing, and calculates a Stop-Price for each stock. These Stop-Prices are based upon 13 week moving averages of closing prices, and are fine-tuned according to each stock's fundamentals.

In the VectorVest system, a stock gets a "B" or an "H" recommendation if its price is

Figure 2.1

Solving the problems above requires a intermediary agent which will allow the user to formulate a structured request, and then the agent will automatically extract the data from the required sources manipulate the data to produce the desired results and then present this result to the user.

The automatic nature of such a system will definitely be limited by the varying structures of stored data e.g. if one requires data from both a web site and an Excel spreadsheet to produce the result of the request. The problems are even clearer when dealing with two different web sites, where data may be stored in completely different formats.

Even though the sites can easily be seen as being tables (providing certain specific attribute) automating the extraction still remains a problem due to the varying structures of the sites. Wrapping the sources can hide these structural differences. By this, I mean using an object that knows how to access a source and extract the information and can then provide information (e.g. what attributes are provided) on that particular source. It's clear that the main information needed here is the list of attributes provided along with the method of extraction, in this system this information is stored in a source specification file. Consider the *yahoo* financial web site. Below there is a extract from a specification file used to wrap this site. Highlighted are the means of accessing the site (Method and URL) and also how the information must be extracted from the site (Pattern); the make up of the specification file is explained in more detail in chapter 5.

```
<PAGE>
      <METHOD> GET </METHOD>
      <URL> http://quote.yahoo.com/q?s=##Ticker##&d=v1&o=t </URL>
      <PATTERN>
<a
href="/q\?s=.*?&d=t">.*?</a>\s+##LastTrade:(.*?)##\s+<b>##Change:(.*?)#
#</b>\s+##Changepts:(.*?)##\s{2,}##Changepct:(.*?)##\s+##Volume:(.*?)##
<small>
      </PATTERN>
 </PAGE>
```

Once the specification files are created the above example queries can be represented in SQL, assuming the existence of the relations *"nyse"*, *"finance"* and *"zacks"*,as shown below.

QUERY_1

```
    SELECT nyse.compname, nyse.stockprice, zacks.recommend
    FROM nyse, zacks
    WHERE nyse.ticker = zacks.ticker AND nyse.stockprice > 50 AND
            zacks.recommend = 3
```

When executed this query gets all the company Tickers and associated company names from the *nyse* web site. Using these Tickers, the system will access the *zacks* web site to extract the recommendation for each company. This information is appended to the tuples with the company name and Ticker. The system also uses the Tickers to access the *finance* web site. The stock price is extracted from this source and appended to the appropriate tuple. The result tuple is then projected out to the user.

QUERY_2

```
SELECT nyse.compname, yahoo.price, yahoo.change, finance.headline
FROM nyse, yahoo, finance
WHERE ticker in ["IBM","ORCL","T","TNT","A"]
```
[4]

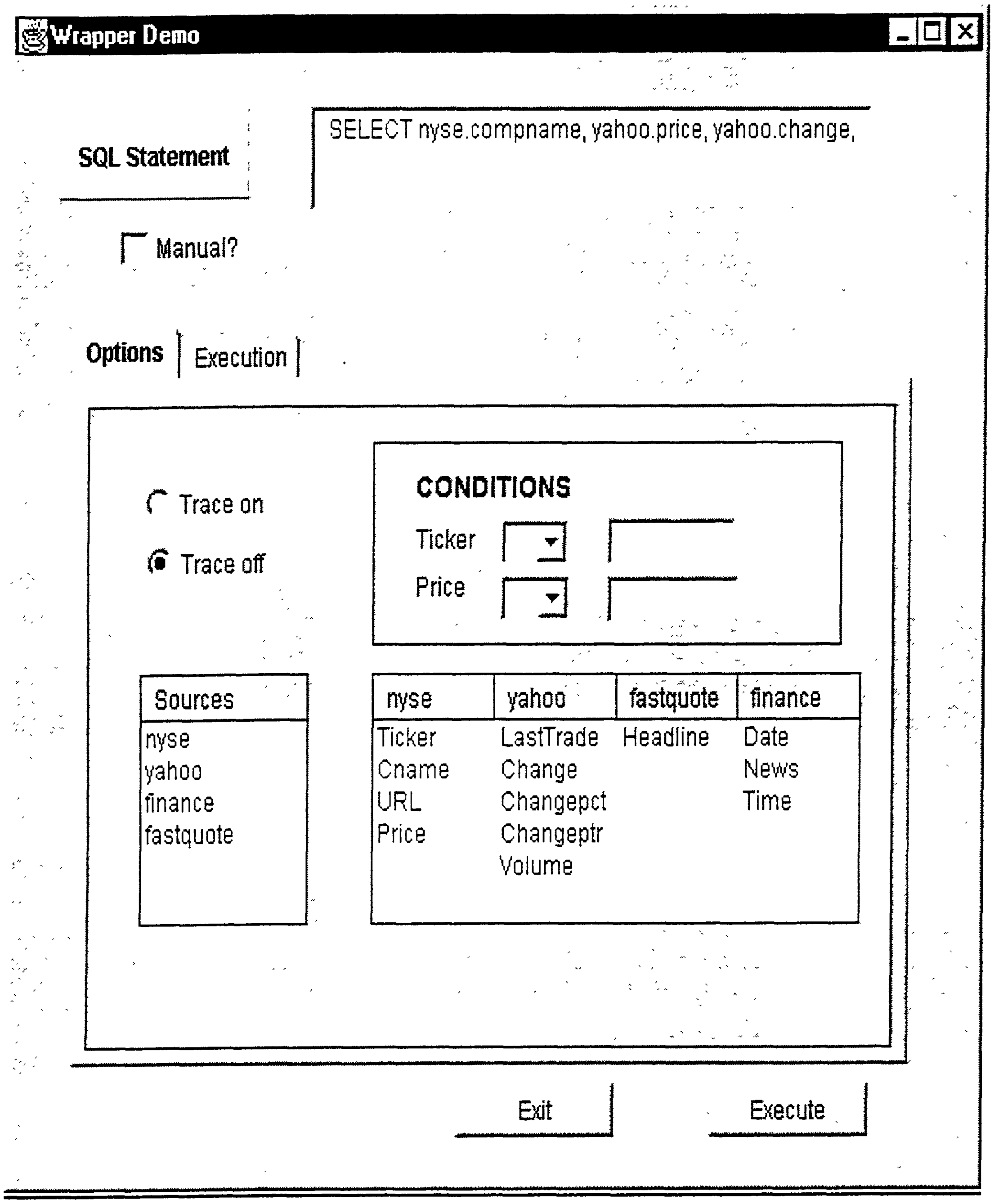
In this query, the Tickers are bound to constant values. Hence when executed, the sources will only be accessed for the tuples within the provided list of Tickers. Each source is accessed, as above, to get the attribute that it provides. These attributes are all appended onto the result tuples and then output.

This form of the query provides a common access to the available sources (relations). This requires an access layer be put into place, which isolates the user from interacting directly with the heterogeneous data sources. This layer will use the SQL query to determine which sources should be used, what is required from each of the sources and how they should be accessed.

A sample application was built to show the functionality of the system. The interface (see below) is a basic GUI, which allows the user to either type in an SQL (as above) directly, or to select those attributes available from a set of sources. Once the query is entered the user can press the '*Execute*' key which will launch the query compilation and execution. The results are then saved in a user designated text file (also shown below) or sent to the GUI output.

[4] This can be extracted from a file given the name but for clarity within the example the list is shown.

The DEMO Interface



Wrapper Demo

SQL Statement: SELECT nyse.compname, yahoo.price, yahoo.change,

[ ] Manual?

Options | Execution

( ) Trace on

(●) Trace off

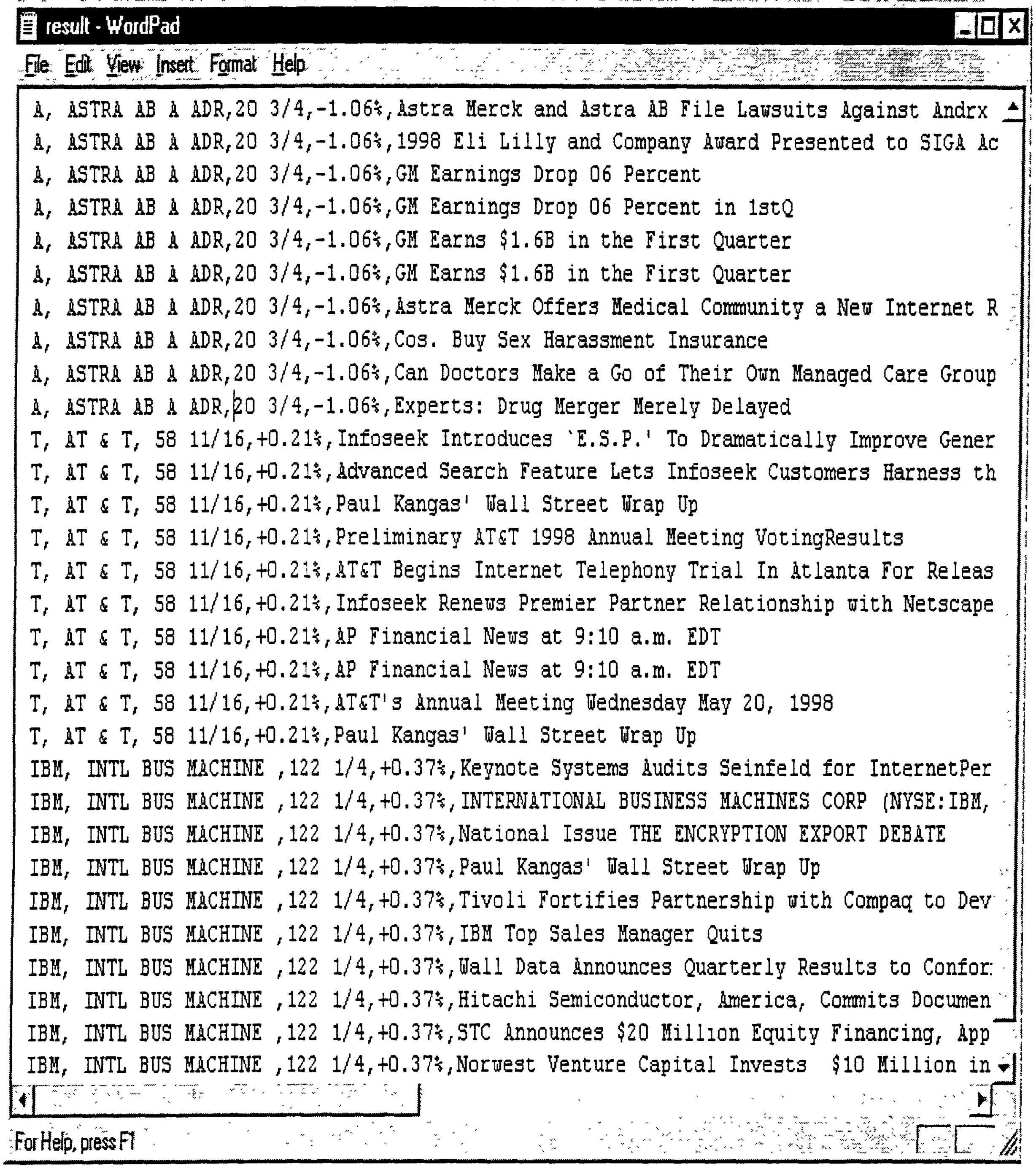
CONDITIONS

Ticker

Price

Sources
nyse
yahoo
finance
fastquote

nyse	yahoo	fastquote	finance
Ticker	LastTrade	Headline	Date
Cname	Change		News
URL	Changepct		Time
Price	Changeptr		
	Volume		



```
A, ASTRA AB A ADR,20 3/4,-1.06%,Astra Merck and Astra AB File Lawsuits Against Andrx
A, ASTRA AB A ADR,20 3/4,-1.06%,1998 Eli Lilly and Company Award Presented to SIGA Ac
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earnings Drop 06 Percent
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earnings Drop 06 Percent in 1stQ
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earns $1.6B in the First Quarter
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earns $1.6B in the First Quarter
A, ASTRA AB A ADR,20 3/4,-1.06%,Astra Merck Offers Medical Community a New Internet R
A, ASTRA AB A ADR,20 3/4,-1.06%,Cos. Buy Sex Harassment Insurance
A, ASTRA AB A ADR,20 3/4,-1.06%,Can Doctors Make a Go of Their Own Managed Care Group
A, ASTRA AB A ADR,20 3/4,-1.06%,Experts: Drug Merger Merely Delayed
T, AT & T, 58 11/16,+0.21%,Infoseek Introduces `E.S.P.' To Dramatically Improve Gener
T, AT & T, 58 11/16,+0.21%,Advanced Search Feature Lets Infoseek Customers Harness th
T, AT & T, 58 11/16,+0.21%,Paul Kangas' Wall Street Wrap Up
T, AT & T, 58 11/16,+0.21%,Preliminary AT&T 1998 Annual Meeting VotingResults
T, AT & T, 58 11/16,+0.21%,AT&T Begins Internet Telephony Trial In Atlanta For Releas
T, AT & T, 58 11/16,+0.21%,Infoseek Renews Premier Partner Relationship with Netscape
T, AT & T, 58 11/16,+0.21%,AP Financial News at 9:10 a.m. EDT
T, AT & T, 58 11/16,+0.21%,AP Financial News at 9:10 a.m. EDT
T, AT & T, 58 11/16,+0.21%,AT&T's Annual Meeting Wednesday May 20, 1998
T, AT & T, 58 11/16,+0.21%,Paul Kangas' Wall Street Wrap Up
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Keynote Systems Audits Seinfeld for InternetPer
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,INTERNATIONAL BUSINESS MACHINES CORP (NYSE:IBM,
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,National Issue THE ENCRYPTION EXPORT DEBATE
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Paul Kangas' Wall Street Wrap Up
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Tivoli Fortifies Partnership with Compaq to Dev
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,IBM Top Sales Manager Quits
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Wall Data Announces Quarterly Results to Confor
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Hitachi Semiconductor, America, Commits Documen
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,STC Announces $20 Million Equity Financing, App
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Norwest Venture Capital Invests  $10 Million in
```

