A Query-Processing Optimization Strategy for Generalized File Structures

Donna Marie Kaminski
Western Michigan University

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation
http://scholarworks.wmich.edu/masters_theses/1535

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
A QUERY-PROCESSING OPTIMIZATION STRATEGY
FOR GENERALIZED FILE STRUCTURES

by

Donna Marie Kaminski

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
April 1984
A QUERY-PROCESSING OPTIMIZATION STRATEGY
FOR GENERALIZED FILE STRUCTURES

Donna Marie Kaminski, M.S.
Western Michigan University, 1984

In processing a Boolean query against a non-inverted file, a subset of the query's keys must be selected. Only the records satisfying these keys need to be retrieved from the file. A dynamic programming algorithm, the Dual-to-minDF, is developed here for arriving at the optimum set of keys, i.e., the minimum number of external record retrievals. Previously suggested greedy methods yield feasible solutions which may or may not be optimal. This algorithm has exponential time complexity. However, it is useful for applications involving relatively short queries with duplication of keys within different conjuncts, and in cases where the files have large numbers of records per keyword. This makes the trade-off of I/O savings vs. extra processing costs worth it. Possible extensions of this research are suggested.
ACKNOWLEDGEMENTS

I would like to express much appreciation to Dr. Dionysios Kountanis for his valuable ideas and time spent in discussions for this thesis. I have gained much from his continual encouragement as well as the formal and informal learning experiences with him towards this degree. I would also like to thank Dr. Dalia Motzkin for her review and helpful suggestions on this thesis. And, as always, I am grateful to Char, my friend and companion, for his patience and uplifting diversions during these years.

Donna Marie Kaminski
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
KAMINSKI, DONNA MARIE

A QUERY-PROCESSING OPTIMIZATION STRATEGY FOR GENERALIZED FILE STRUCTURES

WESTERN MICHIGAN UNIVERSITY M.S. 1984
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ............................................................................................................. ii

Chapter

I. THE PROBLEM ........................................................................................................... 1
   File Structures ................................................................. 3
   Queries ................................................................. 6
   Record Retrieval ................................................................. 6
   Query-Processing Optimization Strategies ...................................................... 8
   Summary ............................................................................................................... 9

II. RELATED RESEARCH ......................................................................................... 11
   Review of Literature ...................................................................................................... 11

III. THE QUERY-PROCESSING OPTIMIZATION STRATEGY ..................................... 26
   Basic Concepts ................................................................................................. 27
   The Proposed Method .................................................................................... 31
   Formal Specification and Proof .................................................................... 37
   An Example ...................................................................................................... 43

IV. OTHER QUERY-PROCESSING OPTIMIZATION STRATEGIES ..................... 45
   Prime Key Word Technique ......................................................................... 46
   Minimum/Maximum Technique ........................................................................ 49

V. GENERAL PROBLEM-SOLVING ALGORITHMS ..................................... 54
   Greedy Method ................................................................................................. 54
   Dynamic Programming Method .................................................................... 57

VI. THE ALGORITHM’S COMPLEXITY ..................................................................... 59

iii

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER I

THE PROBLEM

A significant portion of computer time is spent on the retrieval of information from file systems and databases. The range of application areas varies from traditional data processing to automated bibliographic searches to airline reservation systems to natural language processing in artificial intelligence. Some applications find sequential accessing to be quite suitable. However, most applications today find that accumulating and sequentially processing batches of requests is much too slow, is not ideally suited for the nature of the problem, and does not take adequate advantage of on-line, direct access storage device capabilities. Random access can provide "immediate" answers to the user's questions by considerably reducing the number of records or items that must be considered.

It should be noted that the file-access efficiency problem is also of concern in the file update function. This will not be directly considered in this study. However, much of the discussion here is certainly generalizable.

The continual question, however, is "Can a retrieval system be made efficient?". Hardware technology is developing rapidly, e.g., faster, higher density disks and drums and more varied high-speed access storage like magnetic bubble memory and large core storage
are now available. More user-friendly query languages are being developed for database systems. Research is continually being done on constructing faster searching algorithms and more efficient data structures.

The wait-time for the user in processing a request is largely due to the actual retrieval of records from the external file once the query has been specified. Most file systems cannot afford to keep but a minimum of information actually resident in main memory. Because of their size, files are almost always kept on external storage devices where accessing data involves milliseconds or tens of milliseconds as opposed to only a few microseconds for accessing data in internal memory. Thus, a great time-saving would result if the time for file accessing itself, as opposed to the time for interpreting the query or formatting of the response back to the user, could be reduced. One approach to the problem is to optimize block size to minimize data transfer time for unwanted data. Another approach is to store the more frequently requested files/records where they can be accessed more rapidly either through reducing seek time or through minimizing chaining time to locate synonyms resulting from hashing techniques (Hanson, 1982). Both of these approaches involve fine-tuning of the file structure to fit the types and range of queries expected to be most frequent. Others are concerned with optimizing directory structures, optimizing queries and developing better hashing approaches. Many other approaches to the problem are discussed in the next chapter.

This research provides an algorithm for minimizing the number of
unnecessary records which must be retrieved in order to retrieve the records which do satisfy a Boolean query. It is based on use of the generalized file organization of which other file structures can be seen as a particular case (Hsiao & Harary, 1970). With such a file structure, all of the file accessing information is not available in the directory, but instead allows pointer information in the file itself. This structure complicates the retrieval procedure, as will be shown, as compared with simpler file organizations.

File Structures

Any random access file must have some mechanism for arriving at the address where a particular record can be found. In effect, the user’s request must be transformed into a set of absolute or relative record locations. A file system might be set up so that the user can specify the actual address, e.g., "EMPLOYEE-NUMBER=1825", where this employee’s record is stored at relative location 1825. This, however, would not work if the file had multiple records for a requested key. An alternative is to use a key-to-address transformation, e.g., "SOC-SEC-NUM=366549450", where this record is stored at $F(366549450) = 1293$, i.e., through some hashing function on the key-value. However, neither of these approaches would work for a file which is to be accessed by more than one keyword type, e.g., both NAME and EMPLOYEE-NUMBER.

A key here is an attribute-value pair such as NAME=Bentley or AGE=35. The keyword type is the attribute. All records in the file are really just sets of attribute-value pairs. All attributes in the
file may potentially be used for retrieval. However, generally only those attributes and their values which the user will use to specify the desired characteristics are referred to as keys for the file.

A directory is generally used for a multiple-key file. Every key is listed along with some/all the record addresses which satisfy that key. The directory contains a number of different indices, one for every attribute or keyword type. Each index contains an entry for each different value in the file for that particular attribute. Then, all records in the file are implicitly or explicitly listed for only one value-entry in each index.

There are, however, many different directory structures which are possible. At one extreme is the inverted file structure where each record address in the file is listed in each index for some attribute-value pair. At the other extreme is the multilist file structure where only one record address is stored for any attribute-value pair, in effect, the head pointer for a linked-list.

For files of any significant size, with multiple-key access, a complete directory is generally too large to be fully contained in internal memory for inverted file structures. This is because each address in the file must be included in the directory under each of the different attributes, e.g., the NAME index, the AGE index, etc. Multilists, on the other hand, have relatively small directories. However, accessing such files is much more time-consuming in order to follow the pointer list for a particular key.
Files may thus use a structure which is a cross between an inverted and a multilist file organization, e.g., a partially inverted file or a cellular multilist. With this type of structure, only some of the record addresses are kept in the directory for any particular attribute-value pair. Each of the records which is specified in a directory index contains a link field pointing to another record which has the same keyword value but is not itself listed in the index. Each of these new records, in turn, points to another such unlisted record with the same keyword value. This continues until all records with a particular attribute-value have been linked (indirectly) to one of the records pointed to in the directory. Various criteria may be used to determine the number and length of sublists for a keyword value and the assignment of records to specific sublists. For example, the length of each sublist may be less than or equal to some constant. Or, for a cellular multilist, a sublist would consist of all records in a particular cell, e.g., track, sector, page. Then, since the directory contains a number of different indices, i.e., one for every attribute, each record contains that number of pointer fields, i.e., one for each attribute.

The generalized file organization used in this study is, in effect, a partially inverted file, but with some additional information. Each directory entry has the form: $K_i, N_i, H_i, A_{i1}, \ldots, A_{ih_i}$, i.e., the attribute-value key, the number of records in the file with this key, the number of head pointers in this directory entry and the $H_i$ different head pointer record addresses. All other file organizations can thus be seen as special cases of this generic
file structure (Hsaio & Harary, 1970).

Queries

A query is the user's request for all records with (or without) the stated characteristics, phrased in terms of some specified query language. Each desired characteristic can be stated in terms of an attribute-value pair, i.e., a key. Note that all possible characteristics which the user might want to request must be set up as accessible keys in the file directory. Queries can be one of several types (Horowitz & Sahni, 1978):

- **Simple Queries**: specify a single key, e.g., `MAJOR=CS`.
- **Range Queries**: specify a range of keys, e.g., `AGE=60 TO 65` (or an implicit range e.g. `INCOME<30000`).
- **Functional Queries**: specify a function of key values in the file, e.g., `SALARY>MEDIAN`, or `AGE<AVGAGE`.
- **Boolean Queries**: specify a combination of the above types using logical connectors AND, OR, and NOT, e.g., `(GENDER=M AND AGE<26) OR (GENDER=F AND AGE<24)`.

The first three types can be viewed as special simple cases of the Boolean query. Tahni (1977) considers an alternative form, the fuzzy query, briefly discussed in the next chapter. Such non-conventional query types are not considered here. This study deals with Boolean queries although it does not specifically consider functional queries.

Record Retrieval

In processing a query, the actual retrieval of records from the external storage device is the most expensive operation. Obviously

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
those records which actually satisfy the query must all be retrieved in order to pass that information to the user. An overhead problem arises, however, when any other records are also retrieved. Is this avoidable? Is it possible to determine exactly which records satisfy the query prior to retrieving any records? That depends. For inverted files, the answer is yes, since all record addresses and their attribute-values are actually available in the directory. The answer is no for other types of file organization for anything but a simple query. This is because only some of the record addresses for a particular key are present in the directory. Potential excess records must be retrieved in order to follow all the link-lists for a particular key. This is discussed at length in chapter III.

The goal, then, is to minimize the number of these excess records which must be retrieved. Actually, the goal is to minimize the number of I/O's by taking blocking into consideration. This is considered further in chapter VI. Several alternative strategies are presented for minimizing record retrievals in addition to the one proposed in this research which guarantees the minimum overhead. However, the price for performing fewer external record retrievals is an increase in internal processing. This often proves to be a beneficial trade-off considering that up to a thousand machine instructions can be executed during the time it takes to access just one record from an external file. However, this is dependent on such things as the efficiency of the algorithm, the types of queries, the size of the files, and the distribution of records over the range of values for an attribute. This is discussed further in chapter VI.
Query-Processing Optimization Strategies

Several algorithms have been suggested for minimizing the number of excess records retrieved in processing a query. All assume that the Boolean query is in disjunctive form. All incorporate the concept that only one key in any conjunct need be used for retrieving all records which satisfy that conjunct. The other keys in the conjunct can be compared against the retrieved records.

The two methods proposed in the literature thus far are based on the greedy method. The prime key word and the related minimum/maximum (Ni/Fi) both, in effect, repeatedly parse through the query, each time selecting the best key for retrieval. The two methods differ only in how they define "best". The PKW technique selects the key with the minimum number of records. The min/max selects the key with the minimum number of records relative to the number of query conjuncts that the key covers. At each selection, one or more conjuncts is covered. This process is repeated until no conjuncts remain uncovered, each time considering only those conjuncts not yet taken care of. Some keys may actually take care of several conjuncts. The two methods are both heuristics which produce feasible solutions which may or may not be optimal. They are explained in chapter IV.

The processing strategy proposed in this research guarantees an optimal solution based solely on information within the directory, i.e., only the minimum number of excess records will actually be retrieved. The algorithm is a dynamic programming strategy in which
all likely feasible solutions are produced. All possible feasible solutions need not be generated, those which can be covered by simpler solutions can be eliminated at each stage in the transformation. Each solution is a subset of keys to be used for retrieval which takes care of all conjuncts in the query. The best solution can then be selected based on an optimization criterion. The overall goal is to select the subset of keys which leads to the fewest record retrievals. One selection measure is based strictly on the number of records each key in the subset has. Another possibility is to consider the likelihood that several keys in the subset might access the same record. Chapter III discusses this in greater detail.

Summary

These methods and concepts are explored in greater detail in the chapters which follow.

Chapter II looks at other research which has been done on the problem of improving access efficiency. Many studies have considered optimization of query-processing. However, few consider this particular aspect of it, i.e. minimizing the number of records retrieved. There has been particularly little attention paid to a generalized file structure or any such pointer-in-file organizations. Most work focuses on inverted file structures or relational databases.

