2

RADC-TR-90-404, Vol XII (of 18)
Final Technical Report
December 1990

COMPUTER ARCHITECTURE FOR VERY LARGE KNOWLEDGE BASES

Northeast Artificial Intelligence Consortium (NAIC)

P. Bruce Berra, Arif Ghafoor, Soon Myong Chung,
Nabil I. Hachem, Slawomir J. Marcinkowski,
Periklis A. Mitkas, Donghoon Shin

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-404, Volume XII (of 18) has been reviewed and is approved for publication.

APPROVED: Raymond A. Liuzzi

RAYMOND A. LIUZZI
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: BG Oaks

BILLY G. OAKS
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final Sep 84 - Dec 89

4. TITLE AND SUBTITLE
COMPUTER ARCHITECTURE FOR VERY LARGE KNOWLEDGE BASES

5. FUNDING NUMBERS
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

6. AUTHOR(S)
P. Bruce Berra, Arif Ghafoor, Soon Myong Chung,
Nabil I. Hachem, Slawomir J. Marcinkowski,
Periklis A. Mitkas, Donghoon Shin

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-404, Vol XII (of 18)

11. SUPPLEMENTARY NOTES
(See reverse)
RADC Project Engineer: Raymond A. Liuzzi/COES/(315) 330-3528
This effort was funded partially by the Laboratory Director's fund.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the development of architectures for very large knowledge bases, especially in light of real-time requests, parallelism, and the advent of optical computing.

14. SUBJECT TERMS Artificial Intelligence, Computer Architecture, Very large Knowledge Bases, Real Time Processing, Pattern Matching, Parallel Computing

15. NUMBER OF PAGES
338

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Block 5 (Cont'd)   Funding Numbers

PE - 62702F	PE - 61102F	PE - 61102F	PE - 33126F	PE - 61101F
PR - 5581	PR - 2304	PR - 2304	PR - 2155	PR - LDFP
TA - 27	TA - J5	TA - J5	TA - 02	TA - 27
WU - 23	WU - 01	WU - 15	WU - 10	WU - 01

Block 11 (Cont'd)

This effort was performed by the University of Syracuse, Office of Sponsored Programs.

Table of Contents

12.1 Executive Summary 1

12.2 Introduction 3

12.3 Backend Relational Algebra Machine Based on CCW Surrogate File 7

12.3.1 Concatenated Code Word Surrogate File 7

12.3.2 Structure of the Backend Relational Algebra Machine 9

12.3.3 Relational Algebra Operations in the Backend Database Machine 15

12.3.4 Performance Analysis of the Proposed Architecture 18

12.4 Management of General First Order Terms and Rules 25

12.4.1 General Terms Indexing via Surrogate Fiies 25

12.4.2 Design of an Associative Surrogate Processors for CCW-1 and CCW-2 30

12.5 Inverted Surrogate Files 32

12.5.1 Transformed Inverted Lists 32

12.5.2 The Dynamic Random-Sequential Access Method 38

12.5.3 Performance Evaluation of DRSAM File Structures 55

12.5.4 Inverted Surrogate Files with DRSAM 64

12.6 Optics in Very Large Knowledge Bases 82

12.6.1 Optical Data/Knowledge Base Machines 82

12.6.2 Relational Operations 83

12.6.3 Optical Comparisons 85

12.6.4 Spatial Light Modulators 86

12.6.5 An Optical System for Relational Operations 89

12.6.6 Alternatives for an Optical D/KBM 94

12.7 Implementing Knowledge Base Management Systems Based on Surrogate Files 99

12.8 Conclusion 100

References 102

Appendix 12-A A Relational Algebra Machine Based on Surrogate Files for Very Large Data/Knowledge Bases.

Appendix 12-B Surrogate File Approach to Managing First Order Terms in

Secondary Storage.

Appendix 12-C Computer Architectures for Logic-Oriented Data/Knowledge Bases.

Appendix 12-D Key-Sequential Access Methods for Very Large Files Derived from Linear Hashing.

Appendix 12-E An Optical System for Full Text Search.

Appendix 12-F An Optical Architecture for Performing Relational Data Base Operations.

Appendix 12-G Implementing Knowledge Base Management Systems Based on Surrogate Files.

12. Computer Architecture for Very Large Knowledge Bases

12.1. Executive Summary

The focus of our research is on the development of algorithmic, software and hardware solutions for the management of very large knowledge bases (VLKB) in a real time environment. We approach the problem from electronic and optical points of view. The electronic approach is based on more traditional digital computer technology and we have developed algorithmic and hardware solutions to the VLKB problem. We assume a logic programming inferencing mechanism and a relational model for the management of the knowledge base. The interface between the inferencing mechanism and the extensional database becomes one of partial match retrieval. We bridge the gap between the two parts through the use of a surrogate file structure for the representation of both rules and facts.

In the optical approach we are concerned with the higher speed and massive inherent parallelism of optics and how they might be used to advantage in storage, transport and processing of very large knowledge bases.

In the general case a logic programming front end engine requires equal access to all rules and facts. Because of this generality we have taken a surrogate file approach to the management of the VLKB. Surrogate files are transformations that yield improved performance because of smaller size, more rigid structure and the opportunity for parallel operations. In prior work we analyzed several possible surrogate file structures and selected concatenated code words (CCW) as the approach that offered the most generality and potential performance improvements. Basically, a CCW is a concatenation of transformed values and one can utilize the individual components of the CCW as well as the entire word.

We have designed a parallel back end database machine. The basic idea of the machine is to reduce the amount of fact data transferred from the secondary storage system while satisfying the user query. In order to do this the CCW files are distributed over many disks which are under the control of many surrogate file processors. The CCW entries are used to greatly reduce the amount of data that needs to be searched in response to a query. Relational database operations are performed on the surrogate files thus further reducing the amount of data needed to be retrieved. When all operations are complete the results are then sent to the logic programming engine for further use.

Another important advantage of the CCW surrogate file technique is that it can be used for the indexing of rules expressed as logic programming clauses, where the matching between constants, variables and structured terms is required to test for unifiability. The

CCW is obtained from the arguments ın the clause as well as the predicate name of the head of the clause. Each code word is divided into a tag field and a value field. The tag field can represent any argument type including lists, structured terms, variables and constants. The value field contains the transformed representation of the corresponding argument according to the content of the tag field. Thus, the CCW approach allows for the representation and processing of rules and facts in a unified manner.

We have analyzed the CCW technique in a variety of ways including simulation on the Connection Machine and the development of a demonstration system. The demonstration system consists of Prolog, INGRES and specially developed modules. The system allows for the generation and management of surrogate files of various types, the execution of Prolog programs and the management of rules and facts.

To handle very large dynamic databases we have developed the dynamic random sequential access method (DRSAM). It is based on an order preserving dynamic hashing method derived from linear hashing. The performance of DRSAM was evaluated and found to be efficient for range queries as well as random access. With order preserving hashing, the hashed key values are not generally uniformly distributed over the storage address space. To deal with the nonuniformity we have extended DRSAM with additional control structures.

The use of optics in the management of VLDB's can be divided into three parts; storage, transport and processing. Storage involves the use of optical *disks or holograms*. It appears to be feasible to obtain at least two orders of magnitude increase in optical disk transfer rates through the use of multiple beam reads. These data could be input to optical fiber for transport to optical database processors. We have developed an initial design of a system for the performance of various VLKB operations. It can perform selection, projection and equijoin as well as the filtering of ground clauses. The configuration includes two spatial light modulators and a large photodetector array for photon/electron conversion.

12.2. Introduction

Knowledge based systems consist of rules, facts and an inference mechanism that can be utilized to respond to queries posed by users. The objective of such systems is to capture the knowledge of experts in particular fields and make it generally available to nonexpert users. The current state of the art of such systems is that they focus on narrow domains, have small knowledge bases and are thus limited in their application.

As these systems grow, increased demands will be placed on the management of their knowledge bases. The intensional database (IDB) of rules will become large and present a formidable management task in itself. But, the major management activity will be in the access, update and control of the extensional database (EDB) of facts because the EDB is likely to be much larger than the IDB. The volume of facts is expected to be in the gigabyte range, and we can expect to have general EDB's that serve multiple inference mechanisms. In this report we assume that the IDB is a set of rules expressed as logic programming clauses and the EDB is a relational database of facts.

In order to set the stage for the problem that we are interested in, consider the following simple logic programming problem:

```
1. grandfather(X,Y) ← father(X,Z), parent(Z,Y)
2. parent(X,Y) ← father(X,Y)
3. parent(X,Y) ← mother(X,Y)
4. father(pat, tiffany) ←
   father(don, louise) ←
        .
        .
        .

5. mother(mary, louise) ←
   mother(lisa, tiffany) ←
        .
        .
        .

6. ← grandfather(X, joan)
```

The first three clauses form the IDB of rules for this problem, the next two sets form the EDB of facts and the last statement is the goal. To solve the problem (satisfy the goal), we must find the names of the grandfathers of joan. For this we search the father and mother facts on the second argument position, finding values for the first argument position that can be used later. Thus, we need to find joan's mother and father before finding her grandfathers. If we ask a similar but slightly different query

$$\leftarrow \text{grandfather(tom, X)}$$

we search the first argument of the father and mother facts in attempting to satisfy it.

Consider the following general goal statement of a logic programming language

$$\leftarrow r\,(X_1,X_2, \cdots ,X_n).$$

In this case, values for some subset of the X_i's will be given in the process of trying to satisfy its goal. Since the subset of the X_i's is not known in advance and can range from one to all of the values, this places considerable requirements on the relational database management system that supports the logic programming language. In fact, in order to insure minimum retrieval time from the relational database all of the X_i's must be indexed. With general indexing the index data could be as large as the actual EDB. In order to considerably reduce the amount of index data yet provide the same capability, we have considered surrogate files. Obviously if not all of the X_i's can take part in goal satisfaction then the indexing strategy will change, however in this report we will assume the most general case in which all of the X_i's are active.

Retrieving the desired rules and facts in this context is an extension of the multiple-key attribute partial match retrieval problem because any subset of argument positions can be specified in a query and matching between terms consisting of variables and functions as well as constants should be tested as a preunification step.

In the context of very large knowledge bases the question arises as to how to obtain the desired rules and facts in the minimum amount of time. Three reasonable choices of indexing schemes to speed up the retrieval are superimposed code words (SCW), concatenated code words (CCW) and transformed inverted lists (TIL)* surrogate file techniques. Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen

* SCW, CCW and TIL will be singular or plural depending upon the context.

hashing func'' .s on the original terms. In [BER87a], SCW, CCW and TIL surrogate files were discussed in terms of the structures, updating procedures, performance of relational operauons on the surrogate files, and possible architectures to support them.

We have implemented an experimental deductive database system on the Connection Machine of model CM-1 with 32 K processors without the specialized disk system (data vault), to test the surrogate file schemes for various partial match queries and implicit join operations. This experiment reveals that surrogate file loading from VAX 8800 frontend can be a major bottleneck in parallel processing environments since it not only involves the surrogate file reading time from slow secondary storage but also incurs high communication overheads which is required to broadcast the code words to large numbers of processing elements. The implementation details and performance evaluations are described in Appendix 12-G of this report.

To speed up the relational algebra operations based on the CCW surrogate files, a parallel backend database machine is proposed in section 12.3. The basic idea of the proposed database machine is to reduce the amount of fact data to be transferred from the secondary storage systems to satisfy a query by performing the relational algebra operations on the CCW surrogate file first. The database machine consists of a number of surrogate file processors (SFP's) and extensional database processors (EDBP's) operating in SIMD mode. Each surrogate file processor has an associative memory to speed up the relational algebra operations on the CCW surrogate files. Surrogate file processors and extensional database processors are connected to other processors of the same type through multistage interconnection networks. The performance of the proposed system for parallel relational algebra operations was evaluated.

An important advantage of surrogate file techniques is that they can be easily extended for the indexing of the rules expressed as Prolog clauses, where the matching between constants, variables, and structured terms is required to test the unifiability. [RAM86], and [WAD87] have extended the SCW structure for the indexing of Prolog clauses and [SHI87] has extended the CCW structure to index the rules and facts in unified manner. In section 12.4, we introduced a surrogate file scheme CCW-2 which can be used for partial unifications among first order terms in a very large logic programming environment.

To handle very large dynamic data bases, a new and efficient access method called the dynamic random sequential access method (DRSAM) is introduced in section 12.5. It is derived from linear hashing with order preserving. The performance of DRSAM was evaluated and the file structure found to be efficient for range queries as well as random access. With order preserving hashing, the hashed key values are not generally uniformly distributed over the storage address space. To deal with non-uniform distributions, DRSAM was extended with prOper control mechanisms and the resulting file structure is called EDRSAM.

In section 12.6, we introduced an initial design for the optical implementation of various operations in Very Large Data/Knowledge Bases. The system is capable of performing projection, selection and equi-join as well as filtering of ground clauses in an efficient way because it takes full advantage of the parallel nature of optical information processing. Data stored in optical disks is retrieved and processed optically by a configuration involving two spatial light modulators and a large photodetector array where the photon-to-electron conversion takes place.

12.3. Backend Relational Algebra Machine Based on CCW Surrogate File

12.3.1. Concatenated Code Word Surrogate File

In the context of very large knowledge bases the question arises as to how to obtain the desired rules and facts in the minimum amount of time. A reasonable choice of indexing scheme to speed up the retrieval is concatenated code word (CCW) surrogate file technique discussed in [BER87a]. A surrogate file is constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms.

Suppose we have a fact type called borders which is given as follows:

borders (Country_1, Country_2, Body_of_Water).

For a particular instance

borders (korea, china, yellow sea).

we would first hash the individual values to obtain binary representations.

H(korea) = 100...01

H(china) = 010...00

H(yellow sea) = 110...00

Then the CCW of the fact is generated by simply concatenating the binary representations of all attribute values and attaching the unique identifier of the fact as follows:

100...01| 010...00| 110...00| 00...01.

The unique identifier is also attached to the fact and serves as a link between the two. It is used as a pointer to the EDB or can be converted to an actual pointer to the EDB by dynamic hashing schemes.

The retrieval process with the CCW surrogate file is as follows:

1) Given a query, obtain a query code word (QCW) by concatenating binary representations corresponding to the attribute values specified in the query. The portion of the query code word for the attribute values which is not specified in the query is filled with don't care symbols.

2) Obtain a list of unique identifiers to all facts whose CCW's satisfies

QCW=CCW

by comparing the QCW with all CCW's in the CCW file for that fact type.

3) Retrieve all facts pointed to by the unique identifiers obtained in step 2 and compare the corresponding attribute values of the facts with the query values to discard the facts not satisfying the query. These are called "false drops". The facts satisfying the query are called "good drops". The false drops are caused by the non-ideal property of hashing functions.

4) Return the good drops.

Compared with other full indexing schemes such as inverted lists [CAR75], CCW surrogate file technique yields much smaller amounts of index data; about 20% of the size of the EDB [BER87a] while the inverted lists may be as large as the EDB. In terms of maintenance the surrogate file shows considerable advantages. When a new tuple is added to a relation the CCW is generated and added to the surrogate file. In the case of inverted lists each list must be processed. Similar operations must be performed for deleting tuples from a relation. When changes to an existing tuple are made, the surrogate file entry must be changed and the proper inverted lists must be changed.

An important advantage of CCW surrogate file technique is that they can be easily extended for the indexing of the rules expressed as Prolog clauses, where the matching between constants, variables, and structured terms is required to test the unifiability. [SHI87] extended the CCW structure to index the rules and facts in an unified manner.

An additional benefit obtained from using the CCW surrogate file approach is that relational operations can be performed on the CCW surrogate files [BER87a]. To satisfy a query, interrelated relational algebra operations on the EDB are required, so by performing relational algebra operations on the CCW surrogate file first, considerable processing time can be saved. Relational operations on CCW surrogate files are a kind of relational operation algorithm using indices [BLA77, MEN86]. However, using inverted list type indices in parallel relational algebra operations is very difficult, because the problem of synchronizing accesses to the indices without completely serializing the actions of the processors executing in parallel has not been solved yet [BIT83]. On the other hand, a CCW surrogate file is a set of transformed binary code words corresponding to the tuples of a relation, so it can be horizontally partitioned into subfiles and distributed over the parallel processors to be processed concurrently.

In [CHU88], CCW surrogate file technique was analysed on the basis of storage space required for the surrogate file and time to retrieve the desired facts from the EDB. The analysis shows that most of the query response time for fact retrieval is used for the surrogate file processing when the relation is very large (10^9 bytes) because of the sequential searching of all surrogate file code words. With smaller relations (10^5 bytes) surrogate file processing time is negligible compared with EDB access time.

To speed up the relational algebra operations based on the CCW surrogate file, a back-end database machine is proposed. The database machine consists of a number of surrogate file processors (SFP's) and EDB processors (EDBP's) operating in SIMD mode. Each SFP has an associative memory to speed up the relational algebra operations on the CCW surrogate files. Since CCW's are quite compact and regular, they are mapped well to the associative memories. SFP's and EDBP's are connected to other processors of the same type through multistage interconnection networks.

In section 12.3.2 the proposed architecture is introduced. The relational algebra operation algorithms for the architecture are explained in section 12.3.3. Section 12.3.4 shows the performance of the proposed architecture for relational algebra operations.

12.3.2. Structure of the Backend Relational Algebra Machine

A general structure of a backend database machine which contains multiple processors for the management of a very large extensional database of facts is shown in Figure 12.3.1. We assume that there are gigabytes of data stored on the EDB disk subsystems and there are corresponding CCW surrogate files stored on the SF disk subsystems. Suppose that the user is interested in retrieving fact data satisfying a condition from a particular relation. Then the selection query is transferred to the backend controller from the host computer and a query code word (QCW) is constructed in the surrogate file processor manager (SFPM) using the proper hashing functions. The QCW is then broadcast to the proper Surrogate File Processors (SFP's) to be used as a search argument. The SFP compares the QCW with each CCW and strips off the unique identifiers of matching CCW's. Each extracted unique identifier is sent to the EDB processor manager (EDBPM) and passed on to the EDB processor (EDBP) which contains the fact with that unique identifier. The EDBP will access the block containing the fact, compare the retrieved fact with the original query to check that it is a good drop and then send it to the host computer.

The basic idea of the proposed backend database machine is to reduce the number of EDB blocks to be transferred from the secondary storage systems by performing the relational operations on the surrogate files first. To speed up the relational algebra operations on the

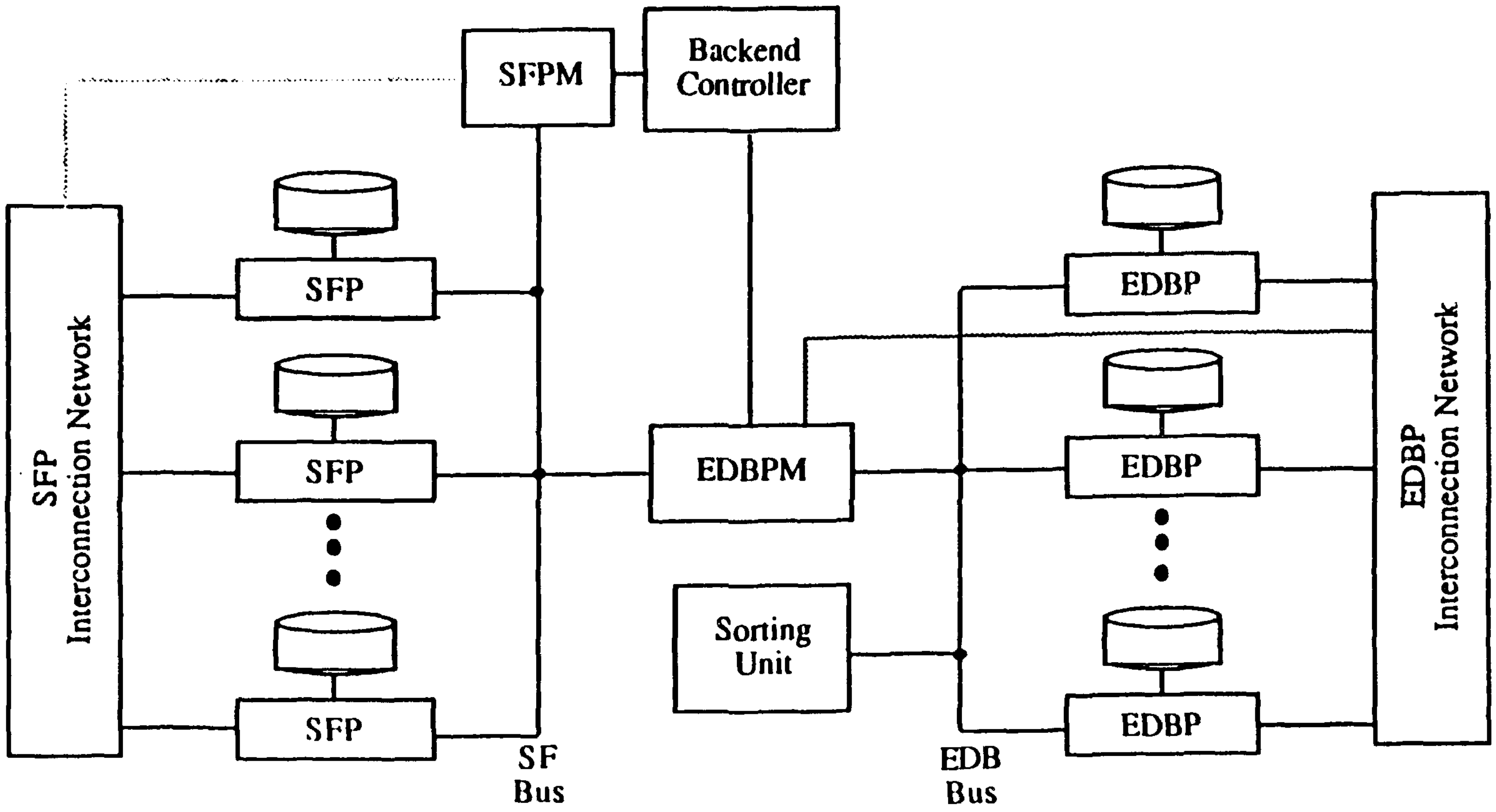


Figure 12.3.1 Relational Algebra Machine Based on Surrogate Files

surrogate files, each relation's surrogate file blocks are evenly distributed over a number of surrogate file disk subsystems so that the SFP's can process the surrogate files concurrently.

As shown in Figure 12.3.2, a SFP is equipped with an associative memory unit to perform the searching operation efficiently. Associative memories are very fast because they use content addressing and parallel searching, but they are generally costly and rigid in data format. However, the format of the surrogate file is regular and maps very well into the associative memory and cost of the associative memory hardware is decreasing as VLSI technology advances. Additionally, associative memories can be used for relational operations, such as selection and join, because associative memories can perform many associative operations such as equal to, not equal to, less than (or equal to), greater than (or equal to), maximum, minimum, between limits, outside of limits, and others depending on the structure. In our design, we used word-parallel bit-serial (WPBS) associative memory which consists of two-dimensionally accessible memory and an array of processing elements. A word slice is a unit for memory read and write and a bit slice is a unit for arithmetic and logical processing. A bit-parallel associative memory [RAM78, DAV86], whose memory cells have comparison logic, is faster than a WPBS associative memory but is much more complex. Current status of associative memories and associative processors are reviewed in [WAL87].

To balance the speed of an associative memory, multiple disks controlled by two disk controllers constitute a surrogate file disk subsystem and are attached to a SFP through double buffers. In our system, one partition of surrogate file of a relation is stored consecutively within each disk subsystem so that the surrogate file blocks of a relation can be sequentially transferred to the associated surrogate file processors. By associating a disk subsystem to each surrogate file processor, we lose some flexibility in allocating processors to a query processing but surrogate file blocks can be accessed rapidly.

The surrogate file processors are connected through the SFP interconnection network. Since there are a number of surrogate file processors, the flexibility and speed of the interconnection are very important factors determining the overall performance. The mapping between SFP's will be permutation, selective broadcasting, or broadcasting depending on the distribution of operand surrogate files among the SFP's (we consider the pair of a SFP and a surrogate file disk subsystem as a single unit and call it a SFP), the number of available SFP's, and algorithms of relational algebra operations. To handle all the mapping modes we chose a multistage Omega network [LAW75] implemented with 2 by 2 switching elements with four functions; straight, exchange, upper broadcast, and lower broadcast. Thus, any one SFP can broadcast a block to the rest of the SFP's with uniform delay. The SFP interconnection network will operate in circuit switching mode to facilitate the surrogate file block transfers.

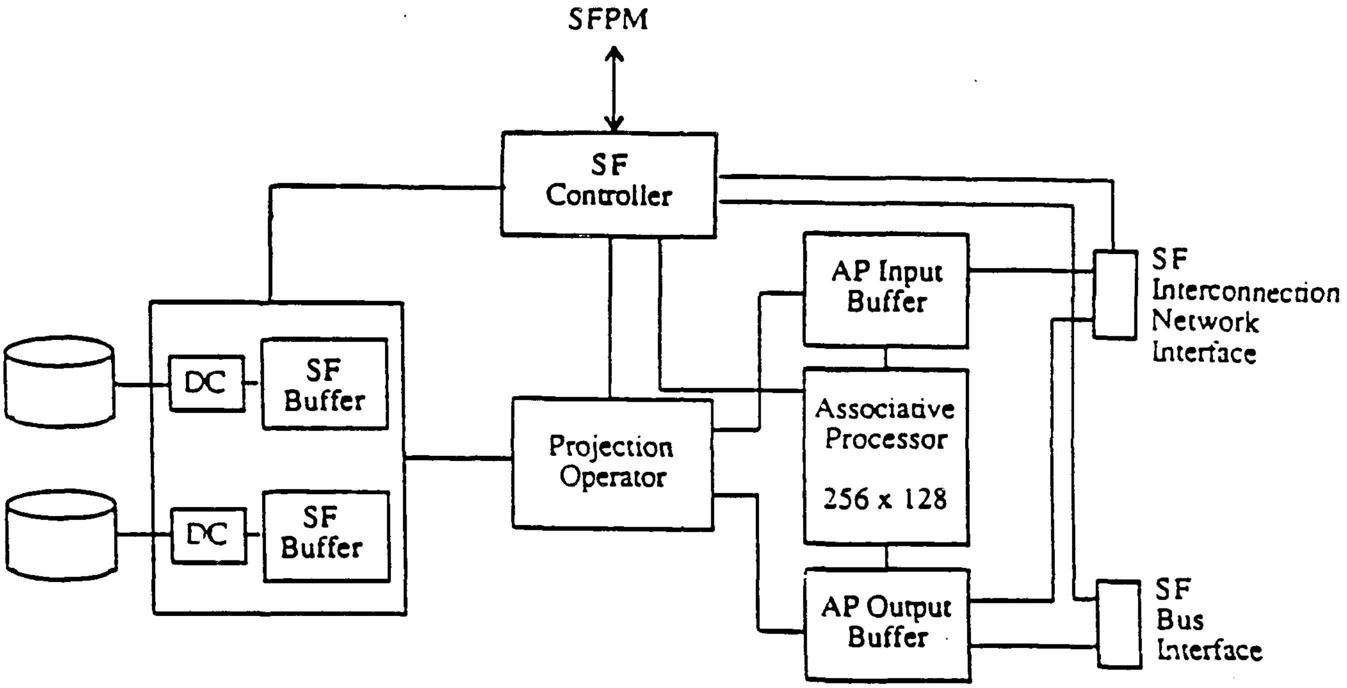


Figure 12.3.2. Structure of a Surrogate File Processor

The structure of the EDB processor manager is shown in Figure 12.3.3. If a fact unique identifier is sent from a SFP to the EDB processor manager (EDBPM), the EDBPM finds the EDB processor (EDBP) containing the corresponding fact by accessing a directory and sends the unique identifier to the EDBP. The EDBPM has a result buffer to collect the operation results from the EDBP's.

The structure of a EDBP is shown in Figure 12.3.4. In case of fact retrieval, a fact block corresponding to the received unique identifier is accessed by the EDBP. We assume that EDB blocks are randomly distributed within a disk subsystem, so to speed up the block access a disk cache is provided per EDBP. Once the block is available in the working memory of the universal operator, the operator searches the block with the unique identifier, extracts the fact corresponding to the unique identifier, and compares the extracted fact with the query to check that it matches. The universal operator is a kind of general purpose processor and perform all the tuple-wise relational algebra operations as well as statistical aggregation functions. Through the EDBP interconnection network, facts can be transferred from

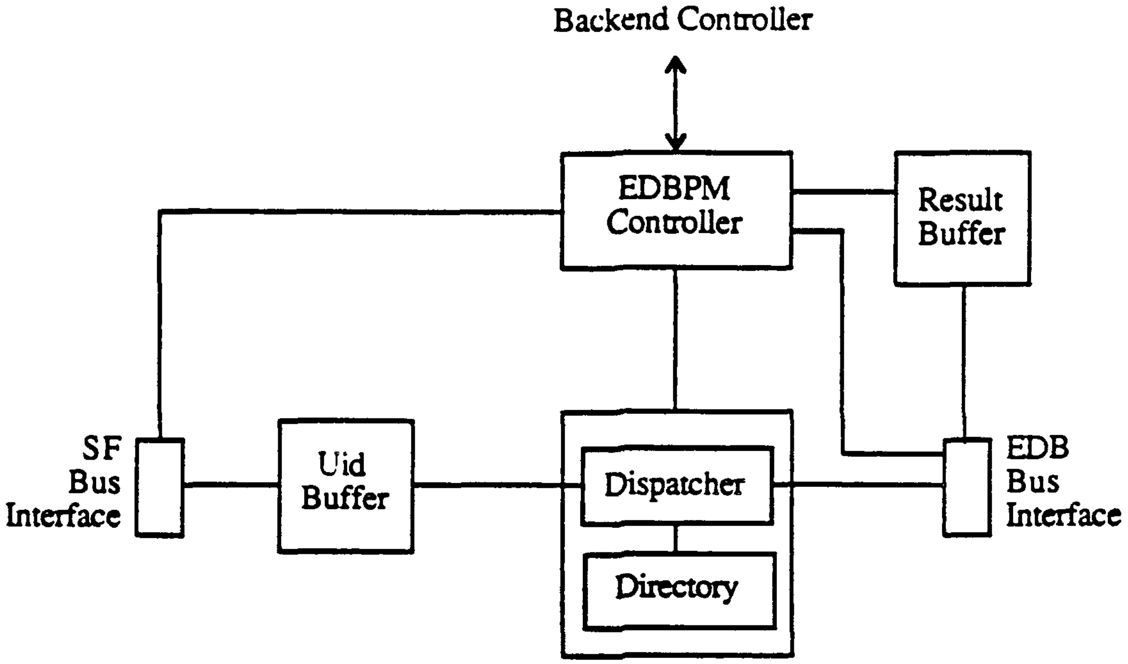


Figure 12.3.3. Structure of Extensional Database Processor Manager

one EDBP to another. We decided to use the multistage Omega network for the interconnection of EDBP's. The EDBP interconnection network operating in packet switching mode to facilitate the frequent transfer of facts between EDBP's in case of join operations.

A hardware sorting unit [KIT87] is available and can be accessed by a EDBP through the EDBP bus. The sorting unit can also be used for duplicate removal which is a part of other operations such as union, difference, and projection.

The processing mode of the backend system is SIMD or MIMD depending on the distribution of surrogate files and relations over the processors and assignment of the processors to a given operation. If all the processors are working for a single operation, then it becomes a SIMD mode, but if processors are partitioned into a number of groups and each group of processors is assigned a different operation, then the processing mode is MIMD at the group level. To operate either in SIMD or MIMD mode, the interconnection network must be

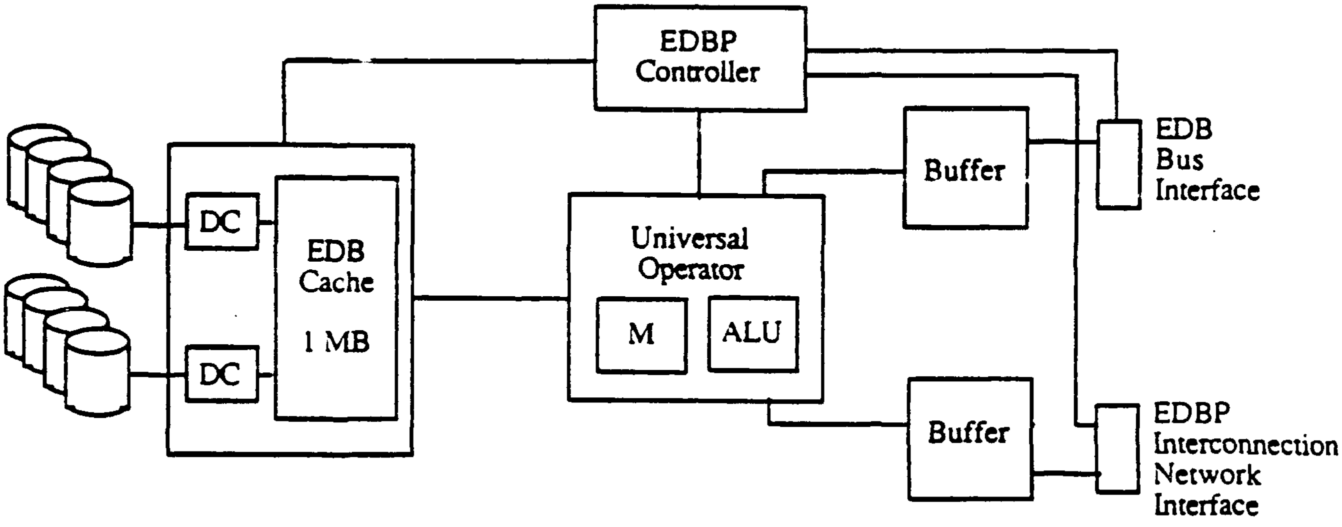


Figure 12.3.4. Structure of a Extensional Database Processor

partitionable. A multistage Omega network of size 2^m can be partitioned into independent subnetworks of different sizes with the requirement thai the addresses of all the I/O ports in a subnetwork of size 2^i agree in (m – i) of their bit positions [SIE80].

As the size of the EDB increases, the system can be upgraded by adding a cluster of SFP's and EDBP's to the existing system configuration. If we store the related relations and their surrogate files on a cluster of SFP's and EDBP's, then each cluster of processors are working on different queries and the processing mode of the system becomes multiple SIMD (MSIMD). In this case, the inter-cluster interconnection would be separated from the intra-cluster interconnections. The backend controller of each cluster would be a cluster controller and be in charge of communication with other clusters and the global backend controller through a cluster bus as shown in Figure 12.3.5.

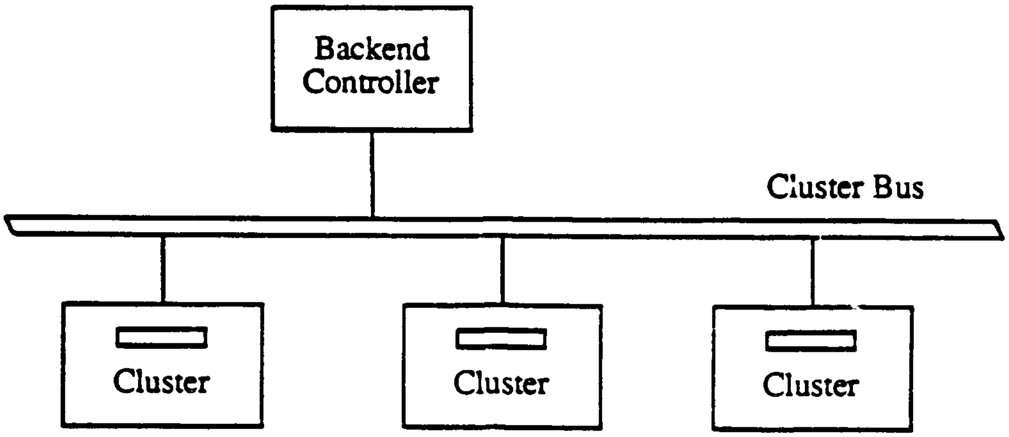


Figure 12.3.5. Multiple Relational Database Machine Configuration

12.3.3. Relational Algebra Operations in the Backend Database Machine

12.3.3.1. Selection Operation

To select on a particular attribute position, the SFP's execute a comparison, such as equal to, not equal to, greater than or equal to, or less than or equal to between the binary representation of a code word and the hashed value of the constant specified in a selection query. To retain the ordering between the binary representations of a attribute position, order-preserving hashing [GAR86] is necessary.

Each SFP retrieves a block of CCW's and does the projection on the binary representation of the specified field and unique identifier, then loads the projected CCW's to the associative memory. The comparand register of the associative memory is loaded with the hashed constant. The bit positions of the input mask register corresponding to the hashed constant is filled with 1's while other bit positions are filled with 0's. If there is any match, the corresponding fact unique identifier is sent to the EDBPM.

As soon as any fact unique identifier is received by the EDBPM, it finds the EDBP containing the corresponding fact block and sends the unique identifier. The EDBP retrieves the fact block by usinȝ the unique identifier, searches the block with that unique identifier, and performs the actual selection operation on the fact. Due to the pre-selection operation on the surrogate file, the number of fact blocks to be accessed from the secondary storage system is usually very small compared to the total number of fact blocks of a relation.

12.3.3.2. Join Operation

There are three main algorithms for the join operation; sort-merge, hash-partition, and nested-loop join algorithms. The performance of the sort-merge join algorithm for the non-equijoin operation is as good as that for equijoin operation, because once the two operand relations or subrelations are sorted, the merging step can handle the equijoin and the non-equijoin in the same way by performing the corresponding comparison operation. The database machine DELTA [SAK86, ITO87] has multiple relational database engines composed of sort-merge units and performs the sort-merge join algorithm. If a database machine has sort-merge units, the selection operation is interpreted as a join operation between a relation and a constant value specified in a query.

The hash-partition join algorithm is adopted by the database machine GRACE [KIT84]. Each operand relation is partitioned into a number of buckets depending on the hash value of the join attribute, then matching is performed within each bucket by a processor assigned to that bucket. Usually the hash-partition join algorithm is better than the sort-merge join algorithm in the case of the equijoin operation because sorting creates a total ordering of the tuples in both relations while the hashing simply groups related tuples together in the same bucket [DEW85]. However, in case of non-equijoin, the operation of each processor is not limited to a single bucket and the workload of the processors may not be uniform. One problem of the hash-partition join algorithm is the bucket overflow caused by the non-uniform distribution of the join attribute value. In this case, rehashing of the overflow bucket is necessary.

It has been shown that the nested-loop join algorithm takes advantage of different operand sizes and the processing time is inversely proportional to the number of processors, while in the case of the sort-merge algorithm after a certain number of processors, duplicating the number of processors causes very little decrease in the execution time. The reason is that, after a certain stage, the degree of parallelism is divided by two at each merge pass [VAL84].

Our proposed database machine adopts the nested-loop join algorithm because the associative memories in each processor can easily perform the parallel execution of the nested-loop join operation.

If we assume that the CCW surrogate files of two operand relations are evenly distributed over a number of SFP's, the nested-loop join algorithm is executed as follows:

1) Each SFP reads a block of CCW surrogate file of the smaller relation from the associated surrogate file disk subsystem, projects it on the join attribute and unique identifier and loads it into the associative memory.

2) Each SFP reads a block of CCW of the larger relation from the associated surrogate file disk subsystem, project it on the join attribute and unique identifier and store it in the associative processor input buffer.

3) One SFP broadcasts the projected block from step 2) to the rest of the SFP's which already have a block of CCW surrogate file of the smaller relation in their associative memories from step 1).

