OPTIMIZING INFORMATION RETRIEVAL FROM DISPARATE MENU DRIVEN SYSTEMS

by

Howard L. Gerber

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Electrical Science and Engineering

at the Massachusetts Institute of Technology

May 1989

Author_____
Department of Electrical Engineering and Computer Science
May 22, 1989

Certified by_____
Professor Stuart E. Madnick
Thesis Supervisor

Accepted by_____
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

OPTIMIZING INFORMATION RETRIEVAL FROM DISPARATE MENU DRIVEN DATABASES

by

Howard L. Gerber

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 1989

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

ABSTRACT

Menu driven database systems provide a user friendly interface to their stored information, though they are not easily accessed in a preprogrammed manner. This is a problem for the composite information system designer, whose goal is to interface with a large number of disparate databases. This thesis proposes a solution to this problem, based on a model developed for Reuters DATALINE service and theoretically extended to Reuters TEXTLINE.

The research done on Dataline yielded the necessity of installing some degree of intelligence about a remote menu driven system in a Local Query Processor(LQP). This intelligence consists of a representation of the output format of the DBMS, which enables the LQP to select the most cost effective menu choices available. An information filter which handles standardly formated data well is also necessary, since the output received from a menu driven system is often in the form of a table. As long as these two issues are kept in mind during the design of the LQP's abstract local query's column and condition lists, most other concepts can be derived from the LQP model developed for SQL type databases.

Thesis Supervisor: Stuart E. Madnick
Title: Professor of Management Science

Dedication

To Stu Madnick, who always understood how long a design would *really* take to implement...

To Rich Wang, who helped me iron out the real *meat and potatoes* of my thesis...

To Mia Paget, who always had a clear view of the issues I was confronting, and who put up with my impatient phone calls and meeting requests...

To Jean-Eloi Dussartre, who helped with the AWK coding...

And finally, to Mom and Dad, Jeff and Nat. Thanks for the support and all the help you've provided, enabling me to reach this momentous occasion in my life...

Table of Contents

List of Figures

Chapter 1

Introduction to Information Systems

Information systems have existed in different forms since the beginning of human enterprise. Histories of governments and battles provide numerous examples of systems established to ensure a proper flow of information to decision makers. This need for complete and accurate information has evolved along with the growing complexity of society and technology. Modern managers need to make decisions involving larger numbers of factors with far greater numbers of outcomes, and it is the computer age and particularly the development of Database Management Systems(DBMSs) that have made this possible.[1]

Prior to the existence of Database Management Systems, companies had to rely on manual retrieval of the information they needed, which was a costly, time consuming and inefficient process. Data was maintained by individuals who organized it to suit their own personal needs. In most sizable corporations now, however, the corporate division is the smallest unit to keep its own databases. Although information is now being kept for a larger unit of a firm, this still presents a large problem to the manager, who desires to integrate all important data into a composite solution. The modern manager has to cope with even more complex data acquisition problems due to the globalization of the world economy, whereby firms have expanded beyond their traditional geographic boundaries. In order to compete on an international level, these firms need accurate and complete information on all markets in which they compete. Research is being done to alleviate these problems, and Composite Information Systems(CIS), which allow a user to access multiple disparate databases from a single system, appear to be the way of the future.[2]

---

[1]For further information, refer to [Radford 73].

[2]Refer to [Champlin 88].

There are many problems involved in overcoming the discrepancies between different databases. One point of major importance is that databases are set up in a number of different formats. For example, ORACLE is a relational system which is accessed through SQL type queries. On the other hand, Reuters DATALINE and TEXTLINE databases are accessed through menu driven interfaces, providing a more user-friendly design for the non-technical user. In either case, the person trying to extract information from a database has to know the specific inputs each system requires. For example, to terminate the current selection and revert to a higher level menu selection in Reuters services, a "\" must be selected. I.P. Sharp, another menu driven system, requires the word "back" instead. It is apparent that a Composite Information System would need to incorporate the specifics of all databases it accesses so that these fine points would be transparent to the CIS user.

The unit within the Composite Information System(CIS/TK) being developed at MIT's Sloan School which deals with overcoming these connection and input format problems is the Local Query Processor(LQP). LQPs are necessary for each database system to be accessed by CIS/TK, but up until this point, had only been available for SQL type systems. This thesis introduces a theoretical approach for creating an LQP for a menu driven system, namely any with a format similar to Reuters Dataline. Although many of the concepts developed here are extendible to other types of menu driven systems, a careful analysis of each system and its corresponding abstract local query is necessary.

This paper progresses from the discussion of what an LQP is through the extendibility of the LQP for a menu driven system designed here. Chapter 2 introduces the LQP and juxtaposes the LQP structure for an SQL system with the one developed in this thesis. The idea of breaking a menu driven system LQP into *retrieval* and *filtering* modules is also presented. Chapter 3 discusses the retrieval module, focusing on developing a scheme for mapping the columns specified to the alphanumeric menu selections available. Chapter 4 studies the filtering module, which consists of the condition list parser, the filter and the

result reader. An analysis of how to approach condition list parsing is stepped through, and *AWK* is introduced as a good filter for data in table form. Chapter 5 extends the concepts developed to Reuters Textline system and lays out potential future work on menu driven system LQPs.

Chapter 2

The Local Query Processor

2.1 Overview

The Local Query Processor(LQP) is the module within the CIS architecture which establishes the physical connection between the host and the appropriate remote machines where information is stored. Using its programmed intelligence, the LQP performs all functions necessary to extract information from a selected database, enabling the common user to avoid having to learn the specifics of all the machines he desires to access. In this sense, the LQP acts as a *virtual driver* of the remote machine, indistinguishable to the machine from the users of real terminals. This is a very powerful approach since very few, if any, changes need to be made to existing systems for physical or logical connections.[3]

2.2 Interface to CIS/TK

The Local Query Processor is defined as an object in the CIS/TK framework. An LQP object exists for each database which is to be integrated into the system. For each of these LQP objects, four methods are supported: self-info, get-tables, get-columns and get-data. The LQP is called into action by the get-data method sent from the Global Query Processor, which is one level up in the CIS/TK architecture(refer to figure 2-1). This method provides the LQP with an abstract local query, which contains enough information for the LQP to know where to find the desired data and what specifically is desired.

-->*SELF-INFO* provides the CIS/TK user with information about the database being accessed. This method requires no arguments.

-->*GET-TABLES* returns a list of tables associated with an LQP for

---

[3]Refer to [Wang 88].

an SQL type system. Since menu driven systems do not work directly (although in some cases implicitly) with tables, this method is only used in this thesis to display which system(i.e. Dataline or Textline) is currently being accessed.

-->*GET-COLUMNS* returns a list of the columns available for use with an LQP. On an SQL type system, these columns can be read directly, though they must be placed on the local machine when dealing with a menu driven system.

-->*GET-DATA* calls the LQP into action. It expects an abstract local query as its argument and returns a list of the data requested.

2.3 The Abstract Local Query

An abstract local query consists of a list of three sublists, each of which contains a different aspect of how to find the information desired by the CIS user(see figure 2-2). The first sublist has the name of the table the user wants to draw data from. On an SQL based system like ORACLE, the actual name of the table would be placed here. However, menu driven systems work with alphanumeric inputs specifying hidden databases, rather than overt tables. Therefore, this sublist of the abstract local query can be used to identify which menu driven system is being accessed, namely DATALINE or TEXTLINE in this thesis.

The second sublist of the abstract local query is a list of the columns desired from the selected table/menu driven system. The term *columns* actually refers to the columns in an SQL system, so the definition is being extended in this analysis. The column list is responsible for specifying the information to be returned by a system, which in Dataline's case is dependent on receiving a proper company code. The columns supported should have the most practical value to potential users of the system, like code, sales, current assets and earnings within Dataline. It is sensible that columns like these will be requested more often than one like net cash surplus. The columns should also stand for values which are applicable to any call to a DBMS, reserving the specific requests(i.e. for a company) for the condition list.

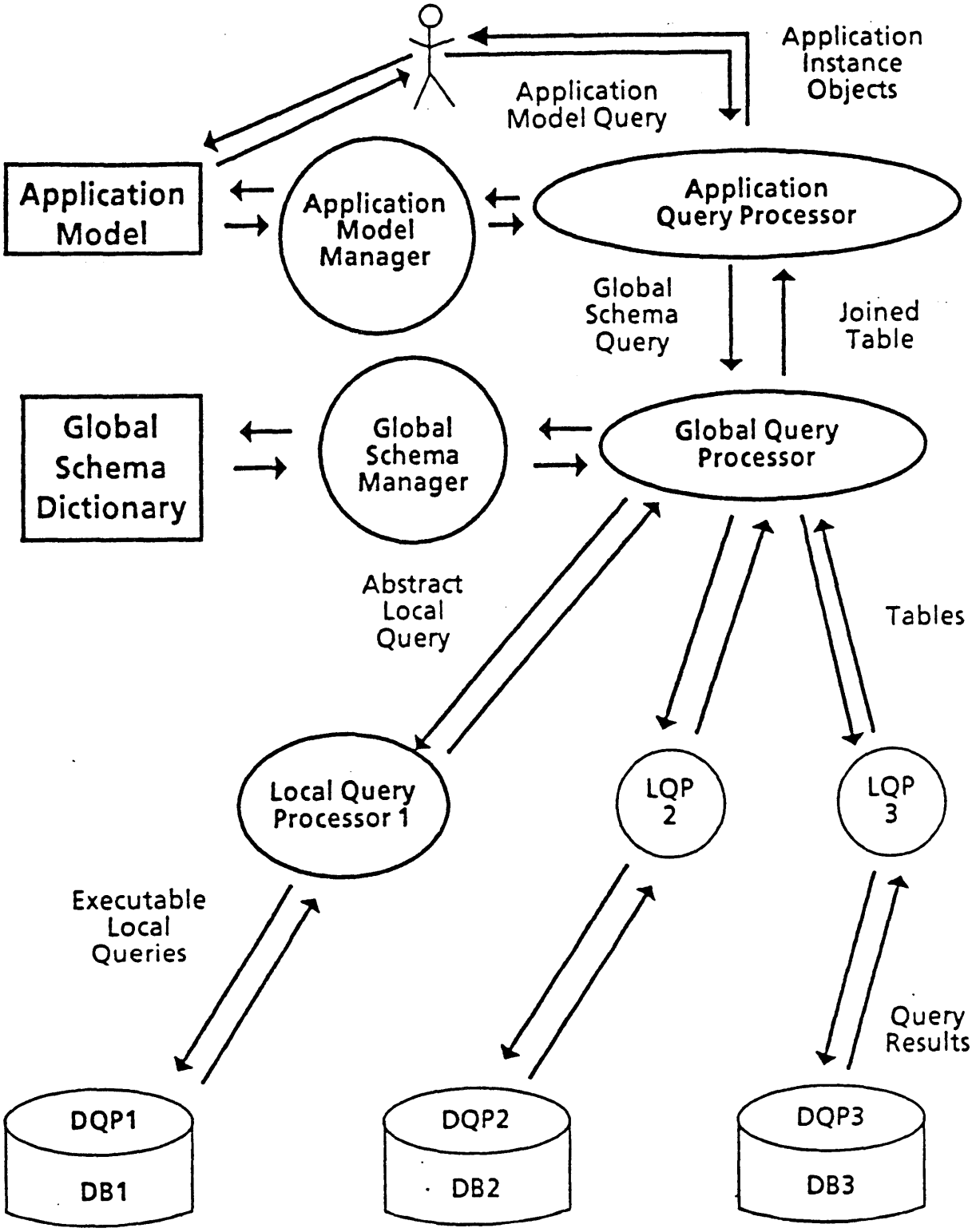

Figure 2-1: The CIS/TK Query Processor Architecture

The final sublist of the trio is called the condition list, which specifies how to filter the

information acquired based on the columns selected in the column list. The condition list is designed to work with boolean expressions, enabling compound searches to be requested in a logical format.

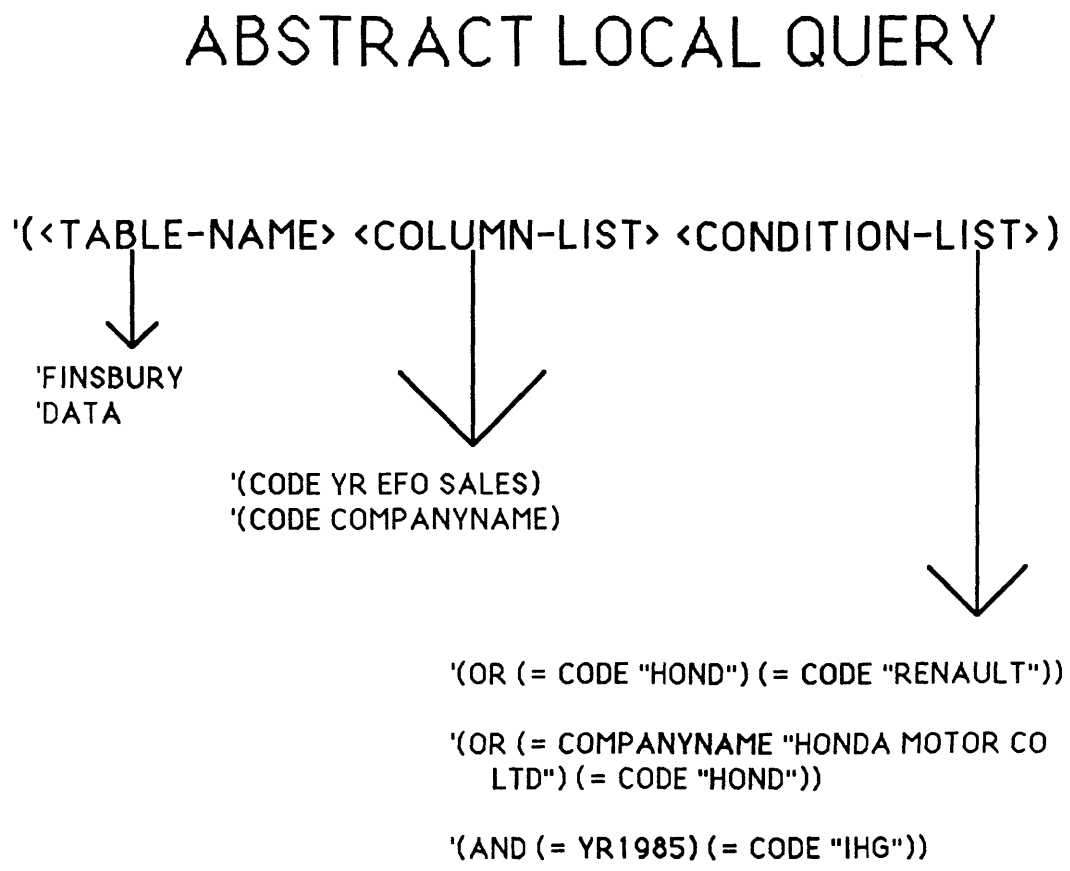

Figure 2-2: The Structure of the Abstract Local Query

2.4 Structure of the LQP

The Local Query Processor transforms an abstract local query received from the CIS Global Query Processor(GQP) into appropriate executable commands for a remote system and receives and transforms the results into a standard format required by the GQP. In order to conquer these tasks, the LQP has been designed for SQL type databases as a four module system, with modules for translating the given abstract local query, communicating with the

external system, filtering the information received and formatting the results properly(see figure 2-3).