Chapter III presents the proposed optimization strategy, the Dual-to-minDF. It first gives a background of the basic
considerations in processing Boolean queries. The proposed method along with examples are presented informally and an intuitive proof given. A formal theorem and proof are then presented.

Chapter IV explains the two alternative strategies: the prime key word and the minimum/maximum. Examples are given to demonstrate cases where each does provide an optimal solution, and cases where they fail to do so.

Chapter V explains and explores the two general problem-solving approaches to the problem: the greedy or hill-climbing method and the dynamic programming method. The reason for the greedy strategy's potential for failure is also discussed.

Chapter VI considers the general processing time characteristics for the proposed Dual-to-minDF algorithm. There is also discussion as to situations where this more complex strategy may and may not be usable in actual query-processing applications.

Chapter VII provides a summary of the work. A list of possible open questions for further research on this problem is then spelled out.
Given the amount of computer time and resources devoted to retrieval of information, it is not surprising that this has been an important topic of research. As databases, data banks, and computer and information science become more popular, the interest of retrieval efficiency is likely to continue to grow.

The literature review presented here is a general survey of research related to retrieval efficiency. It is certainly not a complete, comprehensive review given the size of the field. Few studies were found which dealt with query optimization, per se. There was also relatively little work which considered retrieval efficiency for anything like a generalized file organization, e.g., multilist, partially inverted list. Such a file structure obviously presents different problems as compared with an inverted directory structure which is discussed in detail in the next chapter.

Review of Literature

Lefkovitz (1969) is still the classic and often quoted text on direct access file structures. The basic file organizations are described along with explanations of retrieval and update techniques for each.

Cardenas (1973) presents a model and simulations comparing
various file structures as to their average query retrieval time. The tests consider queries with different levels of complexity: an atomic condition, i.e., one attribute-value relation, an item condition, i.e., a disjunction of atomic conditions, all with the same attribute, a record condition, i.e., a conjunction of item conditions and a query condition, i.e., a disjunction of record conditions.

Hsiao and Harary (1970) develop the concept of generalized file organization, a formal specification of a generic file structure of which other file structures are a special case. This is the file structure view used in this study. Each directory entry has the form of \([K, N, H, A_1, ..., A_H]\), i.e., the attribute-value key, the number of records in the file with this key, the number of head pointers in this directory entry and the \(H\) different head pointer record addresses. The directory thus resembles a partially inverted file directory in that part of the retrieval information is kept in the directory, and part is kept as pointers in the file itself. The authors specify directory search and file search algorithms which are used in both types of retrieval strategies: the serial and the "parallel" processing algorithms for DNF Boolean queries. Their method for selecting which query keys to process against the file is not spelled out in detail. They suggest selecting the prime key word from each conjunct, i.e., the key with the smallest \(N\). However, they do not specify a procedure for doing so.

The study here is chiefly concerned with minimizing the number of records retrieved in response to a query. This perhaps does not
accurately state the real concern of retrieval efficiency, i.e., minimizing the number of I/O's. For example, if two records to be retrieved happen to reside in the same block/bucket/sector, and they are requested one immediately following the other, then only one I/O is needed to get the block containing the two records. This issue is discussed in greater detail in chapter VII. The following several studies focus on the idea of block as opposed to record accesses.

Chan and Niamir (1982) are concerned with reducing the number of blocks which must be retrieved in order to satisfy a query. They provide a general formula for computing the expected number of blocks on which a random sample of records from a file reside under different blocking/packing conditions. In the simplest case the expected number of block accesses to retrieve \( r \) records from a file of size \( n \) is:
\[
( n / b ) * ( 1 - ( C[n-r,b] / C[n,b] ) )
\]
where the blocking factor, \( b \), is an integral multiple of \( r \), \( n \) is an integral multiple of \( b \) and \( C[i,j] \) is the number of combinations of \( i \) things taken \( j \) at a time. Various approximations for this formula are provided as well. Yao (1977) also provides an approximation method for estimating block accesses on a database system.

Cheung (1982) also provides a formula for estimating the number of secondary storage blocks accessed by a particular user request. His work extends earlier estimation techniques by considering cases where duplicate record addresses may be part of the file access set.

Both of the above provide useful estimations for incorporating into the technique proposed in this study. Once the possible
retrieval sets of record addresses have been arrived at, the comparison function might use block estimates rather than record counts as a way of arriving at the likely optimal set.

The following studies focus on the same issue but utilize multi-key hash files rather than the partially inverted directory structure used in this study.

In a paged memory system, the time for accessing a record on the current page in core as opposed to a record not on the current page is considerably less, a difference of several orders of magnitude. Thus optimization efforts need to be directed at reducing the number of new page accesses, not the actual number of records, per se. Rothnie and Lozano (1974) suggest structuring the file based on the probabilities of anticipated retrieval requests using multiple key hashing. In effect, all records which map into the same characteristic tuple are stored on the same page. The major problem, though, is in selecting the best hashing function. They also present a retrieval algorithm based on multiple key hashing as well as a technique which combines the advantages of this structure and the inverted file structure.

Moran’s (1983) algorithm minimizes the number of buckets retrieved to answer a query. Individual buckets are found by a multi-key hashing function. Moran demonstrated his technique to be NP-hard and describes several alternative heuristic algorithms involving local searching.

A whole body of research has developed around deriving efficient hashing techniques in order to minimize the number of synonyms.
Obviously the record retrieval process is speeded up considerably if the hashing function produces the address of the desired record directly, rather than having to do further searching/retrieving in processing all the synonyms. This parallels one of the problems with a partially inverted directory structure, used in this study, where a thread of pointers must be followed, and hence extra records retrieved, in order to locate the desired records. Minimizing the number of retrievals of non-satisfying records is desirable. Hence reducing the amount of overflow attributable to a particular hashing function is desirable.

Another approach to improving access efficiency when hash files are used is to consider alternative methods of handling the overflow problem. The following studies look at this.

Quittner et al. (1981) compare the access times required for two different methods of handling overflow/synonym problems in hashed files. They find that open addressing, i.e., using the "next available open spot", is more efficient than the chaining technique, i.e., separate overflow buckets with pointers to synonyms. They also find that small bucket size results in better access time than large bucket size.

Larson (1982) and Clapson (1977) also consider optimizing file access time by focusing on key-to-address hashing formula and various schemes for handling overflow records.

Each file or database structure may raise its own unique access efficiency problems. In the case of this study, a partially inverted directory structure presents a record access overhead problem as
compared with an inverted file structure. This is due to the fact that only part of the record access information is stored in the directory. The rest is stored in the file itself as record pointers.

The following studies examine special access efficiency problems of distributed database networks as opposed to one centralized set of files.

Sacco and Yao (1982) focus on query optimization when using distributed database systems. This differs from query processing on a centralized system, as used here, in that the dominating time component is the network transmission speed rather than the actual disk access time, of concern here, by several orders of magnitude. The strategy in the present study would thus not be directly usable for distributed systems since location site is not incorporated in the model selection process.

Grapa and Belfod (1977) are concerned with optimal allocation of files among various network sites for a distributed database. Obviously, this has an impact on file access and update time. The authors present several aids which allow an a-priori evaluation as to which sites are definitely not part of the optimal solution. This helps reduce an impractically large problem, time-wise, into one which is of acceptable size in many instances.

Others have proposed alternative file structures as a way of alleviating particular problems and thus improving overall access time. Most of these involve an alternative directory organization, often a variant of an inverted directory.

March (1983) examines various record structures to consider in
designing a database since this has a major impact on retrieval and maintenance time costs. "The task of record structuring is to arrange the database physically so that (1) obtaining 'the next' piece of information in a user request has a low probability of requiring physical access to secondary memory and (2) a minimal amount of irrelevant data is transferred when secondary memory is accessed." (March, 1983:45-6).

Hanani's (1977) work focuses on optimizing query processing by speeding up the actual evaluation of each record as to whether it satisfies the query or not. However, his work is based on the searching of a sequential file, unlike the present study which considers a random access file. Hence, each record in the file must be evaluated. However, rather than testing a record for all attributes in the query, only a subset need be used. These are selected by considering the cost of checking a particular attribute, i.e., the number of assembly language instructions, and the probability that a record has that attribute, i.e., the proportion of records in the file with that particular attribute.

Papakonstantinou (1982) builds on Hanani's work and proposes use of decision trees and decision tables instead of query trees for developing optimal query sets. The optimal evaluations of Boolean expressions, where different variables have different costs and different probabilities of being true, is really a special case of finding the optimal decision tree for a decision table for which algorithms do exist.

Cardenas (1975) and McDonell (1977) both raise an important
issue with respect to optimization of query processing: the problem of the directory itself. Although not specifically addressed in this research, the directory search problem is certainly relevant to the overall process. Both studies focus on inverted file directories, but similar concerns are applicable to the generalized file organization of interest here. However, the directory in the former case would be considerably larger. The directory itself may become another file search problem with decisions as to the best structure and search strategy. This, obviously, is of much greater concern if the entire directory cannot be stored internally. The generally assumed simple sequential organization for the directory is often a poor choice. Cardenas suggests maintaining the directory as a hierarchy of linear indices, while McDonell suggests a hash addressed random access method.

Nicklas and Schlageter (1977) propose an alternative index structure for inverted data files which both preserves the fast set-operation-based retrievals and improves the update performance characteristics. They use a filter organization which organizes the index based on attribute values rather than the attributes themselves.

Hatzopoulos and Kollias (1982) are concerned with selecting optimal file maintenance points by considering the file's structure. Obviously data access time deteriorates considerably, depending on file organization, due to structural inefficiencies caused by file updating.

To process a Boolean expression query against an inverted file
system, set operations are used to merge address lists for the keywords in the query. This minimizes the number of records that must be accessed from external storage. Putkonen (1980) provides an algorithm which minimizes the CPU time needed for merging by choosing an optimum order of doing union and intersection operations.

A number of alternative file structures use multiple attributes together as key entries in the directory. This speeds up processing if records are always requested with such multi-attribute keys. Thus a particular application, i.e., a specified query type, may lead to a more efficient access system. The following studies consider this.

Pfaltz et al. (1980) present a secondary attribute key retrieval technique, the indexed descriptor file, which has superior access time to an inverted file when several attributes are specified. Each record has a bit pattern associated with it based on selected attributes in the file. Each block of records has an associated bit pattern which is the OR-ing together of its constituent record bit descriptors.

Wong and Chiang (1971) propose a retrieval system in which the query structure, e.g., range, single value, determines the optimal structure for the file address lists. File directory lists are not keyword lists, i.e., a list composed of all record addresses with a particular attribute-value combination. Instead, record addresses are grouped into disjoint lists resulting from the conjunction of the keywords in the record and the compliments of keywords not in the record. That is, all/only records sharing the exact same set of keywords belong to one atom. Each list then corresponds to a
Boolean function of the keywords. This structure facilitates retrieval in that the intersection of sets is never necessary, it is built into the structure. And the union of sets requires no elimination of duplicates, since the lists are disjoint. For a Boolean query in DNF, the algorithm only needs to check whether any conjunct is one of the atoms or lists in the file directory.

Papakonstantinou and Kontos (1974) offer an algorithm for organizing and accessing a table of satisfiable terms, i.e., all logical products of keywords satisfied by at least one record. Their modification and optimization of the basic "table of satisfiable terms" directory is possible because of certain restrictions they place on the queries and keywords. Entries in the table are either pointers to other table entries or are record access information. The structure is a modification of the trie structure which permits an efficient table search time. Their search algorithm actually has a retrieval time independent of the table size.

Some work has been done specifically on query optimization for relational databases. A number of studies provide transformations which "optimize" relational queries, but which are really heuristics since they do not promise equivalent expressions of least cost. Others have developed algorithms which do really optimize query expressions but which have exponential time complexity. Others provide polynomial time algorithms for certain subclasses of query-types. These are discussed below.

Aho et al. (1979) develop an algorithm for optimizing a limited class of relational database queries. Queries are stated as
expressions where the operands are represented in a tableau or matrix of information and the operations are restricted to select, project and natural join. The optimization process finds a minimal tableau equivalent to the given query tableau. This equivalence algorithm is really an NP-complete problem. However, with restrictions placed on the types of allowable queries, the problem can be solved by a polynomial time algorithm of $O(n^{**4})$ complexity. The restriction limits the types of queries to only those expressible by simple tableaux.

Further extensions of this method are developed by Sagiv and Yannakakis (Johnson and Klug, 1983). They reduce the algorithm to $O(N^{**2})$ complexity and add several new subclasses of queries to those which can be minimized in polynomial time.