Chapter 3

Design

3.1 System Architecture

The System is essentially divided into two main components. These are the Compiler / Planner / Optimizer and the Execution Engine (fig 3.1). The other identified pieces act as channels through which data is put into the system. The focus of the thesis will be the execution engine as this is where most of the effective innovation has gone. The planner / compiler was built to facilitate the construction of a useful system. The engine is designed to be modular as possible in order to be

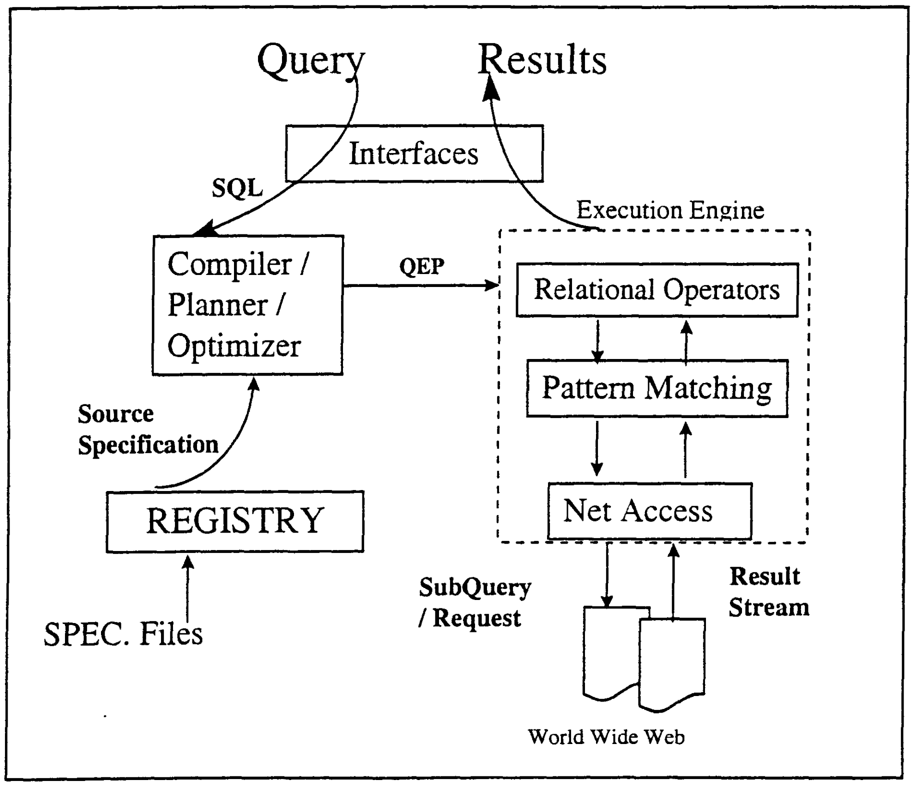


Figure 3.1

easily migrated from this version to future implementations. It can also be operated as a Java library for querying various data sources which can be used by other applications.

The Query planner / optimizer compiles the user request[5] into a query execution plan (QEP, see fig 4.2 in section on Execution Model for more details). The plan is a tree of algebraic operators and data access methods. Specification of the sources in the registry indicates what attributes each

[5] The request can be made using an SQL query.

source can supply. The user query indicates the required attributes. Combining the information from the source specifications and the user query the planner generates the QEP.

The Execution engine is the system component that carries out the instructions held in the query execution plan. It is based on a set of physical operators implementing the relational algebra. The data access methods are also contained in this component. These access methods correspond to the various types of data sources available, e.g. for semi-structured web sources the access method consists of a network access client and a regular-expression based pattern matching component. A sequential version of the engine had been implemented to validate the general system architecture. But the concurrent model was be used to implement a final pipelined version of the engine.

Using the information from the QEP, the execution engine executes the plan and returns the result to the user. To do this the engine has been divided into three functional modules. These are the 'Net Access' module, 'Pattern Matching' module and the 'Relational Operators' module. All access to data sources is requested through the 'Data Access' module, which, in the case of web sources and flat files, will return a stream in which the result of the request will be accessible. Using this stream, the 'Pattern Matching' module will then extract the attributes stored within the stream. These attributes are collated as tuples, which are the base processing unit within the system.

Chapter 4

The Query Execution Engine

The research associated with this thesis is focused on the creation of a framework that will allow the effective and efficient execution of a query, which accesses multiple non-local data sources. The engine will be designed to support read-only queries. The query execution engine is geared toward dealing with the last step required for the processing of a query inputted to the system as shown in the figure 4.1 below. These steps outline what is done from the input of a query, in some pre-determined language, through the compilation and optimization of the query to the final production of results (execution),

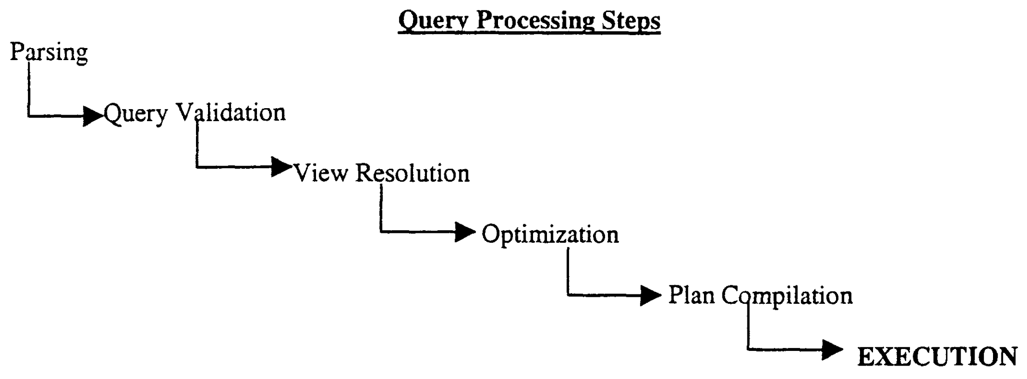


Figure 4.1

4.1 The Global Picture

The multi-database execution engine executes all the instructions required to produce a tabular result. This is the system component which takes the Query Execution Plan generated by the planner/optimizer, extracts the data from the required sources, combines this data in a format described in the plan and returns the result as a table to the user. The web intensive nature of the applications, for which the system will be used, would normally have a very high dependency on the network operation. The time required to access web sites varies according to net congestion, server speed and other factors that are not very easy to control. These delays may cause unnecessary bottlenecks in an application that requires data from the web to produce a result. For instance in our portfolio example getting information from the *NYSE* web site may take a lot longer during working hours than in the night. Given that this information is required early on in the first query, it will cause the processing to be slow. Even in normal operation, a request takes a long time to respond compared to the local processing that needs to be done on that data. Hence given the fact that servers may be able to handle multiple requests from the same source and also that, in a single query, different sources may be required sending multiple simultaneous request (network parallelism) will cut down the average wait required for a response. In an attempt to take advantage of the time that is wasted in getting data from the web, the engine was built using a concurrent model (see below). The reasoning behind this decision was to increase the effective network parallelism. In this section the design of this module will be outlined.

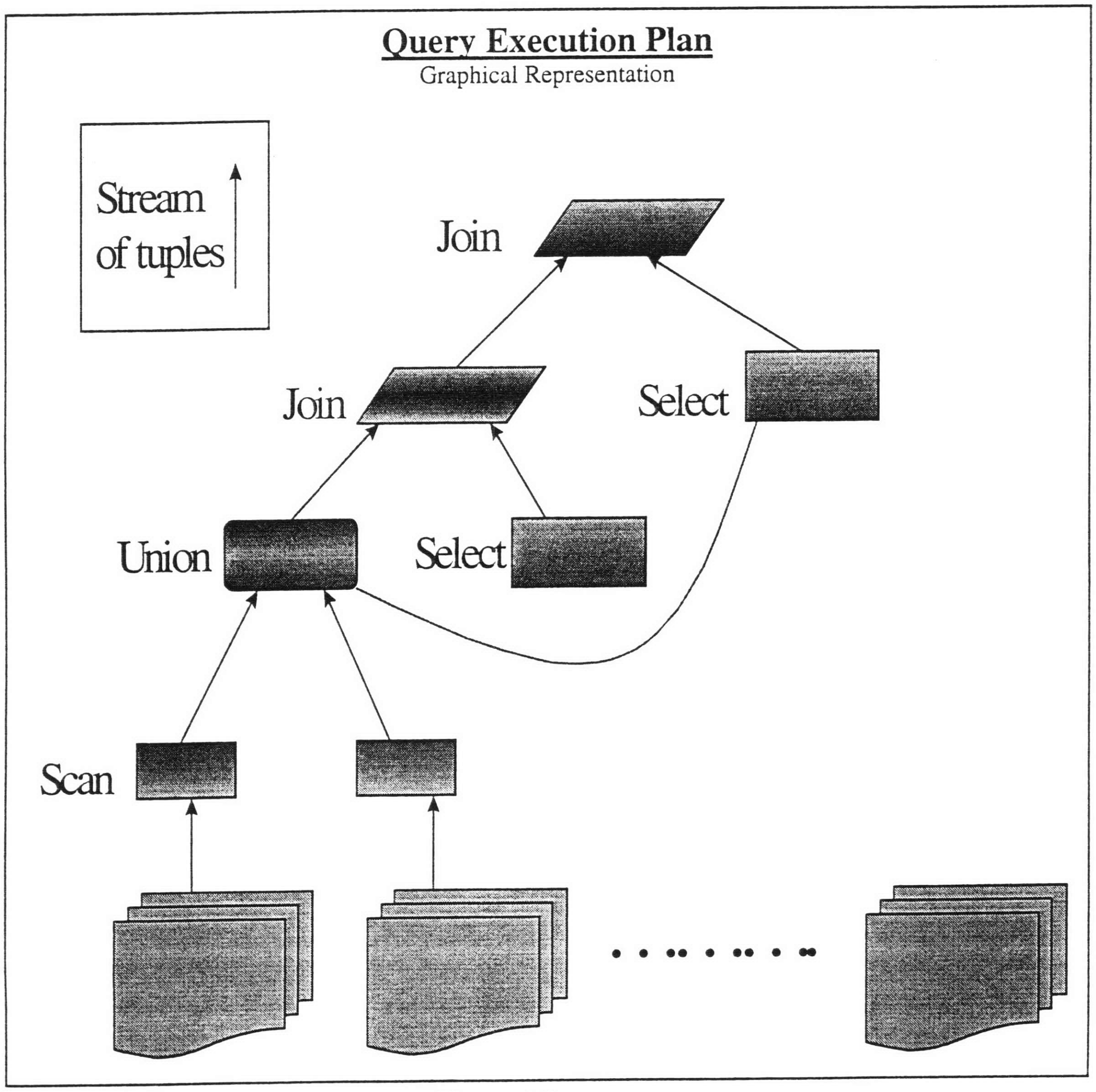


Figure 4.2

4.2 Query Execution Plan

The engine must have a set of instructions to operate. The Query Execution Plan (QEP) is a data structure that identifies the operations that need to be executed in order to get the results of a query. Plans are made up of a tree of relational operators with their required parameters. The specific parameters would differ depending on which operator is being used. For instance the join requires a condition and two data sources, whereas a project requires only a single data source and no condition. The plan is compiled into a directed graph where a single node in the graph represents each relational operator. The QEP structure, figure 4.2 shown above, has examples of the relational operators that may be used. The connections (edges) within the graph represent data streams, which allow data to flow in one direction. A single data table is produced by each relational operator, all edges out of a node represent the same data table produced by the node. Hence the shared data table resource will be used in single producer, multiple consumer framework, as is the case with the union operator (below) which has two consumers. This functionality becomes useful when a QEP requires that a particular set of data be accessed more than once, for example a particular implementation of the OR construct (see 5.3). Because of the concurrency built into the system, this forces strict synchronization rules to be put in place when accessing the data table Planner/Optimizer module.

4.2.1 Logical Operators

The logical operators maps out the data model that the system uses, they are basically the set of standard operators on collections of structured data. In the system operators such as *selection*, *projection*, *union* and *join* are considered. The ***select*** operator takes a table of tuples and chooses only those tuples that meet the required condition. The ***project*** operator constructs a new tuple for each tuple in the table and puts only the attributes that have been identified in a projection list. The ***join*** operator creates a new tuple for each pair of tuples from the two input tables that meet specified criteria, the new tuple is constructed by concatenating the two input tuples. The ***union*** operator merges two or more input tables. Finally the ***scan*** operator reads in the data from the external source. As a strictly relational model is used the operators do realize duplicate elimination. Operators or operations combined with the above mentioned operators, such as the *nest* and *unnest* operators for nested data structures or path expressions are not considered in this document. A minimum set of operations are available for the basic types of the model allowing for instance the evaluation of arithmetic expressions and the evaluation of boolean comparisons and boolean expressions to verify the conditions of a selection or a join. The type of application examples that are being targeted require the use of aggregation functions, the implementation of these operators will not be elaborated on in the thesis. The implementation of the system does provide for the extension of the base operators.