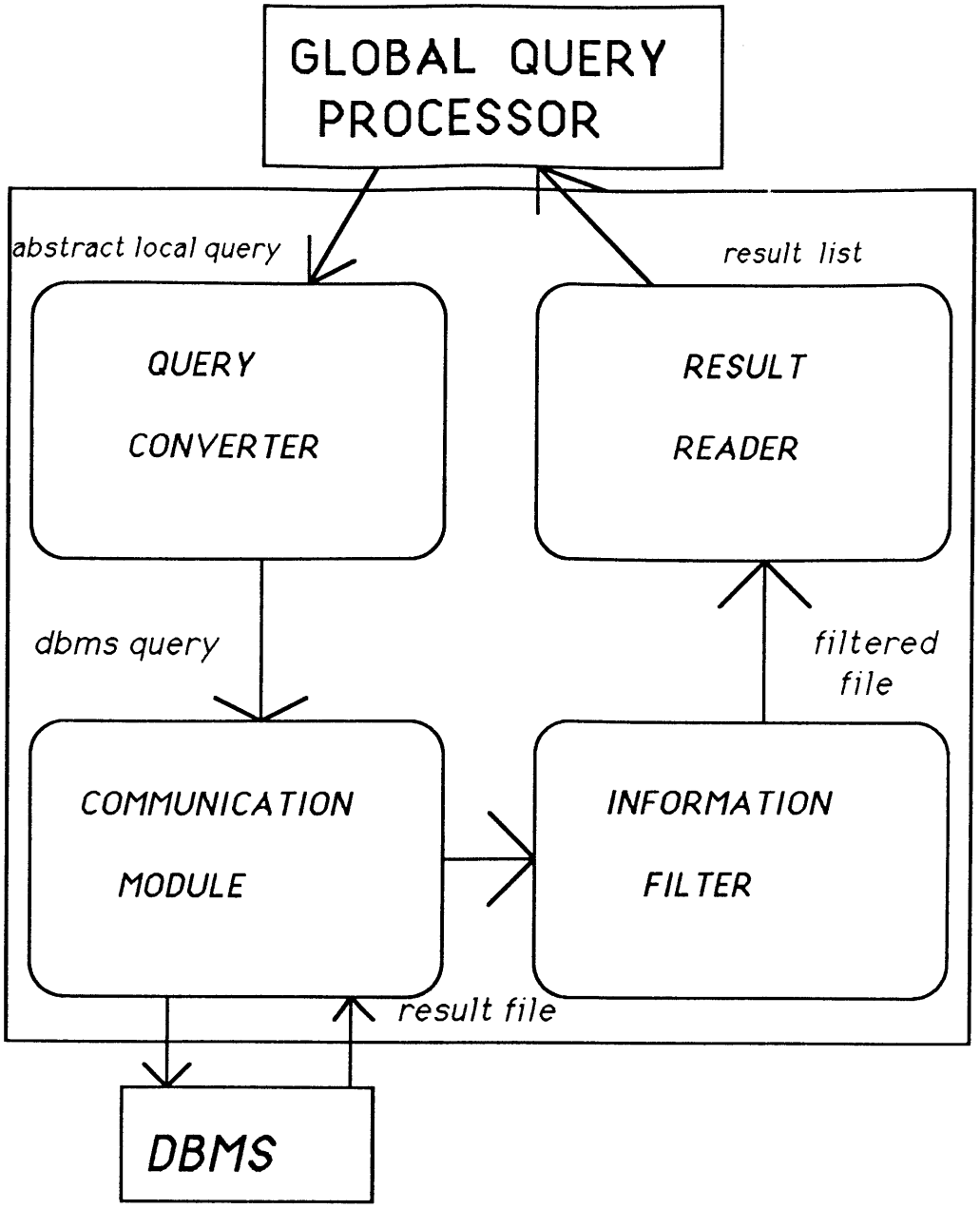

Figure 2-3: The Structure of the LQP

LQPs for any type of database system have to incorporate these four basic components, but not necessarily in the same general format. Unlike SQL type systems,

extracting information from menu driven systems only requires knowing the proper sequence of numbers or characters for the data needed. This enables the designer of a virtual driver to install just enough intelligence into programs to make educated and efficient selections about which menu choices are optimal. This intelligence can be in any form, as long as it ensures that at a minimum, all requested information will be returned to the local host. It is likely that a substantial amount of extra data will come along with that which is necessary, but this can be filtered from a file while off line. This effectively separates the problems of removing information from the foreign host from the specifics of the information desired. The *retrieval module* involves working with the foreign host on menu driven system and communications problems. Once the retrieval module has accurately completed its tasks, extracting the user requested information can be done on the local machine. This requires knowledge of the output format of the menu driven system, which is programmed into the *filtering module*. Refer to figure 2-4 for a diagram of the menu driven system LQP.

The Retrieval Module

-->*The Partial Condition List Parser* **scans through the condition list for any information which is necessary during the call to the menu driven system. Some systems may not need this section, depending on the setup of the abstract local query.**

-->*The Menu System Converter* **module produces the proper sequence of numbers or characters necessary to request the information from the external database. It is ideal to have this work exclusively from the column list, although it may not be possible with some systems.**

-->*The Communication Module* **is in charge of sending the sequence of commands generated by the abstract local query converter to the remote database and receiving a file containing the returned information ready for filtering. This module is generally implemented in shell.**

The Filtering Module

-->*The Full Condition List Parser* **recurses through the condition**

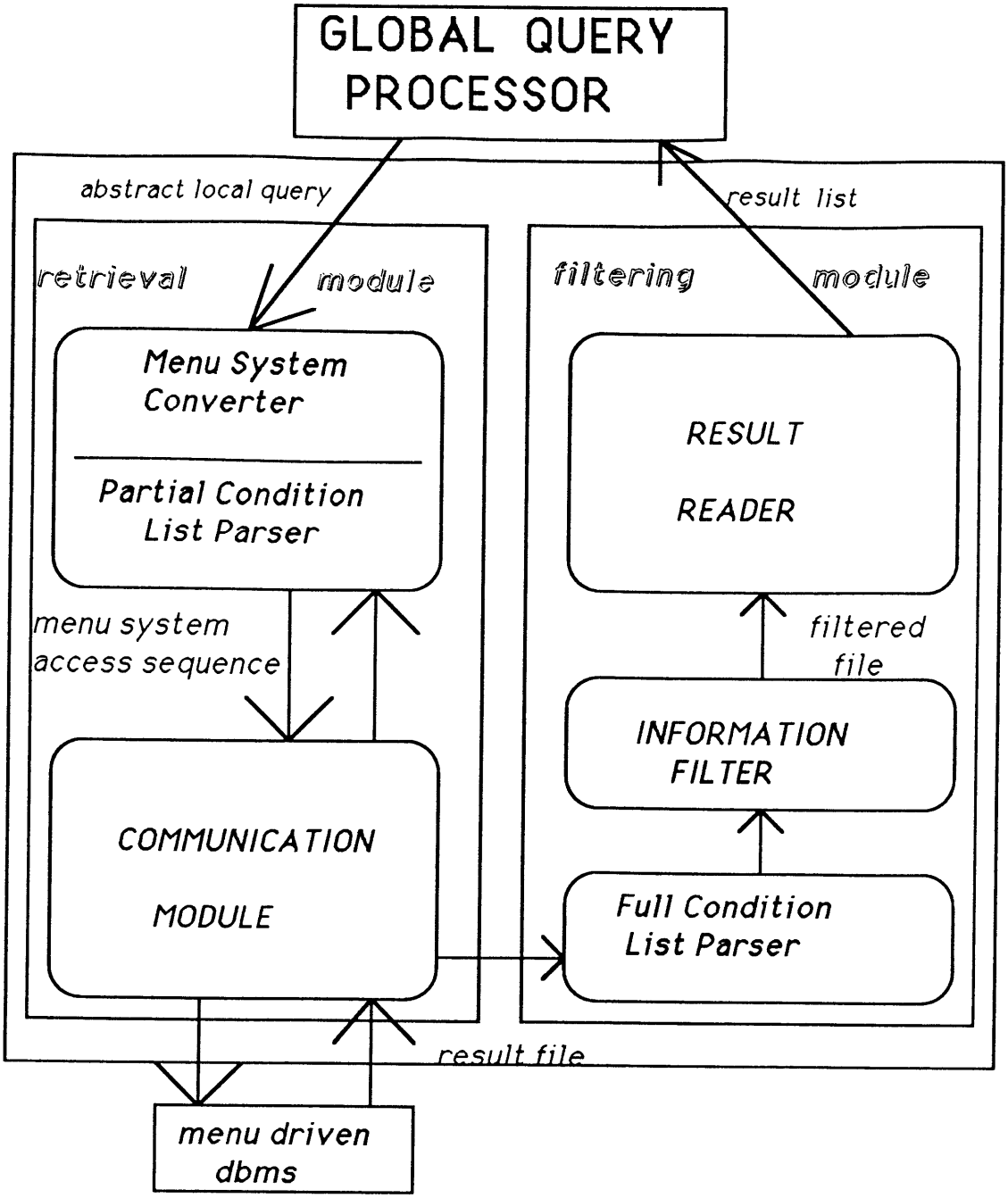

Figure 2-4: The Menu Driven System LQP

list to generate the proper terms for filtering the file received from the retrieval module.

-->*The Information Filter* **extracts the information actually needed by CIS/TK from the local file, based on the rules determined in the full parser.**

-->*The Result Reader* **is responsible for taking the filtered data and reading it back into the format used by the GQP.**

Chapter 3

THE IMPLEMENTATION OF DATALINE'S RETRIEVAL MODULE

The Retrieval Module is made up of the menu system converter, the partial condition list parser and the communication module. It's job is to use a minimum of information from the abstract local query to retrieve the minimum possible amount of data from the table being used. Thus, it is applied before the query is sent to the foreign host. A key for the success of the retrieval module is a simple design for mapping the information needed to specific menu selections.

3.1 Menu System Converter

The first step in determining the simplest method for working with a menu driven system involves examining the menu setups and output format. Dataline's menu system functions in a hierarchical sense, with progressive menus providing more and more complex data on the company in question. After a proper company code has been selected, two levels of menu choices are presented. The first level provides options for the type of data desired.

```
Options available are to display:
     1. Income statement
     2. Balance sheet
     3. Financing table
     4. Accounting ratios
     5. All four statements
```

It is possible that from an accountant's standpoint, every piece of data provided by the options above is equally valuable. However, most users only need specific information on a company, so a logical method for retrieving data from the system would revolve around a single option like the income statement or balance sheet. It is highly unlikely that any query to Dataline would require information from all four options, so making this feature available

in an LQP would be more expensive(through programming and run time) without much benefit. It is for this reason that option 5, "all four statements", is not supported.[4] Options 3 and 4 in the above menu are supported by the Dataline LQP, although they might not be used as much as options 1 and 2. It is necessary to include them, however, in case a manager needs a specific piece of information not available on the income statement or balance sheet.

The second level menu within Dataline enables a user to choose how much information about the previously selected option is to be returned. The following is its format:

```
Is the tabulation to be a:
     1. Summary
     2. Basic analysis
     3. Detailed analysis
```

Selecting the most detailed analysis for every scenario is possible, and it would certainly reduce some of the intelligence logic necessary for the Dataline LQP. The benefit of having this reduced logic(less up front programming, an almost unnoticeably faster logic code run-time) is more than offset, however, by the cost of the retrieval of the extra information. First of all, Reuters charges a certain amount of money for each data request, with prices increasing proportionally with the amount of information returned. A basic analysis and a detailed analysis cost three and eight times, respectively, what a summary does. A detailed analysis costs almost three times what a basic analysis does. This does not even factor in the costs of being attached via phone lines to the system or the extra connect time required to retrieve more information.

The forces driving a theoretical design of the retrieval module are thus apparent. A clever implementation of intelligence for this unit will provide substantial time(during data filtering in the filtering module) and financial savings to users, as this is where the most cost could be accrued.

---

[4]Even if all four options were required by a single query, the procedures implemented in this thesis would properly choose the most efficient menu selections for each of the options.

3.1.1 SEARCH-LIST

The key problem in selecting the proper menu options for Dataline involves taking a column specified and figuring out where it is located on the system. *SEARCH-LIST* is the list structure in lisp which was set up as Dataline's intelligence unit for this purpose. It contains enough information about the Dataline system to enable other procedures to efficiently and accurately select menu options, yet is designed in a very simplistic manner.

Search-list is implemented with four primary sublists, one for each of the options supported by Reuters. The sublists are placed in the same order within search-list as they are numerically available in Dataline, enabling the positioning of sublists to represent option numbers. These four sublists are broken down a step further in order to incorporate the tabulation selection. Three sublists thus exist for each primary sublist of search-list, which are also numerically related to Dataline's menu format. The first sublist stands for data being present in a summary, the second for data being present in a basic analysis and the third for anything only available in a detailed analysis(see figure 3-1).

Columns are placed in each of the primary sublists in which they can be found for reasons which will soon become obvious. They are only loaded into one subsublist, though, that which is of the least detail. For example, if a column is available in a summary, it will also be available in both basic and detailed analyses. This knowledge of the output format is built into the procedures which use search-list, so putting the column into each of the three subsublists would merely slow down the traversal of search-list's theoretical data tree.

Calculating the most efficient menu selections for Dataline is the direct result of a traversal of search-list. Each column specified in the column list is looked for in each sublist of search-list(each of which represents an option number). A list of two integer elements is returned for each sublist in which the column is present, the first specifying the sublist in which it was found, and the second representing which list(1-3, for the tabulation number) within that sublist(the subsublist). In actuality, the returned list is just a representation of

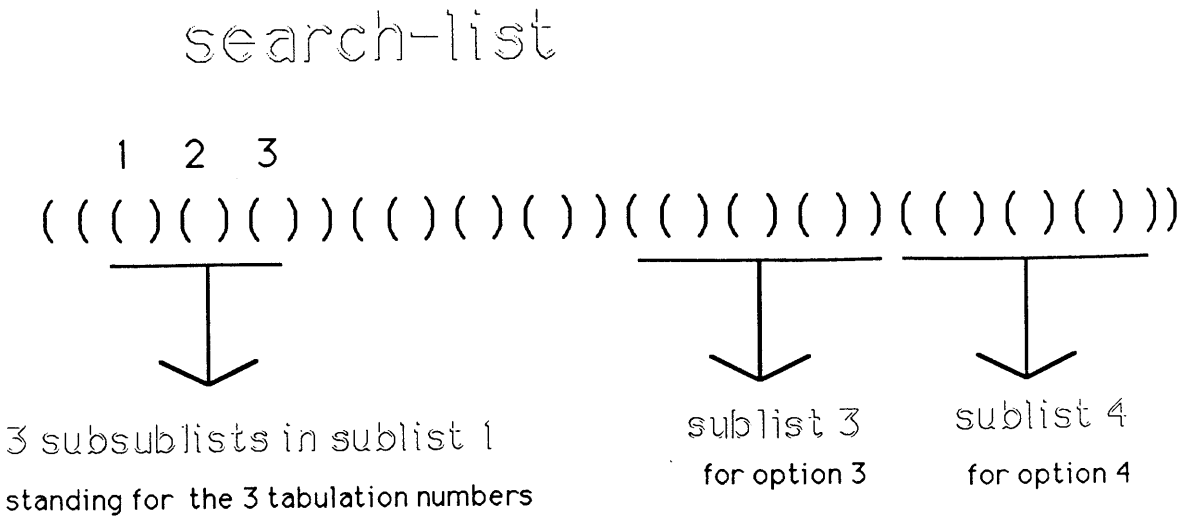

Figure 3-1: The Structure of Search-list

which numbers would have to be selected from Dataline's menu to retrieve the column's value with the least possible amount of detail(lowest tabulation number) selected. These lists are in the form (option-num, tabulation-num). When all of the possible combinations of option and tabulation numbers have been calculated for a specific column, a list of these two element lists is returned. This process is followed for each column in the column list, resulting in a series of lists of two element lists. Within figure 3-2, the results of this step can be seen directly below the "Form lists" heading, as lists of two element lists have been formed for currency and GFA.

At this point, one optimization feature is implemented. The number of occurrences of specific option numbers for all columns(no option number will be present twice for any column, since the least detailed tabulation is used) is summed, the maximally used one being the most profitable menu selection. If two or more option numbers are tied as the maximally used, the lowest number option will be selected.