Johnson and Klug (1983) present a poly-time algorithm for minimizing an even broader class of conjunctive queries, the fan-out free queries. Such queries are more general and powerful than "typed" queries since they also allow queries about the transitive properties of a database, e.g., child-grandparent relations, rather than only queries about fixed attributes of a relation. The run time for their minimization algorithm is $O(n^{**3})$ with a time $O(n^{**4})$ for determining the equivalence of two query expressions. Their minimization algorithm constructs an implication graph for the query, finds and tests the implication closure of each conjunct pair vertex and finds a shrinking self-homomorphism on this for query Q.

Wilson (1977) presents a method for minimizing Boolean
expressions which handles a large number of Boolean variables. It is based on the Quine aproach and no complexity analysis is provided.

Some types of file and database system applications must necessarily introduce an added burden to the retrieval routine, that of maintaining information confidentiality and security. This adds additional checking steps as to whether a particular user can be given certain fields, records or combinations. This in itself becomes a type of query which must be processed, e.g., USER-NUM = ??? AND KEY = SALARY. The following studies look at the issue of file security during query processing.

Sicherman et al. (1983) propose a method for the censoring information by wedding certain artificial intelligence techniques to an information retrieval system. The query-answering system keeps secrets by refusing to answer particular combinations of questions.

In their discussion of future database computers, Baum and Hsiao (1976) extend query languages and data base accessing to include the notions of data security and integrity. Both of these, however, extract a performance penalty from the retrieval system due to the repeated extra symbolic query key-to-address transformations needed, i.e., the name-mapping problem. Two organization factors reduce disk access time: physically clustering, on a single cylinder, of data likely to be simultaneously accessed and segregating the database from its mapping information, i.e., minimizing pointers within the file and increasing such information within the directory. These concepts suggest use of certain physical devices and structures in the computers used for database systems in the
The area of user-interface may at first appear to be peripheral to the topic of information retrieval efficiency. However, upon further consideration, it is apparent that the relative ease with which a user is able to submit a query contributes to the optimization of overall query/response time. In terms of overall access time, much emphasis is placed on minimizing external file handling since it's time is several magnitudes greater than the time for internal machine instructions. Likewise, the time for the initial human entry of the query is several magnitudes greater than external file accessing time.

Reisnes (1981) presents a review of work done on database query languages from the standpoint of the user. Several dimensions are noted on which to compare relative ease of use: syntactic form, e.g., fill in a 2-dimensional menu vs. spell out the query in linear, left to right form, procedurality, e.g., specify the actual retrieval procedures vs. merely specifying a set of desired characteristics, and data model, e.g., the user's conceptual view of the data as relational, hierarchical, or network.

Williams and Tou (1982) address the problem of database access efficiency by developing an optimal user interface called RABBIT. This query language is based on the psychology of human information retrieval. How do humans store, request, search and select information? They suggest that this is done through interactive construction of the query and critique of example instances. This approach should contribute as much to the casual query user's
retrieval time as does the internal search optimization effort.

Tahni's (1977) work on query processing considers techniques for handling "fuzzy" queries. This goes beyond the scope of the research here, but provides an interesting extension to query processing in the direction of greater ease of use for humans. Conventionally, queries are specified in precise terms and a record either fits the query or it doesn't, e.g., AGE=20 where the values for age in the file are integers. The fuzzy concept is an attempt to more closely mimic human intelligence and the real world where the knowledge base, the requests for information and the reasoning process are often imprecise and left at approximations, e.g., almost, great, several, very. This is in contrast to the conventional two-value, either/or logic of most file retrieval systems. The notion of "fuzzy" can be applied to both the query and/or the database. An example of a fuzzy query is \[ \text{AGE} = \text{old} \text{ AND } \text{SALARY} = \text{not very high} \]. To process such a query against a conventional database, for each attribute, each element in its universe of discourse, i.e., range of possible values, must be assigned a likelihood weight by a subjectively defined function for each linguistic value, e.g., few, several, many. Then each record is considered as to how pertinent or satisfying it is to the query as opposed to the conventional all/nothing fit.

Kolodner (1983) looks at an intelligent retrieval system using natural language queries. Extra processing is obviously necessary in such circumstances in analyzing the query in order to extract meaning so as to select the appropriate search category.

Meadow's (1973) text discusses retrieval system efficiency from
the perspective of information science and document retrieval. In such applications, unlike conventional file systems, the type considered in this study, the set of records the user is interested in retrieving is not a discrete, well-bounded set which can be precisely defined. Rather, the user wants a set of some, perhaps all relevant records. However, the concept of relevant is a potential source of imprecision error. Which records or documents are defined to be relevant to a particular query/keyword depends on the original indexer, the keyword vocabulary, the syntax of the document, etc., which may or may not be in agreement with a particular user's definition at the time of the request.
CHAPTER III

THE QUERY-PROCESSING OPTIMIZATION STRATEGY

Any file organization which does not store all record addresses in its file directory, e.g., generalized file organization, multilist organization, necessitates the retrieval of surplus records in processing a query. Lefkovitz (1969) and Claybrook (1983) provide a full discussion of various file organizations. That is, in order to pass to the user those records which satisfy the Boolean query, not only will the satisfying records need to be retrieved, but additional records which satisfy only a part of the query must be retrieved in following the link lists for the subset of query keys. For example, for the query \((K_1 \text{ AND } K_2)\), all records with the key \(K_1\) would be retrieved and each checked for \(K_2\). Some of the retrieved records would satisfy \(K_2\), and hence satisfy the query and be passed to the user. Other records wouldn’t, and hence are surplus record retrievals.

Since most files and data bases are quite large and thus reside on external storage, the actual record retrieval time is the most expensive operation in processing a query. In all but the simplest of Boolean queries, the number of surplus records retrieved can become quite sizable.

The technique presented here demonstrates a strategy for retrieving the minimum number of extra non-satisfying records and...
hence the minimum number of records overall. The technique will first be described and demonstrated informally along with an intuitive proof. A formal theorem and proof will then follow.

Basic Concepts

Simple Query Processing

Consider first a simple query, say \((K1 \text{ AND } K2)\). The program should pass to the user those records in which both \(K1\) is true, e.g., age = 35, and \(K2\) is true, e.g., job = professor. If this query were processed against an inverted file, all \(N1\) record addresses, where \(K1\) is true, and all \(N2\) addresses, where \(K2\) is true, would be contained in the file directory and thus immediately available. Prior to the retrieval of any records, the program would consider which record addresses are in both set \(S1\) in the directory and \(S2\) in the directory, i.e., the intersection of \(S1\) and \(S2\). Then, only those records in the intersection would have to be retrieved from the file. No extra non-satisfying records would have to be retrieved, and \(N_r = \text{NUM (INT (S1,S2))} \leq \text{MIN (N1,N2)}\).

However, in a multilist or generalized file organization, all record addresses are not immediately available in the directory. The program must thus retrieve some records to get the addresses of other records which satisfy a key. In order to identify all records with key \(K1\), the program will necessarily retrieve all \(N1\) records. This concept suggests one rather inefficient approach to the problem, \((K1 \text{ AND } K2)\), i.e., retrieve all \(N1\) records, retrieve all \(N2\) records,
then take the intersection of $S_1$ and $S_2$ as is done with inverted files. But this would result in $N_r = N_1 + N_2$ records being retrieved, and any record satisfying the query would be retrieved twice.

A better approach is to select either keyword, say $K_1$, and retrieve only those $N_1$ records. Any record satisfying the query ($K_1$ AND $K_2$) is bound to be part of the set $S_1$. The program would then check each record in $S_1$ to see if $K_2$ is true or not. In this case, $N_r = N_1$ records are retrieved.

But should $K_1$ or $K_2$ be selected for the retrieval stage? This is the central question in the proposed optimization strategy. The overall goal is to minimize the number of records retrieved because of their time expense. Thus in the above simple query one would select the key which had the minimum number of records satisfying $K$. Thus, $N_r = \text{MIN}(N_1,N_2)$. These $N$'s, of course, would need to be stored in the directory. Likewise, for a query ($K_1$ AND $K_2$ AND $K_3$ AND $K_4$), the program would select the $K$ with the minimum $N$ for retrieval, say $K_3$. This is called the prime key word. Then each of these $N_3$ records would be checked to see if $K_1$, $K_2$ and $K_4$ are all true.

Now consider simple queries such as ($K_1$ OR $K_2$). With an inverted file organization the query would be processed by taking the UNION of all $K_1$ and $K_2$ record addresses from the directory. This set of addresses would be sorted, and duplicates eliminated, prior to retrieval for greater efficiency. This is possible only because all addresses are immediately available in the file directory. Thus, not all $N_1 + N_2$ records would be retrieved, rather, $N_r = N_1 + N_2$ -
To process (K1 OR K2) for a generalized file organization, the program would retrieve all of the K1 and the K2 records, without ordering or removing duplicate record addresses. However, prior to sending any of these records to the user, a special "elimination of duplicates" provision would need to be implemented. That is, if N1 > N2, all K1 records would automatically be sent to the user. Then, each K2 record would first have to be checked as to whether K1 were true and, if so, not be sent to the user. By processing these keys in order of decreasing N's, the number of key comparisons is reduced. However, all N1 + N2 records would still have to be retrieved.

Complex Query Processing

Consider a more complex Boolean query, say [(K3 AND K4) OR (K4 AND K5)]. The query is in disjunctive form, DF, that is, the "OR-ing" together of conjuncts which are sets of keys "ANDed" together. Any query can be expressed in this form by repeatedly applying the properties of a Boolean algebra, e.g., distributive, commutative, associative, deMorgan's laws. Queries in DF are conceptually easier to deal with. A record which satisfies any one conjunct is passed to the user without further consideration of other conjuncts. The program should, however, consider records previously sent to the user to avoid duplication. The subsequent discussion will assume that the query is already in DF.

If the file organization is inverted, the UNION and INTERSECTION
set operations would be applied to the sets of record addresses in
the file directory. Then only the resulting records would be
retrieved. And for the above query, \( Nr = \text{NUM} (\text{INT}(S3,S4)) + \text{NUM} (\text{INT}(S4,S5)) - \text{NUM} (\text{INT}(\text{INT}(S3,S4), \text{INT}(S4,S5))) \).

The strategy is much more complex when processing queries
against a generalized file. The steps for simple query processing
above suggest the following. An AND indicates: select the \( K \) with the
minimum \( N \) and retrieve all such records. An OR indicates: retrieve
all records from both operands. Consider the complex query above.
Suppose the number of records satisfying each key were \( N3=30 \), \( N4=40 \)
and \( N5=50 \). The above method, discussed in detail in the next chapter
as the prime key word strategy, suggests retrieving the 30 \( K3 \) records
and checking if \( K4 \) is true for each, then retrieving the 40 \( K4 \)
records and checking if \( K5 \) is true for each. Hence 70 record
retrievals are needed. This method, however, does not always lead
to the most efficient solution. Obviously fewer records would be
retrieved if only the 40 \( K4 \) records are retrieved, then each checked
for whether \( K3 \) or \( K5 \) is true. This is because \( K4 \) appears in two
conjuncts, i.e., \( K4 \) covers both conjuncts. A key or set of keys is
said to cover a conjunct if it leads to the retrieval of all records
which satisfy the conjunct, and perhaps some additional non-satisfy-
ing records as well. Thus the decision as to which key to select
for retrieval from a conjunct is not independent of the keys
appearing in other conjuncts in the query.

This suggests a second optimization goal. The first was to
choose a key with minimum \( N \). The second is to additionally choose a
key which covers the maximum number of conjuncts, i.e., the key with maximum frequency of appearance in the query. This leads to a second key-selection method: MIN(Ni) vs MAX(Fi) or MIN(Ni/Fi), discussed in detail in the next chapter as minimum/maximum strategy. This method does give the optimal solution in the complex query above, [(K3 AND K4) OR (K4 AND K5)], i.e., retrieval of the 40 K4 records. But it does not work when applied to a similar query, [(K1 AND K4) OR (K4 AND K5)]. On this query this method would retrieve 10 K1 record plus 40 K4 records. This is because early selections of keys for retrieval are not independent of later selections of keys for retrieval.

In decision-making terms, both of these techniques prune alternative decision branches too soon, based on short-term, supposedly independent decisions at each step. The first method doesn't even consider the interdependence of the initial information, the second method does. Neither method considers the interdependence of the decisions, however. No look-ahead or backtracking is done. A series of decisions is made as opposed to one set of decisions being made which are just implemented serially. A feasible solution will result which may or may not be the optimal solution.

The Proposed Method

The following method does guarantee the optimal solution to the problem, based on all information available in the directory. It arrives at a set of keys which both covers all conjuncts in the query and results in the minimum number of record retrievals. This method
is based on the selection of a global "best SET of keys" as opposed to a series of selections of the local "best keys". First, the intuitive explanation of the method is presented.