4) Each SFP searchs the associative memory with the broadcasted projected CCW's as searching arguments one by one. If there is a match, the pair of unique identifiers of the two matched CCW's are sent to the EDBPM.

5) Repeat step 3) and step 4) until all the projected blocks in step 2) are broadcasted.

6) Repeat step 2) to step 5) until all the CCW's of the larger relation are retrieved from the surrogate file disk subsystems.

7) Repeat step 1) to step 6) until all the surrogate file blocks of the smaller relation are retrieved and searched.

In step 4), as soon as any unique identifier pair is received by the EDBPM, the EDBPM finds the EDBP's containing the corresponding facts and transfers the unique identifier pair to those EDBP's. If a single EDBP contains the two operand facts then that EDBP performs the actual join operation on the two facts retrieved, otherwise one EDBP transfers a projected fact to another EDBP containing the other operand fact, then the join is performed. To reduce the amount of communication through the EDB interconnection network the smaller projected fact is transferred. Projection is performed on the join attribute, attributes involved in the output relation, and the unique identifier. The pre-join operation on the CCW surrogate files is overlapped with the actual join operation on the facts.

12.3.3.3. Projection Operation

Performing projection (including the duplicate removal) on the surrogate file as a substitute for projection on the EDB is not useful because of the false drops. Generally, the hashing function is not ideal, so different attribute values may have the same hashing function output. Therefore, we can not remove the duplicate binary representations of surrogate file code words. Thus, projection (including the duplicate removal) must be performed by the EDBP's on the actual relations. EDBP's can use an external sorting unit to remove the duplicate tuples, or can use a duplicate removal algorithm developed for multiprocessor system [BIT83] depending on the size of the relation to be projected, number of processors, and the size of the memory in each processor. Other relational operations such as set union and set difference have the same problem.

12.3.4. Performance Analysis of the Proposed Architecture

In this section, we analyse the performance of the proposed relational algebra machine for the selection and join algorithms. For this purpose, we used a simplied queueing network model shown in Figure 12.3.6 and estimate the average response time of a query by using the product form solution of tandem queues [TRI82]. The service times of EDBPM and the EDBP's may have distributions more regular than the exponential distributions, however as long as we are interested in the average response times, we can use M/M/1 queues safely [KOB78]. We assumed that

1) the workloads of the SFP's and the EDBP's involved in a relational algebra operation are uniform,
2) disk I/O operations and processor operations are executed concurrently whenever possible,
3) pre-operations on the surrogate files by the SFP's and the actual operations on the facts by the EDBP's are executed concurrently whenever possible.

12.3.4.1. Selection Operation

In case of a selection operation, the average arrival rate of extracted unique identifiers to the EDBPM is determined by the size of surrogate file of a operand relation, number of SFP's involved in the pre-selection operation, and the selectivity. We estimated the pre-selection

Parameter	Values
Average seek time of a disk	28 msec
Rotational delay of a disk	8.3 msec
Data transfer rate of a disk	2 MB/sec
Block size	4 KB
Effective EDB block access time	10 msec
Interconnection network speed	10 MB/sec
SF and EDB bus speed	50 MB/sec
Memory bandwidth	10 MB/sec
Unique id. dispatching time	10 μsec
Projection rate	6 MB/sec
Time for loading a word to an associative memory	0.1 μsec
Associative memory searching time for n bit-slices	(0.5 + 0.1 n) μsec
Time for extracting a responded word from the associative memory	0.2 μsec
Byte comparison time in EDBP	0.5 μsec

Table 12.3.1. Summary of Parameter Values Used for Performance Analyses

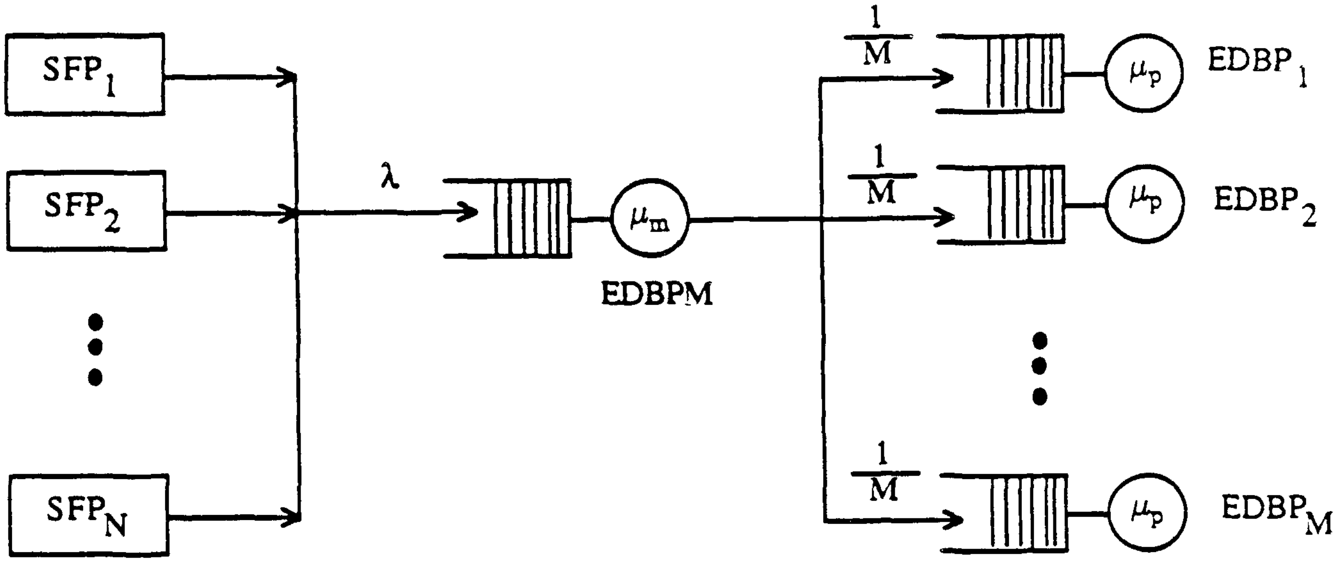


Figure 12.3.6. Simplied Queueing Model of a Selection/Join Operation

time and the average service rate of the EDBPM and a EDMP by applying the parameter values specified in the Table 12.3.1. Since there is no fact transfer between EDBP's, we don't need to consider the effect of network contention.

Figure 12.3.7 shows the total response time of a selection operation on a relation R as a function of the selectivity (defined as the ratio of the cardinality of the output relation to that of the operand relation), the number of SFP's (M), and the number of EDBP's (N) when

Cardinality of an operand relation R = 10^6
Size of a tuple = 100 bytes
Size of a unique identifier = 3 bytes
Size of a concatenated code word = 20 bytes
Size of a selection attribute = 15 bytes
Size of the binary representation of a selection attribute = 3 bytes
Size of an output tuple = 100 bytes

$$\alpha = \frac{\text{number of false drops}}{\text{number of good drops}} = 0.1 .$$

Since the hashing functions used to generate the CCW are not ideal, there are a certain number of false drops. We assumed that the total number of matched code words is (1 + α) times the actual number of facts satisfying a selection query.

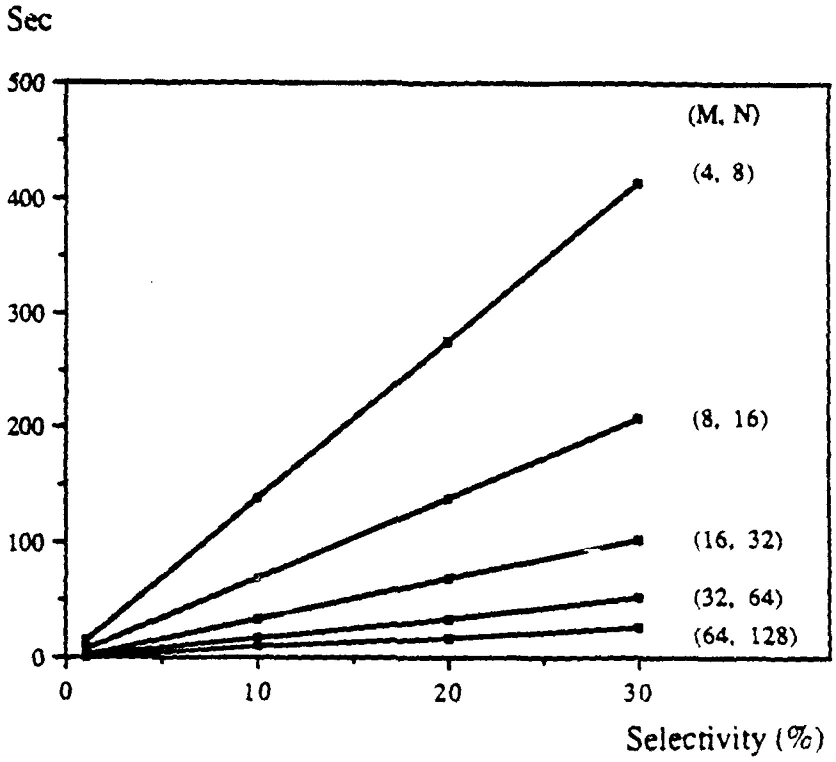


Figure 12.3.7. Performance of Selection Operation
(Cardinality of R = 10^6)

When the selectivity is low, the pre-selection time on the surrogate file is dominant and the total response time will decrease as the number of SFP's increases. As the selectivity increases, the number of EDB blocks accessed will increase and the actual selection time on the facts is dominant. Thus, as the number of EDBP's increases the response time decreases linearly. Actually, as the selectivity increases the effective EDB block access time would be reduced due to an increased disk cache hit ratio. However, we assumed that the effective EDB block access time is constant in evaluating the total selection time.

To reduce the number of EDB block accesses when the selectivity is high, we can search an accessed block with the search attribute values specified in the query, instead of searching the block with a unique identifier. In this way we can find all of the desired fact in that block. Then, we store the block number in memory and whenever a unique identifier of a fact within that block is received, we discard the unique identifier since we already retrieved that fact.

12.3.4.2. Join Operation

In the case of a join operation, the surrogate file size of the two operand relations, the number of SFP's involved, and the join selectivity will determined the average arrival rate of unique identifier pairs to the EDBPM. Since a EDBP performs join operation on two operand facts, one of which may be transferred from other EDBP, we have to consider the network delay caused by the conflict in the network. However, usually the size of the projected fact is small, so the transfer time of a projected fact is very small compared to a EDB block access time. Therefore the network contention would not be serious unless the join selectivity is very high. Furthermore, the fact transfers are overlapped with EDB block accesses and join operations. It has been shown that any permutation can be performed in a Omega network within three passes of the network [VAR87], which corresponds to the analysis given in [THA81]. Thus, we simply assumed that the effective network speed is one third of the nominal network speed. We used the parameter values in Table 12.3.1 to evaluate the average response time.

The response time of a join operation on two operand relation R_1 and R_2 is plotted in Figure 12.3.8 as a function of the cardinality of R_2, the number of SFP's (M), and the number of EDBP's (N) when

Cardinality of $R_1 = 10^6$
Cardinality of the output relation = cardinality of R_2
Size of a tuple in R_1 and R_2 = 100 bytes
Size of a unique identifier = 3 bytes
Size of a concatenated code word = 20 bytes
Size of a join attribute = 15 bytes
Size of the binary representation of a join attribute = 3 bytes
Contribution of each operand relation to an output tuple = 30 bytes

$$\beta = \frac{\text{Number of unique identifier pairs extracted}}{\text{Cardinality of output relation}} = 1.1 .$$

Due to the non-ideal hashing functions, the number of joinable code word pairs is larger than the cardinality of the output relation, and β accounts for this effect.

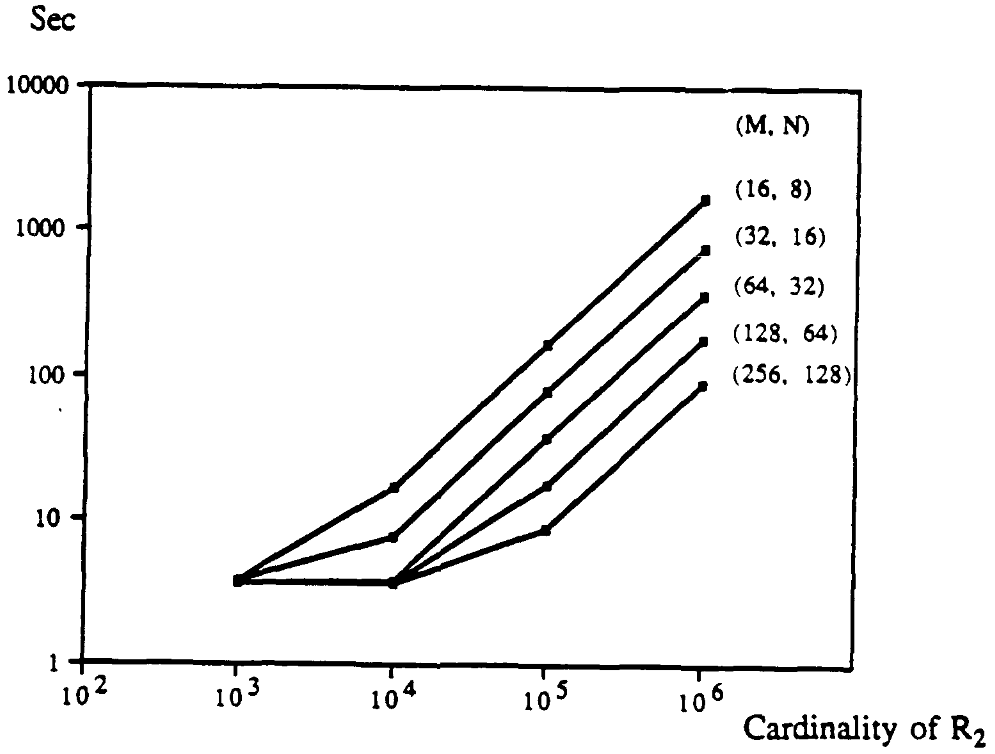


Figure 12.3.8. Performance of Join Operation between R_1 and R_2 (Cardinality of R_1 = 10^6, Cardinality of output relation = Cardinality of R_2)

As the total size of the two operand relations increases, the response time increases. The pre-join time is dominant when the join selectivity is low since the pre-join operation is performed on every pair of surrogate file blocks while the actual join operation is performed on the two operand facts. Therefore, when the selectivity is low, as we increase the number of SFP's we can decreases the total join processing time. When the join selectivity becomes high, actual join operation time is dominant due to the increased number of random EDB block accesses. In this case, we can reduce the number of EDB block accesses by storing the retrieved facts in the working memory and reuse it whenever it is requested. For an example of an equijoin operation, if the average number of tuples in R_1 (R_2) which have same join attribute value is C_1 (C_2), then a tuple of R_1 which is participated in the semijoin of R_1 by R_2

can be joined with C_2 tuples of R_2 on average. Thus, if we store that tuple in the memory, we can reuse it ($C_2 - 1$) times later. For the same reason, the tuple of R_2 can be reused ($C_1 - 1$) times later. In Figure 12.3.8, we used a constant EDB block access time independent of the join selectivity. Additional details are given in Appendix 12-A.

12.4. Management of General First Order Terms and Rules

For conventional database models, a number of physical file organization techniques such as B-tree indexing and hashing are well developed. For complex objects, however, these traditional file organizations would be inappropriate for the following reasons.

1. Primary key, which can be used as a unique identifier, does not exist. Retrieving the desired tuples of knowledge bases can be viewed as an extension of the multiple-key attribute partial match retrieval problem because any subset of argument positions can be specified in a query.
2. Lexicographical orders among general terms cannot be decided when the knowledge base contains variables within tuples. So, we cannot use ordered file organizations such as B-tree or indexed sequential file.
3. An attribute value can be decomposable into atomic values, and thus a query can be based on subcomponents of an argument (e.g. f(r(a,X), Y)). If a fully inverted list is to be used, all the subcomponents should be indexed along with the position in a term. It will result in substantial amount of index data.
4. Frequent execution of join operations are expected due to the transitive clauses defined by rules. As traditional hashing or indexing schemes are based on sequential processing, we need a new file organization suitable for parallel processing.

For efficient processing of VLKB, a *new physical file organization* scheme called the surrogate file [BER87a] has been proposed. In this section, some general term indexing techniques based on surrogate files are introduced.

12.4.1. General Term Indexing via Surrogate Files

This section concerns the design of surrogate files for general terms and clauses so that we can exploit surrogate files as a basis for unification-based retrieval. We consider 5 general term indexing schemes based on the surrogate file techniques described in the previous section. Ramamohanarao and Shepherd [RAM86] developed a superimposed code word scheme for both ground terms and structured terms. Colomb [COL86] developed another superimposed coding technique for structured terms. Wise and Powers [WIS84] proposed the Field Cord Word (FCW) scheme for clauses containing variables and structured terms. Wada et al. [WAD87] presented a similar indexing scheme to the FCW*. An extension of the CCW to terms containing variables was developed by Shin and Berra [SHI87].

* Wada called it as a Structured Superimposed Code (SCCW). But, it can be viewed as a CCW type indexing.

A brief comparison among the five proposed schemes mentioned above is given in Table 12.4.1. In the table, functor encoding means whether the predicate name (principal functor) is included as a part of the code. Intuitively, when the number of clauses with the same predicate name is large, functor encoding is not required (they can be clustered). But, for the rules, the number of clauses with the same predicate name is generally small, and thus it does require the inclusion the primary functor as a part of indexing structure. Variable encoding schemes can be classified into two types; field-set and bit-set. The field-set method can be viewed as simulating the mask register of associative memories. That is, the argument position representing a variable is masked out. On the other hand, the bit-set method uses one bit to indicate that the corresponding argument is a variable. In the bit-set scheme, if the tag bit is set, the rest of the bits are used to indicate common variables. In the SCW scheme, the variables appeared in the QCW do not require special considerations since they are simply not broadcast for matching. However, in CCW and FEW the position of variables in the QCW as well as in the indexing code is important. Underlying computer languages and computer architectures are also briefly summarized.

	SCW-Based Schemes		CCW-Based Schemes		
	Ramamohanarao and Shepherd	Colomb	Wise and Powers	Wada et al.	Shin and Berra
Coding Method	Extended SCW Slice 2 level	Extended SCW Slice 2 level	FEW String 1 level	SSCW String 1 level	CCW-1 String 1 level
Principal Functor	no	yes	yes	yes	yes
Variable	Bit-Set (Index)	Bit-Set (Index)	Field-Set (Q & I)	Field-Set (Q & I)	Bit-Set (Q & I)
Structure	Mask Bits	Position Encoding	Field Encoding	Field Encoding	Tag Set (CCW-2)
Underlying Language	Prolog (Interpreter)	Prolog (Interpreter)	Prolog/Epilog (Interpreter)	Prolog (Interpreter)	Prolog/PARLOG (Compiler)
Computer Architecture	-	Associative Memory	Tightly-Coupled Multiprocessor	PHI (PSI Net.)	Associative Memory

Table 12.4.1 Comparison Among Clause Indexing Schemes

12.4.1.1. Superimposed Code Word (SCW) Type Indexing

Superimposed code words can be used to represent variables and structured terms stored in the knowledge bases. Ramamohanarao and Shepherd [RAM86] proposed the use of extra mask bits to indicate whether a certain argument position is variable. A record descriptor is presented by m bits of SCW code and k bits of mask bits. Thus, when a structured term is not allowed, the value k is the number of arguments in the term. Thus the matching condition is for ith argument specified in the query is

QCW = (SCW .AND. QCW) .OR. M(i)

where M(i) is a mask bit which is set to 1 when the corresponding position is a variable.

Suppose that we use 5 mask bits each of which is corresponding to the positions in a term (1,0), (1,1), (1,2), (2,0) and (3,0)* respectively. Figure 12.4.1 shows the descriptor for p(X,a,X), p(g(a,X),Y,b) and the matching process for the query ← p(g(X,Y),Z,b). To use this scheme for complex term indexing, however, the structure of terms should be known in advance, and it is not indicated how to store the don't care bit in the secondary storage**. The use of bit-slice organization and two level indexing scheme are also proposed to speed up the searching procedure.

Colomb [COL86] eliminates the need for the mask bits by encoding the position information within the SCW, and proposed to encode the principal functor as well as the arguments. When compared to the SCW scheme proposed by Ramamohanarao and Shepherd, Colomb's method can accommodate more position information. But, the retrieval procedure is less efficient than other schemes.

12.4.1.2. Concatenated Code Word (CCW) Type Indexing

Wise and Powers [WIS84] proposed a clause indexing scheme called Field Encoded Word (FEW). Wada et al [WAD87] proposed a similar scheme, called Structured Superimposed Code Word (SSCW). These two schemes are based on the CCW concept. The main idea is to divide the code word for an argument into k, where k is the number of subterms in the argument. For example, if a term has 4 argument and a code word with 256-bits is used,

* This address notation of a term is proposed by Colomb [COL86]. For example, the variable X in the term p(g(a, r(b,X)), c, d) has address (1,2,2) since it is the second argument of r(b,X), which is the second argument of g(a, r(b,X)), which is the first argument of p(g(a, r(b,X)), c, d).

** In case of CCW, the don't care bits are not stored in the disk. They can be considered as setting mask registers in the associative memory

Term	SCW	Mask bits
p(X.a.X)	0100 1000 0000	1**01
p(g(a,X),Y,b)	0111 1100 1000	00110

(a) Bit-String Representation of General Terms

CW(g) → 0001 0000 1000
C1 = Bit(4) .AND. Bit(9) .OR. Mask(1)

CW(b) → 0010 0100 0000
C2 = Bit(3) .AND. Bit(6) .OR. Mask(5)

Matching Condition = C1 .AND. C2

(b) Matching Process for ← p(g(X,Y), Z, b)

Figure 12.4.1 Extending SCW to General Terms (Ramamohanarao and Shepherd)

each argument can occupy 64 bits. Suppose that the first argument is the term f(a,X,r(X)). Then, the 64 bits of code word is further divided into four 16-bits code, each of which represents f, a, X and r(X) respectively. This scheme is shown in Figure 12.4.2.

The CCW-1 scheme was proposed for a parallel logic programming language PARLOG [CLA86]. By using *mode* declarations, the *guarded clauses* of PARLOG can be transformed to the *standard form*, where no structured term appears in the head of clauses. Thus, CCW-1 concerns only pure variables and constants. In this scheme, each CCW code corresponding to an argument has 1 bit tag to indicate whether the argument is a variable. The tag bit is used for bidirectional don't care matches as a preliminary step of unification. CCW-1 provides an efficient mechanism in searching possible candidate clauses as well as in detecting binding conflicts among shared variables in the early stage of execution.

CCW-2 is basically the same structure as CCW-1, which can be constructed by concatenating transformed code words obtained from the arguments along with the predicate name of the head of a clause. Each code word is divided into two fields; tag field and value field. Unlike CCW-1, however, the tag field can represent any argument type including lists and structu~ed terms as well as variables and constants. The value field contains the transformed representation of the corresponding argument according to the content of its tag

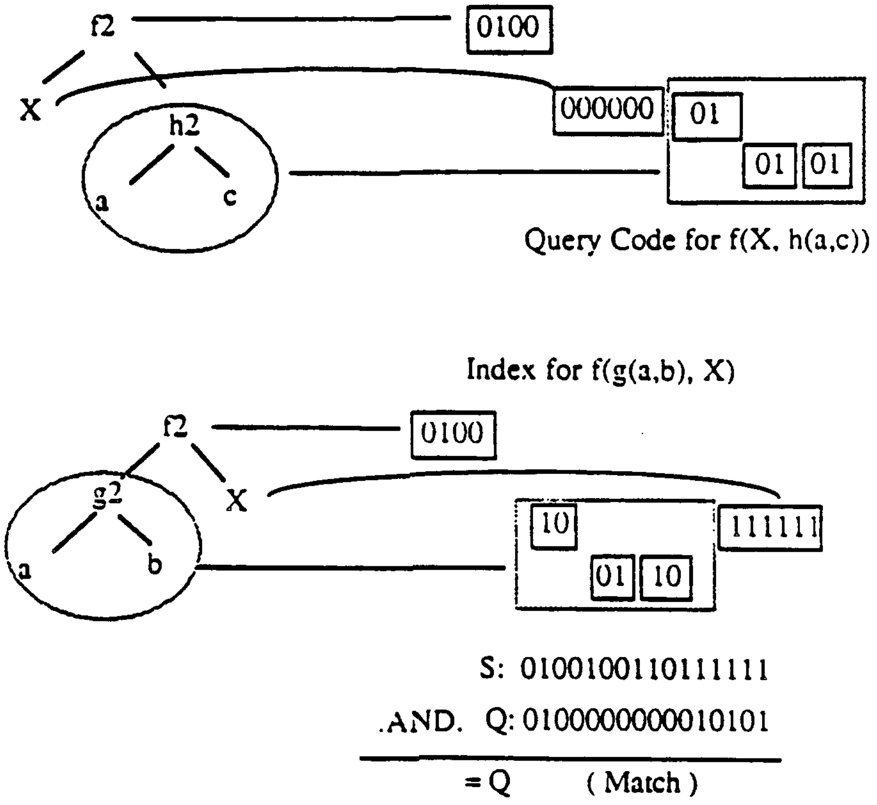


Figure 12.4.2 Extending CCW to General Terms [WAD87]

field. For example, if the tag indicates structured term, then the value field contains the hashed value of the primary functor, while if the tag is for a variable, the corresponding value filed represents the variable identification number. This scheme can be viewed as an augmentation of CCW-1 with the type checking mechanism. Table 12.4.2 shows an initial design of CCW-2 scheme. An example of CCW-2 code is shown in Figure 12.4.3.

Argument Type	Tag Field	Contents of Value Field
Constant	00x	Hashed Value of the constant
Function	01x	Hashed Value of the Primary Functor/Arity
List	100	Hashed Value of the CAR constant
	101	Variable ID for the CAR variable
Variable	11x	Variable ID

Table 12.4.2 CCW-2 Coding Scheme

In contrast to CCW-1, CCW-2 can be used for current Prolog systems and does not require mode declarations. Due to the type checking mechanism, it is expected that false drops

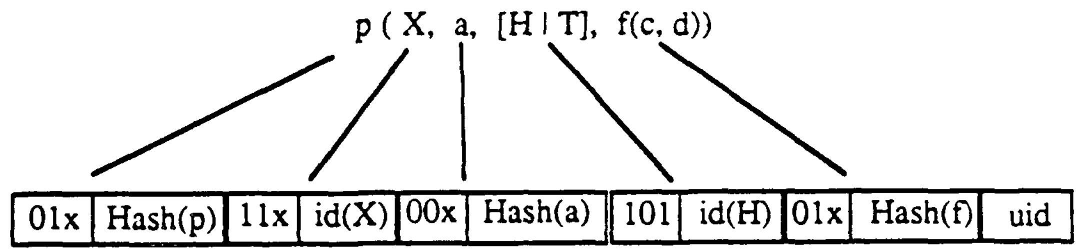


Figure 12.4.3 Extending CCW to General Terms (CCW-2)

can be considerably reduced when compared to previously proposed schemes without sacrificing the compactness and uniformity of CCW.

12.4.2. Design of an Associative Surrogate File Processor for CCW-1 and CCW-2

Due to the uniform and compact data structure of surrogate files, specialized hardware can be efficiently exploited. In this section, we present the design considerations on the hardware implementation of the surrogate files discussed in the previous section.

The CCW-1 scheme can be implemented by using specialized associative memories. Figure 12.4.4 shows an associative memory which can perform bidirectional don't care matches for CCW-1. To deal with the type checking mechanism of CCW-2, a slight modification on the matching logic of the associative memory is required, which is described below.

Assume that the QCW has *n* arguments. Then the matched clause heads should satisfy the following condition

M = m(1) .AND. ...... .AND. m(n)

where m(j) represents the matching condition for the j th argument.

If the ith argument of either QCW or CCW represents a variable, m(i) should be set to 1. Otherwise they should have the same type for the matching. Special care must be taken for the arguments representing a list where either a variable or constant can appear as the first element. Let $q_{i,j}$ and $c_{i,j}$ be the jth tag bits of the ith argument in the QCW and the record descriptor respectively, and let qbr_i and cbr_j be the binary representation for QCW and CCW respectively. Then m(j) can be represented as follows:

m(j) = ($q_{j,0}$ .AND. $q_{j,1}$) ; jth argument of the query is a variable

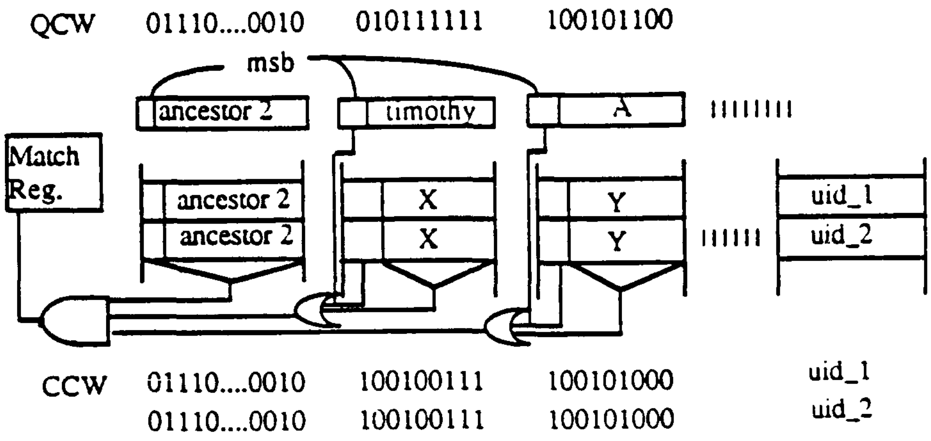


Figure 12.4.4 Hardware Implementation of the CCW-1

.OR. $(c_{j,0}$ .AND. $c_{j.1})$; jth argument of the record is a variable
; the first element of list can be either variable or constant
.OR. $(q_{j,0}$ .AND. $q_{j,1}$ .AND. $c_{j,0}$ .AND. $c_{j,1}$ .AND. $(q_{j,2}$ .OR. $c_{j,2})$
.OR. $(qbr_j = cbr_j)$; conventional matching

Current research regarding indexing general terms based on the surrogate file concept is limited to search operations rather than processing. Unlike the SCW approaches, the central focus of the general term indexing schemes based on the CCW should lie in developing surrogate file schemes suited to relational algebra operations and unification. Thus, we should be able to represent variable bindings in terms of surrogate files so that we can generate new QCWs from them. Additional details are given in Appendix 12-B and 12-C.

12.5. Inverted Surrogate Files

In this section, we present a new inverted model for surrogate files. This model, called the inverted dynamic surrogate files (IDSF), is based on a dynamic file structure which is designed to handle orthogonal range queries. The file structure is the Dynamic Random-Sequential Access Method (DRSAM). It is based on an order preserving dynamic hashing method derived from linear hashing [LIT80]. The main characteristic of this access method over previous ones is the *sequential allocation property* which leads to the natural adaptation of *elastic buckets* [LOM87] to directoriless organizations. DRSAM shows performance improvements over previous methods for both direct access and range queries performance.

Our previous work on inverted surrogate files resulted in the transformed inverted lists (TIL) [BER87, HAC88a]. This file structure is reviewed in Section 12.5.1. In Section 12.5.2, we discuss the design objectives for DRSAM and present the global approach which we followed. Then, we illustrate two variants of DRSAM, namely DRSAM0 and DRSAM1. This is followed by a discussion of new techniques that enhance the performance of DRSAM. In Section 12.5.3, we present the results of a performance analysis of DRSAM files.

Order preserving hashing leads to non-uniform distributions. In Section 12.5.4, DRSAM is extended with proper control mechanisms to adapt to non-uniform distributions. The resulting file structure is the extended DRSAM (EDRSAM). This file structure is applied to inverted surrogate files, leading to the IDSF model. In Section 12.5.5, we draw some conclusions and discuss future research issues.

12.5.1. Transformed Inverted Lists

In this section, the basic inverted surrogate file structure for partial match retrieval applications, namely transformed inverted lists (TIL) is presented. IDSF, described in Section 12.5.4, is an extension of the TIL concept to dynamic data/knowledge bases.

12.5.1.1. System Model

Single or multilevel indexing is a common technique used in database management systems (DBMS) for fast data access. In partial match retrieval, creating index files for more than one field in a record is necessary. The extreme case arises when every entry in a record is indexed independently and is referred to as inverted lists organization [DAT86, CAR75, SAL83]. One problem behind using inverted lists is that the size of the indices can become enormous, equal to or even larger than the database size.

Transformed inverted lists are similar to inverted lists with the main difference that indices are built based on the binary representation (BR) of the hashed output of a given field in a record of the database relation. In other words TIL represents the inversion of the surrogate image of a relation. Two TIL types, TIL1 and TIL2, are overviewed. A simple surrogate file representation of a relation is illustrated in Figure 12.5.1. The fields are referred to as arguments and the BR values for argument position 2 are listed. The actual values for br2, br4 and br5 are shown.

With very large data/knowledge bases, an application environment of the TIL technique would be the management of the extensional database of facts (EDB) within a logic programming context. It is assumed that many different relations (fact types) with varying degrees and cardinalities exist in the very large extensional database. Furthermore, the tuples are stored in such a way that one first accesses the relation followed by an access to a particular tuple via its unique identifier (Uid). The unique identifier could be derived from the "primary key" of the relation or a serially generated number attached to each fact. In this discussion, the Uids are assumed to be serially generated as this would be required in the context of knowledge bases where the existence of a primary key could be irrelevant.

Thus, the storage structure for the actual facts themselves would be very simple and a properly chosen dynamic hashing method could be used to guarantee retrieval of a given fact in at most two disk accesses. This presupposes that all secondary key retrievals will take place on the surrogate file or through post processing of the retrieved tuples if there are many different types of users of the same database.

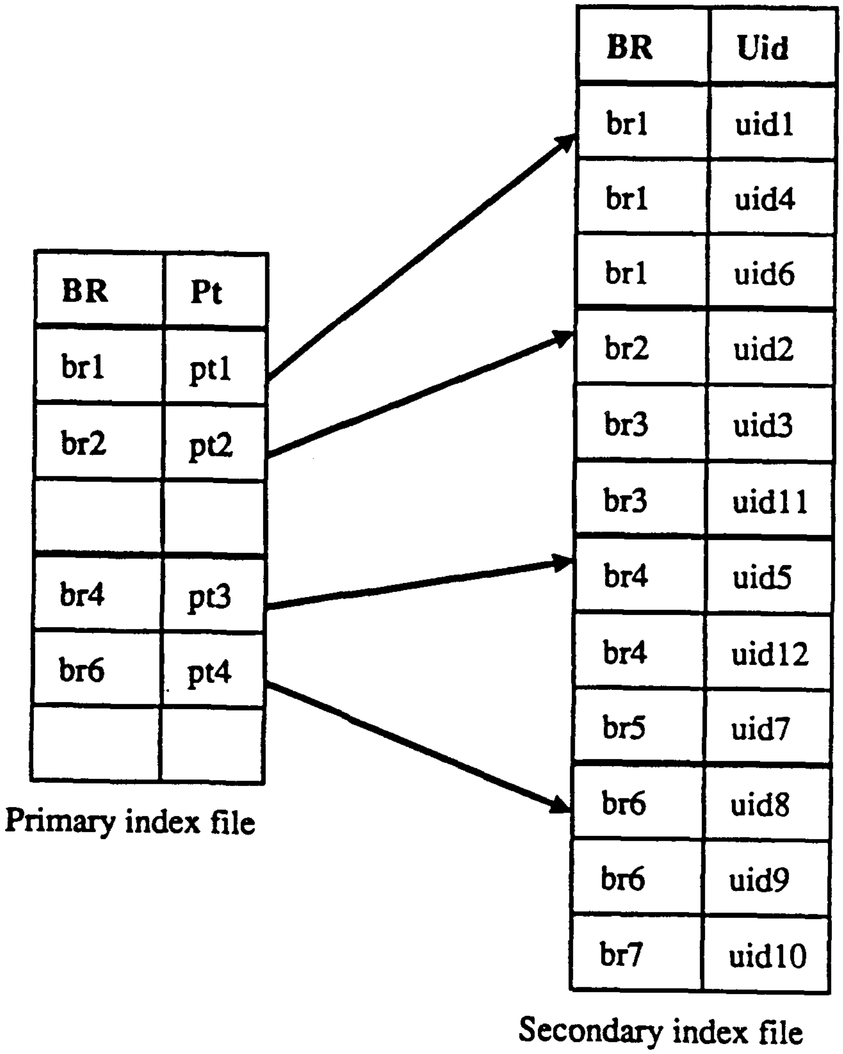
12.5.1.2. TIL1 Description

TIL1 consists of a two level indexed inverted list. Figure 12.5.2 illustrates the TIL1 organization for argument position 2 of the relation of Figure 12.5.1. The blank entries in the primary index file are usually included for updating purposes. The secondary index file for a given argument in a tuple is an ordered list of the BRs of the hashing function output of that argument with the attached unique identifier (Uid). The first entry in each block of this file is duplicated in the primary index file with an attached pointer to the corresponding secondary index block address. Furthermore, index files are partitioned in blocks of B bytes each. It is observed that the entries in the primary index file are ordered as well.

When a given BR is retrieved (say BR=br3), the primary index file is sequentially accessed using the BR as the search argument and the pointer to the secondary block address corresponding to that BR retrieved (pt2 in the example). Then the secondary index file is accessed in a direct mode and the required block(s) retrieved and searched sequentially for the occurrence(s) of the requested BR. The output is a list of Uids (uid3 and uid11 for the example) corresponding to the value of the request.

Uid	ar1	ar2	ar3	ar4
uid1		br1		
uid2		br2=010011010		
uid3		br3		
uid4		br1		
uid5		br4=010101011		
uid6		br1		
uid7		br5=010101110		
uid8		br6		
uid9		br6		
uid10		br7		
uid11		br3		
uid12		br4		

Figure 12.5.1 A surrogate image of a knowledge base relation



BR	Pt
br1	pt1
br2	pt2
br4	pt3
br6	pt4

Primary index file

BR	Uid
br1	uid1
br1	uid4
br1	uid6
br2	uid2
br3	uid3
br3	uid11
br4	uid5
br4	uid12
br5	uid7
br6	uid8
br6	uid9
br7	uid10

Secondary index file

Figure 12.5.2 TIL1 for ar2 of Figure 9.5.1

12.5.1.3. TIL2 Description

TIL2 is a three level indexed inverted list organization and is illustrated in Figure 12.5.3 for the same example relation. The difference between TIL2 and TIL1 lies in that the TIL1 secondary index file is now split into two files: the TIL2 secondary index file and the tertiary index file. Each entry in the tertiary index file consists of a Uid, so that the number of entries in this file is equal to the number of records in the database relation. Each entry in the TIL2 secondary index file consists of three fields: the BR of the hashed function output of an argument value (say BR=br6), a list length entry "L" that provides the number of records in the database that have the same entry value in a given argument position (2 for br6) and a pointer to the address of the first Uid in the tertiary file that has BR=br6. This pointer consists of the block address and a displacement value in the block.

The retrieval process for TIL2 is similar to TIL1, but requires the access of an additional index level.