The individual column representations are then scanned for any occurrences of two element lists without the maximally used option in a grouping with a list for the maximally used one. If they do exist, they are discarded, as the only remaining two element list for each column representation should be the most profitable selection. If a list of lists does not have the option which was used maximally, nothing will be deleted.[5]

The lists of two element lists still remaining are then appended together into one large list, setting the stage for the second optimization feature. This involves forming a list of two element lists with all option numbers still remaining represented, but only paired with the highest associated tabulation number remaining. This is because information available in a summary is necessarily available in a basic or detailed analysis, and retrieving more than one would be a waste. The list returned from this processing has the optimal menu selections for Dataline.[6]

A major benefit of the design of search-list is its flexibility. Since the whole scheme of menu selection is based on where columns are located in search-list, any changes in Dataline's supported columns or output format could easily be implemented in the LQP without modifications to any code.[7]

---

[5]It is possible, though not likely with this system, that subsequently performing this same technique on the second most used option(and then the third, etc.) would result in further optimization. Though this concept has not been implemented here, it may be of substantial value for other menu driven systems.

[6]Refer to figure 3-2 for an example of this process.

[7]A change in the format of Dataline's output may throw off the filtering routine since it scans for specific patterns.

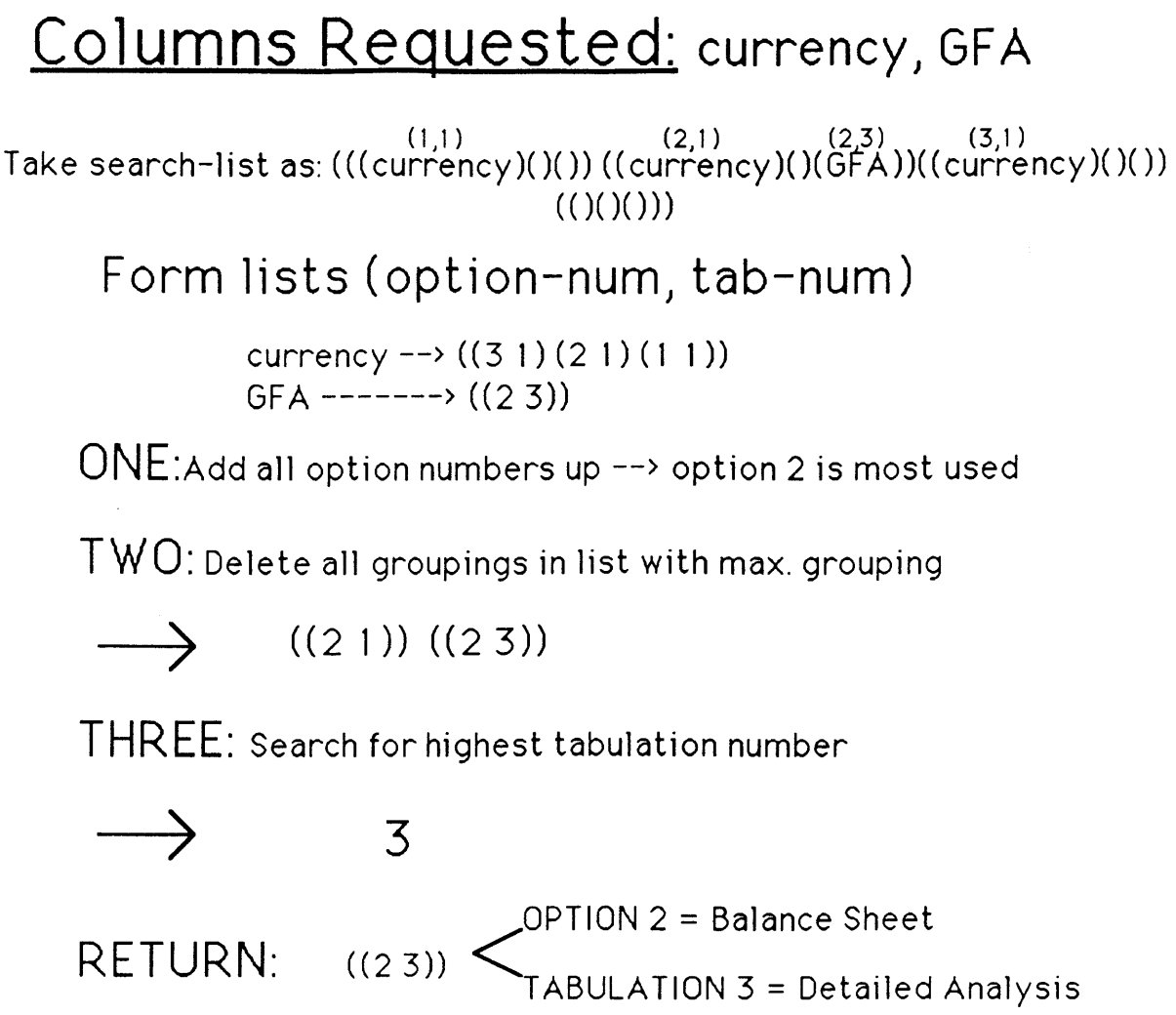

Figure 3-2: Calculating the Optimal Menu Selections

3.2 The Partial Condition List Parser

The concept of isolating the parsing of conditions from anything dealing with the retrieval module depends a lot on how a system and its LQP's abstract local query are designed. If a database system is in a reasonable format, it is a good idea to limit the menu system converter to working with the column list. In the case of Dataline, it is necessary to specify a valid company before being able to retrieve information on it. Since the condition

list is the only part of the abstract local query set up which receives the actual names and codes of companies to be queried by a CIS/TK user, these names must be retrieved prior to calling Reuters.

A routine in the retrieval module scans through the condition list and sets up a list of all company names and codes. All duplicate company names or codes are removed from these lists, as it is only necessary to test one for its validity(using the Dataline "names" option). Any invalid codes or names must be specified as "nil" values in the condition list, signifying an inability to process information on the company. If this process causes the condition list to have a "nil" value overall, Dataline should also not be accessed.

3.3 The Communication Module

Communicating with Dataline is performed by sending a series of unix shell commands through a pipe. The menu system converter and partial condition parser(implemented in lisp) determine what needs to be sent, and pass the values through procedure *unix-format* to shell code. The partial condition parser uses file **cx** to establish the proper name testing sequence, while the menu system converter sends the menu selections to file **cxmenu**. Values from both of these files are appended to **communicate**, which is the actual file sent to Reuters. Prior to the former two file's values being sent, **commtest**, which contains the access codes necessary for dialing up and entering Reuters services, is copied into **communicate**.

When the pipe to Reuters is closed, **tfile** contains the extracted information. **Tfile** is piped through **filter** in order to remove any "^M"'s, setting it up for the filtering module.

While the communication module is currently implemented by piping **communicate** to Reuters, this method does not have the capability of any on-line interaction. Thus, the LQP can not determine whether a name tested in Dataline's names routine is valid until after the pipe has been closed. Unfortunately, this means that the Dataline LQP can currently accept

only valid names. A future goal of this thesis is to support a new communication server developed by Francis Gan which would enable on-line interaction.[8]

[8] For further information, refer to [Gan 89].

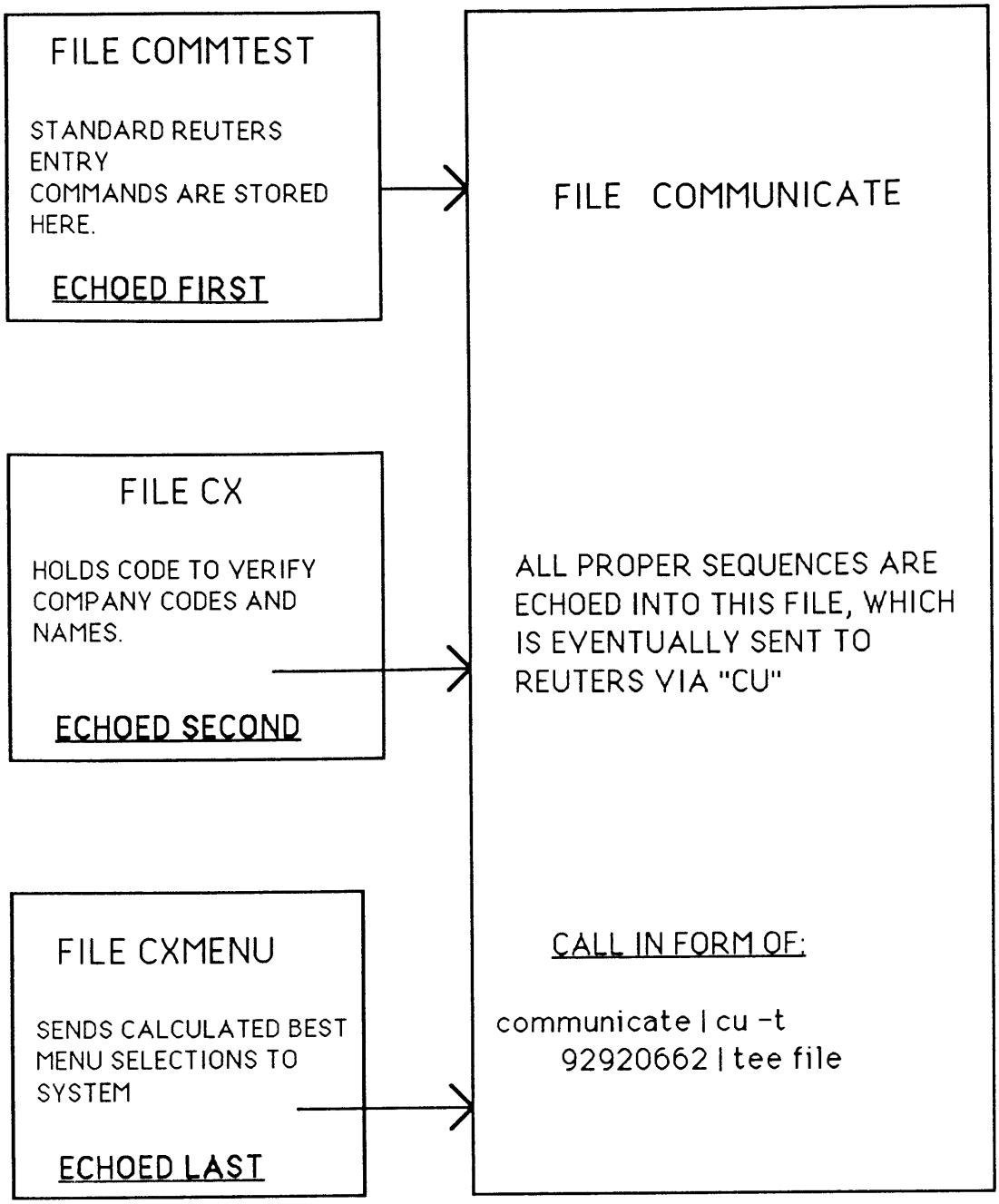

Figure 3-3: The Current Communication Implementation

Chapter 4

IMPLEMENTING DATALINE'S FILTERING MODULE

4.1 Overview

After the retrieval module has performed its tasks, **newtfile** exists on the local host, containing all the information extracted from the remote system. In the case of most database systems, but especially menu driven ones, there will be a lot of information returned which is not needed by the system's user. The key is to isolate the most important pieces of information and filter the local file accordingly. Some of the rules for filtering used by the CIS/TK system are taken directly from the condition list sent within the abstract local query, but other important decisions must be made with respect to the nature of the data returned. One such issue for Dataline is whether the column period ending or year should be returned under certain circumstances. Issues of this sort are tackled when the abstract local query is set up.

The filtering module is broken down into three sections: parsing the condition list, filtering the data, and reading the results back into a format suitable for the Global Query Processor. The parsed condition list provides the basis for filtering the data, and if the filtering is done cleverly, the result reader should be fairly easy to implement.

4.2 The Condition List

The main condition list parser would ordinarily be within the query converter module for an SQL type database system. This is because the conditions specified are used to form a query sent directly to the SQL system. While some parsing(extracting company codes and testing for at least one valid one) is done in the retrieval module for Dataline, the majority can be put off for the filtering module. For the Dataline model developed here, the parser is

considered a major part of the filtering module, though this theory's extendibility to other menu driven systems is dependent on the design of each system's abstract local query.

4.2.1 Design Goals

Virtually all information available on Dataline can be requested in the column list, making it is reasonable to keep the condition list as simple as possible. The only columns which have to be supported by the condition list are *code* and *company name*, so any others which are included would need to have substantial value to justify their addition.[9] This is the case with the *year* column. Users may want to focus in on a specific time frame rather than get information on a whole range of years. Implementing this column, which would work nicely in the existing boolean framework, will usually reduce the amount of data returned by the filtering routine. On the other hand, a column like earned for ordinary(efo) is not as generally useful(eg. (and (= code "HOND") (> efo 100000))). This example represents a specification for returning values for the company Honda in all years in which its *earned for ordinary* value is greater than 100,000. While this feature may be useful at times, being able to specify data retrieval terms based on year would on average help more users. Thus, the marginal benefit of the year column's being supported by the condition list is greater than that for efo. A similar argument can be applied to most other columns with respect year. Since adding the ability to support each extra column in the condition list increases complexity substantially, only the year column has been implemented in this thesis. It may be useful at some future point(or for users with specific needs), however, to support other columns in the condition list.

Once the decisions about which columns to support in the condition list have been made, the next step is to determine which are the most practical boolean operators. An

---

[9]Dataline only constrains on the column *code* (*company names* are translated into codes). As long as a code is properly selected, numerical menus specify any further data selection.

analysis of all possible boolean combinations of the columns which are to be supported should generally yield the practicality of the operators "AND" and "OR". Depending on the circumstances, operators like "NOT", "NAND", "XOR" and "NOR" may also be useful.

4.2.2 Implementation

Dataline's condition list is implemented for the columns *year*, *code* and *company-name* with the boolean operators "AND" and "OR", as shown in the table below. The operator "NOT" is not currently supported, though it may be useful to add it in conjunction with the *year* column at a later time.

YEAR	CODE	COMPANY-NAME
and	*and*	*and*
or	*or*	*or*
not?	-	-

Although *company-name* is supported by the condition list, all names are converted into their respective codes for ease of processing. If a given code is not valid, the value "nil" is replaced in its spot in the condition list. It is thus possible that an entire condition list could be reduced to a "nil" value, and the call to Reuters would not be necessary.

SUMMARY OF CONDITION LIST IMPLEMENTATION

1. CODE and COMPANY-NAME columns supported
   Required by Dataline
2. YEAR column supported
   Likelihood that user would want information on a certain time period
3. Other columns not supported
   Filtering potential is not enough to justify implementation
4. AND and OR boolean operators employed
   Other operators are not as useful as these, though NOT may be good to implement for the YEAR column

Once a design for the condition list has been laid out, it is necessary to analyze every possible scenario that could be encountered during parsing and how to deal with them. During the development of the Dataline LQP, a number of methods were considered. It was determined that the easiest structure to work with would be a nested list, in which the outer parenthesis enclose the entire list of values, the next inner parenthesis enclose each company/year combination requested and the most inner parenthesis enclose the respective second list's company code. The yearlist is positioned within the first parenthesis after the list of the company code.