4.2.2 Physical Operators

A QEP is a directed graph of physical operators. The execution engine evaluates the operators in the QEP in some order determined by the execution model that is used. Each physical operator has zero, one or two sub-trees ordered from left to right. The algebra of logical operators is implemented by means of a set of physical operators. The correspondence between logical and physical operators is many to many. For example a join followed by a selection in the algebraic representation of a query is likely to be implemented by single join-project physical operator. Furthermore, the join-project operator can be one of nested loop join or balanced join[6]. The design of the physical algebra is guided by the necessity to provide the planner with a sufficient set of operations for the evaluation of a query and by the attempt to provide the optimizer with a sufficient range of options to construct an efficient plan.

The main operators implemented are the select, join and union all of which are combined with the projection operator. Duplicate elimination is explicitly performed by a separate operator and it's use is controlled by system preferences which can be set by the user. join-submit, submit and regex are the last operators which are implemented in the system.

<u>The submit operator</u>

The class constructor for this access operator is of the following form:

Submit (Sourceaddress, Sourcetype)

Sourceaddress: A string representation of the absolute address from where the required data has to be extracted.

Sourcetype: The type of data source being accessed.

This is the first of two access operators. Both access operators send requests for data, located at *sourceaddress,* to the system scheduler (see section 4.2.5). The response from the scheduler is a handle to a data stream, which holds the result of the request – from this point on, all data within the system needs to be transferred as sets of tuples passing through the *buffered data streams*

[6] This join is based upon the loop-join but doesn't give outer or inner loop preference. It is used mainly within the concurrent operation of the program. See details in implementation algorithms.

between operators (see section 4.2.3). The data stream is put into a new tuple, as the only attribute, and passed along to the *buffered data stream.*

<u>The join-submit operator</u>

JoinSubmit (Sourceaddress, Sourcetype, Subtree)

Sourceaddress: A parametrized address that has to be completed by one or more attribute values.

Sourcetype: This is same as in the submit operator.

Subtree: This is the data structure for the sub-tree, which gives the set of tuples which are used in this operator.

This operator is used when the address of a data source depends on the value of a particular attribute, e.g. the stock information for ***IBM*** can be accessed from the following web address:

http://qs.cnnfn.com/cgi-bin/stockquote?symbols=IBM

The address includes the value of the Ticker attribute ***IBM*** and is only a partial address without this attribute. The attributes needed to construct the complete address are accessed from another relation that is gotten from the *Subtree.* For each tuple in the *Subtree* relation a new address will be constructed and a request sent to the scheduler. Once the resulting stream returns it will be appended onto the appropriate tuple as a new attribute and the tuple is put on the buffered data stream.

<u>The regex operator</u>

Regexfn (RegularExp, AttrList, Subtree)

RegularExp: A string parameter that holds the regular expression used in the pattern matching process to extract data.

AttrList: The *AttrList* gives a list of all the attributes that the system extracts from the input stream and the types of these attributes. The attributes are represented by the index into the regular expression.

Once a stream result from a source is passed on from one of the access operators, the *regex* operator will take this stream and extract the attributes stored in the stream. The input stream is taken from the last attribute of the incoming result from the *Subtree* execution. Using the pattern matching techniques with the *RegularExp,* this operator produces a list of attributes, which are all appended to the tuple from which the attributes used to complete the address for the request were taken.

The duplicate operator

Duplicate (KeyList, Subtree)

KeyList: This is a list of the key attributes in the relation. The attributes are represented as indexes into their position in the relation.

The duplicate operator is actually used to remove all the duplicate entries from the results, which are produced by the *Subtree*. It will initially compare only those attributes from the KeyList and if there is a match then remove the second tuple being compared. The tuples are removed by simply not propagating up the tree. Although there is still constant propagation up the tree, this operator must have an internal buffer of all the tuples that it has processed, so that comparisons can be made with new tuples.

The select operator

Select (ProjectionList, Conditions, Subtree)

ProjectionList: This is the list of attributes that have to be projected from the relation. These attributes will be involved in the join conditions of the query and those in the final projection list of the query.

Conditions: A set of boolean operations applied to the attributes of the relation. Each attribute is replaced by an index into the relation.

The select node is used to apply conditions to intermediate results. Only the set of tuples that pass the condition will be propagated up the tree.

The join operator

Join (Projectionlist, Conditions, Subtree1, Subtree2)

ProjectionList: This is very similar to that of the select, but the attributes are indices from both the incoming relations.

Conditions: A set of join-conditions on the relations. If no conditions are given then the result of the join will be a full cross product of the two relations.

For each pair of tuples in the two relations, gotten from *Subtree1* and *Subtree2,* the opertaor will check to determine whether the join condition will hold. If it does hold, then concatenating the two tuples forms a new tuple, which is then put onto the result stream for the join operator.

The union operator

Union (Subtree, Subtree)

This is used to get the union (concatenation) of the results obtained by executing the two Subtrees.

4.2.3 The Buffered Data Streams between the Operators

The branches between two nodes represent a stream of tuples, which is buffered to compensate for the difference in speeds of incoming and consumed tuples. The basic model for this buffer stream is a producer consumer monitor. In the system it is implemented as a ***datatable*** structure which gets the tuple from the producer and delivers it to the consumer. Normal processing semantics for the execution engine can either be seen as a *'get_next'* from the operators or a *'put_next'* from the perspective of the datatables. In the traditional definition a stream can only be consumed once, this poses problems if it has multiple consumers that all need all the tuples or is used by a join with no memory. Hence the ***datatable*** has a mechanism to keep track of where each consumer has reached and then only destroy the stream when all the consumers have used the full set of tuples.

4.2.4 Execution

Once the plan has been compiled into the graph of operators, each node (thread) is started. Hence the role of executing the different operators is then left mostly up to the scheduler for Java threads. When an operator is active it will get the next tuple(s) from its data source(s), if no tuple is available then the operator will wait to be notified by the data table when one does come available. Once the tuple is available the operator will perform whatever operation it is supposed to carry out then, if there is a resulting tuple it will be added to the outgoing data table attached to the operator.

A Concurrent design allows the data to be streamed to the root of the graph, as each node can process the data whenever that data is ready. There is no need to wait for the bottleneck problem of collating all the results before proceeding to the next level of execution. Each relational operator is a thread that can be executed whenever the scheduler runs it and there is data available from its data source.

The data access routines have been structured so that full advantage can be taken of the data streaming capabilities provided within the concurrent design. Access to flat files and web sites hide the slow i/o time in the processing time. The stream of data is ready for use as soon as the first line of data is read from a file or web page. To facilitate the continued flow without the introduction of pipes, the data access routines for files and the web simply return the raw data, with no parsing done. Pattern matching extraction from the strings is one aspect of the functionality of the scan operator. Where the attributes are extracted from the string and put together to form tuples and then data tables. There is one special operator built to deal with compiling a table from multiple web pages. For instance the NYSE has a page with trading details for each of the companies on its list. This *join-submit* operator will access the pages for each company, extract the required information and add it to its result table.

Relational databases restrict duplicates within tables. This is a very significant requirement, as the system is being developed in a relational model and hence has the overhead of checking for duplicates. Projection of tables is the only relational operator that may cause duplicates. I have chosen to unify the project and join into a single join and similarly with the project and select into a single select. Hence duplicate elimination can be used after each of those relational operators. This choice did not significantly affect the implementation, rather simply required one less traversal of the result table per select or join operator, as in the normal case after one of those operators the project would traverse the result table to get the required attributes.

4.2.5 The System Scheduler

Limited machine resources require that a monitor be used to ensure that the number of active data access thread does not threaten to use all the available resources. This necessitated the development of a scheduler that controlled the total number *access* threads that could be opened simultaneously. Access operators can only make request for data through the scheduler. Once this request is made the scheduler will assign a new *accessthread* to the operator's request. The thread will only be started if the number of active requests is below some pre-determined number. In addition to being a system that controls the use of machine resources, this functionality can actually be used as a tuning device for the system. Given that the trade-off is between network parallelism and thread processing overhead, changing the maximum number of active threads can affect the both these factors and then observations can be used to determine what sort of dependency the total execution time has on the two factor, and hence find the best setting. The *accessthread* will initialize the data access for the request and deliver the resulting stream to the operator that originally submitted the request to the scheduler. The diagram below (fig 4.3) shows the scheduler's operation with the addition of a cache. All the data that is read is saved on the local disk and the stream handle returned to the access operator is actually an output stream from the local file. Whenever data is requested, the *accessthread* will check the cache registry to determine whether that address has been accessed before and hence the data could be found in the cache.

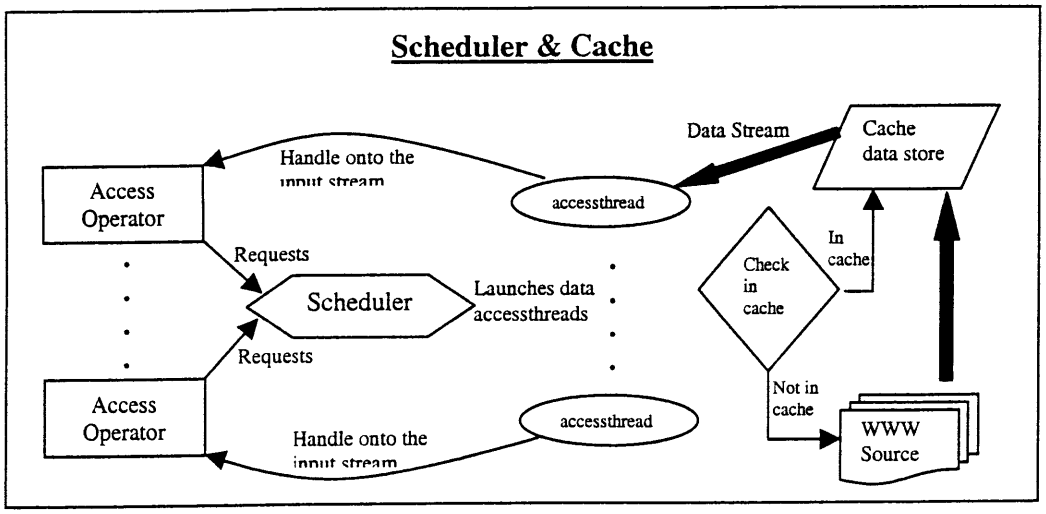


Figure 4.3

4.3 Concurrent Execution

Even though the system was operated on a single-processor machine the model used to implement the concurrency was geared toward leveraging the ability for network parallelism and hence reducing the average network access time. Although the average access time may be reduced there is a definite overhead in the scheduling of various threads. Hence the sequential version will require less local processing. However the concurrent model is fairer in that it produces early results as soon as possible and is able, in this task, to avoid bottlenecks created by data sources with low data rates (e.g. "slow" Web sites). As most of the time is spent on the network retrieving data, the higher network parallelism (as per requests) should result in an improved performance (although this is bound by the network bandwidth and the servers' ability to handle multiple requests. No special effort has been made to explicitly control the concurrency. The task is left to the Java thread scheduler. However, several elements in the plan, such as the choice of an operator or the size of the buffered data stream will influence (in a distributed manner as opposed to a centralized scheduling) the concurrent behavior. For instance, the systematic choice of buffers of size 1 synchronizes the waves of single data production.

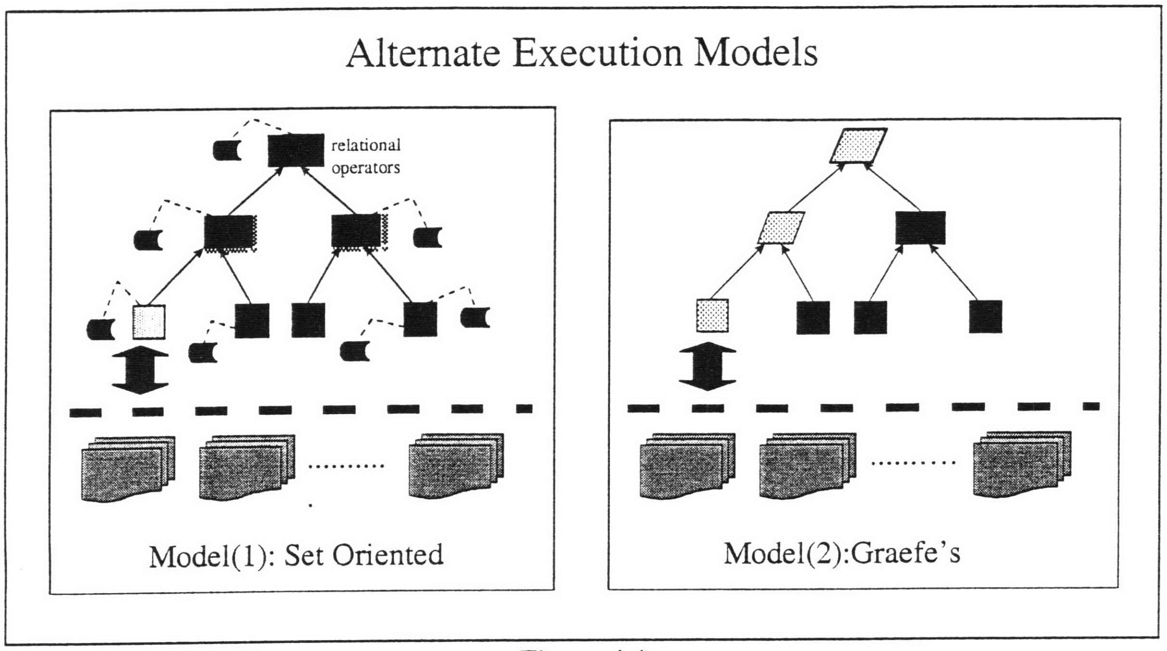


Figure 4.4

4.3.1 Execution model

The execution model being used is the iterator model described by Graefe in [Gra93]. This model is characterized by a pipelined production of results. In this section, three possible execution models[7] that could all be used in this system will be outlined the chosen model will be justified. For reference to the above diagrams, the light shaded blocks represent active operators.