12.5.1.4. Partial Match on Multiple Argument Positions

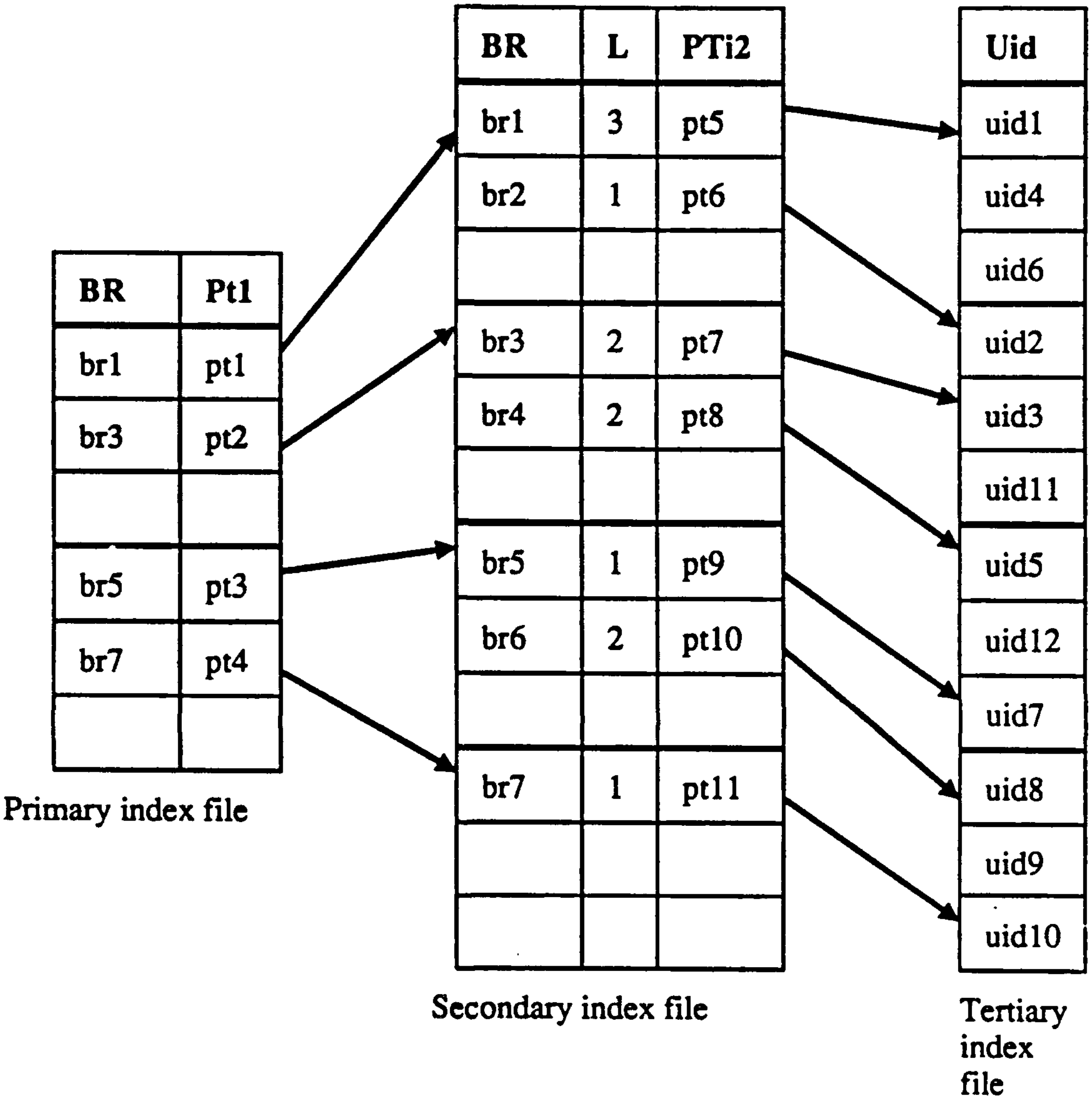
In conjunctive partial match queries, when more than one argument position match is requested in a query, the different outputs from the inverted lists searches need to be intersected. The outcome of the intersection is a set of Uids that complies with the query requirements. Finally, this set of Uids is used to directly access the main database for the retrieval of the matched records. The gain in retrieval time when using transformed inverted list is mainly due to the small size of the surrogate files and the fast access resulting from the indexing scheme. Only conjunctive partial match queries are considered in [HAC88a]. Disjunctive queries have the same level of complexity, with the list intersection operation replaced by a multiple sets union operation [REI72, MUN76, STO79].

It is noted that the inversion level of the surrogate files is determined by the application under consideration. Since the underlying target application involves logic programming and relational databases, fully inverted surrogate files throughout are assumed.

12.5.1.5. Comments on TIL structures

TIL file structures are found to be suitable for partial match queries on static files but with degraded performance and costly update operations when dealing with volatile files [HAC88a].

The results relied mainly on 1) a compact representation of the data and 2) a stable file as defined in [LAR81]. The analytical model which was derived is deterministic in nature and did not account for the overhead of overflow chains. Larson developed a stochastic model for



BR	Pt1
br1	pt1
br3	pt2
br5	pt3
br7	pt4

Primary index file

BR	L	PTi2
br1	3	pt5
br2	1	pt6
br3	2	pt7
br4	2	pt8
br5	1	pt9
br6	2	pt10
br7	1	pt11

Secondary index file

Uid
uid1
uid4
uid6
uid2
uid3
uid11
uid5
uid12
uid7
uid8
uid9
uid10

Tertiary
index
file

Figure 12.5.3 TIL2 for ar2 of Figure 9.5.1

indexed sequential files. With stable files, he determined that the expected number of accesses for a successful search would increase by 0.3. If the file is dynamic, the performance of these files and consequently TIL, degrades fast as the file structure results in unusually long overflow chains. These are detrimental to the retrieval as well as update performance of the file.

Furthermore, considering for example TIL1 file structures, the primary index file size is negligible with respect to the secondary level but still too large to fit in main memory. This is due to the fact that each entry in the primary index points to one physical bucket in the secondary index. To decrease the index size each entry should point to a group of physical buckets. Multibucket nodes are not new and were proposed by Lomet [LOM81, 83a, 83b, 87]. In Section 12.5.2.6 elastic buckets will be also discussed and applied to a sequential file structure (DRSAM).

TIL are built in a compact form where consecutive block ranges are ordered. Thus, if order preserving hashing functions are provided, TIL files can handle range queries. Also, TIL files can handle non uniform distributions because they are based on indexes.

12.5.2. The Dynamic Random-Sequential Access Method

In this section a new class of key-ordered dynamic file structures for associative searching is described. With order preserving hashing (OPH) functions the uniform distribution assumption does not generally hold and the structure should be extended with special control mechanisms to adapt to biased distributions. First a general structure is briefly introduced with its detailed discussion delayed until Section 12.5.4. Then, the motivations behind DRSAM are discussed and two variants of DRSAM (0 and 1) illustrated. To control file growth a new partial expansion technique specifically suited for key-ordered files is used. Finally, a generalization of the overflow bucket chaining method is discussed.

12.5.2.1. General File Structure

The general file structure which is advocated is a two level dynamic file derived from the structure of TIL1 [HAC88a]. The first level is designed to adapt to non-uniform distributions by proper partitioning of the key space into uniformly distributed subranges. The second level is composed of contiguous storage areas that are independently and dynamically managed with DRSAM using OPH. The global file structure is referred to as EDRSAM for Extended DRSAM and a typical EDRSAM file is shown in Figure 12.5.4. The subregions, referred to as DRSAM storage areas, are managed with DRSAM and are assumed to have uniform key distributions. The use of the small table and other control techniques is essential; as generally good randomizing OPH functions are not available and non-uniform distributions are detrimental to hashed files.

In the remainder of this section, different methods to manage the storage areas of EDRSAM are presented; specifically variants 0 and 1. DRSAM is derived from linear hashing LH [LIT80] with the additional feature that the key-sequential order is preserved within consecutive bucket ranges for efficient sequential processing, range queries as well as random access. The analysis assumes a contiguous storage allocation scheme but can be extended, similarly to LH, to accommodate distributed secondary storage allocation environments. Based on the results of this section and Section 12.5.3 a detailed study of EDRSAM is presented in Section 12.5.4.

12.5.2.2. File Design Objectives

The objective is the design of a file structure with the following characteristics:

1. Fast random access: the structure should be such that, given a search key, the access cost to the required record is optimal, i.e one disk access (or very near to the optimal value of one).

2. Fast sequential processing or range query access: given a range for a search key the structure should be such that the number of disk arm movements (i.e. disk seeks) is one followed by successive block reads for each contiguously allocated storage area of the file. This would hold if enough disk buffer (or swap) space is available.

3. Dynamic: the structure should be easily expandable with low maintenance overhead (insertion, deletion, change).

In Section 12.5.3 characteristics 1 and 3 are studied within the same context as LH. Characteristic 2 is achieved if the buckets that qualify for the range query are located in contiguous blocks in a sequential allocation environment, so that one disk access is performed followed by consecutive bucket reads; or the number of disk accesses is minimized in a distributed allocation environment. Therefore data locality between contiguous physically accessed secondary storage blocks is an important issue in the physical design of an access method. TIL [HAC88a] and ISAMf as discussed in [GHO69] files are typical examples of such concepts. The basic idea with DRSAM is to map logical nodes to consecutive physical buckets to achieve, for range queries, the consecutive retrieval (C-R) property introduced by Ghosh [GHO72, 86]. This assures a minimum disk arm movement and is referred to as the *consecutive or sequential allocation property in key-order.* In the next section order preserving dynamic hashing is discussed.

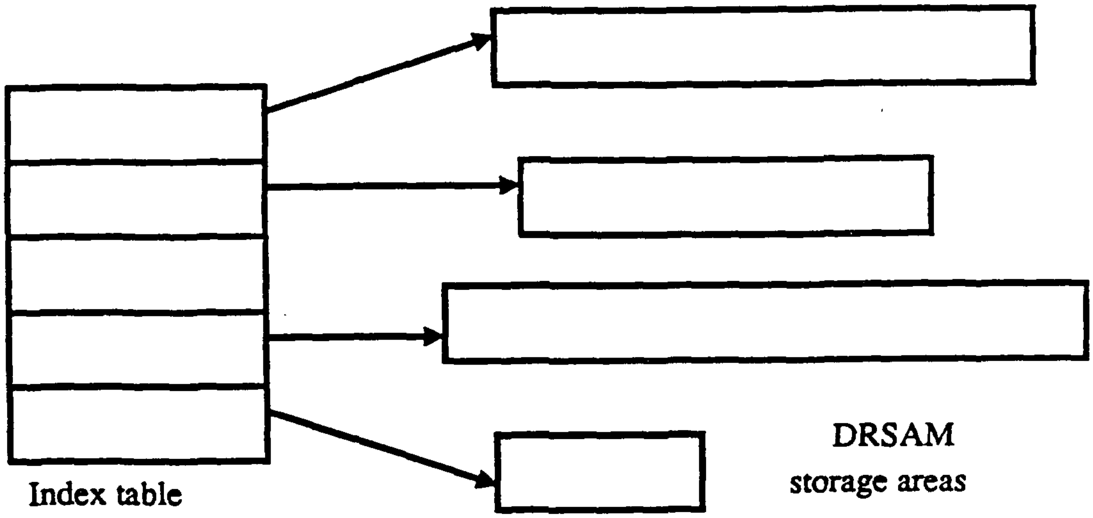


Figure 12.5.4 General file model

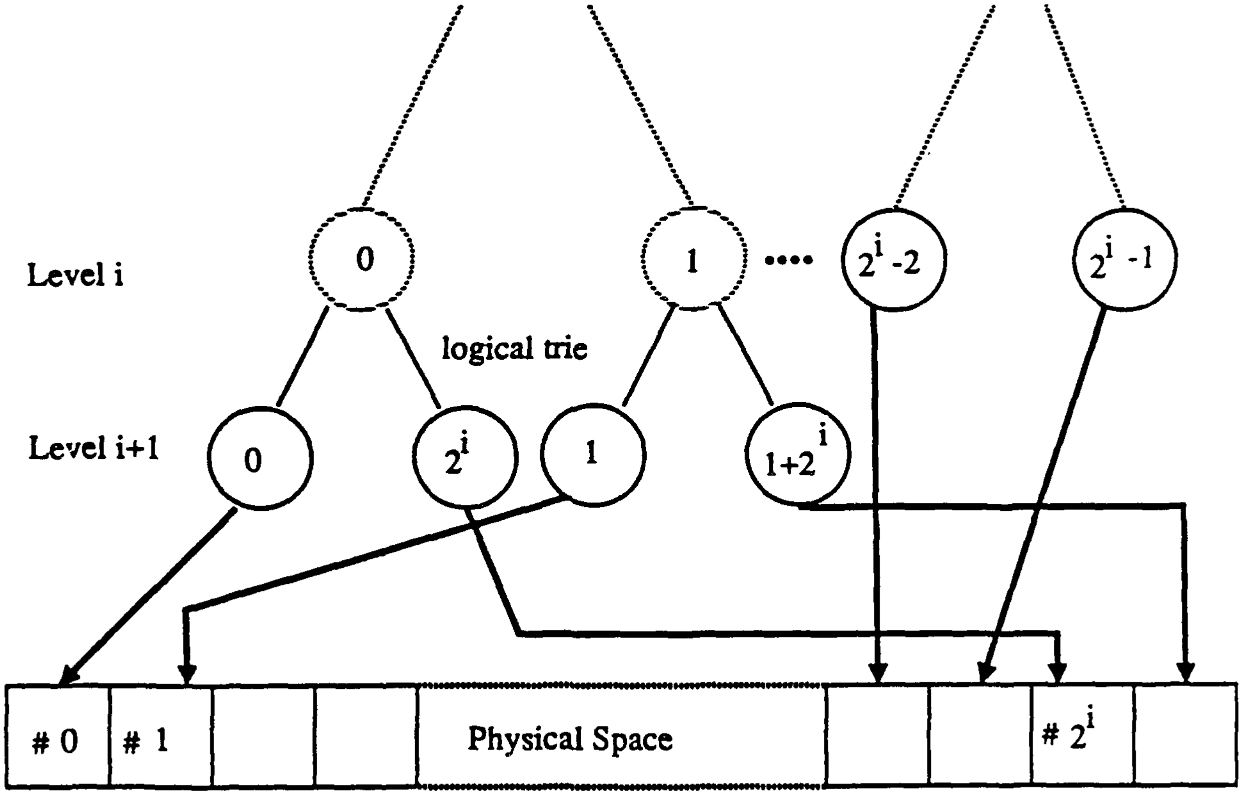


Figure 12.5.5 Logical/physical mapping for order preserving linear hashing

12.5.2.3. Order Preserving Hashing

In order preserving linear hashing schemes [BUR83, ORE83] as well as DRSAM, the file is viewed as a two-level logical trie mapped on a linear storage space. A split expands a parent bucket at level i onto its child buckets at level $i+1$. The leaves of the trie are at either level i or $i+1$. For example, the idea in [ORE83] is to use the mirror image of the leftmost bits of the key to achieve order preserving with linear hashing. This leads to the mapping of Figure 12.5.5: a logical bucket "x" (or node) at level i is the parent of the 2 nodes "x" and "x+N" at level $i+1$, with $N=2^i$. As the file grows the physical locations of buckets containing records in key-order are known but remotely located. Then, range queries that access multiple buckets will require an extra disk seek for each retrieved bucket.

In Figure 12.5.6 we illustrate how DRSAM maintains physical ordering in the mapping from logical to physical storage. A logical bucket "x" at level i is the parent of logical buckets "2x" and "2x+1" at level $i+1$. This mapping achieves data locality with the sequential allocation property. For range queries disk seeks are reduced, which provides a faster and more efficient access to secondary storage.

To achieve the mapping of Figure 12.5.6, one needs a *dynamic, sequential allocating* and *order preserving hashing* function (OPH_i). To implement DRSAM, beginning with a file of N buckets, the sequence of hashing function $(OPH_0, OPH_1, OPH_2, ..., OPH_i)$ should have the following properties:

$$0 \le OPH_i(Key) \le N-1$$

$$OPH_{i+1}(Key) = \begin{cases} 2 \times OPH_i(Key) \\ or \\ 2 \times OPH_i(Key) + 1 \end{cases} \qquad \text{for } all \;\; Key \;\; \text{and } i \ge 0$$

Using *Prefix(Key,i)* as the leftmost i bits of a key, a simple OPH_i for DRSAM is provided with:

$$OPH_i(Key) = Prefix(Key, i).$$

The prefix function is chosen for convenience but is not generally considered to be a good randomizing function. While the use of prefix instead of postfix bits has already been considered in an implementation, its context was completely different than the present one [RAM84]. Other randomizing functions could be devised and the reader is referred to Section 12.5.4 for a discussion of non-uniform distributions.

With dynamic hashing without directory, the splitting bucket is not necessarily the one that overflows. An overflow management scheme is therefore necessary. For linear hashing the sequence begins by splitting bucket 0 then 1 and so on until all buckets at level i split leading to a file at level $i+1$. The process is then repeated. Assuming contiguous allocation, the

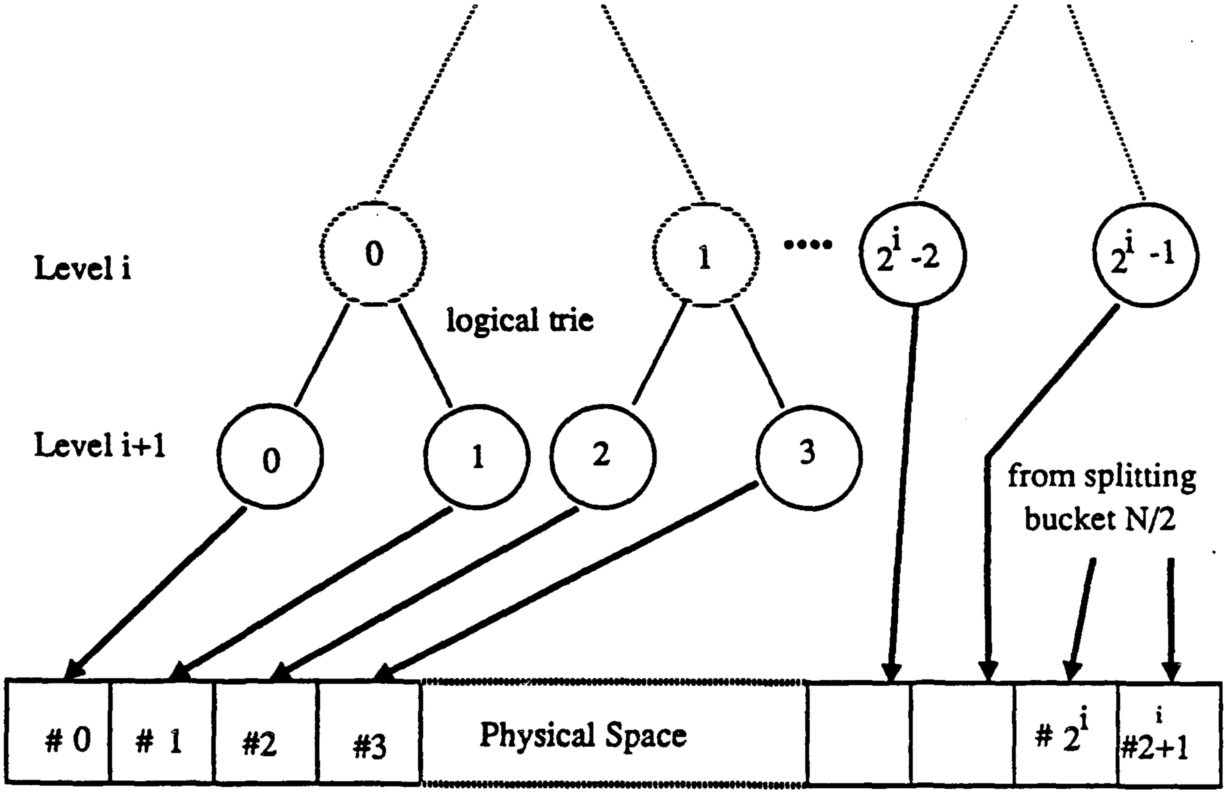


Figure 12.5.6 Logical/physical mapping for a DRSAM file structure

#0	#1	#2	#3
0	70	130	200
10	72	162	235
60			

Figure 12.5.7 DRSAM0 file at the beginning of an expansion cycle

logical and physical bucket addresses would be the same. This function is simple and provides a fast means to compute the address of a key. It is easy to check from comparing Figures 12.5.5 and 12.5.6 that the sequential allocation property does not lie in the class of linear hashing split functions advocated by Litwin. One of the problems is that bucket overwriting can occur if the physical address is equal to the logical address. Some essential modifications are necessary and two DRSAM files are illustrated next.

12.5.2.4. DRSAM Variant 0

In this section, the basic split sequence for DRSAM0 is illustrated. DRSAM0 is based on the same operating system characteristics as those of linear hashing. They are summarized by the one sided expansion of the file storage area on the secondary storage media. In DRSAM0, the split sequence is modified so that no overwriting is possible. Assume block sizes of 3 records and a file at level 2 (i.e. $N=4$ buckets). The home hash function is then $OPH_2(Key) = Prefix(Key,2)$ and the split hash function $OPH_3(Key) = Prefix(Key,3)$. With 8 bits encoding a key, the bucket ranges for level 2 are as follows:

bucket #0: 0 to 63.

bucket #1: 64 to 127.

bucket #2: 128 to 191.

bucket #3: 192 to 255.

For an expansion cycle from level 2 to level 3, each range splits in consecutive buckets. For example, bucket #0 splits onto buckets #0 and #1 and the respective range is then: 0-31 and 32-63, and so on. In general, bucket #x splits onto buckets #2x and #(2x+1). In Figure 12.5.7, the state of the file is shown with 9 insertions. Bucket #0 is full and the split pointer is initially at bucket #(N/2)= #2 (marked by an arrow).

Figure 12.5.8 shows the file state after the insertion of Key=12. This value hashes to bucket #0 and a split occurs with an overflow chain attached to bucket #0. Bucket #2 splits onto buckets #4 and #5. Note that bucket #2 is not used for the moment. It is referred to as the ***hole***. This hole expands and shrinks during the expansion cycle and the maximum number of buckets that would be unused at a given time can be shown to be $\log_2 N = i$. The split pointer is advanced to bucket #3. Assume that keys 120, 131, 121, 122 and 62 are inserted in sequence: first 120 goes in bucket #1, then 131 in bucket #5 (as bucket 2 has already split and is at level 3 now). Figure 12.5.9 shows the status of the file at this stage.

Then comes 121, a collision occurs and bucket #3 splits onto buckets #6 and #7. The bucket split pointer "folds back" to bucket #1 as the consecutive buckets #2 and #3 are now empty and can be used to expand bucket #1. Figure 12.5.10 shows the state of the file after inserting Key=121. With 122 inserted, bucket #1 splits on buckets #2 and #3 and the split pointer folds back to bucket #0 as shown in Figure 12.5.11. Finally, inserting Key=62 induces

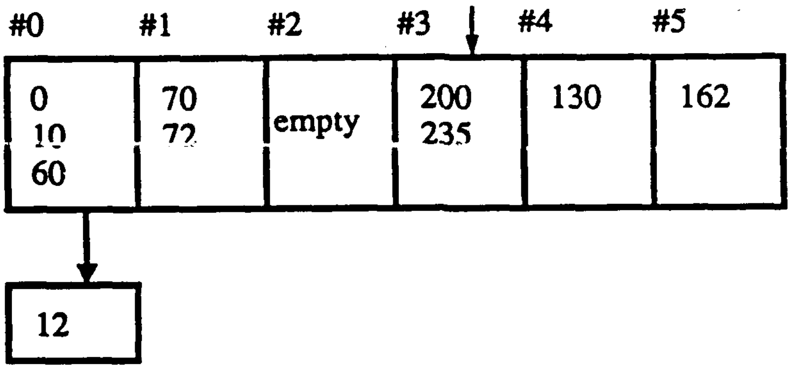


Figure 12.5.8 DRSAM0 example after
one split from Figure 9.5.7

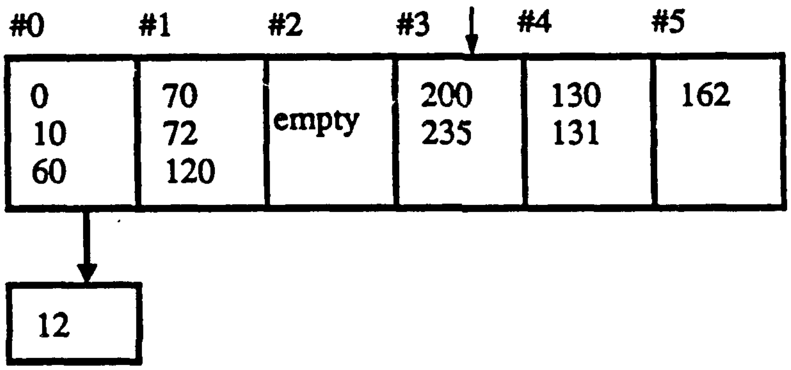


Figure 12.5.9 DRSAM0 example after
key insertions with no splits

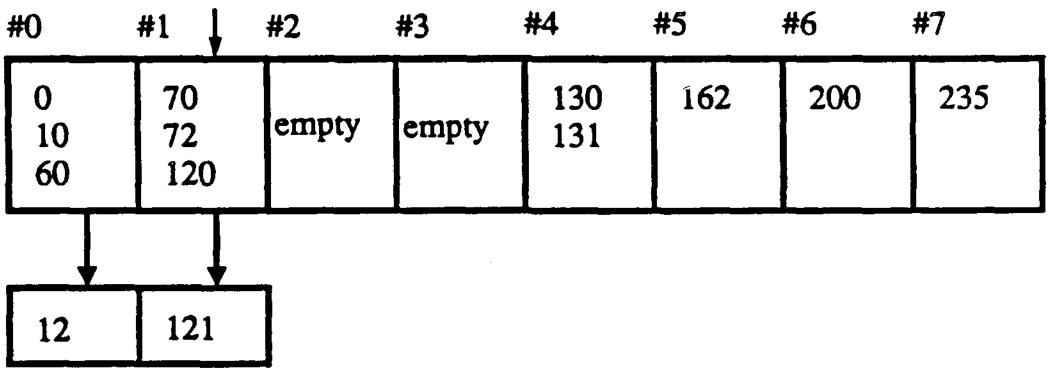


Figure 12.5.10 DRSAM0 example after
two splits from Figure 9.5.7

a collision and bucket #0 splits onto buckets #0 and #1. At the end of the process, the file has undergone a full expansion cycle and is at level 3. The split pointer is advanced to bucket #4 and a new expansion cycle can begin. The status of the file is shown in Figure 12.5.12.

In this example, the resulting load factor is low (0.625). The load factor is expected to be similar to LH, and with an uncontrolled split mechanism Litwin reports an average load factor which is lower than the one of EH (around 0.60). Controlled splitting techniques are applied as well as a new partial expansion method derived from Lomet's elastic buckets [LOM87] to improve on the load factor while keeping the near optimal direct access performance.

12.5.2.5. DRSAM Variant 1

If Litwin's sequential split sequence should be followed, a simple way to avoid overwriting is to use another physical space for the expansion of the file. For every split, two new buckets are created while the old one (parent) is freed. What happens if one needs to merge back? It is clear that the freed bucket should be recovered. Then the file should be able to expand in two directions which is not a fair requirement on the operating system and contradicts the basic assumption of one sided expansion/contraction of the file. In this section, this problem is solved with variant 1 of DRSAM.

Basic Model

For DRSAM1, the same logical expansion/contraction sequence as linear hashing is used. To avoid physical buckets overwriting the operating system file management features are slightly extended: for an expansion, the basic idea is to relocate the logical bucket on a newly allocated larger storage space while the previous space is freed. This operation is nevertheless irreversible and the freed bucket cannot be recorvered for a subsequent merge operation. Like linear hashing, the storage area is only allowed to expand (contract) from one of its boundaries. The extension is that the other boundary is allowed to move to return contiguous storage space to the operating system. For the proper design of DRSAM1, this feature is an essential but realistic characteristic of the operating system.

The split (merge) sequence follows a sequential ascending (descending) order on the set of logical buckets which form a trie structure S. The sequence is based on the full ordering operation defined on the trie as:

For any two logical buckets (or nodes) $LN(x,i)$ and $LN(x',i')$ in the trie S:

$$\textbf{if } x=x' \textbf{ then } LN(x,i) < LN(x',i') \equiv i < i'$$

$$\textbf{if } i=i' \textbf{ then } LN(x,i) < LN(x',i') \equiv x < x'$$

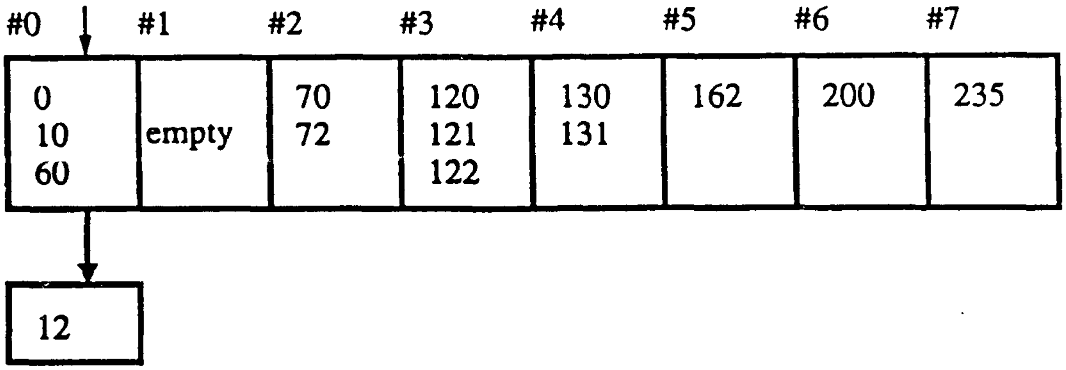


Figure 12.5.11 DRSAM0 example after three splits from Figure 9.5.7

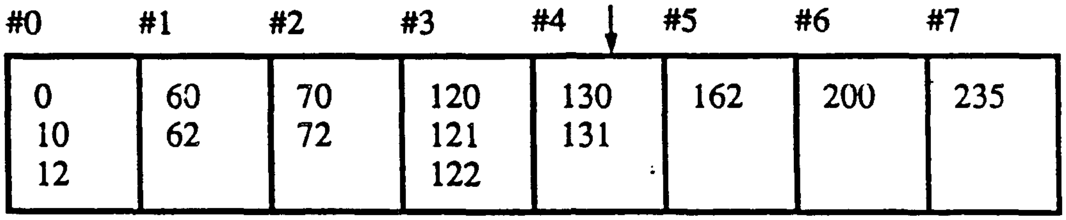


Figure 12.5.12 DRSAM0 example at the end of the expansion cycle

(-,-)	(1,2)	(2,2)	(3,2)	(0.3)	(1,3)

Figure 12.5.13 Type 1 storage area

(1,1)	(0,2)	(1,2)	(-,-)	(-,-)

Figure 12.5.14 Type 2 storage area

where $LN(x,i)$ denotes logical bucket x at level i in the trie, and $0 \leq x < 2^i$. For partial expansions, this ordering is extended to cover elastic logical buckets.

Stornge Areas

In Section 12.5.2.3, it was noted that a file expands by splitting a logical bucket "x" at level "i" and relocating its contents using $OPH_{(i+1)}(Key)$ onto logical buckets "2x" and "2x+1". A contraction is the reverse operation and would merge two buckets from level (i+1) onto their parent bucket at level "i".

Define two storage areas: the type 1 area (T1), generally used for the expansion of the file and the type 2 area (T2) used concurrently with the contraction of the file. A T1 storage space can grow or shrink from its right boundary with its left boundary allowed to return storage to the operating system secondary storage pool while the T2 area operates in the reverse mode. The T1 area is illustrated in Figure 12.5.13. It shows the leftmost bucket expanding onto the rightmost (0,3) and (1,3) buckets. Figure 12.5.14 illustrates a T2 area after contracting the rightmost 2 buckets onto the leftmost bucket (1,1). File movement in both directions is not permitted and the one dimension expansion/contraction of a storage area is preserved.

A storage area is always scanned from left to right on secondary storage. The left boundary in each area is the lowest logical bucket stored in the area and the physical addresses within a T1 (T2) area are defined with respect to its left (right) boundary. This will enable us read the home storage area in key-order through a sequential scan with minimum disk head movement.

The Mechanism of DRSAM1

At a given time, the primary storage space of a DRSAM file can consist of a T1 and a T2 area: the T1 storage area can consist of 2 subareas, the home (HL) and the expansion (E) areas while the T2 storage area can consist of the home (HR) and contraction (C) areas. In Figure 12.5.13 the HL area consists of buckets (1,2), (2,2) and (3,2) while (E) consists of buckets (0,3) and (1,3) and in Figure 12.5.14 the HR area consists of buckets (0,2) and (1,2) while (C) consists of bucket (1,1). Though not required, the subareas are assumed to be contiguously located on secondary storage.

As with LH attach an expansion pointer (expt) with a DRSAM1 file which points to the logical address of the next bucket to expand. If the file expands (contracts), expt is *logically incremented* (decremented) by 1. A *skew counter* (sc) is used to determine the logical boundary between the HL and HR areas. The expansion/contraction rules are designed to guarantee that the (E) and (C) areas will not coexist. The rules are defined as:

Expansion

```
if "C exists"
  then  ($e_1$) "expand from C to HL"
  else if "HR exists"
             then ($e_2$) "expand HR to E"
             else ($e_3$) "expand HL to E"
           endif
endif
```

Contraction

```
if "E exists"
  then ($c_1$) "contract from E to HR"
  else
    if "HL exists"
      then ($c_2$) "contract HL to C"
      else ($c_3$) "contract HR to C"
    endif
endif
```

For expansions, e_1 is used first until all the buckets in the C area are exhausted, then e_2 until HR is fully expanded and finally HL buckets are sequentially expanded. The same scheme holds for file contraction. Within a storage area, "expt" is sequentially increased. It is easy to show that the "lower" logical buckets are in the C area, followed by the ones in the HR, then the HL and finally E areas. For a sequential scan of the file, if C exists then scan HR first, followed by HL and then C; else scan E followed by HR then HL.

Consider the DRSAM file of Figure 12.5.15 with only a home area HL (sc=0) composed of 8 buckets with the expansion pointer pointing to logical node 0 (fl=3 and expt=0). Assuming 8 bit keys, the different bucket hash ranges are included to show the key sequential order of the file (values in []). Through the addition of records, assume that a split condition occurs with the file. Rule e_3 is used and the sequence of linear hashing is followed. The leftmost bucket HL(0,3) splits and its contents relocated the split hash function as the first two buckets E(0,4) and E(1,4) of a newly created T1 expansion area (E). Bucket HL(0,3) is freed as shown in Figure 12.5.16.a and "expt" and "sc" are incremented by 1.

A 2nd expansion splits HL(1,3) by relocating its contents onto E(2,4) and E(3,4). The empty bucket is returned to the storage pool, "expt" and "sc" are incremented and the file status is shown in Figure 12.5.16.b. Continuing with the expansion process, the file will be at level 4 after 8 expansions with the (E) area relabeled as (HL).

HL

(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
[0-31]	[32-63]	[64-95]	[96-127]	[128-159]	[160-191]	[224-255]	[192-224]

Figure 12.5.15 DRSAM1 file with only the HL storage area

HL

(-,-)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
	[32-63]	[64-95]	[96-127]	[128-159]	[160-191]	[224-255]	[192-224]

E

(0,4)	(1,4)
[0-15]	[16-31]

Figure 12.5.16.a. DRSAM1 example after
one expansion from Figure 9.5.15

HL

(-,-)	(-,-)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
		[64-95]	[96-127]	[128-159]	[160-191]	[224-255]	[192-224]

E

(0,4)	(1,4)	(2,4)	(3,4)
[0-15]	[16-31]	[32-47]	[48-63]

Figure 12.5.16.b DRSAM1 example after
two expansions from Figure 9.5.15.

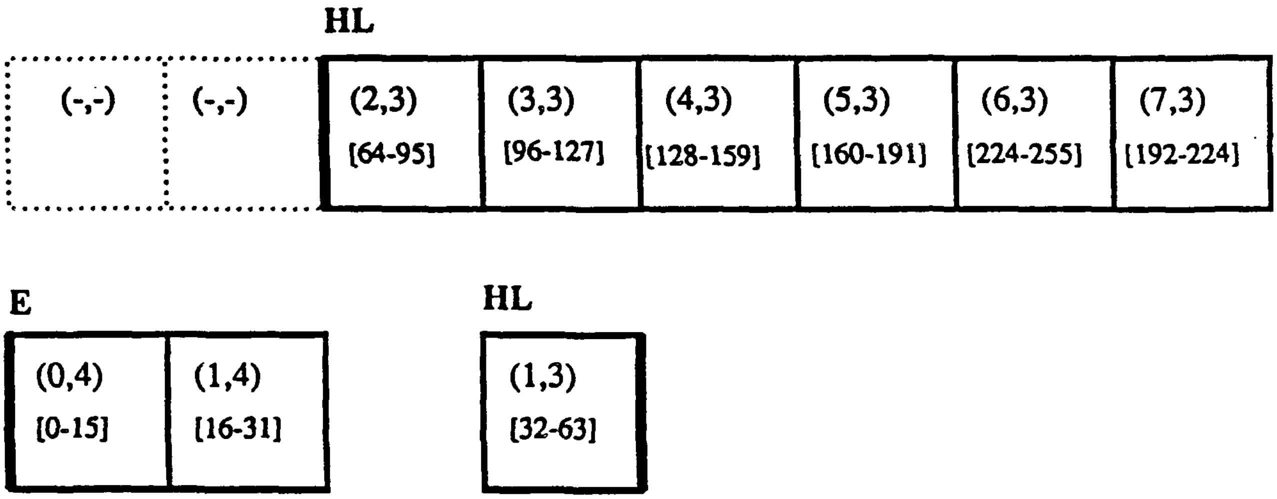


Figure 12.5.17.a DRSAM1 example after one merge from Figure 9.5.16.b

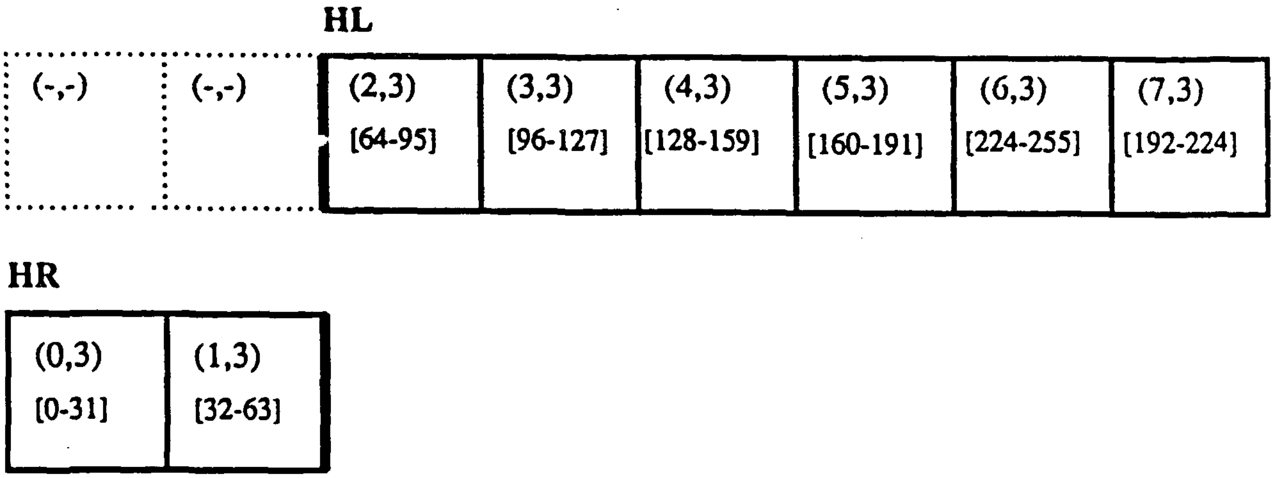


Figure 12.5.17.b DRSAM1 example after two merges from Figure 9.5.16.b

Referring to Figure 12.5.16.b assume that due to successive deletions, a merge is necessary. Referring to Figure 12.5.17.a buckets E(2,4) and E(3,4) are merged back together and relocated onto HR(1,3) in a newly created T2 home area (HR). Pointer (expt) is decremented by 1 while "sc" stays the same; "sc" actually points to the minimum logical bucket address in the HL area and thus (sc-1) points to the maximum logical bucket address of the HR area. Rule c_1 was used here.

