

CHAPTER I

INTRODUCTION

1.0 Motivation

There has been incredible progress in the areas of semiconductor manufacturing and *Integrated Circuits* (IC) in the past few decades. The emergence of automated *Very Large Scale Integrated Circuits* (VLSI) design tools has made it possible to have millions of transistors on a single chip, thus enabling designers to have powerful processing tools available for a competitive price. Personal Computers (PC's) have emerged as a result of powerful and efficient design techniques. The age of popular usage of PC's has come where these machines are used for processing huge amounts of data more than they are used for scientific processing. There has been significant progress in the areas of digital logic optimization and database information storage and retrieval; however, there has been very little transfer of these methods from these two domains. The motivation for the work presented here is based on the idea that digital logic optimization techniques can be used for efficient information processing. The impressive advances that have been accomplished in logic minimization indicate that they are likely to yield good results when applied to the field of information processing.

1.1 Description

The objective of this thesis is to investigate the use of *Computer Aided Design* (CAD) methods in the area of information storage and retrieval. *Binary Decision Diagrams* (BDD) [17] are the main form of decision diagram used for the data structures in this work. Boolean function manipulation has played a key role in the

area of digital system design. Until recently, Boolean functions have been represented using truth tables or cube lists. A cube list is a mere symbolic representation of set of product terms. The use of truth tables to represent Boolean functions requires a considerable amount of memory resources for even a small function. Cube lists may require lesser amounts of memory than representation of a function using truth tables; however, they have the disadvantage that they are not canonical and all possible Boolean functions cannot be represented efficiently using them. BDDs have emerged as a good solution for most functions encountered in modern designs. A detailed description of BDDs is given by R.E. Bryant in [4]. Other data structures used in this work are AND/OR graphs and *Multi valued Decision Diagrams* (MDDs) [33]. AND/OR graphs [11] have been proven to have superior properties as compared to BDDs in some respects [7].

1.2 Query Optimization

Relational database systems are currently the predominant technology for storing, handling, and querying large amounts of data. A key factor in the popular usage of this approach is the introduction and development of query optimization techniques [3]. One of the reasons for the commercial success of relational database systems is that they offer good performance to many business applications, mostly due to the use of sophisticated query processing and optimization techniques [2]. Query optimization has always been one of the most crucial components of database technology and is used to efficiently process a user's query. There has been extensive work in query optimization since the early 1970s [35] [2] [3] [4]. Query optimizers are one of the main means by which modern database systems have an improved

performance. Given a request for data manipulation or retrieval, an optimizer will choose an optimal plan for evaluating the request from among the many different possible strategies. Many optimizers for commercial database systems have been developed and the problem of query optimization has continued to receive considerable research [35] [36]. Numerous query optimization techniques have been proposed for conventional centralized or distributed relational database systems, including static versus dynamic techniques [37], sequential versus parallel strategies [38], heuristics-based versus cost-based optimization, and single-query versus multiple-query optimization [36]. As the size of the databases increase by larger amounts, the necessity for good optimizers also increases. Still, the problem of query optimization is a very much an area of open research.

1.3 Datamining

Datamining is another key area of interest in information processing and has been given a lot of focus in the last decade. Many large companies and corporations have had years of accumulation of data pertaining to the information of their products, suppliers and employees among other items.

The availability of inexpensive storage and progress in data capture technology and also the rapid pace of e-commerce techniques has paved the way for the creation of very large databases and the size of a typical database is expected to grow more in years to come. This explosive growth in data requirements has generated an urgent need for new techniques and tools that can intelligently and automatically transform the processed data into useful information and knowledge.

Datamining, also known as “*knowledge discovery in databases*”, is the efficient discovery of unknown patterns in large databases [30]. Database discovery seeks to reveal noteworthy and unrecognized associations between data items in an existing database. The potential for discovery comes from the realization that alternate contexts may reveal additional valuable information. Thus, datamining helps in finding trends and correlations that can guide strategic decision making. One possible reason for the limited success of database systems is that current systems do not have adequate features for the user interested in taking advantage of available information. Datamining is one such concept that assists in extracting valuable information from the data already present in the DB. Reasonably sufficient amounts of work have been performed for developing association rules for datamining applications [27] but very little attention has been paid towards memory requirements for storage of the database. Although good association rules exist for efficient datamining algorithms, the consumption of memory resources has been a problem, that does not receive a fair share of attention. Given a large database of customer transactions, not only should there be efficient algorithms that generate association rules to extract useful information from the available transactions, but also there should be an efficient data structure that consumes less memory, since, as the number of transactions increases, the memory requirements increase. BDDs are found to be one such data structure. They have several desirable features and are very efficient data structures for storing huge amounts of transactions using less memory. The importance of this data structure for data mining applications is outlined in chapter 3 of this thesis.

1.4 Multi-valued Query Optimization

Binary-valued logic, based on only two values of logic have been used to represent a vast amount of logic functions and in the design of logic functions, have been proven very effective for representation. Although binary-valued logic has been very useful and effective, many complex circuits cannot be represented in binary logic. This problem has created the necessity of having more numbers of logic levels greater than two and thus *Multi-valued Logic* (MVL) has become very popular and many useful commercial products are being developed with MVL [6]. MVL was proposed as a means for reducing the power, improving the speed, and increasing the packing density of VLSI circuits. Although researchers have been working on MVL for a large amount of time, the development of MVL circuits continues to be stalled by technological factors. Inexpensive MVL circuits are still not available, but a more serious problem is the absence of efficient synthesis methodologies. The synthesis techniques developed to date are not complete in some aspects [6]. In some cases, the synthesis methods that are currently used are quite expensive. In this work, a data structure for the representation of MVL networks MDD (*Multi-valued Decision Diagram*) is presented that is used to achieve the synthesis of *Multi-valued logic networks* (MVLN) and is shown that such structures are a natural extension of the techniques described for query optimization. MDDs are projected as useful data structures for query optimization in the multi valued domain.

1.5 Contribution

The work performed in this thesis presents an innovative approach of using hardware synthesis techniques for efficient information processing. Two key areas in

the area of information processing, query optimization and data mining are discussed and an efficient method has been developed for each of the issues. An MVL synthesis method has been provided and its relation to query optimization process is discussed and shown to be efficient and complete using MDDs instead BDDs.

1.6 Outline

In chapter 2, a technique for optimizing an SQL query is presented. First, a brief introduction is given about the query optimization process and relational databases. The significance of query optimization in relational database systems is outlined. Secondly, a brief introduction is given for AND/OR graphs which are crucial in the optimization technique used in this work. Finally, the query optimization process using AND/OR graphs as an intermediate data structure is explained in detail with the help of an example along with some experimental results.

In Chapter 3, a synthesis technique for MVLN is presented and MDDs are used to carry out the synthesis process. First, background about MVLNs and MDDs is presented and then the advantages of MVLNs over binary-valued networks are outlined. Secondly, the synthesis process is described in detail with the help of an example. Next, a mapping process that maps an MDD to MAX and MIN gates is described and finally, experimental results are given which underscore the significance of this mapping technique. It is shown in this chapter that MDDs are good extension for query optimization in the multi-valued domain.

In Chapter 4, a key research area of interest in the area of database systems known as “datamining” is discussed. First, concepts of datamining are discussed outlining their significance. Secondly, an efficient storage technique for datamining is

proposed and the significance and reasoning behind using this data structure for data mining applications is explained. Finally, experimental results are shown which show the effectiveness of this storage technique.

Chapter 5 includes the conclusions and some possible future research areas for the work presented in this thesis.

CHAPTER II

DD BASED QUERY OPTIMIZATION

In this Chapter, a query optimization technique is described. First, the concepts involved behind this technique are presented. Second, a brief description of AND/OR graphs is given. Third, the methodology for achieving query optimization using AND/OR graphs is described in full detail. Finally, experimental results are provided for some sample queries using this technique.

2.0 Background

In this section the concepts involved behind this technique, brief descriptions about digital logic optimization, relational databases, and AND/OR graphs is given. The problem of query optimization is discussed and the reasoning behind using AND/OR graphs for representing a query is given.

Most of the complex electronic devices seen currently are composed of digital logic circuits. These are the circuits that operate on only two fixed voltages assigned to the Boolean values '1' and '0'. Given a specific functionality, a digital function can be represented in several ways and similarly several forms of digital circuits can be designed for that specific functionality. Some of these functions are more optimal than others. The criteria of optimization can be the number of literals involved in the function. The less the number of literals, the more compact the corresponding circuit will be. A literal is an occurrence of a variable in an expression in complemented or uncomplemented form. Several methods have been developed that minimize the number of literals in a function. These methods include the "Karnaugh map reduction technique" or "cube list reduction". A minimal sum of products expression for a logic

function can be obtained using karnaugh map reduction technique. A minimal sum of products expression is equivalent to the original expression but contains minimum number of terms and minimum number of literals. The problem of finding an absolute optimum function for a given functionality has remained a challenging problem belonging to a class of problems known as *NP-Hard* problems [39]. Since it is difficult to find an absolute optimal function, several heuristics are applied to obtain a near-optimal solution and many researchers are still working on improving solutions for the problem of logic minimization.

2.1 Relational Data Bases and Query optimization

To store and access huge amounts of data, a good storage technique is needed along with an efficient technique to retrieve the information from the *DataBase (DB)*. In order to serve this purpose, *Data Base Management Systems (DBMS)* [15] were developed. A DBMS is used to access and manipulate the data in a DB easily. Typically, information is retrieved from a DB using a query. After a query is presented to a DBMS, it initiates a search over the records present and extracts information requested by the query. *Structured Query Language (SQL)* [40] is a very widely used query language. To obtain the same information from the DBMS, a query can be formulated in multiple ways. Some queries being more efficient than others depending on the way they are represented. Query optimization is the process of optimizing a query before it is presented to the DBMS.

The problem of query optimization is described here along with the basic idea behind using AND/OR graph for optimizing an SQL query. As a query can be formulated in multiple ways, the way it is presented to the DB has a tremendous

impact on the efficiency of the corresponding data retrieval. The more efficient a query is formulated, the quicker becomes the process of information retrieval for that query. A query can be represented efficiently using a data structure called an AND/OR graph as explained in [1]. Sub-graph isomorphism frequently occurs in an AND/OR graph and such a graph is reduced in size by replacing all the isomorphic sub-graphs with a single graph. This crucial property of an AND/OR graph is the main reason for representing the query in this form. Also, representation of a query in this form will allow the methods previously developed for digital logic circuit optimization to be easily applied to information retrieval from a DBMS. Several powerful CAD techniques for manipulating *Decision Diagrams* (DDs) developed over the past couple of decades can be used if an AND/OR graph is used for representing a DBMS query.

2.2 AND/OR graphs

DDs play a very prominent role in the area of design automation [7]. A considerable amount of problems can be solved if the corresponding DD's are built for the problem at hand. For some problems, a DD may not be a suitable data structure because the DD may explode in size and not be able to fit into main memory. Many problems in decision-making and *Artificial intelligence* (AI) can be classified as graph search problems. In most of these cases the aim is to determine the most efficient path from the root node to a set of leaf nodes. An OR-search has been studied extensively. In this kind of search, each outgoing edge from a node represents a possible move from some current state to another. The main difference between DDs and AND/OR graphs arises from the underlying search criteria. DDs use OR-

search, whereas AND/OR graphs are based upon AND/OR search. The difference between AND/OR searching and OR-searching can be illustrated by an example. Let us consider a puzzling situation faced by *Joe Frank* when he is trying to attend a meeting at his workplace and had experienced a flat tire. *Joe* has to attend the important meeting and now he has couple of obvious choices among which he has to decide. The way *Joe* can analyze his situation can be represented by using both an AND/OR graph and a BDD as illustrated in figure 2.1. The obvious choices that *Joe* faces are:

- a) He can ask for a ride from a person who passes by.
- b) He can repair the car on his own.

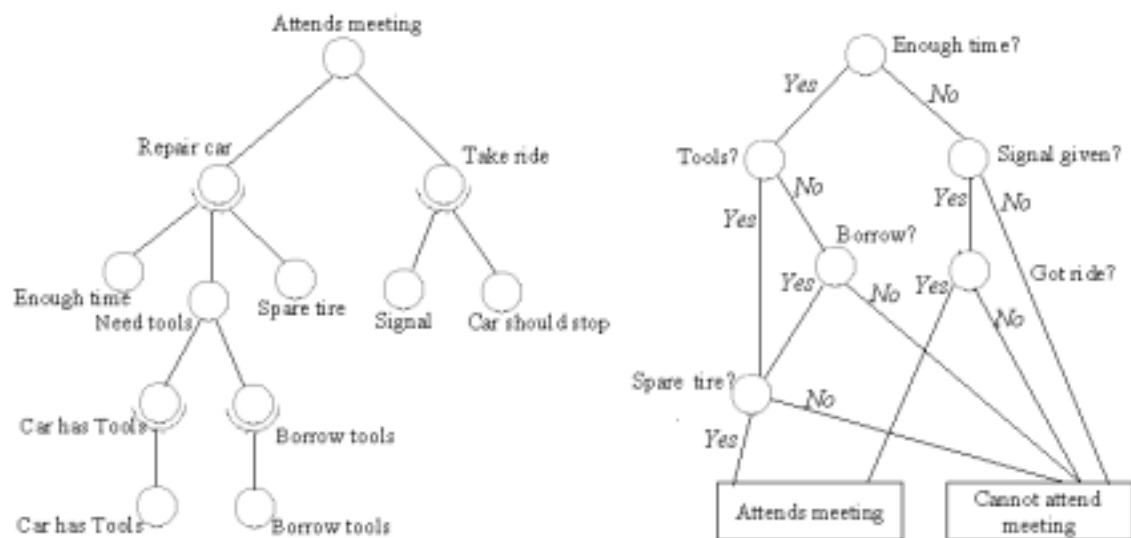


Figure 2.1 BDD and AND/OR graph

If *Joe* wants to take a ride then he has to give an indication that he needs a ride *and* someone must stop to give him a ride. This situation can be illustrated using an AND node in the AND/OR graph, as it is necessary that both the conditions be satisfied. Similarly, if he decides to repair the car then he has to have necessary tools

to repair the car, a spare tire and enough time to repair the car before his meeting. This can also be represented by an AND node. With this type of representation we can reach a conclusion that if Joe doesn't have a spare tire, he cannot repair the car even if he has the necessary tools and enough time. This is a noticeable advantage of AND/OR representation over other types of decision diagrams, since a conclusion can be reached even before the entire graph is traversed. Other nodes in the AND/OR graph in Figure 2.1 are self-explanatory. This AND/OR graph is built with an assumption that Joe will be able to attend the meeting. Similarly a BDD can be built for a similar situation as shown in Figure 2.1. More than one BDD can be built for a single function if the order of variables is varied. If the order of variables is varied in a BDD, the size and shape of the resulting BDD will vary but not the functionality. So, for the situation described above, more than one BDD can be built depending on the condition of the root node. One such possible BDD is shown in Figure 2.1 and we can see from the BDD that we cannot reach a conclusion until we traverse the BDD all the way through to the terminal nodes. If we begin the search process at the root node, then according to the BDD shown in Figure 2.1, if *Joe* has enough time to repair the car then he can go a step further and think about the possible requirements necessary to repair the car. If he does not have enough time then he has to forget about repairing the car and use some other means to get to the meeting (i.e. taking a ride). The rest of the nodes in the graph are self-explanatory but an important thing to be noticed is that each node has two out-going edges and only one, not both, of the two edges is activated while traversing the BDD. This kind of searching technique used while traversing BDD is an OR-searching technique because only one of the two

possibilities can occur depending on whether or not the current condition is satisfied. OR-search has proven useful for many applications especially in the field of AI. By using an AND/OR search, logical consequences for a set of given assumptions can be obtained. AND/OR graphs have also shown some promise in the area of design automation and some of the problems that exist if DDs are used can be overcome if AND/OR graphs are used as an alternative data structure. AND/OR graphs are developed due to the need of an efficient underlying data structure to solve many CAD problems. The problem of Boolean satisfiability is a very significant problem in the area of design automation. It is the problem of finding whether or not a function can assume a logic value **1**. The satisfiability problem can be formulated by an AND/OR graph provided the graph is built according to the recursive learning technique as described in [8]. Every Boolean expression can be represented as an AND/OR graph. AND/OR graphs are also very useful in performing Boolean reasoning for multi-level circuits. Another interesting feature of AND/OR graphs is that it is sufficient to enumerate an AND/OR graph only partially to obtain some crucial information whereas very little information can be obtained from a partially constructed DD. AND/OR graphs can be better understood with the help of an example. Below is an example of building an AND/OR graph from an arbitrary Boolean function.

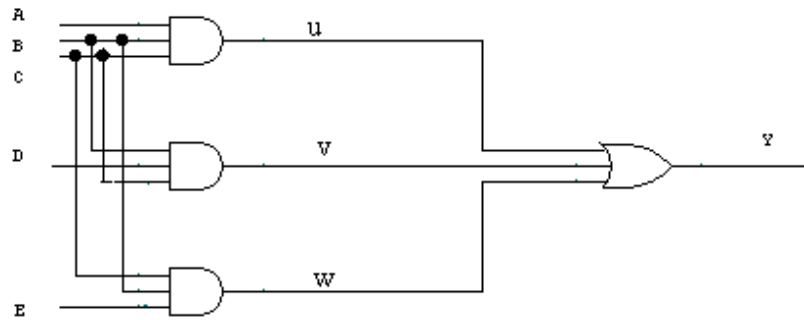


Figure 2.2 Digital circuit for the expression $y=abc+bcd+bce$

Figure 2.2 represents a two level combinational digital circuit and figure 2.3 represents corresponding AND/OR graph for the implication $y=0$. The AND nodes are represented using an arc below a circle to distinguish it from an OR node. The Boolean function considered in this example is $y = abc+bcd+bce$. If we consider the implication $y=0$ (generally AND/OR graphs represent the implication that $f=0$ where f is a Boolean function) then there are 3 conditions that are necessarily to be satisfied in order for this condition to be true and are given by $u=0$, $v=0$, $w=0$. So an AND node is used to represent $y=0$ as all the three conditions are mandatory. Similarly, for $u=0$, it is sufficient if either a or b or c is zero and an OR node is used to realize this condition and the circuit is traversed in this fashion from the output to the inputs and the corresponding AND/OR graph is developed considering the implications for each node. This AND/OR graph can be further reduced without any loss of information. The reduced AND/OR graph is shown in Figure 2.4 and it contains a fewer number of nodes than the original AND/OR graph but functionally it is equivalent to the original AND/OR graph.