General Description

The strategy involves two processes: the generation of feasible solutions and the choice of the best solution from those feasible. In the first stage, all simple feasible solutions are generated. All possible feasible solutions are not generated. Those that can be reduced to other simpler, feasible solutions are eliminated. For example, if both [K1,K3,K4] and [K1,K3] are possible feasible solutions, i.e., cover the query, then only [K1,K3] will be retained as a simple feasible solution since it leads to retrieval of fewer records than [K1,K3,K4].

The second stage involves choosing the optimal solution from among the feasible solutions. Several optimizing criteria can be applied. The simpler one involves comparing the solutions as to the straight sum of the number of records to be retrieved. An alternative technique would also take into account the probability of duplicate records being retrieved within a solution when comparing the possible solutions' sums.

Stage One

The first stage can be explained in greater detail by considering the following general query in DF. The first "subscript" for K is the conjunct number, 1,2,..m, the second is the
key number within the conjunct, \(1..x, 1..y, \ldots, 1..z\). Any \(K_{ij}\) may equal any \(K_{uv}\), if \(i \neq u\).

**QUERY:**

\[
(K_{11} \text{ AND } K_{12} \text{ AND } \ldots \text{ AND } K_{1x})
\]
\[
\text{OR } (K_{21} \text{ AND } K_{22} \text{ AND } \ldots \text{ AND } K_{2y})
\]
\[
\text{OR } \ldots
\]
\[
\text{OR } (K_{m1} \text{ AND } K_{m2} \text{ AND } \ldots \text{ AND } K_{mz})
\]

For any conjunct to be covered, at least one of its keys must be part of the solution set of keys, as explained for simple query processing. Thus, any feasible set of keys selected for retrieval, including the optimal set, must include:

- one of the keys in conjunct 1
- one of the keys in conjunct 2
- . . .
- one of the keys in conjunct \(m\).

Each phrase above could be further refined. For example, "one of the keys in conjunct 1" means:

\[
\text{key 1 in conjunct 1}
\]
\[
\text{OR key 2 in conjunct 1}
\]
\[
\text{OR } \ldots
\]
\[
\text{OR key } x \text{ in conjunct 1}.
\]

Thus, the feasible sets of keys selected for retrieval must include:

\[
(K_{11} \text{ OR } K_{12} \text{ OR } \ldots \text{ OR } K_{1x})
\]
\[
\text{AND } (K_{21} \text{ OR } K_{22} \text{ OR } \ldots \text{ OR } K_{2y})
\]
\[
\text{AND } \ldots
\]
\[
\text{AND } (K_{m1} \text{ OR } K_{m2} \text{ OR } \ldots \text{ OR } K_{mz}).
\]

Note that this is just the dual of the original query, assuming no negations. But it is in conjunctive form, CF, which is less intuitive. This CF dual can be transformed to its equivalent DF dual for greater clarity. This form results in a series of conjuncts, each of which implies a corresponding feasible set of keys for retrieval. These two notations, the logical conjunct of keys and the set of keys, will be treated as equivalent in subsequent
discussion. Brzozowski and Yoel (1976) discuss this in greater
detail. Then following from the above, any feasible solution must
include:

\[(K_{11} \text{ AND } K_{21} \text{ AND } \ldots \text{ AND } K_{m1})
\text{ OR } (K_{12} \text{ AND } K_{21} \text{ AND } \ldots \text{ AND } K_{m1})
\text{ OR } \ldots
\text{ OR } (K_{li} \text{ AND } K_{2j} \text{ AND } \ldots \text{ AND } K_{mh})
\text{ OR } \ldots
\text{ OR } (K_{lx} \text{ AND } K_{sy} \text{ AND } \ldots \text{ AND } K_{mz})\]

for any \(i = 1..x, j = 1..y, \ldots, h = 1..z\).

The conjuncts are, in effect, all possible sets which can be
constructed by taking a single key from each conjunct of the original
query. However, the actual list of minimal feasible sets is likely
to be substantially shorter than this both in terms of number and
length of conjuncts. This is because any duplicate sets and any sets
which are covered by other simpler sets are either not generated or
are eliminated during the transformation process. This is
demonstrated in the example below.

Just as in the query, the above logical expression is satisfied
if any one conjunct is satisfied. Thus any conjunct is a feasible
solution. If all keys in any conjunct in the solution are used for
retrieval, the query is covered, i.e., all records satisfying the
query will be retrieved. As a concrete example, the query:

\[(K_3 \text{ AND } K_8) \text{ OR } (K_4 \text{ AND } K_8) \text{ OR } (K_6 \text{ AND } K_8) \text{ OR } (K_6 \text{ AND } K_9)\]

would result in the following minimal feasible solutions:

\[(K_3 \text{ AND } K_4 \text{ AND } K_6) \text{ OR } (K_6 \text{ AND } K_8) \text{ OR } (K_8 \text{ AND } K_9)\].

Notice that many non-simple feasible solutions are not included. For
example:
(K3 AND K4 AND K6 AND K8 AND K9) is not even produced since there are only four conjuncts in the query itself; it is also covered by all of the minimal feasible solution conjuncts.

(K3 AND K4 AND K6 AND K6) is reduced to (K3 AND K4 AND K6) in the CF-DF transformation because of duplicate keys.

(K3 AND K4 AND K8 AND K9) is eliminated in transforming to DF because it is covered by (K8 AND K9), a minimal feasible solution conjunct.

**Stage Two**

The first stage thus results in three feasible solutions for the above example, one of which is optimal. Stage two compares these three possibilities on the optimizing criteria. Since the objective of the overall process is to retrieve the minimum number of records, the three possible solutions are compared on the total number of records each would retrieve. The most intuitive approach would be to use the straight sums. An alternative approach involves taking into account the probability of retrieving duplicate records within any solution set.

Using the straight sums method on the example above, let N3=30, N4=40, N6=60, N8=80, N9=90. If solution one is selected, (K3 AND K4 AND K6), then 30+40+60=130 records would be retrieved. Solution two, (K6 AND K8), would result in 60+80=140 records being retrieved. Solution three, (K8 AND K9), would result in 80+90=170 records being retrieved. By this criterion, the optimal solution would be solution one.

Using the probability of overlap criterion on the example above, 130, 140 and 170 would only be upper bounds on the numbers of records.
retrieved for solutions one, two, and three. Chances are that all of these 130 or 140 or 170 records would not need to be retrieved. For example, some of the K3 records are probably also K4 records and/or K6 records, and so on. Thus solution 1 would result in the retrieval of 130 records minus the K3-K4 overlap, the K3-K6 overlap and the K4-K6 overlap, plus the K3-K4-K6 overlap. Since the overlap cannot be determined from the directory, probabilities are used. For example, assuming K3 and K4 are of different keyword types, e.g., age, occupation, the probability of overlap for K3 and K4 is \((N3/N) \times (N4/N)\). If the file had 1000 records total, then the probability of overlap is \((30/1000) \times (40/1000) = 0.0012\) or 0.12%. Thus 0.12% of the 1000 records, or 1.2 records, are probably both K3 and K4 records. Similarly, the K3-K6 overlap is probably 1.8 records, for K4-K6, 2.4 records, and for K3-K4-K6, 0.072 records. Thus the solution \((K3 \text{ AND } K4 \text{ AND } K6)\) would probably need to retrieve \((30+40+60)-(1.2+1.8+2.4)+.072=124.672\) or 125 records. In comparison, solution two, \((K6 \text{ AND } K8)\), would drop from 140 to 136 retrievals and solution three, \((K8 \text{ AND } K9)\), would drop from 170 to 163.

In the example above, the three solutions have the same relative ranking with both the straight-sum and the probability-of-overlap comparison criteria. However, the probability criterion becomes more accurate in the following situations. One solution has several more keys than another, but their straight sums are quite similar. Several keys in a solution are of the same keyword type and hence there is guaranteed to be no overlap. One of the solutions involves a key-range, thus the keys in the expanded list are all of the same
keyword type. The number of records specified by the keys are sizable proportions of the total number of records in the file.

Formal Specification and Proof

Theorem

Let $Q$ be the DF query to the file $F$, $R$ be a record in file $F$, $Q^D$ be the dual of query $Q$, $Q^D'$ be the minimum DF of $Q^D$, $(K_x \land K_y \land \ldots \land K_z)$ be the minimum-sum term in $Q^D'$, i.e., the optimal solution, and $[K_x, K_y, \ldots, K_z]$ be the set of keys used for retrieval.

Then, 1) for all records $R$ satisfying $Q$, $R$ must contain at least one of $[K_x, K_y, \ldots, K_z]$, i.e., any record satisfying the query must be retrieved; and 2) the number of records retrieved is minimum.

Definitions

Let $K$ be a keyword in $Q$, $Q^D$, and/or $Q^D'$, $C$ be a conjunct in $Q$, $C^D$ be the dual of $C$, $N_i$ be the number of records in $F$ containing key $K_i$, $m$ be the number of conjuncts in $Q$, $m_m$ be the number of keys in conjunct $m$ in $Q$, $^\land$ and $v$ be two binary relations, AND and OR.

"SUBSCRIPTS" are one-two numbers and/or lower-case letters.
following K, C, CD or N. Where three "subscripts" appear, the last one is actually a "subscript" to the previous "subscript", e.g., for 
K\text{mnm}, the last m specifies which n. When two or three 
"subscripts" are appended to a K, they refer to K's original conjunct 
number in Q and its position number within that conjunct, respectively, e.g., K12 or K\text{nm} or K\text{nm}.  

A set S is said to CORRESPOND to a term T, i.e., a conjunct in 
QD', if for all Ki in T, Ki is in S. 

The number of records RETRIEVED for a term (K1 \wedge K2 \wedge \ldots \wedge Kp) 
where Ki \neq Kj, 1 \leq i \leq p, 1 \leq j \leq p, is defined as SUM (N1, N2, \ldots Np). 

A logical expression is said to COVER another logical expression 
if the first is true whenever the second is true. The first may be 
true or false whenever the second is false. Thus K1 covers (K1 \wedge 
K2) which in turn covers (K1 \wedge K2 \wedge \ldots \wedge Kn). 

An expression is said to be simplified if no keys within a 
conjunct are duplicates and any conjunct covered by another conjunct 
in the expression is eliminated. As a reminder of this, a conjunct 
is written as (K1 \wedge \ldots \wedge Kn)\ast.

**Strategy. Part One**

Since R satisfies Q, R contains all the keys in at least one 
conjunct of Q, say Ka, Kb, \ldots Kc. Then R will be retrieved if it 
contains at least one of the retrieved keys, Kx, Ky, \ldots Kz. That 
is, R will be retrieved if any key in the set \{Kx, Ky, \ldots Kz\} 
covers the conjunct (Ka \wedge Kb \wedge \ldots \wedge Kc). Thus, to demonstrate that 
R will be retrieved, it must be shown that:
[Ka, Kb, ... Kc] INTERSECT [Kx, Ky, ... Kz] NOT EMPTY.

Proof, Part One

Let \( Q = C_1 \lor C_2 \lor ... \lor C_i \lor ... \lor C_m \)

where \( C_i = (K_{i1} \lor K_{i2} \lor ... \lor K_{in_i}) \) for all \( 1 \leq i \leq m \),
\( K_{ij} \neq K_{st} \) for \( i=s \), and \( K_{ij} \neq K_{st} \) for \( i \neq s \),
\( 1 \leq s \leq m, \ 1 \leq j \leq n_i, \ 1 \leq t \leq n_s \).

Let any \( C_i \), say \( C_1 \), be the conjunct \( (K_a \land K_b \land ... \land K_c) \) which \( R \)
satisfies, giving the query in the form:

\( Q = (K_a \land K_b \land ... \land K_c) \lor C_2 \lor ... \lor C_m \).

The dual of the query is thus:

\( Q_D = (K_a \lor K_b \lor ... \lor K_c) \land C_2 \land ... \land C_m \).

This is in CF. Any CF expression may be transformed to an equivalent expression in DF by repeatedly applying the Boolean properties.

Using the distributive property:

\( Q_D = [K_a \land (C_2 \land ... \land C_m)] \land [K_b \land (C_2 \land ... \land C_m)] \land ... \land [K_c \land (C_2 \land ... \land C_m)]. \)

Consider only the CD portion of \( Q_D \) for the moment:

\( (C_2 \land ... \land C_m) = (K_{21} \lor K_{22} \lor ... \lor K_{2n_2}) \land ... \land (K_{m1} \lor K_{m2} \lor ... \lor K_{mn_m}). \)

If the distributive property were repeatedly applied to this portion of \( Q_D \), it would result in:

\( (C_2 \land ... \land C_i \land ... \land C_m) = \lor (K_{2f} \land ... \land K_{ig} \land ... \land K_{mh}) \)

for \( f = 1..n_2, \ g = 1..n_i, \ h = 1..n_m \),
where "\( \lor \)" is the "OR-ing" of terms, varying \( f, g \) and \( h \).