A second merge relocates the contents of buckets E(0,4) and E(1,4) onto HR(0,3) as shown in Figure 12.5.17.b. The file is at level 3 and at the beginning of the expansion cycle (expt=0). The existence and size of HR is determined by the skew pointer (sc=2). This file is logically equivalent to the one in Figure 12.5.15 though physically different (different "sc" values). With the knowledge of "expt, "sc", the home level fl, and the existence or non existence of the areas C and E, one can uniquely determine the physical and logical status of the file.

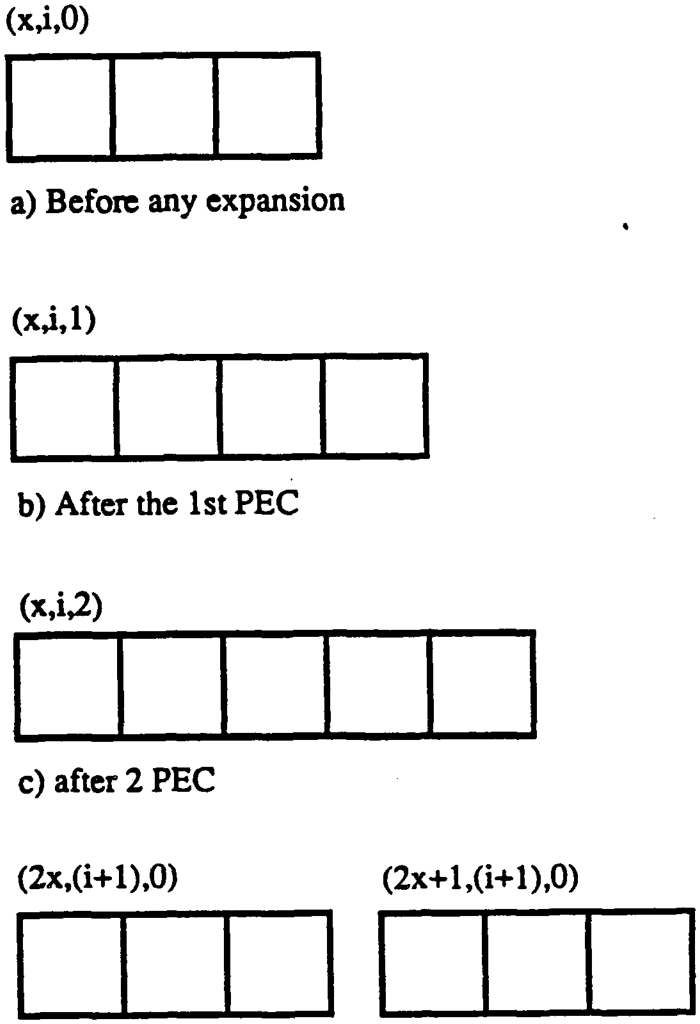
Further file contraction sets the home level of the file to 2 and would merge the HL buckets in sequence from right to left onto the contraction area (C) at the left of the HR area (use rule c_2). This is followed by the HR buckets (from right to left using rule c_3). If all buckets from HR contract onto C, a contraction cycle is completed and C is relabeled as HR. From Figure 12.5.17.b if subsequent splits are accomodated, rule e_2 is used and the leftmost bucket of HR is expanded first; which corresponds to logical bucket 0. It is observed that all possible cases require the simultaneous existence of at most 3 out of the 4 storage subarea types and the situation where C and E exist together. This complies with the expansion/contraction rules. In contrast with DRSAM0, the algorithms that govern DRSAM1 are fairly straightforward and easy to implement. Details are found in [HAC88b].

12.5.2.6. The General Case with Partial Expansions

In this section, DRSAM is generalized with partial expansions (PE) using elastic buckets [LOM87]. This concept was first introduced by Lomet [LOM87] within the context of indexed file organizations and further applied to B^+ trees by Baeza-Yates et al. [BAE87].

Over a *partial expansion cycle* (PEC) each bucket is expanded elastically by a *partial bucket* and is relocated to a new physical address. During the last partial expansion, the bucket at level fl is split (and its contents rehashed with the split hash function) onto two new buckets of minimal capacity at level fl+1. The process of doubling the number of logical buckets is referred to as *full expansion cycle* (FEC) and consists of r PEC.

An example of an elastic home bucket is illustrated in Figure 12.5.18. It shows the different partial expansion steps of a logical node "x" at level "i" for the case when r=3. Initially, in Figure 12.5.18.a the node consists of 3 partial physical buckets, and during the 1st and 2nd partial expansion, the node elastically expands by adjoining to it one partial physical



a) Before any expansion

b) After the 1st PEC

c) after 2 PEC

d) after the last PEC splitting takes place

Figure 12.5.18 An elastic logical node

bucket. This is shown in Figures 12.5.18.b and 12.5.18.c. For the last partial expansion, adding a new partial physical bucket, doubles the capacity of the node with respect to its initial state of Figure 12.5.18.a. In this case, the bucket forks into two new child buckets "2x" and "2x+1" at level "i+1" and with the minimum capacity of 3 partial buckets.

From the expansion mechanisms of DRSAM files, it is clear that elastic buckets are a natural extension to this file structure with partial expansions. DRSAM0 expands logical buckets on contiguous, physical locations that are suited for elastic buckets. It is observed that Larson's approach [LAR80b] could be adapted without rehashing to DRSAM0 as well, but elastic buckets are preferred as they inherently do not need rehashing during a partial expansion cycle (PEC) and lead to bucket sizes that grow in small steps with a lower level of granularity.

An elastic logical node (ELN) is defined as a logical bucket with the capacity of a number of partial physical buckets. With each ELN associate the triplet $ELN(lba, lev, exp)$, where lba refers to the logical bucket address, lev the level and exp the PEC cycle that has been completed by the ELN under consideration. This triplet determines its physical status with respect to the global status of the file denoted by $F(NL(fl), cexp)$, NL as the number of logical buckets, fl the current file level, and $cexp$ the urrent expansion cycle under consideration. Obviously: $0 \le lba < 2^{fl}$ and $0 \le exp < r$. The capacity of an ELN which has undergone i partial expansions is $B_i = (r + i) \times pb$, where pb is the capacity of a partial physical bucket. The minimum ($B_{\min}$) and maximum ($B_{\max}$) capacities are: $B_0 = B_{\min} = r \times pb$ and $B_{r-1} = B_{\max} = (2r - 1) \times pb$.

For DRSAM0, and during the cexp-th partial expansion, the physical address (pad) of a logical bucket (lba) is determined by:

$$pad(lba, exp) = lba \times r \times \prod_{j=1}^{exp} \frac{r+j}{r+j-1} = lba \times (r + exp)$$

where $exp = cexp \bmod r$ if logical bucket lba has expanded else $exp = cexp - 1$. For the r-th PEC the logical bucket of maximum size ($B_{\max}$) at level fl splits onto two new logical buckets of minimal size ($B_{\min}$) at level $fl+1$.

Like for DRSAM0, elastic buckets are the natural extension to DRSAM1 file structures with partial expansions. The process is similar to the one for DRSAM0 at the elastic node level: over a PEC each bucket is elastically expanded by a partial bucket and is relocated to a new physical address. This address is within the expansion area (E) if it exists or in the left justified home storage area (HL) area if "C" exists. During the last partial expansion, the bucket at level fl is split (and its contents rehashed with the split hash function) onto two new buckets of minimal capacity at level $fl+1$. The global system model of elastic logical nodes is the same as the one defined for DRSAM0. The main difference lies in the expansion sequencing of the ELNs. The full ordering operation, previously defined on the logical trie S, is extended to cover ELNs. The ordering should properly define the chain of buckets in logical ascending order (for file expansion) or descending order (for file contraction). The full ordering operation on S is extended to elastic logical nodes with the following properties:

$$ELN(x_1,i,p_1) < ELN(x_2,i',p_2) \equiv i < i'$$

$$ELN(x_1,i,p) < ELN(x_2,i,p') \equiv p < p'$$

$$ELN(x,i,p) < ELN(x',i,p') \equiv x < x'$$

Then the infinite chain which results is:
<ELN(0,0,0),ELN(0,0,1)...ELN(0,0,r-1),ELN(0,1,0),ELN(0,1,1)...

ELN(0,i,0),...ELN(x,i,0),ELN(x+1,i,0),....ELN(2^i −1,i,0),

ELN(0,i,1),...ELN(x,i,1),ELN(x+1,i,1),....ELN(2^i −1,i,1),...

ELN(0,i,r-1),...ELN(x,i,r-1),ELN(x+1,i,r-1),....ELN(2^i −1,i,r-1),.......>

Denote by $pad(0)$ to be the physical address of the node $ELN(0,i,p)$ with lowest logical address within a storage area. During the cexp-th partial expansion, the physical address of a node with logical address lba would be offset with respect to $pad(0)$ by a number of partial physical buckets. This offset is computed using the equation derived for DRSAM0 and is given by:

$$offset(lba,exp) = lba \times (r+exp)$$

Where $exp = cexp \bmod r$ if logical bucket lba has expanded else $exp = cexp-1$. For DRSAM1, the area is one of HL, HR, C or E. For example, if the expansion area is considered then $pad(0)$= E. Considering the contraction area, the address "C" points to the right boundary which corresponds to $ELN(2^{fl}-1,fl,cexp)$. Then the physical address ($pad(0)$) of $ELN(0,fl,cexp)$ is given by:

$$pad(0) = \mathrm{C}-offset(2^{fl}-1,cexp).$$

The details of the algorithms for DRSAM files are found in [HAC88b].

12.5.2.7. Overflow Management of DRSAM Files

In this section a new overflow management scheme is described based on elastic buckets. The assumption of contiguous allocation for file expansion can be easily extended to deal with overflow chains. This concept is similar although not equivalent to the one of elastic buckets which was discussed in the previous section.

The technique is referred to as as *elastic overflow chains* and is a generalization of the usual overflow chaining used for collision resolution. The method assumes an overflow storage pool with capacities b_0 to $b_{el-1}=b_{\max} = el \times b_0$ in incremental capacities of b_0 records ($el \geq 1$ is referred to as the elasticity of the overflow storage pool). When an overflow bucket of capacity $b_{i-1} = b_0 \times i$ becomes full (i.e. overflows) its contents, with the inserted record, are written onto an overflow bucket of capacity $b_i = b_0 \times (i+1)$. This process is continued until a capacity of $b_{\max}$ is reached whereby a new bucket of capacity b_0 is attached to the chain thus increasing its length by one. An elastic overflow bucket for $el=3$ is illustrated in Figure 12.5.19 It is clear that the usual overflow chaining method corresponds to the case where $el=1$.

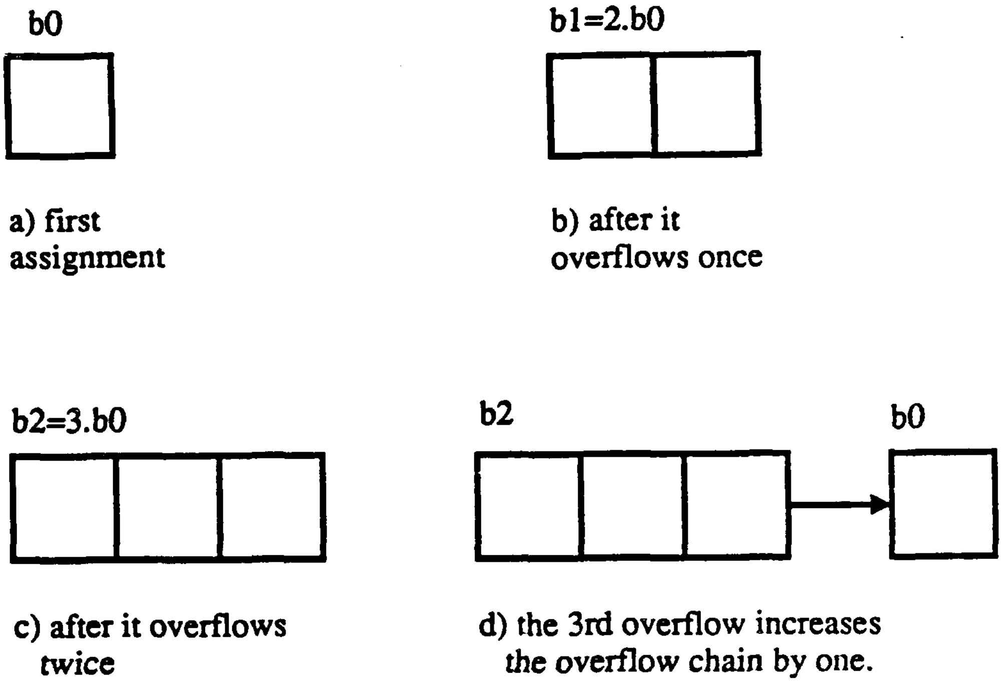
Elastic overflow chaining has definite advantages over the usual overflow chaining method. Qualitatively, as el increases the overflow chain length decreases which results in an improvement of the average as well as worse case successful and unsuccessful search costs. The effect of elastic overflow chaining will be quantified in the next section.

12.5.3. Performance Evaluation of DRSAM file Structures

In this section we discuss the results of a performance evaluation of DRSAM with elastic buckets. We target the application of DRSAM to secondary key retrieval, like transformed inverted lists [BER87, HAC88a]. Thus we study the effect of ordered insertions on the performance characteristics of DRSAM. This is followed by a performance comparison of LPE and EB for partial expansions. Finally, sequential and range query processing with DRSAM files are discussed and compared to similar OPH methods.

12.5.3.1. System Model

We developed an analytical model for very large DRSAM files and performed extensive trace driven simulations of the file structure. The results from the analytical model closely match those from the simulations. As we are interested in secondary storage systems, the



a) first assignment

b) after it overflows once

c) after it overflows twice

d) the 3rd overflow increases the overflow chain by one.

Figure 12.5.19 Elastic overflow node with el= 3

performance measures are in terms of disk access cost; where disk seek time is the predominant component in the cost measure. The details of the performance analysis can be found in [HAC88b].

The model assumes a uniform distribution over the hashed domain. Furthermore, main memory buffer storage is large enough to hold an accessed bucket with its overflow chain. For a random insertion one needs to perform at least one disk access to read the home bucket and subsequent reads to traverse its potential overflow chain. It is followed by a disk access to write back the bucket after inserting the new record at the end of the chain. With ordered insertions, the record may be inserted at any location in the chain with equal probability. This implies that, in general, one needs to write back more than one bucket. Elastic overflow chains are used and the cost to expand or create a new chain is always equal to two disk accesses (operating systems overhead not included).

With lfc as the number of record insertions between file expansions, the model follows the load factor control mechanism described in [RAM82]. The file is expanded by the addition of one partial bucket after every lfc inserted records. The load factor (lf) is almost constant and is computed as $\frac{lfc \times r}{hb}$ *; with hb as the minimum home bucket capacity. For an expansion operation we need at least one disk read to retrieve the bucket undergoing the expansion. This is followed by a disk access to write the generated blocks into consecutive locations on secondary storage. Obviously, the existence of overflow buckets increases the number of disk accesses accordingly. The performance parameters which were determined are the average of the storage utilization factor (auf), the average costs for a successful search ($acss$), unsuccessful search ($acus$), random insertion ($acri$) and ordered insertion ($acoi$).

12.5.3.2. Results and Discussion

The results which we report are taken from the analytical model for a home bucket capacity of hb=48 records [HAC88b].

Effect of Elastic Buckets

We illustrate the effect of the number of partial expansions (r) on storage utilization in Figure 12.5.20. With a fixed average load factor (lf = 1.25), the utilization factor increases with increasing r from an average of 0.91 at r=1 to 0.94 at r=3. To evaluate the retrieval performance, we chose the basic overflow bucket capacity (ob_0) such that the average storage utilization factor is around auf=0.91 for the different values of r. Then we plotted the successful and unsuccessful search costs in Figure 12.5.21 and 12.5.22 respectively. It is clear

* The load factor is defined as: $lf = \frac{\textit{\# of records inserted}}{\textit{home storage space in records}}$.

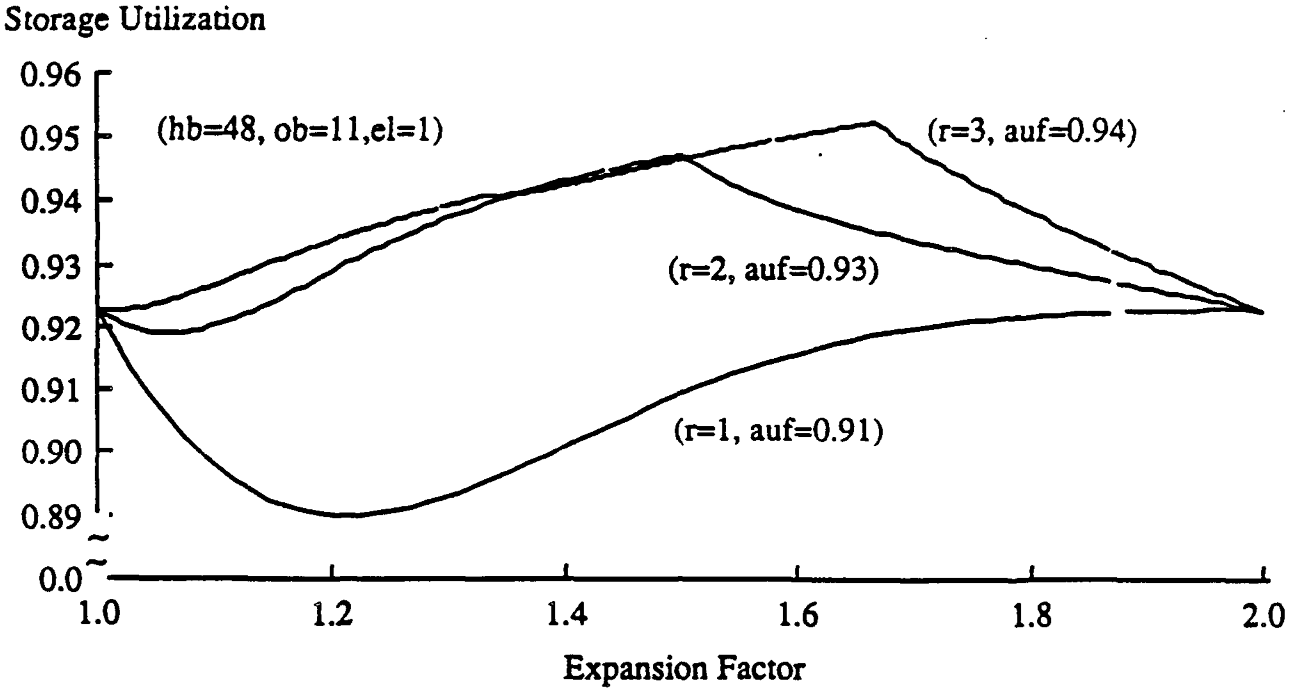


Figure 12.5.20 DRSAM storage utilization (r=1 to 3)

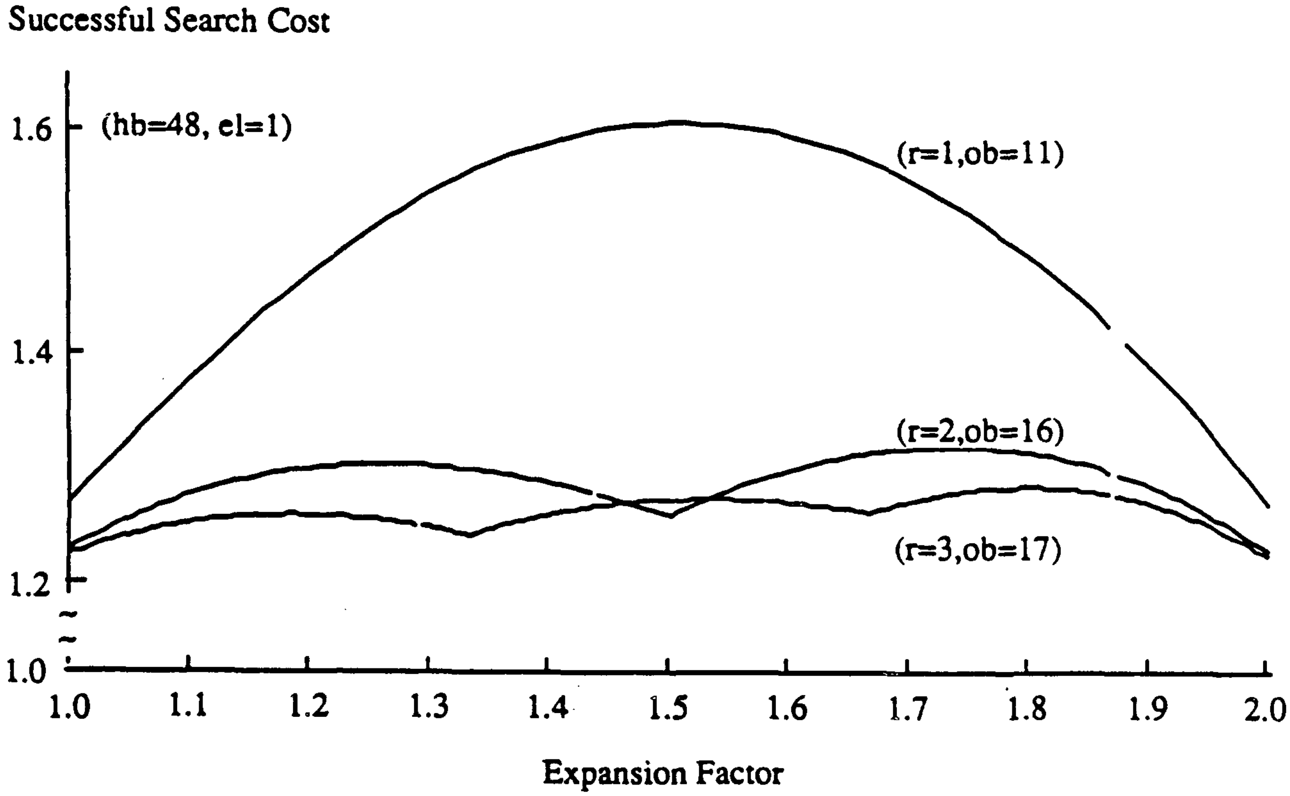


Figure 12.5.21 DRSAM successful search cost (auf = 0.91)

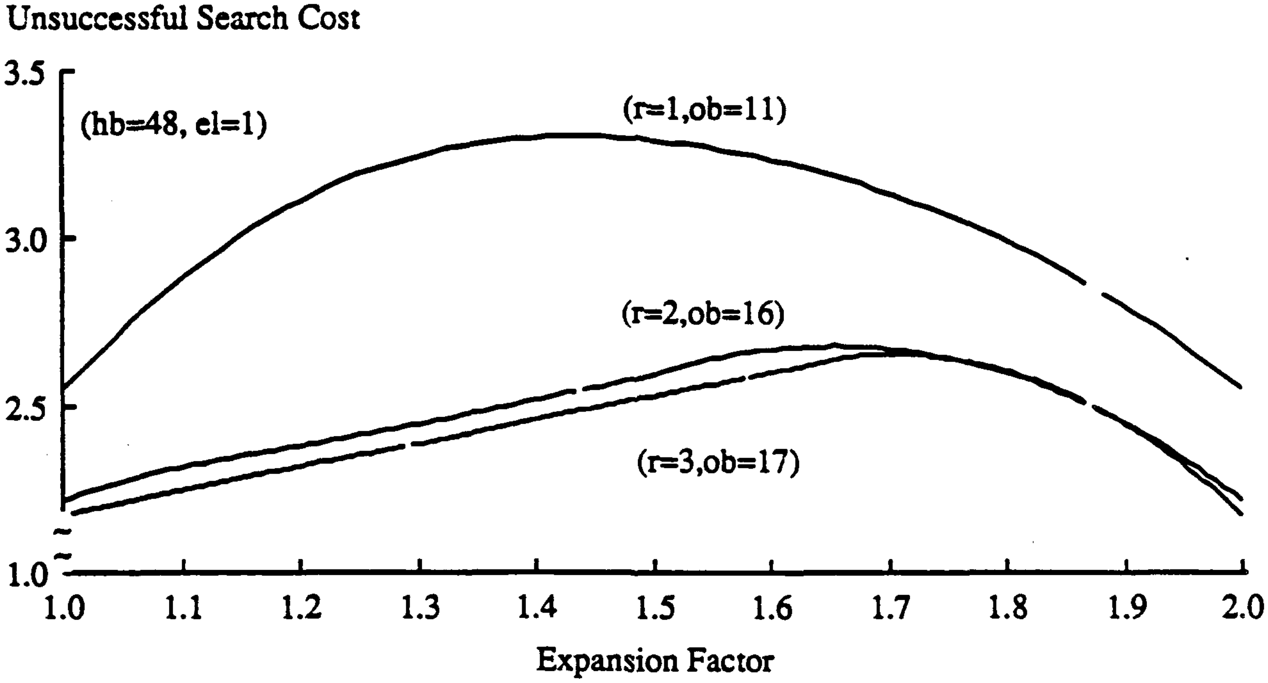


Figure 12.5.22 DRSAM unsuccessful search cost (auf = 0.91)

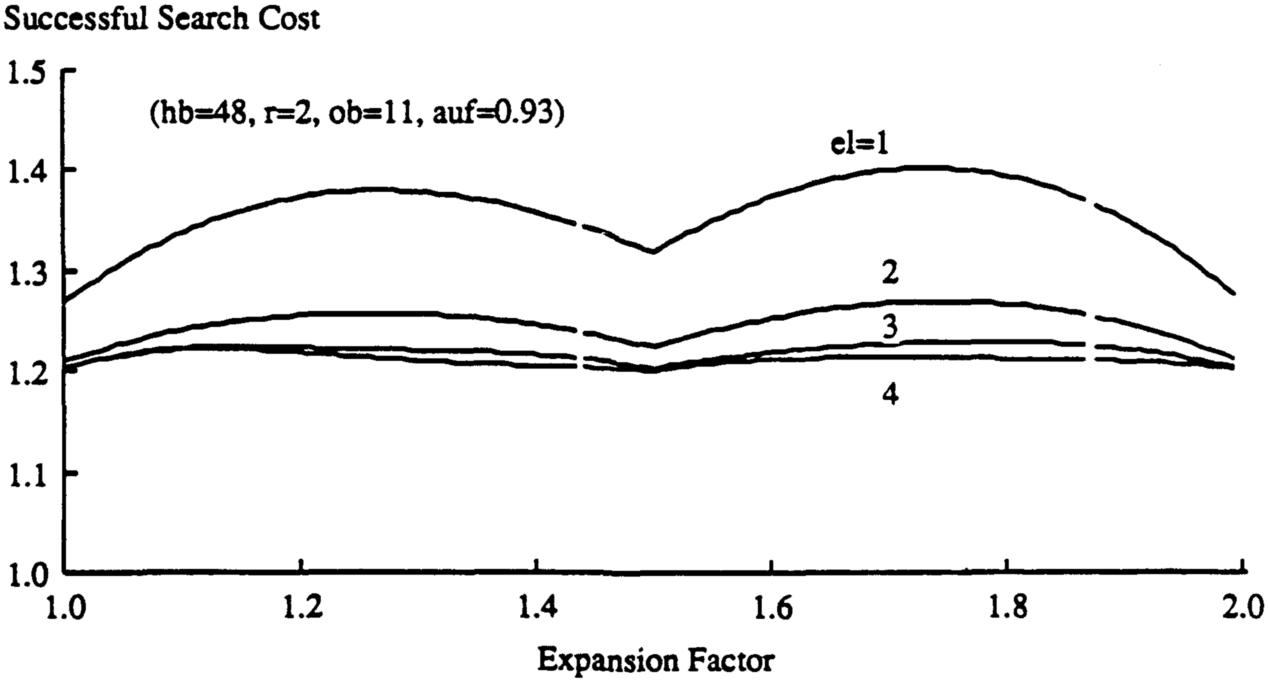


Figure 12.5.23 DRSAM successful search cost (el= 1 to 4) that elastic buckets improve the random access performance (*acss* and *acus*) of DRSAM.

A higher elasiticity for the overflow buckets does not affect the storage utilization of the file, but it results in a decrease of the overflow chain length. This tends to improve all performance parameters as reported in [HAC88b]. Figure 12.5.23 illustrates the effect of increasing the elasticity *el* of the overflow chains on successful search cost. For a sustained average storage utilization of 93%, the average successful search cost improves with an increase in the elasticity of the overflow chains. The variations of the performance curves over a FEC decrease as well.

With a storage utiization factor around 94%, *hb*=48, *r*=3 and *el*=4, our results show that the performance of DRSAM is excellent with *acss* ~ 1.20, *acus*<2.00 and *acri*<3.25. Considering that the performance of file organizations degrade very fast with increasing utilization factors makes the elastic buckets techniques an important extension to DRSAM file structures and to non indexed dynamic hashing schemes in general.

Effect of Insertion Methods on the Unsuccessful Search

We chose the case where *hb*=48 records and *r*=3 to compare the random and ordered insertion methods. In Table 12.5.1 we report the average insertion and average unsuccessful search costs for *el*=1 and *el*=4. With *el*=4 ordered insertions result in *acus* = 1.196 disk access; while it is equal to 1.934 if random insertions are used. This is an improvement in performance of 38%. On the other hand, the average insertion cost increases by 20%. The cumulative effect of ordered insertions and elastic overflow buckets is to improve unsuccessful search cost from 2.966 to 1.196 (a factor of 2.48).

12.5.3.3. Elastic Buckets Versus Larson's Partial Expansion

To compare Larson's partial expansion [LAR82b] (LPE) with elastic buckets (EB) we used DRSAM as the basic file structure. The test case was for *hb*=48 records, *r*=2, *el*=1 and ob_0=11 records; *lfc* was adjusted to result in the same average storage utilization for both methods (*auf* ~0.93). Table 12.5.2 shows the averages for LPE and EB.

From the results in Table 12.5.2, it is clear that elastic buckets outperform Larson's partial expansions scheme. However, the cost paid is a wider fluctuation in storage utilization factor. This is clear from Figure 12.5.24 where the *auf* for EB has a peak fluctuation of ~ 1.5% around the average while the *auf* for LPE is almost constant. Figure 12.5.25 compares the successful search cost for EB and LPE. LPE required a higher average load factor in order to achieve an equivalent average storage utilization to EB. This causes lengthier overflow chains to exist and explains the poorer retrieval performance.

It is noted that the performance measure is in terms of random disk access cost and does not account for differences in data transfer time. For EB the transfer time is higher due to the

hb=48, ob$_0$=11, r=3, *lfc*=20, *auf* =0.94				
insertion	*el*=1		*el*=4	
method	acus	insertion cost	acus	insertion cost
random	2.966	4.372	1.934	3.227
ordered	1.312	6.006	1.196	3.883

Table 12.5.1 Effect of the insertion method on the cost of an insertion and unsuccessful search

hb=48, ob$_0$=11, r=2, el=1				
	auf	*acss*	*acus*	*acri*
LPE	0.932	1.567	3.496	4.820
EB	0.933	1.359	2.949	4.253

Table 12.5.2 Comparison of LPE and EB for partial expansions

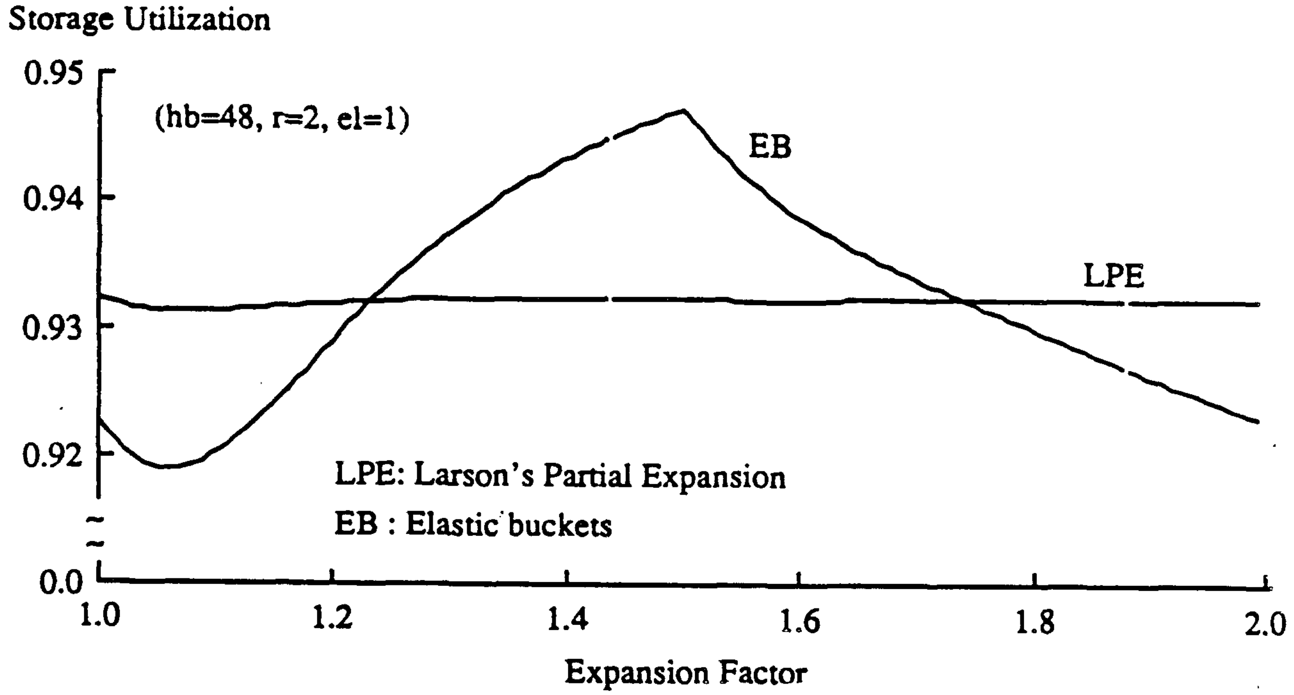


Figure 12.5.24 LPE versus EB: storage utilization (auf=0.93)

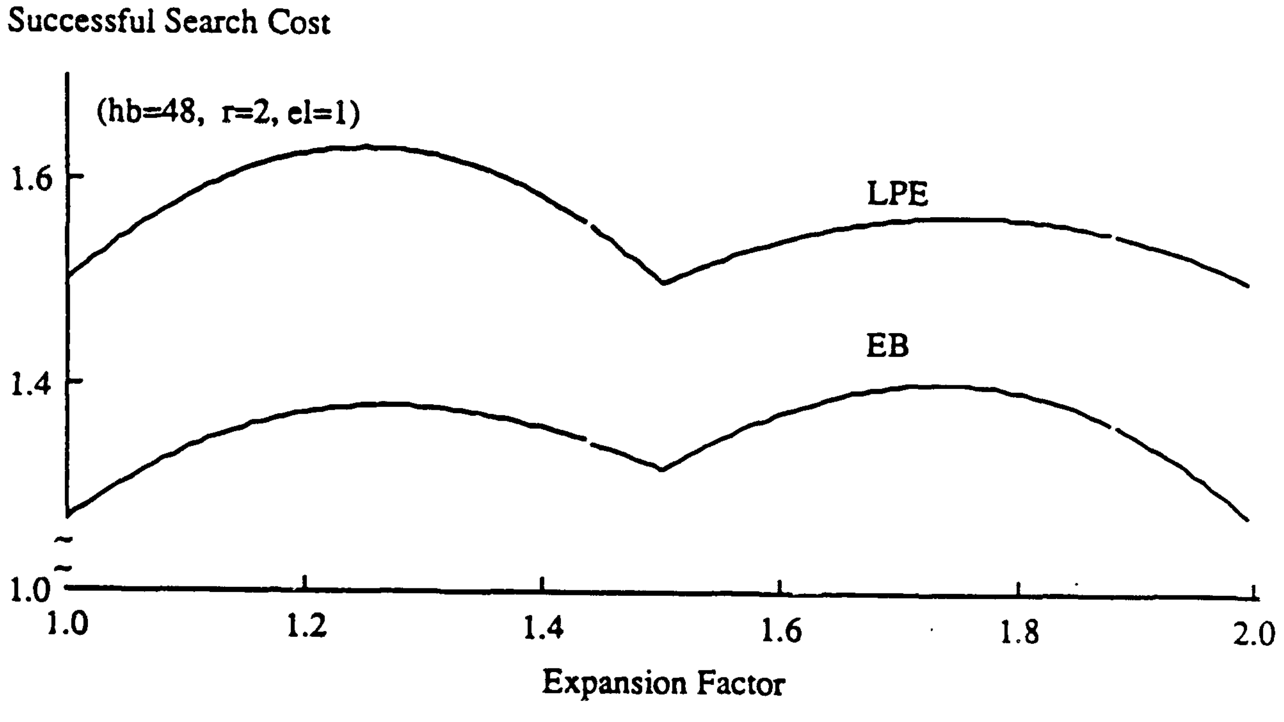


Figure 12.5.25 LPE versus EB: successful search (auf=0.93)

use of variable bucket capacities. But the difference in performance which is observed in Figure 12.5.25 and Table 12.5.2 is large and the unaccounted transfer time becomes irrelevant.

12.5.3.4. Sequential and Range Processing with DRSAM

In this section the efficiency of DRSAM files in handling range queries and sequential processing is discussed and compared with other relevant order preserving linear hashing techniques. The analysis covers home (primary) buckets and does not account for the overflow chains which have to be scanned as well.

DRSAM Variant 0

In [HAC88b], it is shown that if enough disk buffer space is available, a range query requires at most 2 disk seeks to the primary storage area of a DRSAM0 file. These accesses are followed by consecutive block transfers from disk. For the case where $r=1$, a range query which overlaps two or more physical buckets typically requires one disk seek and a complete sequential scan of the N primary buckets theoretically requires at most $2 \times fl = 2 \times \log_2 N$ seeks. Then for $r=1$, the sequential processing of a DRSAM file's primary buckets requires an $O(\log_2 N)$ disk seeks. For the general case ($r \geq 1$), and for the PEC i under consideration, the sequential scan for the home buckets of the file requires $O(\log_\alpha N)$ disk accesses; where $\alpha = \frac{r+i}{r+i-1}$.

DRSAM Variant 1

For the sequential scan of a DRSAM1 file, one traverses the storage areas in sequence: the E and C areas cannot coexist simultaneously. Then the storage areas which are traversed are either E then HR followed by HL, or HR then HL followed by C. Thus, if a range query is specified the number of storage area boundaries to be traversed is at most three. Therefore, the home storage area of a DRSAM1 file can be theoretically scanned in O(1) disk seeks. This is compared to O(N) for an order preserving hashing scheme based on linear or extendible hashing [FAG79] , tries and B-trees [BAY72, 77] ; and the $O(\log_2 N)$ of DRSAM0.