Model (1)

In the sequential version, each operator is equipped with three methods: ***open(), close(),*** and ***get_next()***, to initialize, finalize and request the production of data. The sequential version is *set oriented* as it performs all the entire process (initializing, requests and finalizing) as a block operation.

```
open()
   get_next()
         :
   get_next()
close()
```

[7] The first two models were used in previous versions of the implementation.
Model(1): grenouille ver1.0, ver1.1
Model(2): Grenouille ver2.0

At each stage of processing each operator produces the entire resulting set of tuples. This is stored and then presented to the parent operator for processing. The graphical representation of the model shows only the access node being active and that the only requests generated from this node will be sent to the network. This model requires the least processing overhead of the three.

Model (2)
Inter-operator parallelism is basically pipelining, or parallel execution of different operators in a single query. In this second model there is synchronous pipelining of data up a branch in a tree, hence the name vertical inter-operator. Hence there are more operators working concurrently, but it is always limited to a single line from root to leaf (there are now splits), this is shown by the three stripped nodes in the model (2) diagram above. To achieve the concurrency in this model the high level operations (get_next(), open(), close()) from the sequential model are interleaved for the different relational operators. Using this model definitely increases the possibility of parallel network request but at the cost of higher processing overhead.

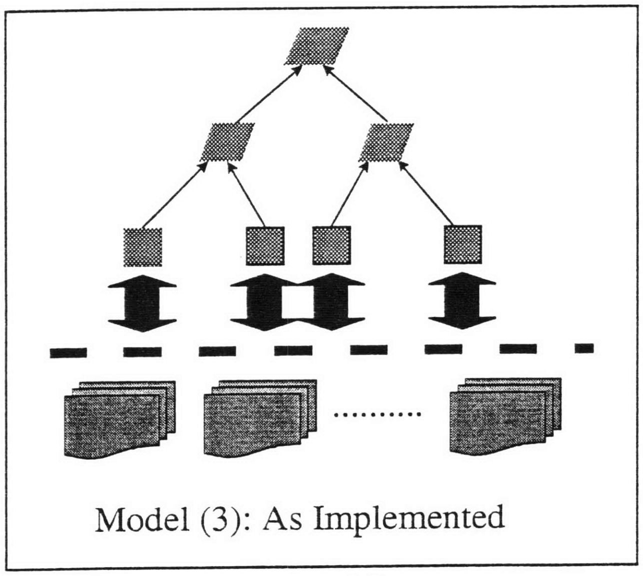


Figure 4.5

Model (3)
The final model is the one that is implemented in the system. The concurrency goes both vertically and horizontally. Each operator is thread that will start to execute once the engine begins executing the QEP. This allows interleaving of lower level instructions, hence increased parallelism. As shown in the diagram, the fact that all the operators can execute concurrently maximizes the number of active access (leaf) nodes, where the higher network parallelism is

manifested. The main difference between the three models is the number of operators that can be active at any one time, and hence the level of network parallelism facilitated.

4.3.2 Non-Blocking Join

Elimination of possible bottlenecks is one of the major concerns within the system. The chosen model does well to hide the effect of slow network responses, by sending a stream of results up the tree. Any bottleneck within the system will nullify the effect of this average network access speedup. The traditional *nested-join* algorithms all favor one of the input data steams. If the favored data stream stems from a slow web site this will cause a delay. To alleviate this problem we used a concurrent balanced-join, which opportunistically uses the next available tuple produced by either input stream.

Consider the cross product of sets of tuples, this can be represented as a two dimensional array, as in the diagram below, with one set (n tuples) on one axes and the other set (m tuples) on the other axes (m and n not necessarily known). Once a new tuple comes, it can be identified by it's position in the 2-dimensional array, given information on the position of the last entry. The next step would be to join this tuple with all the tuples from the other set, which the operator has already received and is ready to process. Hence at any point in time, with k1 tuples from one set and k2 tuples from the other, the operator would have completed the join operation on that subset of tuples. From the diagram the shaded portions show the grouping which is made for a particular sequence of tuple arrivals. The main idea would be to take the incremental subsets created after a new tuple is added, produce an entry (as described below) for them which would describe the bounds of the subset, and then do the join on all the subsets.

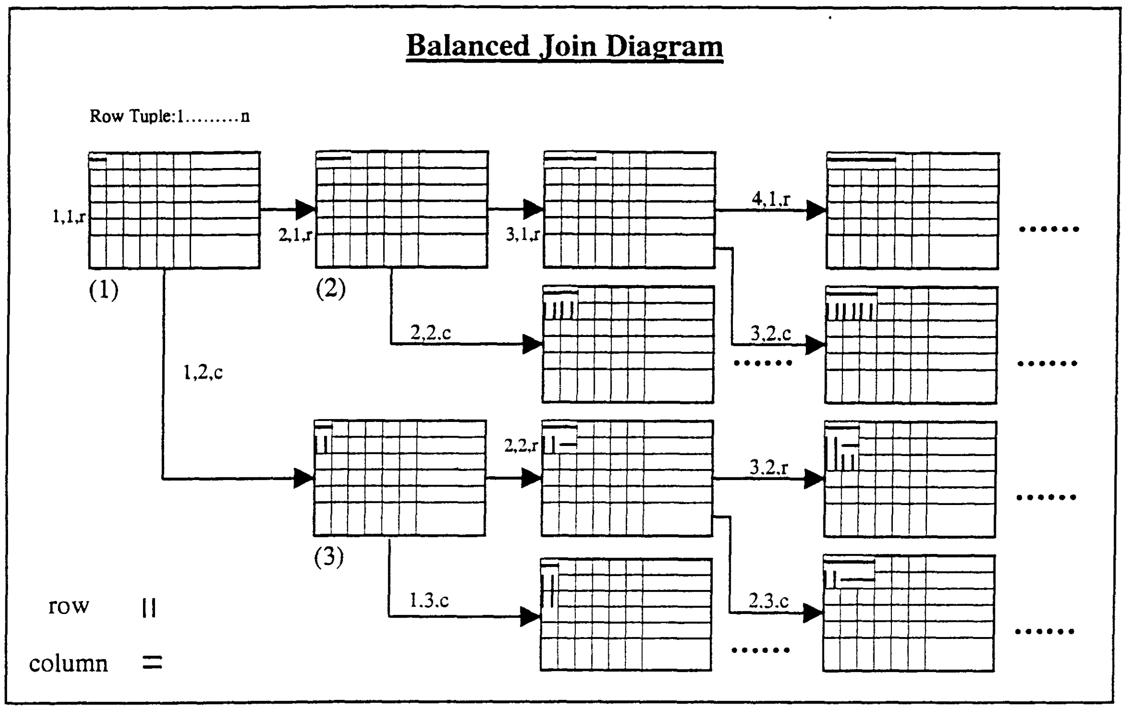


Figure 4.6

In the diagram the two tables in the join are represented on the two axes of the matrix referred to as row and column axes. Whenever a row tuple becomes available, the new subset available for processing is represented by the new set of horizontal stripes. The matrix after a row tuple is available, has been placed directly next to the old available matrix whereas the matrix formed by the entrance of a new column tuple is seen below. At each stage the coordinate of the incoming tuple is recorded along with whether it was a row tuple or a column tuple. This information is sufficient to describe the new set of tuples available to be processed and hence once these tuple designations[8] are queued there will be a serial ordering of tuple pairs that can be used within the main join algorithm.

To take advantage of the serialized ordering and still allow tuples from either source to enter the join in any arbitrary order, the algorithm needs to be implemented in a parallel mode. Three parallel processes will be needed to: 1) get the tuples from the first source, 2) get the tuples from the second source and 3) use the ordering and available tuples to carry out the actual join. The algorithm for each process is shown below. There is one shared data structure that holds the serialized entries, this is a queue with an additional method used to view the last entry without

[8] A data triplet containing (row number, column number, row/column)

removing it (implemented as a pop followed by a push, while a lock[9] is held on the queue). The first two processes use this queue to determine what has come before and then place a new entry, the third process takes the entries from the top of the queue to carry out the traditional join algorithm on that subset of the tables. The implementation of the balanced join is just as efficient as a normal join operation, as the same combinations of tuples are compared. The only significant difference is the use of three processes rather than a single process, which add a little overhead, but given the nature of the incoming data, this overhead is much less than the possible wait because of input bottlenecks.

4.3.3 Relation Reuse

Another operation that the execution engine will provide is the reuse of the stream of results of a sub-tree of operators. This can be very useful if a one operator requires the results form the same sub-query that another of the relational operators in the QEP also uses. Instead of redoing the query, the results can simply be shared between the multiple consumers. This function is implemented within the *buffered data stream (**datatable**)*. The QEP provided should ensure that appropriate objects are physically linked as in the diagram above. Once this is done, the consumers are 'registered' with the ***datatable*** and will all have access to all the tuples coming into the ***datatable.***

4.3.4 Parallel processors

There have been no experiments with parallel implementations of Java in which the threads can take advantage of multiple processors. The standard Java abstract machines do not necessarily take advantage of the multiple processors available on a machine nor do they usually take advantage of the system native threads (as for instance the new Sun Solaris Java Thread Library).

[9] Only one process can access the data structure at any one time.

Chapter 5

The Planner

Although the focus of the thesis was the execution, there was a missing connection between the user supplied SQL query and a query execution plan. Hence, a compile/planner/optimizer had to be developed to fill this gap. In this section, I will outline what the planner is required to do and how the choice of design was able to meet with these requirements. As in the other components of the system, the design is highly modularized to allow for easier updates to the functional pieces.

5.1 The Global Picture

The planner takes the user query and produces a query execution plan. The user query can actually be in the form of an SQL or constructed from options selected in a graphical interface, as described in chapter 2. The planner ensures that a QEP, which, if processed according to the API definitions of the execution engine, will produce a result that adequately, satisfies the initial query. The planner is divided into three integrated modules. These are the compiler, the planner and the optimizer. Although their functional differences are very distinct the

In addition to the information fed in from the user query the planner also requires access information on the various sources that will be used to answer the query. This data is encoded within the *specification files* (referred to as SPEC files) which will all have entries in a registry. Both the user query and the required SPEC files are compiled before the planner can use the information. Using the compiled data structures produced from the SQL query and the SPEC files, the planner tries to determine a possible ordering of the sources and then, with this ordered list of sources, generate a suitable QEP with the appropriate graph of physical operators.

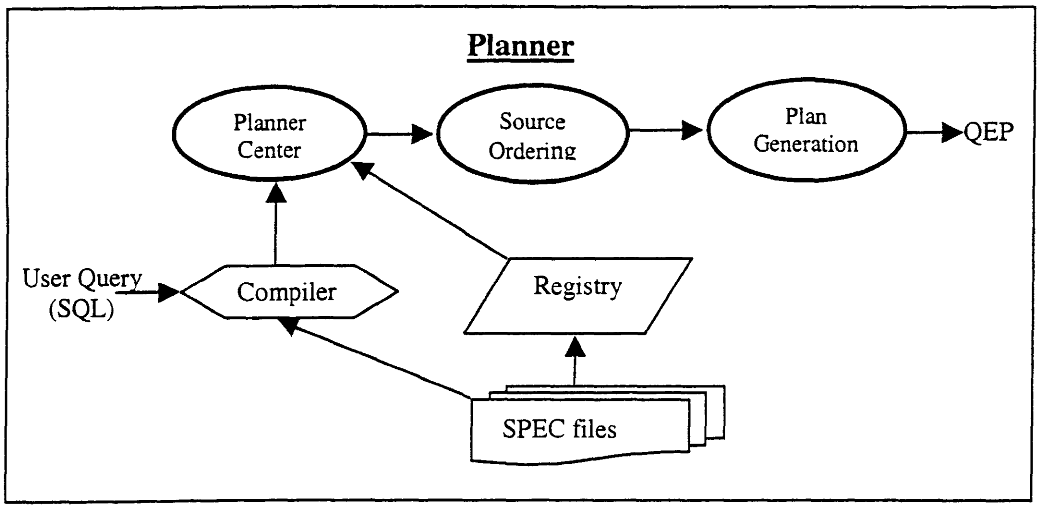


Figure 5.1

5.2 Source Specification

Before a source can be used by the system, information such as the type of source, the source address a means of data extraction, a list of provided attributes and their types must be put into a SPEC file. The type of source can vary from web pages to local flat files to relational databases. Each of the source types require a particular set of access procedures hence the necessity of having the source type as one of the parameters stored in the source specification. In addition to the type, in the case of web pages the access method is also a necessary piece of information as Web Sites use a combination of POST and GET access methods in the HTTP protocol. As a guide to the user and for initial validation checks within the planner, the source description will also have a list of the attributes that are provided. The SPEC files have a tagged-based syntax to make it simple to identify the various parts of the specification description both for the system administrators who will be writing the SPEC files and to simplify the SPEC file compiler. They tagged components are neatly cased into the *header* and the *body* of the Specification.