This would leave \( (n_2 \times n_3 \times ... \times n_m) \) terms. However, because there may be duplicate keys throughout the conjuncts, applying the
absorption, idempotent and commutative properties along with the
distributive property substantially reduces the number of terms. For
example,

\[(K_i \lor K_j) \land (K_i \lor K_k) = (K_i \land K_i) \lor (K_i \land K_k)\]
\[\lor (K_j \land K_i) \lor (K_j \land K_k)\]
\[= (K_i) \lor (K_j \land K_k).\]

Thus the simplified expression is:

\[(CD_2 \land \ldots \land CD_m) = (K_1 \land \ldots \land K_{11})\]
\[\lor \ldots \lor (K_2n_2 \land \ldots \land K_{2n_2})\]

where no keys within a conjunct are duplicates,
keys from different conjuncts may be duplicates and
conjuncts covered by other conjuncts are eliminated.

Thus, the "\&" on the conjunct is a reminder that a particular
conjunct may not actually exist if it is covered, or may not
explicitly contain \(m_{i-1}\) keys \((K_{2i}, K_{3i}, \ldots, K_{m_i})\) due to
simplification.

Substituting this in the query dual above,

\[Q_D = (K_a \land [(K_{21} \land \ldots \land K_{11})\]
\[\lor \ldots \lor (K_2n_2 \land \ldots \land K_{2n_2})]\]
\[\lor \ldots \lor (K_c \land [(K_{21} \land \ldots \land K_{11})\]
\[\lor \ldots \lor (K_2n_2 \land \ldots \land K_{2n_2})]\]]

Again applying the distributive and "simplifying" properties,

\[Q_D = (K_a \land K_{21} \land \ldots \land K_{11})\]
\[\lor \ldots \lor (K_a \land K_{2n_2} \land \ldots \land K_{2n_2})\]
\[\lor \ldots \lor (K_c \land K_{21} \land \ldots \land K_{11})\]
\[\lor \ldots \lor (K_c \land K_{2n_2} \land \ldots \land K_{2n_2})\]

Thus,

\[Q_D = Q_D',\]

The conjunctive form of the dual of the query has been
transformed into the minimum disjunctive form. All terms are of the
form: \((K_r \land K_{2f} \land \ldots \land K_{mh})\), where \(r\) is an element of \([a, b, \ldots,\]
c], f is an element of [1, 2, ... n2], h is an element of [1, 2, ... nm]. Consequently, Kr is an element of [Ka, Kb, ... Kc]. Thus, for all terms in \( QD' \),

\[ [Kr, K2f, \ldots Kmh]^* \text{ INTERSECT } [Ka, Kb, \ldots Kc] \text{ NOT EMPTY.} \]

That is, for any term \( (Kr \wedge K2f \wedge \ldots \wedge Kmh)^* \) in the DF of the dual of the query, the corresponding set of keys to be retrieved \( [Kr, K2f, \ldots Kmh]^* \) contains at least one key from the set \( [Ka, Kb, \ldots Kc] \) contained in \( R \) from a conjunct of \( Q \). Thus, retrieval of any set of keys corresponding to a term in \( QD' \) will necessarily retrieve \( R \). Every term in \( QD' \) contains a \( K \) which covers \( C=(Ka \wedge Kb \wedge \ldots \wedge Kc) \). This is true where \( C \) is \( C1, C2, \ldots \ Cm \). Thus, every corresponding set contains some \( K \) which covers \( C \), for all \( C \) in \( Q \).

Then since \( (Kx \wedge Ky \wedge \ldots \wedge Kz) \) is defined as a term in \( QD' \), actually, the minimum-sum term, it follows that:

\[ [Kx, Ky, \ldots Kz] \text{ INTERSECT } [Ka, Kb, \ldots Kc] \text{ NOT EMPTY.} \]

**Strategy, Part Two**

It must be proven that \( [Kx, Ky, \ldots Kz] \) leads to retrieval of the minimum number of records over all possible sets from \( Q \) which satisfy part one of the theorem. Only the straight sums criterion is considered here. The proof itself looks at two cases.

Any set of keys from \( Q \) falls into one of three categories:

1) it leads to the retrieval of \( R \) and corresponds to a term in \( QD' \);

2) it leads to the retrieval of \( R \) but does not correspond to a term in \( QD' \);

or 3) it does not lead to the retrieval of \( R \).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Case one is easily demonstrated by definition. Case two needs to demonstrate that the SUM of a case two set is greater than some SUM of a case one set, and consequently is greater than the SUM of the minimum-sum term in QD'. Case three need not be considered in the proof since such sets do not satisfy part one of the theorem.

Proof, Part Two

Regarding case one, note that \((Kx \times Ky \times ... \times Kz)\) is defined as the minimum-sum term in QD'. Then, for all terms \((Kil \times Ki2 \times ... \times Kini)\) in QD',

\[
\text{SUM (Nx, Ny, ... Nz)} \leq \text{SUM (Nil, Ni2, ... Nini)}
\]

for \(1 \leq i \leq p\), where \(p\) is the number of terms in QD'.

Thus, by definition the number of records retrieved from the set \([Kx, Ky, ... Kz]\) is minimum over all sets corresponding to terms in QD'.

Consider case two. Part one of the proof demonstrated that any set, SR*, which corresponds to a term in QD' contains some \(K\) which covers \(C\), for all \(C\) in \(Q\). The same must be true for a case two set, S2, since it too must retrieve R. But without the simplification procedure, the terms in QD' and their corresponding sets, [SR], represent all possible combinations of \(K\)'s from \(Q\) such that each term contains one \(K\) from each \(C\) in \(Q\), i.e., \(m\) \(K\)'s. Thus for some S2, SR and SR*, either:

\[
S2 = SR \quad \text{such that } SR \neq \text{any } SR*,
\]

or \(S2 = SR* \cup [Kl, ... Ki, ... Km]\)

for some number of \(Ki\) in \(Q\), \(1 \leq i \leq m\).

Clearly S2 could not have fewer \(K\)'s that this since it must retrieve
any R that satisfies Q. But [SR] represents all possible combinations of K in Q which do this.

In the first instance, S2 corresponds to a conjunct in the unsimplified QD'. S2 is not simplified, either because it has duplicate keys, or because some other conjunct in QD' simplified covers it. To demonstrate both, let S2 = [K1, ..., Ki, Ki, ..., Km] and SR* = [K1, ..., Ki]. Clearly,

\[ \text{SUM} (N1, ..., Ni, Ni, ..., Nm) > \text{SUM} (N1, ..., Ni) \geq \text{SUM} (Nx, Ny, ..., Nz), \]

the minimum-sum term in QD'.

In the second instance, S2 has all the K's of some set SR*, from the simplified QD', plus some number of additional K's. To demonstrate, let S2 = [K1, ..., Ki, Kj] and SR* = [K1, ..., Ki]. Clearly,

\[ \text{SUM} (N1, ..., Ni, Nj) > \text{SUM} (N1, ..., Ni) \geq \text{SUM} (Nx, Ny, ..., Nz), \]

the minimum-sum term in QD'.

In conclusion, part two of the theorem has been demonstrated. Any term in QD' leads to the retrieval of more records than the minimum-sum term in QD'. And any set of K from Q which satisfies part one of the theorem but does not correspond to a term in QD' will retrieve more records than some term in QD' and thus more records than the minimum-sum term in QD'.

An Example

The following example demonstrates the Dual-to-minDF optimization method:
\[ Q = (K_3 \land K_8) \lor (K_4 \land K_8) \lor (K_6 \land K_8) \lor (K_6 \land K_9) \]
\[ Q_D = (K_3 \lor K_8) \land (K_4 \lor K_8) \land (K_6 \lor K_8) \land (K_6 \lor K_9) \]

QD in CF must be transformed to QD', the minimum DF of QD, algebraically by applying the distributive, commutative, associative, idempotent and absorption properties.

\[ Q_D = (K_3 \lor K_8) \land (K_4 \lor K_8) \land (K_6 \lor K_8) \land (K_6 \lor K_9) \]
\[ = (K_8 \lor (K_3 \land K_4 \land K_6)) \land (K_6 \lor K_9) \]
\[ = (K_8 \land K_6) \lor (K_8 \land K_9) \lor (K_3 \land K_4 \land K_6 \land K_6) \]
\[ \lor (K_3 \land K_4 \land K_6 \land K_9) \]

Note that K_6 appears twice in the third conjunct and hence can be simplified, by applying the idempotent property. Also, the fourth conjunct can be covered by the third conjunct, by applying the absorption property.

\[ Q_D' = (K_8 \land K_6) \lor (K_8 \land K_9) \lor (K_3 \land K_4 \land K_6) \]

These three conjuncts correspond to the three minimal feasible retrieval sets:

\[ [K_8, K_6], \quad [K_8, K_9], \quad [K_3, K_4, K_6] \]

Each of the three sets covers all of the four conjuncts in the original query. To select the optimal set from among these, consider the straight sums criterion as a comparison measure. The N's for each key can be obtained from the directory. Assume these to be:

\[ N_3 = 30 \quad N_4 = 40 \quad N_6 = 60 \quad N_8 = 80 \quad N_9 = 90 \]

The first solution would lead to the retrieval of 80 + 60 = 140 records. The second would yield 80 + 90 = 170 records. And the third would lead to 30 + 40 + 60 = 130 record retrievals. The optimal solution is thus [K_3, K_4, K_6].
CHAPTER IV

OTHER QUERY-PROCESSING OPTIMIZATION STRATEGIES

Two other strategies have been suggested for optimizing query-processing: the prime key word and the minimum/maximum techniques. Both are based on the greedy method for problem solving discussed in the subsequent chapter. These two strategies both lead to an optimal solution under certain conditions, but neither guarantees retrieval of the minimum number of records in all cases. The prime key word, PKW, technique is a simple, straightforward procedure as compared with either the min/max or the proposed Dual-to-minDF method. Though more cumbersome, the min/max generally leads to a better solution than the PKW method. Both provide optimal solutions if there are no duplicate keys. However, both always provide at least feasible solutions. This may be acceptable for some applications. It may also be preferable in some applications where queries are extremely long and complex, but a quick, relatively good solution is sufficient, e.g., if there are relatively few records for any key. However, only the DUAL-to-minDF method guarantees that the optimal solution will be found.

This chapter describes the two alternative techniques, provides examples to demonstrate how they work and discusses particular types of situations where they do and don’t provide optimal solutions. The examples used below are fairly simple query-types involving
relatively few records. Such queries are used so as to demonstrate the technique and basic principles underlying their deficiencies. The simple examples do not clearly convey the potential magnitude of the problem. To retrieve a few extra records is perhaps of little consequence. However, most files have many, many records for each key. Thus, a poor choice of retrieval keys may result in hundreds more records being retrieved than is necessary.

Prime Key Word Technique

The Method

This technique builds on the idea used for processing simple queries in conjunct form. Consider an example, \((K1 \wedge K2)\). Either \(K1\) or \(K2\) would cover this query. That is, if a record satisfies the query, it will necessarily satisfy either key taken singly. Thus the key with the minimum number of records would be selected for retrieval. This is called the prime key word of the conjunct.

Any complex query can be transformed into disjunctive form. Then, a loop is performed which selects the best PKW, i.e., smallest \(N\), out of all the uncovered conjuncts. All conjuncts covered by this selected key are eliminated from future consideration. The procedure is repeated until all conjuncts are covered.

As an example, consider the following query:

\[
Q: (K3 \wedge K8) \vee (K4 \wedge K8) \vee (K6 \wedge K9) \vee (K3 \wedge K4).
\]

For the examples given in this chapter, for the sake of simplicity, it will be assumed that the number of records for each key is the
same as the key number. The PKW's are K3, K4, K6, K3. Thus, K3 would be selected first, covering conjuncts 1 and 4, K4 would then be selected, then K6. The set [K3, K4, K6] would retrieve 13 records. In this example, the PKW technique does yield the optimal solution.

Problems

However, the technique does not work optimally if the above example were altered slightly as shown below. K8 in the third conjunct is the only change.

Q: (K3 ^ K8) v (K4 ^ K8) v (K6 ^ K8) v (K3 ^ K4).

Here the set of prime keys to be retrieved would be the same as above. However, [K3, K8] is the optimal solution with 11 records retrieved. This is because K8 appears in so many different conjuncts. In this case, it happens that K8's appearance in conjunct 1 is irrelevant. However, in order to cover the second and third conjuncts, K8 is more efficient than K4 and K6 taken together.