Comparison with Other Techniques

With DRSAM files, if enough buffer space is available, a range query which overlaps two or more primary buckets would generally require one disk seek (excluding overflow access). Consider now other proposed order preserving linear hashing schemes [ORE83, BUR83]) or order preserving extendible hashing [TAM81]. With these methods a range query spanning two buckets will require two disk seeks to the primary storage area. Furthermore, these structures cannot make use of the availability of a large buffer space. As the number of buckets to be retrieved increases the improvement in range query performance increases with the use of DRSAM file structures. This improvement is based on the locality of data on consecutive home buckets and results from the sequential allocation property.

Nevertheless, the existence of the overflow area means that the complete sequential scan of a DRSAM file would still incur O(N) disk seeks. But, because of the consecutive allocation property, DRSAM is shown to outperforms previous methods. Furthermore, the use of elastic overflow bucket chains follows the same concept and implies further improvement of the performance of DRSAM files for range queries.

While achieving absolute near optimality is still an open problem, a promising approach is to apply the concept of repeated hashing [LAR80a] or similarly recursive hashing [RAM84] to DRSAM files (specifically DRSAM1). Recursive hashing uses the same technique for the management of the overflowing records repeatedly and Ramamohanarao showed its efficiency as applied to linear hashing. Moreover, recursive hashing can be transformed into an iterative scheme suitable for files that are stored and accessed on multiple storage units. With DRSAM files recursive hashing would lead to file structures that are near optimal for both sequential processing and random access. Another possibility is to adapt overflow management schemes that use the primary storage area of the file while keeping the sequential allocation property of DRSAM.

If good solutions for randomizing OPH functions are made available, range queries and sequential processing performance using DRSAM are expected to be comparable to indexed sequential files and better than B-tree based structures. This subject is part of the next section.

12.5.4. Inverted Surrogate Files with DRSAM

In this section, the DRSAM technique is extended to inverted surrogate files with the Inverted Dynamic Surrogate File (IDSF). Section 12.5.4.1 defines access keys in database systems. In Section 12.5.4.2 the problem of non uniform distributions is qualitatively discussed. This is followed by extended DRSAM (EDRSAM) in Section 12.5.4.3. The analysis covers the index table and local control mechanisms to the DRSAM storage areas of Figure 12.5.4. EDRSAM is proposed to be the core file structure for physical models targeted to replace the usual inverted lists [DAT86, SAL83]. In Section 12.5.4.4, EDRSAM is applied to ISF and results in the inverted dynamic surrogate files model (IDSF). The IDSF model extends TIL to cover very large *dynamic* files. Finally, the storage overhead as well as the query response time with IDSF are discussed.

12.5.4.1. Access Keys to Relational Tables

The concepts of primary and secondary keys in a data/knowledge base relation should be clearly distinguished. In the remainder of this section, discussions are based on the following concepts.

Notation: In a VLDKB considering the set of relations R, the *set of attributes* of a relation $r \in R$ is denoted by A_r. The domain of $ar \in Ar$ is $D(ar)$. For a relational table the number of tuples in the relation is denoted by N_r.

Definitions: An *access key* or simply *key* (μ) to a relation is a function on a subset $I \subseteq Ar$: $\mu = f(ar \mid ar \in I \subseteq Ar)$. The domain of a key μ is $KS(\mu) = \underset{ar \in I}{\times} D(ar)$ (where $\times$ is the cartesian product). The set of distinct values for a key μ is denoted by $ks \subseteq KS$ with cardinality $|ks|$. $C(k,\mu)$ is the value distribution factor for $k \in KS(\mu)$ and refers to the number of occurrences of k in the N_r tuples. The selectivity of k is denoted by $\alpha(k,\mu)$ and is written as:

$$\alpha(k,\mu) = \frac{N_r}{C(k,\mu)}$$

Clearly, $C(k,\mu) \le N_r$ and $\alpha(k,\mu) \ge 1$. $\alpha(k,\mu)$ approximates the probability of occurrence for a key value k ($P_Z(Z=k)$). The average of $C(k,\mu)$ is:

$$Cav(\mu) = \sum_{k \in KS(\mu)} \frac{C(k,\mu)}{|ks|}$$

and $\alpha(\mu) = \dfrac{N_r}{Cav(\mu)}$ is the average selectivity of μ. Obviously $\alpha(\mu) = |ks|$ is the number of distinct key occurrences in the N_r tuples. Using the above definitions, *primary* and *secondary* keys are defined as follows:

A *primary key* is a key μ_p such that:

for all $k \in KS(\mu)$, $C(k,\mu) \in \{0,1\}$.

Any other key type is referred to as a *secondary key* μ_s or more formally μ_s is such that:

there is a $k \in KS(\mu)$ such that $C(k,\mu) \in \{2,...,N_r\}$

One also refers to pseudo-primary keys. These are the keys with high selectivities.

In inverted files, or models based on many single attribute single files, each set I is a singleton of A_r and in fully inverted files a partition of A_r is determined by the set of Is.

With these models, $C_i(k)$ is used to denote the value distribution factor for k in attribute i and the average value for an attribute in a relation is denoted by $C(i)$. $\alpha_i(k)$ and $\alpha(i)$ are the corresponding selectivities.

12.5.4.2. Non Uniform Distributions

In this section, the implications of non uniform distributions on file structures performance, especially dynamic hashing methods, are discussed.

One adverse phenomenon common to all order preserving dynamic hashing schemes and therefore expected with DRSAM is the following: due to uneven distributions within a bucket range, splitting a node redistributes the records unequally onto its child buckets. The result is that one of the new buckets will be loaded with most of the keys while the other is sparse. The degenerate case happens when splitting one level only would move all the data into one block instead of dividing it evenly onto the 2 buckets.

The degenerate case is illustrated in Figure 12.5.26. For the home hashing depth of 2 bits the splitting function tries to redistribute the records by dividing them in two groups whose keys differ through the 3rd bit. But in the example of Figure 12.5.26, all the keys in the splitting bucket have the same 3rd bit (from the left) but differ through higher level bits like the 5th and higher. Then the 1st split results into an empty bucket and a full bucket with the possibility that an overflow record is attached to it. This means that if the attribute values distribution is highly non uniform, LH, EH and also DRSAM may result in file structures with long overflow chains and low load factor.

If controlled splitting is used to set the load factor then the degradation in retrieval performance would restrict the application of order preserving dynamic hashing schemes like DRSAM. Even if single file versions [LAR82a, LAR85] are used the number of search probes increases very fast with biased distributions. This behavior is typical of tries and trie hashing schemes [LIT81] and is mentioned in [REG88]. Returning to Figure 12.5.26, one needs to hash up to level 5 to be able to differentiate between the 3 keys in the example. The resulting multi-level trie structure will have empty nodes as illustrated in Figure 12.5.27.

When designing access methods for secondary key retrieval applications, the value distribution factor ($C(k;\mu)$) of a key becomes an important parameter. With bc as the bucket capacity (in records), if $C(k,\mu) > bc$ then bucket $OPH_i(k)$ corresponding to k will automatically have an overflow chain. This can happen even if the average value distribution factor of the file (Cav) is less than bc. The effect is similar with uneven distributions. But in this case, multi-level hashing will not resolve the overflow and the capacity of the bucket should be extended. It implies that the peak value of the value distribution factor of a key $C(k,\mu)$ is restricted to be less than bc. This argument holds for any dynamic hashing technique based on tries. In the case of inverted surrogate lists, this restriction is not overwhelming and would

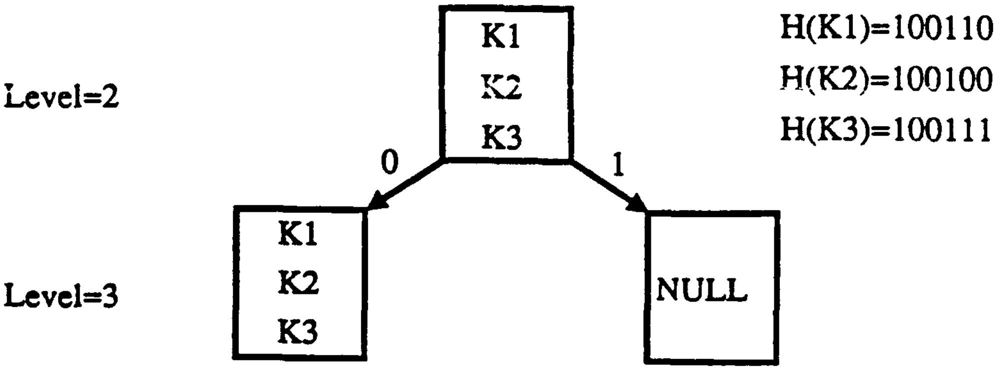


Figure 12.5.26 Degenerate split

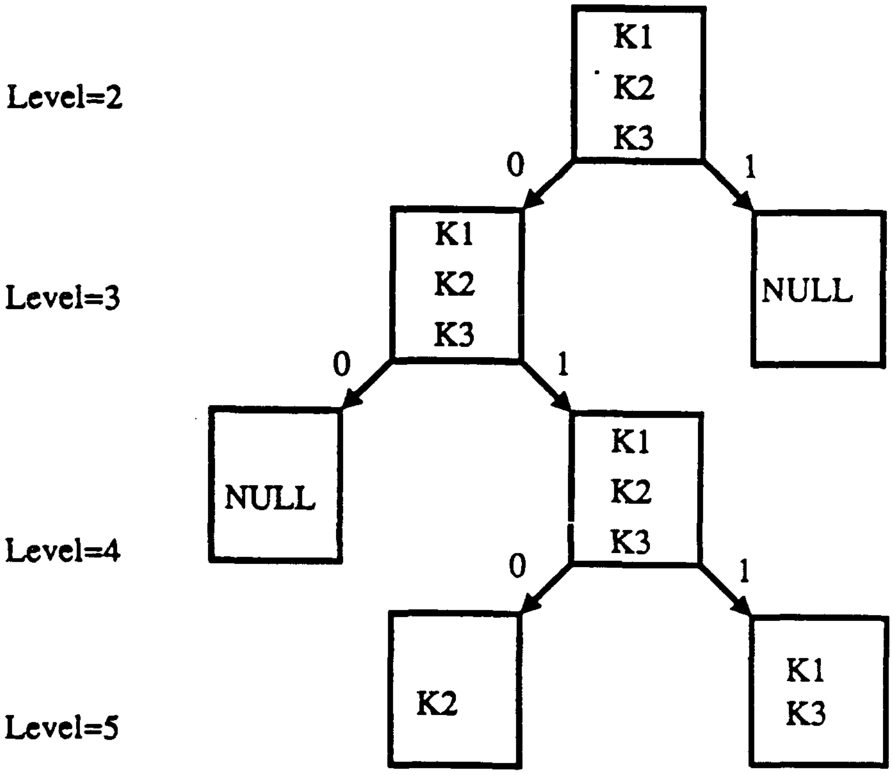


Figure 12.5.27 Multi-level trie hashing

be easily relaxed as the surrogate file records are small in size so that *bc* is expected to be relatively large (typically more than 100 per bucket).

12.5.4.3. Extended DRSAM

Possible solutions for robust OPH file structures were proposed with the "quantile" methods [BUR84, KRI87], the statistic based approach [ROB86] and the indexed bounded disorder method [LIT87]. Also to be mentioned is the linear piecewise method [GAR86] based on distribution dependent hashing. For DRSAM we propose similar control mechanisms to alleviate the problems incurred by non uniform distributions and low selectivities of key values in secondary indices. These mechanisms rely on the sequential allocation property to maintain the overall performance of the file, and the resulting structure is referred to as extended DRSAM (EDRSAM). Two control strategies are discussed: 1) global file control with an index and 2) iocai control with a multi-level trie embedded as a sequentially allocated structure. The dynamic administration of EDRSAM with the proposed strategies is discussed in [HAC88b].

Global Control with an Index

The global control mechanism consists of a two level file structure similar to TIL1 [HAC88a]. As shown in Figure 12.5.4 the first level is a memory resident index table. The table entries are such that the domain of a key is partitioned into intervals with quasi-uniform distributions. The effect of the index table entries is to digitize the probability density function or equivalently create a piecewise linear approximation of the cumulative distribution. Each digital level implies the creation of a DRSAM storage area pointed to by an entry in the index table. This area grows according to the estimated cumulative distribution over the interval. Each storage area is independently managed resulting in a multi-level DRSAM structure. This approach is suitable to a distributed environment where storage is allocated in quanta of contiguous physical blocks.

The use of the index table is essential, as generally good randomizing OPH functions are not available and non-uniform distributions are detrimental for hashed files. We estimated that a table size of 4 Kbytes can accommodate around 200 index entries. A file size of 2^{24} elastic buckets can be achieved. These are more values than one expects to use for very large files. If the table cannot fit in main memory an additional disk access would be required for the index. But this situation is not expected to arise unless highly irregular and biased distributions are considered. Paging algorithms can be used in this case. Details of the index table entries are found in [HAC88b].

Local Control

The DRSAM storage areas in Figure 12.5.4 are said to have a quasi-uniform distribution of keys. As the file dynamically evolves excessive unbalance due to non-uniform hashing will appear and degrade the performance within a storage area of the file. Proper local control for fine tuning is needed to adjust to localized unbalanced hashing. We briefly introduce two such schemes, namely ***sub-hashing*** and ***super-hashing***.

1. Sub-Hashing

The idea of sub-hashing is illustrated in Figure 12.5.28. Assume that two brother buckets at level "i" are such that one is highly loaded and the other is sparse. The highly loaded bucket will have a lengthy overflow chain while the sparse brother will be almost empty. This situation leads to a degradation in the retrieval performance as well as in the storage utilization of the file. We resolve this situation by merging the contents of the two buckets at a sub-level "i-1".

Tag entries are used to determine the relative level with respect to the home level of the storage area. A tag value of 0 refers to the home hash level while a 1 is used for the split hash level. In Figure 12.5.28 the grouped buckets are at sublevel "i-1". This concept follows the same idea introduced by Orenstein for MLOPLH [ORE83]. With *bc* as the bucket capacity, the physical contiguity is used and the resulting group can use a capacity of 2×*bc* records. Overflow is thus handled more efficiently and the problem arising from a low selectivity on some secondary keys is naturally resolved. This characteristic stems out of the sequential allocation property.

2. Super-Hashing

Super-hashing is illustrated in Figure 12.5.29 and is the counterpart of sub-hashing. When all records in a bucket at level "i" have the same (i+1)-th bit, one of the resulting child buckets is empty. we need to hash with a higher level to differentiate the keys. Tag information is also used to identify relative super-hashing depth. In Figure 12.5.29, the split hash level "i+1" is used first and results in the empty bucket on the right. One additional hashing depth is required to differentiate the contents of the left brother bucket. The resulting buckets are at level "i+2" and have a tag value of 2.

Sub and super-hashing can be applied recursively resulting in a embedded multi-level and sequentially allocated trie structure. With a 4 bit tag per bucket, 16 different levels of local control can be accommodated.

Some Performance Implications

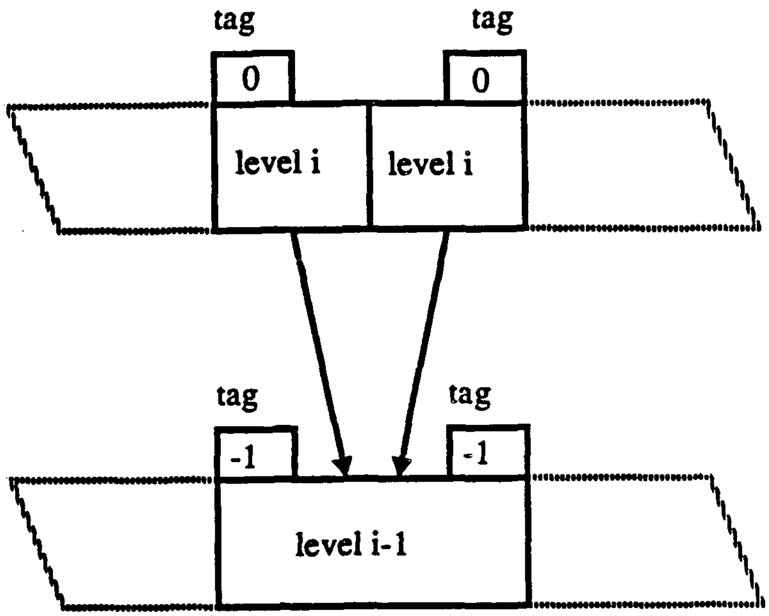


Figure 12.5.28 Two buckets regrouped with sub-hashing

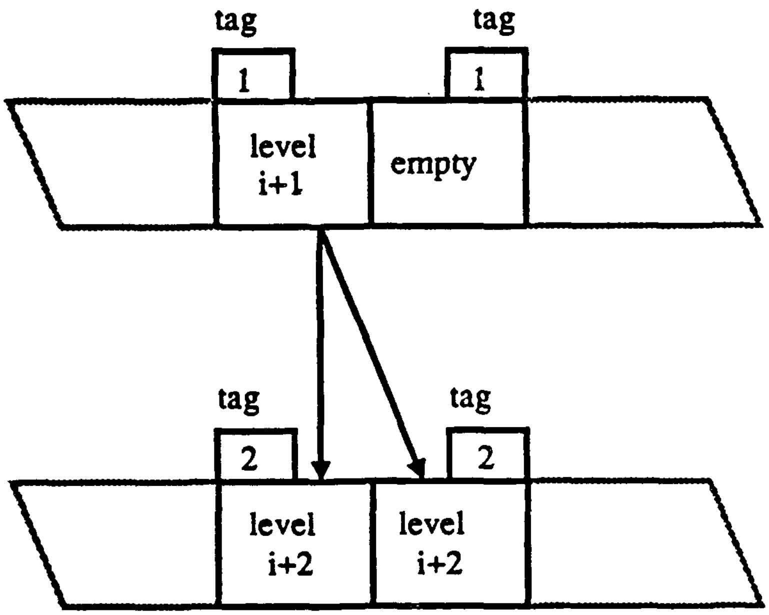


Figure 12.5.29 A bucket uses empty space with super-hashing

For the local control mechanisms, we observed some similarities with Orenstein's MLOPLH [ORE83]. But the two mechanisms we propose, result in a file structure with its own characteristics: the use of tag information induces a maximum of one additional access. Furthermore due to bucket contiguity this access will not need disk head movement. In comparison, Orenstein's method could require a maximum of $\log_2 fl$ disk seeks.

If enough buffering is provided, the random access performance is degraded by the increased transfer time overhead. In the case where buffering is provided for one single chain, the additional overhead would be measured in disk rotations. This is compared with other order preserving linear hashing methods which require one disk seek for each home bucket access. Furthermore, larger disk buffers cannot be used by such schemes.

The control mechanisms do not assume an underlying probability distribution of the keys over the key domain. They are designed so that DRSAM files adapt to most key distributions. The dynamic administration of these methods rely on heuristic strategies which will be evaluated with real data files. Details of these methods are found in [HAC88b].

12.5.4.4. Inverted Dynamic Surrogate Files

In this section, EDRSAM is discussed within its application to inverted surrogate files leading to the dynamic counterpart of transformed inverted lists. This physical model extends TIL for information/knowledge retrieval applications.

System Model

Using proper hashing functions on the attributes of a tuple in a relational table, its surrogate file representation is built. In Figure 12.5.1, the example surrogate file image of a knowledge base relation has the following values for argument position 2: br2=010011010, br4=010101011 and br5=010101110. A fully inverted dynamic surrogate file (IDSF) would consist of $|A_r|$ EDRSAM files. The EDRSAM surrogate file records for attribute "i" are composed of the BR of the hashed values (instantiations) for that attribute, with the corresponding Uid. Assuming a file level of 3 and using the given values a typical EDRSAM surrogate file bucket with its associated records is shown in Figure 12.5.30. br2, br4 and br5 hash to the same bucket #2.

In actual implementations, only Prefix(BR,(#BR-fl)) bits are needed to be attached with the unique identifier; where Prefix(K,n) is the right "n" bits of "K" and fl the home level of the DRSAM bucket under consideration. This would mean that the inverted files would use the space more efficiently as the file grows: if the block structure of Figure 12.5.31 is followed, variable length records would be considered with sizes depending on the hashing level of the addressed block. This would certainly increase the complexity of the managing software and its associated hardware to deal with such a blocking scheme. But it leads to an

BR	Uid
010011010	uid2
010101011	uid5
010101011	uid12
010101110	uid7

Figure 12.5.30 Block structure with fixed length records

Prefix	Uid
(BR,(#BR-1))	
011010	uid2
101011	uid5
101011	uid12
101110	uid7

Figure 12.5.31 Block structure with variable length records

efficient use of the storage space for the inverted surrogate lists.

Storage Overhead

In this section, the storage overhead for the IDSF model is analyzed. The notation in Table 12.5.3 is followed. Assuming 15 characters per argument, the number of tuples is computed as $N_r = \frac{S_{db}}{15 \times |A_r|}$ and the minimum number of bits required for a BR is $\left\lceil \log_2 \frac{N_r}{C(i)} \right\rceil$. Then, the surrogate list size is $S(i) = \frac{N_r \times sfix}{8}$ $(bytes)$.

The capacity of a block (bc) is computed as $bc = \frac{B \times 8}{sfix}$ records. Estimating the ratio of overflow records to N_r to be $\%ovf$, the number of home buckets $(\#hb)$ is $\#hb = \frac{N_r \times (1 - \%ovf)}{bc \times auf}$ and the file level is determined by $fl = \left\lfloor \log_2 \#hb \right\rfloor$. This estimate assumes one partial expansion for each full expansion.

In the numerical evaluation two relation file sizes (S_{db}) of 10 Mbytes and 1 Gbytes are assumed with the following characteristics: each file has six arguments $(|A_r| = 6)$ of 15 characters each. Disk block size is set to $B = 2048$ bytes and $C(i) = 1$. Uids are encoded with a 4 bytes word (32bits), a file storage utilization (auf) of 90% and 20% overflow records $(\%ovf)$ are assumed. The results are reported in Table 12.5.4.

The value of BR-fl = 8 bits checks with the intuitive feeling that variable length records, as advocated in Figure 12.5.31, efficiently use the storage space allocated for EDRSAM. The ratio of the variable length to fixed length records is 0.82 and 0.71 for the 10 Mbytes and 1 Gbyte files respectively. This presents a substantial saving on the inverted surrogate list size with fixed record formats. A value of 8% of S_{db} is a good estimate for the inverted surrogate list size.

In [BER87 and HAC88a] it is shown that the storage overhead of TIL surrogate files lies within a reasonable range of the database size (10 to 30 %). The analysis presented in those papers assumes static files (or stable files) that are initially loaded and stored in compact form. For an IDSF built with EDRSAM, the storage overhead is slightly larger and is caused by the additional space required to manage dynamic files. The total overhead for the IDSF model is within 50% of the original database.

Query Response Time with the IDSF Model

Notation	Meaning
B	Minimum block size that is transferred from main memory (Blocking Factor)
bc	Capacity of a bucket (in number of records)
$L(i)$	Average number of matches for an argument i specified in a query. $L(i) = C(i)$ for partial match queries
R_q	Average number of arguments in a query
S_{db}	Database size
$S(i)$	Inverted surrogate list size for argument "i"
t_{wc}	Average word comparison time
wl	Word length
fl	DRSAM file level
$sfix$	size of a fixed length surrogate record: $sfix$ = Uid+BR
$svar$	size of a variable length surrogate record: $svar$ = Uid+BR$-fl$
$\%S_i$	surrogate list size to database size ratio: $\frac{S(i)}{S_{db}}$
$\#A_r$	size of an argument in the database file (bytes)
v/f	compression ratio: $\frac{svar}{sfix}$

Table 12.5.3 Notation for IDSF

Meaning	S_{db} = 10^7 bytes	S_{db} = 10^9 bytes
BR (bits)	17	24
N_r	11×10^4	11×10^6
sfix	49	56
$\%S_i$	7.6%	7.7%
fl	9	16
BR-*fl*	8	8
svar (bits)	40	40
v/f	0.82	0.71

Table 12.5.4 Storage overhead for IDSF

In this section, an insight into the equations that govern the query response time (QT) for inverted surrogate files is presented. For IDSF, QT is divided into three processes:

1) Surrogate file processing and Uid retrieval (SFT).

2) Uid intersection time (IT).

3) Database access time (DA) to read the identified record(s) satisfying the query.

The query response time is written as: $QT = SFT + IT + DA$

1. Surrogate File Processing Time

SFT is determined by the number of disk accesses required to retrieve the matching Uids. Denote by $AC(i)$ the average disk access cost for the surrogate inverted list of argument i and Q the query specification with R_q arguments. Then, the average surrogate file processing time can be written as:

$$SFT = \sum_{i \in Q} AC(i) \propto R_q$$

Denote by $Ovl(i)$ as the average overflow chain length for the DRSAM file of argument i and t_a the disk access time which could be t_{rda}, t_{sda} or t_{fsda}. With $IC(i)$ as the index cost, the access cost to the surrogate inverted list is written as:

$$AC(i) = (1 + Ovl(i) + IC(i)) \times t_a$$

where $IC(i)= 0$ if the index table resides in main memory. For a DRSAM file and a storage utilization over 90%, $AC(i)$ is expected to be less than 1.2 disk accesses. The search time of the retrieved blocks is not accounted for as it can be overlapped with the retrieval process and is neglected [HAC88a].

2. Intersection Time

With no loss of generality conjunctive queries are assumed, as the union operation for disjunctive queries has the same level of complexity as the intersection operation. Two cases are considered:

1) $R_q = 1$: no intersection is required.

2) $R_q > 1$: when more than one argument value is specified in a query, the lists of retrieved Uids must be intersected. Denoting by $NC(R_q)$, the number of comparisons required to perform the intersection operation, the total intersection time is then written as:

$$IT = \begin{cases} t_{wc} \times \left\lceil \frac{\lceil \log_2 N_r \rceil}{wl} \right\rceil \times NC(R_q) & \text{if } R_q > 1 \\ 0 & R_q = 1 \end{cases}$$

where t_{wc} is the average word comparison time and wl the word length in bytes.

An estimate of the number of comparison steps, $NC(R_q)$, for the intersection operation is derived, based on Stockmeyer and Wong's work [STO79]: the bounds on the number of comparisons, $I(m,n,k)$, required to intersect two lists, m and n $(m \le n)$, of arity k is given by:

$$I(m,n,k) \le (m+n) \times \log_2 m + (m+n-1) \times k - m + 1$$
$$I(m,n,k) \ge Max[(m+n) \times \log_2 m - 2.9m, (m+n-1) \times k - m + 1]$$

In this case the arity $k=1$ and the number of comparisons, $NC(2)$, to intersect two lists of cardinalities $C_1 \le C_2$ is :

$$Max[(C_1 + C_2) \times \log_2 C_1 - 2.9C_1, C_2] \le NC(2) \le (C_1 + C_2) \times \log_2 C_1 + C_2$$

The upper bound is based on sorting the list of smaller cardinality prior to performing the cross lists comparison in at most $C_2 \times \lceil \log_2(C_1+1) \rceil$ comparisons. It is known that two_way merge sort on a uniprocessor requires at most $C_1 \times \log_2 C_1$ comparison steps. It is easy to derive an algorithm that would perform within the specified bounds [KNU73]. Furthermore to intersect more than 2 lists, the number of additional comparisons depends on the expected number of "hits" from the first two_list intersection. Denoting this number by GD, $GD = \frac{C_1 \times C_2}{N_r}$. For $R_q = 3$, one needs $C_3 \times \log_2 \lceil (GD + 1) \rceil$ additional comparisons. So that $NC(3)$ is written as:

$$NC(3) \le NC(2) + C_3 \times \log_2 \left\lceil \left(\frac{C_1 \times C_2}{N} + 1 \right) \right\rceil$$

The process can be extended to include additional intersection steps for larger values of R_q. It is noted that Cardenas [CAR75] does not attempt to give an estimate of the intersection time and Fedorowicz's approach [FED87] is different from the one presented here.

An evaluation of the intersection time IT was performed for file sizes of 10 Mbytes and 1 Gbytes and with the same characteristics as the one assumed for the storage overhead evaluation. The plots in Figures 12.5.32 and 12.5.33 illustrate the intersection time with $t_{wc} = 3$ μsec and $wl = 16$ bits. While acceptable for low values of $C(i)$, the intersection time increases with an increasing value distribution factor. In Figure 12.5.32 for $S_{db} = 10$ Mbytes, the intersection time is less than 100 msec for $C(i) \leq 2^9$. In Figure 12.5.33 for a file size of 1 Gbytes, the intersection time is within 200 msec up to $C(i) = 1024$.

It is noted that the plots represent the intersection time computed for equal attribute selectivities. For example if $R_q = 2$, the same $C(i)$ (or $L(i)$ for range searches) is assumed for both arguments in the query. If one follows the reasoning that the probability of both arguments in the query having high redundancy factors is low, then the plots are pessimistic and realistic values for IT would be noticeably smaller. This argument can be made for any database size.

Moreover these plots represent upper bounds on the expectation for the intersection time. Consider the intersection of two lists of constant total length: $C_1 + C_2 = Ct$. Following the equal selectivity approach to estimate intersection time is equivalent to setting $C_1 = \left\lfloor \frac{Ct}{2} \right\rfloor$ and $C_2 = \left\lceil \frac{Ct}{2} \right\rceil$. Then it is clear that $(C_1 + C_2)\log_2(C_1)$ is maximized. Nevertheless, with a VLDKB the plots in Figures 12.5.32 and 12.5.33 reflect a potential computation bottleneck when lengthy lists are intersected. This implies that inverted surrogate files are not efficient when the selectivity of the individual arguments in a query are relatively small. In this case, special intersection hardware could be designed to speed up the intersection operation.

3. Database Access Time

With GD as the number of good responses to a query and the probability $\left(\frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}\right)$ of a given response to be in a specific block, the database access time is [CAR75]:

$$DA = T_a \times \left\lceil \frac{S_{db}}{B} \right\rceil \times \left(1 - \left(1 - \frac{1}{\left\lceil \frac{S_{db}}{B} \right\rceil}\right)^{GD}\right)$$

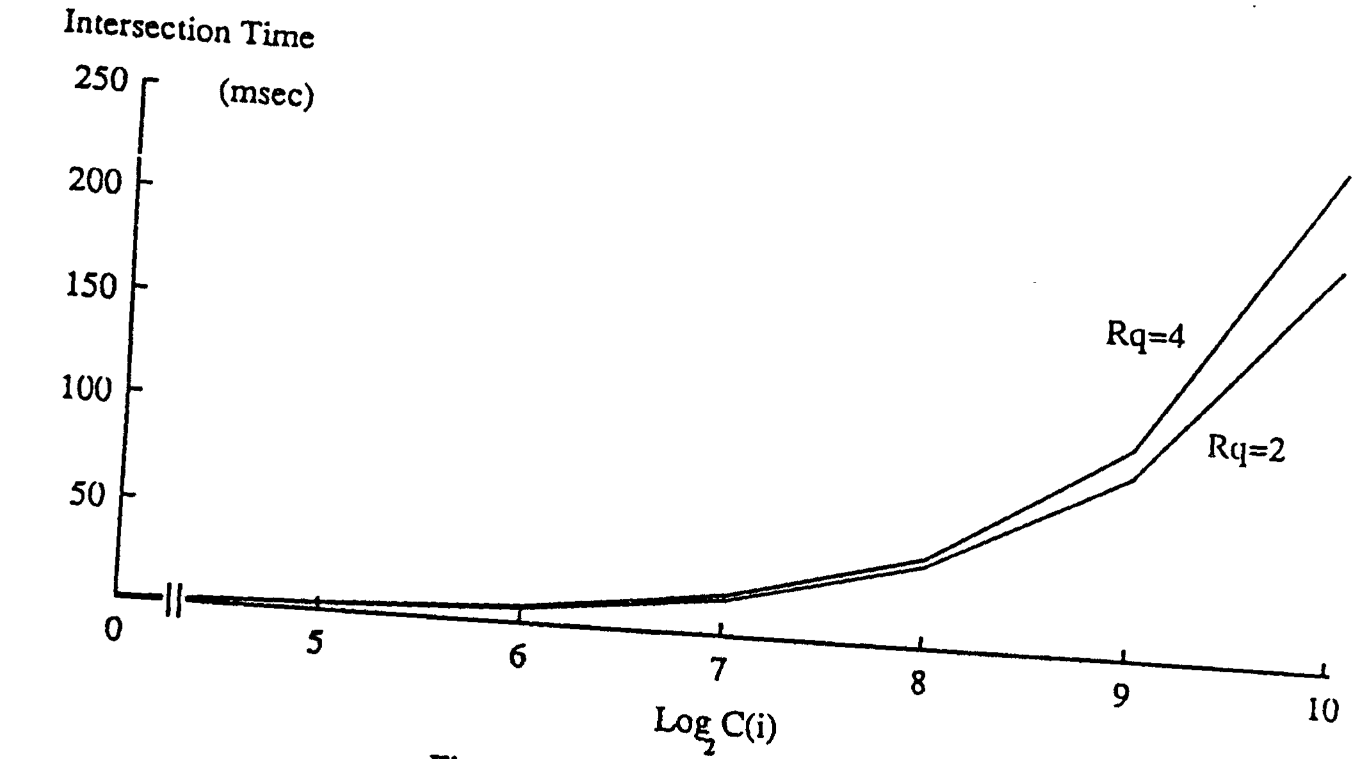


Figure 12.5.32 Intersection Time
(Sdb= 10 Mbytes, Ar=6)

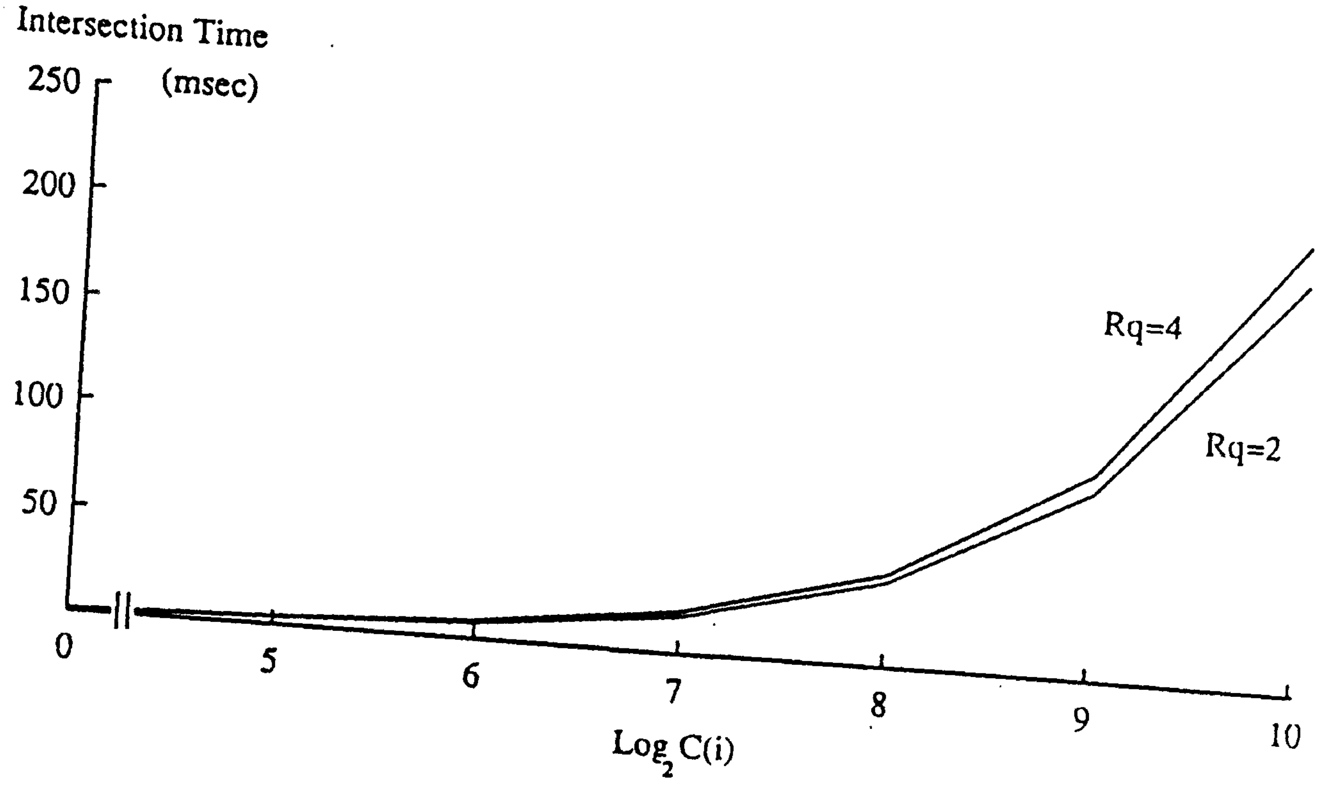


Figure 12.5.33 Intersection time
(Sdb= 1 Gbytes, Ar=6)

In the analysis of the intersection operation the number of good responses for $R_q = 2$ is written as $GD = \frac{C_1 \times C_2}{N_r}$. Assuming uniform distributions for the values of an argument, the number of good drops is extrapolated to:

$$GD = N_r \prod_{i \in R_q} (\frac{C_i}{N_r})$$

where C_i is replaced by L_i when dealing with range searches.

The database access time equation follows Cardenas' equation [CAR75] and assumes direct access to the main database. It is based on successive selections with replacement. Yao [YAO77] discusses selection without replacement and points out the cases where Cardenas' equation gives rise to a significant error. For most purposes, Cardenas' approach is satisfactory as the number of good responses is expected to be small for very large knowledge bases. Following proofs by Christodoulakis [CHR84], it is observed that the equation for database access time is an upper bound and presents a pessimistic evaluation of this response time components.

Better bounds for DA could be derived but they would depend on the locality of the good responses and would be determined by the clustering scheme for the tuples in the existing database. In the derivation for DA, a uniform distribution was assumed for the records over the secondary storage blocks. In a multi-user environment, clustering can improve DA for some applications. Database clustering [JAK80, OZK86] is still an open design problem that lies in the class of NP_Complete problems.

In conclusion, as with inverted lists, the efficiency of the IDSF model depends on the selection of the arguments which are used as secondary indices [SCH75a, SCH75b, AND77]. If the selectivities of the arguments is relatively high the model becomes very efficient and presents the added flexibility to be used for multi-user and dynamic environments. Compared to conventional inverted lists, IDSF have the following characteristics:

1) the storage overhead for IDSF is less than the main database size. While the size of the indices of conventional inverted lists is equivalent and even in excess of the database size.

2) IDSF is based on a dynamic hashing method, namely DRSAM. This file structure is near optimal for random access and efficient for range queries. Typically, a random access and range query processing requires one disk seek, excluding overflow chains. This should be compared to at least two disk seeks for indexed-sequential files. Finally, the update cost of indexed-sequential files with overflow chains [LAR81] degrades fast when the file size changes. On the other hand, DRSAM files adapt well to dynamic

environments. Additional details are given in Appendix 12-D.

12.6. Optics in Very Large Knowledge Bases.

Optical computing (especially in its analog form) has been widely us^ed in applications like optical image processing, pattern recognition and signal processing due to its highly parallel nature. Another area that can benefit significantly from the advances in optical technology is that of the Very Large Knowledge Bases (VLKB). Optics can play a key role in the future VLKB [BER87b] providing larger storage capacity, higher transfer rates and parallel data manipulation. This section discusses some of the possible improvements in the VLKB performance if optical computing is involved. It focuses on a scheme for the efficient implementation of relational data base operations using optical techniques.

12.6.1. Optical Data/Knowledge Base Machines.

A common approach to deal with the problems associated with the processing of enormous amounts of data in VLKB is the incorporation of a Data/Knowledge Base Machine (D/KBM). A D/KBM with multiple storage units, multiple processors and the appropriate interconnection network will operate as a back-end machine to a host, thus removing the bulk load of searching through the knowledge base from the front-end computer.