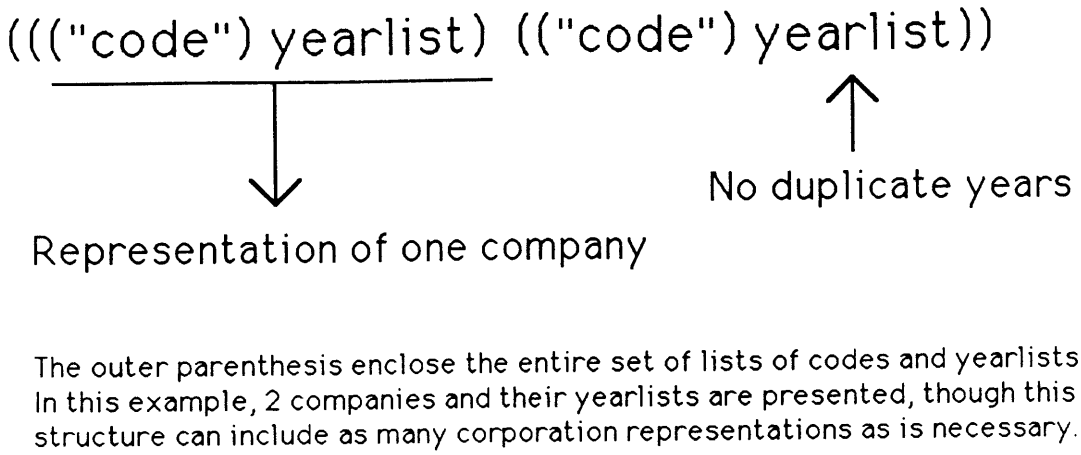

Figure 4-1: The Structure Used to Resolve the Condition List

4.2.2.1 Low-level Condition Parsing

The following is a run down on scenarios encountered during parsing Dataline's condition list:

Case 1: OR'ing two companies

Any two codes or company names can be ORed together. Data is being requested for either

or both companies. If the same company is specified twice, it is only represented once since both codes request the same information.

```
eg. (OR (= code "HOND") (= code "RNLTL"))
                        ==>((("HOND")) (("RNLTL")))

    (OR (= code "HOND") (= companyname "Honda Motor Co"))
                ==>((("HOND")) (("HOND"))) ==>((("HOND")))
```

Case 2: OR'ing years or codes and years

The only way in which a year can be ORed is if it is combined with another year. Dataline requires having a company code to process a request, so asking for all information with year = 1986(which would be specified by the format "CODE OR YR") would not be practical. In essence, the literal meaning of this boolean expression would necessitate returning information on *every* company during the year specified.

ORing two years together establishes a yearlist, which subsequently will have to be ANDed with code/name for it to have any practical value.

```
eg. (OR (= yr 1985) (= yr 1986)) ==>((NIL 1985 1986))

    (OR (= yr 1987) (= code "HOND")) ==>ERROR
```

Case 3: ANDing two codes/names

Although it may seem logical to be able to request data for "Honda AND Renault", from a boolean standpoint it is not reasonable. Thus, the only way that two codes/names can be ANDed without a nil value being generated is if the codes/names stand for the same firm during the same time period(AND signifies that everything is identical).

```
eg.(AND (= code "HOND") (= companyname "Honda Motor Co"))
                                    ==>((("HOND")))

  (AND (= code "HOND") (= code "RNLTL")) ==>ERROR
```

Case 4: ANDing a code/name with a year

It is logically valid and actually quite beneficial to AND a code/name with a year or a list of years(generated by ORing the years together). This is the primary way to use the year filtering feature to request a more focused range of data.

```
eg. (AND (= code "HOND") (= yr 1986)) ==>((("HOND") 1986))

    (AND (= code "HOND") (OR (= yr 1986) (= yr 1985)))
                              ==>((("HOND") 1986 1985))
```

Case 5: ANDing years together

Two years can not be ANDed together unless they are identical.

```
eg. (AND (= yr 1986) (= yr 1985)) ==>ERROR
    (AND (= yr 1986) (= yr 1986)) ==>((NIL 1986))
```

4.2.2.2 Upper-level Condition Parsing

These list structures developed during parsing build on themselves as more and more conditions are applied. This is where the true value of this scheme comes into play. It is designed in such a way that LISP code can easily manipulate the structures as necessary during a recursion through all the boolean conditions. This ability to handle nested recursions should be built into condition list parsers for all LQPs. The system memory limitation for nesting, around nine levels deep, should not be a factor except in the unlikely case that the conditions on a single company are *extremely* specific.

A compound list is one of the structures set up in the lower-level parser which can be combined with other compound lists or lower-level expressions. The following are some of the possible parsing scenarios encountered when at least one element is a compound list. A shortened form is used for demonstration purposes.

Case 1: ANDing a compound list with a code or a year

```
eg. ((("HOND"))) AND (= code "RNLTL") ==>ERROR
    ((("HOND"))) AND (= code "HOND") ==>((("HOND")))
    ((("HOND"))) AND (= yr 1982) ==>((("HOND") 1982))
```

```
Case 2: ORing a compound list with a code or a year

  eg. ((("HOND"))) OR (= code "RNLTL")
                        ==>((("HOND")) (("RNLTL")))
      ((("HOND"))) OR (= yr 1983) ==>ERROR
      ((NIL 1983)) OR (= yr 1984) ==>((NIL 1984 1983))


Case 3: ANDing two compound lists

  eg. ((("HOND") 1983)) AND ((("HOND") 1984)) ==>ERROR
      ((("HOND") 1983)) AND ((("HOND") 1983))
                              ==>((("HOND") 1983))
      ((("HOND") 1983)) AND ((NIL 1983 1984))
                              ==>((("HOND") 1983 1984))


Case 4: ORing two compound lists

  eg. ((("HOND") 1983)) OR ((("HOND") 1984))
            ==>((("HOND") 1983) (("HOND") 1984))
                  ==>((("HOND") 1983 1984))
      ((("HOND") 1983)) OR ((("RNLTL") 1982))
                  ==>((("HOND") 1983) (("RNLTL") 1982))
      ((("HOND") 1982)) OR ((NIL 1983 1984)) ==>ERROR
      ((NIL 1982 1983)) OR ((NIL 1983 1984))
                  ==>((NIL 1982 1983 1984))
```

4.3 Filtering the Data

Selecting the proper method for filtering data requires a careful analysis of the type of data being received and the format it is in. Filtering routines generally incorporate string searches of one type or another, so a language like lisp, which is weak in this aspect, is virtually useless. This is a problem for the CIS/TK system(which is mostly implemented in lisp) since a new language must be incorporated, and the proper variable passing between languages must be worked out.

Reuters' Dataline database system produces information on companys in a tabular form. Some general company information is at the top of each report(eg. Name, country,

currency, etc.), and the financial data follows in a standard format. Since reports are in a standard format using standard keywords, the AWK language within shell was chosen for implementing Dataline's filter. AWK works on the basis of pattern-action statements, searching for a standard pattern(or keyword) and performing the associated action when it is found. AWK is also programmable, supporting many "C" style commands. This provides great flexibility, and enables one to make use of some good string processing features.

4.3.1 Programming with AWK

AWk is best at processing data which is a mixture of words and numbers separated into fields by blanks and tabs. The output of Dataline is of this form, making AWK particularly valuable in this instance. AWK provides programmable field separators(eg. blank, tab), which mark how to break up the columns of data in a record. Once the fields have been distinguished, the different columns of data can be accessed by merely using a dollar sign followed immediately by the number of the desired column, as in $2 for the data in the second column. $0 stands for the request of an entire record. Thus, this scheme enables one to do a quick calculation of a large column of data(through summing certain columns, etc).

Being able to reference specific columns of data in this format creates an opportunity to exploit AWK's pattern statement, as a specific column can be tested to see if it matches(~) a given pattern(/pattern/). For example, if a file of data is known to have the phrase "Mets" in a record(one line) with the user's desired data in the record following that one, a pattern $0 ~ /Mets/ is specified to find the string. Subsequently, a jump to the next record can be made. Dataline, as another example, returns actual column names in its output, which serve as fine patterns to attempt to match.

AWK provides a number of string functions which simplify the processing of data(performing an action after a pattern has been matched). These are similar to "C"

language string functions, but are more geared towards working with columns in the discussed format. The more general functions include *length* and *substring*, available with most string packages. The useful new functions include *sub*, *gsub* and *split*. Sub and gsub substitute a parameter they are passed for a specified string in a record, either on the first finding of the string(sub) or globally in the current record(gsub). Split splits a passed value(eg. a record) into an array, based on either the field separator or a given parameter. This function was useful, for example, in breaking up the date term(of the form xx-xx-xx) in Dataline, by specifying "-" as the value separator.

AWK supports a variety of control flow statements, many of which are similar to "C" type looping commands. *If-else*, *while*, *do-while* and *for* statements are some of the standard commands available. A couple of other functions also prove very useful, namely *break* and *next*. The break statement causes an immediate exit from an enclosing while or for loop. The next statement causes AWK to skip immediately to the next record and begin matching patterns from the first pattern-action statement. This is very useful when a record has matched a specified pattern and processing should then be done in subsequent records.

AWK also has some SHELL-like capabilities. It can interact reasonably well with both pipes and files. The function most useful in the work on Dataline was the ability to print output into files instead of to the standard output. This is performed within an action statement by using the > and >> redirection operators, where > overwrites original file contents and >> appends. For example, print $3 > "bigpop" writes what is in the current record's third field into file bigpop. The name of a file can be constructed(eg. to attach a path name) by enclosing the juxtaposed name terms in parenthesis(eg. print $3 > (path "bigpop")).

4.3.2 Filtering Dataline with AWK

AWK programs read through given files trying to match every pattern specified to each line in sequence. Even if AWK has matched one pattern to a line, it will still attempt to

match the remaining ones. It is for this reason that complex programming in AWK(trying to match more than one type of pattern in a file) should be done with states. A state diagram of Dataline's filter is shown in figure 4-2. It can be seen that typical states for Dataline are before any data of value has been found, after a long string of -'s, and while looking for a valid company name. In an AWK program, the state is kept as a variable which is ANDed with some pattern statements, enabling the state to turn on or off the ability to match a pattern(eg. $0 ~ /Period ending/ && (STATE == 1)). Once the scheme for finding patterns in a standard output has been established, the processing of the data can be done with AWK's comprehensive string package.

Filter2.nawk is the main filtering program for Dataline, actually implemented in the NAWK language.[10] It scans through the file set up by the call to Dataline, looking first for a series of -'s(#1 in figure 4-3). This signifies the start of a new output record. The company code is then tested, to see if the current record is the proper one(#2). If it is, relevant values are saved(eg. years), and the program continues on to see if the desired column is in this specific record(#3). If a column is not found, *filter2.nawk* will scan for a new line of -'s, thus overlooking any other garbage in the file it is reading. Obviously, there may be more than one record for a company on a given use of the retrieval module. This routine is repeated until the desired column is found.

It must be emphasized that a program like AWK is most useful when the format of a file or incoming data is in a known tabular form. If the incoming information were arbitrary, one would not know what specific patterns to look for, thus defeating AWK's purpose.

---

[10]NAWK is very similar to AWK, though it has a few extra features. The most important one used here was the ability to work with procedures. The GAWK language is also available.