The PKW technique is liable to fail because it only considers local minimums. It makes a greedy decision at each stage, yet this is inappropriate since each decision is not really independent of future decisions. The method clearly does always work in cases where there are no duplicate keys. In such a case the PKW from each conjunct could be selected with no iterative procedure really being necessary. However, when there is a key which appears in two or more different conjuncts, the accuracy of the PKW method is dependent upon the relative cost, i.e., the N's, of the other keys in those conjuncts. Consider four similar simple examples and their
solutions, two of which yield optimal solutions, two of which don't.

Q1: \((K1 \land K4) \lor (K2 \land K4)\) \(\implies [K1, K2]\) optimal
Q2: \((K3 \land K4) \lor (K2 \land K4)\) \(\implies [K2, K3]\) not optimal
Q3: \((K3 \land K4) \lor (K5 \land K4)\) \(\implies [K3, K4]\) not optimal
Q4: \((K6 \land K4) \lor (K5 \land K4)\) \(\implies [K4]\) optimal

In all four cases, the set \([K4]\) would be a feasible solution. In two of these it is optimal. Whether it is optimal or not is clearly dependent upon the \(N's\) for the other keys, taken jointly, not independently. For Q1 and Q2, the alternative keys all have smaller \(N's\) than \(K4\) does. However, taken together

\[
\text{SUM (N1, N2)} < \text{SUM (N4)} \quad 3 < 4
\]

\[
\text{but SUM (N2, N3)} \text{ not } < \text{SUM (N4)} \quad 5 \text{ not } < 4.
\]

Q2 demonstrates the problem of taking a local approach as opposed to a global approach to query-processing optimization.

The Q3 solution exhibits a similar problem. \(K3\) is selected first as the best prime key. Conjunct two still needs to be covered, so \(K4\) is then selected. However, hindsight shows that \(K4\) alone would've covered both conjuncts. \(K3\) is unnecessary.

The solutions for Q2 and Q3 are cases of the foothill and related cliff problems in the classic hill climb search analogy. The strategy dictates heading towards the goal using only the immediate altitude improvement as the deciding factor as to which direction to proceed. For Q2, going up, \(K2\), and then down the other side, \(K3\), of the foothill proves to be more costly than going around it, \(K4\). For Q3, going up the foothill, \(K3\), only to find a cliff forcing a return proved to be extraneous to having to take the path around the foothill anyway, \(K4\).

Q4's solution was the optimal one. \(K4\) was the PKW in both
conjuncts. K4 was an especially good solution key because it covered two conjuncts for the price of one key. This idea is exploited in the query-processing optimization strategy described in the next section.

The possibilities and problems demonstrated in Q1-Q4 are further exacerbated when queries become more complex, i.e., more conjuncts, more keys, more duplicate keys, more extensive overlapping of coverage. Clearly a strategy which takes more information into account on each decision step is necessary. The next alternative method considers how much "coverage" there is per key cost.

**Minimum/Maximum Technique**

**The Method**

This technique expands upon the prime key word method. The optimization criterion used above was to minimize the number of records in the file which contained a particular key. An iterative procedure was used, each time selecting the MIN (Ni) over all Ki in yet uncovered conjuncts. But, as was demonstrated above, another relevant factor was the frequency with which a particular key appeared. A key which appeared in several conjuncts would be a better choice than one which appeared in only a single conjunct, given similar N's. Several conjuncts would thus be covered at the expense of retrieving records with only the one key. However, the importance of minimizing Ni can not be discarded.

This technique incorporates both ideas. The procedure is
similar to the PKW technique except for the optimization criterion. Here, each selection is based on minimizing $N_i$ while maximizing $F_i$, i.e., the frequency of $K_i$. In this method, then, a loop is performed which selects the best $K_i$, i.e., \( \min (N_i/F_i) \), out of all the uncovered conjuncts. Then all $K$'s in conjuncts covered by this selected key are eliminated from future consideration. Note, however, that $F_i$ must be recomputed in each loop, where $F_i$ is the frequency of $K_i$ over all remaining uncovered conjuncts. The procedure is repeated until all conjuncts are covered.

As an example, consider the following query:

\[Q: (K_7 \land K_8) \lor (K_4 \land K_8) \lor (K_6 \land \neg K_8) \lor (K_6 \land K_9).\]

This would yield the following figures on the initial loop:

\[
\begin{align*}
K_4 & : \quad N_4 / F_4 = 4 / 1 = 4 \\
K_6 & : \quad N_6 / F_6 = 6 / 2 = 3 \\
K_7 & : \quad N_7 / F_7 = 7 / 1 = 7 \\
K_8 & : \quad N_8 / F_8 = 8 / 3 = 2.7 \quad \text{**minimum**} \\
K_9 & : \quad N_9 / F_9 = 9 / 1 = 9
\end{align*}
\]

$K_8$ would thus be selected on the first iteration. This covers the first three conjuncts. Thus only $(K_6 \land K_9)$ need be considered on the next iteration.

\[
\begin{align*}
K_6 & : \quad N_6 / F_6 = 6 / 1 = 6 \quad \text{**minimum**} \\
K_9 & : \quad N_9 / F_9 = 9 / 1 = 9
\end{align*}
\]

$K_6$ is then added to the selected set. This covers the remaining conjunct and hence the procedure terminates. \([K_6, K_8]\) leads to the retrieval of 14 records. This set is the optimal solution.

Problems

However, with just a minor change in the query above, this
technique will not give the optimal solution. Consider the following query. K3 in the first conjunct is the only change.

\[ Q: (K3 \land K8) \lor (K4 \land K8) \lor (K6 \land K8) \lor (K6 \land K9). \]

On the first iteration the following would be used:

- \( K3 : \frac{N3}{F3} = \frac{3}{1} = 3 \)
- \( K4 : \frac{N4}{F4} = \frac{4}{1} = 4 \)
- \( K6 : \frac{N6}{F6} = \frac{6}{2} = 3 \)
- \( K8 : \frac{N8}{F8} = \frac{8}{3} = 2.7 \quad \text{**minimum**} \)
- \( K9 : \frac{N9}{F9} = \frac{9}{1} = 9 \)

On the second iteration the figures to be used would be:

- \( K6 : \frac{N6}{F6} = \frac{6}{1} = 6 \quad \text{**minimum**} \)
- \( K9 : \frac{N9}{F9} = \frac{9}{1} = 9 \)

The same set of keys \([K6, K8]\) is selected leading to the retrieval of 14 records. However, \([K3, K4, K6]\) would cover the query while retrieving only 13 records. Why does this happen?

Note that K8 is initially a very good choice. Three conjuncts are covered at the expense of retrieving only 8 records. The alternative set of keys needed to cover these first three conjuncts would be \([K3, K4, K6]\), i.e., 13 records. K8 is certainly better than this. However, this is a local view of the problem. In the final overall view, the second key selected, K6, also covers conjunct three. Thus, K8's value, on hindsight, is less than initially thought. Since K6 must be part of the retrieval set anyway due to covering conjunct four, K8's value really diminishes to its coverage of only the first two conjuncts. Thus \([K8]\) should be compared with \([K3, K4]\), and hence does not remain the best choice.

The min/max technique fails on some types of queries for some of the same reasons that the PKW technique does. It takes a local view
as opposed to a global one. By using a greedy selection criterion at each step, it does not consider future key selections. Hindsight is needed. There is also a question of whether Ni/Fi is appropriately weighted.

Several simple examples demonstrate situations where a feasible, but not optimal, solution is found.

\[
\begin{align*}
Q_1: & \quad (K_3 \land K_4) \lor (K_4 \land K_5) \lor (K_5 \land K_6) \\
Q_2: & \quad (K_1 \land K_4) \lor (K_4 \land K_5) \\
Q_3: & \quad (K_1 \land K_5) \lor (K_2 \land K_5) \lor (K_3 \land K_5)
\end{align*}
\]

Applying the method to \(Q_1\) leads to \(K_4\) as the first key selected, covering conjuncts one and two, then \(K_5\), covering conjunct three. \([K_4, K_5]\) leads to 9 records being retrieved, but the optimal solution \([K_3, K_5]\) leads to 8 retrievals. The problem is the double coverage of conjunct two, similar to the problem in the previous query. With hindsight, \(K_4\) was a poorer choice since \(K_5\) usurped part of \(K_4\)'s coverage anyway.

\(Q_2\)'s solution, \([K_1, K_4]\) has the problem of having an extraneous key, \(K_1\). This occurs because of the relatively large difference in \(N_i\)'s between \(K_1\) and the other keys. \(K_4\) would have to have appeared in four conjuncts before its \(N_4/F_4\) equaled that of \(K_1\). This is rooted in the relatively equal importance given to \(N_i\) and \(F_i\). For example, halving the number of records a key has is treated as equivalent in result to doubling the number of conjuncts the key appears in at any stage.

The problem with \(Q_3\)'s solution has a similar basis. However, unlike \(Q_2\), it is not a matter of extraneous keys being selected in addition to the optimal ones. The solution under this method is \([K_1, \]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
K2, K3] leading to 6 record retrievals. The optimal solution is [K5] with 5 record retrievals. The problem here is that there is a relatively large difference between N's which is not large enough to make the small N keys good selections ultimately.

In all three cases the relative weighting of Ni vs. Fi is not appropriate. And it is implausible that any constant weighting factors would be applicable in all situations.

Discussion

In both techniques, one thing lacking in the comparison criteria is some measure of the alternatives if some key is selected. That is, what are the costs avoided if some decision is made? Also, a measure of the distance to the goal, i.e., coverage of all conjuncts, is needed. The major problem here, however, is that hindsight is needed. This is, in effect, what the Dual-to-minDF technique simulates. With such a technique, all alternatives are laid out for the actual decision process. One decision is made about a set of keys rather than a series of decisions about individual keys. This clearly is a more appropriate approach to decision-making when non-independent events are involved, where possible.

The tradeoff, however, in using either of the two less efficient techniques is the relative simplicity of the two. Both have polynomial time complexity and are fairly straight-forward to implement. And both heuristics arrive at acceptable solutions in many cases.
CHAPTER V

GENERAL PROBLEM-SOLVING ALGORITHMS

Each of the three optimization strategies described is an example of some more general problem-solving algorithm. Both the prime key word and the minimum/maximum strategies use what's called the greedy or hill-climbing method. The Dual-to-minDF is a type of dynamic programming approach. Both of these will be described and explored as to why one, the dynamic programming algorithm, always works on this problem, while the other, the greedy method, does not necessarily work.

Greedy Method

The greedy or hill-climbing method is a fairly straight-forward, intuitively appealing problem-solving approach. Given a set of inputs, the algorithm arrives at a subset of these which satisfies some constraint. There are many possible feasible solutions or subsets, i.e., all those which satisfy the constraint. But only one of these will be the optimal solution, the one which maximizes or minimizes some overall objective function.

At each stage the algorithm selects an input in a greedy way, maximizing or minimizing some optimization measure. This may or may not be the same as the overall objective function. In effect, it tries to climb the hill to the goal as quickly as possible. If the
selected input can be included with the partial solution set and the set still remains feasible, then this becomes the new partial solution set for the next stage. This continues until all inputs have been considered or until the goal has been reached. The following spells this out in more specific terms.

\[
\begin{align*}
I &= \text{set of inputs } [1..n] \\
S &= \text{solution set; } S \text{ a subset of } I \\
x &= \text{a selected element of } I \\
OPT &= \text{the optimization measure} \\
GOAL &= \text{a possible early ending point} \\
CONSTRAINT &= \text{the feasibility requirements}
\end{align*}
\]

**GREEDY** procedure \( (I, S) \);
\[ S = \text{EMPTY}; \]
index = 1;
repeat
\[
\begin{align*}
&\text{SELECT} \ (x, I); \\
&\quad \text{if FEASIBLE} \ (x, S) \\
&\quad \quad \text{then } S = S \cup x
\end{align*}
\]
until index = n or S satisfies GOAL.

**SELECT** procedure \( (x, I) \);
\[ x = 1; \]
for \( i = 2 \) to n
\[
\begin{align*}
&\quad \text{if OPT} \ (I(i)) \text{ better than OPT} \ (x) \\
&\quad \quad \text{then } x = I(i); \\
&\quad I = I \text{ without } x.
\end{align*}
\]

**FEASIBLE** Boolean function \( (x, S) \);
\[ \text{if } (S \cup x) \text{ satisfies constraint} \]
\[ \quad \text{then Feasible} = \text{True} \]
\[ \quad \text{else Feasible} = \text{False}. \]

(Horowitz and Sahni, 1978)

In order to fit the above algorithm, the query-processing problem can be re-stated using set notation. The general process and result are the same as the previous specification in terms of logical expressions.