The need for large capacity and high bandwidth secondary storage will be satisfied by using optical disks [ALT86, CHE86] which can hold up to 10 GBytes per platter (14-inch diameter). Currently, access times of optical disks are larger than those of magnetic disks [CAR86]. The reason is that the focusing optics are bulkier than the 'flying' miniature heads of magnetic disks. Data rates are comparable, with potential for improvement since optical disk technology is relatively new.

However, in contras: "vith magnetic media, there are two promising possibilities for increased optical disk performance by at least two orders of magnitude both in terms of access time and sustained data rates. First, the read/write beam could be deflected from track to track very rapidly (on the order of 100 microseconds) by entirely optical means. Second, due to the non-interference of light beams and the relatively large head to medium spacing one could imagine multiple beams being used for reading data with a single head carriage assembly [CAR84]. Alternatively, an unfocused beam could simultaneously read data from more than one point of a transmissive disk surface [MOS87]. This, coupled with the possibility of multiple heads will allow for enormous data rates. If we assume achievement of access times of 100 microseconds and data rates of 300 MBytes/sec, this represents almost two orders of magnitude improvement over current magnetic disks.

Input/Output systems will have to be designed with these rates in mind. Current electronics would be hard pressed to handle them. However, if data could be preprocessed "on the fly" in its optical form, then the ultimate data rate to the electronics would be much lower on the average and the data much "richer" in information. Intelligent use of optical pattern matching could provide us with a set of primitive operations that could help efficiently implement higher order functionality like, for instance, a subset of relational algebra operators.

For applications which demand fast searching of many megabytes of data all this is very promising. But with current electronics technology if every subsystem of a machine needs to cater to such high rates then its cost will be much higher than necessary.

12.6.2. Relational Operations

The most common relational operations in knowledge and data base applications are projection, selection and join. Other important operations include sorting and text retrieval. Search of fixed format data (e.g. indices or pointers) could make effective use of optical content-addressable memory which can be implemented by multiplexing a large number of holograms in a thick recording material like lithium niobate [GAY85].

The performance of a data base machine with a high degree of parallelism depends largely on the efficiency of the interconnection network. Therefore, the capabilities and limitations of the interconnect technology utilized in realizing a computational or signal processing unit are essential in determining the speed and flexibility of the operations that can be achieved by that unit. Optical signals can flow through three-dimensional space to achieve the required interconnect pattern between elements of a two-dimensional data array before executing the desired operation between them [GOO84]. To examine more closely these advantages, four categories of operations must be considered.

In the first category belongs only projection since it is the only operation that does not require comparisons (except possibly in a second stage where any duplicates have to be removed). Each element of a one- or two-dimensional array is dropped or retained based only on its position in the array and not its value.

The second category is that requiring single element operations like selection and text retrieval. In sucn computations, each elemen*t* in a one- or two-dimensional array is processed independently from the rest of the array elements. The interconnectivity required by these operations is the loading and unloading of data to a processor array. Clearly, optical interconnections have the advantage of being able to input an entire data array in parallel using the third dimension for data propagation. On the other hand, in our electronic associative processor, data can be input and output only along the edges of a two-dimensional array, one row-column at a time. Optics have a lot to offer in D/KB systems where single-element operations are common.

Another category of operations is that of sorting, which is especially important in D/KB systems. Computations of this type require global interconnections between all the elements of the input array, that is, every element of the output array is dependent on all the entries in the input array. The structure of the sorting problem suggests an efficient algorithm in which computations grow as O(N*logN). In order to achieve these computational savings, complex interconnect configurations are necessary among the input elements of the array. Additionally, these interconnections have to be changed during the different stages of the computation. The requirement for dynamic interconnections can be exploited by employing the perfect shuffle function configuration. The perfect shuffle can be applied repeatedly at each stage of the computation to produce the currently desired interconnect pattern, presumably at the expense of extra time required to complete the interconnections. Optics offer the perfect shuffle function efficiently, hence its use in hardware sorting units would lead to improvements in system throughput.

The fourth category includes space and time variant operations like equi- and theta-join. The input relations can form two one-dimensional arrays and each element of the first array must be compared to all the elements of the second array. The interconnectivity pattern for these operations varies in space and time. Furthermore, the various interconnections are data dependent, making it impossible to predict in advance the appropriate interconnection patterns required at the different stages of the computation. The throughput of a parallel machine implementing this type of operations is critically affected by the availability of a dynamic and global interconnect network. Many processors could be idle for a significant number of cycles waiting for data to be properly routed to them. The overhead associated with the supervision of a controller in such a multiprocessor environment lacking space and time variant interconnection network may severely degrade all the advantages of parallel processing. Optics again offer great interconnection flexibility.

The last three categories of operations involve extensive comparing of the input values to some reference either fixed (selection, text retrieval) or constantly changing (join, sorting). As a result, the design must be capable of performing multiple parallel comparisons rapidly. Optical comparisons can be achieved efficiently using the AND-OR-INVERT (Exclusive-Or) logic primitive which will be presented in the following section.

12.6.3. Optical Comparisons.

Two n-bit words A and B are equal if $A_i\overline{B}_i + \overline{A}_iB_i = 0$ for each pair of corresponding bits A_i and B_i. Since for the comparison both the value of the bit and its complement are needed, each n-bit word will be represented by 2n light beams (i.e. the 4-bit word 1011 will become 10-01-10-10). Using this method a 00 combination corresponds to a "don't-care" character while the 11 combination always produces a "Not-Equal" result. The coding scheme for the two logical values, 1 and 0, can be either light and no light or horizontal and vertical polarization respectively.

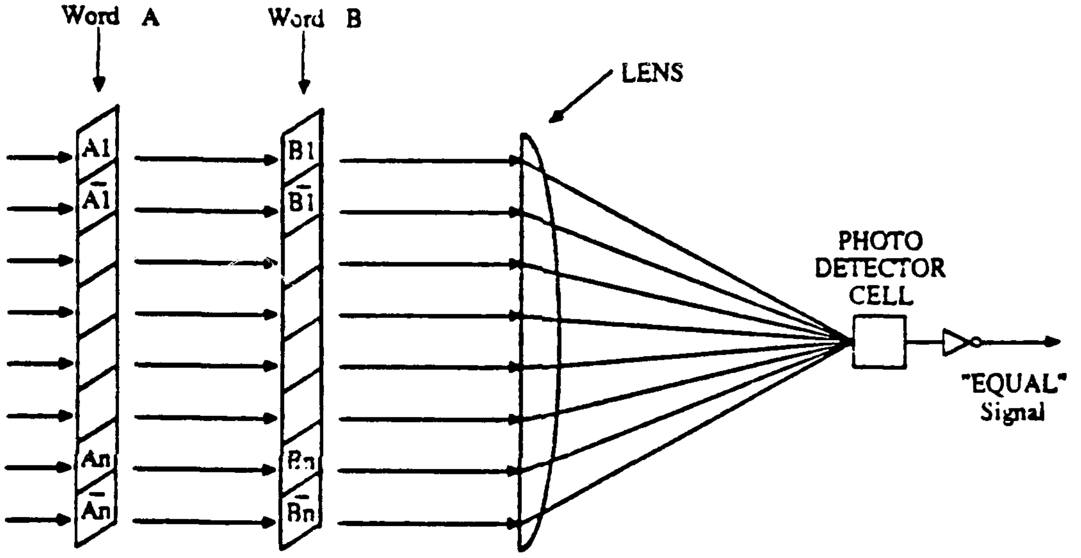
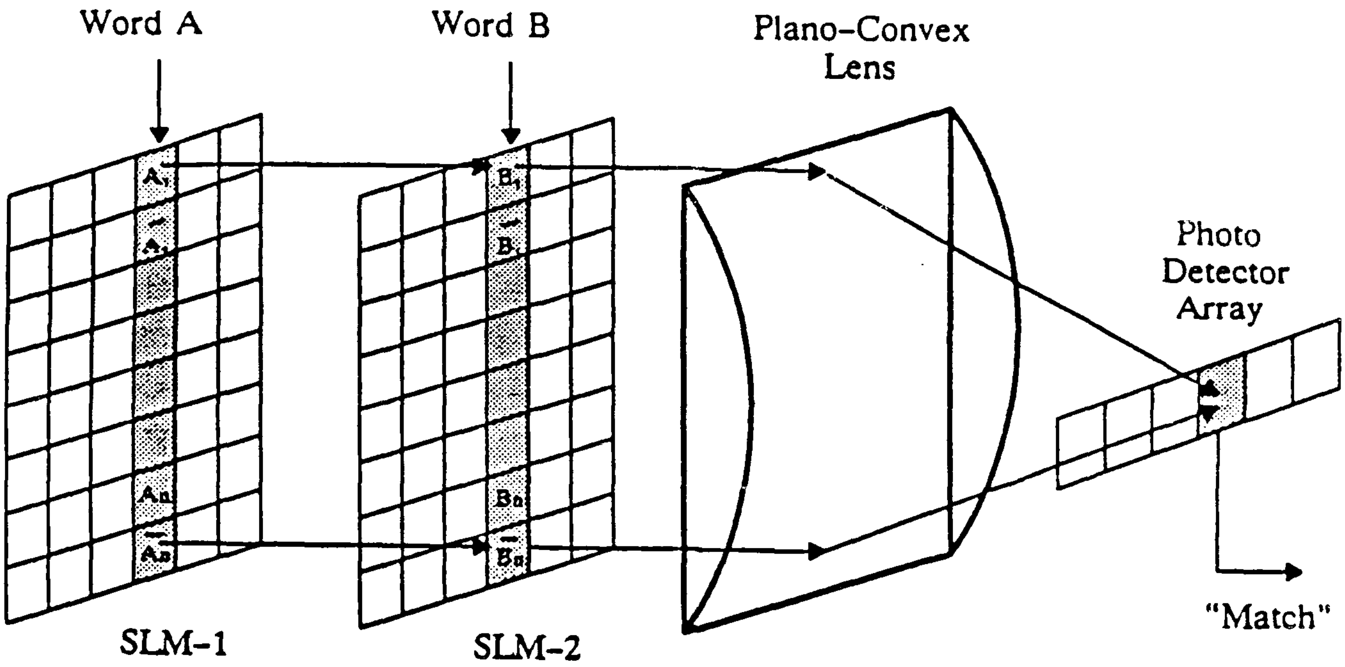


Figure 12.6.1 Optical comparison of two n-bit words.

As can be seen on Figure 12.6.1, the light beams are superimposed bit-wise and are focused by means of a convex lens on a single photodetector which performs the logical OR (or summation) of all the beams. If no light is detected the two words are equal while any level of light intensity other than zero indicates that the two words differ in at least one bit. The output of the photodetector is electronic [GUI86].

Multiple word comparisons can be performed in parallel if two 2-dimensional arrays, A and B, are employed each having 2n rows and m columns. At any instance in time the word at the i-th column of the A array is compared to the word at the i-th column of the B array. The result (equal or not equal) is recorded on the i-th cell of a row of photodetectors. This configuration, depicted on Figure 12.6.2, allows for m comparisons of n-bit words to take place simultaneously.



Less-Than or Greater-Than comparisons are feasible with the same configuration if some additional logic is employed. In such a case comparison proceeds row-wise starting from the most significant bit of the words to be compared.

The information on each array has to be recorded on two-dimensional spatial light modulators (SLMs). The next section discusses the characteristics and the capabilities of some currently available SLM devices.

12.6.4. Spatial Light Modulators

Spatial light modulators constitute an essential part of any optical data processing system [WAR87, PEN86, KNI81, CAS77]. These active optical devices have the ability to: a) store on a one- or two-dimensional array information encoded in an input (write) electrical or optical pattern, and/or b) spatially modify or amplify some of the optical characteristics (phase, amplitude, intensity, polarization) of a readout light distribution as a function of space and time.

The spectrum of SLM applications is impressingly broad. It ranges from real time operations (convolution, correlation, heterodyne detection) to pattern recognition techniques, white-light and color image processing, analog and digital optical computing (matrix-vector products, matrix inversion, binary multiplication, solution of systems of linear equations, least squares solution etc), spatial filtering and associative processing.

SLMs may operate in either transmissive or reflective mode. They can be classified to electrically or optically addressed SLM (E-SLM and O-SLM respectively) according to the nature of the control or write signal. Different versions can process optical signals in 1-D, 2-D or 3-D formats.

A different classification divides SLMs into three classes on the basis of their functional capabilities.

1) Signal Multiplication and Amplification SLMs.

These are three-port devices with a structure similar to that illustrated in Figure 12.6.3.

In a typical amplitude-modulation application the amplitude at each point in the output image is determined by the product of the input signal at a corresponding point on the device and the readout image amplitude. The input write image generates a distribution pattern of electric fields which in turn cause the light-modulating material to modify the polarization, phase and/or amplitude of the readout light. The mirror and light-blocking layers permit the written information to be read out by reflection and prevent leak through of the readout light to the photosensor.

Modulators in this category can be liquid crystal devices (Hughes Liquid Crystal Light Valve, LCLV [GRI75, BLE78]), nonholographic photorefractive signal-multiplying devices (Pockels Readout Optical Modulator, PROM [FEI72, NIS72]), microchannel plate devices (Microchannel Spatial Light Modulator, MSLM [WAR81, HAR86] and Photoemitter Membrane Light Modulator, PEMLM [FIS86, SOM72]), thermoplastic modulators, volume holographic devices or semiconductor O-SLMs (Silicon/PLZT modulator [LEE86]). The Photo-DKDP (Deuterated Potassium Dihydrogen Phosphate [ARM85, DON73]) light valve must be cooled down to -51° C to operate properly. One-dimensional E-SLMs can be magnetooptic (LIGHTMOD [ROS83]), micromechanical, electrooptic or acoustooptic devices. Acoustooptic Bragg cell modulators are the best developed and the most widely used SLMs.

2) Self-Modulating devices.

In these two-port optically addressed SLMs the input light modifies the optical properties of the modulating material which in turn modifies the light beams as they pass through or

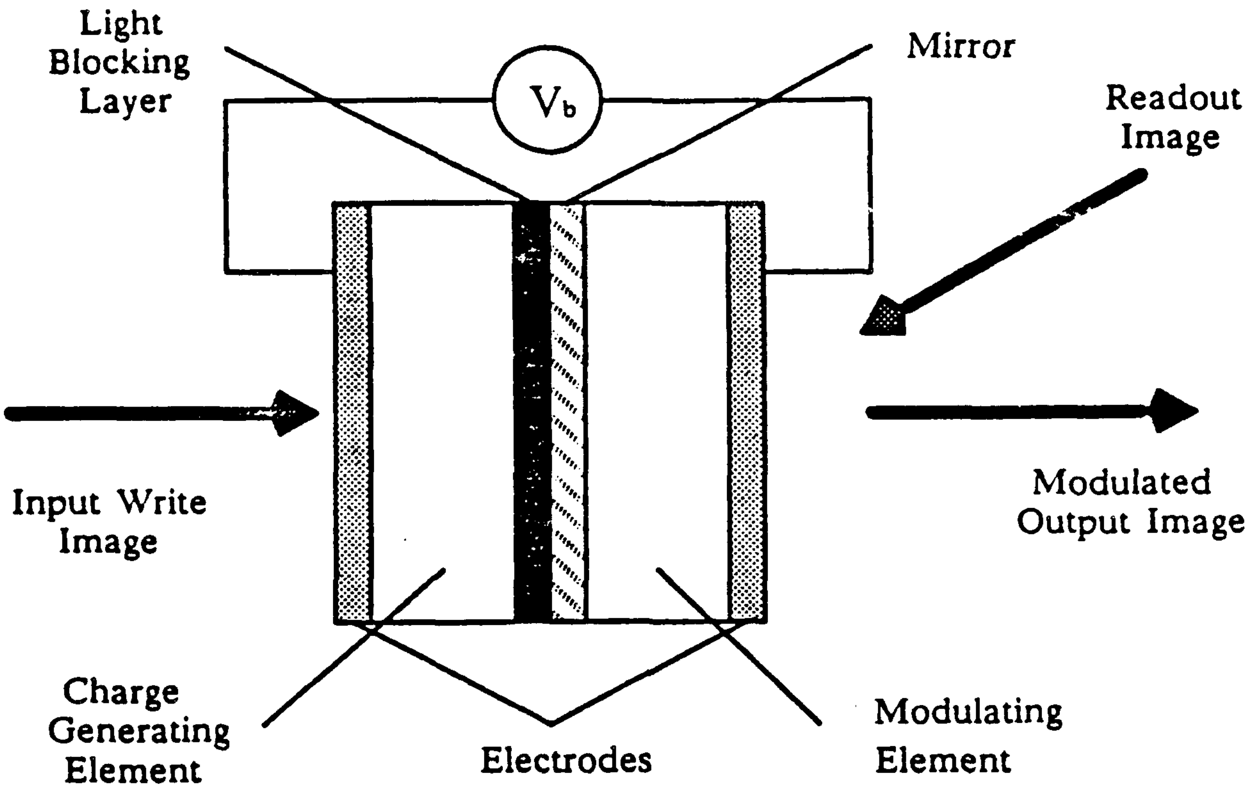


Figure 12.6.3 The structure of a signal-multiplying spatial light modulator. (Reflective mode)

reflected from the device to become the output beam. Semiconductor etalons, Fabry-Perot cavities and other bistable devices are typical self-modulating SLMs.

3) Self-Emissive devices.

Self-emissive modulators generate a coherent or incoherent spatial light distribution under control of electrical or optical input. Examples include CRTs, LED arrays, image intensifier tubes, photoconductor-accessed electroluminescent devices etc.

For our purposes we need a two-dimensional, optically addressed spatial light modulator on which binary information can be recorded and used in a fast and reliable way. A resolution in the order of 1000×1000 pixels or better will be adequate if the total time needed to write and erase a frame (framing speed) is kept in the order of 10^{-4} seconds and lower. These numbers bring the potential time-space bandwidth of the device to 10^{10} operations/sec with roughly 10^{7} records being processed every second.

Table 12.6.1 summarizes some of the physical characteristics of 2-D O-SLMs. In terms of potential throughput PEMLM and Si-PLZT seem to be the best candidates for the job since they exhibit the largest time-space bandwidth.

	LCLV	PROM	DKDP	MSLM	PEMLM	Si-PLZT
Exposure Sensitivity μJ/cm^2	1-5	10	10	0.01	0.01	1
Storage Time	msecs	~1hour	long	days	days	10s
Contrast Ratio	<100:1	1000:1	70:1	1000:1	1000:1	10:1
Framing Speed (Hz)	40	30	30	200	1000	10^4
Spatial Resolution (cycles/mm)	20	5-10	75	5-15	20	10
Time-Space BW (pixel op/sec)	10^7	$9*10^6$	10^7	$2*10^9$	10^{10}	10^{10}

TABLE 12.6.1 Some characteristics of O-SLM devices.

The initial design of a system that performs the relational operations projection, selection and equi-join using the above devices will be presented in the next section.

12.6.5. An Optical System for Relational Operations.

Figure 12.6.4 illustrates the initial design of a back-end all-optical Data/Knowledge Base Machine. It is assumed that binary information is read from some optical storage medium using a single or multiple laser beams and is available in its optical form for further processing. The way in which records of a certain relation are retrieved is not a major issue since the distribution of light beams can be spatially modified by means of various optical elements (lenses,

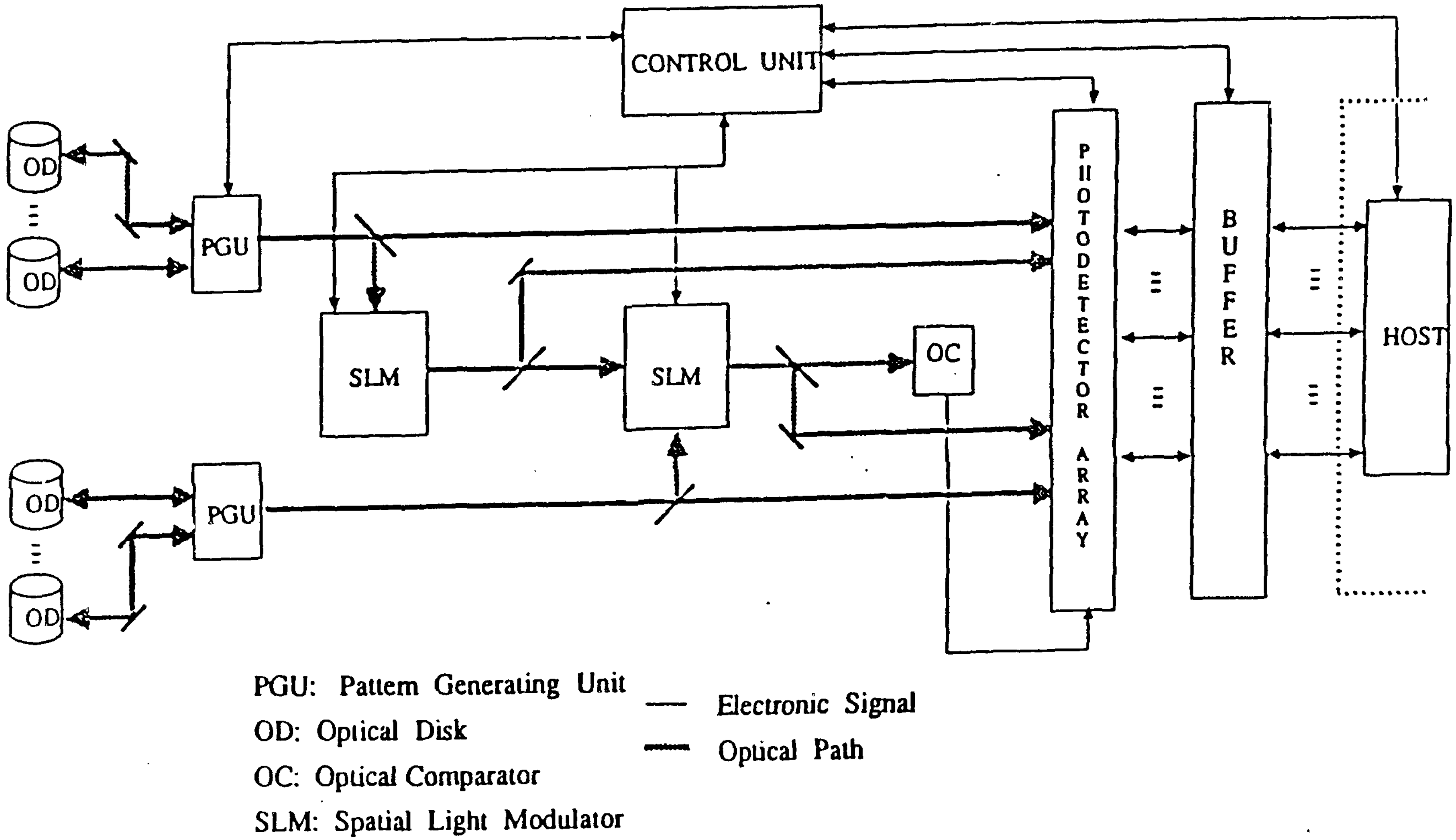


Figure 12.6.4 An Optical Data/Knowledge Base Machine

mirrors, beam splitters etc). However, every relation has to reside in a different storage volume thus allowing for simultaneous access to any subset of the data base relations.

When the host issues a request for a transaction to the data base the participating data is located on the optical disk units, retrieved and an appropriate beam distribution is formed in the Pattern Generating Unit. The optical data is processed, if needed, by the main processing unit of the system, the two-SLM configuration described earlier, and the result is recorded on a photodetector array.

The Pattern Generating Unit (PGU) will contain the necessary optical elements to generate "on-the-fly" the input optical patterns for the corresponding SLMs. There will be one PGU for each SLM. The output of the PGU can be sent either directly to the SLM or, via a beam splitter, to the detector array. An active optical device is required to produce any fixed optical patterns representing the constant comparison values provided by the particular query. At a later stage a feedback optical path from the SLMs to the PGUs may be included to allow for optical processing in a loop.

The final result of a transaction is recorded on an array of photodetector cells which transforms the optical information to electronic. The resulting electronic pattern is stored in a fast semiconductor buffer (which may operate in a ring-type fashion) and eventually passed to the host. The size of the photodetector array depends on the length of the records in the data base and must be large enough to accommodate the answer to any type of query. In any case it has to be no longer than the sum of the sizes of the two longest records in the data base. Each element of the array can be active or inactive according to control electronic signals issued either by the Control Unit or the comparator.

The Control Unit accepts a transaction request from the host and translates it to the appropriate control signals to the disks, the PGUs, the SLMs and the detector array. It also informs the host when the result is in the electronic buffer.

12.6.5.1. Projection

Projection can be easily performed without the use of SLMs when it is the only or the last operation required by a certain transaction. All that is needed is the deactivation of the rows or columns of the photodetector array that correspond to the data fields to be masked out. As a result, only the useful part of each record is passed to the buffer. Since the result of a projection may contain multiple similar entries, the removal of the duplicates will be performed electronically.

However, projection is frequently followed by additional operations. In this case the necessary mask out will take place "on-the-fly" inside the pattern generating unit and the remaining data fields will be spatially compressed by a set of prisms before they are written on one SLM for further processing.

12.6.5.2. Selection

Selection is based mostly on comparison of a particular data field to a constant value. The records of the participating relation are written, one per column, on the first spatial light modulator, SLM_1. On the second modulator, SLM_2, an optical pattern is written consisting of the constant value at the bit positions corresponding to the data field to be compared and "don't-cares" for the remaining bit positions. N comparisons take place simultaneously, where N is the number of resolvable pixels in the horizontal dimension of the SLMs. If the entry in a data field of a record is equal to the constant value a match is detected at the optical comparator and the corresponding column of the photodetector array is activated to record the qualified tuple which follows the alternate optical path shown on Figure 12.6.4. During the next cycle another N records are loaded to the S_1 and compared. Therefore, the service time, T_{Sel}, for a selection operation on a relation R with N_R records is given by:

$$T_{Sel} = T_{su} + \frac{N_R}{N} * T_{Fr}$$

where T_{su} is the initial set-up time and T_{Fr} is the time needed to input an entire frame to the SLM.

If the size of a record is larger than the number of rows in the SLM then two or more columns of the array can be used. Another possible solution is to perform a projection beforehand to drop the undesired data fields, if any.

12.6.5.3. Equi-Join

In the equi-join operation the fields of two records belonging to two different relations, R_1 and R_2, are concatenated only if the entries in a common data field are equal. In our system the records of relation R_1 are loaded on SLM_1 as described before while those of R_2 are written on SLM_2 in such a way that the common data field occupies corresponding bit positions. The decision at the optical comparator is based only on the comparison of these bit positions.

If a match is detected during the ith cycle and the respective photodetector columns are activated, the tuples from both modulators are recorded on the detector array physically concatenated. During the i+1th cycle the entries in SLM_1 are shifted (actually the entire array is erased and rewritten) one column to the left and N new comparisons take place while in the i+2th cycle the entries in SLM_2 are shifted to the opposite direction for another N comparisons (see Figure 12.6.5).

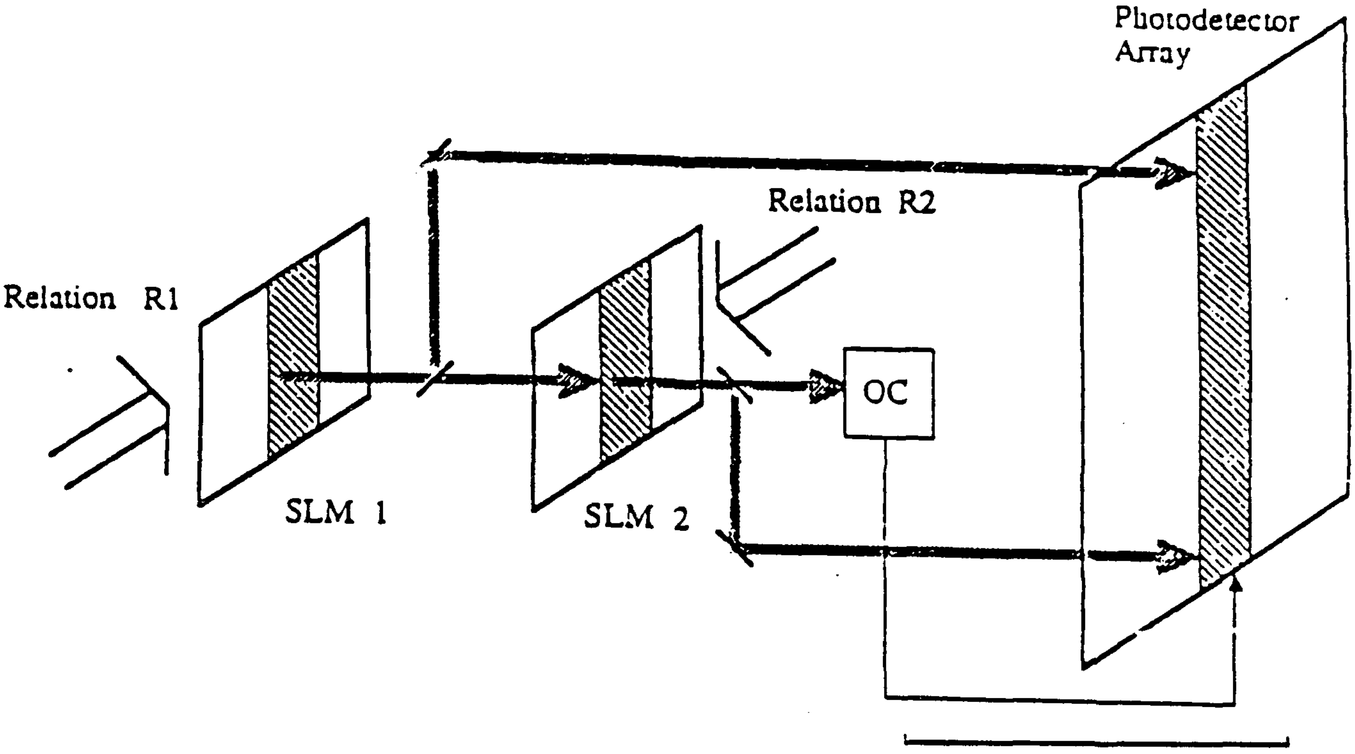


Figure 12.6.5 Performing the equi-join of two relations.

Using a slightly different approach, which may prove more efficient when the number of records of a relation is larger than N, the pattern on SLM_2 remains fixed while the tuples of R_1 slide through SLM_1. When they are exhausted a new pattern corresponding to the next N tuples of R_2 is written on SLM_2 and the process is repeated. Detailed analysis and simulation of various algorithms will provide the optimal solution. Additional details can be found in Appendix 12-E and 12-F.

12.6.5.4. Logic Operations

The proposed architecture can perform filtering of ground clauses in logic-based knowledge base environment. Selection on a conjunction of exact-match criteria is simply accomplished by incorporating all of them in the reference pattern. Disjunction-based selection could be done by using concatenated search patterns if the total length is less than N (and matching on a subset of the detectors), or by connecting more than one optical matcher in a pipeline. Optical inference engines [WAR86] should be more efficient than their electronic counterparts because the parallel searching operation eliminates the need for backtracking through the knowledge base.

12.6.6. Alternatives for an Optical D/KBM.

This section briefly presents various aspects of optical processing that may provide satisfying alternatives to an optical D/KBM. Currently, optical technology is not yet available to efficiently support most of the following ideas but, nevertheless, they all exhibit a high potential.

12.6.6.1. Computer Generated Holograms

Holograms are able to record information about both the amplitude and the phase of a complex wavefront. In Computer Generated Holograms (CGH) [LEE88] the complex wavefront need not come from an existing physical object but can be computer generated as well. Several methods exist for encoding the amplitude and the phase of the wave. They can be classified into two major categories: a) cell oriented holograms (detour phase, non-detour phase), and b) point oriented holograms (threshold, pulse width modulation, pulse density modulation). Binary CGHs have also been proposed.

For our purposes, a computer generated hologram may be constructed as a data filter corresponding to a particular query on the data/knowledge base. Optical data representing records of relations are flashed through the CGH and only the qualified tuples are collected into a buffer for further use. Although it sounds like a simple principle, significant problems remain to be solved, the most important being the type of encoding to be used.

Issues related to real-time computer holography have also been investigated [FEI85]. A critical factor is the delay between the time the hologram is exposed and the time when it is viewed. Naturally, it has to be as short as possible. Currently it ranges from a few picoseconds (using liquid CS_2 as the holographic material) to half a minute (using thermoplastic film). Photographic film is not suitable because of the relatively long time required for its developing process and because it is not reusable.

Two categories of especially useful materials in real-time holography are photorefractive materials and thermoplastic film. They can both store images.

Photorefractive materials contain electric charges that are "frozen" in the dark but can move when illuminated. They are characterized by the strength of the photorefractive effect they exhibit (Pockel's coefficient) and by the speed with which the charges migrate through the material. The speed depends on the intensity of the incident light and response times from minutes to nanoseconds have been demonstrated. $Bi_{12}SiO_{20}$, GaAs and InP are weak but mildly fast while $BaTiO_3$ is strong and slow. A mixture of $KTaO_3$ and $KNbO_3$ (KTN) is strong and fast but crystals of high optical quality are difficult to grow.

Thermoplastic film can be used to record a hologram due to its ability to repeatedly soften and harden when heated and cooled. The recording process typically consists of three different phases: a) positively charging the surface by a corona discharge, b) exposing the film to the image and c) developing. A complete cycle, however, is still considered quite long although it takes less than a minute.

12.6.6.2. Optical Content Addressable Memories

At a higher level, the use of electronic content addressable memory has been considered for improving the performance of database operations. Most of these efforts have not met with much success primarily because of the small size and the high cost of these devices and the slow data loading time. On the other hand, optical content addressable memories have the potential for holding megabytes of data at an appreciably lower cost [BER88]. In addition, they offer parallel output. Since they are hologram-based their major disadvantage is that they are read-only. However, for very large data/knowledge bases indexing structures can be devised which are rather insensitive to updates provided that the update rate remains moderate. Thus, holographic content addressable memories could serve in the future for processing indices to very large databases. As the field develops, they may even be adopted as a primary storage medium.

12.6.6.3. Nonlinear Optics

Nonlinear optics [GIB86] can contribute decisions to optical signal processing and computing. The optical nonlinearity makes the device's transmission intensity dependent, so one can obtain the thresholding needed for logic decision making. Nonlinear decision-making devices can be constructed as waveguides in which the light is guided in tie plane of the nonlinear thin film or as etalons in which the light is imaged from one nonlinear thin film to the next in such a way that its intensity is highest as it interacts with each film.

Guided-wave devices are most likely to find application where data are handled in a pipeline manner, for example, optical-fiber communication and interconnect systems, data encryption, etc. However, waveguides are much like wires except their higher bandwidth. Etalons permit massive parallelism and global interconnectivity; that is, one can perform many operations simultaneously and interconnect in the next plane two or more pixels far apart in the present plane.

Both guided-wave devices and etalons can be used for the implementation of relational operations in a data/knowledge base but the additional features of etalons make them preferable. Consequently we anticipate the use of guided-wave devices in the near term and increased introduction of etalons in the long term.

Of particular interest is a study of the *nonlinear Fabry-Perot semiconductor* etalons [WAK87] used for the design of all-optical logic devices. GaAs etalons may be used for both parallel processing schemes utilizing large two-dimensional arrays of all-optical logic gates and serial processing of data as logic devices for integrated optical circuits compatible with fiber-optic data transmission. The potential application of these optical switching elements in parallel massive data processing is essential to the design of an all-optical Very Large Knowledge Base Architecture.

12.6.6.4. Multivalued Logic

Multivalued logic is easier to implement in optics than in electronics. Light intensity can be set in a large number of values and detected using already available detectors. The conversion of multivalued optical information to the conventional binary code is not a difficult problem. Returning to the previous scheme of a hologram used as a data filter, different levels of light intensity and/or different size of the reading laser beam (corresponding to different certainty factors or degrees of belief) may allow for the implementation of inference engines based on fuzzy logic.

A particular spatial light modulator, called the Variable Grating Mode (VGM) liquid-crystal modulator [TAN83] exhibits the powerful feature of transforming the light intensity to spatial frequency. The readout beam is scattered by gratings whose local periodicity varies with the local intensity of the input-write beam. With such an intensity-to-spatial frequency converter a wide range of nonlinear functions of the input intensity can be implemented by spatially filtering the readout beam.

Consider, for example, the sorting operation whose optical implementation remains a difficult task since its very nature calls for some kind of temporary storage. A method has been proposed by Stirk and Athale [STI88] that uses optical compare-and-exchange modules. A simple optical sorting network based on Fredkin gates can be constructed but its upgrading for larger applications is questionable.

If, however, enough intensity levels are available to encode the entries of a table which is to be sorted, the sorting process will be concluded in just one step using the intensity-to-spatial frequency converter. A detector array capable of detecting spatial frequencies with the desired accuracy will record the sorted list since there exists a bijection between spatial frequencies and table entries.

12.6.6.5. Optics and Artificial Intelligence

The application of optical computing to Artificial Intelligence has been studied. Areas that are of special interest to us include knowledge representation, learning ability, expert systems and neural networks [TAK86, MOS87].

The semantic network scheme is preferable because of its structural resemblance to the the relational data base model. Semantic networks support two important forms of reasoning, namely, inheritance and categorization. Inheritance allows an agent to infer properties of a concept based on the properties of its ancestors. On the other hand, given a partial description consisting of a set of property values, categorization amounts to finding a concept that best matches this description.

The connectionist model has been proposed to encode semantic networks. A connectionist network consists of a large number of simple computing elements (units) connected via links. A unit may have multiple inputs and some local memory and communicates with the rest of the network by transmittiing its output to all units it is connected to. The information encoded in the network is accessed by activating relevant nodes to the network. Thereafter, all the nodes compute in parallel. The answer to the query is available at the end of a specified interval proportional to the number of levels in the conceptual hierarchy. The connectionist model can be applied with some modifications to a relational data base for the efficient solution of complex queries. The difference is that a data base permits massive parallelism without major requirements for adaptive learning.

Optical implementations of one- and two-dimensional distributions of neurons have been considered in coherent and incoherent light. The performance of such networks has been found to conform to the theoretical predictions of storage capacity (the number of entities that can be reliably stored). Architectures and optical implementations of two-dimensional neural net arrangements can provide content addressable associative memory modules that are suited for use with two-dimensional data arrays where the tuples of one or two relations are stored. These associative memories can significantly improve the response to a query, especially when it involves join-type operations. While optics provides the parallelism and massive interconnectivity needed in the implementation of neural networks and their modules, these on their part provide the robustness, fault tolerance and power of non-linear processing and feedback that are generally lacking in optical processing.

12.7 Implementing Knowledge Base Management Systems Based on Surrogate Files

We have implemented two prototypes of knowledge base management systems to demonstrate the use of surrogate files in two different environments. The first system is developed for a combined environment where an existing database system, INGRES, is used as a backend to a Prolog interperter. Here surrogate files serve as an alternative indexing scheme to the traditional ones such as B-tree or Hashing. The second system is for an integrated environment where rules and facts are handled uniformly, which is implemented in *Lisp on the Connection Machine having 32 K processors. This second system shows that the surrogate file technique lends itself well to the parallel processing environment, yet special care must be taken for the possible I/O bottlenecks. Future work involves the use of surrogate files in handling more complex objects such as unrestricted Horn Clauses and multimedia databases. Additional details are given in Appendix 12-G.