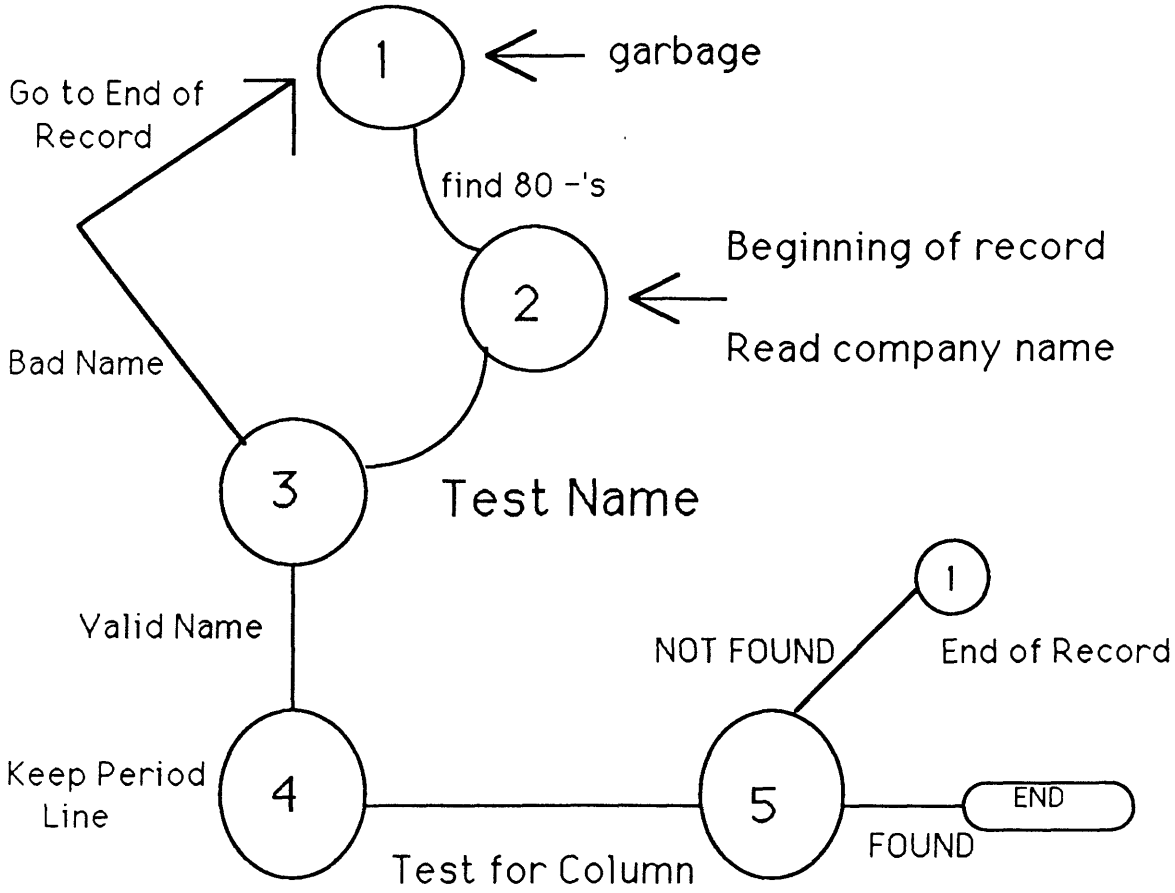

Figure 4-2: Dataline Filter's State Diagram

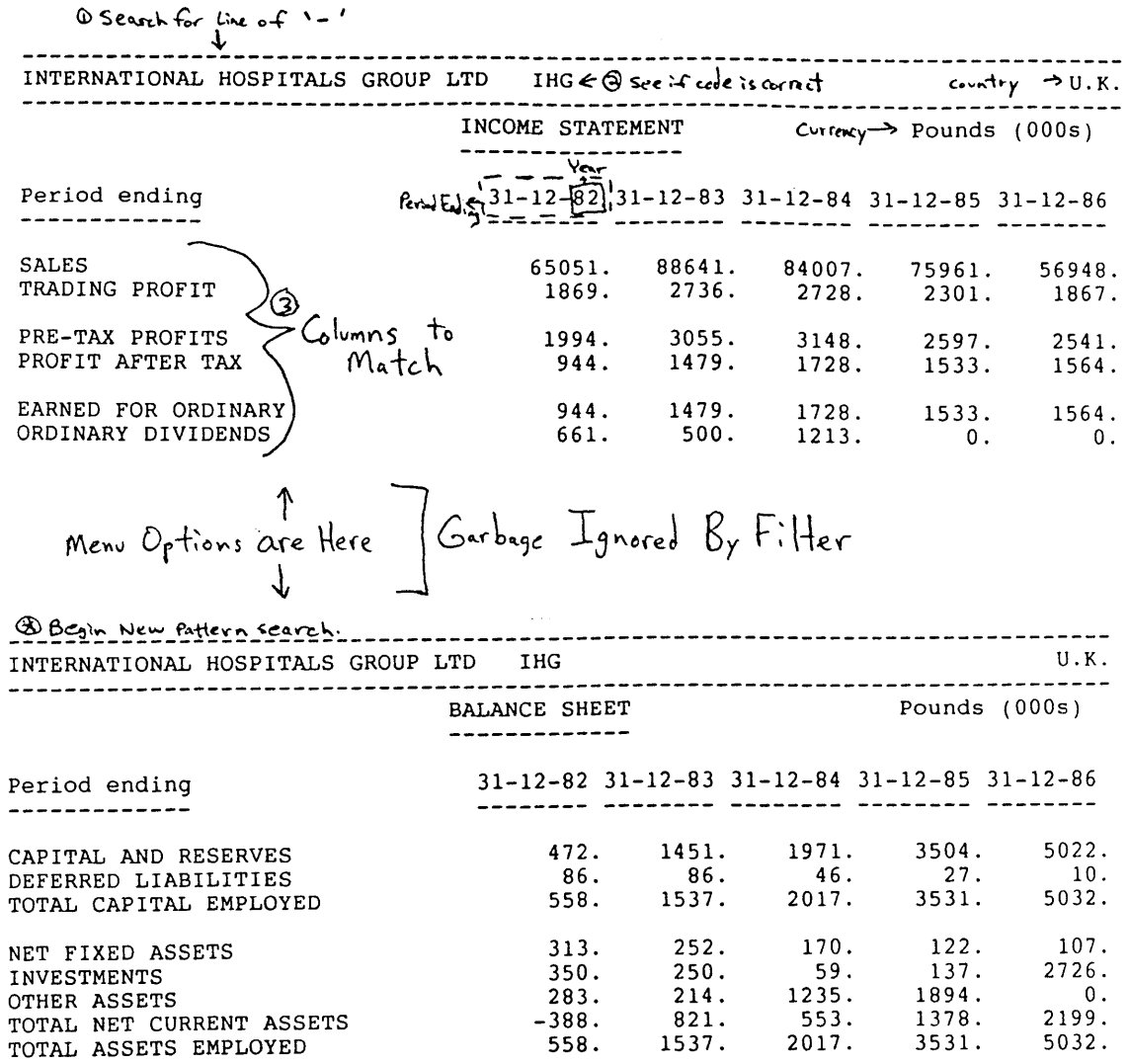

Figure 4-3: Searching Dataline's Output With AWK

4.4 Reading the Results

The result reader takes data from a file set up by the information filter, changes it into the format accepted by the GQP, and subsequently returns it. It has been implemented for the Dataline LQP with both the AWK program *readformat.awk*, which groups the data into a format suitable for easy reading into lisp, and the lisp routine *read.lsp*, which actually performs this reading. The output from the information filter(which was left in file *readfile*) was organized with Column, New Company and EOF flags in order to provide a set of patterns easily recognized by *readformat.awk*. *Readformat.awk* establishes *FINALFILE*, from which the routine in *read.lsp* can directly read the values back into LISP. Performing this function with only LISP code would have been rather difficult, so the AWK programming has been used to remove this burden.

Chapter 5

Extension to Another Menu Driven System

5.1 Overview

The LQP model developed in this thesis has been designed for extension of its basic issues to a broad range of menu driven systems. The key goal driving this model is to minimize system cost and maximize potential usefulness by determining which features of a system should be incorporated into an LQP and how flexible the data specifying conditions should be made. These concepts are implemented through the design of the abstract local query, the structure and contents of the LQP's intelligence unit and the nature of the filtering routine. Once the aforementioned have been sorted out, the more general problems like communication and programming specifics can be tackled.

One problem of all LQPs is developing the communications format necessary to drive a database. For example, it is necessary to send a "/r/c" at the end of each line piped to Reuters, as this system needs a return without a line feed. One must also figure out how to gain entry into a system in a preprogrammed way, echoing shell scripts through a pipe in the proper format. Since communication problems for the Reuters system providing both Dataline and Textline were solved during the development of the Dataline LQP, Textline is a good model for extending many theoretical concepts not related to communication.

5.2 Reuters Textline

Reuters Textline provides important news on a wide range of companies, industries and products. Since it is logical to assume that a user may at some point need information on any of these topics, restraint is necessary before excluding anything supported by Textline from an LQP. The following theoretical design incorporates most of Textline's options in a

format consistent with the generic LQP developed in chapters two through four. The three major design issues are covered, leaving the implementation and other general LQP topics for future work.

5.2.1 Intelligence Required

An analysis of Textline revealed that three of its features need to be represented as intelligence in an LQP: the names of the geographic and industry databases available, the indexing codes and the general codes for increased precision which are presented in Textline's full search facility. These are the three sets of specifications that a search can be constructed of. In order to use Textline's built in searching capabilities most efficiently, it is ideal to narrow down the amount of data to be scanned as much as possible. All these specifications must be supported, so that whenever possible, any applicable item can be used to increase searching precision.

It can be argued that since these options are available interactively on Textline, an LQP which scans them on line could be developed. This would reduce the amount of memory taken up on the local system(through stored intelligence) and guarantee that a database or code is currently supported by Textline(eg. It is likely that Reuters will add or delete specific indexing codes at times. Thus, a fixed set of codes loaded into an LQP is not necessarily accurate at a given time.). These factors are far overshadowed by the benefits of establishing local intelligence. Most importantly, it is very expensive to be connected to a Reuters system. By bringing relevant information onto the local system, the proper searching format can be determined while costs are not being accrued. Second, an LQP operates in a preprogrammed sense, so in order to select databases or indexing codes on line, there must be interaction between Textline and the LQP.[11] Although the new communication server developed by Francis Gan should enable interaction, it has not been properly tested yet.

---

[11]This is the same type of problem encountered in Dataline's *Names* option.

5.2.2 The Abstract Local Query

The two key parts of the abstract local query which have to be designed are the column list and the condition list. The column list should support only items general to *any* type of query to a database system, while the condition list must support specific requests(i.e. the necessary input to the system, like a code in Dataline). Since the constraining part of this design is the condition list, it should be laid out first.

Textline needs to receive a word/phrase or indexing code in order to have a specific condition to search on.[12] The condition list has been designed for menu driven system LQPs to hold this system required input, which basically specifies the current company/topic to deal with. Thus, the condition list should support the column *PHRASE*, which specifies what term(s) to look for based on what has been selected in the column list. The condition list must also support those items which have substantial value when used in a boolean combination with the system required term(and can not generally stand alone in the column list). In the case of Textline, both the indexing codes and the general codes from the full search facility should be supported. This design will enable a user to specify a term to look for and directly narrow down the search's range. For example, *PHRASE* "ALASKA" can be ANDed with the code for a company, so that articles on companies like Alaska Airlines would be returned without topics related only to the state, like the Exxon Oil spill(eg. (and (= phrase "ALASKA") (= companyname "Alaska Airlines"))).[13]

The design of the column list must be considered next. The only source of intelligence not supported in the condition list, the search narrowing databases, can be represented here. The names of the available databases can displayed by a get-columns request, enabling a user to make reasonable selections for narrowing down geographical and industry ranges for

---

[12]If only databases were selected, Textline would not know which articles from this domain to return.

[13]It must be noted that the ideas presented here provide just one possible method of attack.

searching. Using these databases in the column list provides great flexibility in the event that more than one phrase is desired in a particular query. Assume that two different topics, cellular radio and the New York Mets, were specified as phrases. Databases related to either of these phrases could be selected in the column list without adding too much overlap searching. The slight loss in efficiency is more than compensated for by the ability to make a single call to Textline for multiple *PHRASES* within a particular query.

5.2.3 Filtering Textline's Output

Textline's output presents a major problem for the filtering module of an LQP. Systems like Dataline retrieve records with specific values and their related headings, like sales of a number of dollars. This type of format is ideal for a filtering program like AWK, which can search for a specific string of text which is always available under a given menu option. Textline, however, merely returns the articles which matched the search conditions it was sent. These articles have no specific format, as what is included is only relative to what the article is about. Thus, only three options for filtering this output file seem sensible.

The first option involves just returning an entire article to the user. While this guarantees not missing any relevant information within that specific article, it can be a burden to a user who is looking for a specific piece of data but matched a large number of articles. Second, a short summary can be removed from Textline(or generated somehow from each retrieved article), which would enable the user to quickly determine if the article were truly of value to him. The last option available is to scan the retrieved articles for an input string of some sort, though this search would probably be performed better directly by Textline.

Chapter 6

Conclusion

6.1 Summary

The importance of integrating multiple, disparate database systems is increasing rapidly with the growth and globalization of large firms. The CIS/TK system at MIT's Sloan School seeks to overcome the integration problems by providing a modular framework suitable for implementing Local Query Processors without making many changes to existing systems. Thus, one goal for the CIS/TK group may be to make the LQP's design simple enough that a new one could be implemented in a day.

Many of the important LQP issues were discussed by Alec Champlin in the context of SQL type DBMSs [Champlin 88]. The goal of this thesis has been to extend his findings to a different type of DBMS format, those using menus. Working with Reuters Dataline and Textline systems, a theoretical model of an LQP for a menu driven system has been developed. Many concepts have remained similar to those for an SQL DBMS, though research has yielded two important features exclusive to menu driven systems. First of all, there is a necessity to install some degree of intelligence about a DBMS in an LQP in order to make educated menu selections. This enables the LQP to remove the proper information from the remote system without doing any filtering or any analyzing of the condition list, effectively separating a *retrieval* module and a *filtering* module. Second, a powerful filtering program is needed in order to process the data retrieved. AWK, developed at Bell Labs, is a valuable tool when the output representation of data is known.

There are many types of DBMSs available today, of which menu driven systems are by far the most used. Thus, the model developed here should serve as a major step towards reaching the ultimate CIS/TK goal of being able to communicate with ALL DBMSs.

6.2 Future Work

One major problem encountered in the work on Dataline involves getting interaction between Dataline and the LQP driving it. In order to determine if a name or code in the condition list is valid, it has to be tested within Dataline's 'Names' program. The results of this test, which must be retrieved prior to continuing a query, are currently unaccessible until the pipe to Reuters is closed. This may be overcome through the LQPs integration with Francis Gan's communication server. One other possible solution involves teeing the call to Reuters into two files and closing one of the files after the 'Names' routine, thus making it readable. This method may be practical for Dataline(since there is only one situation requiring interaction), but would present trouble with other systems in which more interaction is necessary. It is not practical to tee a large number of files in order to close the pipes to them sequentially(as each bit of interaction is needed).

Once the problem with interaction has been sorted out, changes to the existing code will be necessary. Most importantly, since Dataline works primarily with company codes, the LQP must be able to test the company names in a query's condition list and convert the valid ones into codes. Codes must be replaced for valid company names in order to both request information from Dataline and parse the condition list. Thus, the current implementation of Dataline's LQP will not function correctly if it is sent company names through a condition list.

This thesis provides a framework for tackling the problems associated with generating LQPs for menu driven type databases. The process of extending the model developed in this thesis can begin with an implementation of Reuters Textline service. The design laid out in chapter five should act as a foundation for the work, though more specific problems are certain to become apparent as the implementation is carried out. The concepts in this research have been developed with an eye towards application to any type of menu driven system, although the work has generally focused on systems similar to Reuter's Dataline. Therefore,

Textline, which is of a substantially different format from Dataline, will present its own unique problems. It is likely that adding any new LQP will present some new types of problems, though these problems should be able to be dealt with within the framework established here. More generally, the theories developed in this thesis must be extended to a wide variety of menu driven systems in order to determine an overall generalization of this LQP mechanism.

Chapter 7

Demo Run

```
>(print-frame 'finsbury)

FINSBURY:
    MACHINE-NAME:
        (VALUE foreign)
    TYPE-OF-DBMS:
        (VALUE menu)
    LOCAL-DBMS?:
        (VALUE NIL)
    DATABASE-DIRECTORY:
        (DEFAULT IRRELEVANT)
    DATABASE:
        (DEFAULT datalinedb)
    LQP-COMMON-DIRECTORY:
        (DEFAULT /usr/cistk/demo/v2/lqp)
        (VALUE /USR/CISTK/DEMO/V2/LQP/FINSBURY)
    LQP-SPECIFIC-DIRECTORY:
        (DEFAULT /usr/cistk/demo/v2/lqp/finsbury)
        (VALUE /USR/CISTK/DEMO/V2/LQP/FINSBURY)
    COMM-SERVER-DIRECTORY:
        (DEFAULT /usr/cistk/demo/v2/lqp/dev)
        (VALUE /USR/CISTK/DEMO/V2/LQP/DEV)
    COMMUNICATIONS-SCRIPT:
        (VALUE communicate)
    EFFICIENT-COMM-SCRIPT:
        (VALUE irrelevant)
    PHONE-NUMBER:
        (DEFAULT 92920662)
    ACCOUNT:
        (DEFAULT c202430)
    PASSWORD:
        (DEFAULT rna *325250)
    METHODS:
        (MULTIPLE-VALUE-F T)
        (VALUE (SELF-INFO DISPLAY-DATALINE-SELF-INFO)
               (GET-TABLES GET-DATALINE-TABLES)
               (GET-COLUMNS GET-DATALINE-COLUMNS)
               (GET-DATA GET-DATALINE-DATA))

NIL

>(print-frame 'datalinedb)

DATALINEDB:
    SUPERIORS:
        (MULTIPLE-VALUE-F T)
        (VALUE INFORMIX-2C)
    INSTANCE-OF:
        (MULTIPLE-VALUE-F T)
        (VALUE INFORMIX-2C)
    DATABASE:
        (VALUE dataline)
    DATABASE-DIRECTORY:
        (VALUE /usr/pagetm/cis)

NIL

>(send-message 'finsbury :self-info)

>Do you want to see a description of the Dataline system?  (Y or N)
```

REUTERS DATALINE

Reuters' Dataline is a menu driven system which provides information on a wide range of corporations, concentrating on those in the United Kingdom. The data available includes what is on typical, condensed balance sheets, income statements, financing tables and accounting ratio summaries.

Dataline is a menu-driven database system, so it does not operate with tables as the standard SQL database would. Currently, the table name location within the abstract local query call is being called 'data or 'finsbury, although it does not serve a purpose. Perhaps it will eventually in distinguishing between Dataline and Textline, for example.
Hit return to continue.

```
 ->(send-message 'finsbury :get-tables)
(("data"))

 ->(send-message 'finsbury :get-columns)
(("Column name") ("CODE") ("COMPANYNAME") ("YR") ("CURRENCY")
 ("COUNTRY") ("SALES") ("TOT-SALES") ("EFO") ("TOT-CURR-ASSETS")
 ("TOT-ASSETS-EMP") ("NET-FIXED-ASSETS") ("TOT-STOCK")
 ("CURR-LIABILITIES") ("TOT-DEF-LIABILITY") ("CAPITAL-RESERVES")
 ("ADJ-EARN-SHARE") ("ROSE") ("MIN-INTEREST") ("TOTTAX"))
 ->(send-message 'finsbury :get-data '(data (code sales) (and (= code "rnltl") (
= yr 1987))))

Your selection is being formatted for Reuters...
converting to string
converting to string

("rnltl")
("rnltl")
T
0
("rnltl")
NIL
("CODE" "SALES")
"curr-column-list"
(CODE SALES)
"curr-column-list"
(SALES)
"curr-column-list"
NIL

TELENET
617 113E

TERMINAL=

@c 202430


202 430 CONNECTED
```

```
>hello rna132*325250

REUTER TEXTLINE
---------------

1.  TEXTLINE
    Articles for the period 1-1-80 to 19-5-89

2.  NEWSLINE
    News headlines for the period 13-5-89 to 19-5-89

3.  ACCOUNTLINE & DATALINE
    Company accounts - published and standardised formats

4.  ADDITIONAL SERVICES

5.  TEXTLINE INTERNATIONAL LANGUAGE SERVICES

6.  BULLETIN
    New UK Company Annual Reports Service from Reuter Textline

Enter code number for service required: 3


ACCOUNTLINE Published Formats

1      UK Company Accounts Service

DATALINE     Standardised Formats

2      Last five years accounts
3      Internally generated forecast
4      User generated forecast

DATALINE     NAMES Company mnemonics

Enter code number required or NAMES: names

COMPANY MNEMONICS LIST

Is the requirement to enter:
    1. a company name to find a mnemonic?
    2. a company mnemonic to find a name?
Enter code number required: 2

* = Company available for use with DATALINE 2 and 3

Enter characters for company mnemonic: rnltl

  Code          Company Name                      Last Accts Country

  RNLTL     RENAULT                               (31-12-87) (FRANCE)
? \
```

```
ACCOUNTLINE Published Formats

1     UK Company Accounts Service

DATALINE    Standardised Formats

2     Last five years accounts
3     Internally generated forecast
4     User generated forecast

DATALINE    NAMES Company mnemonics

Enter code number required or NAMES: 2

DATALINE -        LAST FIVE YEARS ACCOUNTS

Company required: rnltl

RENAULT
The data is now being collated.

Options available are to display:
        1. Income statement
        2. Balance sheet
        3. Financing table
        4. Accounting ratios
        5. All four statements
Enter code number required: 1

Is the tabulation to be a:
        1. Summary
        2. Basic analysis
        3. Detailed analysis
Enter code number required: 1
```