Given: A collection \( Q \) of possibly overlapping subsets, \( C_1, \ldots, C_m \), of a finite set \( KW \) where element \( K_i \) has:

- weight \( N_i \) \( (i = 1 .. \mid KW \) \) and
- frequency \( F_i = \#C_j \) \( (j = 1 .. m) \) such that \( K_i \text{ INTERSECT } C_j \)
Problem: Find a subset $R$ of $KW$ such that:

1) $R \text{ INTERSECT } C_j$ for $j = 1..m$
   i.e., $R$ is a feasible solution;

and 2) for all $Ki$ in $R$, $\text{SUM}(Ni)$ is minimum over all $R'$ in $KW$
   i.e., $R$ is an optimal solution.

The algorithm definitions can be specified for the prime key word and
the minimum/maximum strategies as follows:

$I = KW :$ a set of inputs $[K_1,...,K_n]$ divided into possibly
   overlapping subsets $[C_1,...,C_m]$

$S = R :$ the solution/retrieval set; a subset of $KW$

$x = Ki :$ a selected element of $KW$

$OPT = \text{MIN} (Ni) \quad i = 1..n$ \hspace{1cm} [for PKW]

$= \text{MIN} (Ni \setminus Fi) \quad i = 1..n \hspace{1cm} [for \text{min/max}]$

$GOAL = R \text{ INTERSECT } C_j$ for $j = 1..m$

$\text{CONSTRAINT} = R \text{ INTERSECT } [Ki] = \text{empty}$
   and $(R \cup [Ki]) \text{ INTERSECT } C_j = Ki \quad (C_j \in \text{new } I)$
   i.e., the key covers some conjunct not yet covered.

Note that the algorithm does not specifically mention the
overall objective function. For query-processing optimization this
would be as stated in part two of the initial problem statement:
$\text{SUM} (Ni) = \text{MIN}$ for $Ki$ in $R$, i.e., the minimum number of record
retrievals. But if the algorithm does not incorporate such a
function, there is no guarantee that the solution will satisfy it.
This is precisely the problem with the algorithm and thus the two
strategies.

The subset which this algorithm finds will definitely be a
feasible solution. However, it may or may not be the optimal
solution. This depends on whether an appropriate selection
optimization measure can be found. If, at each and every stage, the
partial solution is part of the optimal solution, an optimal solution
will result. But this is not built into the algorithm.

The optimization measure is applied at each stage using only local optimum. But the objective function is the specification of the overall goal that one hopes the solution subset will satisfy. It is a global optimization measure applied after-the-fact.

The greedy method is thus a heuristic which is a relatively fast and easy algorithm to implement, usually giving a good, though not necessarily optimal, solution.

Hill-climbing methods . . . have a certain goal in mind, and they try to do whatever they can, whenever they can to get closer to that goal. This tends to make them a bit short-sighted. . . [They] may be useful if we are willing to settle for a fast approximate solution.

(Goodman and Hedetniemi, 1977:970).

Dynamic Programming Method

The dynamic programming method is like the greedy method in that it too is applicable to problems in which the solution results from a series of decision steps. However, this method does not actually solve the problem by making one decision at a time. Rather, the approach here is to consider all likely decision sequences. This is not, however, a complete enumeration of all possible sequences with each being checked against the objective function. The dynamic programming algorithm generates only a subset of these sequences: those sets which are likely to be the optimal solution. This is done in part by appealing to the principle of optimality. That is, at each stage, the partial solution must be part of a potential optimal solution.
A tree structure is useful in describing the efficiency of the algorithm. The problem to be solved can be seen as a tree, where all root-to-leaf paths represent sequences of decisions which are potential solutions, i.e., all feasible solutions. The algorithm itself dynamically generates a solution tree which may actually be considerably smaller than the problem tree since many feasible solution branches are, in effect, pruned. And some may be pruned at an early stage thus eliminating a number of potential root-to-leaf paths. Actually, the algorithm does not generate such branches at all. At each stage in the algorithm, prior to adding a branch to a node, the algorithm assures that the new partial path would still be part of a potentially optimal solution path.

Thus, the dynamic programming algorithm generates many decision sequences, i.e., all potential optimal solutions, unlike the greedy method which generates only one decision sequence which is feasible and perhaps optimal. This is because, for some types of problems, no optimal solution can be determined except at the final stage.

The notational specification of the algorithm varies widely from problem to problem. The Dual-to-minDF strategy described in a previous chapter is not fully a direct application of the dynamic programming algorithm in its use of recursion, look-ahead/back or true sequences or sets. And the method actually described uses an algebraic approach as opposed to set notation as used to demonstrate the greedy algorithm notation. The basic process and result, however, are similar. The actual algorithm will be presented in the subsequent time analysis chapter.
CHAPTER VI

THE ALGORITHM'S COMPLEXITY

The algorithm presented in chapter III for selecting the set of retrieval keys will arrive at an optimal solution for any instance of the problem. However, for any algorithm, one must also be concerned with its time bound and order of complexity. These measures suggest whether the algorithm is potentially a practical one for actual applications or whether it is even do-able on today's computers given data sets of any significant size. In cases where algorithms cannot be carried out in polynomial time, i.e., NP-complete problems, alternative algorithms may have to be considered which only approximate the desired solution.

This chapter discusses the complexity of the Dual-to-minDF algorithm, whether it can be done in polynomial time and its practical significance for query processing in actual applications.

Complexity of Algorithms

Several concepts are important in considering the complexity of an algorithm. Baase (1978) and Aho et al. (1978) provide a full discussion on this. A main concern in computer science is that of efficiency, both for theoretical and practical reasons. And generally the main resource of interest to conserve is time. Time is money! Algorithms which solve a problem quickly are better than
those which don't within certain space constraints. There are two main ways of comparing algorithms and measuring the time needed to carry them out: a priori analysis of the raw algorithm itself and a posteriori testing of the language-coded algorithm running actual data sets on a computer. The former will be considered here.

A priori analysis of algorithms measures time in units which are expressed as a function of the input data size. Algorithms can then be classified according to the category of their time function—e.g., linear, polynomial, exponential.

There is general agreement that problems which can only be solved in exponential or greater time are considered intractable. In effect, the problem cannot be solved in a "reasonable" amount of time. A problem which has a polynomial time algorithm is considered efficient since it is solvable in a reasonable amount of time. These two categories are referred to as NP-complete and P problems, respectively. Obviously there are cases where an exponential time algorithm for a problem takes less time than a polynomial time algorithm, but this is only for small N's. For example, \(2^n < n^4\) up until \(n=16\). However, a main concern for the problem solution time is with its growth rate expressed as a function of the input size. And for exponential time algorithms, this is explosive.

In measuring the time complexity of an algorithm, its worst case behavior is generally considered. If several smaller algorithms are combined to solve a problem, only the worst time complexity measure need be used to characterize the problem. Coefficients and lower order terms of the time measure don't affect the overall growth rate.
significantly. These may, however, be important in a comparison of two algorithms of the same order of complexity to enable the selection of the most efficient solution. The time category of an algorithm is expressed in terms of an order of complexity, e.g., $O(N)$, $O(N^{*2})$, $O(2^{*N})$.

Although time complexity is usually based on analysis of the worst case, the typical application generally yields much better actual performance. Hence, exponential time algorithms may be useful in actual practice in dealing with average cases, small data sets or special subsets of the general problem (Garey and Johnson, 1979).

"Most exponential time algorithms are merely variations on exhaustive search, whereas polynomial time algorithms generally are made possible only through the gain of some deeper insight into the structure of a problem." (Garey and Johnson, 1979:8). This is often the case with optimization problems. All feasible solutions must be explored in order to arrive at the optimal solution. In some cases, some pruning of part of the search space may be possible which will reduce the search time, but generally not the order of complexity. Such is the case here.

Optimization problems are concerned with selecting the "best" configuration or set of parameters to achieve some goal. This is discussed further by Papadimitriou and Steiglitz (1982). Some of these types of problems have been found to be solvable in polynomial time, most have not. One class of optimization problems has the property that the local optimum are part of the global optimal solution. These would be solvable in polynomial time using an
approach such as the greedy method, for example. Another class consists of problems which require the selection of the optimal solution from among a finite set of feasible solutions. These generally require an approach such as dynamic programming or branch and bound which elaborates all feasible solutions. These are often exponential time problems. It should be noted, however, that two problems with very similar specifications may not both fall into the same category of complexity. Many researchers are searching for polynomial time algorithms for problems currently defined as exponential. At the present, heuristics are being used to arrive at sub-optimal solutions or problems are being more narrowly defined so as to use a poly-time algorithm.

Analysis of Complexity

The Dual-to-minDF algorithm involves three steps:

1) take the dual of the query;
2) convert the query-dual to minimum disjunctive form;
3) apply the comparison function to each conjunct's set of keys in order to select the optimal set.

For easier analysis of the algorithm complexity, it will be assumed that the query is originally entered in disjunctive form, DF. Then, taking the dual of query Q will give a conjunctive form, CF expression for Q-dual. This assumption is not necessary for the strategy in general, however, and actually provides a "worst case" for analysis of this strategy. Since step two converts Q-dual to the minimum DF, fewer steps are involved if Q-dual is already partially converted as opposed to being in CF which needs maximum conversion.
For the following analysis, consider the disjunctive form query:

\[ Q = (K_{11} \land \ldots \land K_{lj} \land \ldots \land K_{ln_l}) \lor \ldots \lor (K_{1l} \land \ldots \land K_{ij} \land \ldots \land K_{ini}) \lor \ldots \lor (K_{ml} \land \ldots \land K_{mj} \land \ldots \land K_{nn_m}) \]

where \( m \) is the number of conjuncts in \( Q \), and each conjunct contains \( n_i \) keys, \( i=1..m \). Let \( N \) be the total number of keys in \( Q \), \( N = \sum (n_i) \) for \( i=1..m \), where any two or more keys may be duplicates. Let \( QD \) be the logical expression for the dual of the query. And let \( QD^\prime \) be the logical expressions for the minimized, disjunctive form of the dual of the original query.

Step one has a linear time complexity. To convert a logical expression to its dual, only the operators need to be changed, i.e., the AND's and the OR's. This necessitates one pass through the query, changing the \( N-1 \) operators. There will be \( m-1 \) OR's in \( Q \). Each conjunct will have \( n_i-1 \) AND's making a total of \( N-m \) AND's. Thus the total number of connectors is \((m-1) + (N-m) = N-1\). This gives an order of complexity for this step of only \( O(N) \).

Step two really performs two functions: it converts \( QD \) from CF into DF and it minimizes \( QD \). An algebraic solution is used. A Boolean function may be converted into DF by repeatedly applying the distributive, associative and commutative properties. Minimization is done by repeatedly applying the absorption and idempotent properties. How much time will it take to transform and minimize \( QD \) into \( QD^\prime \)?

First consider how many conjuncts there will be in the final \( QD^\prime \). In the worst case, assume that no minimization is possible.
Thus there will be $n_1 \times n_2 \times \ldots \times n_m = \prod (n_i)$ (for $i=1..m$) conjuncts in $QD'$. Let $C$ equal this product, i.e., the number of conjuncts in $QD'$. This represents all possible combinations of keys from $Q$ such that one key comes from each conjunct in $Q$ and no keys were duplicates. This can be expressed in a different way. Note that on the average $n_i = N/m$, i.e., the average number of keys per conjunct in $Q$. Thus, there will be approximately $(N/m)^m$ conjuncts in $QD'$ if no minimization is possible.

Next consider how many keys there will be in each conjunct in $QD'$. Again assume the worst case with no minimization possible during the transformation process. Each conjunct in $QD'$ will then have $m$ keys, one key from every conjunct in $Q$. Thus the total number of keys in $QD'$ is $mC$. Obviously there are many duplicates since the original $Q$ had $N$ distinct keys.

The number of moves in step two in order to "transform to DF" is thus the number of keys in $QD'$ which is $mC$. Each key $K_j$ was moved to $\text{MULT} (n_i) / n_j$ (for $i=1..m$) different terms in $QD'$. Thus considering the worst case, the order of complexity for this part of step two is $O(mC)$.

The second part of step two involves minimization of the disjunctive form of $QD$. Two types of simplification are possible. The idempotent property reduces conjuncts with duplicate keys by eliminating the duplicates, e.g., $[K_1 \text{ AND } K_2 \text{ AND } K_2]$ becomes $[K_1 \text{ AND } K_2]$. The absorption property eliminates conjuncts which are covered by other conjuncts, e.g., $[K_1 \text{ AND } K_2]$ covers $[K_1 \text{ AND } K_2 \text{ AND } K_3]$. Obviously the minimization procedures can be done in parallel with
the CF-to-DF conversion process described above. However, for ease of analysis, assume that these two will be done separately, i.e., that the minimization is done on the DF expression.