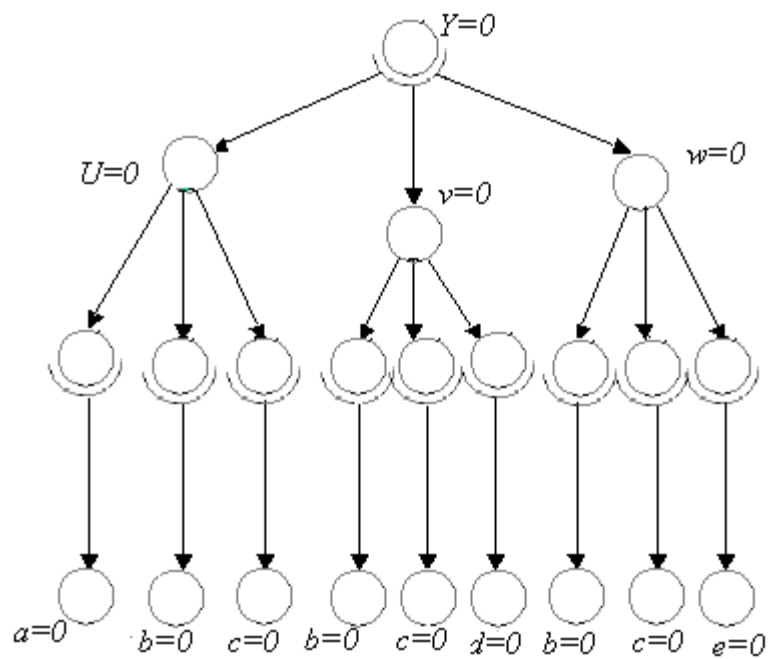


Figure 2.3 AND/OR graph for the above digital circuit

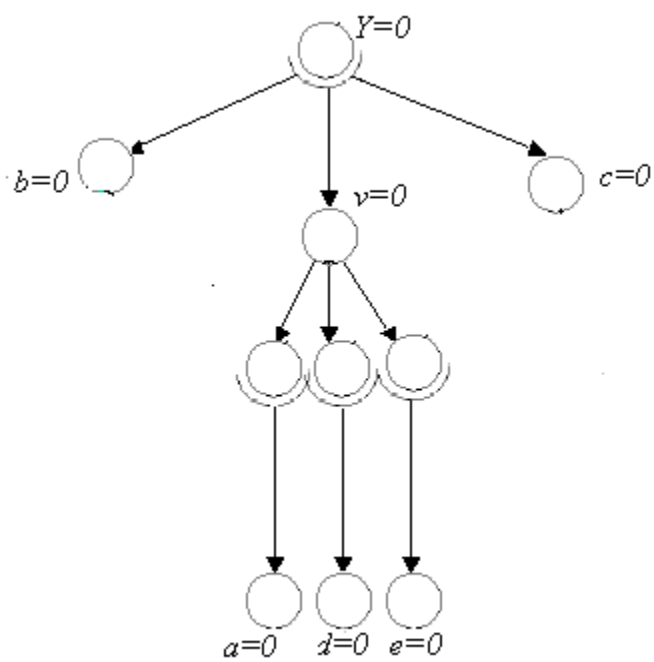


Figure 2.4 Reduced AND/OR graph

Some of the necessary conditions to be taken care of while building an AND/OR graph are:

- a) There should be a single root node
- b) AND and OR nodes occur in alternate levels (i.e. no two AND nodes or OR nodes can be in adjacent levels).
- c) The leaf nodes are always OR nodes.

2.3 Query Optimization using AND/OR graphs

AND/OR graphs are efficient for representing a DB query, the advantage being that most of the logic minimization techniques developed for AND/OR graphs can be applied to the DB query. The experimental set up which is used to obtain query optimization using AND/OR graphs as an intermediate data structure is shown below in figure 2.5.

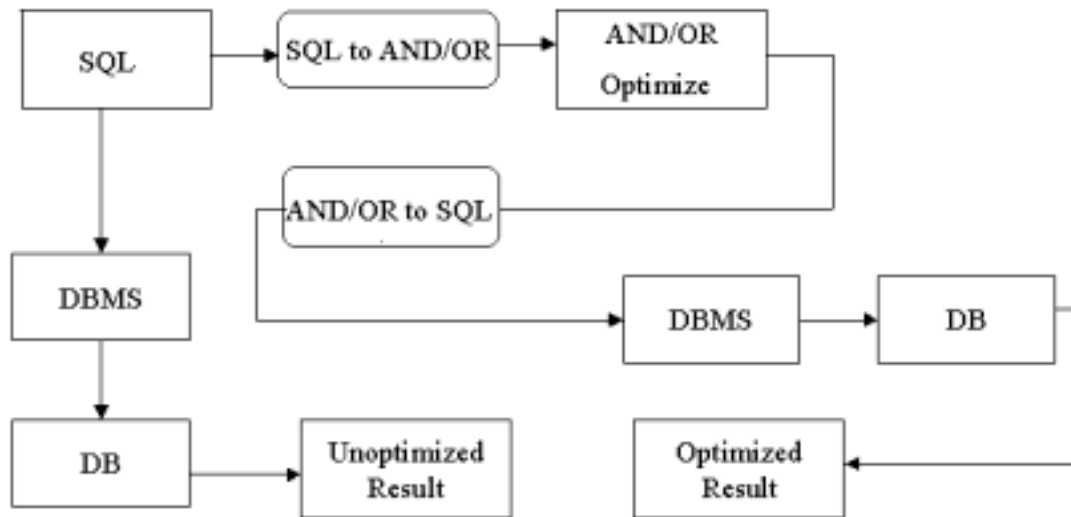


Figure 2.5 Experimental setup used for optimizing query in this approach

The query which is initiated by the user for the purpose of retrieving information from the database is converted into an AND/OR graph. This part is

achieved by interfacing an SQL parser with the AND/OR package, the intermediate step being generation of a digital circuit represented by a Berkeley Logic Interchange Format (BLIF) file.

A BLIF model is developed to describe logic level hierarchical circuit in textual form and this is a very useful representation even for complex circuits. The skeleton of a BLIF file can be outlined as below.

.model < *model_name* >

.inputs < *input_list* >

.outputs < *output list* >

.names < *operation* >

.names < *operation* >

.end

“*.model*” is used to specify a name for a model. This is mainly useful in identifying different models and nesting different models. This is an optional statement. If “*.model*” line is not specified then the model name is assigned to the name of the BLIF file.

“*.inputs*” is used to specify names of the inputs. These input names are the inputs for the model being considered. Each input is separated by a blank space and if multiple input lists are written then all those inputs are concatenated.

“*.outputs*” is used to specify names of the outputs. These output names are the outputs for the model being considered. Each output is separated by a blank space and if multiple output lists are written then all those outputs are concatenated.

“*.names*” is used to specify an operation performed on inputs. Temporary signals can be added in the “*.names*” statement. Most of the functionality of the circuit can be expressed in this part of the model.

“*.end*” statement is used to signal the end of the model. This is always used to indicate that the model description is complete.

There are several other statements used in BLIF files that are not covered in the skeleton shown above but above statements are enough to completely represent a circuit. Figure 2.7 shows an example of a BLIF file for the circuit shown in figure 2.6.

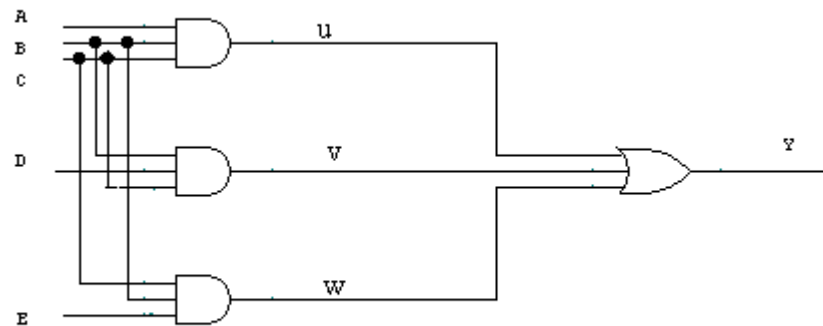


Figure 2.6 circuit model for the BLIF file

```
.models digcircuit
inputs a b c d e
outputs y
.names a b c u # u is an intermediate signal and '#' is used to write a comment
111 1
.names b d e v
111 1
.names b c e w
111 1
.names u v w y
1 - 1
-1 - 1
--1 1
.end
```

Figure 2.7 BLIF file for the digital circuit shown in figure 2.6

Each *.names* statement in the figure 2.7 corresponds to a gate in the actual circuit. So for each of the four gates in the circuit there is a “*.names* “ statement and it can be noticed that there are some temporary signals generated such as *u*, *v* and *w* as per the necessity. A *.end* statement is used to signal the end of the description of the circuit.

After the generation of BLIF file, an AND/OR graph is built corresponding to the circuit represented in the BLIF file. An AND/OR package [10] is used to build an AND/OR graph from a BLIF file, and this package replaces all the isomorphic sub graphs with a single graph at the time of constructing the AND/OR graph from the query. Thus all the redundancy carried over by the query will be removed. Thus AND/OR package generates a reduced AND/OR graph which is transformed back into an SQL query which will be of reduced length when compared to the length of the original query in most cases. The query obtained at this point is an optimized query and this query is presented to the DBMS to retrieve the desired information, the whole of optimization process takes place before the query is presented to the DB.

The example to follow will emphasize the process of query optimization even better. Consider the following SQL query :

Select pname from project where (Pteam = CAD and Pcity = Starkville and P#=P1006) or (Pcity = Starkville and EmpName = John and Pteam = CAD) or (Pteam = CAD and Pleader = Chris and Pcity = Starkville);

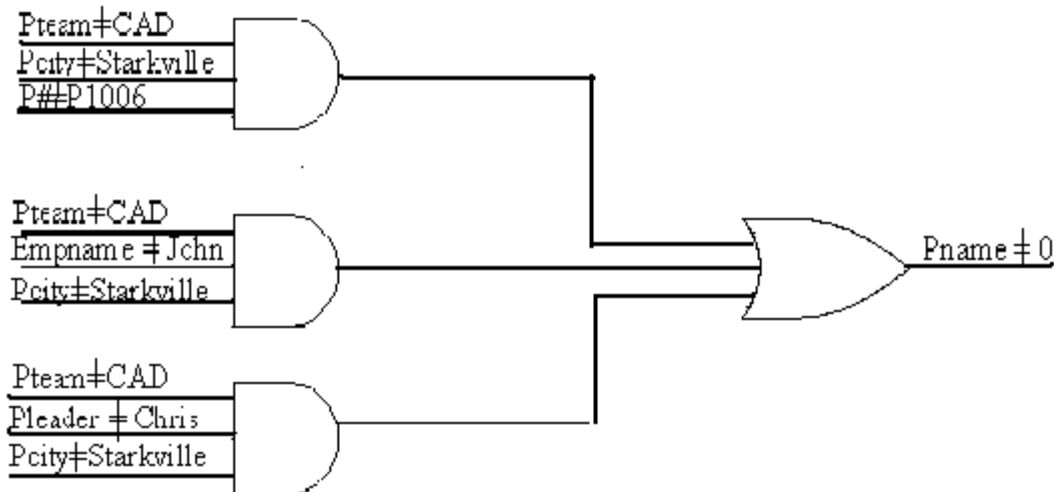


Figure 2.8 Digital circuit for the example query

```

.model  examplequery
.inputs  Pteam=CAD Pcity=Starkville
.inputs  P#=P1006 Empname=John Pleader=Chris
.outputs Pname
.names  Pteam=CAD Pcity=Starkville P#=P1006 I1
111 1
.names  Pteam=CAD Pcity=Starkville Empname=John I2
111 1
.names  Pteam=CAD Pcity=Starkville Pleader=Chris I3
111 1
.names  I1 I2 I3 Pname
1 - - 1
-1 - 1
- - 1 1
.end

```

Figure 2.9 Digital circuit for the example query

This query is converted into a digital circuit, which is represented as a BLIF file in the actual implementation of this technique. This circuit represents the same query but in a different form. The digital circuit for the above query is shown in Figure 2.8 and the corresponding BLIF file is shown in Figure 2.9. This circuit represents a case of a query failure ($Pname=0$). At this stage optimizing the query can be thought of as optimizing the corresponding digital logic circuit, so, the problem of

query optimization can now be considered as the problem of digital logic optimization. As it is a very difficult to find out an absolute optimal digital function, finding an absolute optimal query is also a very hard problem but close optimal solution can be reached in many cases. An AND/OR graph is generated corresponding to this digital circuit and this graph is shown in Figure 2.10. This graph also represents the case of a query failure. This graph generated from the given query contains fair amount of redundant information which increases the execution time of the query.

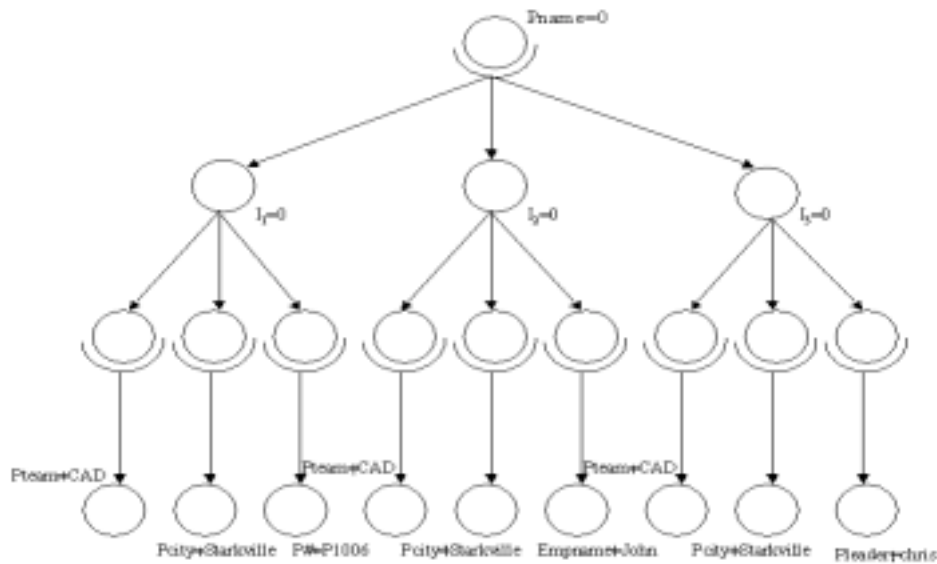


Figure 2.10: AND/OR graph of example query

All the redundant nodes in the AND/OR graph are removed in the AND/OR package which results in a reduced AND/OR graph. The reduced graph is shown in Figure 2.11. This reduced graph is functionally equivalent to the original AND/OR graph i.e. it carries the same information as the original graph. Another advantage of such a graph is it is easy to predict the failure of a query only by partially building the

graph, for example, if we consider the reduced graph in Figure 2.11 then only by knowing that **Pname** is not equal to 'CAD' (or by knowing that **pcity** is not equal to 'Starkville') the query can be considered as a failure. This may prove very useful when large queries are considered because the processing time taken for querying the database can be reduced if the failure of the query is known early.

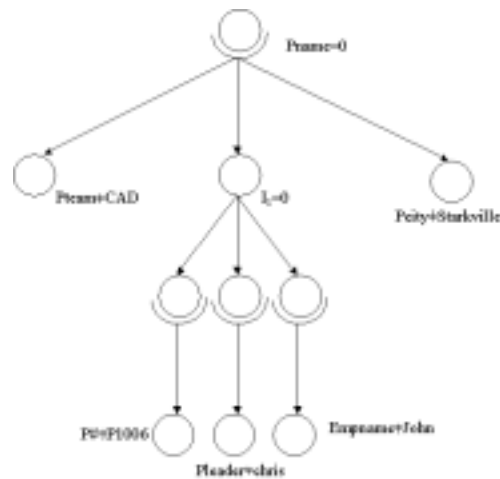


Figure 2.11 : Reduced AND/OR graph for the example query

After the reduced AND/OR graph is obtained the next step is to obtain the query back from the AND/OR graph, again the intermediate step being the digital circuit. The digital circuit corresponding to the reduced AND/OR graph is shown in Figure 2.12

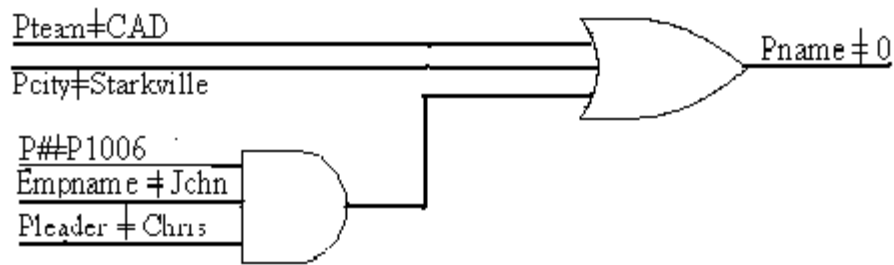


Figure 2.12 digital circuit for the reduced AND/OR graph

The query can be reconstructed from the above circuit but it represents the cases for the failure of the query rather than its success, since we are interested in the success of the query the above circuit should be modified in some way so that the circuit represents the output $Pname = 1$ instead of representing $Pname = 0$. This can be achieved by placing inverters at the outputs and at all the inputs and the circuit obtained by doing such a kind of a translation is shown in Figure 2.13. Circuit in Figure 2.13 is further reduced resulting in the circuit shown in Figure 2.14.

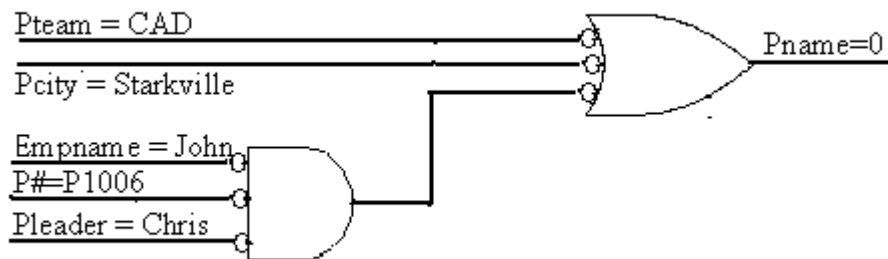


Figure 2.13 Circuit negated to represent success of the query

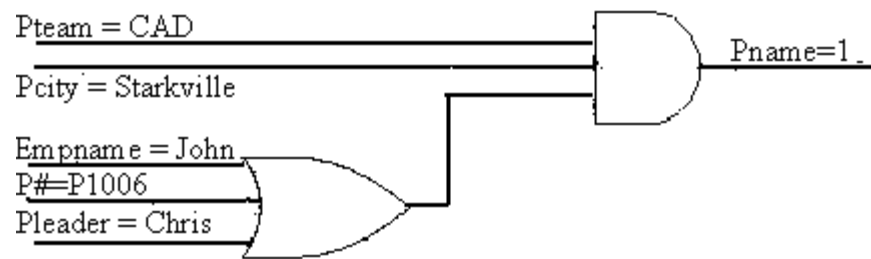


Figure 2.14 circuit representing successful query

The query is reconstructed back from the circuit in Figure 2.9, doing so yields the following query:

Select Pname from project where Pteam = CAD and Pcity=Starkville and (P#=P1006 or Empname=John or Pleader=Chris);

The length of the query obtained after optimizing it is much smaller than the original query. The example discussed above is a trivial example but it emphasizes the use of AND/OR graphs in reducing the length of the query. Most of the redundant information carried over by the query will be discarded if it is represented by an AND/OR graph.