5.2.1 Header

The header contains the general information for each source. Most of the information is not actually used in the planning or execution but is there for identification and registry purposes. The *<HEADER>* tags clearly identify the region of the file that represents the SPEC file header. The name of the relation is also the name of the system file in which the information is stored. The string between the *<RELATION>* tags always identify the relation. The example SPEC file below describes the **nyse** (New York Stick Exchange) web source that was used in the example from chapter 2. Another aesthetic parameter is the href, which holds the base address of the main[10] source stored within the file. Given the automated data extraction a lot of information from web pages may be discarded, so this main address can give the user a start point to do further investigation or it can be used by the administrator when updating the information stored in the SPEC file.

```
<HEADER>
        <RELATION> nyse </RELATION>
        <HREF> http://www.nyse.com/ </HREF>
        <SCHEMA>
                        Ticker:string, Compname:string, URL:string
        </SCHEMA>
</HEADER>
```

The last and probably most important parameter in the header is the schema. All the attributes that the source provides or requires is stored within the schema. Types are necessary to facilitate comparisons and certain types of operations[11] (e.g. arithmetic) that may be carried out by the execution engine. All the attributes in the schema are represented by the name and type separated by a colon.

5.2.2 Body

The second section of the SPEC file is conveniently called the body and is demarked by the *<BODY>* tags. This section is made up of all the information that is necessary to allow the planner to make a QEP. The body is made up of a set of pages. Each page has a method, a URL and pattern. In the case of pages with POST methods there may also be a content parameter. Within the body section all attributes are referenced with the attribute name surrounded by two pairs of # characters, as seen with Ticker, Compname and URL in the example below.

[10] Some SPEC files may describe a variety of sources, it's up to the SPEC file author to determine which source can be referenced as the main source.
[11] Data within the system is all used within the data package which requires a type in order to create a particular data object

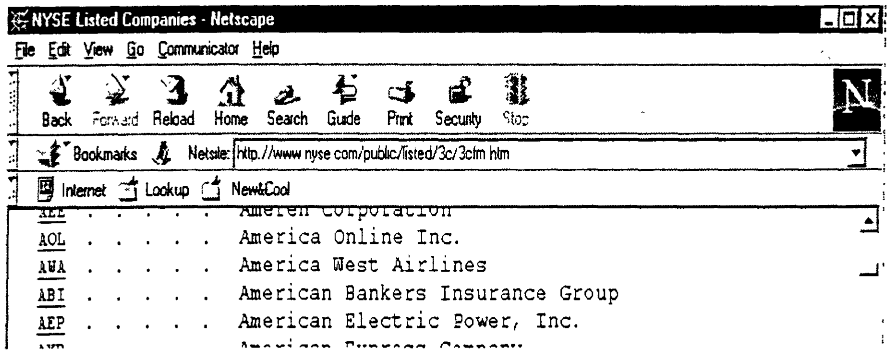
```
<BODY>
  <PAGE>

        <METHOD> GET </METHOD>
        <URL>
              http://www.nyse.com/public/listed/3c/3cfm.htm
        </URL>
        <PATTERN>
<A HREF="##URL:(.*?)##">##Ticker:(.*?)##</A>.*?<BIG>##Cname:(.*?)##</BIG>
        </PATTERN>

   </PAGE>
     .
     .
</BODY>
```

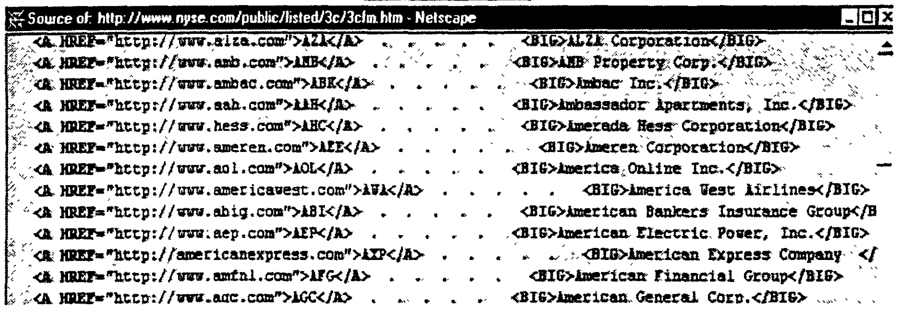
The URL gives the access location of the source. The pattern is a regular expression that is used to extract the attributes from the source. Each attribute that needs to be extracted is named along with the regular expression with which it must be matched. This regular expression is applied to page source. The diagram below shows the nyse web page from the specification file and the source (HTML) of the specified nyse web page and the location of the attributes extracted.

NYSE Web Page


NYSE Listed Companies - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Netsite: http://www.nyse.com/public/listed/3c/3cfm.htm
Internet Lookup New&Cool

```
AEE . . . . . Ameren Corporation
AOL . . . . . America Online Inc.
AWA . . . . . America West Airlines
ABI . . . . . American Bankers Insurance Group
AEP . . . . . American Electric Power, Inc.
AXP           American Express Company
```

Page Source


Source of: http://www.nyse.com/public/listed/3c/3cfm.htm - Netscape

```
<A HREF="http://www.alza.com">AZA</A> . . . . . .  <BIG>ALZA Corporation</BIG>
<A HREF="http://www.amb.com">AMB</A> . . . . . .  <BIG>AMB Property Corp.</BIG>
<A HREF="http://www.ambac.com">ABK</A> . . . . . .  <BIG>Ambac Inc.</BIG>
<A HREF="http://www.aah.com">AAH</A> . . . . . .  <BIG>Ambassador Apartments, Inc.</BIG>
<A HREF="http://www.hess.com">AHC</A> . . . . . .  <BIG>Amerada Hess Corporation</BIG>
<A HREF="http://www.ameren.com">AEE</A> . . . . . .  <BIG>Ameren Corporation</BIG>
<A HREF="http://www.aol.com">AOL</A> . . . . . .  <BIG>America Online Inc.</BIG>
<A HREF="http://www.americawest.com">AWA</A> . . . . . .  <BIG>America West Airlines</BIG>
<A HREF="http://www.abig.com">ABI</A> . . . . . .  <BIG>American Bankers Insurance Group</B
<A HREF="http://www.aep.com">AEP</A> . . . . . .  <BIG>American Electric Power, Inc.</BIG>
<A HREF="http://americanexpress.com">AXP</A> . . . . . .  <BIG>American Express Company </
<A HREF="http://www.amfnl.com">AFG</A> . . . . . .  <BIG>American Financial Group</BIG>
<A HREF="http://www.agc.com">AGC</A> . . . . . .  <BIG>American General Corp.</BIG>
```

5.2.3 Capability

Due to the nature of some of the data sources, the planner doesn't guarantee full relational capability. For example in the yahoo SPEC file (see appendix C) the schema indicates that the Ticker is one of the attributes found on this site, but this attribute is actually required before the site can be accessed. All attributes that are required are used in either the URL or content parameters within the body, they cannot be inferred from the schema. The capability describes which attributes it provides and which it requires.

5.2.4 Views

Specification files can be used to represent a view on a set of data sources. In this mode, the pages within the body can be from any set of sources, once the union of all required and provided attributes are listed within the schema. Views are especially useful when used with sources whose capabilities compliment each other, so that all the attributes in the schema can be provided by at least one of the pages described within the SPEC file's body. The views are also useful when the user wants to combine a set of regularly used sources.

5.3 Ordering the Sources

The capability and requirements of the sources implicitly determine the order in which the selected sources can be accessed. This module determines if such an execution ordering is necessary and returns the ordered list of sources, if such an ordering can be found. In essence this module carries out the capability check and ordering of sources as one operation. The main idea is to treat all pages (as defined within the SPEC files) as individual sources with a list of attributes it requires and a list of attributes provided. Starting with the set of required attributes from the query (combination of attributes in the result and those needed to handle the query conditions), the system would construct the list of sources by checking for those that supplied the required attributes. Query conditions normally simply adds to the initial list of required attributes, but in the case of an assignment constraint[12] the condition statement then becomes a source. This source will be represented as a constant data source in the execution engine, with tuples being made up from the values used in the assignment. During the ordering and plan generation phases these *constant sources* are treated as normal sources but with higher preference for their use, so that they will be definitely be chosen once needed.

[12] An attribute being set equal to a constant or needing to be in list of constants

5.3.1 The Ordering Algorithm

The ordering algorithm uses a beam search, which doesn't guarantee the best solution but has many of it's own advantages. Implementation of the search is very simple as it is modeled after a Depth First Search (DFS), with the major difference being the fact that at any one point in time there are **n** (where **n** is the size of the beam) active nodes that can be extended. The beam search is a heuristic based search which uses a distance to goal measure to determine the best **n** extended paths with which to continue the search. Heuristics can be tuned to make the choice of sources be more plausible.

The Source Ordering algorithm is shown below (fig 5.3).

```
Algorithm:

Order (Beam B)
While no complete path found in B
        NewP = new empty set of paths
        For each path p in B
                ExtendP = p.extendpaths
                NewP = Union(NewP, ExtendP)

        For each path p2in NewP
                If p2.betterthan(B.pathAt(0))
                        B.setpathAt(p2, 0)
                        While ((i<bmsize) && (p2.betterthan(B.pathAt(pos))))
                                Shift B.pathAt(pos) down one position

Return first path in B that gives a complete path
```

Algorithm:

Betterthan(Path P)

Checks to determine which path is better using predetermined criteria. In this application the number of attributes still to be provided along with the kind of source (WEB or CONSTANT) is used to determine which is better.

Algorithm:

Extendpath()

Uses the sources that have not already been used in that path to determine which can supply attributes still required. If a source supplies a set of required attributes, it is used to extend the path. The new node will have one less unused source and only the attributes still required after using the source. The algorithm returns a set of extended paths.

Figure 5.3

5.3.2 Alternate Algorithm

The beam search does not give a optimal solution. To achieve such a solution a shortest path algorithm could be used. This will ensure that only sources required to solve the query will be accessed; in the present system, according to what source relation is being used in the *join-submit* may cause extra, unnecessary web sites may be accessed. The shortest path algorithm should also guarantee a solution once a path exists. In some cases, with a sufficiently small beam the solutions may actually be removed from the beam, because they are not among the best *beamsize* paths at that point.

5.4 The Planning Phase

Once an appropriate list of data sources is produced, the plan generation module takes the initial set of required attributes, the query conditions and the resulting list of sources and uses them to construct the final **QEP**. This is done by traversing the list of sources and creating the appropriate set of relational operators needed to extract the attributes required from that source and to use those attributes as they are need within the entire query execution. At each phase of the plan construction the attributes extracted thus far are compared with those needed to apply a condition, if sufficient attributes are available the condition is placed at that position in the plan.

Considering the portfolio example, the ordering module will output the following set of sources.

Source(Condition, [79], [])
Source(www.nyse.com/public/listed, [79,78,77], [])
Source(www.cnnfn.com/, [81], [79])
Source(fastquote, [80], [79])
Source(yahoo, [82, 83, 84, 85, 86, 87], [79])

In essence, given that each of the sources which have the data required to answer the query need the Ticker before they can be accessed, the first source accessed must be one that supplies the Ticker. For this example the condition sets the Ticker to particular values, which thus made the condition a source (see 5.3). The condition source was then put as the first in the access order.

Plan constructed for query_2

```
project ( true, true, true, true, false, true, false, false, false)

  regex_fn(Perl5Pattern@1cefbb)

   join-submit ( [quote.yahoo.com/q?s=, 0, &d=v1&0=t] )

    select (true, true, true, false, true)

     regex_fn(Perl5Pattern@1cefbb)

      join-submit ( [fast.quote.com/fq/quotecom/headlines?symbols=,
0,&key=&mode=NewsHeadlines] )

         regex_fn (Perl5Pattern@1cf03e)

          join ( [(=,0,0)] )

            join-submit ([qs.cnnfn.com/cgi-bin/stockquote?symbols=, 0,])

               regex_fn (Perl5Pattern@1cef32)

                  submit ( www.nyse.com/public/listed/3c/3cfm.htm )

            constanttable ([IBM,ORCL,T,TNT,A])
```

5.5 Dynamic Optimization

Whether the engine is to be used for ad hoc queries, i.e. queries which are not know ahead of time and therefore need to go through the entire process of compilation, planning and optimization, and execution, or queries which can be compiled, planned and optimized in advance (queries which are systematically reused such as, for instance, the example query which monitors the activities of an investment portfolio), the availability of dynamic optimization mechanisms is critical for the system to be able to react to unpredictable and uncontrolled behaviors and performances of the network and sources. Every user on the Web has experienced slow or non-responding servers and networks being randomly congested. Given that many of these phenomena are largely unpredictable, provision must be made for allowing the system to react dynamically.

A very simple way to reduce the system susceptibility to these random events is to eliminate unnecessary access. As sub-queries and requests may not be fully determined at planning-optimization time, factoring may not be optimum in the QEP. The caching system implemented in the data access module in the execution engine. A more general solution to the problem of unexpected behaviors of the network and sources is the dynamic reorganization of the QEP. In many cases sources are not-responding, taking into consideration that useful work towards answering the original query can still be done with the remaining sites, a partial answer could be returned using a form of query scrambling. A partial answer is twofold: it contains a incremental query that can be later processed in order to obtain the complete answer, as well as some data that could be obtained from the available sites using an auxiliary query.

Chapter 6

Implementation Details

As in the design the implementation focus is put on the modularity of the implementation so that in each package extenuation can be easily made. This approach is useful when trying to enhance the range of allowable queries that can come into the system. One of the main differences between this system and the existing wrappers is the emphasis on parallel execution. At each stage of the implementation modules were written in sequential and parallel modes. This helped determine in which modules full advantage could be taken of the innate parallelism. In this section I will outline some of the completed modules in this phase of the project.