For the first minimization phase of eliminating duplicate keys, in general, all keys in a conjunct have to be compared with all other keys in the conjunct. In the worst case, given that there are \( m \) keys in each conjunct, there would be \( m! / 2(m-2)! = m(m-1) / 2 \) for a conjunct. However, the second minimization phase needs the keys in a conjunct to be sorted. Hence a simple sort algorithm can be used in which comparisons which find duplicate keys eliminates one key rather than re-writing it. Each conjunct could thus be sorted and reduced in \( m \times \log(m) \) time, where \( m \) is the number of keys in each conjunct. Thus the number of key comparisons for the entire disjunctive form \( Q_D \) would be: \( C \times m \times \log(m) \).

The second minimization phase involves eliminating entire conjuncts if they're found to be covered. Every conjunct in the DF \( Q_D \) must be compared with every other one to see if one contains all the keys contained in the other. If so, the one with extra keys or the duplicate is eliminated. This results in an upper bound of \( C(C-1)/2 \) conjunct comparisons. It should be noted that there will generally be fewer comparisons than this since once a conjunct is eliminated, it is not used in future comparisons. Thus the worst case complexity is \( O(C^2) \).

Step three is fairly straightforward. Each conjunct's cost must be calculated. Then the costs must be compared in order to arrive at the minimum cost. The cost of a conjunct is the sum of
the number of records in the file for each key in the conjunct.

As was shown above, in the worst case where no minimization is possible, \( QD' \) has \( C \) conjuncts with \( m \) keys per conjunct resulting in \( mC \) keys overall. Then, in actually carrying out the evaluation function, for each key in \( QD' \) there will be one directory look-up for getting the key cost. There will also be one addition per key in \( QD' \) in deriving the total costs for each conjunct in \( QD' \). Then, the minimum cost conjunct must be found. Thus there will be \( mC \) directory look-ups plus \( mC \) additions plus \( m-1 \) comparisons in arriving at the optimal set. This results in an order of complexity of \( O(mC) \).

Thus, in the worst case, the entire algorithm will take:

- **Step 1**: \( N - 1 \)
- **Step 2**: \( m \times C \times \log(m) \)
- **Step 3**: \( C \times (C - 1) / 2 \)

where \( m \) is the number of conjuncts in \( Q \),
\( N \) is the number of keys in \( Q \),
\( C \) is the product of the number of keys in each conjunct in \( Q \),
\( (n1 \times n2 \times \ldots \times nm) \approx (N / m)^m \).

Assuming that \( m \times \log(m) < (C - 1) / 2 \), then the most time-consuming step is part three of step two: \( C \times (C-1) / 2 \). This gives an order of complexity of \( O(C^2) \). However, \( C \) itself is exponential on \( m \). The algorithm thus has exponential complexity: \( O[(N/m)^{2m}] \). The time is affected by both the number of conjuncts and the number of keys in each conjunct. Obviously this represents an upper bound on the time it takes to find an optimal solution for a query. The average query has much better time.
Theoretical vs. Practical Significance

The above discussion of the time complexity of the algorithm proposed here demonstrates that it has an exponential order of complexity. One would thus conclude that the proposed method is not a reasonable solution for the problem of finding the optimal retrieval set. It is commonly accepted that an exponential time algorithm is just too "large" to be efficient enough.

However, such a finding may or may not be of great significance in considering the actual application of such an algorithm in a practical setting. Several issues are relevant.

First of all, the analysis is based on "worst case" assumptions.Obviously this provides an upper bound on the time. In effect, theoretically, there is at least one problem instance which would require this maximum amount of time. However, in actual applications, what are the data sets really like? In the case of query-processing, what are the commonest types of queries used on the file? What are the range and distribution of query formats usually found? Are they entered in disjunctive form? What proportion fall into or anywhere near the worst case category? These are questions that need to be addressed in viewing the algorithm from a practical vs. theoretical point of view. Consider parallel situations in other areas. For example, although quicksort doesn't have the best "worst case" behavior \(O(n^{**2})\), it does have very good "average case" behavior, both in a-priori analysis, \(O(n*log(n))\), as well as in empirical comparison testing against other sorts. Even some
well-known exponential complexity algorithms are useful in practice. "The simplex algorithm for linear programming ... has a record of running quickly in practice. Likewise, branch-and-bound algorithms for the knapsack problem have been so successful that many consider it to be a 'well-solved' problem." (Garey and Johnson, 1979:9).

A second point to consider is the size of the data in the actual application. For small data sets, an exponential order algorithm may actually be more efficient than a polynomial time algorithm. Again one must consider the types of queries. Will the queries be relatively short, involving few conjuncts? How many possible keys are there in the file? How many different keys are likely to appear in typical queries? Obviously the method is impractical for extremely large queries. However, this is not the case in many applications. This leads to a third issue.

What are the alternatives? What are the trade-offs? The goal of the algorithm here, and indeed any query processing optimization method, is to reduce retrieval time. The specific goal of the proposed Dual-to-minDF and the two greedy methods is to reduce the number of excess non-satisfying records which must be retrieved. What is the cost or retrieving X extra external records? And what is the cost of the extra internal processing time needed in order to come up with the optimal set of record addresses as opposed to arriving at merely a feasible set? How far off is the feasible solution? That is, how good are the alternative heuristic algorithms? These costs and trade-offs must be evaluated bearing in mind that the time for one external record retrieval is a magnitude greater than
the time for the execution of one machine instruction.

Consider a simple example. The query:

\[
Q = (\text{Grad-Student AND Math-Major}) \text{ OR } (\text{Grad-Student AND Engin-Major}) \text{ OR } (\text{Grad-Student AND CS-Major}) \text{ OR } (\text{Senior AND CS-Major})
\]

Assume that the N's for the file are as follows:

- Grad-Student: 400
- Senior: 800
- CS-Major: 300
- Engin-Major: 140
- Math-Major: 170

Using the proposed method, the keys to be retrieved would be:

\[(\text{CS-Major AND Engin-Major AND Math-Major})\]

giving 300 + 170 + 140 = 610 records to be retrieved. If instead the min/max technique had been used, then the retrieval keys would be:

\[(\text{CS-Major AND Grad-Student})\]

giving 300 + 400 = 700 records to be retrieved. Thus, 90 fewer records are retrieved with the Dual-to-minDF as compared with the min/max technique.

This assumes that there is no duplication of records among the retrieval keys. This may or may not be realistic. The second comparison function discussed in chapter III does take the probability of overlap into account. The magnitude of the difference is similar, however. Also, the N's assigned here were selected so as to demonstrate the point. Obviously cases can be constructed where the difference is less. The example, however, is not unrealistic.

What is the price paid for retrieving 90 fewer records? Is the exponential complexity unwieldy with four conjuncts of eight keys?
More importantly though, how does the algorithm fare on an "average case" set of input such as this query?

Step one requires seven operator changes. On step two, the minimization procedures will be performed intermixed with the transformation procedures, i.e., expand the query dual with the distributive property, then reduce QD with the idempotent and absorption properties. On the first pass through the query, 16 variable moves, 8 variable compares and 8 conjunct compares are done for the three properties above, respectively. The second pass requires 12 variable moves, 12 variable compares and 4 conjunct compares. QD is now in minimum disjunctive form. Step three includes 7 directory look-ups, 7 additions and 2 variable compares.

Certainly the overall processing time spent to find the optimal solution is significantly less than the time necessary for 90 external record retrievals. Whether this is a realistic representative example certainly depends on the application one has in mind. But it does demonstrate that if queries are relatively short, and there is some duplication of keys across conjuncts, and many keys have several hundred records, then the proposed Dual-to-min-DF is a useful algorithm. Further research needs to be done on specifying the threshold points for these parameters.
CHAPTER VII

CONCLUSIONS AND RECOMMENDATIONS

The main focus of this research is on developing a strategy for reducing the number of external record retrievals in order to minimize query processing time. A generalized file organization is assumed which presents a problem not found when inverted file directories are used, i.e., the retrieval of surplus records. This is because all record addressing information is not kept in the directory itself, but rather, part is kept in the file in the form of address pointers. The retrieval system must thus decide which keyword lists to follow based only on the information in the directory. Given that lists for all keys in a query need not be used for retrieval, what is the optimal set of keys?

This study proposes an algorithm which arrives at an optimal set of keys. It will definitely find the optimal solution whereas the two methods previously proposed in the literature merely arrive at feasible solutions which may or may not be optimal. The price, however, is that the proposed strategy has exponential time complexity. But its potential usefulness is very dependent upon the particular application.

The tradeoff of record retrievals saved vs. extra processing time spent is no doubt beneficial in applications with the following characteristics. Users generally enter relatively short Boolean
queries with some duplication of keys within the query. The file is relatively large, resulting in many records satisfying a particular key. This makes the optimal choice of retrieval keys more critical and causes the two alternative algorithms to be potentially less reliable. The size of the query results in fairly fast transformation and minimization times. The duplication of keys within the query also contributes to the potentially inefficient solutions of the two greedy methods.

Recommendations for Further Research

There are, obviously, a number of areas where further research might be done. A number of questions remain open. And other aspects of the problem offer potential for extension. Further empirical evaluation of the method might also be done to clarify the specific situations in which it demonstrates its practical usefulness. The following are some possible areas in which to extend investigation on the topic.

Complexity analysis is based on worst case behavior. How does this algorithm perform on the average? This might be done with empirical testing on various types of queries and files. For a more thorough understanding, several parameters should be varied such as query length in terms or number of conjuncts and/or number of keys, extent of duplication of keys within the query, file size, key address-list length for the file, number of keys in the file system. Through such testing it would be easier to pinpoint specific parameter boundaries within which the algorithm proved to be
efficient.

An extension of this would be to compare the behavior of the proposed algorithm against that of the two alternative heuristic techniques. It would be very useful to be able to specify under what conditions that the Dual-to-minDF algorithm was and wasn’t worth the extra processing time. Is there some threshold function which could be developed which could be used to judge the potential usefulness of the algorithm for a particular system with certain file and query characteristics? Further, within a particular application, are there measurable characteristics of the queries themselves which would allow the retrieval system to select the best of the three algorithms?

Another direction in which to extend this research is in looking at the strategy itself. An algebraic manipulation of the Boolean query is used here. Perhaps there might be an extension of the Quine-McCluskey minimization method useful on conjunctive form queries. The problem might also lead to a more efficient algorithm if it is treated using set language where the goal is to find the minimum cost subset which hits every "conjunct-subset" from the query-set of keys. Or perhaps a graph representation might suggest that the problem is similar to a covering or partition problem.

Additional attention should be given to minimizing block accesses rather than merely record accesses. What is the probability of consecutive records being within the same block? This is, to some extent, considered in the "probability of overlap" solution comparison of the proposed method. The issue perhaps could be
addressed more directly. Several authors mentioned in the literature review provide formulas for making such estimates.

A point to note here, however, is the different implications of the "same block" concept when applied to inverted files as opposed to a generalized file. To take advantage of the savings from "2+ records per 1-block access", the two plus records must be requested in sequence by the retrieval system. This is difficult to do when much of the addressing information is kept in the file itself.

This study assumed a generalized file organization. Might there be additional refinements or shortcuts possible on related file structures, e.g., cellular multilist or partially inverted files with list-lengths $\leq$ some constant? How might this strategy operate on a file where directory keys are really conjunctions of keys inverted together as one key?

An alternative comparison measure might be used to select the optimal solution from among the feasible solutions based on the idiosyncratic characteristics of particular applications. The method as presented here assumes all keys have equal probability of being requested. Also, any combination of keys is assumed to be equi-probable. This is not always a valid assumption. Consider a simple example of an airline reservation/inquiry system with the following attributes: First Name, Last Name, Address, Phone Number, Flight Number, and Flight Date. Assume for demonstration purposes that all of these are key fields. Certainly Last Name or Flight Number are more likely to be part of a user query than First Name or Phone Number. Obviously the latter two would not be set up as keys,
but the example clearly demonstrates the point. Also, First Name is more likely to appear in conjunction with Last Name than with Flight Number. Likewise Flight Date and Last Name might be likely to be requested together. Thus a weighting technique might be incorporated in the final comparison step of the algorithm, similar to the probability of overlap measure suggested.

In conclusion, this research contributes to the body of research focused on making retrieval systems more efficient. Studies have investigated many different parts of this process. The concern with reducing I/O time, as focused upon here, is certainly a significant aspect of this problem.
BIBLIOGRAPHY


Kountanis, Dionysios. "Notes from CS342, CS625, CS631, and CS632." Kalamazoo, Mi.: Western Michigan University, 1982-83. (Class Notes.)