2.4 Experimental Results

The original queries and queries optimized using AND/OR graphs are executed using ORACLE DBMS and the execution times taken by these two queries are compared. The results obtained on doing so are tabulated in table 2.1

Table 2.1 Experimental results

Query	Orig	AND/OR	Opt.Query	Total
1	0.29	0.04	0.09	0.13
2	0.10	0.02	0.05	0.07
3	0.08	0.03	0.04	0.07
4	0.12	0.03	0.03	0.06
5	0.13	0.02	0.09	0.11
6	0.09	0.03	0.04	0.07
7	0.09	0.03	0.03	0.06
8	0.36	0.09	0.13	0.22

“Orig” is the execution time taken by the original query, “AND/OR” is the time taken for building an AND/OR graph for the query in the AND/OR package, “opt. query” is the time taken by the DBMS to process the optimized query and the total time taken for the whole optimization process is given by “total”. All results are given in seconds of CPU runtime. These results indicate that this technique is very effective for some queries. In almost all the queries tested the execution time taken by the optimized query is either equal to or less than original query but not more.

2.5 Conclusion and future work

A method is introduced for optimizing an SQL query with the use of AND/OR graphs as an intermediate data structure. The experiments indicate that in most of the cases the execution time of the query is optimized at least by 30%. In almost all the queries tested so far the query obtained by the reduced AND/OR graph will have an execution time either equal to or lesser than the execution time of the original query but not more than the original query. The next possible step in this area would be to pass set of queries simultaneously to AND/OR package and identify

subgraph isomorphism between the queries. This would be an interquery optimization where as we examined only intra-query optimization in this paper.

All the sets of queries cannot be optimized using this technique. Only some subset of the queries can be optimized because the technique examined here works well only in Boolean space. So the next logical extension would be to optimize multivalued queries. This can be done by using Multivalued Decision Diagrams(MDD) to represent the query instead of an AND/OR graph.

CHAPTER III

MULTI VALUED QUERY OPTIMIZATION

In Chapter 2, an optimization technique for SQL queries using AND/OR graphs is described in which query optimization is possible for only queries that operate in the Boolean space. A logical extension of this technique is to develop an optimizer for a multi-valued query and multi-valued query optimization can be achieved in a similar way when a query is mapped to a *Multi-valued Decision Diagram* (MDD) instead of an AND/OR graph. AND/OR graphs operate in the Boolean domain whereas MDDs work in multi-valued domain. An intermediate step in generating an MDD from an SQL query would be representing the query as a *Multi-Valued Logic Network* (MVLN). There should be an efficient synthesis technique for the synthesis of MVLNs in order to physically realize the multi valued query optimization process. In this chapter, an efficient synthesis technique for synthesizing MVLNs is described using logic gates called MIN and MAX gates and then process of multi valued query optimization is explained. Section 3.0 gives an introduction about Multi-valued logic and it's advantages over binary logic. Section 3.1 gives background information about MVLNs and MDDs and section 3.2 describes the methodology used for synthesizing MVLNs using MDDs, the need for multi-valued query optimization using MDDs and it's possible outcome is given in section 3.3 and section 3.4 gives the experimental results for the MVLN synthesis using MDD and the chapter is concluded with a conclusion in section 3.5.

3.0 Introduction

Logic design has been mostly thought of in terms of binary signals, while design using binary signals has proved very useful and effective in many applications, it is difficult to implement more complex designs using binary valued logic, for complex designs, variables with symbolic values are desired. In the recent years, *Multiple Valued Logic* (MVL) [6] has played an increasing role in the evolution of many commercial products and the use of MVL has made many designs simpler and efficient. MVL has been basically proposed as a means for reducing the power, improving the speed, and also increasing the packing density of VLSI circuits.

Decision Diagrams (DDs) are the state of the art data structures for design automation problems. DD's have been successfully applied to solve many problems in the area of *Computer Aided Design* (CAD) and DD's are becoming extremely popular in the area of design automation. Boolean functions are generally represented using *Binary Decision Diagrams* (BDDs) as many functions can be represented in a very compact form using BDD's and many logic optimization and manipulation techniques can be performed on Boolean function when represented using BDDs. Boolean functions are the functions which satisfy the relation $f: \{0,1\}^n \rightarrow \{0,1\}$. Every node in a BDD has two outgoing edges and either of the edges can be active. A data structure known as Multivalued Decision Diagram (MDD) has been developed to represent Multivalued functions. Multivalued functions satisfy the relation $f: \{0,1,2,...,k-1\}^n \rightarrow \{0,1,2,...,k-1\}$. Most of the efficient algorithms that are efficiently used to manipulate a BDD can be used to manipulate an MDD.

In this chapter, synthesis of MVLN's using MDD's is examined. Every DD has to be mapped to a target architecture and several approaches to achieve this kind of mapping have been proposed, for example 2-input multiplexers can be used to represent every node in a BDD as only one of the edges emerging out from the node is active. MIN and MAX gates are used for the mapping process for synthesis of MVLNs.

3.1 Background

In this section information about MVLN's and MDD's is given and they are explained in detail with specific examples.

3.1.1 Multi Valued Logic Networks

In a MVLN, each primary input may take the values from the set $\{0,1,...k-1\}$ where k denotes the number of logic levels and so is the case with each primary output. A primary input node does not have any incoming edges and a primary output node does not have any outgoing edges. The basic cells used in an MVLN are MIN, MAX, INV and LITERAL. MIN gate in MVL corresponds to AND gate in binary logic, MAX gate corresponds to OR gate and INV corresponds to NOT gate. An MVLN is generally a directed acyclic graph and every vertex in this graph is labeled by one of these basic cells.

3.1.2 Multi Valued Decision Diagrams

An MDD is developed as an efficient data structure to represent multi-valued functions. An MDD can be built from an MVLN by traversing it in topological order. Each node in an MDD has k outgoing edges where k denotes the number of permissible logic values. Size of an MDD depends on the order of the multi-valued

variables used to build the MDD. A node in an MDD is considered to be redundant if it has all the successors pointing to the same node and another type of redundancy occurs when an MDD has isomorphic sub graphs. An MDD devoid of these kind of redundancies is termed as a reduced MDD. The example to follow explains the behavioral nature of an MDD.

Consider a two valued three variable function with the following truth table:

Table 3.1 Truth table of a multi-valued function

x1	x2	x3
0	0	0
0	1	1
0	2	0
1	0	1
1	1	1
1	2	2
2	0	1
2	1	2
2	2	2

An MDD can be constructed for the above multi valued function and the MDD obtained by doing so is shown in figure 1. From the figure it can be observed that each internal node in an MDD has 3 outgoing edges (equal to number of logic values).

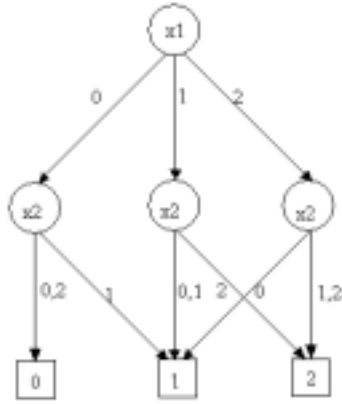


Figure 3.1 MDD for function with truth table in Table 3.1

3.2 Methodology

A mapping method that maps every edge of an MDD onto a set of MAX and MIN gates is described in this section. For this mapping process an assumption is made that an MDD representing a k -valued function is given initially. Another assumption made is that each characteristic function is available for each primary input. The characteristic function is given by the set of $J_j(x_i)$ values such that $J_j(x_i) = k-1$ if $x_i=j$ and $J_j(x_i) = 0$ otherwise. Given an MDD each edge in it is translated into a set of MAX and MIN gates in the following way:

- a) The characteristic functions of input x_i drive one MAX gate and another MAX gate is driven by predecessor edge values.
- b) The outputs of these two MAX gates serve as inputs to a MIN gate which produces the final value for that edge.

Mapping an MDD in this way generates a circuit with a size proportional to the size of an MDD, the property which can be used for several optimization

techniques. The above mapping process is explained in more detail with the help of an example. Consider the following simple MDD shown in Figure 3.2.

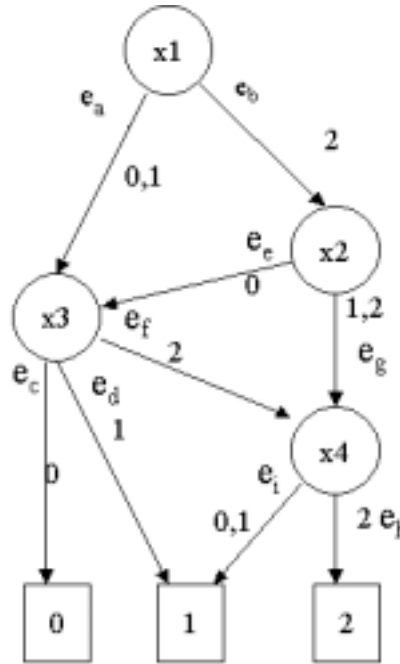


Figure 3.2 : example MDD to show mapping process

Each edge emerging from a node of an MDD is mapped to sets of MIN and MAX gates. Figure 3.3, Figure 3.4, Figure 3.5, Figure 3.6, Figure 3.7, Figure 3.8, Figure 3.9, show the steps involved in mapping process. Figure 3.10 shows the synthesized circuit for all the edges in MDD. If there are any MAX or MIN gates with a single input then that gate is redundant and is removed.

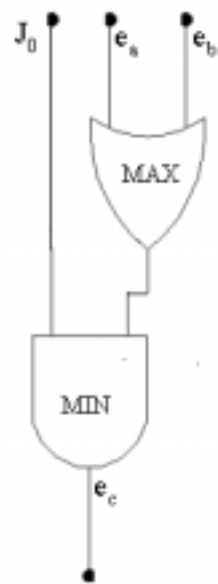


Figure 3.3 Mapping for edge e_c

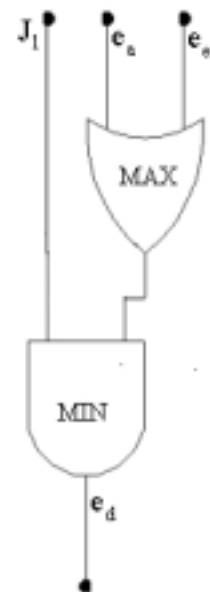


Figure 3.4 Mapping for edge e_d

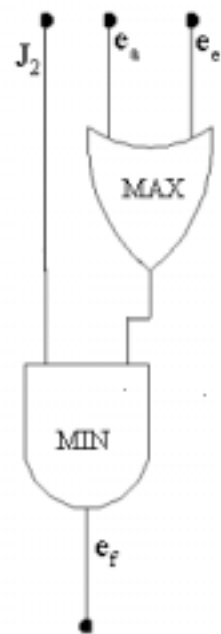


Figure 3.5 Mapping for edge e_f

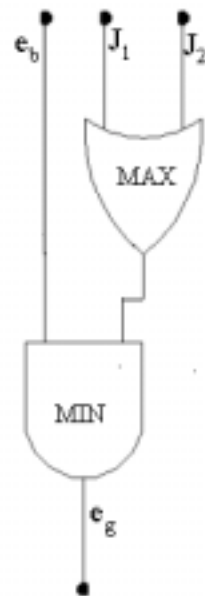


Figure 3.6 Mapping for edge e_g



Figure 3.7 Mapping for edge e_e

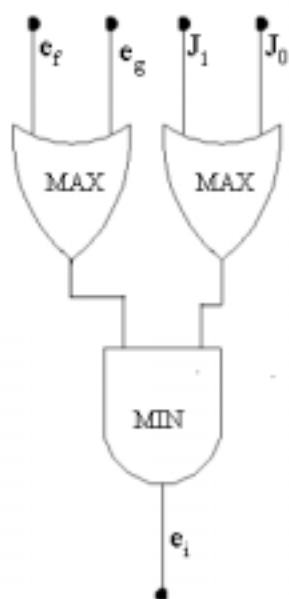


Figure 3.8 Mapping for edge e_i

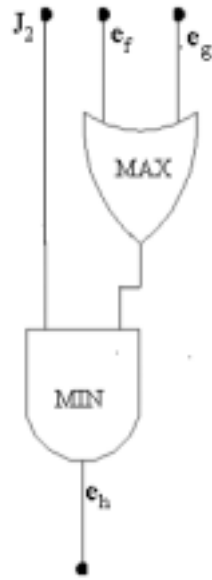


Figure 3.9 Mapping for edge e_i

3.3 Multi valued query optimization

In chapter 2, it was explained that a query can be optimized by representing it using an AND/OR graph. Differences between BDDs and AND/OR graphs are also explained with the help of an example, so, a query can also be optimized by using a BDD as an intermediate data structure instead of an AND/OR graph and several BDD circuit optimization techniques can be used for optimizing SQL queries if they are represented using a BDD. Both BDDs and AND/OR graphs work in the Boolean space and to make the query optimization process more complete and practically useful there should be a data structure which operates in multi-valued domain. An MDD is one such data structure which operates in multi-valued domain.

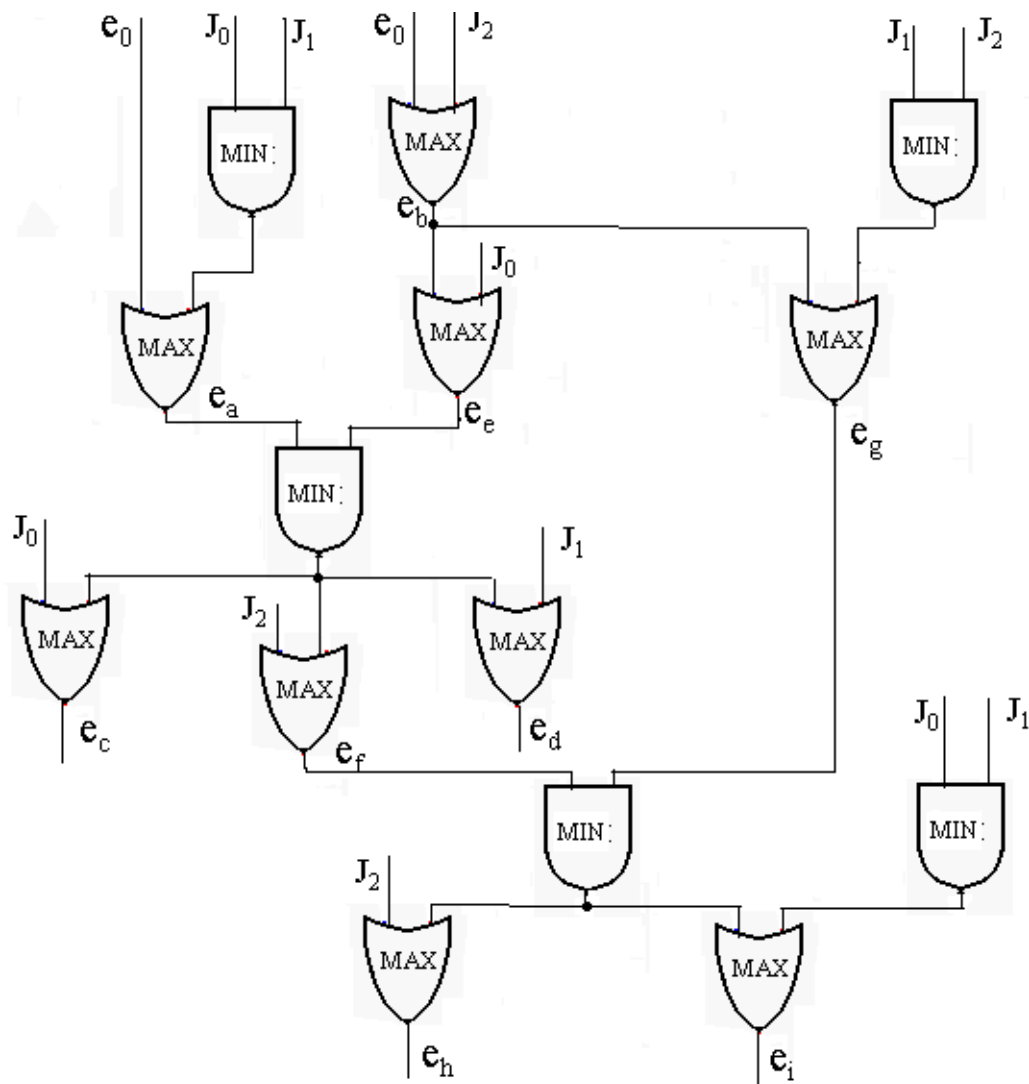


Figure 3.10 Mapping for all edges in example MDD

MDDs are generalizations of BDDs in the multiple valued domain. MDDs often allow efficient representation of functions with multi valued input variables similar to BDDs in the binary case. All the optimization techniques that were developed for optimization of BDDs over the past few decades can be used even for MDDs and thus if an SQL query is represented using an MDD then several rich optimization techniques can be used to optimize the query and recently several new

MVL synthesis and MVL optimization tools are appearing and all these newly appearing tools certainly contribute to the query optimization process to make the process of optimization more efficient if query is represented in this way. Query optimization using multi valued logic has not been implemented in this work but strong and positive results obtained after representing the query as an AND/OR graph in this work is a sure indication of the possible success of query optimization in multi valued domain. This will be the future area of research in the query optimization process.

3.4 Experimental results

Table 4.1 shows the experimental results after using this mapping technique for several benchmark circuits. The number of MIN and MAX gates required for mapping the benchmark circuits using this mapping technique are given. Further reductions are possible and so the number of MIN and MAX gates given here are upper bounds. The number of logic levels used are also given for the benchmark circuits along with the number of cubes in the file.

The benchmark files shown in the experimental results are binary files containing binary list of cubes, which are mapped to corresponding MVL cubes by pairing of variables. The MDD package used to build MDD's from ".pla" or ".ml" files can currently build MDD's only for multi valued cubes obtained from mapping binary cubes to multi valued cubes.

Table 3.2 Experimental Results for benchmark circuits

<i>S.NO</i>	<i>CIRCUIT</i>	<i>Logic Levels</i>	<i>MIN</i>	<i>MAX</i>	<i>#cubes</i>
1	Z5xpl.pla	4	125	157	127
2	Test1.pla	3	15	9	16
3	Tial.pla	4	2086	5447	640
4	Ts10.pla	3	523	357	127
5	In2.pla	4	857	1762	137
6	Risc.pla	3	297	276	73
7	Add6.pla	4	4648	9337	1091
8	Alu2.pla	3	166	246	86
9	C8.pla	4	183	327	173

3.5 Conclusion

An edge mapping synthesis method for generating netlist for multi valued logic networks is examined and the significant property of this kind of mapping is that the resulting circuit size is proportional to the number of nodes in an MDD. It has been shown that SQL query optimization in multi-valued domain can be achieved by using MDD as an intermediate data structure.