12.8 Conclusion

In this report, we have discussed a relational algebra machine based on the CCW surrogate files and analysed the performance of the architecture for parallel selection and join algorithms The basic idea of the proposed architecture is to reduce the number of EDB blocks to be transferred from the secondary storage systems by performing the relational operations on the CCW surrogate file first. It has been shown that we can obtain almost linear speed up in response time for a relational algebra operation by increasing the processors working concurrently.

We proposed CCW-2 surrogate file structure which can be used for partial unifications among first order terms in a very large logic programming environment. Each code word of CCW-2 is associated with a tag field and a value field. The tag field represents any argument type including lists, structured terms, variables and constants. The value field contains the transformed representation of the corresponding argument. We are currently investigating the Extended Concatenated Code Word (ECCW) to directly perform unification on surrogate files.

A new and efficient access method for very large dynamic files, called the dynamic random sequential access method (DRSAM) was developed. It is derived from linear hashing with order preserving. The performance of DRSAM was evaluated and the file structure found to be efficient for range queries as well as random access. With order preserving hashing, the hashed key values are not generally uniformly distributed over the storage address space. To deal with non-uniform distributions, DRSAM was extended with proper control mechanisms and the resulting file structure is called EDRSAM. One application for this method is to index surrogate files which are compressed images of very large databases. The resulting physical model is called inverted dynamic surrogate files (IDSF) and is proposed as an alternative to the conventional and static inverted lists. With IDSF, the reduced storage overhead of surrogate files is combined with the high performance of EDRSAM to provide a flexible and efficient physical model to multi-user dynamic VLDBs.

An initial design for the optical implementation of various operations in Very Large Data/Knowledge Bases has been described. The system is capable of performing projection, selection and equi-join as well as filtering of ground clauses in an efficient way because it takes full advantage of the parallel nature of optical information processing. Data stored in optical disks is retrieved and processed optically by a configuration involving two spatial light modulators and a large photodetector array where the photon-to-electron conversion takes place. The next step will be a more detailed analysis and simulation of the proposed architecture to define its true potential and limitations. Both single and multi-beam parallel read from the optical disks will be considered. Other parameters include the size and the framing speed of the modulators, the size of the relations and their records and the different algorithms for relational operations.

References

[ALT86] W.P. Altman, G.M. Claffie, M.L Levene, "Optical storage for high performance applications in the late 1980's and beyond", RCA Engineer Magazine, 31-1, January/February 1986.

[AND77] H.D. Anderson, P.B. Berra , "Minimum Cost Selection of Secondary Indexes for Formatted Files," ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, pp 68-90.

[ARM85] D. Armitage et al., "High-Speed Spatial Light Modulator", IEEE J. Quantum Electronics, Vol QE-21, Aug. 1985, pp 1241-1248.

[BAE87] R.A. Baeza-Yates, P.-A. Larson, "Analysis of B^+ - trees with Partial Expansions," Research Report CS-87-04, Department of Computer Science, University of Waterloo, Waterloo, Canada, February 1987, pp 20.

[BAY72] R. Bayer, E. McCreight, "Organization and Maintenance of Large Ordered Indexes," Acta Informatica by Springer-Verlag, Vol. 1, 1972, pp 173-189.

[BAY77] R. Bayer, K. Unterauer, "Prefix B-trees," ACM Transactions on Database Systems, Vol. 2, No. 1, March 1977, pp 11-26.

[BEL86] T. E. Bell, "Optical Computing: A field in flux", IEEE Spectrum, pp 34-57, August 1986.

[BER87a] P. B. Berra, S. M. Chung, N. I. Hachem, " Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base," IEEE Computer Vol. 20, No.3, March 1987, pp. 25-32.

[BER87b] P. B. Berra, N. Troullinos, "Optical Techniques and Data/Knowledge Base Machines", IEEE Computer, Vol. 20, Oct. 1987, pp 59-70.

[BER88] P. B. Berra, S. J. Marcinkowski, "Optical Content Addressable Memories for Managing an Index to a Very Large Data/Knowledge Base", Data Engineering Bulletin, Vol. 11, No 1, March 1988.

[BIT83] D. Bitten, H. Boral, et al., " Parallel Algorithms for Execution of Relational Database Operations," ACM Trans. Database Systems, Vol. 8, No. 3, 1983, pp. 324-

353.

[BLA77] M. W. Blasgen, K. P. Eswaran, " Storage and Access in Relational Data Bases," IBM Systems Journal, Vol. 16, No. 4, 1977, pp. 363-377.

[BLE78] W. P. Bleha et al., "Application of the Liquid Crystal Light Valve to Real-Time Optical Data Processing", Optical Engineering, Vol. 17, July-August 1978, pp 371-384.

[BRE86] K.H. Brenner, A. Huang, N. Streibl, "Digital optical computing with symbolic substitution", Applied Optics, Vol 25, 1986, pp 3054-3060.

[BUR83] W.A. Burkhard, "Interpolated-Based Index Maintenance," BIT 23, 1983, pp 274-294.

[BUR84] W.A. Burkhard, "Index Maintenance for Non-Uniform Record Distributions," ACM SIGACT-SIGMOD 3rd Symposium on Principles of Database Systems, Waterloo, Ontario, April 2-4 1984, pp 173-180.

[CAR75] A. F. Cardenas, " Analysis and Performance of Inverted Data Base Structures," CACM, Vol. 18, No. 5, 1975, pp. 253-263.

[CAR84] D.B. Carlin, J.P. Bednarz, C.J. Kaiser, J.C. Connolly, M.G. Harvey, "Multichannel optical recording using monolithic arrays of diode lasers", Applied Optics, vol 23, no 22, 14 November 1984, pp 3994-4000.

[CAR86] W. Carpenter, G. Herman, "Comparative Analysis of Online Data Storage Technologies", Tech. Report MTR-86W117, MITRE Corp., Oct. 1986.

[CAS77] D. Casasent, "Spatial Light Modulators", Proc. of IEEE, Vol. 65, Jan. 1977, pp 143-157.

[CHE86] P. P. Chen, "The Compact Disk ROM: How it Works", IEEE Spectrum, April 1986, pp 44-49.

[CHR84] S. Christodoulakis, "Implications of Certain Assumptions in Database Performance Evaluation", ACM Transactions on Database Systems, Vol. 9, No. 2, June 1984, pp 163-186.

[CHU88] S. M. Chung, P. B. Berra, " A Comparison of Concatenated and Superimposed Code Word Files for Very Large Data/Knowledge Bases," Proc. Int'l Conf. on Extending Database Technology, EDBT 88, Springer-Verlag, 1988, pp. 364-387.

[CLA86] K. Clark, S. Gregory, " PARLOG: Parallel Programming in Logic," ACM Trans. on Programming Languages and Systems, Vol. 8, No.1, January 1986, pp. 1-49

[COL86] R. M. Colomb, A Hardware-Intended Implementation of Prolog Featuring a General Solution to the Clause Indexing Problem, Ph. D. Dissertation, University of New South Wales, Australia, October 1986

[DAT86] C.J. Date, An Introduction to Database Systems, Volume 1, Addison-Wesley Systems Programming Series, 1986, Chapter 21.

[DAV86] W. A. Davis, D. L. Lee, " Fast Search Algorithms for Associative Memories," IEEE Trans. on Computers, Vol. 35, No. 5, 1986, pp. 456-461.

[DEW85] D. J. Dewitt, R. Gerber, " Multiprocessor Hash-based Join Algorithms," Proc. VLDB, 1985, pp. 151-164.

[DON73] J. Donjon et al., "A Pockels-Effect Light Valve: Phototitus - Applications to Optical Image Processing", IEEE Trans. on Electron Devices, Vol. 20, Nov. 1973, pp 1037-1042.

[FAG79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing- A Fast Access Method for Dynamic Files," ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pp 315-344.

[FED87] J. Fedorowicz, "Database Performance Evaluation in an Indexed File Environment," ACM Transactions on Database Systems, Vol. 12, No. 1, March 1987, pp 85-110.

[FEI72] J. Feinleib, D. S. Oliver, "Reusable Optical Image Storage and Processing Device", Applied Optics, Vol. 11, 1972, pp 2752-2759.

[FEI85] J. Feinberg, "Applications of Real-Time Holography", SPIE Proceedings, Vol. 532, January 1985, pp 119-136.

[FIS86] A. Fisher et al., "Photoemitter Membrane Light Modulator", Optical Engineering, Vol. 25, Feb. 1986, pp 261-268.

[GAR86] A. K. Garg, C. C. Gotlieb, " Order-Preserving Key Transformations," ACM Trans. Database Systems, Vol. 11, No. 2, 1986. pp. 214-234.

[GAY85] T.K. Gaylord, M.M. Mirsalehi, C.C. Guest, "Optical digital truth-table look-up processing," Optical Engineering, vol 24, January/February 1985, pp 48-58.

[GHO69] S.P. Ghosh, M.E. Senko, "File Organization: On the Selection of Random Access Index Points for Sequential Files", Journal of the ACM, Vo. 16, No. 4, October 1969, pp 569-579.

[GHO72] S.P. Ghosh, "File Organization: The Consecutive Retrieval Property", Communications of the ACM, Vol. 15, No. 9, September 1972, pp 802-808.

[GHO86] S.P. Ghosh, "Data Base Organization For Data Management", 2nd Edition, Computer Science and Applied Mathematics Series, Academic Press, 1986.

[GIB86] H. Gibbs, N. Peyghambarian, "Nonlinear Etalons and Optical Computing", SPIE 634, Optical and Hybrid Computing, March 1986, pp.142-148.

[GOO84] J. Goodman, F. Leonberger. S.Y.Kung, R.A. Athale, "Optical interconnections for VLSI systems," Proc. IEEE Vol. 72, July 1984, pp 850-866.

[GRI75] J. Grinberg et al., "A New Real-Time Non-Coherent to Coherent Light Image Converter - The Hybrid Field Effect Liquid Crystal Light Valve", Optical Engineering, Vol. 14, May-June 1975.

[GUI86] P. S. Guilfoyle, W. Jackson Wiley, "Combinatorial Logic Based Optical Computing," Proceedings SPIE, Vol 639-17, April 1986.

[HAC88a] N.I. Hachem,P.B. Berra, "Back End Architecture based on Transformed Inverted Lists, A Surrogate File Structure for a Very Large Data/Knowledge Base," Proc. 21st Hawaii International Conference on System Sciences, Vol. I, January 1988, pp 10-19.

[HAC88b] N.I. Hachem, "Key-Ordered File Structures for the Random and Sequential Access to Very Large Data/Knowledge Bases," Ph.D. Dissertation, Electrical and

Computer Engineering Department, Syracuse University, Syracuse, NY, June 1988.

[HAR86] T. Hara et al., "Microchannel Spatial Light Modulator Having the Functions of Image Zooming, Shifting and Rotating", Applied Optics, Vol. 25-14, 1986, pp 2306-2310.

[ITO87] H. Itoh, M. Abe, et al., " Parallel Control Techniques for Dedicated Relational Database Engines," Proc. Int'l Conf. on Data Engineering, 1987, pp. 208-215.

[JAK80] M. Jakobsson, "Reducing Block Accesses in Inverted Files by Partial Clustering," Information Systems, Vol. 5, 1980, pp 1-5.

[KIT84] M. Kitsuregawa, H. Tanaka, T. Moto-Oka, " Architecture and Performance of Relational Database Machine Grace," Proc. of Int'l Conf. on Parallel Processing, 1984, pp. 241-250.

[KIT87] M. Kitsuregawa, W. Yang, et al., " Design and Implementation of High Speed Pipeline Merge Sorter with Run Length Tuning Mechanism," Proc. 5th Int'l Workshop on Database Machines, 1987, pp. 144-157.

[KNI81] G. R. Knight, "Interface Devices and Memory Materials," in Optical Information Processing-Fundamentals, Topics in Applied Physics, Vol. 48, Springer-Verlag, 1981, pp 111-180.

[KOB78] H. Kobayashi, Modeling and Analysis, Chapter 3, Addison-Wesley, 1978.

[KRI87] H.-P. Kriegel, B. Seeger, "Multidimensional Dynamic Quantile Hashing is Very Efficient for Non-Uniform Record Distributions," Proceedings of the International Conference on Data Engineering, 1987, pp 10-17.

[LAR80a] P.-A. Larson, "Analysis of Repeated Hashing", BIT 20, 1980, pp 25-32.

[LAR80b] P.-A. Larson, "Linear Hashing with Partial Expansions," Proc. Sixth International Conference on Very Large Databases, 1980, pp 224-232.

[LAR81] P.-A. Larson, "Analysis of Index-Sequential Files with Overflow Chaining," ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, pp 671-680.

[LAR82a] P.-A. Larson, "A Single-File Version of Linear Hashing with Partial Expansions," Proc. 8th Conference on Very Large Data Bases, 1982, pp 300-309.

[LAR82b] P.-A. Larson, "Performance Analysis of Linear Hashing with Partial Expansions," ACM Transactions on Database Systems, Vol. 7, No. 4, December 1982, pp 566-587.

[LAR85] P.-A. Larson, "Linear Hashing with Overflow-Handling by Linear Probing," ACM Transactions on Database Systems, Vol. 10, No. 1, March 1985, pp 75-89.

[LAW75] D. H. Lawrie, " Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. 24, No. 12, 1975, pp.1145-1155.

[LEE86] S. H. Lee et al., "Two-Dimensional Silicon/PLZT Spatial Light Modulators: Design Considerations and Technology", Optical Engineering, Vol. 25, Feb.1986, pp 250-260.

[LEE88] S. H. Lee, "Computer Generated Holography", SPIE's O-E LASE '88, T68, January 1988.

[LIT80] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," Proc. Sixth International Conference on Vary Large Databases, 1980, pp 212-223.

[LIT81] W. Litwin, "Trie Hashing," Proceedings ACM SIGMOD Conference on Management of Data, Ann Harbor, Michigan, 1981, pp 19-29.

[LIT87] W. Litwin, D.B. Lomet, "A New Method for Fast Data Searches with Keys," IEEE Software, March 1987, pp 16-24.

[LOM81] D.B. Lomet, "Digital B-trees," Proceedings of the International Conference on Very Large Data Bases, France, Sept. 9-11, 1981, pp 333-344.

[LOM83a] D.B. Lomet, "Bounded Index Exponential Hashing," ACM Transactions on Database Systems, Vol. 8, No. 1, March 1983, pp 136-165.

[LOM83b] D.B. Lomet, "A High Performance, Universal, Key Associative Access Method," SIGMOD 83, San Jose, California, May 23-26, 1983, pp 120-133.

[LOM87] D.B. Lomet, "Partial Expansions for File Organizations with an Index," ACM Transactions on Database Systems, Vol. 12, No. 1, March 1987, pp 65-84.

[MEN86] J. Menon, " Sorting and Join Algorithms for Multiprocessor Database Machine," Database Machines: Modern Trends and Applications, Eds A.K. Sood, A.H. Qureshi, Springer-Verlag, 1986, pp.289-322.

[MOS87] Y. S. Abu-Mostafa, D. Psaltis, "Optical Neural Computers," Sci. Amer., March 1987, pp 88-95.

[MUN76] I. Munro, M.P. Spira, "Sorting and Searching in Multisets," SIAM Journal on Computing, Vol. 5, Nb. 1, March 76, pp 1-8.

[NIS72] P. Nisenson, S. Iwasa, "Real-Time Optical Processing with $Bi_{12}SiO_{20}$", Applied Optics, Vol. 11, 1972, pp 2760-

[ORE83] J.A. Orenstein, "A Dynamic Hash File for Random and Sequential Accessing," Proc. Ninth International Conference on Very Large Databases, 1983, pp 132-141.

[OZK86] E. Ozkarahan, Database Machines and Database Management, Prentice-Hall, Englewwod Cliffs, NJ, 1986.

[PEN86] W. Penn, "Liquid Crystals and Spatial Light Modulators", Tut. T24, SPIE 30-th ITS on Opt. and O-E ASE, August 1986.

[RAM78] C. V. Ramamoorthy, J. L. Turner, B. W. Wah, " A design of a Fast Cellular Associative Memory for Ordered Retrieval," IEEE Trans. on Computers, Vol. 27, No. 9, 1978, pp. 800-815.

[RAM84] K. Ramamohanarao, R. Sacks-Davis, "Recursive Linear Hashing," ACM Transactions on Database Systems, Vol. 9, No. 3, September 1984, pp 369-391.

[RAM86] K. Ramamohanarao, J. Shepherd, " A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," Proc. 3rd Int'l Logic Programming Conference, 1986, pp.569-576.

[REG88] M. Regnier, "Trie Hashing Analysis," Proceedings of the International Conference on Data Engineering, 1988, pp 377-381.

[REI72] E. M. Reingold, "On the Optimality of Some Set Algorithms," Journal of the ACM, Vol. 19, Nb. 4, October 72, pp 649-659.

[ROB86] J.T. Robinson, "Order Preserving Linear Hashing Using Dynamic Key Statistics," 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1986, pp 91-99.

[ROS83] W. E. Ross, D. Psaltis, R. H. Anderson, Optical Engineering, Vol. 22, 1983, pp 485-491.

[SAK86] H. Sakai, K. Iwata, et al., " Development of Delta as a First Step to a Knowledge Base Machine," Database Machines: Modern Trends and Applications, Edited by A. K. Sood and A. H. Qureshi, Springer-Verlag, 1986, pp. 159-181.

[SAL83] G. Salton, M.J. McGill, Introduction to Modern Information Retrieval, Computer Science Press, McGraw-Hill Book Company, 1983.

[SCH75a] M. Schkolnick, "Secondary Index Optimization," ACM SIGMOD Proceedings of the International Conference on Management of Data, May 14-16 1975, pp 186-192.

[SCH75b] M. Schkolnick, "The Optimal Selection of Secondary Indices for Files," Information Systems, Vol. 1, December 1975, pp 141-146.

[SIE80] H. J. Siegel, " The Theory Underlying the Partitioning of Permutation Networks," IEEE Trans. on Computers, Vol. 29, No. 9, 1980, pp. 791-800.

[SHI87] D. Shin, P. B. Berra, " An Architecture for Very Large Rule Bases Based on Surrogate Files," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.555-568.

[SMI81] P. W. Smith, W. J. Tomlinson, "Bistable Optical Devices promise subpicosecond switching," IEEE Spectrum. Vol. 8 , June 1981, pp 26-33.

[SOM72] L. Somers, "The Photoemitter Membrane Light Modulator Image Transducer", Advances in Electronics and Electron Physics, Vol 33a, Academic Press, New York, 1972.

[STI88] C. W. Stirk, R. A. Atale, "Sorting with Optical Compare-and-Exchange Modules", Applied Optics, Vol. 27-9, May 1988, pp 1721-1726.

[STO79] L.J. Stockmeyer, C.K. Wong, "On the Number of Comparisons to Find the Intersection of Two Relations," SIAM Journal on Computing, Vol. 8, No. 3, August 79, pp 388-404.

[TAK86] M. Takeda, J. W. Goodman, "Neural Networks for Computation: Number Representations and Programming Complexity", Applied Optics, Vol. 25-18, 1986, pp 3033-3046.

[TAM81] M. Tamminen, "Order Preserving Extendibie Hashing and Bucket Tries," BIT 21, 1981, pp 419-435.

[TAN83] A. R. Tanguay, C. S. Wu, P. Chaval, T. C. Strand, A. A. Sawchuck and B. H. Soffer, Optical Engineering, Vol. 22, 1983, pp 687-692.

[THA81] S. Thanawastien, V. P. Nelson, " Interference Analysis of Shuffle/Exchange Networks," IEEE Trans. on Computers, Vol. 30, No. 8, 1981, pp.545-556.

[TRI82] K. S. Trivedi, Probability and Statistics with Reliability, Queueing, and Computer Science Applications, Chapter 9, Prentice-Hall, 1982.

[VAL84] P. Valduriez, G. Gardarin, " Join and Semijoin Algorithms for Multiprocessor database Machine," ACM Trans. Database Systems, Vol. 9, No. 1, 1984, pp. 133-161.

[VAR87] A. Varma, " Rearrangeability of Multistage Sfuffle/Exchange Networks," Proc. Annual Symp. on Computer Architecture, 1987, pp.154-162.

[WAD87] M. Wada, Y. Morita, et al., " A Superimposed Code Scheme for Deductive Databases," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.569-582.

[WAK87] M. Warren, S. Koch, H. Gibbs, "Optical Bistability, Logic Gating, and Waveguide Operation in Semiconductor Etalons," IEEE Computer, Vol. 20-12, Dec. 1987, pp. 68-81.

[WAL87] K. Waldschmidt, " Associative Processors and Memories Overview and Current Status," Proc. COMPEURO 87, 1987, pp. 19-26.

[WAR81] C. Warde et al., "Optical Iformation Characteristics of the Microchannel Spatial Light Modulator", Applied Optics, Vol. 20, 1981, pp 2066-2074.

[WAR86] C. Warde, J. Kottas, "Hybrid Optical Inference Machines -- Architectural Considerations", Applied Optics, Vol. 25, pp. 940, 1986.

[WAR87] C. Warde, A. Fisher, "Spatial Light Modulators: Applications and Functional Capabilities", in Optical Signal Processing (edited by J. Horner), Academic Press, 1987, pp 478-524.

[WIS84] M. J. Wise, D. M. W. Powers, "Indexing Prolog Clauses via Superimposed Code Words and Field Encoded Words," Proc. 1984 Int'l Symposium on Logic Programming,

[YAO77] S.B. Yao, "Approximating Block Access in Database Organization," Communications of the ACM, Vol. 20, No. 4, April 77, pp 260-261.

List of Appendices

12-A A Relational Algebra Machine Based on Surrogate Files for Very Large Data/Knowledge Bases.

12-B Surrogate File Approach to Managing First Order Terms in Secondary Storage.

12-C Computer Architectures for Logic-Oriented Data/Knowledge Bases.

12-D Key-Sequential Access Methods for Very Large Files Derived from Linear Hashing.

12-E An Optical System for Full Text Search.

12-F An Optical Architecture for Performing Relational Data Base Operations.

12-G Implementing Knowledge Base Management Systems Based on Surrogate Files.

Appendix 12-A

A Relational Algebra Machine Based On Surrogate Files For Very Large Data/Knowledge Bases

Soon Myoung Chung
P. Bruce Berra

ECE Department
111 Link Hall
Syracuse University
Syracuse, NY 13244-1240
U.S.A.
(315) 443-4445

chung@sutcase.case.syr.edu
berra@sutcase.case.syr.edu

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

ABSTRACT

Concatenated code word (CCW) surrogate files are very useful as indices for very large knowledge bases to support logic programming inference mechanisms because of their small size and simple maintenance requirement. Moreover, interrelated relational algebra operations can be performed on the CCW surrogate files. In this paper a parallel backend database machine is proposed to speed up the relational algebra operations based on the CCW surrogate files. The basic idea of the proposed database machine is to reduce the amount of fact data to be transferred from the secondary storage systems to satisfy a query by performing the relational algebra operations on the CCW surrogate file first. The database machine consists of a number of surrogate file processors (SFP's) and extensional database processors (EDBP's) operating in SIMD mode. Each surrogate file processor has an associative memory to speed up the relational algebra operations on the CCW surrogate files. Surrogate file processors and extensional database processors are connected to other processors of the same type through multistage interconnection networks. The performance of the proposed system for parallel relational algebra operations was evaluated.

1. Introduction

Knowledge based systems consist of rules, facts and an inference mechanism that can be utilized to respond to queries posed by users. As these systems grow, increased demands will be placed on the management of their knowledge bases. The intensional database (IDB) of rules will become large and present a formidable management task in itself. But, the major management activity will be in the access, update and control of the extensional database (EDB) of facts because the EDB is likely to be much larger than the IDB. The volume of facts is expected to be in the gigabyte range, and we can expect to have general EDB's that serve multiple inference mechanisms. In this paper we assume that the IDB is a set of rules expressed as logic programming clauses and the EDB is a relational database of facts. Retrieving the desired rules and facts in this context is a partial match retrieval problem where any subset of attributes can be specified in a query and matching between terms consisting of variables and functions as well as constants should be tested as a preunification step.

In the context of very large knowledge bases the question arises as to how to obtain the desired rules and facts in the minimum amount of time. Two reasonable choices of indexing schemes to speed up the retrieval are concatenated code word (CCW) and superimposed code word (SCW) surrogate file techniques discussed in [BER87]. Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms.

Suppose we have a fact type called borders which is given as follows:

borders (Country_1, Country_2, Body_of_Water).

For a particular instance

borders (korea, china, yellow sea).

we would first hash the individual values to obtain binary representations.

H(korea)	=	100...01
H(china)	=	010...00
H(yellow sea)	=	110...00

Then the CCW of the fact is generated by simply concatenating the binary representations of all attribute values and attaching the unique identifier of the fact as follows:

100...01| 010...00| 110...00| 00...01.

The unique identifier is also attached to the fact and serves as a link between the two. It is used as a pointer to the EDB or can be converted to an actual pointer to the EDB by dynamic hashing schemes. A SCW is constructed by converting the binary representations to binary code words with pre-determined length and weight and bitwise logical ORing the binary code words [ROB79].

The retrieval process with the CCW surrogate file is as follows:

1) Given a query, obtain a query code word (QCW) by concatenating binary representations corresponding to the attribute values specified in the query. The portion of the query code word for the attribute values which is not specified in the query is filled with don't care symbols.

2) Obtain a list of unique identifiers to all facts whose CCW's satisfies

QCW=CCW

by comparing the QCW with all CCW's in the CCW file for that fact 'ype.

3) Retrieve all facts pointed to by the unique identifiers obtained in step 2 and compare the corresponding attribute values of the facts with the query values to discard the facts not satisfying the query. These are called "false drops". The facts satisfying the query are called "good drops". The false drops are caused by the non-ideal property of hashing functions.

4) Return the good drops.

Compared with other full indexing schemes such as inverted lists [CAR75], SCW and CCW surrogate file techniques yield much smaller amounts of index data; about 20% of the size of the EDB [BER87] while the inverted lists may be as large as the EDB. In terms of maintenance the surrogate file shows considerable advantages. When a new tuple is added to a relation the SCW or CCW is generated and added to the surrogate file. In the case of inverted lists each list must be processed. Similar operations must be performed for deleting tuples from a relation. When changes to an existing tuple are made, the surrogate file entry must be changed and the proper inverted lists must be changed.

An important advantage of SCW and CCW surrogate file techniques is that they can be easily extended for the indexing of the rules expressed as Prolog clauses, where the matching between constants, variables, and structured terms is required to test the unifiability. [RAM86] and [WAD87] extended the SCW structure for the indexing of Prolog clauses and [SHI87] extended the CCW structure to index the rules and facts in an unified manner.

An additional benefit obtained from using the CCW surrogate file approach is that relational operations can be performed on the CCW surrogate files [BER87]. To satisfy a query, interrelated relational algebra operations on the EDB are required, so by performing relational algebra operations on the CCW surrogate file first, considerable processing time can be saved.

A relational operation on CCW surrogate files is a kind of relational operation algorithm using indices [BLA77, MEN86]. However, using inverted list type indices in parallel relational algebra operations is very difficult, because the problem of synchronizing accesses to the indices without completely serializing the actions of the processors executing in parallel has not been solved yet [BIT83]. On the other hand, a CCW surrogate file is a set of transformed binary code words corresponding to the tuples of a relation, so it can be horizontally partitioned into subfiles and distributed over the parallel processors to be processed concurrently.

In [CHU88], CCW and SCW surrogate file techniques were analysed on the basis of storage space required for the surrogate file and time to retrieve the desired facts from the EDB. It is shown that the storage overhead and the query response time with the CCW surrogate file is smaller than those of the SCW surrogate file when the average number of attributes specified in a query is small. However, the analysis shows that most of the query response time for fact retrieval is used for the surrogate file processing when the relation is large (10^9 bytes) because of the sequential searching of all surrogate file code words. With smaller relations (10^6 bytes) surrogate file processing time is negligible compared with EDB processing time.

To speed up the relational algebra operations based on the CCW surrogate file, a backend database machine is proposed. The database machine consists of a number of surrogate file processors (SFP's) and EDB processors (EDBP's) operating in SIMD mode. Each SFP has an associative memory to speed up the relational algebra operations on the CCW surrogate files. Since CCW's are quite compact and regular, they are mapped well to the associative memories. SFP's and EDBP's are connected to other processors of the same type through multistage interconnection networks.

In section 2 the proposed architecture is introduced. The relational algebra operation algorithms for the architecture are explained in section 3. Section 4 shows the performance of the proposed architecture for relational algebra operations.

2. Backend Database Machine

A general structure of a backend database machine which contains multiple processors for the management of a very large extensional database of facts is shown in Figure 1. We assume that there are gigabytes of data stored on the EDB disk subsystems and there are corresponding CCW surrogate files stored on the SF disk subsystems. Suppose that the user is interested in retrieving fact data satisfying a condition from a particular relation. Then the selection query is transferred to the backend controller from the host computer and a query code word (QCW) is constructed in the surrogate file processor manager (SFPM) using the proper hashing functions. The QCW is then broadcasted to the proper Surrogate File Processors (SFP's) to be used as a search argument. The SFP compares the QCW with each CCW and strips off the unique identifiers of matching CCW's. Each extracted unique identifier is sent to the EDB processor manager (EDBPM) and passed on to the EDB processor (EDBP) which contains the fact with that unique identifier. The EDBP will access the block containing the fact, compare the retrieved fact with the original query to check that it is a good drop and then send it to the host computer.

The basic idea of the proposed backend database machine is to reduce the number of EDB blocks to be transferred from the secondary storage systems by performing the relational operations on the surrogate files first. To speed up the relational algebra operations on the surrogate files, each relation's surrogate file blocks are evenly distributed over a number of surrogate file disk subsystems so that the SFP's can process the surrogate files concurrently.

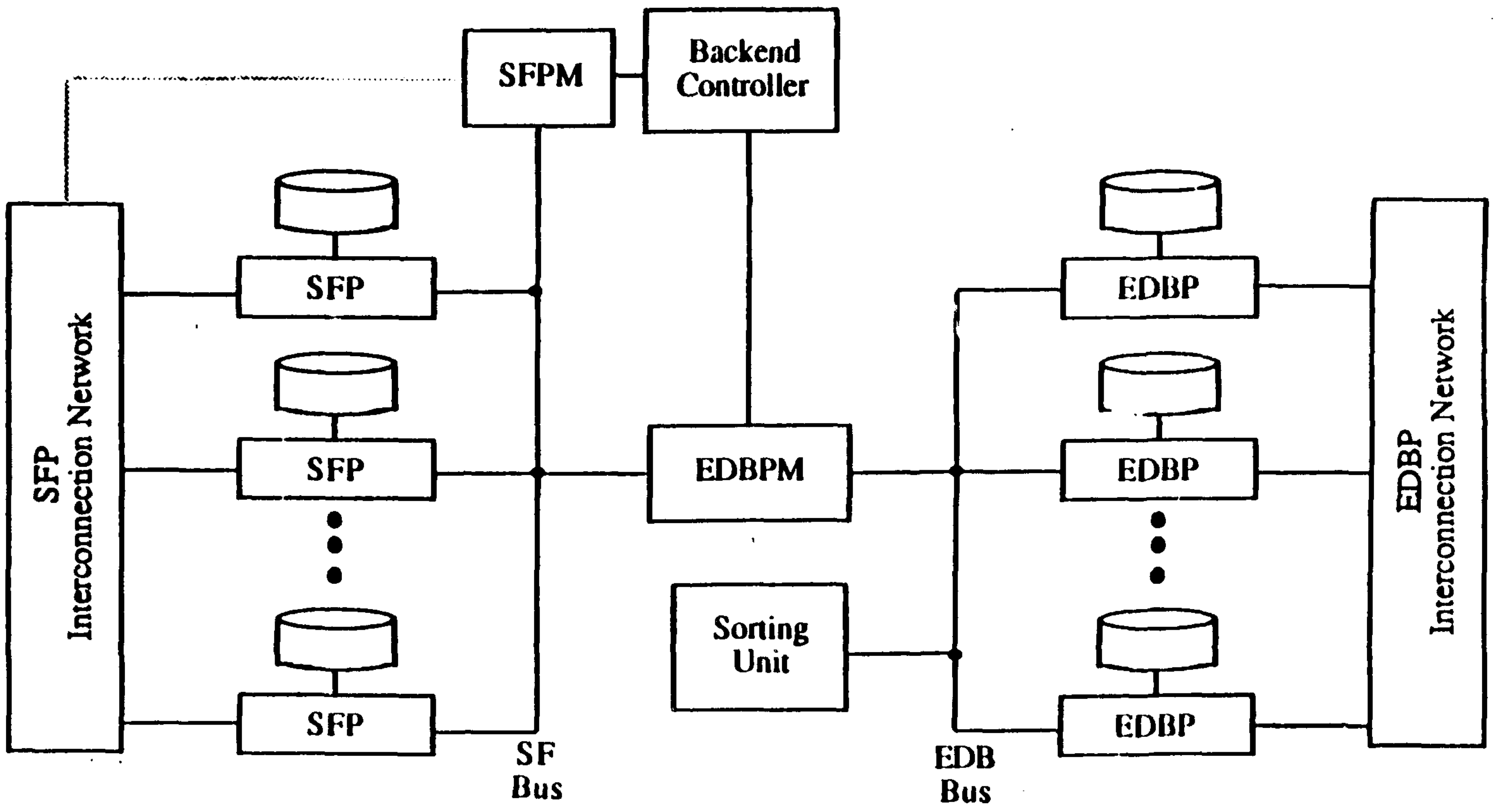


Figure 1. Relational Algebra Machine Based on Surrogate Files

As shown in Figure 2, a SFP is equipped with an associative memory unit to perform the searching operation efficiently. Associative memories are very fast because they use content addressing and parallel searching, but they are generally costly and rigid in data format. However, the format of the surrogate file is regular and maps very well into the associative memory and cost of the associative memory hardware is decreasing as VLSI technology advances. Additionally, associative memories can be used for relational operations, such as selection and join, because associative memories can perform many associative operations such as equal to, not equal to, less than (or equal to), greater than (or equal to), maximum, minimum, between limits, outside of limits, and others depending on the structure. In our design, we used word-parallel bit-serial (WPBS) associative memory which consists of two-dimensionally accessible memory and an array of processing elements. A word slice is a unit for memory read and write and a bit slice is a unit for arithmetic and logical processing. A bit-parallel associative memory [RAM78, DAV86], whose memory cells have comparison logic, is faster than a WPBS associative memory but is much more complex in structure and less flexible in functionality. Current status of associative memories and associative processors are reviewed in [WAL87].

To balance the speed of an associative memory, multiple disks controlled by two disk controllers constitute a surrogate file disk subsystem and are attached to a SFP through double buffers. In our system, one partition of surrogate file of a relation is stored consecutively within each disk subsystem so that the surrogate file blocks of a relation can be sequentially transferred to the associated surrogate file processors. By associating a disk subsystem to each surrogate file processor, we lose some flexibility in allocating processors to a query processing but surrogate file blocks can be accessed rapidly.

The surrogate file processors are connected through the SFP interconnection network. Since there are a number of surrogate file processors, the flexibility and speed of the inter-

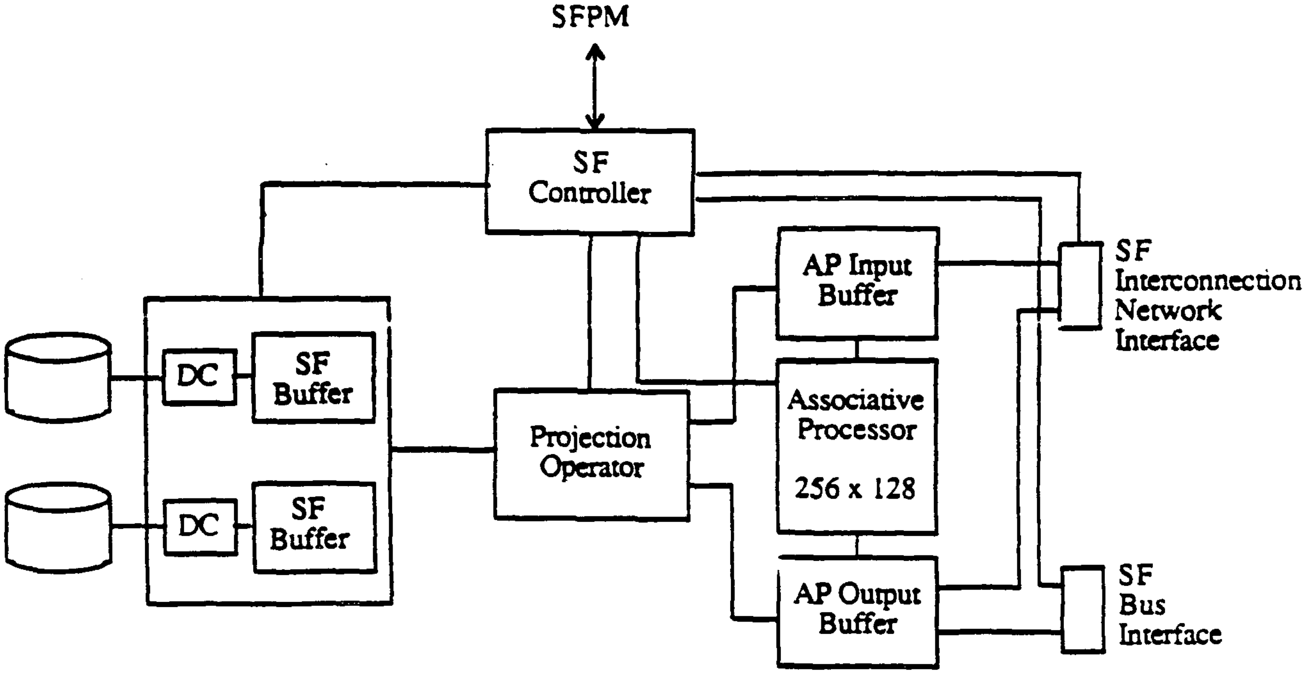


Figure 12.3.2. Structure of a Surrogate File Processor

connection are very important factors determining the overall performance. The mapping between SFP's will be permutation, selective broadcasting, or broadcasting depending on the distribution of operand surrogate files among the SFP's (we consider the pair of a SFP and a surrogate file disk subsystem as a single unit and call it a SFP), the number ot available SFP's, and algorithms cf relational algebra operations. To handle all the mapping modes we chose a multistage Omega network [LAW75] implemented with 2 by 2 switching elements with four functions; straight, exchange, upper broadcast, and lower broadcast. Thus, any one SFP can broadcast a block to the rest of the SFP's with uniform delay. The SFP interconnection network will be operating in circuit switching mode to facilitate the surrogate file block transfers.

The structure of the EDB processor manager is shown in Figure 3. If a fact unique identifier is sent from a SFP to the EDB processor manager (EDBPM), the EDBPM finds the EDB processor (EDBP) containing the corresponding fact by accessing a directory and sends the unique identifier to the EDBP. The EDBPM has a result buffer to collect the operation results from the EDBP's.