6.1 Implementation Language

The entire system implementation was done in JAVA. The primary object was to develop a lightweight database execution engine that could be integrated into the COIN system and several other architectures including client software on personal computers. The fact that Java's abstract machine is almost transparently supported for most operating systems allows for an implementation that is very portable. The various Java APIs, JavaBeans, JDBC, RMI, allow the smooth integration of the code into many system configurations. The garbage collected memory management that is built into Java allowed for a much faster implementation time but was a definite problem when dealing with very large data sets that require the use of many objects. This is one pitfall to the choice of Java but the effects will be out-shadowed by the other benefits of using Java. The availability of lightweight threads is probably the main reason for Java being the implementation language of choice. Concurrent data access and processing are the primary means of reducing bottlenecks that may develop when trying to import data not stored on the local disk. This could be done on in another language such as Perl or C but the processes that these languages launch will cause too much of a drain on the processor. Java provides a very good mix of *lightweight* threads and synchronization primitives that greatly simplify the implementation of concurrency. All of the implementation was done using Semantic Visual Café. The choice of Café over other programming environments was basically due to the fact that it offered the ability to compile the Java code into Native Win32 DLLs or into a Win32 executable.

6.2 Communication Protocols

All communication to remote sources was done via the Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). They were accessed using the base ***java.net*** package and an imported ***HTTPClient*** package downloaded from the WWW. The communication medium that was provided by these protocols was general and widely used enough to facilitate the integration of many data sources into the system.

6.3 Data Extraction

The wrappers will have general multi source access. An access interface between the classes that require access to data sources and the actual data access routines for the particular sources is necessary for a seamless implementation. This enables new data sources to be easily appended to the system, with no change to other modules. The access routines implemented for this thesis include flat file, http and ftp access classes.

Accessing the data from the sources is the first step to data extraction. Once the file is in main memory (or as it is being read) a relation must be constructed to store the data. In the semi-structured data sources (such as web pages) the string read during the access must be broken up into tuples and fields. The data to do such is extracted by using pattern matching routines.

6.4 Typing System

An object relational model is being used to implement the typing in the system. All data that flows through the system is instantiated as one of the type classes offered by the wrapper engine. The datatypes used are implemented using a datatype package (see appendix). The system wraps data into an object, with one method that indicates what data type it is. The base types are integers, floats and strings. These are sub-classed to represent the different methods that are necessary to access the values of the data, e.g. data_str_const is a subclass of data_str. In the example, the subclass is a constant, which returns a str when as it's value (the actual string is a attribute of the data_str_const class). This implementation extends to the attribute extraction from a tuple, e.g. data_int_att is a class that returns an attribute of type int from a tuple. The position of the attribute in the tuple is stored in the data_int_att class, hence when given a particular tuple the attribute is extracted, wrapped as a data_int object and returned. Arithmetic operations are also dealt with in the system. It is actually implemented quite similarly to the attribute extraction. In the case of arithmetic operations a class such as data_float_calc will store the type of operation to take place and the data to be used. This recursive structure conveniently allows for nested operations. The value of float is calculated only when needed, to calculate the value the object will be given the tuples required just as in the attribute data type. New data types are easily added to the system by including the definition for the base class and the required subclasses.

6.5 Parsing and Compilation

The first phase in the planner/compiler is to parse the query and the specification files for the relations identified in the query. The two parsers were actually implemented very differently. Java Compiler Compiler (JavaCC) automatically generated the SQL parser with a limited grammar used as input. JJTree, the tree building preprocessor which is used with JavaCC, enables the generation of a compilation tree that can hold the components of the SQL that are needed. The compilation tree is converted into an internal *SQL* data structure[13], which is used throughout the planning. Specification files have been written in many different formats. The structure and content has changed as the system developed. The final design of the SPEC files was simple enough to write a parser from scratch. This compiler uses the same regular expression (pattern matching) technology as in the data extraction. As with the SQL compilation, there is a *SPEC* data struture that holds all the relavent parameters needed from the specificaion files.

6.6 Code Structure & API's

All of classes implemented in the system were grouped within packages of similar classes. In this section, I will describe the three major packages built for the system.

6.6.1 Relational Operators

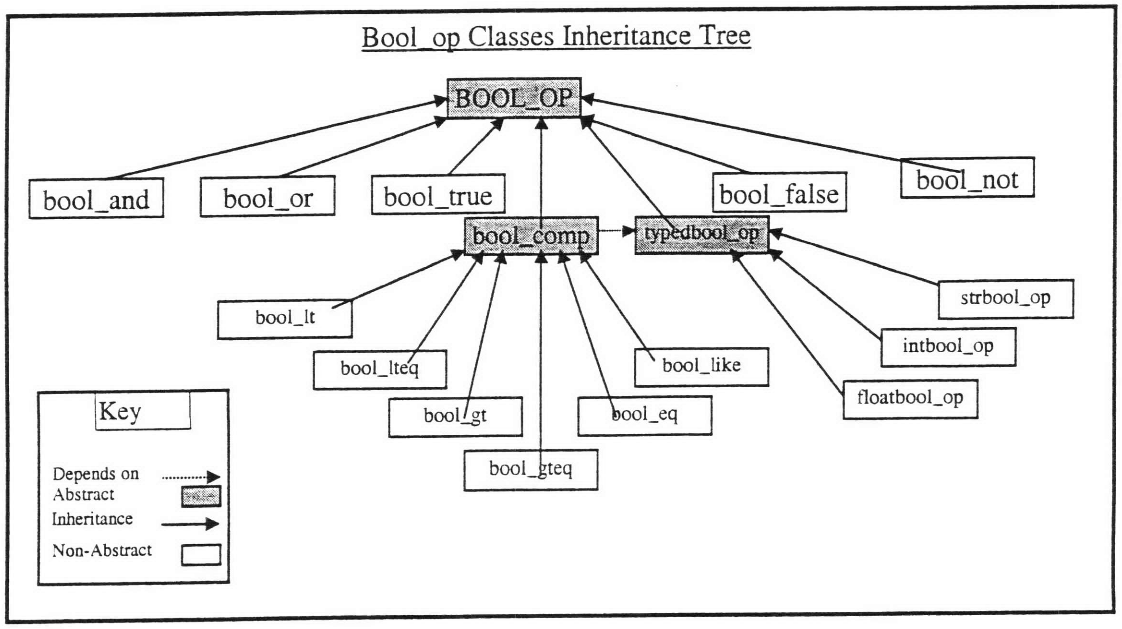
The system groups all data into tuples and then into tables which can be manipulated to get the final result required from a query. The relational operators enable the manipulation of the tables. Specific tuples can be extracted or specific attributes from a selected tuple can be combined to build new tables. Five basic operators from relational algebra have been implemented. These are Scan, Select, Project, Union and Join. Extensions of these operators, such as multi-join, are implemented to make more efficient the execution of complex plans.

A plan consists of a sequence operators acting on one or more data sources. The plans are constructed a tree of relational operators. The degree of each node in the tree depends on the type of operator situated at that node e.g. select will have a single branch, join will have two branches and the multi-join can have any number of edges. Within the plan each edge represents a link between two relational operators. Whereas during the execution phase the edges represent the stream of tuples (data table) resulting from the child operator and being used as input to the parent. All the leaf nodes on the plan hold scan operators.

6.6.2 Boolean Operators

The where clause in an SQL query must evaluate to true or false. The conditions involved in the clause can be broken up into various specific elements. The Boolean Operators package is used to represent those specific elements and provide an effective way of constructing and evaluating a conditional statement. The base operators (true, false) always return their named values (true, false). Another set of operators (and, or, not) use the results from other operators to determine their value. Hence a condition can be described as a tree of Boolean operator. This structure lent itself to a few optimizations in the experimental concurrent implementation, as not all sub-trees (sub conditions) had to be evaluated. Even though evaluation of all children would be started, in some cases the result for an operator could be determined by simply using the fastest returning result.

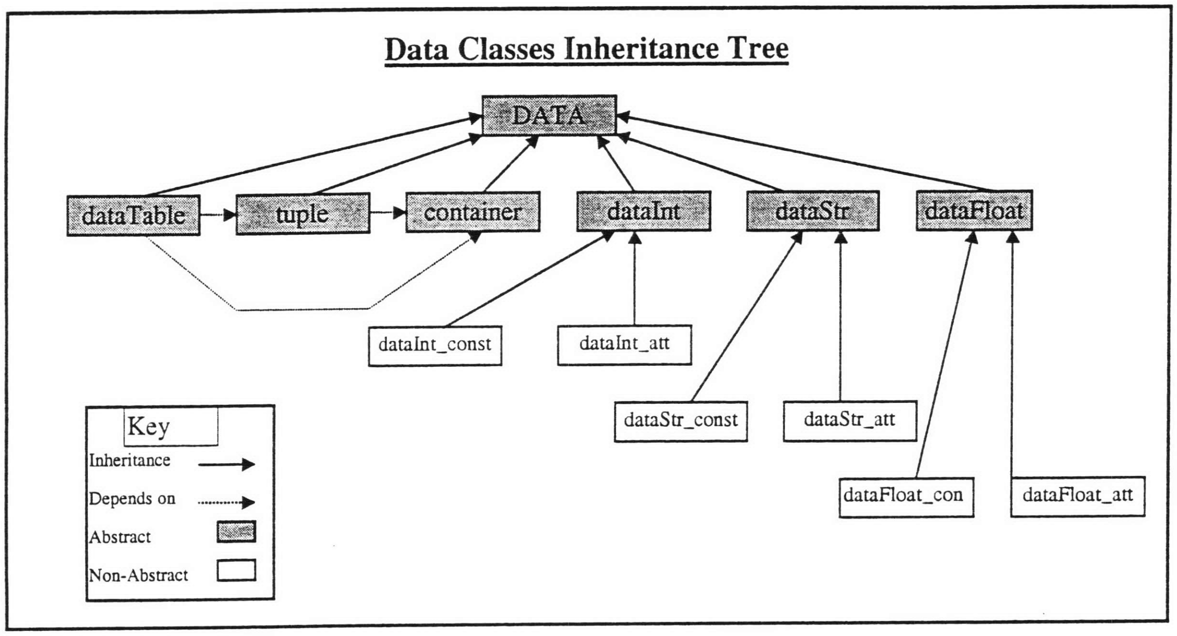
```
bool_op             ==    const_bool_op | static_bool_eval | norm_bool_op
const_bool_op       ==    true | false
norm_bool_op        ==    not bool_op | and bool_op bool_op | or bool_op bool_op
static_bool_eval    ==    static_op data data
static_op           ==    > | => | <= | < | =
```



[13] A class developed for the express use of this system.

6.6.3 Data

This package is primarily used for the containers it offers that makes manipulation of single and multiple pieces of data more manageable. The Hierarchy of classes can be represented as an n-span tree. As will be evident in most of the other packages that have been implemented, only classes that are leaves of the inheritance tree can be instantiated. All inner nodes are abstract classes. The major class in the package is <data.class>.



Container : this class is the base for implementation of the tuple and datatable classes. A tuple is a container for attributes and a datatable is a container for tuples. Hence all access to data stored in the system will be via a container. This demands that the container allow for fast searches and updates. Additions to the container should take constant O(1) time, whereas searches should be at least as efficient as a linear O(n) time search. These crude requirements can be meet by many data structures. Two data structures have been used in the current implementation. The initial one is the Vector class offered in the java.util package, this class provides the functionality of a dynamically growing array. The management of memory required to increase the size of an array is abstracted away from the user. The other method used was a standard linked list. This structure gave a better handle on adding and removing elements from the list. Querying the list took O(n) time. Although with the specific use of the list, the cost of access should be O(1) as all in the general case most or all elements of a table or tuple are accessed sequentially.

Chapter 7

System Evaluation and Conclusion

The execution engine described in this document promises to be a very useful lightweight tool that can be easily used as a stand-alone application or incorporated into a bigger system such as COIN. It is very effective in providing answers to a very broad range of SQL queries and has been successfully used to demonstrate a portfolio upkeep application (see chapter 2). The next step would be to extend the system into a fully multi-database engine.

The use of the updated system in coin is a relatively simple process, if only the bare features are required. All API calls can be adjusted to accept the format of the query, which comes out of the mediator, and a procedure can be written to make the output into a tabular HTML format. In the current COIN implementation no provision is made for streamed results to be output from the execution engine, so this fact may require adjustments made to the COIN top layer. Once all the APIs are properly set and the system made capable to accept a stream rather than all the results together, there should be a black box installation of the multi-database engine into COIN. This installation can actually be functionally divided; i.e. Initially the system can be integrated as a Web Wrapper, and hence be called by the ODBC drivers or current multi-database engine to carry out web access queries. The next phase would be integration as a multi-database engine, which requires the addition of more types of sources and protocols for accessing the data. The planner would need to be altered to account for the fact that sources may be able to accept queries, hence the source capability has more direct influence in planning.

7.1 Related Work

During the development of this thesis work, a couple very interesting and similar projects have been identified. The Disco project at Dyade and the WinMWrap project at the University of Maryland are both developing prototype mediation networks, which encounter the problem of getting a structured query-processing engine to provide the basic functionality for the construction of mediators and wrappers.

The Disco research is very similar to this project. They are developing the system to study dynamic optimization, partial answers and scrambling, i.e. what can be done during query execution to change the plan if one or more sources is not responsive. Their system runs in a sequential execution mode.

WinMWrap is also very similar. They need to use the system as a backbone for source selection research. It is uses OQL rather than SQL and has an OO data model. Their execution engine is also runs in concurrent mode.