CHAPTER IV

EFFECIENT STORAGE TECHNIQUE FOR DATAMINING APPLICATIONS

4.0 Background

The computerization of many business and government transactions has resulted in huge amounts of useful data to be stored in a database. There are millions of databases used today in almost every organization, and each database contains a huge collection of data that, if analyzed properly, will reveal very important information. This may be a good starting point for future ventures. One of the main reasons for the limited success of the database systems in this area is that the current database systems do not provide necessary functionality for a user interested in taking advantage of the information available to him or her. Researchers have been successful in building powerful and affordable database systems, but there is still an urgent need to develop new techniques for intelligently and automatically transforming the processed data into useful information.

The size of a typical database has been increasing at a tremendous pace. Datamining essentially involves discovering patterns from the database and inferring certain rules from the discovered patterns. These rules are called *Association Rules*. Discovering association rules is very much an important part of datamining. One has to search the database for patterns and infer certain rules from the discovered patterns. This forms the crux of the datamining problem.

The main concentration of researchers in this area of datamining has been revolving around association rules for data mining. Numerous algorithms for association rules were developed but the issue of memory requirement for storing the

data, which is a crucial issue, has been overshadowed. Memory requirement is a very crucial issue because of the multifold increase in the sizes of the databases of late. Given a large database of customer transactions, not only there should be efficient algorithms that generate association rules but there should also be an efficient data structure that consumes less memory and has desirable properties to retrieve information easily, since the number of transactions increases as the memory requirements increase. This chapter presents a data structure that can be used to achieve amazing reduction in memory requirements for storing a data file. We also present results for storage requirements for a sample sales data file showing the effectiveness of this data structure.

4.1 Introduction

Of late, there has been an upward trend in the size of the databases, the reason being the accumulation of data. At the same time, there have been technological advances in leaps and bounds in the field of computer hardware components and in the field of computer networks. One can make use of these technological advances to maximize the efficiency of extracting the right kind of data from the database.

Section 4.2 gives the methodology and experimental results are provided in Section 4.3 and finally the chapter is concluded in section 4.4.

4.2 Methodology

A storage technique is developed to store transactions of a data file in a BDD and this technique can currently be applied for “**Market Basket Analyses**”. A *market basket* is a collection of items purchased in an individual transaction. For example, in a

case where a customer visits a grocery store, all the items he has purchased in that particular outing to the grocery store come under one transaction. All these kind of transactions are accumulated into a transaction data file where each transaction contains the item-id of the item that has been purchased. One common analysis that can be run over transactions is the concept of an *itemset*. Itemsets are set of items that appear together in many transactions.

If we consider a transaction data file then the ideal data structure that is used to store the items in the data file should have the following properties:

- i) There should be a hash key for each entry of a transaction to retrieve the transaction quickly by using some kind of a hash function.
- ii) There should be an easy comparison method between two transactions so that same hash key can be used for identical transactions (i.e. sharing of similar transactions should be possible).
- iii) Finally, the data structure should take a minimum of memory resources, which is really crucial because transaction data files consume a lot of memory resources.

BDD data structure satisfies all the requirements and can be considered to be a very good data structure for datamining application. Some of the important features that are useful for the storage of datamining applications are listed below.

- i) Every node in a BDD has a unique id that can be used as a hash key.
- ii) Sharing of similar transactions can be easily achieved using a BDD because BDD is built using a hash table and if a similar transaction to an

already existing transaction is found its pointer value is made equal to the pointer value of the already existing transaction. This is a very useful feature as there is a high probability that transactions can be similar in a transaction data file and there will be wastage of memory and thus slower execution speed if these identical transactions are stored without sharing.

- iii) BDD uses minimum amount of memory as all the similar transactions are shared and there is even sharing of sub-transactions apart from sharing whole of the transactions.

Thus, BDD happens to be a very good storage data structure for datamining applications and another useful feature is that many manipulations can be performed on the data stored in a BDD using BDD manipulation techniques developed over the past few decades.. This may be very useful for future work on datamining for efficient manipulation of transactions.

4.2.1 Methodology process: BDD package is used to build BDD's for different transactions from a transaction file. A BDD is built for every transaction and all these BDD's for several transactions are managed by a single DD manager. If there are two transactions which are somewhat similar, then the DD manager builds the BDD's in such a way that most of the nodes are shared between those transactions, as the number of transactions increase, the number of nodes shared between different BDD's for different transactions increase, so on the whole, there is reduction in the usage of memory because there is huge amount of sharing of nodes. Another very important feature is that there is a unique pointer pointing to the root

node of every BDD representing a transaction. This is especially crucial for datamining applications because this can be used as a hash key to identify the transaction.

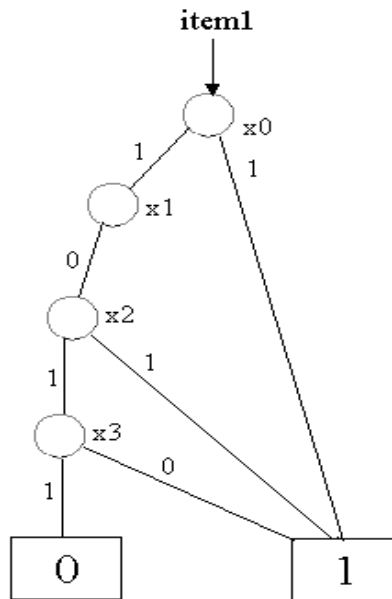


Figure 4.1 : BDD for item '13'

Let us consider a transaction in a transaction data file with the following items in a single transaction, “13 3 1311 10 15” where each number represents an item number that has been purchased in that particular transaction. For example, “13” is a code for a specific item in the store and presence of this item number in a transaction indicates that it is one of the items that has been purchased by the customer.

First, a BDD is built for each of the items in a transaction. Item number ‘13’ can be represented in binary as “1011” and this can be expressed as a logic function given by x_0, x_1, x_2, x_3 (x_2 represents the complemented form of x_2). So, this item can

be represented as a Boolean function of four variables and the BDD that represents such a logic function is shown in figure 2.1

Similarly a BDD can be built for the second item in the transaction, which is '3'. The Boolean function representing this item is given by x_0, x_1, x_2, x_3 and the corresponding BDD for this function is shown in figure 2.2.

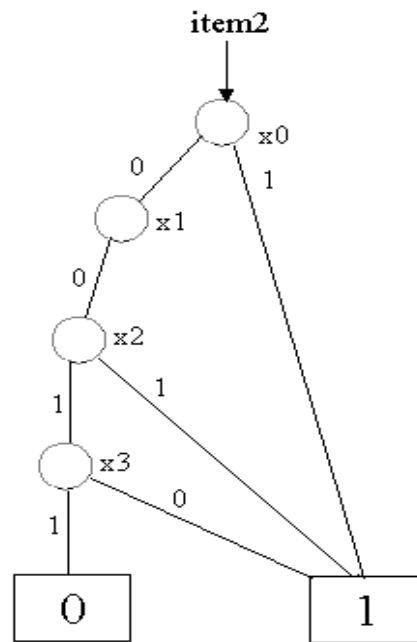


Figure 4.2 : BDD for item '3'

It can be easily observed that *item1* and *item2* share some similar nodes and have some similar structure, the only difference being that the complemented and uncomplemented edges of x_0 are reversed in both the cases. BDD's for these two items can be OR'ed together to form a reduced BDD, which is shown in figure 2.3. It can be easily observed that the BDD obtained when the BDD's of *item1* and *item2* are combined is a much compact BDD and needs storage space for only 5 nodes when compared to the storage space of 8 nodes needed by *item1* and *item2*. This is a significant improvement in storage. As the number of items in the transaction file

increase, the number of shared nodes also increase and more compact will be the size of the resulting BDD.

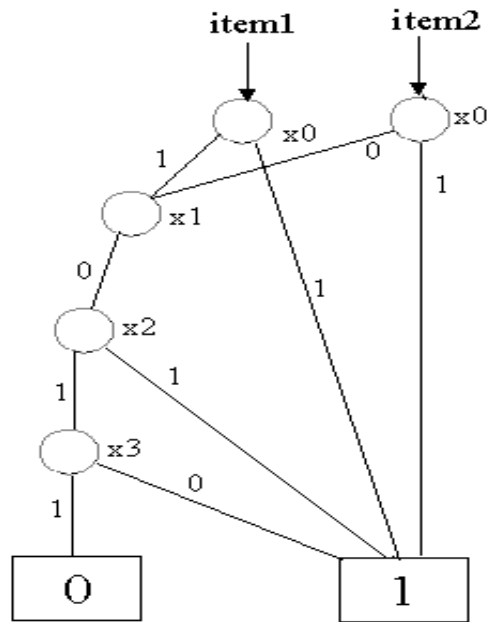


Figure 4.3 BDD for item1 and item2

Another interesting feature in this process is that BDD corresponding to each item can be identified with a unique pointer that points to the specific item and each item can be easily retrieved using the pointer value. The same process can be used for building the BDDs for other items in the transaction. BDDs for all the transactions in the file are built using the same DD manager, so sharing of nodes takes place not only within a single transaction but also with in sets of transactions. In this way almost all the redundancy involved in storing unnecessary data will be eliminated and as the number of transactions in a transaction file increase, the more reduction in storage space could be achieved as there is a fair possibility that many similar transactions are possible. By using this approach there won't be any duplication of memory resources as all similar transactions can be represented by a single BDD with each item

recognized by its corresponding pointer value. The experimental results obtained by using this approach are shown below.

4.3 Experimental Results

Table 2.1 shows the experimental results of the usage of memory for several data files and when the comparison is made between the amount of storage space used for the actual data file and the storage space used when BDD is used to store the items, There is a huge amount of reduction in the memory when BDD is used as a data structure.

Table 4.1: Results

File Size	No: of transactions	No: of BDD nodes (Size)
13MB	100000	8153 (127.4 KB)
7MB	64000	8323 (130.1 KB)
9MB	84000	11253 (175.8 KB)

4.4 Conclusion

An efficient data structure for storing huge transaction files is proposed and experimental results indicate that it is highly efficient. If BDD's are used to represent transactions then not only there is a good reduction in the usage of memory but also powerful manipulation techniques can be used. This is a very desirable feature in the area of datamining. Future work in this direction can include manipulation of transaction data files using BDD's. Several manipulation algorithms can possibly be performed on transactions to extract valuable information. Furthermore, and since BDD has been used as a versatile data structure over the past couple of decades in the area of computer-aided-design, there is a high probability that it will augur well even for data mining applications.

CHAPTER V

CONCLUSION AND FUTURE WORK

5.1 Conclusions

In this thesis some problems in the area of information processing have been addressed, described in detail, and an efficient hardware technique has been described for each of the problems. It has been shown that the tremendous developments over the past decade in the area of digital logic optimization have been used to solve problems in the area of information processing. Some problems that can be solved this way have been identified and some more problems in information processing area are expected to be solved using hardware techniques.

In chapter 2, a technique for optimization of a query is described. The problem of query optimization has been described in detail. The need for optimizing a query is described, the optimization problem is analyzed, previous strategies used to optimize a query are explained and a method is introduced for optimizing an SQL query using AND/OR graphs. Experimental results are given indicating the effectiveness of optimizing a query using this technique and they show that in most of the queries tested, the execution time of the queries is optimized by 30% using this query.

In chapter 3, an Multivalued Decision Diagram (MDD) based synthesis of Multi Valued Logic Networks (MVLN) networks is described, a description about

MVLN's and MDD's is given, an edge mapping process in which MIN and MAX gates are used for the mapping process instead of substituting each node by a multiplexer is described and it is shown that mapping an MVLN in this way has an advantage that resulting circuit size is proportional to the number of nodes in the corresponding MDD. It has been shown that this representing the query using MDD is a logical extension of representing the query using AND/OR graphs. Experimental results were given to prove that this mapping technique is very effective mapping technique.

In Chapter 4, a very important problem in data base systems known as datamining is addressed, the concepts of datamining are discussed in detail and some previously developed algorithms for generating association rules are discussed. It has been shown that not only the concentration should be on building better algorithms for the association rules, but there is also a great need for developing an efficient storage technique because datamining applications usually involve large datafiles and need a complex data structure with indexing and hashing to store the items in the transactions. A data structure requiring much less memory and well suited to the application is proposed. Experimental results are generated for a sample data file indicating that this is a very effective storage technique. Another very interesting feature is that using this data structure, not only the memory requirements are less but powerful manipulation techniques can be used (for example, manipulating the transactions) if this data structure is used for storage of transactions.

BIBLIOGRAPHY

- [1] V. Komaragiri, M. A. Thornton and R. Drechsler, "*Application of a hardware synthesis technique for Database query optimization*", IEEE Pacific Rim Conference on Communications, Computers and signal Processing (PACRIM), Pages: 716-719, 2001.
- [2] S. Chaudhuri, "*An Overview of Query Optimization in Relational Systems*", Proceedings of the 17th ACM SIGACT-SOGRMOD-SIFART Symposium on Principles of Database Systems (PODS), Pages 34-43, 1998.
- [3] M. Jarke and J. Koch, "*Query Optimization in Database Systems*", Computing Surveys, Vol. 16, No. 2, Pages 111-52, 1984.
- [4] C. T. Yu and W. Meng, "*Principles of Database Query Processing for Advanced Applications*", Morgan Kaufmann Publishers, Inc., 1998.
- [5] W. Ziarko, Rough sets, "*Fuzzy sets and Knowledge discovery*", Springer-verlag, 1994.
- [6] D.C. Rine, "*Computer science and Multiple Valued Logic*", North-Holland 1984.
- [7] R. Drechsler, W. Kunz and D. Stoeffel, "*Decision Diagrams and AND/OR graphs for Design Automation problem*", Proceedings of the International Conference on Information, Communication and Signal Processing, Pages 67-72, 1997.
- [8] W. Kunz and D. K. Pradhan, "*Recursive Learning, A New Implication Technique for efficient solutions to CAD problems: Test, Verification and Optimization*", IEEE Transactions on CAD, Pages 1143-1158, 1994.
- [9]. R. E. Bryant, "*Graph Based Algorithms for Boolean function manipulation*", IEEE Transactions on computers, Pages 677-691, 1986.

- [10] A. Zuzek, R. Drechsler and M. A. Thornton, "*Boolean Function Representation and Spectral Characterization using AND/OR graphs*". Integration, The VLSI journal. Vol. 29, Pages 101-106, September 2000.
- [11] D. Stoeffel, W. Kunz and S. Gerbe, "*AND/OR reasoning graphs for determining prime implicants in Multilevel combinational circuits*", Proceedings of the ASP Design Automation Conference, Pages 25-32, 1997.
- [12] R. Drechsler and B. Becker, "*Decision Diagrams in Synthesis, algorithms , applications and extensions*", VLSI Design Conference, Pages 46-50, 1997.
- [13] R. E. Bryant, "*Binary Decision Diagrams and beyond: Enabling techniques for formal verification*", Int'l Conf on CAD, Pages 236-243, 1995.
- [14] D. Stoeffel, W. Kunz, S. Gerber, "*AND/OR Graphs*", Technical Report, MPI-I-95 -602.
- [15] H. Korth and A. Silberschatz, **Database System Concepts**, McGraw-Hill, Inc., New York, NY, 2nd edition, 1991.
- [16] F. M. Brown, **Boolean Reasoning**, Kluwer academic publishers, Boston, MA 1990.
- [17] Akers S, "*Binary Decision Diagrams*", IEEE transactions on computers, Vol.27, Pages 509-516, June 1978.
- [18] Drechsler, R. Gunther, W. Somenzi. "*Using lower bounds during dynamic BDD minimization*", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Pages 51-57, Jan. 2001.
- [19] R. Drechsler and B. Becker, "*Decision Diagrams in Synthesis, algorithms, applications and extensions*", VLSI Design Conference, Pages 46-50, 1997.

- [20] R. Drechsler and B. Becker, "*Efficient graph based representation of multi valued functions with application to genetic algorithms*", Intl.symp., on multiple valued logic, Pages 64-72, 1994.
- [21] K. S. Brace, R. L. Ruddell and R.E Bryant, "*Efficient implementation of a BDD package*", Design Automation Conf., Pages 40-45, 1990.
- [22] R. K. Brayton, Sunil. P. Khatri, "*Multi Valued Logic synthesis*", Intl conference on VLSI design, January 1999.
- [23] S. Malik, A. Srinivasan, "*Algorithms for discrete function manipulation*", Proceedings of int'l conference on computer-aided-design, 1990.
- [24] E. Clarke, M. Fujita and P. Mcgeer, "*Multi Terminal Binary Decision Diagrams: An efficient data structure for matrix representation*", Intl workshop on logic synthesis, Pages 1-15, 1993.
- [25] R. Drechsler, D. Jankovic, and Stankovic, "*Generic implementation of DD packages in MVL*", EUROMICRO Conference, Proceedings. Volume: 1, Pages 352–359, 1999.
- [26] P. Arunachlam, C. Chase, D. Moundanos, "*Distributed binary decision diagrams for verification of large circuits*" Computer Design: VLSI in Computers and Processors, ICCD '96. Proceedings, Pages: 365–370, 1996.
- [27] J. S. Park, M. S. Chen, and P. S. Yu " *An effective hash-based algorithm for mining association rules*", Proc. ACM SIGMOD conf, Management of data, May 1995.
- [28] R. Srikanth and R. Agarwal, "*Mining centralized association rules*", Proceedings. 21st Int'l Conf. Very Large Databases, Pages: 1-23, Sept 1995.

- [29] R. T. Ng and J. Han. “*Efficient and Effective clustering methods for Spatial Data Mining*”, Proc. 20th Int’l Conf. Very Large Databases, Pages: 144-155, Sept 1994.
- [30] U. M. Fayyad, “ *Advances in Knowledge Discovery and Data Mining*”, AAAI/MIT press, 1996.
- [31] R. Agarwal, T. Imielinski and A. Swamy, “*Database Mining: A Performance perspective*”, Proc. 18th Int’l Conf. Very Large Databases, Pages: 914-925, August, 1992.
- [32] U. M. Fayyad, P. Smyth and R. Uthruswamy, “*Advances in knowledge discovery and data mining*”. AAAI/MIT Press, 1996.
- [33] M. A. Thornton, R. Drechsler and D. Wessels, “*MDD based synthesis of multi-valued logic networks*”. IEEE ISMVL, Pages: 41-46, May 23-25, 2000.
- [34] V. M. Sarathy, L. V. Saxton D. V. Gucht, “*Algebraic foundation and optimization for object based query languages*”, Proceedings, Ninth International Conference on Data Engineering, Pages: 81-90, 1993.
- [35] A. S. Chiou, J. C. Sieg, “*Optimization for queries with holistic functions*” Proceedings. Seventh International Conference on Database Systems for Advanced Applications, Pages: 327-334, 2001.
- [36] A. Gupta, S. Sudarshan, S. Vishwanathan, “*Query scheduling in multi query optimization*”, International Symposium on Database Engineering & Applications Pages: 11 –19, 2001.
- [37] G. Antoshenkov, “*Dynamic query optimization in Rdb/VMS*”, Proceedings. Ninth International Conference on Data Engineering, Pages: 538-547, 1993.