RENAULT	RNLTL				FRANCE
	INCOME STATEMENT			Francs (m)	
Period ending	31-12-83	31-12-84	31-12-85	31-12-86	31-12-87
SALES	101714.	106911.	111382.	134935.	147510.
TRADING PROFIT	434.	-11363.	-8007.	-1237.	5701.
PRE-TAX PROFITS	-1674.	-12803.	-12255.	-5210.	3562.
PROFIT AFTER TAX	-1576.	-12555.	-10925.	-5858.	3689.
ADJUSTMENTS	227.	166.	-28.	176.	433.
EARNED FOR ORDINARY	-1803.	-12721.	-10897.	-6034.	3256.

```
Options available are to display:
        1. Income statement
        2. Balance sheet
        3. Financing table
        4. Accounting ratios
        5. All four statements
Enter code number required: \
```

```
DATALINE -        LAST FIVE YEARS ACCOUNTS

Company required: \


ACCOUNTLINE Published Formats

1     UK Company Accounts Service

DATALINE    Standardised Formats

2     Last five years accounts
3     Internally generated forecast
4     User generated forecast

DATALINE    NAMES Company mnemonics

Enter code number required or NAMES: \


REUTER TEXTLINE
---------------

1.  TEXTLINE
    Articles for the period 1-1-80 to 19-5-89

2.  NEWSLINE
    News headlines for the period 13-5-89 to 19-5-89

3.  ACCOUNTLINE & DATALINE
    Company accounts - published and standardised formats

4.  ADDITIONAL SERVICES

5.  TEXTLINE INTERNATIONAL LANGUAGE SERVICES

6.  BULLETIN
    New UK Company Annual Reports Service from Reuter Textline


(AND (= CODE "rnltl") (= YR 1987))
(= CODE "rnltl")
(= YR 1987)CODE RNLTL 1987
CODE RNLTL
SALES RNLTL 1987
CODE RNLTL
STATE 1
Yearlist Period ending                    31-12-83 31-12-84 31-12-85 31-12-86 31
-12-87
made match
147510.000000
got here
Result : 0
 RNLTL 147510.

RNLTL
147510
((CODE SALES) ("RNLTL" "147510"))
```

Appendix A

Common LISP Files

A.1 File DATALN.LSP

```
; ++++++++++++++++++++++++++++++++++
; ++++++++++DATALN.LSP+++++++++++++++
; ++++++++++++++++++++++++++++++++++

(defun display-DATALINE-self-info ()
  (if (y-or-n-p "Do you want to see a description of the Dataline system?")
      (progn (format t "~%
                    REUTERS DATALINE
                    ----------------~%

Reuters' Dataline is a menu driven system which provides information
on a wide range of corporations, concentrating on those in the United
Kingdom. The data available includes what is on typical, condensed
balance sheets, income statements, financing tables and accounting
ratio summaries. ~%

Dataline is a menu-driven database system, so it does not operate with
tables as the standard SQL database would. Currently, the table name
location within the abstract local query call is being called 'data
or 'finsbury, although it does not serve a purpose. Perhaps it will
eventually in distinguishing between Dataline and Textline, for
example.~% ")
             (format t "Hit return to continue.~%")
             (read-char))))


(defun get-DATALINE-tables ()
  '(("data")))

(defun get-DATALINE-columns ()
  '(("Column name") ("CODE") ("COMPANYNAME") ("YR") ("CURRENCY")
    ("COUNTRY") ("SALES") ("TOT-SALES") ("EFO") ("TOT-CURR-ASSETS")
    ("TOT-ASSETS-EMP") ("NET-FIXED-ASSETS") ("TOT-STOCK")
    ("CURR-LIABILITIES") ("TOT-DEF-LIABILITY") ("CAPITAL-RESERVES")
    ("ADJ-EARN-SHARE") ("ROSE") ("MIN-INTEREST") ("TOTTAX")))

(defun get-DATALINE-data (abstract-local-query)
  (format t "Your selection is being formatted for Reuters...~%")
  (setq menu-list '())
  (menu-choice (get-current-object) abstract-local-query))
```

A.2 File ABSTQ.LSP

```
(defun menu-choice (DBMS-obj abstract-local-query)
                          ;;this procedure controls access
  (setq code-list '())                    ;;to Reuters
  (setq company-list '())
  (let* ((lqpdir   (get-object DBMS-obj 'lqp-specific-directory))
         (devdir   (get-object DBMS-obj 'comm-server-directory))
         (script   (get-object DBMS-obj 'communications-script))
         (phone    (get-object DBMS-obj 'phone-number))
         (account  (get-object DBMS-obj 'account))
         (passwd   (get-object DBMS-obj 'password))
         (columns-needed (parse-columns (cadr abstract-local-query)))
         (conditions (name-codelists (caddr abstract-local-query))))
   (print code-list)
   (setq commtest (nstring-downcase (format nil "~A/commtest" lqpdir)))
   (setq communicate (nstring-downcase (format nil "~A/~A" devdir script)))
   (setq readfile (nstring-downcase (format nil "~A/readfile" lqpdir)))
   (setq tfile (nstring-downcase (format nil "~A/tfile" lqpdir)))
   (setq filter (nstring-downcase (format nil "~A/filter" lqpdir)))
   (setq newtfile (nstring-downcase (format nil "~A/newtfile" lqpdir)))
;; (setq finalfile (nstring-downcase (format nil "~A/finalfile" lqpdir)))
   (setq readformat.awk (nstring-downcase (format nil "~A/readformat.awk"
                                                  lqpdir)))
    (setq awkcall (nstring-downcase (format nil "~A/awkcall" lqpdir)))
    (setq cx (nstring-downcase (format nil "~A/cx" lqpdir)))
    (setq cxmenu (nstring-downcase (format nil "~A/cxmenu" lqpdir)))
    (system (unix-format "rm ~A" communicate))
                                    ;;clear file of previous routines
    (system (unix-format "cat ~A > ~A" commtest communicate))
                        ;;dump standard dialup routine
    (system (unix-format "chmod +x ~A" communicate))
                                    ;;make file executable
    (setq no-dup-list '());;return list which has no duplicate codes
   ;;remove-duplicates doesn't work since names are within " "(need equal)
    (print code-list)
    (print conditions)
    (setq code-list (remove-duplic-names code-list));;all distinct codes
    (print code-list);;remove does not work
    (setq no-dup-list '());;return list with no duplicate company names
    (setq company-list (remove-duplic-names company-list));;distinct comps.
    (commun code-list '2 company-list);;send all codes through cx
    ;;companylist is sent to trip the code '1' out of names routine
    (print company-list)
    (commun company-list '1 '());;send all companies through cx
    (print columns-needed)
    (menu-slot (cadr abstract-local-query));;using columns-needed would
              ;;give strings in capitals--if the elements in search-list
              ;;were strings, they would be treated as lower case
              ;;thus, no changing letters for this
    (menu-choice-format-call final-list code-list)
    (system (unix-format "~A | cu -t ~A | tee tfile"
             communicate phone));;the call to Reuters
    (system (unix-format "rm ~A" readfile))
```

```
;;     (system (unix-format "rm ~A" finalfile))
     (system "rm finalfile")
     (system (unix-format "cat ~A | ~A > ~A" tfile filter newtfile))
     (link-parser-to-filter (cadr abstract-local-query)
            (parse-conditions (caddr abstract-local-query)))
     (system (unix-format "awk -f ~A ~A" readformat.awk readfile))
     (read-values "finalfile" columns-needed)
     ))


(defun parse-columns (column-list);;unquoted column will be capitalized
  (cond ((null column-list)        ;; quoted column remains as given
         (system "echo 'no list of columns present'")
         'ERROR)
        ((atom column-list)
         (if (stringp column-list);;if format of "column"[double quotes]
             (list column-list)   ;;this prevents error
             (progn (system "echo 'converting to string'")
               (list (format nil "~A" column-list)))));;put column in ""
        ((listp column-list)
         (cond ((equal 1 (length column-list))
                (parse-columns (car column-list)))
               (t (append (parse-columns (car column-list))
                          (parse-columns (cdr column-list))))))))


(defun name-codelists (condition-list);;form lists of codes & companies
  (cond ((or (equal (car condition-list) 'and)
             (equal (car condition-list) "AND")
             (equal (car condition-list) 'or)
             (equal (car condition-list) "OR"))
         (name-codelists (cadr condition-list))
         (name-codelists (caddr condition-list)))
        ((relation-p (car condition-list))
         (cond ((equal (second condition-list) 'code)
                (setq code-list (cons (third condition-list) code-list)))
               ((equal (second condition-list) 'companyname)
                (setq company-list (cons (third condition-list)
                                         company-list)))
               (t )))))



(defun relation-p (relation);;test for a relation
  (or
   (eq relation '=)
   (eq relation '>)
   (eq relation '<)
   (eq relation '>=)
   (eq relation '<=)
   (eq relation '<>)))
```

```
(defun remove-duplic-names (code-or-name-list)
  (cond
   ((null code-or-name-list)
    no-dup-list)
   (t (setq flag 0)
      (print flag)
      (let ((test (car code-or-name-list)))
        (defun check-rest-of-list (name rest-of-list)
          (cond ((null rest-of-list)
                 (if (eq flag 0)
                     (setq no-dup-list (cons name no-dup-list))))
                (t (if (equal name (car rest-of-list))
                       (setq flag 1))
                   (check-rest-of-list name (cdr rest-of-list)))))
        (check-rest-of-list test (cdr code-or-name-list)))
      (remove-duplic-names (cdr code-or-name-list)))))



(defun commun (send-list num next-list)
  ;;send codes and names through cx to Dataline
  (cond
   ((null send-list)
    )
   ((null (cdr send-list))
    (if (null next-list)
        (system (unix-format "~A ~A ~A ~A" cx num
                             (string (car send-list)) "yes"))
      (system (unix-format "~A ~A ~A ~A" cx num
                           (string (car send-list)) "no")))
    (commun (cdr send-list) num next-list))
   (t (system (unix-format "~A ~A ~A ~A" cx num
                           (string (car send-list)) "no"))
      (commun (cdr send-list) num next-list))))



(defun menu-choice-format (best-list-for-menu code first-pass?)
  (cond ((null (car best-list-for-menu))
         )
        (t(system (unix-format "~A ~A ~A ~A ~A ~A" cxmenu
                               (string first-pass?) (string code) "true"
                               (caar best-list-for-menu) (cadar best-list-for-menu)))
          (setq first-pass? '"no")
          (menu-choice-format (cdr best-list-for-menu) code first-pass?))))


(defun menu-choice-format-call (best-list code-list)
  (cond ((null (car code-list))
         (system (unix-format "~A ~A ~A ~A" cxmenu "false" 0 0)))
        (t (menu-choice-format best-list (car code-list) '"first")
           (menu-choice-format-call best-list (cdr code-list)))))
```

A.3 File DOIT.LSP

```
(defun unix-format(str &rest args)
       ;;enables passing values from lisp to shell
  (setq args (mapcar #' (lambda (x) (format nil "~C~A~C" #\" x #\"))
                     args))
  (apply #'format (cons nil (cons str args))))



(setq search-list '(((code country currency periodending sales companyname
                              efo yr)
        (tot-def-liability)(adj-earn-share tottax))
        ((companyname code net-fixed-assets yr tot-assets-emp currency
               country capital-reserves)
         (curr-liabilities)(tot-curr-assets tot-stock))
        ((yr code companyname currency country)()())
        ((rose code country companyname)()())))
;;this is the prototype list set up to get the
;;   correct number code for Dataline

(defun menu-slot (column-list);jump through column-list to get most
  (setq elem-list '())         ;effective calls in Reuters Dataline menu
  (print "curr-column-list")
  (print column-list)
  (setq return-list '())
  (setq zap-same-column '())
  (setq found 0)
  (setq option-num 0)
  (setq tab-num 0)
  (cond ((null column-list);;when done
         (setq final-list (pick-best (remove-extras menu-list
                                              (which-called-max menu-list)))))
        (t (setq some-list (find-options (car column-list) search-list))
           (if (not (null some-list))
               (setq menu-list (cons some-list menu-list)))
           (menu-slot (cdr column-list)))))


(defun find-options (column-name sch-lst);;find which option in Dataline
  (setq tab-num 0)
  (setq option (car sch-lst))
  (setq option-num (+ 1 option-num))
  (cond ((null option)
         return-list)
         (t (find-tabs column-name option)
            (if (not (null elem-list))
                (progn
                  (setq return-list (cons elem-list return-list))
                  (setq elem-list '())
                  (setq found 0)))
            (find-options column-name (cdr sch-lst)))))
```

```
(defun find-tabs (column-name option);;find correct tabulation
  (setq tab-type (car option))
  (setq tab-num (+ 1 tab-num))
  (cond ((null tab-type)
         )
        (t (element? column-name tab-type)
           (if (equal found 0)
               (find-tabs column-name (cdr option))))))

(defun element? (column-name tab-type);;sees if column-name is in
  (cond ((null tab-type)                ;;   this tabulation
         )
        (t (if (equal (car tab-type) column-name) ;;;found
               (progn
                 (setq elem-list (list option-num tab-num))
                                ;;mark position
                 (setq found 1))   ;;;flag set
             (element? column-name (cdr tab-type))))));;try next



(defun pick-best (m-slot-list)
  (setq crunched-list '())
  (setq do-list m-slot-list)
  (do ((counter 1)
       (max 0))
      ((> counter 4) crunched-list)
      (cond ((null do-list)
             (if (> max 0)
              (setq crunched-list (cons (list counter max) crunched-list)))
             (setq counter (+ counter 1))
             (setq do-list m-slot-list)
             (setq max 0))
            ((equal (caar do-list) counter)
             (if (> (cadar do-list) max)
                 (setq max (cadar do-list)))
             (setq do-list (cdr do-list)))
            (t (setq do-list (cdr do-list))))))


(defun which-called-max (bulk-menu-list);;determine option called most
  (setq do-list bulk-menu-list)
  (setq running-tot 0)
  (do ((counter 1)
       (max 1)
       (tot 0))
      ((> counter 4) max)
      (cond ((null do-list)
             (if (> running-tot tot)
                 (progn
                   (setq tot running-tot)
                   (setq max counter)))
             (setq running-tot 0)
             (setq do-list bulk-menu-list)
             (setq counter (+ counter 1)))
            (t (setq running-tot (scan-list (car do-list) counter
```

```
                                                    running-tot))
              (setq do-list (cdr do-list))))))

(defun scan-list (find-max-list counter running-tot)
  (cond ((null find-max-list)
         running-tot)
        (t (if (equal (caar find-max-list) counter)
               (setq running-tot (+ running-tot 1)))
           (scan-list (cdr find-max-list) counter running-tot))))


(defun remove-extras (extra-calls-list max) ;sets up for extra-crunch
  (setq work-list (car extra-calls-list))   ;one column-list at a time
  (cond ((null extra-calls-list) ;procedure eliminates duplicate calls
         zap-same-column)        ;for a single column name(in most cases)
        (t (if (max-in-list? work-list max)
               (extra-crunch work-list max)
             (setq zap-same-column (append work-list zap-same-column)))
           (remove-extras (cdr extra-calls-list) max))))




(defun max-in-list? (test-list max) ;Find if the column-list has the max #
  (cond ((null test-list)
         nil)       ; Not there, return false
        (t (if (equal (caar test-list) max)
               t    ;If so, return true
             (max-in-list? (cdr test-list) max)))))



(defun extra-crunch (zap-list max) ;routine to crunch all in column list
  (cond ((null zap-list)
         zap-same-column)
        (t (if(equal (caar zap-list) max)  ;which aren't of maximum value
               (setq zap-same-column (cons (car zap-list) zap-same-column)))
           (extra-crunch (cdr zap-list) max)))) ;check next element
```