7.2 Future Work

There are a number of improvements that can be made to the current implementation to improve the execution and range of functionality. The major changes are the migration to a totally multi-database mode and secondly, use of OR statements in query condition and arithmetic parsing for the planner. An *accessdata* package was written to deal with getting the data into the system. This package has classes that deal with Web access and local file access. Classes need to be added to deal with additional types of sources, such as ODBC sources. This change doesn't require much effort, but it goes hand in hand with improvements required in the planner, so that queries, which can be handled at the other sources, are sent to the sources in and SQL format. If the sources use a different query language, the *accessdata* class written for the source should be able to convert the SQL to whatever type of query is required.

The second set of improvements deals specifically with the planner. Given that the focus was on the execution engine, the currently available planner wasn't implemented in a manner that would guarantee the best QEP solution. An improvement would be to use a shortest path algorithm for the implementation. There could also be some changes made to factors being considered when comparing two paths. Currently the choice of sources is based on which decreases the number of required attributes by the highest amount. Taking the cost of accessing the data from the source into account would improve the efficiency of the resultant QEP. Allocating a cost to this access requires a cost model and an extension of the algorithm to include the costs.

7.3 Conclusion

Overall the implementation matches up very well with the initial requirements of the system and in some ways surpasses those expectations. Given the inconsistency of WWW access it is very difficult to measure the execution improvements over past implementation, but there is a definite observable difference for queries that require a lot of data from web sites. After this thesis, I hope to help in any improvements being made to the system, as I do believe that the software can be very useful to very large range of applications.

Appendix A: User Guide

Installation

All the required class files along with the source code are stored in a fixed directory structure in a zip file on the installation diskette. The software has **not** been compiled into an executable due to the unavailability of certain libraries which were used for the implementation. Using the Semantic Café development environment the user can exeute the demo by simply typying:

```
java demo
```

A base interface is being built which will allow the user to execute this program even if Café is not available. Hence once Java is installed on the machine the above command can be executed.

Specification Files

All specification files must be put into the registry. The syntax is very similar to the HTML tagged syntax.

Creating

The specification file

- Find a source that you want to wrap (this can be a single page or a set of pages).
- Determine what attributes can be extracted from the source, using these attributes, along with their type (you specify), put together the schema for the specification file.
- For each page determine the regular expression needed to extract the attributes. Put this regular expression in the pattern section of the page.

  Place the name of the attribute along with the expression required to extract the attribute, separated by a colon, between two pairs of #'s

  e.g. ##Ticker:(.*?)##
- Save the file as whatever relation name you like, something that represents the source being accessed would be better. Thhe file should be saved *your_relation_name*.spec in the registry.

Using

Once the specification file is placed into the registry directory, which is a directory called registry in the directory to which the software is installed, then it can be accessed by the system, simply using the name of the spec file in an SQL statement.

Appendix B: Code Statistics

The first 8 packages listed below have been implemented within the system, and the last 2 were downloaded from the web. All other packages used are core Java classes.

PACKAGE	Public Classes	Private Classes	Code Size	Description
accessdata	2	0	200	The interface between the engine and the data sources that it must access. Each class in the package allows access to a particular type of data source. There is one main class '*data_access*' that the system calls which activates the correct object.
bool_op	12	3	800	Boolean Operators which when evaluated return true or false. These operators have logical and comparative types. New comparative operators can be appended if following the format of the *bool_lt* or *bool_eq* operators. Additional logical operators can be added following the format of the *bool_and* or *bool_not* operators.
datatype	14	0	700	All data that passed through the system is contained within a specific datatype object. These objects represent all the types that the system can handle. Each class represents a different data type. Using any of the classes as a base, new data type classes can be added to the system.
SQLParser	23	0	1360	JJTree and JavaCC automatically generated this set of classes. Once generated the classes were amended so that the stored information could be extracted from the parse tree.
SPECParser	1	0	120	This package is made up of a single class *SPECRegex* which uses the regular expression package to parse the specification files.
planner	3	9	1400	This package has data structures required to store and manipulate source information and SQL attributes. The central planning algorithms are held in the class *planner,*
myutil	3	0	200	This contains all the classes that are used throughout the system but do not lie within one of the functional descriptions of the other packages.
rel_op	16	3	1200	Relational Operator classes are held in this package. Once again,each operator is represented by a single class. There are a couple classes which are used to store the relational operator which are only used within the rel_op package.
HTTPClient	n/a	n/a	n/a	Used for net access using HTTP, FTP and other common protocols. This packages was downloaded from ...
OROMatcher	n/a	n/a	n/a	Allows Perl regular expression in Java.

Appendix C: Specification Files

Fastquote Specification

```
<HEADER>
<RELATION> fastquote </RELATION>
<HREF> GET http://fast.quote.com </HREF>
<SCHEMA> Ticker:string, NewsURL:string, Headline:string </SCHEMA>
</HEADER>

<BODY>
<PAGE>
      <METHOD> GET </METHOD>
      <URL>
http://fast.quote.com/fq/quotecom/headlines?symbols=##Ticker##&key=&mod
e=NewsHeadlines </URL>

      <PATTERN> <TD>\s*<FONT\sSIZE\=\-1><A
HREF="##NewsURL:(.*?)##">##Headline:(.*?)##</A><BR> </PATTERN>
</PAGE>
</BODY>
```

Yahoo Specification

```
<HEADER>
<RELATION>yahoo</RELATION>
<HREF>GET http://quote.yahoo.com</HREF>
<SCHEMA> Ticker:string, LastTrade:string, Change:real, Changepts:real,
Changepct:real, Volume:integer </SCHEMA>
</HEADER>
<BODY>
 <PAGE>
      <METHOD> GET </METHOD>
      <URL> http://quote.yahoo.com/q?s=##Ticker##&d=v1&o=t </URL>
      <PATTERN>
<ahref="/q\?s=.*?&d=t">.*?</a>\s+##LastTrade:(.*?)##\s+<b>##Change:(.*?
)##</b>\s+##Changepts:(.*?)##\s{2,}##Changepct:(.*?)##\s+##Volume:(.*?)
## <small>
      </PATTERN>
 </PAGE>

</BODY>
```

New York Stock Exchange (nyse) Specification

```
<HEADER>
<RELATION> nyse </RELATION>
<HREF> GET http://www.nyse.com/ </HREF>
<SCHEMA> Ticker:string, Cname:string, URL:string, Rec:string,
Last:string </SCHEMA>
</HEADER>
<BODY>

 <PAGE>
      <METHOD> GET </METHOD>
      <URL> http://www.nyse.com/public/listed/3c/3cfm.htm </URL>


      <PATTERN> <A
HREF="##URL:(.*?)##">##Ticker:(.*?)##</A>.*?<BIG>##Cname:(.*?)##</BIG>
      </PATTERN>

 </PAGE>

 <PAGE>
      <METHOD> POST </METHOD>
      <URL>
      http://www.ultra.zacks.com/cgi-
bin/ShowFreeCompRepUSAToday?ticker=##Ticker##
      </URL>

      <PATTERN>
            Current Average Recommendation:  <font
color=#4003a6>[\s]*##Rec:(.*?)##[\s]*<\/font>
      </PATTERN>

 </PAGE>

 <PAGE>
      <METHOD> GET </METHOD>
      <URL>
            http://qs.cnnfn.com/cgi-bin/stockquote?symbols=##Ticker##
      </URL>

      <PATTERN> >last.*?<b>##Last:(.*?)##<\/FONT> </PATTERN>

 </PAGE>


</BODY>
```

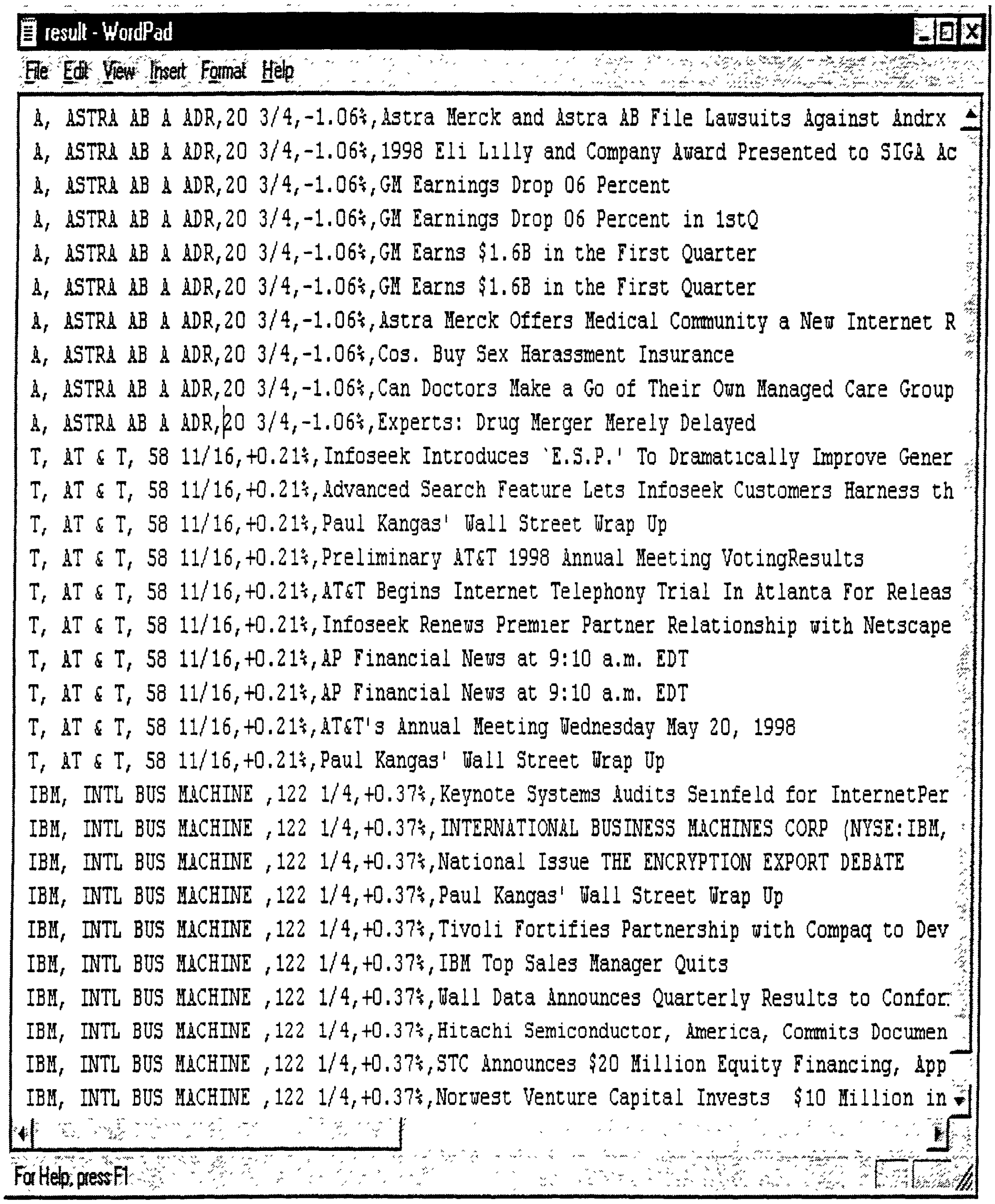
Finance Specification

```
<HEADER>
<RELATION> finance </RELATION>
<HREF> GET http://www.moneynet.com/ </HREF>
<SCHEMA> Ticker:string, News:string, Time:string, Date:string </SCHEMA>
</HEADER>
<BODY>
 <PAGE>
      <METHOD> GET </METHOD>
      <URL>
http://www.moneynet.com/MONEYNET/coNews/newsHeadlines.mhtml?SYMBOL=##Ti
cker## </URL>

      <PATTERN> <FONT SIZE=-
1>\s*##Date:(.*?)##\s+##Time:(.*?)##\s+</FONT></FONT>\s+</TD>\s+<TD
VALIGN=BASELINE>\s*##News:(.*?)##\s*<\/TD> </PATTERN>

 </PAGE>
</BODY>
```

Appendix D: Query Results



```
A, ASTRA AB A ADR,20 3/4,-1.06%,Astra Merck and Astra AB File Lawsuits Against Andrx
A, ASTRA AB A ADR,20 3/4,-1.06%,1998 Eli Lilly and Company Award Presented to SIGA Ac
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earnings Drop 06 Percent
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earnings Drop 06 Percent in 1stQ
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earns $1.6B in the First Quarter
A, ASTRA AB A ADR,20 3/4,-1.06%,GM Earns $1.6B in the First Quarter
A, ASTRA AB A ADR,20 3/4,-1.06%,Astra Merck Offers Medical Community a New Internet R
A, ASTRA AB A ADR,20 3/4,-1.06%,Cos. Buy Sex Harassment Insurance
A, ASTRA AB A ADR,20 3/4,-1.06%,Can Doctors Make a Go of Their Own Managed Care Group
A, ASTRA AB A ADR,20 3/4,-1.06%,Experts: Drug Merger Merely Delayed
T, AT & T, 58 11/16,+0.21%,Infoseek Introduces `E.S.P.' To Dramatically Improve Gener
T, AT & T, 58 11/16,+0.21%,Advanced Search Feature Lets Infoseek Customers Harness th
T, AT & T, 58 11/16,+0.21%,Paul Kangas' Wall Street Wrap Up
T, AT & T, 58 11/16,+0.21%,Preliminary AT&T 1998 Annual Meeting VotingResults
T, AT & T, 58 11/16,+0.21%,AT&T Begins Internet Telephony Trial In Atlanta For Releas
T, AT & T, 58 11/16,+0.21%,Infoseek Renews Premier Partner Relationship with Netscape
T, AT & T, 58 11/16,+0.21%,AP Financial News at 9:10 a.m. EDT
T, AT & T, 58 11/16,+0.21%,AP Financial News at 9:10 a.m. EDT
T, AT & T, 58 11/16,+0.21%,AT&T's Annual Meeting Wednesday May 20, 1998
T, AT & T, 58 11/16,+0.21%,Paul Kangas' Wall Street Wrap Up
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Keynote Systems Audits Seinfeld for InternetPer
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,INTERNATIONAL BUSINESS MACHINES CORP (NYSE:IBM,
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,National Issue THE ENCRYPTION EXPORT DEBATE
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Paul Kangas' Wall Street Wrap Up
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Tivoli Fortifies Partnership with Compaq to Dev
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,IBM Top Sales Manager Quits
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Wall Data Announces Quarterly Results to Confor
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Hitachi Semiconductor, America, Commits Documen
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,STC Announces $20 Million Equity Financing, App
IBM, INTL BUS MACHINE ,122 1/4,+0.37%,Norwest Venture Capital Invests  $10 Million in
```

Appendix E: Main Packages

In the following pages you will find the Java Documentation created for the packages in the system. Each package is made up of a set of classes and each class is made up of a set of methods and attributes. The list of classes in the package will be shown and then selected classes with the method and attributes will be shown.