- [38] A. Hameurlain, F. Morvan, “*An overview of parallel query optimization in relational systems* “,Proceedings. 11th International Workshop on Database and Expert Systems Applications, Pages: 629-634, 2000.
- [39] L. Stockmeyer, D. S. Modha, “*Links between complexity theory and constrained block coding*”, IEEE Transactions on Information Theory, Jan. 2002 Pages: 59-88.
- [40] C.J Hursch, “**SQL, the structured query language**”, Windcrest publishers, 1992.

APPENDIX A - Multi Valued Query Optimization

```

/*****
THIS SECTION DESCRIBES DIFFERENT MVDD ROUTINES FOR BUILDING
AN MVDD AND ALSO MAPPING IT TO MIN AND MAX GATES. SOME
SECTION OF THIS CODE IS A MODIFIED VERSION DR. M. Miller's MDD
PACKAGE.
*****/

```

```

void
skip(char c)
/* This routine is used to Skip characters to end of line. */
{
    while(1)
    {
        if(c=="\n")break;
        scanf("%c",&c);
    }
}

```

```

void
collect(void); /* Collect garbage */

```

```

DDcubeinput(DDedge f[],int *p,int *n,int *m)
/* Reads a list of cubes and returns the function root edges in f[]. Also sets p =
number of values,
n = number of inputs and m = number of outputs.
INPUT ORDER RESTRICTION AND LIMITATION
Input must be a .r command followed by a .i command
followed by a .o command followed by the list of cubes
followed by the .e command.
*/
{
    char ch1,ch2,s[80];
    int i,j,k,cnt,x;
    int value[4];
    DDedge c,cube,e,edge[maxp];
    /* Read number of logic values. */
    fflush(stdin);
    scanf("%c%c%d\n",&ch1,&ch2,p);
    fflush(stdin);
    /* Init MVDD package. */
    DDinit(*p);
    fflush(stdin);
    /* Read number of input variables */
    scanf("%c%c%d\n",&ch1,&ch2,n);

```



```

/* Read number of output functions */
fflush(stdin);
scanf("%c%c%d\n",&ch1,&ch2,m);
/* Init function edge pointers. */
for(i=0;i<*m;i++)
{
f[i]=DDlogic[0];
DDincref(f[i]);
}
/* Loop one cube at a time */
while(1)
{
scanf("%c%c",&ch1,&ch2);
/* Check for end of cube list */
if((ch1=='.')&&(ch2=='e')) break;
/* Put characters of cube in s[] */
s[0]=ch1;
s[1]=ch2;
cnt=2;
for(i=2;i++)
{
scanf("%c",&ch1);
if(ch1=='\n')break;
s[i]=ch1;
cnt++;
}
/* Convert input side of cube one input at a time */
j=0;
for(i=0;i<*n;i++)
{
for(k=0;k<*p;k++) value[k]=0;
while(1)
{
if(s[j]==' '||s[j]=='-') break;
value[s[j]-'0']=2;
j++;
}
j++;
if(i==0)
{
cube=DDliteral(i+1,value);
}
else
{
for(k=0;k<DDradix;k++)
if(value[k]==2) edge[k]=cube;
}
}
}

```

```

else
{
edge[k]=DDlogic[0];
}
cube=DDfunc(i+1,edge);
}
}
/* Now combine cube with appropriate output functions */
for(i=0;i<*m;i++)
{
if(s[j]!='0'&&s[j]!='-'&&s[j]!='~')
{
k=s[j]-'0';

if(k<DDradix-1)
/* Form min of cube with output value if necessary */
{
e=DDlogic[k];
e=DDmin(cube,e);
apply(minid);
e=DDcleanup(e,ptr(e)->flag+1);
} else
e=cube;
/* Max cube with output function to date */
DDdecref(f[i]);
f[i]=DDmax(f[i],e);
apply(maxid);
f[i]=DDcleanup(f[i],ptr(f[i])->flag+1);
DDincref(f[i]);
}
j++;
}
/* Collect garbage */
collect();
}
}

```

```

/*****
This section contains a programs which takes the
initial input file, builds an MDD and also synthesizes
the obtained MDD using MAX and MIN gates
*****/

#include <iostream>
#include <fstream>
#include <stdlib>
#include <memory>
#include "MVDDpackage.h"
#include "MVDDpackage.c"
#include "MVDDread.c"
#include "timing.c"
using namespace std;
typedef struct MDDmap Childedge;
struct MDDmap
{
    string child[8];          /* specifies the number of the children */
    string terminal[8];      /* dummy nodes to identify the terminal
node */
    string nodename;        /* identifying a particular node */
    int prevnodes;          /* ancestors */
    int pos;                /* determines the position of the current
child of the node */

    int termpos;
    int prevnodecntr;
    int prevnodepos[8];     /* Gives information about which child
of the previous node is connected to the
current node.*/

    Childedge cedge[0];
};

void
main(void)
{
    DDedge f[10],pntr;
    int i,p,m,n;
    long time1,time2;
    ofstream outfile("esp.out")
    time1=usertime();
    /* Read a MVL cube specification from STDIN */
    /* and build the (shared) MVDD */
    /* NOTE DDcubeinput calls DDinit */
    DDcubeinput(f,&p,&n,&m);
    time2=usertime();

```

```

for(i=0;i<m;i++)
{ printf("\n next output \n");
  DDprint(f[i]);
/* Print function is described below */
DDprint(DDedge e)
/*****
  Prints and MVDD but not in a graphical format
  but in a format which is a bit difficult to comprehend
*****/
{
  int i;
  if(ptr(e)->redundant) printf("R");
  outfile<<" "<< (long)ptr(e); //(long)ptr(e)->alias;
  if(DDterminal(e))
  {
    outfile<<" "<<DDgetcycle(e);
  }
  else if(DDid(e)<maxn)
  {
    // printf("v%ld[" ,DDid(e));
    outfile<<"[" ";
    for(i=0;i<DDradix;i++)
    {
      DDprint(DDchild(e,i));
    }
    outfile<<"]";
  } else
  { if(DDid(e)==maxid) printf("max[";
    else if(DDid(e)==minid) printf("min[";
    else if(DDid(e)==sumid) printf("sum[";
    else outfile<<"???[";
    for(i=0;i<2;i++)
      DDprint(DDchild(e,i));
    outfile<<"]";
  }
}
/*DDprint(f[i]);*/
printf("\n");
}
/* Display stats */
DDstats();
printf("Time to read: ");
printtime(time2-time1);
printf("\n");

```

```

}
int stackptr=0,nodecptr=0,No_of_nodes=0,childpos=0,Nodes_done=0;
    MDDmap node[200];          /*Stack of 200 MDD nodes */
    int nodestackptr=0,tempcptr=0;
    bool flag = false;
    int maxchild =      DDradix;    /* Maximum number of children
allowed */
/* illegal to use more number of children
*/
    int No_Of_Max_Nodes =0;
    int No_Of_Min_Nodes =0;
    bool init=true;
    string expression;
    ifstream infile("esp.out");
    ofstream outfile("viv.txt");
    for(int cnt=0;cnt<200;cnt++)
    {
        node[cnt].pos = 0;          /* initializing the position pointer to
child "Zero" */
        node[cnt].prevnodes=0;      /* initializing the previous nodes to "Zero" */

        node[cnt].prevnodecptr=0;
    }
    if(!infile)
        cout<<"Error in opening the file"<< endl;
    while(infile>>expression)
    {
        if(nodecptr== -1)
            /* Used to ensure that traversal will not go beyond root node */
            break;
        if(init==true)
        {
            node[nodecptr].nodename= expression;
            init = false;
        }
        else if(expression=="")
        { /* This loop is used to decrease the node count when a
terminal node appears */
            Nodes_done++;
            /* Used as a reference number to traverse back to the right node */
            nodecptr=No_of_nodes - Nodes_done;
        }
        while(1)
        {
            if(node[nodecptr].pos >= maxchild)
            {
                nodecptr--;
            }
        }
    }
}

```

```

else
    break;
}
cout<<"nodecptr is "<< nodecptr<< endl;
cout<<"No_of_nodes is "<< No_of_nodes<<endl;
}
else if(flag == true )
{
    childpos=node[nodecptr].pos;
    /* position of the current active child of the node */
    node[nodecptr].child[childpos] = expression;
    node[nodecptr].pos++;
    nodecptr=No_of_nodes+1;
    /* Increase the node pointer so that the pointer points to next
    available node in the traversal */
    No_of_nodes++;
    node[nodecptr].nodename = expression;
    node[nodecptr].prevnodes++ ;
    node[nodecptr].prevnodecptr++;
    node[nodecptr].prevnodepos[prevnodecptr]=    node[nodecptr-
1].pos -1;

    flag= false;
}
else if(expression=="[")
{
    flag = true;
    /* This flag is used to signal that the next node is the
    first child of the current of the node */
}
else
{
    if(expression[0]!='e')
    {
        /* This loop is used to identify whether the child is a terminal
        node or not
        and also to carry out MDD building operations*/
        childpos = node[nodecptr].pos;
        node[nodecptr].child[childpos] = expression;
        node[nodecptr].pos++;
        nodecptr=No_of_nodes+1;
        No_of_nodes++;
        node[nodecptr].nodename= expression;
        node[nodecptr].prevnodes++ ;
        node[nodecptr].prevnodecptr++;
        node[nodecptr].prevnodepos[prevnodecptr]=    node[nodecptr-
1].pos -1;

```

```

    }
    else if(expression[0]=='e')
    {
        /* If the current node is a terminal node */
        nodecctr--; /* decrease the node pointer */
        No_of_nodes--;
        childpos=node[nodecctr].pos-1;

        /* decide which terminal child it is */
        if(expression == "e0")

node[nodecctr].child[childpos]="TERM0";
        else if (expression == "e1")

node[nodecctr].child[childpos]="TERM1";
        else if (expression == "e2")

node[nodecctr].child[childpos]="TERM2";
        else if (expression == "e3")

node[nodecctr].child[childpos]="TERM3";
        if(node[nodecctr].pos > maxchild)

        outfile<<" ERROR :more number of children
than allowed"<<endl;

        outfile<<"child["<< childpos<<"] of"<<
node[nodecctr].nodename<<"is "<< node[nodecctr].child[childpos]<<endl;
    }
}

for(int i=0;i<maxchild;i++)
{ /*display of the children of each node in the MDD. Strictly for
troubleshooting purpose */
    for(int k=0;k<maxchild;k++)
        outfile<<"child["<<k<<"] of " << node[i].nodename <<"is
"<< node[i].child[k]<<endl;
        outfile<<endl;
    }
for (int i1=0; i1<No_of_nodes ; i1++)
{
    if(node[i1].prevnodes == 0)
    {
        int OutGoingNodes = maxchild;
        /* This is the root node */
        for ( int j=1; j<maxchild; j++)
        { int n=1;
            while (n<= maxchild)

```

```

        {
            while(1)
            {
                if(node[i1].child[j].nodename==      node[i1].child[j-
n].nodename)

                {
                    OutGoingNodes--;
                }
                n++; // temp =n;
                if(n== j)
                    break;
            }
        }

        No_Of_Max_Nodes = OutGoingNodes -1;
        No_Of_Min_Nodes++;
    }
    else if (node[i1].prevnodes >0)
    {
        /*for each outgoing edge*/
        int *TotalEdges;;
        TotalEdges = new int[maxchild]; /*used as a temporary array
*/

        int total=maxchild;
        for(int j=0; j<maxchild;j++)
        {
            TotalEdges[j] = node[i1].child[j].pos;

        }
        int temp=0;
        max1 = TotalEdges[0];
        for(int j1=1;j1<maxchild;j1++)
        {
            temp=j1;
            for(int k=0;k<temp;k++);
            {
                if(temp== maxchild)
                    break;
                if(TotalEdges[j1]== TotalEdges[k])
                {
                    total--;
                    temp++;
                }
            }
            else
            {
                max1=DDmax(TotalEdge[j1],max1);
            }
        }
    }
}

```



```

        No_Of_Max_Nodes++;
    }
}
}
max2 = 0;
for (int p=0; p< maxchild; p++)
{
    temp=node[i].prevnodes[p];
    J[temp]=1;
    max2= DDmax(J[temp],max2);
    No_Of_Max_Nodes++;
}
DDmin(max1,max2);
No_Of_Min_Nodes++;
}
cout<<"total number of MAX nodes is given by "<< No_Of_Max_Nodes<<endl;
cout<<"total number of MIN nodes is given by "<< No_Of_Min_Nodes<<endl;
}
}
}
/* definitions of some sample functions used in this mapping technique.*/
DDedge
getnode(int p)
/******
    Get a node from the avail list or allocate a new node.
    Initialize the node fields.
    *****/
{
    DDedge np;

    if(avail==NULL)
    {
        DDtotalnodes++;
        np=malloc(sizeof(node)+p*sizeof(DDedge));
    }
    else
    {
        np=avail;
        avail=avail->next;
    }
    np->ref=0;
    np->redundant=0;
    np->flag=0;
    np->alias=NULL;
    return(np);
}

```

```

DDedge
DDchild(DDedge e,int p)
/*****
Returns pth child of node pointed to by e with cycle
accumulation.
*****/
{
    DDedge ep;

    ep=ptr(e)->e[p];
    DDsetcycle(ep,(DDgetcycle(ep)+DDgetcycle(e))%DDradix);
    return(ep);
}
void
DDinit(int r)
/*****
Initialize the MVDD package with r as the max number of logic
values to be used this time.
*****/
{
    int i,j;

    printf("MVDDpackage V. 3.0\n");
    printf("-----\n\n");
    printf("node size: %ld\n\n",sizeof(node)+(r-1)*sizeof(DDedge));
    printf("\n\n");

    /* Init globals */
    avail=NULL;
    DDradix=r;
    DDtotalnodes=0;
    DDactivenodes=0;
    DDzero=getnode(DDradix);
    DDzero->v=0;
    for(i=0;i<maxp;i++)
        DDzero->e[i]=NULL;

    /* Set unique table to empty */
    for(i=0;i<nid;i++)
        for(j=0;j<nbucket;j++)
            unique[i][j]=NULL;

    /* Init DDconst vector of constants */
    for(i=0;i<DDradix;i++)
    {

```

```

    DDlogic[i]=DDconst(i);
    printf("DDlogic[%d] is given by %p\n", i,DDlogic[i]);
}
/* Init variable ordering vectors */
for(i=0;i<=maxn;i++)
    order[i]=orderinv[i]=i;

}

DDedge
DDmax(DDedge e0,DDedge e1)
{
    DDedge e[maxp];
    int i;

    /* Check terminal cases */
    if(e0==DDlogic[DDradix-1]||e1==DDlogic[DDradix-1])
        return(DDlogic[DDradix-1]);
    if(e0==DDlogic[0]) return(e1);
    if(e1==DDlogic[0]) return(e0);
    /* Create the appropriate operator node */
    e[0]=e0;
    e[1]=e1;
    for(i=2;i<DDradix;i++) e[i]=NULL;
    return(DDfunc(maxid,e));
}

DDedge
DDmin(DDedge e0,DDedge e1)
{
    DDedge e[maxp];
    int i;

    /* Check terminal cases */
    if(e0==DDlogic[0]||e1==DDlogic[0])
        return(DDlogic[0]);
    if(e0==DDlogic[DDradix-1]) return(e1);
    if(e1==DDlogic[DDradix-1]) return(e0);
    /* Create the appropriate operator node */
    e[0]=e0;
    e[1]=e1;
    for(i=2;i<DDradix;i++) e[i]=NULL;
    return(DDfunc(minid,e));
}

```

```

DDedge
DDsum(DDedge e0,DDedge e1)
{
    DDedge e[maxp];
    int i;

    /* Check terminal cases */
    if(e0==DDlogic[0]) return(e1);
    if(e1==DDlogic[0]) return(e0);
    /* Create the appropriate operator node */
    e[0]=e0;
    e[1]=e1;
    for(i=2;i<DDradix;i++) e[i]=NULL;
    return(DDfunc(sumid,e));
}

DDedge
DDfunc(int v,DDedge e[])
/*****
    Create a node representing a function.
    *****/
{
    DDedge p;
    int i,q,q0,q1;
    if(v>maxn){
        /* Operator node. */
        /* Terminal value cases. */
        if(DDterminal(e[0])&&DDterminal(e[1]))
        {
            q0=DDgetcycle(e[0]);
            q1=DDgetcycle(e[1]);
            if(v==maxid)
            {
                if(q0>q1) q=q0; else q=q1;
            }
            else if(v==minid)
            {
                if(q0<q1) q=q0; else q=q1;
            }
            else if(v==sumid)
            {
                q=(q0+q1)%DDradix;
            }
            return(DDlogic[q]);
        }
    }
    //end of if(v>maxn)

```

```
/* Create node. */  
p=getnode(DDradix);  
p->v=v;  
for(i=0;i<DDradix;i++)  
    p->e[i]=e[i];  
/* Locate it. */  
p=locate(p,1);  
return(p);  
}
```

APPENDIX B – Storage Technique For Datamining

/*******
 THIS PROGRAM IS USED TO BUILD A DATA STRUCTURE FOR
 DATAMINING APPLICATIONS. THE MAIN CONCENTRATION IS TO BUILD
 A DATA STRUCTURE FOR DATA TRANSACTION FILES. THIS PROGRAM
 INCLUDES ROUTINES DEVELOPED IN THE “BDD” PACKAGE.