A.4 File PARSER.LSP

```
;This file will contain the routines for parsing the condition list
;  so that the proper format is sent to the awk filter file for analysis.
;  It will establish lists with years specified, along with code names
;  to be sent. Columns will also be sent to awk, but perhaps from routine
;  menu-choice.


(defun parse-conditions (condition-list)
  (print condition-list)
  (cond ((equal condition-list 'ERROR)
         'ERROR)
        ((null condition-list)
           condition-list)
        ((or (equal (car condition-list) 'and)
             (equal (car condition-list) "AND")
             (equal (car condition-list) "and")
             (equal (car condition-list) "And"))
         (link-and    (parse-conditions (cadr condition-list))
                      (parse-conditions (caddr condition-list))))

        ((or (equal (car condition-list) 'or)
             (equal (car condition-list) "OR")
             (equal (car condition-list) "or")
             (equal (car condition-list) "Or"))
         (link-or     (parse-conditions (cadr condition-list))
                      (parse-conditions (caddr condition-list))))

        ((equal (car condition-list) '=)
         condition-list)

        ((not (equal (car condition-list) '=))
         condition-list)))

;; next version(t 'error)))


(defun link-and (list-one list-two)
  (cond
   ((or (equal list-one 'ERROR)
        (equal list-two 'ERROR))
    'ERROR)
   ((or (equal list-one '()) (equal list-two '()))
                          ;;either list nil, return nil
    '())
   ((and (equal (car list-one) '=) (equal (car list-two) '=))
    (cond;;rework this
     ((and (not (null (member (cadr list-one) '(code company-name))))
           (not (null (member (cadr list-two) '(code company-name)))))
      ;;test if names are different
      ;; if same company, proceed
      (if (equal (caddr list-one) (caddr list-two));;same name, return one
          (list (list (list (caddr list-one))))
```

```
      (progn
        (format t "You can not 'and' two different companies~%")
       'ERROR)))
  ((and (not (null (member (cadr list-one) '(code company-name))))
        (equal (cadr list-two) 'yr))
   ;;test for bad name
   (list (list (list (caddr list-one)) (caddr list-two))))
  ((and (equal (cadr list-one) 'yr)
        (not (null (member (cadr list-two) '(code company-name)))))
   ;;test for bad name
   (list (list (list (caddr list-two)) (caddr list-one))))
  ((and (equal (cadr list-one) 'yr)
        (equal (cadr list-two) 'yr))
   ;;test later to see if all lists have codes!!!!!
   (if (equal (caddr list-one) (caddr list-two))
       (list (list (list ) (caddr list-one)))
     (progn
       (format t "You can not 'and' 2 different years~%")
       'ERROR)))
  (t (format t "Use only columns 'code', 'company-name' and 'yr'~%")
     'ERROR)));;Make code more rigorous to avoid future problems

((and (not (equal (car list-one) '=)) (equal (car list-two) '=))
 ;;first is compound list, second is not
 (cond
  ((or (equal (cadr list-two) 'code)
       (equal (cadr list-two) 'company-name))

   (if (null (caar list-one));;want to 'and' code with just yearlist
       (list (append (list (list (caddr list-two))) (cdar list-one)))
     (progn
       (if (equal (look-for-code list-one (caddr list-two)) 'notvalid)
           ;;routine to test if all codes in list one
           ;;are same as that in list-two--if false,error
           (progn
            (format t "Can not 'and' another company here...use 'or'~%")
            'ERROR)
         list-one))))
  ((equal (cadr list-two) 'yr)
   (if (null (caar list-one));;only years in compound list--yr && yr
       (if (member (caddr list-two) (cdar list-one));same yr in both?
           (list (list (list ) (caddr list-two)));yes-return list of yr
         '())                                        ;;no, return nil

     ;;make sure year is in each sublist of list one
     (progn
       (setq return-list '())
       (attach-year list-one (caddr list-two))
       return-list)))
  (t (format t "Codes available are 'code', 'company-name' and 'yr'~%")
     'ERROR)));;rigorous code
 ((and (equal (car list-one) '=) (not (equal (car list-two) '=)))
  (cond
   ((or (equal (cadr list-one) 'code)
        (equal (cadr list-one) 'company-name))
    (if (null (caar list-two));;want to 'and' code with yearlist
```

```
          (list (append (list (list (caddr list-one))) (cdar list-two)))
         (progn
           (if (equal (look-for-code list-two (caddr list-one)) 'notvalid)
               ;;see if all codes in list2 are same as new code,f-err
               (progn
                (format t "Can not 'and' another company here..use 'or'~%")
                 'ERROR)
             list-two))))
      ((equal (cadr list-one) 'yr);;just a year
       (if (null (caar list-two));;just yearlist
           (if (member (caddr list-one) (cdar list-two));yr in yearlist??
               (list (list (list ) (caddr list-one)));yes-return list of yr
             '());;no, return nil
         ;;make sure year is in each sublist of list two
         (progn
           (setq return-list '())
           (attach-year list-two (caddr list-one))
           return-list)))
       (t (format t "Available columns -> 'code', 'companyname' and 'yr'~%")
          'ERROR)));;rigorous code
     ((and (not (equal (car list-one) '=)) (not (equal (car list-two) '=)))
      ;;test if any 2 names in either list are different---then error
      ;;if names are the same, perform logical and on their yearlists
      (if (or (null (caar list-one));;if either is list of just years, error
              (null (caar list-two)))
          (progn
            (format t "You can not 'and' two lists if any has just years~%")
            'ERROR)
        (progn
          (if (equal (test-double-compounds list-one list-two) 'notvalid)
              (progn
                (format t "Can not 'and' these two lists here...use 'or'~%")
                'ERROR)
      ;;;perform and on the yearlists
           ))))))


(defun look-for-code (which-list which-code)
  (cond  ;;this procedure tests for the occurence of any code other than
         ;;the specified one(in the current working list for return)
         ;;if there is, it returns "notvalid", otherwise "valid"
   ((null which-list)
    'valid)
   (t   ;;Only one code per sublist, so recursing it is not necessary
    (if (not (equal (caaar which-list) which-code))
        'notvalid
      (look-for-code (cdr which-list) which-code)))))



(defun test-double-compounds (list-one list-two)
  (cond  ;;this recurses through list-two, as look-for-code takes care of
   ((null list-two)  ;; list-one
    'valid)    ;;did not find two unequal codes in lists
   (t (if (equal (look-for-code list-one (caaar list-two)) 'notvalid)
          'notvalid  ;;most probable---found unequal codes in lists
```

```
        (test-double-compounds list-one (cdr list-two))))))


(defun attach-year (which-list year-to-attach);;AND year onto compound list
  (cond
   ((null which-list)
    return-list)
   (t (if (not (member year-to-attach (cdar which-list)));yr not in sublist
          (setq return-list (cons (append (list (caar which-list)) (cons
                    year-to-attach (cdar which-list))) return-list))
            ;;attach on the newly binded year
        (setq return-list (cons (car which-list) return-list)))
           ;;do not bind the year, it's already there
      (attach-year (cdr which-list) year-to-attach))))




;;this used to be link-and, shall be turned into link-or
(defun link-or (list-one list-two);take 2 sublists & join by boolean or
  (cond
   ((or (equal list-one 'ERROR)
        (equal list-two 'ERROR))
    'ERROR)
   ((and (null list-one) (not (null list-two)))
    list-two)
   ((and (not (null list-one)) (null list-two))
    list-one)
   ((and (null list-one) (null list-two))
    '())
   (t
    (cond
     ((and (equal (car list-one) '=) (equal (car list-two) '=))
                   ;;both are smallest
      (cond ;units sent from original cond. list-eg.(= code "hond")
       ((and (not (null (member (cadr list-one) '(code company-name))))
             (not (null (member (cadr list-two) '(code company-name)))))
        ;; both are codes or company-names
        ;test for validity of names of companies
        (if (equal (caddr list-one) (caddr list-two));repetition of company
            (list (list (list (caddr list-one))));;return only one
         (list (list (list (caddr list-one)))
                              (list (list (caddr list-two))))))
       ;;2 distinct codes
       ;;returned in form ((("hond")) (("renlt")))

       ((and (not (null (member (cadr list-one) '(code company-name))))
             (equal (cadr list-two) 'yr));2nd list is a year
          (format t "you can not 'or' a year with a code~%")
```

```
    'ERROR)

  ((and (equal (cadr list-one) 'yr)
        (not (null (member (cadr list-two) '(code company-name)))))
    (format t "you can not 'or' a year with a code~%")
    'ERROR)

  ((and (equal (cadr list-one) 'yr) (equal (cadr list-two) 'yr))
   (if (equal (caddr list-one) (caddr list-two))
       (list (list (list ) (caddr list-one)))
     (list (list (list ) (caddr list-one) (caddr list-two)))))
;;list has only years, no code---look back at 'and' code due to this
  (t (format t "Only columns available are 'code'
                                  'company-name' and 'yr'~%")
    'ERROR)));;more rigorous code


((and (not (equal (car list-one) '=)) (equal (car list-two) '=))
 ;;list-one is a compound list, list 2 is in original format
 (cond
  ((null (caar list-one));;a list of only years or'ed
   (if (or (equal (cadr list-two) 'code);try to 'or' code with yrlist
           (equal (cadr list-two) 'companyname));;return code in list
       (progn
         (format t "Can not 'or' a yearlist with a code[use and]~%")
         'ERROR)
     (if (member (caddr list-two) (cdar list-one))
         list-one
       (list (append (list '()) (cons (caddr list-two)
                                      (cdar list-one)))))))
       ;;this establishes a new list of just years
  ((or (equal (cadr list-two) 'code) ;;'or' a code with compound list
       (equal (cadr list-two) 'company-name))
   (union list-one (list (list (list (caddr list-two))))));'OR' code
  (t ;;try to 'or' a year with list which has codes
   (format t "you can not 'or' a year in here--try 'and'~%")
     'ERROR)))
((and (equal (car list-one) '=) (not (equal (car list-two) '=)))
 ;;list-two is a compound list, list 1 is in original format
 (cond
  ((null (caar list-two));;just compound yearlist--no code associated
   (if (or (equal (cadr list-one) 'code)
           (equal (cadr list-one)'company-name))
       (progn
         (format t "Can not 'or' a code with a yearlist[use and]~%")
         'ERROR)
     (if (member (caddr list-one) (cdar list-two))
         list-two
       (list (append (list '()) (cons (caddr list-one)
                                      (cdar list-two)))))))
  ((or (equal (cadr list-one) 'code);;'or' compound list with code
       (equal (cadr list-one) 'company-name))
   (union list-two (list (list (list (caddr list-one))))));'OR' code
  (t (format t "you can not 'or' a year in here--try 'and'~%")
     'ERROR)))
```

```
     ((and (not (equal (car list-one) '=)) (not (equal (car list-two) '=)))
      ;;both lists are compound
      (cond ((and (null (caar list-one));;both are yrlists, so perform 'or'
                  (null (caar list-two)))
             (return-one-year-list (cdar list-one) (cdar list-two)))
            ((or (null (caar list-one));;only one is yearlist, so error
                 (null (caar list-two)));;***Should this be so?????
             (format t "You can not or a compound list of just years~%")
             'ERROR)
            (t (union list-one list-two)))))))));;neither is a yearlist


(defun return-one-year-list(list-one list-two)
  (cond ((null list-two)
         (list (cons '() list-one)))
        (t (if (not (member (car list-two) list-one))
               (setq list-one (cons (car list-two) list-one)))
           (return-one-year-list list-one (cdr list-two)))))
```

A.5 File LNKPARSAWK.LSP

```
;;This program will contain the necessary call to provide the interface
;;   between the condition parser and the filter written in awk.


(defun link-parser-to-filter (column-list parsed-list)

  (if (equal (car parsed-list) '=);;takes care of parsing for case
                                  ;;with no boolean operator
      (if (not (null (member (cadr parsed-list) '(code companyname))))
          (setq parsed-list (list (list (list (caddr parsed-list)))))
        (progn
          (format t "You must use only code or companyname")
          (break)
          'ERROR)))
  (print parsed-list)
   (cond
        ((null parsed-list)
         (system (unix-format "echo EOF >> ~A" readfile))
         )

;;This sends a string of the column lists, the company code
;;  and a string of the yearlist to shell file awkcall to specify
;;  the filtering terms

        (t
         (system (unix-format "echo New company >> ~A" readfile))
         (loop-through-columns column-list parsed-list)
         (link-parser-to-filter column-list (cdr parsed-list)))))


(defun loop-through-columns (column-list parsed-list);;sequentially
                                                     ;; provide columns
    (cond
     ((null column-list)
      );;;sends shell program awkcall proper values
     (t    (system (unix-format "echo Column >> ~A" readfile))
           (system (unix-format "~A ~A ~A ~A" awkcall
                                (string (car column-list))
                (string (caaar parsed-list)) (form-string
                                (cdar parsed-list))))
           (loop-through-columns (cdr column-list) parsed-list))))




(defun form-string (yearlist) ;;This will turn a company's yearlist and
        ;;column list into a string for easier processing in shell & AWK
  (cond
   ((null (cdr yearlist))
    (format nil "~A" (car yearlist)))
   (t (format nil "~A ~A" (car yearlist) (form-string (cdr yearlist))))))
```

A.6 File READER.LSP

```
;;this program will perform the reading of values from readfile
;;  back into the form needed by the GQP


(defun read-values (file columns-needed)
  (if (not (probe-file file))
      (format t "file does not exist, so results can not be read~%")
    (progn
      (setq tmp '())
      (setq info '())
      (with-open-file (data file)
                      (loop  (let ((line (read data nil 'EOF)))
                               (cond ((equal line 'EOF)
                                      (return (cons columns-needed
                                                    (reverse info))))
                                     ((equal line 'blank)
                                      (setq info (cons (reverse tmp) info))
                                      (setq tmp '()))
                                    (t (print line)
                                     (setq tmp (cons (format nil "~A" line)
                                                     tmp)))))))))))
```

A.7 File LOADFILES.LSP

```
;This is the file which is called to load all the other files
; necessary for the DATALINE LQP

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/abstq.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/ABSTQ.LSP is missing!!! ~%"))

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/doit.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/DOIT.LSP is missing!!!! ~%"))

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/parser.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/PARSER.LSP is missing!!!! ~%"))

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/dataln.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/DATALN.LSP is missing!!!! ~%"))

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/lnkparsawk.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/LNKPARSAWK.LSP is
                                                  missing!!!! ~%"))

(if (not (probe-file "/usr/cistk/demo/v2/lqp/finsbury/reader.lsp"))
 (format t "/usr/cistk/demo/v2/lqp/finsbury/READER.LSP is missing!!!! ~%"))

(load "/usr/cistk/demo/v2/lqp/finsbury/abstq.lsp")
(load "/usr/cistk/demo/v2/lqp/finsbury/doit.lsp")
(load "/usr/cistk/demo/v2/lqp/finsbury/parser.lsp")
(load "/usr/cistk/demo/v2/lqp/finsbury/dataln.lsp")
(load "/usr/cistk/demo/v2/lqp/finsbury/lnkparsawk.lsp")
(load "/usr/cistk/demo/v2/lqp/finsbury/reader.lsp")

;system can then be called with just get-DATALINE-data
```