The packages shown are as follows:

Relational Operator Package (rel_op)

Boolean Operator Package (bool_op)

Data Access Package (accessdata)

Data Package (datatype)

Package index

Other Packages

- package accessdata
- package boolean_operator
- package datatype
- package mask
- package myutil
- package rel_op

package accessdata

Class index

- data_access
- web_page

Class accessdata.data_access

```
accessdata.data_access
```

public class **data_access**

Constructor index

- **data_access**(String)

Method index

- **done**()
- **get_data**()
- **getdata**()

Constructors

data_access

```
public data_access(String address)
```

Methods

getdata

```
public DataInputStream getdata()
```

get_data

```
public String get_data()
```

done

```
public boolean done()
```

Class accessdata.web_page

```
accessdata.web_page
```

public class **web_page**

Constructor index

- **web_page**(String)

Method index

- **get_page**()
- **get_page**(String)
- **getpage**(String)
- **main**(String[])

Constructors

web_page

```
public web_page(String address)
```

Methods

main

```
public static void main(String arg[])
```

getpage

```
public DataInputStream getpage(String args)
```

get_page

```
public static String get_page(String args)
```

get_page

```
public String get_page()
```

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

package boolean_operator

Class index

- bool_and
- bool_eq
- bool_false
- bool_gt
- bool_gteq
- bool_like
- bool_lt
- bool_lteq
- bool_not
- bool_op
- bool_or
- bool_true
- int_bool_op
- str_bool_op

Overrides:
run in class bool_op

● killprocesses

```
public void killprocesses()
```

Overrides:
killprocesses in class bool_op

All Packages Class Hierarchy This Package Previous Next Index

Class boolean_operator.bool_eq

```
boolean_operator.bool_op
   |
   +----boolean_operator.bool_eq
```

public class **bool_eq**
extends bool_op

Constructor index

- **bool_eq**(data, data)
- **bool_eq**(data_int, data_int)
- **bool_eq**(data_str, data_str)

Method index

- **eval**(llist)
- **killprocesses**()
- **run**()
- **toString**()

Constructors

bool_eq

```
public bool_eq(data_str left,
               data_str right)
```

bool_eq

```
public bool_eq(data left,
               data right)
```

bool_eq

```
public bool_eq(data_int left,
               data_int right)
```

Methods

All Packages Class Hierarchy Index

package datatype

Class index

- data
- data_float
- data_float_att
- data_float_const
- data_instream
- data_int
- data_int_att
- data_int_const
- data_str
- data_str_att
- data_str_const
- datatable
- llist
- tuple

Class datatype.data

```
datatype.data
```

public abstract class **data**

Constructor index

- **data**()

Method index

- **display**()
- **toString**()
- **type**()

Constructors

data

```
public data()
```

Methods

type

```
public abstract String type()
```

display

```
public abstract void display()
```

toString

```
public abstract String toString()
```

Class datatype.datatable

```
datatype.data
   |
   +----datatype.datatable
```

public class **datatable**
extends data

Constructor index

- datatable()
- datatable(int)
- datatable(String)

Method index

- addtuple(tuple)
- cwritefirst()
- display()
- getpos()
- gettupleat(int)
- gotostart()
- hasMoreElements()
- nexttuple()
- setdone()
- size()
- toString()
- type()
- writeto(DataOutputStream)

Constructors

datatable

```
public datatable()
```

datatable

```
public datatable(String strdata)
```

datatable

```
public datatable(int n)
```

Methods

size

```
public int size()
```

type

```
public String type()
```

Overrides:
type in class data

addtuple

```
public void addtuple(tuple t)
```

hasMoreElements

```
public boolean hasMoreElements()
```

setdone

```
public void setdone()
```

gettupleat

```
public tuple gettupleat(int pos)
```

gotostart

```
public void gotostart()
```

getpos

```
public String getpos()
```

cwritefirst

```
public void cwritefirst()
```

nexttuple

```
public tuple nexttuple()
```

toString

```
public String toString()
```

Overrides:
toString in class data

display

```
public void display()
```

Overrides:
display in class data

writeto

```
public void writeto(DataOutputStream dos)
```

All Packages Class Hierarchy This Package Previous Next Index

Class datatype.tuple

```
datatype.data
   |
   +----datatype.tuple
```

public class **tuple**
extends data

Constructor index

- **tuple**()
- **tuple**(String)
- **tuple**(tuple)

Method index

- **addelement**(data)
- **display**()
- **elementAt**(int)
- **getDisplayString**()
- **numattributes**()
- **size**()
- **toString**()
- **type**()

Constructors

tuple

```
public tuple()
```

tuple

```
public tuple(tuple orig)
```

tuple

```
public tuple(String sdata)
```

Methods

size

```
public int size()
```

type

```
public String type()
```

Overrides:
type in class data

toString

```
public String toString()
```

Overrides:
toString in class data

getDisplayString

```
public String getDisplayString()
```

display

```
public void display()
```

Overrides:
display in class data

addelement

```
public void addelement(data d)
```

elementAt

```
public data elementAt(int pos)
```

numattributes

```
public int numattributes()
```

All Packages Class Hierarchy This Package Previous Next Index

Class datatype.data_int

```
datatype.data
   |
   +----datatype.data_int
```

public abstract class **data_int**
extends data

Constructor index

- **data_int**()

Method index

- **type**()
- **value**()
- **value**(llist)

Constructors

data_int

```
public data_int()
```

Methods

type

```
public String type()
```

Overrides:
type in class data

value

```
public abstract int value()
```

value

```
public abstract int value(llist lotup)
```

All Packages Class Hierarchy This Package Previous Next Index

Class datatype.data_int_att

```
datatype.data
   |
   +----datatype.data_int
           |
           +----datatype.data_int_att
```

public class **data_int_att**
extends data_int

Constructor index

- **data_int_att**(int, int)

Method index

- **display**()
- **toString**()
- **value**()
- **value**(llist)

Constructors

data_int_att

```
public data_int_att(int tn,
                    int an)
```

Methods

value

```
public int value(llist lotup)
```

Overrides:
value in class data_int

value

```
public int value()
```

Overrides:
value in class data_int

● toString

```
public String toString()
```

Overrides:
toString in class data

● display

```
public void display()
```

Overrides:
display in class data

Class datatype.data_int_const

```
datatype.data
   |
   +----datatype.data_int
           |
           +----datatype.data_int_const
```

public class **data_int_const**
extends data_int

Constructor index

- **data_int_const**(int)

Method index

- **display**()
- **toString**()
- **value**()
- **value**(llist)

Constructors

data_int_const

```
public data_int_const(int intval)
```

Methods

value

```
public int value()
```

Overrides:
value in class data_int

value

```
public int value(llist lotup)
```

Overrides:
value in class data_int

toString

```
public String toString()
```

Overrides:
toString in class data

display

```
public void display()
```

Overrides:
display in class data

All Packages Class Hierarchy This Package Previous Next Index

Class datatype.data_instream

```
datatype.data
   |
   +----datatype.data_instream
```

public class **data_instream**
extends data

Constructor index

- **data_instream**(InputStream)

Method index

- **display**()
- **toString**()
- **type**()
- **value**()

Constructors

data_instream

```
public data_instream(InputStream ins)
```

Methods

type

```
public String type()
```

Overrides:
type in class data

display

```
public void display()
```

Overrides:
display in class data

toString

```
public String toString()
```

Overrides:
toString in class data

value

```
public InputStream value()
```

All Packages Class Hierarchy This Package Previous Next Index

All Packages Class Hierarchy Index

---

package rel_op

Class index

- constanttable
- environment
- frmAccess
- join
- llist_new
- multiscan
- paramaddress
- plan
- project
- regex_fn
- relational_op
- scan
- scheduler
- select
- submit
- union

All Packages Class Hierarchy This Package Previous Next Index

Class rel_op.select

```
rel_op.relational_op
   |
   +----rel_op.select
```

public class **select**
extends relational_op

Constructor index

- **select**(llist_new, llist, bool_op)

Method index

- **execute**()
- **run**()
- **toString**()

Constructors

select

```
public select(llist_new lstorelop,
              llist lstomask,
              bool_op bo)
```

Methods

execute

```
public datatable execute()
```

Overrides:
execute in class relational_op

run

```
public void run()
```

toString

```
 public String toString()
```

All Packages Class Hierarchy This Package Previous Next Index

Class rel_op.constanttable

```
rel_op.relational_op
   |
   +----rel_op.constanttable
```

public class **constanttable**
extends relational_op

Constructor index

- **constanttable**(Vector)

Method index

- **execute**()
- **run**()
- **toString**()

Constructors

constanttable

```
public constanttable(Vector vct)
```

Methods

run

```
public void run()
```

execute

```
public datatable execute()
```

Overrides:
execute in class relational_op

toString

```
public String toString()
```

All Packages Class Hierarchy This Package Previous Next Index

Class rel_op.scheduler

```
rel_op.scheduler
```

public class **scheduler**

Constructor index

- **scheduler**()

Method index

- **cschedule**(String, sub_rel_op, int)
- **run**()
- **schedule**(String)
- **setdone**()

Constructors

scheduler

```
public scheduler()
```

Methods

schedule

```
public DataInputStream schedule(String addr)
```

cschedule

```
public synchronized void cschedule(String addr,
                                   sub_rel_op sro,
                                   int tn)
```

run

```
public void run()
```

● setdone

```
public void setdone()
```

All Packages Class Hierarchy This Package Previous Next Index

Class rel_op.regex_fn

```
rel_op.relational_op
   |
   +----rel_op.regex_fn
```

public class **regex_fn**
extends relational_op

Constructor index

- **regex_fn**(llist_new, String, Vector)

Method index

- **execute**()
- **run**()
- **toString**()

Constructors

regex_fn

```
public regex_fn(llist_new lorelops,
                String regex,
                Vector loattr)
```

Methods

execute

```
public datatable execute()
```

Overrides:
execute in class relational_op

run

```
public void run()
```

toString

```
 public String toString()
```

Class rel_op.environment

```
rel_op.environment
```

public class **environment**

Constructor index

- **environment**()

Method index

- **closestream**()
- **getregistrylocation**()
- **getscheduler**()
- **numstreams**()
- **openstream**()
- **setregistrylocation**(String)

Constructors

environment

```
public environment()
```

Methods

numstreams

```
public int numstreams()
```

openstream

```
public void openstream()
```

closestream

```
public void closestream()
```

● getscheduler

```
public scheduler getscheduler()
```

● getregistrylocation

```
public String getregistrylocation()
```

● setregistrylocation

```
public void setregistrylocation(String addr)
```

All Packages Class Hierarchy This Package Previous Next Index

Class boolean_operator.bool_false

```
boolean_operator.bool_op
   |
   +----boolean_operator.bool_false
```

public class **bool_false**
extends bool_op

Constructor index

- **bool_false**()

Method index

- **eval**(llist)
- **killprocesses**()
- **run**()
- **toString**()

Constructors

bool_false

```
public bool_false()
```

Methods

eval

```
public boolean eval(llist lotup)
```

Overrides:
eval in class bool_op

run

```
public void run()
```

Overrides:
run in class bool_op

● killprocesses

```
public void killprocesses()
```

Overrides:
killprocesses in class bool_op

● toString

```
public String toString()
```

All Packages Class Hierarchy This Package Previous Next Index

Bibliography

1. Cheng Hian Goh. *Representing and Reasoning about Data Semantics in Heterogeneous Systems.* PhD dissertation, Massachusetts Institue of Technology, Sloan School of Management, December 1996.

2. Ricardo Ambrose, Phillipe Bonnet, Stéphane Bressan, and Jean-Robert Gruser. *Three Light-Weight Execution Engines in Java for Web Data-Intensive Applications.* MIT Sloan School of Management, March 1998

3. Marta Jakobisiak. Programming the web – design and implementation of a multidatabase browser. Technical Report CISL WP #96-04, MIT Sloan School of Management, May 1996

4. G. Graefe. *Query evaluation techniques for large database.* ACM Computing Surveys, 1993.

5. Kofi Duodu Fynn. *A Planner/Optimizer/Executioner for Context Mediated Queries.* MIT Sloan School of Management, May 1997