/*******/

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <time.h>
#include "util.h"
#include "cudd.h"
#include "bnet.h"
#include "ntr.h"
#include "cuddInt.h"
#include "netlist.h"
#define Item_Bit_len 16 /* Bit length for each item */
#define Max_Basket_Size 100
#define Block_Size 100
static NtrOptions * mainInit ARGS(());
extern void BddEdgeMap(DdManager * manager,BnetNetwork *netlist);
extern void AddEdgeMap(DdManager * manager,BnetNetwork *netlist, int );

void AddEdgeMap( char* Transaction_File_Name, DdManager *manager)
{
    GdManager *gateManager;
    int result = 0;
    int k = 0;
    int j = 0;
    int p=0;
    int tran_count = 0;
    int totalVars = 0;
    int numVars = 0;
    int orgVars = 0;
    int nvars = 0;
    int *varIn;
  
```

```

int    i = 0;
DdNode *support;
DdNode *scan;
DdNode *fo;          /* pointers to the nodes of a BDD. Most of the pointers
                        as temporary pointers */

DdNode *fc;
DdNode *fx;
DdNode *tempfx;
DdNode *storefx;
DdNode *fAdd;
DdNode *shortA;
BnetNode *circ_node=NULL;
DdNode **ff;
DdNode **trans;
DdNode **Temptrans;
DdNode **fMiddle;
FILE    *fpx;
FILE    *fp;
char    basket[ Max_Basket_Size * 8 ]; /* storage structure for storing each
itemset */
char    Itemset_Str[ 8 ];
char    tempstr[ 8 ];
int     Itemset_Len = 0;
int     Itemset_Num = 0;
unsigned short int Item_No = 0;
        unsigned short int bit=0;
unsigned short int bit1=0;
unsigned short int tempbit=0;
int storebit=0;
int check=0;
int count = 0;
long initialtime=0; /* used to calculate cpu seconds */
long totaltime=0;
long tempfortime=0;
int s,r=0;
        long timeStart;
orgVars = manager->size;
        /* Open transaction file */
        if(( fp = fopen( Transaction_File_Name, "r" ) ) == NULL )
            printf("Can't open the file: %s!", Transaction_File_Name );
        /* Itemset_Num: the number of the itemsets in the file */
        Itemset_Num = 0;
        while( !feof( fp ) ){
check=check+1;
            fgets( basket, Max_Basket_Size * 8, fp );
            Itemset_Num++;

```

```

/* Get each transaction in a item until end of file is reached */
}
printf( "Itemset_Num = %u\n", Itemset_Num );
fseek(fp,0,SEEK_SET);
ff =ALLOC( DdNode*, Item_Bit_len * sizeof( DdNode* ) );
trans =ALLOC( DdNode*, Itemset_Num * sizeof( DdNode* ) );
Temptrans =ALLOC( DdNode*, Itemset_Num * sizeof( DdNode* ) );
Cudd_AutodynEnable( manager, CUDD_REORDER_SIFT );
/* Calling a CUDD function */
for(i=0;i<Item_Bit_len;i++ )
{
    fc = Cudd_bddNewVar( manager );
    ff[i] = fc;
    /* Create a new BDD variable for each of the 16 bits*/
}
printf( "Tran_No    Memory    Time    Address    \n" );
cuddInitInteract(manager); /* must use this before use swapInPlace */
cuddGarbageCollect(manager, 1);
k = 0;
while( !feof( fp ) ){
    memset( basket, '\0', Max_Basket_Size * 8 );
    fgets( basket, Max_Basket_Size * 8, fp );
    trans[0] = Cudd_ReadZero( manager );
    /* pointing to a constant node '0' initially */
    storefx = Cudd_ReadOne(manager);
    Itemset_Len = strlen( basket );
    for(s=0;s<8;s++)
    {
        tempstr[s]=0;
    }
    timeStart = util_cpu_time();
    /* Starting the time counter */
    for( i = 0; i < Itemset_Len; i++ )
    {
        if(basket[i]=='\t') /*indicates an end of an item */
        {
            Item_No = atoi(tempstr );
            s=0;
        }
        else
        {
            tempstr[s]=basket[i];
            /*concatenate individual digits into a string */
            s++;
        }
        tempbit = Item_No << 2;
    }
}

```



```

        printf("Item_No is %d\n", Item_No);
        bit1=tempbit;
        fx = Cudd_ReadOne(manager);
/* The below for loop is used to AND current BDD to the new bit
   if it is a '1' and NOT the BDD if the bit is a '0'. This
   is used to build a BDD for a single item in an itemset */
        for( j = 0; j <= Item_Bit_len-3; j++ )
        {
            bit = bit1>>Item_Bit_len-1;
            if(bit > 0 )
            {
                fx = Cudd_bddAnd(manager,fx,ff[j]);
            }
        }
    else
    {
        tempfx=Cudd_Not(ff[j]);
        fx = Cudd_bddAnd(manager,fx,tempfx);

    }
    bit1=tempbit<<j+1;
    printf( "a--0-1%IX %IX %IX\n",trans[k], manager, fx );
    Cudd_Ref(fx);
}
trans[k] = Cudd_bddOr(manager, fx, storefx);
/* OR the BDD's of different items */
Cudd_RecursiveDeref(manager,fx);
/* Taking care of reference counts */
storefx=trans[k];
Cudd_Ref(trans[k]);
}
p++;
Temptrans[k]=storefx;
Cudd_RecursiveDeref(manager,trans[k]);
printf("result is %d \n", result);
k++;
/*Cudd_RecursiveDeref(manager,trans[k]);*/
Cudd_ReduceHeap( manager, CUDD_REORDER_SIFT, 0 );
Cudd_AutodynDisable( manager );
result = Cudd_ReadNodeCount( manager );
/* counting the total number of nodes in the BDD */
printf("result is %d \n", result);
tempfortime++;
initialtime= (util_cpu_time() - timeStart);
totaltime=totaltime+initialtime;
printf("%-15u%-15u%-15s%-15u\n", k, result,
util_print_time(util_cpu_time() - timeStart), trans[k] );

```

```

    }
    if( fclose(fp) == NULL )
        printf( "Can't close the file: %s!  \n", Transaction_File_Name );
}

```

```

NtrOptions *mainInit( )
{
    NtrOptions    *option;
    /* Initialize option structure. */
    option = ALLOC(NtrOptions,1);
    option->initialTime  = util_cpu_time();
    option->verify       = FALSE;
    option->second       = FALSE;
    option->file1        = NULL;
    option->file2        = NULL;
    option->traverse     = FALSE;
    option->depend       = FALSE;
    option->image        = NTR_IMAGE_MONO;
    option->imageClip    = 1.0;
    option->approx       = NTR_UNDER_APPROX;
    option->threshold    = -1;
    option->from         = NTR_FROM_NEW;
    option->groupnspcs   = NTR_GROUP_NONE;
    option->closure      = FALSE;
    option->closureClip  = 1.0;
    option->envelope     = FALSE;
    option->scc          = FALSE;
    option->maxflow      = FALSE;
    option->zddtest       = FALSE;
    option->sinkfile     = NULL;
    option->partition    = FALSE;
    option->char2vect    = FALSE;
    option->density      = FALSE;
    option->quality      = 1.0;
    option->decomp       = FALSE;
    option->cofest       = FALSE;
    option->clip         = -1.0;
    option->noBuild      = FALSE;
    option->stateOnly    = FALSE;
    option->node         = NULL;
    option->locGlob      = BNET_GLOBAL_DD;
    option->progress     = FALSE;
    option->cacheSize    = 32768;
    option->maxMemory    = 0;          /* set automatically */
    option->slots        = CUDD_UNIQUE_SLOTS;
    option->ordering     = PI_PS_FROM_FILE;
}

```

```

option->orderPis    = NULL;
option->reordering   = CUDD_REORDER_NONE;
option->autoMethod   = CUDD_REORDER_SIFT;
option->autoDyn      = 0;
option->treefile     = NULL;
option->firstReorder = DD_FIRST_REORDER;
option->countDead    = FALSE;
option->maxGrowth     = 20;
option->groupcheck   = CUDD_GROUP_CHECK7;
option->arcviolation = 10;
option->symmviolation = 10;
option->recomb       = DD_DEFAULT_RECOMB;
option->nodrop       = TRUE;
option->signatures   = FALSE;
option->verb         = 0;
option->gaOnOff      = 0;
option->populationSize = 0;      /* use default */
option->numberXovers = 0;      /* use default */
option->bdddump       = FALSE;
option->dumpFmt       = 0; /* dot */
option->dumpfile      = NULL;
option->store         = -1; /* do not store */
option->storefile     = NULL;
option->load          = FALSE;
option->loadfile      = NULL;
return(option);
} /* end of mainInit */
/* Part of the program used to read the actual input transaction file and
   call the function to build a BDD for the corresponding transaction file*/
int main()
{
    DdManager *manager;
    DdNode     *f, *fc, *fcomp, *shortA, **ff;
    NtrOptions *option;
    BnetNetwork *netlist=NULL;

    /* Set up the options structure for converting the netlist to a BDD */
    option = mainInit();

    /* Initialize the DD manager */
    manager = Cudd_Init(0, 0, CUDD_UNIQUE_SLOTS,
CUDD_CACHE_SLOTS, 0);
    AddEdgeMap( "ttd.dat", manager); /*reading the input data file and calling the
function to create a BDD for the Transaction data file */
    return( 0 );
}

```

APPENDIX C - SQL Query Optimization

/*****
 THIS SECTION IS A MODIFIED VERSION OF THE LEX & YACC FILES FOR
 AN SQLPARSER DEVELOPED BY DR. M. Kernsten. THIS PARSER HAS BEEN
 MODIFIED TO SUITE THE PROCESS OF SQL QUERY OPTIMIZATION

*****/

```

D    [0-9]
L    [a-zA-Z_]
E    [Ee][-+]?{D}+
S    [\t\n ]*
% {
/* Type conflict resolution */
char getch();

#include <stdio.h>
#include <ctype.h>
#undef input
# define input() (((yytchar=yysptr>yysbuf?U(*--
yysptr):(char)getch(yyin))==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)

#define Symbol(X)    {/printf("symbol(%d)\n", X);*/ yylval=X; return(X);}
#define Vector(X,Y)  {/printf("vector(%d,%s)\n", X,Y);*/ yylval=Y; return(X)}

#define TRUE  1
#define FALSE 0
#define LISTING 1

FILE      *lexout;
int       endoffile =0;
long      atol();
float     atof();

typedef struct {
    char    *Lex_name;
    unsigned Lex_return;
} Lex_tab;

#include "tokens.h"

Lex_tab *lex_lookup();
% }
%%
  
```

```

"prolog".*"\\n" { printf("%s\\n", yytext+6); fflush(stdout);}
quit          { stop();}
"<="          {return LEQ;}
"<>"          {return NEQ;}
">="          {return GEQ;}
"^^"          {return DHAT;}
".."          {return DDOT;}
{L}({L}|{D})* {
register Lex_tab *lp;
char buf[20];
if (lp = (Lex_tab *) lex_lookup()){
Symbol(lp->Lex_return);
}else{ /* HIO 92 - remove Capital as first character*/
if isupper(yytext[0]) yytext[0]=tolower(yytext[0]);
/* END HIO */
Vector(IDENTIFIER, newstr("",yytext));
}
}
{D}+ {
Vector(INTEGERVALUE, newstr("int",yytext));
}
{D}+"."{D}+({E})? |
"."{D}+({E})? |
{D}+{E} {
Vector(FLOATVALUE, newstr("flt",yytext));
}
\[^\n\]*" { if ( yytext[yytextlen-1] == "\\")
              yymore();
  else {
    yytext[yytextlen-1] = '\0';
    Vector(STRINGVALUE, newstr("str",yytext+1));
  }
}
\[^\n\]*' { if ( yytext[yytextlen-1] == "\\")
              yymore();
  else {
    yytext[yytextlen-1] = '\0';
    Vector(STRINGVALUE, newstr("str",yytext+1));
  }
}
"--" { /* Skip comment */
while( yytchar != '\n' && yytchar!=0) input();
return(yylex());
}
{S} ;
";" { return 0;}

```

```

{ /*fputc(*yytext,stderr); */ Symbol( *yytext);}
%%
#define MAXLINELENGTH 2048
char linebuf[MAXLINELENGTH], arrowbuf[MAXLINELENGTH], *arrow;
char *cptr;
int lnerr;
char getch(fd)
FILE *fd;
{
extern int listing;
if( !cptr || !*cptr)
{
if( lnerr) printf("/ * %s*\n",arrowbuf);
lnerr=0;
/* Fill the buffer */
cptr= linebuf;
arrow= arrowbuf;
if( fgets(linebuf,MAXLINELENGTH,yyin) == NULL){
stop();
}
if( listing){
/* printf("%s",linebuf);*/
savetxt(cptr);
}
}
*arrow= *cptr=="\t" ? "\t": ' ';
arrow++;
*arrow= 0;
return(*cptr++);
}

/*
**MODULE lex_lookup
**FUNCTION Take the identifier in the buffer yytext and
** find out if it denotes a key word. Note that
** the identifier is mapped to lowercase as required
** by the report.
*/

Lex_tab *lex_lookup()
{
/* Search the lexical table for keywords */
register Lex_tab *t,*l;
register char *n,*n2;
char localname[200];
/* First map identifier to one case */

```

```

for(n=yytext, n2=localname; *n && n2<localname+200; n++, n2++)
*n2= isupper(*n)? *n-'A'+ 'a':*n;
*n2=0;
n=localname;
if( *n<'a' || *n >'w') return(0);
t = Keywords +Keyind[*n-'a'];
for(l= Keywords+Keyind[*n-'a'+1]; t<l;t++)
if (strcmp(t->Lex_name, n) == 0)
return(t);
return((Lex_tab *) 0);
}
char *lex_name(tokenid,value)
int tokenId;
char * value;
{
    int i;
    static char num[30];
    switch(tokenid){
    case GEQ:    return ">=";
    case LEQ:    return "<=";
    case NEQ:    return "<>";
    case DOT:    return "!";
    case DDOT:   return "!!";
    case HAT:    return "^";
    case DHAT:   return "^^";
    case '>':    return ">";
    case '=':    return "=";
    case '<':    return "<";
    case '+':    return "+";
    case '-':    return "-";
    case '*':    return "*";
    case '/':    return "/";
    /* HIO92 AvL added */
    case CORRELATION: return "corr";
    case ATTRIBUTE: return "attr";
    case DEFINEREL: return "defrel";
    /* end AvL */
    case IDENTIFIER:
    if(value && *value=='_') *value='x';
    case INTEGERVALUE:
    case FLOATVALUE:
    case STRINGVALUE:
    /* printf("ID %s\n",value); */
    return value;
    case LITERAL :
    return "";

```

```

    }
    for(i=0;Keywords[i].Lex_name;i++)
    if( Keywords[i].Lex_return== tokenid)
        return Keywords[i].Lex_name;
    if( value) return value;
    return "(unknown)";
}

newstr(hdr,str)
char *hdr,*str;
{
    /* Make a copy of a string */
    char *cell;
    int len;
    len= strlen(str);
    cell= (char *) malloc(len+12);
    if( *hdr)
        switch(*hdr){
case 's':
            sprintf(cell,"%s('%s')",hdr,str);
            break;
default:
            sprintf(cell,"%s(%s)",hdr,str);
        }
    else
        sprintf(cell,"%s",str);
    return((int) cell);
}

stop()
{
    printf("writeschema.\n");
    fflush(stdout);
    exit();
}

yywrap(){
    if( endoffile) return(0);
    endoffile = TRUE;
    return(-1);
}

/*sql.yacc*/

%token ATTR      ALTER      ADD
%token ALL       AND        ANY      AS
%token ASC       AVG        AUTHORIZATION
%token BEGINSYM  BETWEEN    BY

```


%token CHARACTER	CHECK	CLOSE	COBOL
%token COMMIT	CONSTRAINTS	CONTINUE	COUNT
%token CREATE	CURRENT	CURSOR	CORRELATION
%token COPY	COLUMN	DATABASE	
%token DECIMAL	DECLARE	DEFAULT	DELETE
%token DESC	DISTINCT	DOUBLE	DROP
%token ERASE	ESCAPE	EXEC	
%token EXISTS	FETCH	FIRST	
%token FIXED	FLOAT	FOR	FOREIGN
%token FORTRAN	FOUND	FROM	
%token GRANT	GOTO	GROUP	HAVING
%token HASH	INDEX		
%token INDICATOR	IN	INSERT	INTEGER
%token INTO	IS	KEY	LANGUAGE
%token LAST	LIKE	MAX	MIN
%token NOT	NULLSYM	NUMERIC	NEXT
%token OF	ON	OPEN	OPTION
%token OR	ORDER	PRIOR	
%token PASCAL	PLI	PRECISION	PRIMARY
%token PRIVILEGES	PROCEDURE	PUBLIC	
%token REAL	REFERENCE	ROLLBACK	RENAME
%token SCHEMA	SECTION	SELECT	SET
%token SMALLINT	SOME	SQL	SQLERROR
%token SQLCODE	SUM	TABLE	TO
%token TUPLE	TYPE	THEN	
%token UNION	UNIQUE	UPDATE	USER
%token VALUES	VIEW	WHENEVER	
%token WHERE	WITH	WORK	

%left SELECT
 %left FROM
 %left WHERE
 %left GROUP
 %left HAVING
 %left OR
 %left AND
 %left NOT
 %left '<' '>' '<=' '>=' '=' '<>'
 %left '*' '/'
 %left '+' '-'
 %left UNARY
 %left OF '.' DDOT '^' DHAT '->'

%token ENDSYM	BODY
%token LIST	METHOD
%token RESULT	STRING
%token GEQ	NEQ
	LEQ

RETURNS

```

%token DOT  HAT
%token APPEND

/* Internal tokens for marking the syntax tree */
%token IDENTIFIER  INTEGERVALUE  FLOATVALUE
STRINGVALUE
%token LITERAL  TRUEVALUE  FALSEVALUE  ATTRIBUTE
%token DEFINEREL
%{

/* #include "sqlvar.h"*/
#include <malloc.h>
#include "yytree.h"
%}

%%

session : query
        | schema
        | module
        | statement
        | table_definition
        | view_definition
        | create_body
        /* | assign_stmt */
        | ingresstmt
        | error

literal : INTEGERVALUE
        | FLOATVALUE
        | STRINGVALUE

column_list: column_1          {yyval(0);}

column_1: IDENTIFIER
        | column_1 ',' IDENTIFIER  {yydelete(1);}

table_name : IDENTIFIER
        | IDENTIFIER '.' IDENTIFIER  {yydelete(1); yymark('.');}

module_name : IDENTIFIER

cursor_name: IDENTIFIER

procedure_name: IDENTIFIER

```

```

data_type: character_string_type
    | exact_numeric_type
    | approximate_type

character_string_type: CHARACTER
    | CHARACTER '(' INTERVALVALUE ')'
    { yydelete(3); yydelete(1); yypromote(0); }

exact_numeric_type: NUMERIC prec_scale { yypromote(0); }
    | DECIMAL prec_scale { yypromote(0); }
    | INTEGER
    | SMALLINT

prec_scale: INTERVALVALUE
    | INTERVALVALUE INTERVALVALUE
    | /* empty */ { yydelete(0); }

approximate_type: FLOAT INTERVALVALUE { yypromote(0); }
    | FLOAT
    | DOUBLE
    | REAL
    | DOUBLE INTERVALVALUE { yypromote(0); }

value_list : '(' value_1 ')'
    { yydelete(2); yylist(1); yydelete(0); }

value_1 : value
    | value_1 ',' value { yydelete(1); }

value: USER
    | IDENTIFIER INDICATOR IDENTIFIER
    | literal
    | NULLSYM

/* column_name : primary_path */ /* TSQL */
column_name : IDENTIFIER { yymark(ATTRIBUTE); }
    | IDENTIFIER '.' IDENTIFIER { yydelete(1); yymark(ATTR); }

set_function:
    COUNT '(' alldistinct column_name ')'
    { yydelete(4); yydelete(1); yypromote(0); }
/*
    | funcname '(' alldistinct column_name ')'
    { yydelete(4); yydelete(1); yypromote(0); yytval->token=$1; }
*/

```