A.8 File LOADDEMO.LSP

```
(load "/usr/cistk/demo/v2/cis-tk")
(load "/usr/cistk/demo/v2/lqp/finsbury/loadfiles.lsp")

;;(put-object 'finsbury 'lqp-specific-directory '/usr/cistk/hgerber)
;;(put-object 'finsbury 'comm-server-directory '/usr/cistk/hgerber)
;;(put-object 'finsbury 'lqp-common-directory '/usr/cistk/hgerber)

(put-object 'finsbury 'lqp-specific-directory
                      '/usr/cistk/demo/v2/lqp/finsbury)
(put-object 'finsbury 'comm-server-directory
                      '/usr/cistk/demo/v2/lqp/dev)
(put-object 'finsbury 'lqp-common-directory
                      '/usr/cistk/demo/v2/lqp/finsbury)
```

Appendix B

UNIX Script Files

B.1 File COMMTEST

```
sleep 20
echo ""
sleep 2
echo ""
sleep 2
echo ""
sleep 2
echo "c 202430\r\c"    #account should eventually be passed as
sleep 5                      # parameter $1
echo "\r\c"
sleep 5
echo "hello rna132*325250\r\c"  #password should be passed as $2
echo "\r\c"
sleep 10
echo "3\r\c"
sleep 8
```

B.2 File CX

```
DATALINEDIR=/usr/cistk/demo/v2/lqp/dev ## path name independency

echo echo '"'names'\\'r'\\'c'"' >> $DATALINEDIR/communicate
echo "sleep 6" >> $DATALINEDIR/communicate
echo echo '"'$1'\\'r'\\'c'"' >> $DATALINEDIR/communicate
echo "sleep 5" >> $DATALINEDIR/communicate
echo echo '"'$2'\\'r'\\'c'"' >> $DATALINEDIR/communicate
echo "sleep 10" >> $DATALINEDIR/communicate
echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
# first \ guards first quote, then quotes send 2 \'s
# another quote is guarded, then regular \r\c is sent
echo "sleep 7" >> $DATALINEDIR/communicate
if test $3 = "yes"
        then
           echo echo '"2\\r\\c"' >> $DATALINEDIR/communicate
           echo "sleep 5" >> $DATALINEDIR/communicate
fi
```

B.3 File CXMENU

```
## path name dependency

DATALINEDIR=/usr/cistk/demo/v2/lqp/dev

if test "$1" = first
   then
        echo echo '"'$2'\\'r'\\'c'"' >> $DATALINEDIR/communicate
        echo "sleep 8" >> $DATALINEDIR/communicate
fi

if $3 = "true"
   then
        echo echo '"'$4'\\'r'\\'c'"' >> $DATALINEDIR/communicate
        echo "sleep 5" >> $DATALINEDIR/communicate
        echo echo '"'$5'\\'r'\\'c'"' >> $DATALINEDIR/communicate
        echo "sleep 25" >> $DATALINEDIR/communicate
        echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
        echo "sleep 7" >> $DATALINEDIR/communicate
   else
        echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
        echo "sleep 4" >> $DATALINEDIR/communicate
        echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
        echo "sleep 4" >> $DATALINEDIR/communicate
        echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
        echo "sleep 4" >> $DATALINEDIR/communicate
        echo echo \''\\\\'\''"\\r\\c"' >> $DATALINEDIR/communicate
        echo echo ~. >> $DATALINEDIR/communicate
fi
```

B.4 File AWKCALL

```
## This shell file links the conditions parser to the filter.
## It is called awkcall, as the filter it calls is written in awk.
##   $1 is the column list string
##   $2 is the company code string
##   $3 is the year list string

## for path name independency

DATALINEDIR=/usr/cistk/demo/v2/lqp/finsbury

temp=`echo $1 | tr [a-z] [A-Z]`
                          #turn column into all capital letters
COL=`awk '$1 ~ /^'"$temp"'/{tot_column = "";
                          # take columns 2->NF as column name
                       for (loop = 2; loop < NF; loop++)
                           tot_column = tot_column $loop " ";
                       tot_column = tot_column $loop;
                                   #do not want extra space
                       print tot_column;}' column-search`
                                   #print sets COL

COMP=`echo $2 | tr [a-z] [A-Z]`
                     #turn company name into all capital letters


nawk -f $DATALINEDIR/filter2.nawk -v YEAR="$3" COMPANY="$COMP"
                              COLUMN="$COL" $DATALINEDIR/newtfile


##File newtfile must be the version which is filtered free of ^M
##This is accomplished by form---- cat tfile | filter > newtfile
##Filter is the file in this directory
```

Appendix C

AWK Files

C.1 File FILTER2.NAWK

```
#=============================================================#
# Use nawk to run this awk program
# 3 variables are expected from the call to awk :
#    YEAR : string of years requested
#    COMPANY : a code for a company (usually less than 4 char.)
#    COLUMN : column name of the information you are looking for.
#  (This file is called one time for each 3 variable combination)

# STATE VARIABLES :
#  STATE : 0 -> ignore input, wait for 80 '-' . Once you get it test
#               company code. If interredsting record set state to 2
#  STATE : 1 ->  I am in a relevant record waiting for the period line
#               when I get this line I keep it and switch to state 2
#  STATE : 2 ->  I am in a relevant record waiting  for the line
#               the field I am looking for
# INTERNAL VARIABLES :
#  YEARLIST : This is the line for the years to select
#                          a particular item
#  COLUMN1 : Pattern to test the column name.
# Remark : The input file lets control-M appear.
           To suppress these, pass through shell file filter
                      (sed type commands)
#=============================================================#
#                          FUNCTION LIBRARY                   #
#=============================================================#
# getvalue : Selector function-look for year y in array y_arr
             and then return the corresponding value from v_arr.

function getvalue (y, y_array, v_array)

# y              specific year you are looking for.
# y_array        year list
# v_array        value list
{if (y != "NIL"){
    # 2 cases--if y = "NIL" return all years, else those asked.
 for (loop = 1; loop < length(y); loop = loop + 3)
  #loop through all years
  #needed--only last 2 digits to represent each year (eg. "86 87")
        {FLAG = 0
         for (i in y_array)
              #go through all years available-1 year could have
              #two period endings
          if (match(y_array [i], (substr(y, loop, 2))))#found year
           { printf ("%f\n",v_array [i]);
             print "19" (substr(y, loop, 2)), v_array[i] >>
```

```
                                        (DATALINEDIR "readfile");
                    FLAG = 1;
                    printf("got here\n")
                          };
          if (FLAG == 0) {
             # The proposed date is not in the date array
           printf ("Error 101: Unable to get value-bad year\n");
           print (substr(y, loop, 2)), 0 >> (DATALINEDIR "readfile");
                        }       #print 0 value for bad yr
                }; # for end
        return;
 }#matches y!= "NIL"
 else {#return all years
        for (i in y_array)
          #put last 2 digits of year & value in readfile
         print substr(y_array[i],7), v_array[i] >>
                          (DATALINEDIR "readfile");
      };
} # end of getvalue
#=============================================================#
#                          PROLOGUE                           #
#=============================================================#
BEGIN {
        DATALINEDIR = "/usr/cistk/demo/v2/lqp/finsbury/";
        printf("%s %s %s\n",COLUMN, COMPANY, YEAR)
        # Defensive programming
        if (YEAR == ""){
                  YEAR = "NIL"};
        if ((COLUMN == "") || (COMPANY == ""))
        { #printf("%s %s %s\n",COLUMN, COMPANY, YEAR)
        printf("Error: must define company, column and year!\n");
                exit 1;
        };
        # Initialization for variables.
        gsub (/19/, "", YEAR)
        COLUMN1 = COLUMN "[\\ \\t]+[0-9\\.\\-\\+N*]";
                  #Added N condition 4 N/A
                  #in case of new firm/little info available
                  #star for missing period ending
                  #also took out "^" at beginning since field may be
                  #indented in output from Reuters
        STATE = 0;
      } # end of begin
#=============================================================#
#                          MAIN LOOP                          #
#=============================================================#
/^-----------------------------------/          {
                # Found a record to start with.
                # SKIP LINE TO GET COMPANY NAME.
                getline;
                # GET THE CURRENT COMPANY CODE
                #2 spaces separate fields properly
                FS = " [ ]+";#Field Separator is minimum of 2 spaces
                printf("CODE %s\n", $2);
                CODE = $2;
                COMP_NAME = $1;
```

```
                COUNTRY = $3;
                # TEST IF PROPER COMPANY
                getline; # Skip the next line--> it's 80 '-'
                if (CODE != COMPANY)
                {       # ASSUME REST OF RECORD IS GARBAGE
                        STATE = 0
                        next;
                };
                if (COLUMN == "CODE") {
                        print CODE >> (DATALINEDIR "readfile");
                        exit}
                if (COLUMN == "COMPANY NAME") {
                        print COMP_NAME >> (DATALINEDIR "readfile");
                        exit}
                if (COLUMN == "COUNTRY") {
                        print COUNTRY >> (DATALINEDIR "readfile");
                        exit}
                # WE GET A POSSIBLE RECORD FOR OUR SEARCH
                STATE = 1;
                getline;
                if (COLUMN == "CURRENCY")
                        if ($1 != "ACCOUNTING RATIOS") {
                                #currency not in accounting
                           print $3 >> (DATALINEDIR "readfile");#Not $2?
                           exit}#May be error here
                printf("STATE %d\n",STATE);
                } # end of pattern

$0 ~ /^Period ending/ && (STATE == 1)     {
                #took ^ out of beginning of pattern
                # Get the year list for this record.
                YEARLIST = $0;
                printf("Yearlist %s\n",YEARLIST)
                STATE = 2;
                } # end of pattern

($0 ~ COLUMN1) && (STATE == 2)          {
                printf("made match\n");
                # I am on the right line. I have to get the right
                # year now. Create the arrays and call getvalue.
                gsub (/Period ending/, "", YEARLIST);
                split (YEARLIST , y_arr, " ");
                 #form y_arr from spaces in YEARL.
                VALUELIST = $0;
                gsub (COLUMN, "", VALUELIST);
                split (VALUELIST, v_arr, " ");
                printf ("Result : %s\n", getvalue(YEAR, y_arr, v_arr));
                exit;
                } # end of pattern
#=============================================================#
```

C.2 File READFORMAT.AWK

```
#=============================================================================#
# This file formats the data in "READFILE", which is output by filter2.nawk
#    (and has all relevant values to be returned), so that the information
#     can be read back into lisp in the exact format necessary for the GQP.


#=============================================================================#
#                                  PROLOGUE                                   #
#=============================================================================#

BEGIN {
        DATALINEDIR = "/usr/cistk/demo/v2/lqp/finsbury/";
        CURRENT_SLOT = 1;
        NUMBER_OF_YEARS = 0;
        TOTAL_NUMBER_OF_YEARS = 0;

} #end of BEGIN

#=============================================================================#
#                                  MAIN LOOP                                  #
#=============================================================================#

$1 ~ /^[A-Za-z]+/{  #match a pattern of text

    if ((NUMBER_OF_YEARS > 0) && (TOTAL_NUMBER_OF_YEARS == 0))
        TOTAL_NUMBER_OF_YEARS = NUMBER_OF_YEARS;

    NUMBER_OF_YEARS = 0;

     if (($0 ~ /New company/) || ($0 ~ /EOF/)) {
     if (CURRENT_SLOT != 1) { #not first code used --must process
        counter = 0; #keep track of how many years returned
        while (counter < TOTAL_NUMBER_OF_YEARS) {
          for (loop = 1; loop < CURRENT_SLOT; loop++){
              if (RETURN_ARRAY[loop] !~ /([0-9]+|N\/A)/){  #number
                  RETURN_STRING = RETURN_STRING " " RETURN_ARRAY[loop];
                }
               else {
                RETURN_STRING = RETURN_STRING" "RETURN_ARRAY[loop+counter];
                loop = loop + TOTAL_NUMBER_OF_YEARS - 1;
                };

           }
            counter = counter + 1;
            print RETURN_STRING;
            print RETURN_STRING >> (DATALINEDIR "finalfile");
            print "blank" >> (DATALINEDIR "finalfile");
            RETURN_STRING = "";

        }

     for (i=1; i <= CURRENT_SLOT; i++)   #this is not working????
```

```
        delete RETURN_ARRAY[i]; #clear array for new company
      CURRENT_SLOT = 1;
      TOTAL_NUMBER_OF_YEARS = 0;
      }
     }

       else {
           if ($0 ~ /Column/){
                 next;
           }
           else{
                       RETURN_ARRAY[CURRENT_SLOT] = $0;
                       CURRENT_SLOT = CURRENT_SLOT + 1;
                       next;
                 }
           };

}

$1 ~ /^[0-9*]+/{    #match number dependent pattern(check \-)

         RETURN_ARRAY[CURRENT_SLOT] = $NF;
         CURRENT_SLOT = CURRENT_SLOT + 1;
           #add value in field 2(last field) to array for return
         NUMBER_OF_YEARS = NUMBER_OF_YEARS + 1;

}
```

Appendix D

Other Files Used

D.1 File COLUMN-SEARCH

The term in column 1 matches the exact representation of the column in Dataline(which is stored in column 2).

PERIODENDING	Period ending
CODE	CODE
COMPANYNAME	COMPANY NAME
YR	YR
CURRENCY	CURRENCY
COUNTRY	COUNTRY
SALES	SALES
TOT-SALES	TOTAL SALES
EFO	EARNED FOR ORDINARY
TOT-CURR-ASSETS	TOTAL CURRENT ASSETS
TOT-ASSETS-EMP	TOTAL ASSETS EMPLOYED
NET-FIXED-ASSETS	NET FIXED ASSETS
TOT-STOCK	TOTAL STOCK + WORK IN PROGRESS
CURR-LIABILITIES	CURRENT LIABILITIES
TOT-DEF-LIABILITY	TOTAL DEFERRED LIABILITIES
OTHER-LOANS	OTHER LOANS
CAPITAL-RESERVES	CAPITAL AND RESERVES
ADJ-EARN-SHARE	ADJ.EARNINGS PER SHARE
ROSE	RETURN ON SHAREHOLDERS EQUITY
MIN-INTEREST	MINORITY INTERESTS
TOTTAX	TOTAL TAXATION

D.2 File FILTER

```
sed s/\//g
```

D.3 File READFILE

```
The flags in this file are for the benefit of file READFORMAT.AWK.
  They serve as patterns which readformat.awk has been programmed
  to look for. A "New company" will be flagged whenever output is
  related to a new code which was input. "Columns" are flagged
  within any company's output section wherever a new column's
  representation is to begin. "EOF" marks the end of the data.

These flags are put into this file within function
  link-parser-to-filter in lnkparsawk.lsp.

Request in this example is for columns country, code and sales
   for the Wellcome Foundation, without any specified year.

New company      **Flag a new company
Column           **Flag a new column
U.K.
Column           **Flag a new column
WELCOM
Column           **Flag a new column
19** N/A
1986 1005400.
1987 1132400.
1988 1250500.
19** N/A
New company      **New company?
EOF              **No, end.
```

References

[Cardenas 89] Cardenas, Alfonso F.
Data Base Management Systems, 2nd Ed.
Wm. C. Brown Publishers, 1989.

[Champlin 88] Champlin, Alec R.
Interfacing Multiple Remote Databases in an Object-Oriented Framework.
May, 1988.

[Gan 89] Gan, Francis C.K.
An Architecture Design and Implementation of a Communication Server to Access Disparate Databases.
May, 1989.

[Murdick 71] Murdick, Robert G. and Ross, Joel E.
Information Systems for Modern Management.
Prentice-Hall, Inc., 1971.

[Radford 73] Radford, K.J.
Information Systems in Management.
Reston Publishing Company, Inc., 1973.

[Wang 88] Wang, Richard and Madnick, Stuart.
Evolution Towards Strategic Applications of Databases Through Composite Information Systems.
Connectivity Among Information Systems , 1988.

[Winston 84] Winston, P.H. and Horn, K.P.H.
LISP.
, 1984.