```

    | funcname '(' alldistinct value_expression ')'
    { yydelete(4); yydelete(1); yypromote(0); yytval->token=$1;}
funcname: AVG
    | MAX
    | MIN
    | SUM

value_expression: term
    | value_expression '+' term    { yypromote(1);}
    | value_expression '-' term    { yypromote(1);}

term : factor
    | term '*' factor              { yypromote(1);}
    | term '/' factor              { yypromote(1);}

factor : '+' primary %prec UNARY    { yydelete(0);}
    | '-' primary %prec UNARY
    {
        yytval->yysons[0]->token=INTEGERVALUE;
        yytval->yysons[0]->val= (int) "0";
        yymark('-');}
    | primary

primary : value
    | set_function
    | column_name /* column_name */
    | '(' value_expression ')'      { yydelete(2); yydelete(0);}

predicate: comparison_predicate
    | between_predicate
    | in_predicate
    | like_predicate
    | null_predicate
    | quantified_predicate
    | exists_predicate

comparison_predicate: value_expression LEQ value_expression
    { yypromote(1);}
    | value_expression GEQ value_expression
    { yypromote(1);}
    | value_expression NEQ value_expression
    { yypromote(1);}
    | value_expression '>' value_expression
    { yypromote(1);}
    | value_expression '<' value_expression

```

```

        { yypromote(1);}
    | value_expression '=' value_expression
        { yypromote(1);}
/*    | value_expression comp_op sub_query*/

comp_op: '=' { $$='=';}
        | NEQ { $$=NEQ;}
        | '<' { $$='<';}
        | '>' { $$='>';}
        | LEQ { $$=LEQ;}
        | GEQ { $$=GEQ;}

between_predicate: value_expression BETWEEN
                  value_expression AND value_expression
    | value_expression NOT BETWEEN
      value_expression AND value_expression

quantified_predicate: value_expression comp_op quantifier sub_query
                     { yymark($2); yydelete(1);}

quantifier: ALL
           | some_any

some_any: SOME
         | ANY

in_predicate : value_expression IN sub_query {yypromote(1);}
            | value_expression NOT IN sub_query %prec NOT
              { yypromote(1);yytval->token = strcat($1,$2);}
            | value_expression IN value_list {yypromote(1);}
            | value_expression NOT IN value_list %prec NOT
              { yypromote(1);yytval->token = strcat($1,$2);}

like_predicate : column_name LIKE STRINGVALUE escape
               | column_name NOT LIKE STRINGVALUE escape

escape : ESCAPE STRINGVALUE
        | /* empty */

exists_predicate: EXISTS sub_query {yypromote(0);}

null_predicate : column_name IS NULLSYM
               { yymark('='); yydelete(1);}
               | column_name IS NOT NULLSYM
               { yymark(NEQ); yydelete(1);}

```

```

search_condition: search_condition OR boolean_term {yypromote(1);}
    | boolean_term

boolean_term: boolean_factor
    | boolean_term AND boolean_factor
    { yypromote(1);}

boolean_factor: boolean_primary
    | NOT boolean_primary
    { yypromote(0);}

boolean_primary: predicate
    | '(' search_condition ')' {yydelete(2);yydelete(0);}
    | value_expression /* shiftreduce */

table_expression: from_clause where_clause group_clause having_clause

from_clause: FROM table_reference_list {yylist(1); yypromote(0);}

table_reference_list: table_reference
    | table_reference_list ',' table_reference { yydelete(1);}

where_clause: WHERE search_condition {yypromote(0);}
    | /* empty */ { yymark(WHERE);
        yytval->yysons[0]= (YYSTREE)
yybool(TRUEVALUE);}

group_clause: GROUP BY column_name_list
    { yylist(2); yydelete(1); yypromote(0);}
    | /* empty */ { yymark(GROUP); yylist(0);}

column_name_list: column_name
    | column_name_list ',' column_name {yydelete(0);}

having_clause: HAVING search_condition {yypromote(0);}
    | /* empty */ { yymark(HAVING);
        yytval->yysons[0]=
(YYSTREE)yybool(TRUEVALUE);}

sub_query: '(' SELECT alldistinct select_list table_expression ')'
    { yydelete(5); yylist(3); yypromote(1);yydelete(0);yyswap(1,2); }

query: SELECT alldistinct select_list table_expression
    { yylist(2);yypromote(0);
    /* HIO92 AvL - swap <select_list> and <table_expression> for output */

```

```

        yyswap(1,2);
    }

select_list : select_value
    | select_list ',' select_value {yydelete(1);}
    | '*'

select_value: value_expression
/*      | subquery */

query_expression: query_term
    | query_expression UNION query_term {yypromote(1);}
    | query_expression UNION ALL query_term {yypromote(1);}

query_term: query
    | '(' query_expression ')'
    { yydelete(2); yydelete(0);}

order_clause: ORDER BY sort_list
    { yylist(2); yydelete(1); yypromote(0);}

sort_list: sort_elm
    | sort_list sort_elm

sort_elm: sort_spec ASC {yypromote(1);}
    | sort_spec DESC {yypromote(1);}
    | sort_spec

sort_spec: INTEGERVALUE
    | column_name

/* ---- DEFINITION OF DATABASE ----- */

schema : CREATE SCHEMA authorization_clause schema_list
    { yylist(3); yypromote(1); yydelete(0);}
    | DATABASE IDENTIFIER {yypromote(0);}
    | drop_stmt
    | alter_stmt
    | index_stmt

authorization_clause: AUTHORIZATION IDENTIFIER {yypromote(0);}

schema_list: schema_elm
    | schema_list schema_elm

schema_elm: table_definition /* create_stmt */

```

```

    | view_definition
    | privilege_definition

/* create_stmt: CREATE table_definition { yylist(1);yypromote(0);} */

table_definition: CREATE TABLE table_name '(' table_element_list ')'
    { yylist(4); yydelete(5);yydelete(3);
      yydelete(1);yypromote(0);
    }
    | CREATE TABLE table_name AS query
    { yydelete(3); yydelete(1);yypromote(0);}
    | CREATE TABLE table_name LIKE table_name
    { yydelete(1); yypromote(0); }

table_element_list: table_element          {yylist(0);}
    | table_element_list ',' table_element {yylist(2);yydelete(1);}

table_element: column_definition
    | CONSTRAINTS unique_constraint_definition

column_definition: IDENTIFIER data_type
    | IDENTIFIER data_type column_constraint

column_constraint: NOT NULLSYM
    | NOT NULLSYM UNIQUE

unique_constraint_definition: UNIQUE '(' column_list ')'
    { yydelete(3); yydelete(1); yypromote(0);}

view_definition: CREATE VIEW view_table_name AS query viewcheck

view_table_name: table_name
    | table_name '(' column_list ')'
    { yydelete(3); yydelete(1);}

viewcheck: WITH CHECK OPTION
    { yydelete(2); yypromote(1); yydelete(0);}
    | /* empty */
    { yydelete(0);}

drop_stmt: DROP TABLE IDENTIFIER { yypromote(0);}
    | DROP VIEW IDENTIFIER { yypromote(0);}

alter_stmt: ALTER TABLE IDENTIFIER ADD COLUMN column_definition
    { yydelete(5); yydelete(2); yypromote(0);}

```



```

| ALTER TABLE IDENTIFIER DROP COLUMN column_definition
{ yydelete(5); yydelete(2); yypromote(0);}
| ALTER TABLE IDENTIFIER RENAME COLUMN IDENTIFIER TO
IDENTIFIER
{ yydelete(5); yydelete(2); yypromote(0);}

index_stmt : CREATE INDEX ON column_name { yydelete(3); yypromote(0);}
| CREATE HASH ON column_name { yydelete(3); yypromote(0);}

privilege_definition:
    GRANT privilege_list ON table_name TO grantees
    grantoption

grantoption    : WITH GRANT OPTION
                { yydelete(2); yypromote(1); yydelete(0);}
                /* empty */
                { yydelete(0);}

privilege_list : ALL                {yypromote(0);}
                | ALL PRIVILEGES    {yypromote(0); yydelete(0);}
                | action_list       {yylist(0);}

action_list: action
            | action_list ',' action {yydelete(1);}

action      : SELECT
            | INSERT
            | DELETE
            | UPDATE
            | UPDATE '(' column_list ')'
            { yydelete(3); yydelete(1); yypromote(0);}

grantees    : PUBLIC
            | IDENTIFIER

module : module_name_clause language_clause module_authorization
        procedure_list
        { yylist(3);}
        | module_name_clause language_clause module_authorization
        cursor_declaration_list procedure_list
        { yylist(3); yylist(4);}

module_authorization: AUTHORIZATION IDENTIFIER
                    { yypromote(0);}
                    /* empty */

```

```

procedure_list: procedure
    | procedure_list procedure

module_name_clause: MODULE
    | MODULE module_name { yypromote(0); }

cursor_declaration_list: cursor_declaration
    | cursor_declaration_list cursor_declaration

cursor_declaration:
    DECLARE cursor_name CURSOR FOR cursor_expression
    { yypromote(2); yydelete(2); yydelete(0); }

cursor_expression : query_expression
    | query_expression order_clause

procedure: PROCEDURE procedure_name parameter_declaration_list ';'
    statement ';'
    { yylist(2); yydelete(5); yydelete(3); yypromote(0); }

procedure: PROCEDURE procedure_name parameter_declaration_list ';'
    block
    { yylist(2); yydelete(3); yypromote(0); }

parameter_declaration_list: parameter_declaration
    | parameter_declaration_list parameter_declaration

parameter_declaration: IDENTIFIER data_type
    | SQLCODE

statement: open_statement
    | close_statement
    | cursor_declaration
    | select_statement
    | fetch_statement
    | insert_statement
    | searched_update_statement
    | positioned_update_statement
    | searched_delete_statement
    | positioned_delete_statement
    | commit_statement
    | rollback_statement

block : BEGINSYM stmtlist ENDSYM
    { yydelete(2); yylist(1); yypromote(0); }

stmtlist : statement

```

```

    | stmtlist ';' statement
    { yydelete(1);}

open_statement: OPEN cursor_name      {yypromote(0);}

close_statement: CLOSE cursor_name    {yypromote(0);}

select_statement: SELECT alldistinct select_list INTO parameter_list
                  table_expression
                  {yypromote(3); yydelete(0);}

alldistinct: ALL                      {yypromote(0);}
            | DISTINCT                {yypromote(0);}
            | /* empty */              { yymark(ALL);}

parameter_list: column_name
              | parameter_list ',' column_name    {yydelete(1);}

fetch_statement: FETCH direction cursor_name INTO parameter_list
                { yylist(3); yydelete(2); yypromote(0);}
fetch_statement: FETCH direction cursor_name
                { yypromote(0);}

direction:     NEXT | PRIOR | FIRST | LAST | /* empty */

insert_statement: INSERT INTO table_name insert_value
                 { yydelete(1); yypromote(0);}
                 | INSERT INTO table_name query
                 { yydelete(1); yypromote(0);}
                 | INSERT INTO table_name '(' column_list ')' insert_value
                 { yydelete(5); yydelete(3); yydelete(1); yypromote(0);}
                 | INSERT INTO table_name '(' column_list ')' query
                 { yydelete(5); yydelete(3); yydelete(1); yypromote(0);}

insert_value : VALUES '(' insert_value_list ')'
              {yylist(2);yydelete(3);yydelete(1);yypromote(0);}
              /* | VALUES call      {yypromote(0);} */

insert_value_list: insert_value
                 | insert_value_list ',' insert_value    {yydelete(1);}

insert_value: value

searched_update_statement: UPDATE updatehead where_clause {yypromote(0);}

positioned_update_statement: updatehead WHERE CURRENT OF cursor_name

```

```

        { yydelete(3); yypromote(2); yydelete(1);}

updatehead: def_table_name SET set_clause_list
        { yydelete(1);yylist(1);}

def_table_name: table_name

set_clause_list: set_clause
        | set_clause_list ',' set_clause      { yydelete(1);}

set_clause      : column_name '=' value_expression      { yypromote(1);} /* NEW */

searched_delete_statement:
        DELETE FROM table_name
        where_clause
        { yydelete(1);yypromote(0);}

positioned_delete_statement:
        DELETE FROM table_name
        WHERE CURRENT OF cursor_name
        { yydelete(5); yydelete(3);yydelete(1);yypromote(0);}

commit_statement: COMMIT WORK
        { yydelete(1); yypromote(0);}

rollback_statement: ROLLBACK WORK
        { yydelete(1); yypromote(0);}

/* ----- TSQL SECTION ----- */

/* ----- 2 TYPE CONSTRUCTORS ----- */

type_constructor: tuple_type

data_type: STRING /* a new basic type */

literal : TRUEVALUE | FALSEVALUE
        | literal_type '(' literal_list ')'
        { yydelete(3); yylist(2); yypromote(0); yydelete(0);}

literal_type : TUPLE { $$=TUPLE;}

literal_list : literal
        | literal_list ',' literal { yydelete(1);}

tuple_type : TUPLE OF tuple_body

```

```

        { yydelete(1);yydelete(0);}
    | tuple_body

tuple_body : '(' tuple_comp ')'
           { yydelete(2); yylist(1); yydelete(0); yymark(TUPLE);}

tuple_comp: tuple_comp ',' component {yydelete(1);}
           | component

component: named_type
           | named_type column_constraint      /* FOR SQL ONLY */
           | unique_constraint_definition      /* FOR SQL ONLY */

named_type :
    IDENTIFIER type_constructor
    { yymark(':');}
    | '[' type_constructor ']'
    { yymark(UNION);yydelete(2);yydelete(0);}
    | '[' IDENTIFIER type_constructor ']'
    { yymark(UNION);yydelete(3);yydelete(0);}
    | type_constructor

/* ----- 3 THE PATH CONSTRUCTOR -----*/

/* all columns are now interpreted as paths. For tables this may
   require some addition checks */

/*
primary_path : primary_factor      {$==$1;}
              | primary_factor OF primary_path
              { yymark(DDOT);
                yytval->yysons[1] = yytval->yysons[0];
                yytval->yysons[0] = yytval->yysons[2];
                yydelete(2);
              }

primary_factor : primary_elm
               | sub_query
               (Scomment)| call (Ecomment)
               | '*'

primary_elm: IDENTIFIER

```

```

expression_list : value_expression
    | value_expression ',' expression_list { yydelete(1);}
*/

/* ----- 3 THE FROM CONSTRUCTOR ----- */
/* table_name =usedtobe= table_selector */
table_reference: table_name
    | table_name IDENTIFIER { yymark(CORRELATION);}

/*
table_selector : selector
    | '!' selector      { yydelete(0); yymark(ATTR);}

selector : path

path: path_term
    | path_term '.' path  { yydelete(1); yymark(ATTR);}

path_term: IDENTIFIER
*/

/* ----- 4 THE UPDATE STATEMENTS ----- */

/* table_name =usedtobe= table_selector */
/*
updatehead: UPDATE table_selector SET set_clause_list
    { yydelete(2); yypromote(0);}
*/
/* column_name =usedtobe= primary_path */
/*
assign_stmt: SET column_name '=' value_expression
    { yymark(SET); yydelete(2); yydelete(0);}
    | SET column_name '=' query
    { yymark(SET); yydelete(2); yydelete(0);}

*/
create_body: CREATE BODY IDENTIFIER body_list
    { yypromote(1); yydelete(0); yylist(1);}

body_list: body_impl
    | body_list ',' body_impl
    { yydelete(1);}

body_impl: IDENTIFIER type_constructor { yymark(':');}
    | IDENTIFIER body { yymark(':');}

```

```

body  : AS search_condition {yydelete(0);}
      | AS query             {yydelete(0);}
      | AS language_clause   {yydelete(0);}

ingresstmt: COPY TABLE IDENTIFIER copyarg FROM STRINGVALUE
          { yydelete(4); yydelete(1); yypromote(0);}
ingresstmt: COPY TABLE IDENTIFIER copyarg INTO STRINGVALUE
          { yydelete(1); yypromote(0);}

copyarg: '(' copyitem ')'
        { yydelete(2); yydelete(0); yylist(0);}
copyitem: IDENTIFIER '=' IDENTIFIER
        { yydelete(1);}
        | copyitem ',' IDENTIFIER '=' IDENTIFIER
        { yydelete(3); yydelete(1);}

%%

#include "lex.yy.c"
#include "yyerror.h"
#undef yyerror
#define yyerror(X) {error(":",yystate,yylineno); myerror(yystate);}

```

```

/*****

```

THIS PART OF THE PROGRAM IS USED TO GENERATE A "BLIF" FILE FOR THE CORRESPONDING QUERY. SOME PART OF THE QUERY IS TRANSFORMED INTO A 'BLIF" FILE IN THIS SECTION.

```

*****/

```

```

#include <iostream>
#include <fstream>
#include <string>
#include <stack>
using namespace std;
void main()
{
int var,var1,varforhide,varforoperator,varin,i=0,j=0,k=0,len,count=0;
string store[5], inputvalue[3], dupeelement[20];
int vardupe=0, tempstackptr=0;
int stackptr=0,StoreInt=0,StoreIntTwo=0,
stackholdptr=0,stackholdoneptr=0,stackholdtwoopr=0;
int interpointer=0; /*for intermediate nodes in the AND/OR graph*/
stack<string> elementstack[150]; /*stack for storing part of a query */
stack<string> tempstack[20]; /*Temporary stack for manipulating query */
stack<string> stackhold[5]; /* Stack for holding temporary stack items*/
stack<string> stackholdone[5]; /* Stack for holding temporary stack items without
paranthesis */
stack<string> stackholdtwo[30];
stack<string> InterStack[10];
string element, temp, temp1,temp2; /*temporary strings for holding temporary
values*/
ifstream inFile("query.txt"); /*input query file */
ofstream outFile("query1.blif"); /* intermediate BLIF file */
if (!inFile)
outFile << "Cannot find fruit.txt" << endl;
outFile<<".model "<< " " << "query1.blif" << endl;
while (inFile >> element)
{
var++; /* variable to check for the output list after "select" */
var1++;/* variable to check for the list of coloumns in the table */
varin++;/* variable to check for the input list after "where" */

```



```

        dupeelement[vardupe] = element;
        vardupe++;
/*duplicate variable helpful in parsing the query and building BLIF file for it*/

        if((var1== 3 && var==3)|| (var1== 3 && varin==3)|| (varin== 3 && var==3))
        {

                outFile<<"there is a clash at the element: "<< element<< endl;
/*If the query is properly parsed by the parser this conflict never occurs*/
        }

        if((var1>3) && (element == ",")) /*for storing coloumns in the table */
        {
                var1=2;
        }
        if (element == ";") /*end of the query */
        {
                outFile<< " .end"<<endl;
                break;
        }
        else
        {
                len = element.length();
                temp = element;
        }
        if(temp=="select") /* Recognizing Start of the SQL query */
        {
                var=2;
        }
        if(temp == "from") /* signals the start of the list of coloums to be queried */
                var1=2;
        if(temp == "where") /* signals the start of actual conditions for the query */
        {
                varin=2;
        }
        if (var == 3)
        {
                temp1 = element;
        outFile<<".outputs "<< temp1<<endl; /* Actual output of the digital circuit */
        }
        if(var1==3)
        {
                if(element[len-1] == ',')
                {
                        element=element.erase(len-1,1);
                        var1=2;

```

```

        }
        temp2= element;
        store[i]=element; /* for storing the coloumn names in the table */
        i++;
    }
    if(varin==3)
    { varforhide++;
      varforoperator++;
      count++;
      varin=2;
      if((element=="=") || (element == ">")||(element=="<") || (element == "<="))
      {
          varforoperator=2;
      }
      if(varforoperator==3)
      {
          inputvalue[j]= element;
          j++;
      }
    }
    if((element != "=")&&(element != ">")&&(element != "<")&&(element !=
"<=")&&(element != ">=")&& (count%4!=0))
    {
        if(count==1)
            outFile<< ".inputs"; /* Actual inputs of the digital
circuit */

        if(varforhide!=3)
            outFile<<" " <<element;
        }
        else if(count%4!=0)
            varforhide = 2;
    }
}

if(var==2) /*Buildng some part of intermediate section of the BLIF file */
{
    for(stackptr=0;stackptr<200;stackptr++)
    {
        if(elementstack[stackptr].top() == "where" || elementstack[stackptr].top() ==
"Where")
            break; /* Parsing the query after the "where" statement */
        stackptr++;
    }
    stackptr++;
    while(elementstack[stackptr].top() != ";")
    { /*Build the circuit until the end of the query is reached */
        if(elementstack[stackptr].top() == "(")
        { StoreInt = stackptr;

```

```

        while(1)
        { /* Store the elements in a temporary tack to make the process of mapping
the query to the digital circuit simple. Also helps if there are nested queries */
            tempstack[tempstackptr].top() = elementstack[stackptr].top();
            stackptr++;
            if(elementstack[stackptr].top() == ")")
            { /* Signals the end of the sub portion of the query */
                Interstack[interpointer].push(elementstack[stackptr+1].top());
                interpointer++;
                break;
            }
        }

        for(i=stackptr;i>=StoreInt;i--)
        {
            elementstack[stackptr].pop();
/* Popping out the portion of the query on which mapping operation is completed */
            stackptr--;
        }
        StoreInt=0;
        for(i= tempstackptr;i>=0;i--)
        {
            if(((tempstack[tempstackptr].top() == "and" )||(tempstack[tempstackptr].top() ==
"AND" )) &&
                ((tempstack[tempstackptr-2].top() != "and" )
                ||(tempstack[tempstackptr].top() != "AND" )))
            { stackholdptr = tempstackptr-3;
              stackholdoneptr = tempstackptr+1;
              for (int j=0;j<3;j++)
              { /*collecting the inouts to the binary gate */
                  stackhold[j].top()=tempstack[stackholdptr].top();
                  stackholdptr++;
              }
              for (j=0;j<3;j++)
              {
                  stackholdone[j]=tempstack[stackholdoneptr];
                  stackholdoneptr++;
              } /*Statement corresponding to the query in BLIF file */
              outFile<<"names "<<
stackhold[0].top()<<stackhold[1].top()<<stackhold[2].top()<< " "<<

              stackholdone[0].top()<<stackholdone[1].top()<<stackholdone[2].top()<< " ";
              outFile<< "I"<< interpointer<<endl; /*Intermediate node used
at second

              or higher level stages in the circuit*/

```

```

        interpointer++;
        outFile<<"11 1"<<endl;
    }
    else if(tempstack[tempstackptr].top() == "or" || tempstack[tempstackptr].top() ==
"OR")
    {
        stackholdptr = tempstackptr-3;
        stackholdoneptr = tempstackptr+1;
        for (j=0;j<3;j++)
        {
            stackhold[j].top()=tempstack[stackholdptr].top();
            stackholdptr++;
        }
        for (j=0;j<3;j++)
        {
            stackholdone[j].top()=tempstack[stackholdoneptr].top();
            stackholdoneptr++;
        }
        outFile<<".names " <<
stackhold[0].top()<<stackhold[1].top()<<stackhold[2].top()<< " " <<

        stackholdone[0].top()<<stackholdone[1].top()<<stackholdone[2].top()<< " ";
        outFile<< "I" << interpointer<<endl;
        interpointer++;
        outFile<<"11 1"<<endl;
    }
    else if(tempstack[tempstackptr].top() == "between" || tempstack[tempstackptr].top() ==
"BETWEEN")
    {
        stackholdptr = tempstackptr-3;
        stackholdoneptr = tempstackptr+1;
        for (j=0;j<3;j++)
        {
            stackhold[j].top()=tempstack[stackholdptr].top();
            stackholdptr++;
        }
        for (j=0;j<3;j++)
        {
            stackholdone[j].top()=tempstack[stackholdoneptr].top();
            stackholdoneptr++;
        }
        outFile<< ".names " <<
stackhold[2].top()<<">"<<stackholdone[0].top()<<
stackhold[2].top()<< "<"<<stackholdone[2].top()<< " ";
        outFile<< "I" << interpointer<<endl;
        interpointer++;
        outFile<<"11 1"<<endl;
    }

```

```

    }

else
{ /*If the query does not have a ")" then the BLIF file is constructed as below
*/
    StoreIntTwo=stackptr;
    while(1)
    {
        stackholdtwo[stackholdoneptr].top() = elementstack[stackptr].top();
        stackptr++;
        stackholdtwooptr++;
        if((elementstack[stackptr].top() == "(")||(elementstack[stackptr].top()
        == ";")|| (elementstack[stackptr].top() ==
        ")")||(elementstack[stackptr].top() == "Union"))
        {
            break;
        }
    }
    stackptr=StoreIntTwo;
    /*Rest of the portion is self explanatory */
    for(i=stackptr;i>=StoreIntTwo;i--)
    {
        elementstack[stackptr].pop();
        stackptr--;
    }
    StoreInt=0;
    for(i=stackholdtwooptr;i>=0;i--)
    {

        if(((stackholdtwo[i].top() == "and" )||(stackholdtwo[i].top() == "AND"
)) &&
        ((stackholdtwo[i-2].top() != "and" ) ||(stackholdtwo[i-2].top() !=
"AND" )))
        {
            stackholdptr  = stackholdtwooptr-3;
            stackholdoneptr = stackholdtwooptr+1;
            for (int j=0;j<3;j++)
            {
                stackhold[j].top()=stackholdtwo[stackholdtwooptr].top();
                stackholdptr++;
            }
            for (j=0;j<3;j++)
            {
                stackholdone[j]= stackholdtwo[stackholdtwooptr];
                stackholdoneptr++;
            }

```

```

        outFile<<".names "<<
stackhold[0].top()<<stackhold[1].top()<<stackhold[2].top()<< " "<<

        stackholdone[0].top()<<stackholdone[1].top()<<stackholdone[2].top()<< " ";
        outFile<< "I"<< interpointer<<endl;
        interpointer++;
        outFile<<"11 1"<<endl;
    }
else if(stackholdtwo[i].top()=="or" || stackholdtwo[i].top()=="OR")
{
    stackholdptr = stackholdtwo[ptr]-3;
    stackholdoneptr = stackholdtwo[ptr]+1;
    for (j=0;j<3;j++)
    {
        stackhold[j].top()=stackholdtwo[stackholdptr].top();
        stackholdptr++;
    }
    for (j=0;j<3;j++)
    {
        stackholdone[j].top()=stackholdtwo[stackholdoneptr].top();
        stackholdoneptr++;
    }
    outFile<<".names "<<
stackhold[0].top()<<stackhold[1].top()<<stackhold[2].top()<< " "<<

    stackholdone[0].top()<<stackholdone[1].top()<<stackholdone[2].top()<< " ";
    outFile<< "I"<< interpointer<<endl;
    interpointer++;
    outFile<<"11 1"<<endl;
}
else if(stackholdtwo[i].top()=="between" || stackholdtwo[i].top()=="BETWEEN")
{
    stackholdptr = tempstackptr-3;
    stackholdoneptr = tempstackptr+1;
    for (j=0;j<3;j++)
    {
        stackhold[j].top()=tempstack[stackholdptr].top();
        stackholdptr++;
    }
    for (j=0;j<3;j++)
    {
        stackholdone[j].top()=tempstack[stackholdoneptr].top();
        stackholdoneptr++;
    }
    outFile<< ".names "<<
stackhold[2].top()<<">"<<stackholdone[0].top()<< stackhold[2].top()<< "<"

```

```

        <<stackholdone[2].top()<<" ";
        outFile<<"I"<< interpointer<<endl;
        interpointer++;
    outFile<<"11 1"<<endl;
    }
}
}
}
i=0;
while(1)
{
    if((InterStack[i].top() == "or") || (InterStack[i].top() == "OR"))
    { outFile<<".names "<<" "<<"I"<<i<<" "<<"I"<<i+1<<"
"<<I<<interpointer<< endl;
        if(i<interpointer)
            interpointer++;
        outFile<<"00 0"<< endl;
        i++; }
    if((InterStack[i].top() == "and") || (InterStack[i].top() == "AND"))
    {
        outFile<<".names "<<" "<<"I"<<i<<" "<<"I"<<i+1<<"
"<<I<<interpointer<< endl;
        outFile<<"11 1"<< endl;
        if(i<interpointer)
            interpointer++;
        i++; }
    if(i>=interpointer)
    { if((InterStack[i-1].top() == "and") || (InterStack[i-1].top() == "AND"))
        {
            outFile<<"I"<<i-2<<" "<<"I"<<i-1<<" "<<temp1<< endl;
            outFile<<"11 1"<<endl;
        }
        else if((InterStack[i-1].top() == "or") || (InterStack[i-1].top() ==
"OR"))
        {
            outFile<<"I"<<i-2<<" "<<"I"<<i-1<<" "<<temp1<< endl;
            outFile<<"00 0"<< endl;
        }
        break;
    }
}
}
}
}

```

/*****
 Most of this part of the program is from AND/OR package developed by Dr. Alanka
 Zuzek. This portion of the AND/OR package has been modified according to the
 convenience.

*****/

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<iostream.h>
#include<math.h>
#include "options.h"
#include "tree.h"
#include "output.h"
#include "aotr.h"
#include "hash.h"

#ifdef DMALLOC
#include <dmalloc.h>
#endif

int suma_all_nodes = 0;
int suma_loc_nodes = 0;
int suma_iso_nodes = 0;
float max_diff_iso = 0;
float max_diff_loc = 0;

int BLTsize;

BnetLookupTable* BLT; // BnetLookupTable
BnetNetwork *net; // network
int rmax; // r_max; See Kunz, Stoffel */

short* nJset;
short* namesJ; // auxiliary array for set of justifications
unsigned char* valsJ; // auxiliary array for their values

short* signs;
unsigned char* io_val;
unsigned char* kids_val;
unsigned char* node_val;

TreeNode *root; // temp root AND/OR grafa

```



```

int max_nodes;           // holds the maximal number of nodes;
                        // actual when building AND/OR graphs for all nodes
int *szAO_inp;           // holds the sizes of AND/OR graphs, but only for
                        // primary inputs; used for ord_alg = 0 (sizes AO)

#ifdef DEBUG_PATH
short* tempP;           // for paths
#endif

/*-----*/
/* function prototypes                                     */
/*-----*/

/**AutomaticStart*****/

/*Function*****

Synopsis   [Opens a file.]

Description [Opens a file, or fails with an error message and exits.
Allows '-' as a synonym for standard input.]

*****/

FILE *open_file(char *filename, char *mode)
{
    FILE *fp;

    if (strcmp(filename, "-") == 0) {
        return mode[0] == 'r' ? stdin : stdout;
    } else if ((fp = fopen(filename, mode)) == NULL) {
        perror(filename);
        exit(1);
    }
    return(fp);
} /* end of open_file */

```

```

/*****

```

Synopsis [Compute the maximal path of the generated tree]

Description []

```

*****/

```

```

double computeMaxSizeofTree(int depth)
{
    // number of internal nodes on the longest path
    int path = rmax;
    cout << "\n#max_path = " << path;
    double sum = 1;
    int i,j,temp;

    for (i = 0; i < rmax-depth; i++){
        temp = 1; // for the root node
        for (j = 0; j < i + 1; j++){
            temp = temp * (path - j);
            sum += pow((BLT->max_io()-1), i + 1) * temp;
        }
        return sum;
    }
}

```

```

/*****

```

Synopsis [Writing output tree to a dot file]

Description []

```

*****/

```

```

void dumpDot(TreeNode* node, char* title)
{
    toDot << "\ndigraph Before {\n";
    toDot << "size=\"10,18\";\n"; //
    toDot << "label=\"\"<<title<<\"\";\n"; // title of the graph
    toDot << "orientation=\"landscape\";\n"; //
    toDot << "center=\"page\";\n";
    node->writeDot();
    toDot << "\n}";
}

```

```

/*****

```

Synopsis []

Description [initialize and allocate auxiliary data structures
(globally declared), used for AND/OR enumeration and
recursive learning]

```

*****/

```

```

int init_and_alloc_auxData(recurOptions *option)
{
    BLT = new BnetLookupTable();
    BLT->write();

    BLTsize = BLT->size();
    int BLTmax_io = BLT->max_io();

    signs = new short[BLTmax_io];
    io_val = new unsigned char[BLTmax_io];

    // for creating justifying sets
    namesJ = new short[BLTmax_io + 1];
    valsJ = new unsigned char[BLTmax_io+1];
    nJset = new short[BLTsize-1];

    // these two are for learning
    kids_val = new unsigned char[BLTsize];
    node_val = new unsigned char[BLTsize];

    // this var hold the size of the biggest AND/OR graph
    max_nodes = 0;

    // initization of auxiliary pointers
    szAO_inp = NULL;

    // the array for ordering, used for ord_alg=0, and for ord_alg=1,
    // these algorithms operate on the sizes of AND/OR graphs
    if (option->ord_alg == 0 || option->ord_alg == 1){

        szAO_inp = new int [net->ninputs];
        // it must be initialized
        int i;
        for (i = 0; i < net->ninputs; i++)
            szAO_inp[i] = 0;
    }
}

```

```

    }
    else if (option->ord_alg == 2)
    // else we prepare array for the traversing through the tree
    {
        // under construction!!
    }
    // size of the BnetLookuptable is returned
    return BLT->size();
}

```

/******

Synopsis [Free auxilliary arrays]

Description [Free auxilliary arrays]

*****/

```

void free_auxData()
{
    delete[] kids_val;
    delete[] node_val;
    delete[] nJset;
    delete[] namesJ;
    delete[] valsJ;
    delete[] signs;
    delete[] io_val;
    delete BLT;
}

```

/******

Synopsis [AND/OR enumeration used for node assignment specified in
option]

Description [Perform the AND/OR enumeration for node specified in
option]

*****/

```

void do_AOtree(recurOptions *option)
{
    long localTime = util_cpu_time();

    // @ in this version hash table is allocated for each single tree
    // allocate Hash Table

```

```

Hash = new hsh_tabel(BLTsize*3);

root =
  new TreeNode(option->depth,option->node,
  (option->val) ? val1 : val0, option);

if (root){ // exists

  // learning if such option
  if (option->learn || option->dumpImps)
    root->learning();

  // dumping implications if such option
  if (option->dumpImps){
    root->append_dumpImps((option->val) ? 0 : 1);
  }

  // dump tree in Dot format to dumpfileDot
  if (option->dumpDot){
    cout << "\ndumping..";
    dumpDot(root, option->file_net);
  }

  // dumping in default (robust) format the final AND/OR graph
  if (option->dump)
    root->writeTree();

  cout << ", time = " << util_print_time(util_cpu_time() - localTime);

#ifdef DEBUG_PATH
  // this was used for debugging
  cout << "\nPaths with unjustified nodes are:";
  tempP = new short[BLTsize];
  root->write_rootPaths(rmax-option->depth);
#endif

  //Hash->write(1);
  Hash->statistics();
}

// this also if root = NULL
delete Hash;
}

```

```

/*****

```

Synopsis [Main function]

Description []

```

*****/

```

```

int main(int argc, char *argv[])
{
    recurOptions *option;    /* options */
    FILE *fp;                /* network file pointer */
    int pr;                  /* verbosity level */
    long localTime;          /* stores elapsed CPU times */
    int i;                   /* loop index */
    int ok;                  /* overall return value from main() */
    int BLTsize;
    long localTimeAll;
    char* out_fileimps = NULL; /* output file for implications */

    if (argc <= 1) {
        cout << "usage: recler [options] <circuit.blif>" << endl;
        cout << "recler -help" << endl;
        exit(-1);
    }

    // read control options
    option = new recurOptions();
    option->Read_recurOptions(argc, argv);

    // initialization
    net = NULL;              /* network */
    pr = option->verb;        /* verbosity level */
    fp = open_file(option->file_net, "r"); // input file

    // not possible to use from stinput
    i = strlen(option->file_net);

    // reading input blif
    if (i > 5 && strcmp(option->file_net+i-5, ".blif") == 0)
        net = Bnet_ReadNetBlif(fp,pr);
    else
        cout << "Warning input UNKNOWN!";

    if (net == NULL) {
        (void) fprintf(stderr,"Syntax error in %s.\n",option->file_net);
    }
}

```

```

    exit(2);
}

// end of reading input file
(void) fclose(fp);

// output file for implications if needed
if (option->dumpImps){
    out_fileimps = strdup(option->file_net);
    *strchr(out_fileimps, '.') = 0;
    out_fileimps = strcat(out_fileimps, ".1.imps");
    open_output(out_fileimps);
}

// prepare dot-output file if needed
if (option->dumpDot)
    open_outputDot(option->dumpfileDot);

// initialization and allocation of auxiliary structures
BLTsize = init_and_alloc_auxData(option);

// cheking options; @@ need to be done

// used when measuring time
localTimeAll = util_cpu_time();

// start building AND/OR graph for single graph, specified in options
if (option->single){

    do_AOtree(option);

} // end of building single tree
else{
    // loop for running recursive learning for all
    // nodes for values 0 and 1

    // if doing for ordering, AND/OR graphs will be build only for inputs
    int limit = (option->ord_alg==0||option->ord_alg==1)? net->ninputs:BLTsize;

    for (i = 0; i < limit; i++){

        // performing and/or for node i
        option->node = i;

        // first for logic "1"

```

```

    option->val = 1;

    do_AOtree(option);

    // and then for logic "0"
    option->val = 0;

    // do the tree
    do_AOtree(option);

}
if (option->ord_alg == 0 || option->ord_alg == 1)
    do_AOordering(NULL, option);
}
// closing output files
if(option->dumpDot) // dot-file
    close_outputDot();

if (option->dumpImps){ // implication-file
    close_output();
    delete out_fileimps;
}

// freeing auxiliary structures
free_auxData();

// freeing main allocated structures
Bnet_FreeNetwork (net);
delete option;

#ifdef DMALLOC
    dmalloc_log_unfreed();
    dmalloc_shutdown();
#endif
    exit(ok);
    /* NOTREACHED */
} /* end of main */

```