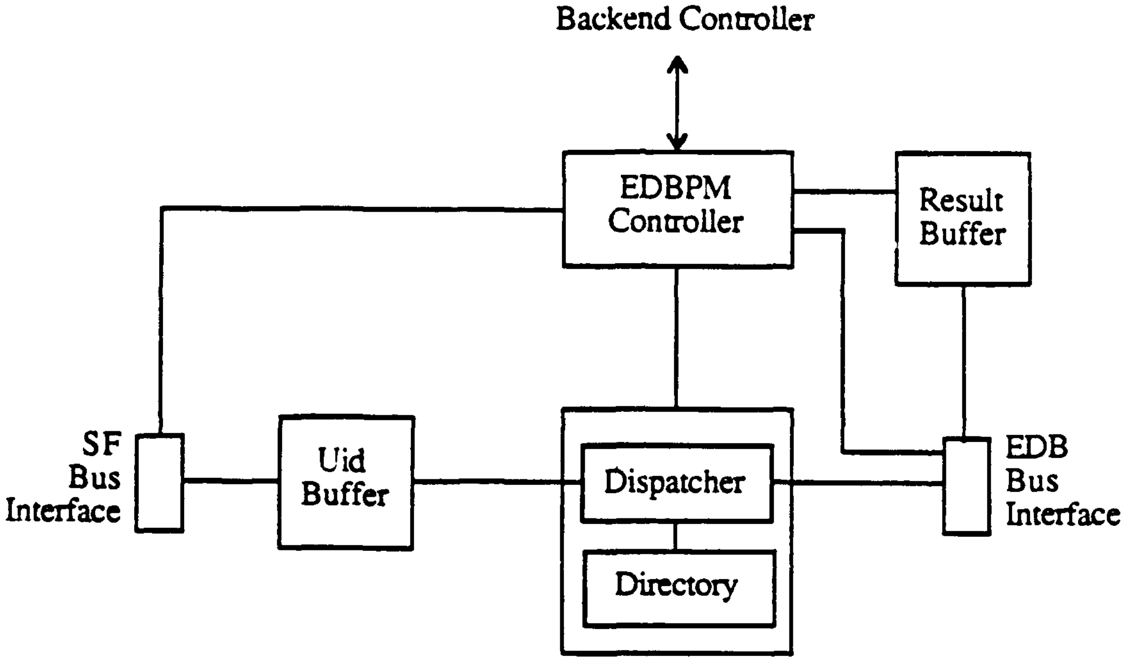


Figure 3. Structure of Extensional Database Processor Manager

The structure of a EDBP is shown in Figure 4. In case of fact retrieval, a fact block corresponding to the received unique identifier is accessed by the EDBP. We assume that EDB blocks are randomly distributed within a disk subsystem, so to speed up the block access a disk cache is provided per EDBP. Once the block is available in the working memory of the universal operator, the operator searches the block with the unique identifier, extracts the fact corresponding to the unique identifier, and compares the extracted fact with the query to check that it matches. The universal operator is a kind of general purpose processor and perform all the tuple-wise relational algebra operations as well as statistical aggregation functions. Through the EDBP interconnection network, facts can be transferred from one EDBP to another. We decided to use the multistage Omega network operating in packet switching mode to facilitate frequent transfer of facts between EDBP's in case of join operations.

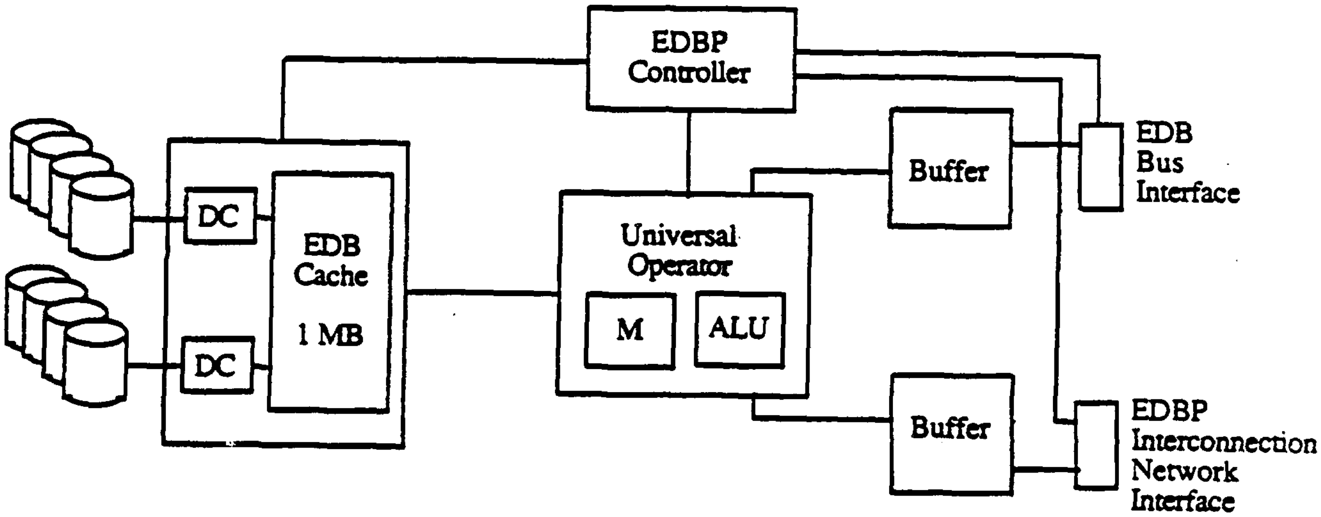


Figure 4. Structure of a Extensional Database Processor

A hardware sorting unit [KIT87] is available and can be accessed by a EDBP through the EDBP bus. The sorting unit can also be used for duplicate removal which is a part of other operations such as union, difference, and projection.

The processing mode of the backend system is SIMD or MIMD depending on the distribution of surrogate files and relations over the processors and assignment of the processors to a given operation. If all the processors are working for a single operation, then it becomes a

SIMD mode, but if processors are partitioned into a number of groups and each group of processors is assigned a different operation, then the processing mode is MIMD at the group level. To operate either in SIMD or MIMD mode, the interconnection network must be partitionable. A multistage Omega network of size 2^m can be partitioned into independent subnetworks of different sizes with the requirement that the addresses of all the I/O ports in a subnetwork of size 2^i agree in (m − i) of their bit positions [SIE80].

As the size of the EDB increases, the system can be upgraded by adding a cluster of SFP's and EDBP's to the existing system configuration. If we store the related relations and their surrogate files on a cluster of SFP's and EDBP's, then each cluster of processors are working on different queries and the processing mode of the system becomes multiple SIMD (MSIMD). In this case, the inter-cluster interconnection would be separated from the intra-cluster interconnections. The backend controller of each cluster would be a cluster controller and be in charge of communication with other clusters and the global backend controller through a cluster bus as shown in Figure 5.

3. Relational Algebra Operations in the Backend Database Machine

3.1. Selection Operation

To select on a particular attribute position, the SFP's execute a comparison, such as equal to, not equal to, greater than or equal to, or less than or equal to between the binary representation of a code word and the hashed value of the constant specified in a selection query. To retain the ordering between the binary representations of a attribute position, order-preserving hashing [GAR86] is necessary.

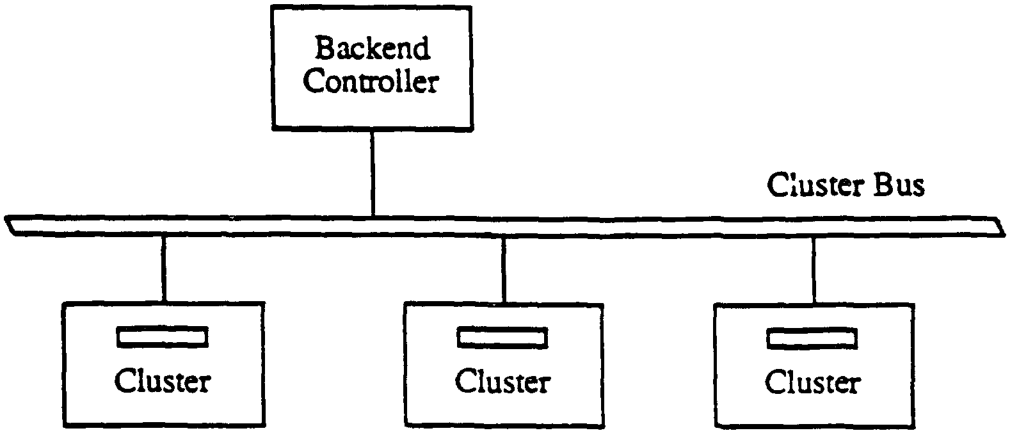


Figure 5. Multiple Relational Database Machine Configuration

Each SFP retrieves a block of CCW's and does the projection on the binary representation of the specified field and unique identifier, then loads the projected CCW's to the associative memory. The comparand register of the associative memory is loaded with the hashed constant. The bit positions of the input mask register corresponding to the hashed constant is filled with 1's while other bit positions are filled with 0's. If there is any match, the corresponding fact unique identifier is sent to the EDBPM.

As soon as any fact unique identifier is received by the EDBPM, it finds the EDBP containing the corresponding fact block and sends the unique identifier. The EDBP retrieves the fact block by using the unique identifier, searches the block with that unique identifier, and performs the actual selection operation on the fact. Due to the pre-selection operation on the surrogate file, the number of fact blocks to be accessed from the secondary storage system is usually very small compared to the total number of fact blocks of a relation.

3.2. Join Operation

There are three main algorithms for the join operation; sort-merge, hash-partition, and nested-loop join algorithms. The performance of the sort-merge join algorithm for the non-equijoin operation is as good as that for the equijoin operation, because once the two operand relations or subrelations are sorted, the merging step can handle the equijoin and the non-equijoin in the same way by performing the corresponding comparison operation. The database machine DELTA [SAK86, ITO87] has multiple relational database engines composed of sort-merge units and performs the sort-merge join algorithm. If a database machine has sort-merge units, the selection operation is interpreted as a join operation between a relation and a constant value specified in a query.

The hash-partition join algorithm is adopted by the database machine GRACE [KIT84]. Each operand relation is partitioned into a number of buckets depending on the hash value of the join attribute, then matching is performed within each bucket by a processor assigned to that bucket. Usually the hash-partition join algorithm is better than the sort-merge join algorithm in the case of the equijoin operation because sorting creates a total ordering of the tuples in both relations while the hashing simply groups related tuples together in the same bucket [DEW85]. However, in case of non-equijoin, the operation of each processor is not limited to a single bucket and the workload of the processors may not be uniform. One problem of the hash-partition join algorithm is the bucket overflow caused by the non-uniform distribution of the join attribute value. In this case, rehashing of the overflow bucket is necessary.

It has been shown that the nested-loop join algorithm takes advantage of different operand sizes and the processing time is inversely proportional to the number of processors, while in the case of the sort-merge algorithm after a certain number of processors, increasing the number of processors causes very little decrease in the execution time. The reason is that the degree of parallelism is divided by two at each merge pass after a certain stage [VAL84].

Our proposed database machine adopts the nested-loop join algorithm because the associative memories in each processor can easily perform the parallel execution of the nested-loop join operation.

If we assume that the CCW surrogate files of two operand relations are evenly distributed over a number of SFP's, the nested-loop join algorithm is executed as follows:

1) Each SFP reads a block of CCW surrogate file of the smaller relation from the associated surrogate file disk subsystem, projects it on the join attribute and unique identifier and loads it into the associative memory.

2) Each SFP reads a block of CCW of the larger relation from the associated surrogate file disk subsystem, project it on the join attribute and unique identifier and store it in the associative processor input buffer.

3) One SFP broadcasts the projected block from step 2) to the rest of the SFP's which already have a block of CCW surrogate file of the smaller relation in their associative memories from step 1).

4) Each SFP searchs the associative memory with the broadcasted projected CCW's as searching arguments one by one. If there is a match, the pair of unique identifiers of the two matched CCW's are sent to the EDBPM.

5) Repeat step 3) and step 4) until all the projected blocks in step 2) are broadcasted.

6) Repeat step 2) to step 5) until all the CCW's of the larger relation are retrieved from the surrogate file disk subsystems.

7) Repeat step 1) to step 6) until all the surrogate file blocks of the smaller relation are retrieved and searched.

In step 4), as soon as any unique identifier pair is received by the EDBPM, the EDBPM finds the EDBP's containing the corresponding facts and transfers the unique identifier pair to those EDBP's. If a single EDBP contains the two operand facts then that EDBP performs the actual join operation on the two facts retrieved, otherwise one EDBP transfers a projected fact to another EDBP containing the other operand fact, then the join is performed. To reduce the amount of communication through the EDB interconnection network the smaller projected fact is transferred. Projection is performed on the join attribute, attributes involved in the output relation, and the unique identifier. The pre-join operation of the SFP's is overlapped with the actual join operation of the EDBP's.

3.3. Projection Operation

Performing projection (including the duplicate removal) on the surrogate file as a substitute for projection on the realtion is not useful because of the false drops. Generally, the hashing function is not ideal, so different attribute values may have the same hashing function output. Therefore, we can not remove the duplicate binary representations of surrogate file code words. Thus, projection (including the duplicate removal) must be performed by the EDBP's on the actual relations. EDBP's can use an external sorting unit to remove the duplicate tuples, or can use a duplicate removal algorithm developed for multiprocessor system [BIT83] depending on the size of the relation to be projected, number of processors, and the size of the memory in each processor. Other relational operations such as set union and set difference have the same problem.

4. Performance An:'lysis of the Proposed Architecture

In this section, we analyse the performance of the proposed relational algebra machine for the selection and join operations. We used the parameter values specified in Table 1 and assumed that

1) The workloads of the SFP's and the EDBP's involved in a relationa' algebra operation are uniform.
2) Disk I/O operations and the processor' operations are executed concurrently whenever possible.
3) Pre-operations on the surrogate files by the SFP's and the actual operations on the facts by the EDBP's are executed concurrently whenever possible.

4.1. Selection Operation

In case of a selection operation, the response time is mainly determined by the size of surrogate file of a operand relation, number of SFP's and EDBP's involved in the operation, and the selectivity.

Figure 6 shows the total response time of a selection operation on a relation R as a function of the selectivity (defined as the ratio of the cardinality of the output relation to that of the operand relation), the number of SFP's (M), and the number of EDBP's (N) when

Parameter	Values
Average seek time of a disk	28 msec
Rotational delay of a disk	8.3 msec
Data transfer rate of a disk	2 MB/sec
Block size	4 KB
Effective EDB block access time	10 msec
Interconnection network speed	10 MB/sec
SF and EDB bus speed	50 MB/sec
Memory bandwidth	10 MB/sec
Unique id. dispatching time	10 μsec
Projection rate	6 MB/sec
Time for loading a word to an associative memory	0.1 μsec
Associative memory searching time for n bit-slices	(0.5 + 0.1 n) μsec
Time for extracting a responded word from the associative memory	0.2 μsec
Byte comparison time in EDBP	0.5 μsec

Table 1. Summary of Parameter Values Used for Performance Analyses

Cardinality of an operand relation R = 10^6

Size of a tuple = 100 bytes

Size of a unique identifier = 3 bytes

Size of a concatenated code word = 20 bytes

Size of a selection attribute = 15 bytes

Size of the binary representation of a selection attribute = 3 bytes

Size of an output tuple = 100 bytes

$$\alpha = \frac{\text{number of false drops}}{\text{number of good drops}} = 0.1 \ .$$

Since the hashing functions used to generate the CCW are not ideal, there are a certain number of false drops. We assumed that the total number of matched code words is (1 + α) times the actual number of facts satisfying a selection query.

When the selectivity is low, the pre-selection time on the surrogate file is dominant and the total response time will decrease as the number of SFP's increases. As the selectivity increases, the number of EDB blocks accessed will increase and the actual selection time on the facts is dominant. Thus, as the number of EDBP's increases the response time decreases linearly. Actually, as the selectivity increases the effective EDB block access time would be reduced due to an increased disk cache hit ratio. However, we assumed that the effective EDB block access time is constant in evaluating the total response time of a selection operation.

To reduce the number of EDB block accesses when the selectivity is high, we can search an accessed block with the search attribute values specified in the query, instead of searching the block with a unique identifier. In this way we can find all the desired facts in that block. Then, we store the block number in memory and whenever a unique identifier of a fact within that block is received, we discard the unique identifier since we already retrieved that fact.

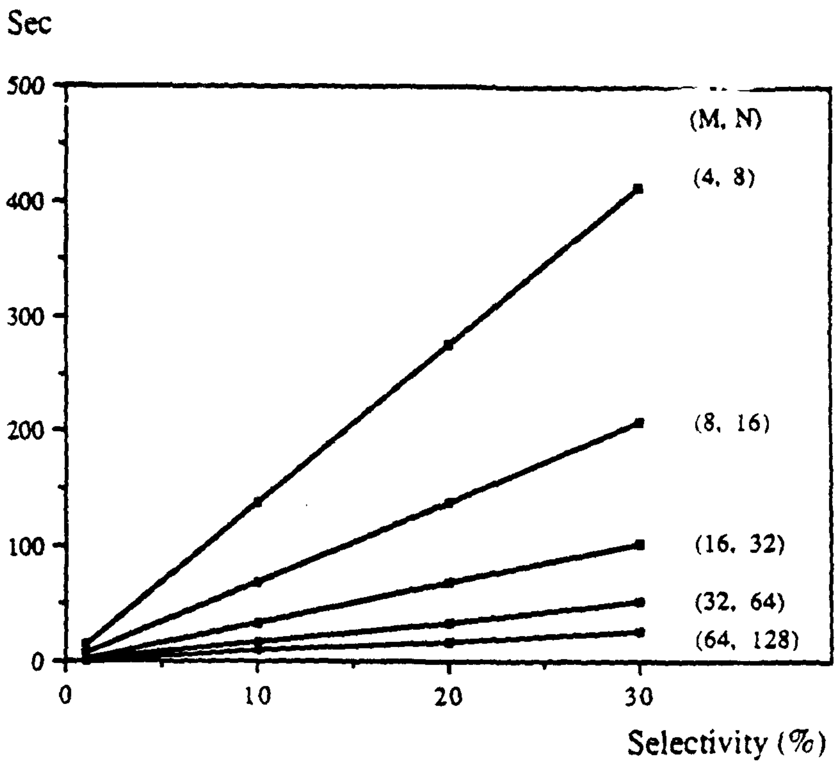


Figure 6. Performance of Selection Operation

(Cardinality of R = 10^6)

4.2. Join Operation

In the case of a join operation, the surrogate file size of the two operand relations, the number of SFP's and EDBP's involved, and the join selectivity will mainly determine the response time. Since a EDBP performs join operation on two operand facts, one of which may be transferred from other EDBP, we have to consider the network delay caused by the conflict in the network. However, usually the size of a projected fact is small, so the transfer time of a projected fact is very small compared to a EDB block access time. Therefore the network contention would not be serious unless the join selectivity is very high. Furthermore, the transferring of projected facts are overlapped with EDB block accesses and join operations. It has been shown that any permutation can be p^r^formed in a multistage Omega network within three passes of the network [VAR87], which corresponds to the analysis given in [THA81]. Thus, we simply assumed that the effective network speed is one third of the nominal network speed.

The response time of a join operation on two operand relation R_1 and R_2 is plotted in Figure 7 as a function of the cardinality of R_2, the number of SFP's (M), and the number of EDBP's (N) when

Cardinality of $R_1 = 10^6$

Cardinality of the output relation = cardinality of R_2

Size of a tuple in R_1 and R_2 = 100 bytes

Size of a unique identifier = 3 bytes

Size of a concatenated code word = 20 bytes

Size of a join attribute = 15 bytes

Size of the binary representation of a join attribute = 3 bytes

Contribution of each operand relation to an output tuple = 30 bytes

$$\beta = \frac{\text{Number of unique identifier pairs extracted}}{\text{Cardinality of output relation}} = 1.1 .$$

Due to the non-ideal hashing functions, the number of joinable code word pairs is larger than the cardinality of the output relation, and β accounts for this effect.

As the total size of the two operand relations increases, the response time increases. The pre-join time is dominant when the join selectivity is low since the pre-join operation is performed on every pair of surrogate file blocks while the actual join operation is performed on

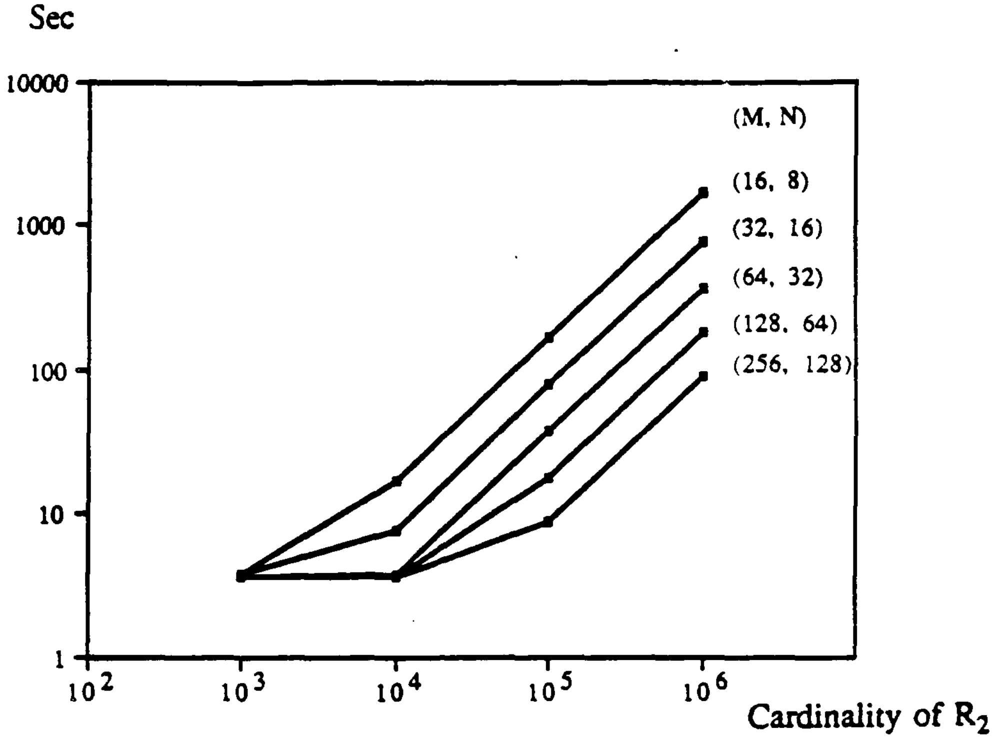


Figure 7. Performance of Join Operation between R_1 and R_2

(Cardinality of R_1 = 10^6, Cardinality of output relation = Cardinality of R_2)

the two operand facts. Therefore, when the selectivity is low, as we increase the number of SFP's we can decreases the total join processing time. When the join selectivity becomes high, actual join operation time is dominant due to the increased number of random EDB block accesses. In this case, we can reduce the number of EDB block accesses by storing the retrieved facts in the working memory and reuse it whenever it is requested. For an example of an equijoin operation, if the average number of tuples in R_1 (R_2) which have same join attribute value is C_1 (C_2), then a tuple of R_1 which is participated in the semijoin of R_1 by R_2 can be joined with C_2 tuples of R_2 on average. Thus, if we store that projected tuple of R_1 in the memory, we can reuse it ($C_2 - 1$) times later. For the same reason, the projected tuple of R_2 can be reused ($C_1 - 1$) times later. In Figure 7, we used a constant EDB block access time independent of the join selectivity.

5. Conclusion

In this paper, we have proposed a relational algebra machine based on the CCW surrogate files and analysed the performance of the architecture for parallel selection and join algorithms. The basic idea of the proposed architecture is to reduce the number of EDB blocks to be transferred from the secondary storage systems by performing the relational operations on the CCW surrogate file first. It has been shown that we can obtain an almost linear speedup in response time for a relational algebra operation by increasing the number of processors working concurrently. Our current research is focused on extending the surrogate file techniques to a deductive database system and developing a computer architecture for supporting the system.

References

[BER87] P. B. Berra, S. M. Chung, N. I. Hachem, " Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base," IEEE Computer Vol. 20, No.3, March 1987, pp. 25-32.

[BIT83] D. Bitten, H. Boral, et al., " Parallel Algorithms for Execution of Relational Database Operations," ACM Trans. Database Systems, Vol. 8, No. 3, 1983, pp. 324-353.

[BLA77] M. W. Blasgen, K. P. Eswaran, " Storage and Access in Relational Data Bases," IBM Systems Journal, Vol. 16, No. 4, 1977, pp. 363-377.

[CAR75] A. F. Cardenas, " Analysis and Performance of Inverted Data Base Structures," CACM, Vol. 18, No. 5, 1975, pp. 253-263.

[CHU88] S. M. Chung, P. B. Berra, " A Comparison of Concatenated and Superimposed Code Word Files for Very Large Data/Knowledge Bases," Proc. Int'l Conf. on Extending Database Technology, EDBT 88, Springer-Verlag, 1988, pp. 364-387.

[DAV86] W. A. Davis, D. L. Lee, " Fast Search Algorithms for Associative Memories," IEEE Trans. on Computers, Vol. 35, No. 5, 1986, pp. 456-461.

[DEW85] D. J. Dewitt, R. Gerber, " Multiprocessor Hash-based Join Algorithms," Proc. VLDB, 1985, pp. 151-164.

[GAR86] A. K. Garg, C. C. Gotlieb, " Order-Preserving Key Transformations," ACM Trans. Database Systems, Vol. 11, No. 2, 1986, pp. 214-234.

[ITO87] H. Itoh, M. Abe, et al., " Parallel Control Techniques for Dedicated Relational Database Engines," Proc. Int'l Conf. on Data Engineering, 1987, pp. 208-215.

[KIT84] M. Kitsuregawa, H. Tanaka, T. Moto-Oka, " Architecture and Performance of Relational Database Machine Grace," Proc. of Int'l Conf. on Parallel Processing, 1984, pp. 241-250.

[KIT87] M. Kitsuregawa, W. Yang, et al., " Design and Implementation of High Speed Pipeline Merge Sorter with Run Length Tuning Mechanism," Proc. 5th Int'l Workshop on Database Machines, 1987, pp. 144-157.

[LAW75] D. H. Lawrie, " Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. 24, No. 12, 1975, pp.1145-1155.

[MEN86] J. Menon, " Sorting and Join Algorithms for Multiprocessor Database Machine," Database Machines: Modern Trends and Applications, Eds A.K. Sood, A.H. Qureshi, Springer-Verlag, 1986, pp.289-322.

[RAM78] C. V. Ramamoorthy, J. L. Turner, et al., " A design of a Fast Cellular Associative Memory for Ordered Retrieval," IEEE Trans. on Computers, Vol. 27, No. 9, 1978, pp. 800-815.

[RAM86] K. Ramamohanarao, J. Shepherd, " A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," Proc. 3rd Int'l Logic Programming Conference, 1986, pp.569-576.

[ROB79] C. S. Roberts, " Partial Match Retrieval via the Method of Superimposed Codes," Proceedings of the IEEE, Vol. 67, No. 12, 1979, pp.1624-1642.

[SAK86] H. Sakai, K. Iwata, et al., " Development of Delta as a First Step to a Knowledge Base Machine," Database Machines: Modern Trends and Applications, Edited by A. K. Sood and A. H. Qureshi, Springer-Verlag, 1986, pp. 159-181.

[SIE80] H. J. Siegel, " The Theory Underlying the Partitioning of Permutation Networks," IEEE Trans. on Computers, Vol. 29, No. 9, 1980, pp. 791-800.

[SHI87] D. Shin, P. B. Berra, " An Architecture for Very Large Rule Bases Based on Surrogate Files," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.555-568.

[THA81] S. Thanawastien, V. P. Nelson, " Interference Analysis of Shuffle/Exchange Networks," IEEE Trans. on Computers, Vol. 30, No. 8, 1981, pp.545-556.

[VAL84] P. Valduriez, G. Gardarin, " Join and Semijoin Algorithms for Multiprocessor database Machine," ACM Trans. Database Systems, Vol. 9, No. 1, 1984, pp. 133-161.

[VAR87] A. Varma, " Rearrangeability of Multistage Shuffle/Exchange Networks," Proc. Annual Symp. on Computer Architecture, 1987, pp.154-162.

[WAD87] M. Wada, Y. Morita, et al., " A Superimposed Code Scheme for Deductive Databases," Proc. 5th Int'l Workshop on Database Machines, 1987, pp.569-582.

[WAL87] K. Waldschmidt, " Associative Processors and Memories Overview and Current Status," Proc. COMPEURO 87, 1987, pp. 19-26.

Appendix 12-B

Surrogate file approach to managing first order terms in secondary storage [1]

Donghoon Shin
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244, USA

P. Bruce Berra
Dept. of Computer and Electrical Engineering
Syracuse University
Syracuse, NY 13244, USA

January 14, 1989

[1] This work was supported by the Air Force Systems Command, Rome Air Development Center at Griffiss Air Force Base in New York, and the Air Force Office of Scientific Research at Bolling AFB in Washington DC under Contract No. F30602-85-C-008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Abstract

This paper concerns the efficient management of very large logic programs stored in secondary storage by proposing a physical data organization scheme called an *extended concatenated code word* (ECCW) which is based on the *surrogate file* concept.[1] The ECCW can be constructed by concatenating transformed code words obtained from the arguments. Associated with each code word are two fields; a tag field and a value field. The tag field can represent any argument type including lists and structured terms as well as variables and constants. The value field contains the transformed representation of the corresponding argument according to the content of its tag field. The ECCW uses several storage encoding techniques: *multilevel coding* to represent nested structures by using normalizing storage model, *tagged coding* to discriminate attribute types such as variables, lists, complex terms, and constants, *storage partitioning* and *tag collection* to reduce search space.

1 Introduction

As computer applications become larger and more complex, functionalities of database management systems (DBMSs) should be extended to deal with diverse environments. In this context, the need for supporting a data model with the ability to represent arbitrary complex objects without restrictions on structure is widely recognized. Examples of recent database systems requiring this ability include object - oriented database systems, knowledge base management systems, and multimedia database systems. It is also clear that the DBMS supporting complex objects requires a more sophisticated internal storage scheme since the I/O bottleneck problem becomes worse.

Among the various research issues raised by logic-oriented knowledge base systems, our study is focused on the issue of physical data organization of complex objects stored on secondary storage. The main purpose of this study is to provide logic-oriented knowledge base systems with a universal data organization scheme so that basic operations required for processing complex objects can be well supported on the data organization.

Our basic approach to solving this problem is the use of surrogate files.[1] Surrogate files are a class of storage model designed for highly efficient secondary storage accesses to very large data/knowledge bases, which can be viewed as a compact image of the actual knowledge bases primarily based on hashing methods. By first processing surrogate files and delaying actual knowledge base accesses until a set of relevant pointers to the actual knowledge base is obtained, the performance of the knowledge base systems can be considerably enhanced.

It is well known that the entire system performance of most data intensive applications is dominated by secondary storage accesses, that is, by loading data from secondary storage to a processing level. Due to the dramatic performance improvements of current parallel computer architectures, this I/O bottleneck problem becomes more serious. Thus, it is important for a data organization scheme to perform well in parallel computers as well as in conventional computers, as the traditional file structures do not have this property. The uniform structure of surrogate files offers the promise of an order of magnitude speed up for parallel computing. But most surrogate file schemes proposed so far do not support operations on complex objects, and still require a large amount of I/O time to load surrogate files especially for a sequential computer architecture.

In this paper, we propose a new storage model called Extended Concatenated Code Word (ECCW). The ECCW uses several storage encoding techniques: *multilevel coding* to represent nested structures by using a normalized storage model, *tagged coding* to discriminate attribute types such as variables, lists, complex terms, and constants, *storage partitioning* and *tag collection* to reduce search space. We are aiming at performing diverse operations in both sequential and parallel processing environments while minimizing space overheads.

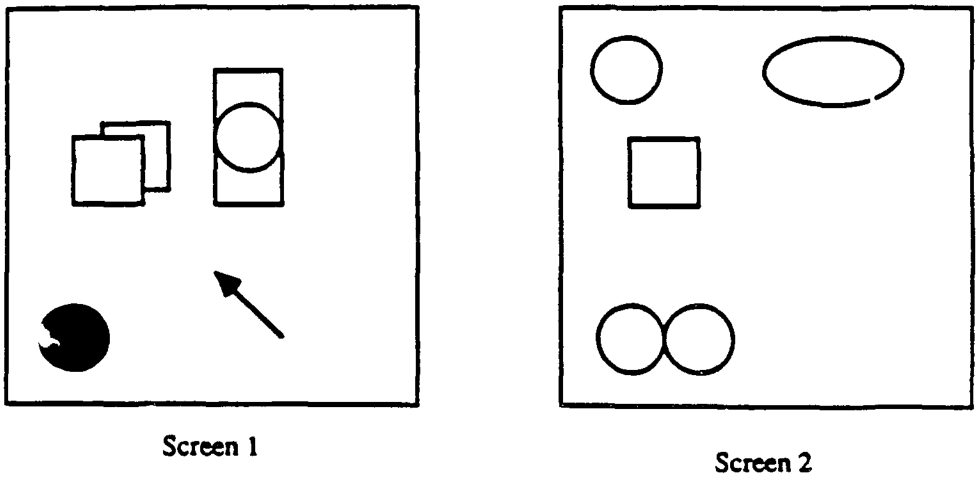
This paper is organized as follows: In the next section, database models supporting complex objects are described. Section 3 introduces the basic principle of surrogate files by describing various Concatenated Code Word (CCW) schemes proposed for logic-oriented knowledge base systems. The details of the ECCW is discussed in section 4. We conclude in section 5, and discuss the use of ECCW for various operations as future works.

2 Complex Objects in Databasᵉ Systems

The classical form of database system such as the relational model was designed to handle an important but limited class of applications such as financial records or inventories. The common characteristic of such applications is that they have large amount of data, but the operations to be performed on the data are simple. For efficient storage management, traditional database systems are based on records of uniform structures called first normal form (1NF) records defined by Data Definition Languages (DDLs). An obvious disadvantage of using DDL is that for an attribute having variable length records, the expected maximum length of the records should be assigned to the attribute, resulting in considerable storage waste. However, the more serious problem of the 1NF databases stems from its limitation in modeling real world information. To illustrate this point, suppose we wish to use a relational database system to store visual images given in Figure 1.

In the figure, four types of drawing objects - circle, oval, rectangle, and vector - are presented. For each object type, we need information on the type of drawing object, X and Y coordinate of the origin, color to fill in the object (except for vector objects), and the line type to be used in drawing the object. For the circle object, we need only one parameter (diameter), while for the oval, rectangle, and vector object we need two parameters to represent the shape of the object. Thus, in order to use the relational model, we need the following relations.

DRAW(SCREEN#, OBJECT#, X-LOCATION, Y-LOCATION).
CIRCLE(OBJECT#, DIAMETER, COLOR, LINE).
RECTANGLE(OBJECT#, V-LENGTH, H-LENGTH, COLOR, LINE).


Screen 1

Screen 2

Figure 1: Screens in a drawing database

OVAL(OBJECT#, V-DIAMETER, H-DIAMETER, COLOR, LINE).
VECTOR(OBJECT#, DEGREE, LENGTH, LINE).

The major problems using the above scheme are that the OBJECT# in each relation must be explicitly specified by user and it requires N + 1 relations to represent N types of drawing objects. The second problem is more serious since a drawing object can be arbitrarily complex. For example, to represent a general polygon N parameters are required.

The drawing database can be represented by one relation if complex objects are allowed as attribute values:

DRAW-NF(SCREEN#, SHAPE, [PARAMETERS], X-LOCATION, Y-LOCATION).

Or, when a large number of repeated patterns are expected,

DRAW-NF'(SHAPE, [PARAMETERS], {(SCREEN#, X-LOCATION, Y-LOCATION)})

where the attribute enclosed by a pair of curly brackets represents a set attribute (Figure 2). The above relations that allow complex objects are called *non first normal form* (NFNF) relations.[2]

For a more formal definition of complex objects, assume that a set of attribute names and a collection of atomic data types such as integers, float, and strings are given. Then, (complex) objects[3] are defined as follows:[*]

[0*] The list object was not defined by Bancilhon and Khoshafian[3]. It can be viewed as a generalized set object.

DRAW-NF

Screen#	Shape	[Parameters]	X-location	Y-location
1	circle	[10, black, 1]	10	10
1	circle	[10, none, 1]	30	40
1	rectangle	[10, 10, none, 1]	10	10
1	rectangle	[10, 10, none, 1]	13	33
1	rectangle	[25, 10, none, 1]	30	30
1	vector	[135, 15, 1]	40	10
2	circle	[10, none, 1]	10	10
2	circle	[10, none, 1]	20	10
2	circle	[10, none, 1]	10	50
2	rectangle	[10, 10, none, 1]	10	30
2	oval	[10, 20, none, 1]	30	50

DRAW-NF'

		{ Location}		
Shape	[Parameters]	Screen#	X-location	Y-location
circle	[10, black, 1]	1	10	10
circle	[10, none, 1]	1	30	40
		2	10	10
		2	20	10
		2	10	50
rectangle	[10, 10, none, 1]	1	10	30
		1	13	33
		2	10	30
rectangle	[25, 10, none, 1]	1	30	30
oval	[10, 20, none, 1]	2	30	50
vector	[135, 15, 1]	1	40	10

Figure 2: Non first normal form relation for the drawing database

1. Atomic data elements are objects.

2. If $O_1, O_2, \ldots, O_n$ are objects and $a_1, a_2, \ldots, a_n$ are attribute names, then $O =< a_1 : O_1, a_2 : O_2, \ldots, a_n : O_n >$ is an (tuple) object.

3. If $O_1, O_2, \ldots, O_n$ are objects then $\{O_1, O_2, \ldots, O_n\}$ is an (set) object. An ordered set object is called a list object and represented by $[O_1, O_2, \ldots, O_n]$.

The drawing application needs considerably more powerful data operations than conventional business applications. In this context, the integration of the data manipulation and host languages becomes important. Zaniolo[4,5] indicated that by introducing logic to database systems, the database system can gain two important functionalities, namely the ability to handle complex objects and powerful rule supports including recursive views. The database system with such functionalities is called *logic-oriented knowledge base system.*

The basic data elements that logic-oriented knowledge bases deal with are *terms.* A *term* is defined as follows:

1. A variable is a term denoted by a capital letter such as X, Y, Z, ...

2. A constant is a term denoted by a lower case letter such as a, b, ...

3. If f is an n-ary function and $t_1, \cdots, t_n$ are terms, then f$(t_1, \cdots, t_n)$ is a term.

The syntax of the term subsumes that of the complex object. For example, the drawing database defined by the NFNF database can be represented by terms as shown in Figure 3.

Since the list object can be viewed as a binary function, a polygon with n edges can be represented by the term given below.

$$\text{polygon}([(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]).$$

In addition, by using rules and logical variables, more complex objects can be constructed. Consider, for example, the construction of a "disk" type shown in Figure 4. Suppose that an object type is of the form

object-type([Parameters], X-location, Y-location, Color, Linetype)

and a built-in predicate "group" is used to construct a new object. Then, the disk can be recognized by the following rule:

```
draw(1, circle(10, black, 1), 10, 10).
draw(1, circle(10, none, 1), 30, 40).
draw(1, rectangle(10, 10, none, 1), 10, 30).
draw(1, rectangle(10, 10, none, 1), 13, 33).
draw(1, rectangle(25, 10, none, 1), 30, 30).
draw(1, vector(135, 15, 1), 40, 10).
draw(2, circle(10, none, 1), 10, 10).
draw(2, circle(10, none, 1), 20, 10).
draw(2, circle(10, black, 1), 10, 50).
draw(2, rectangle(10, 10, none, 1), 10, 30).
draw(2, oval(10, 20, none, 1), 30, 50).
```

(a) Conversion from the DRAW-NF

```
draw'(circle(10, black, 1), [(1,10,10)]).
draw'(circle(10, none, 1), [(1,30,40), (2,10,10), (2,20,10), (2,10,50)]).
draw'(rectangle(10, 10, none, 1), [(1,10,30), (1,13,33), (2,10,30)]).
draw'(rectangle(25, 10, none, 1), [(1,30,30)]).
draw'(oval(10, 20, none, 1), [(2,30,50)]).
draw'(vector(135, 15, 1), [(1,40,10)]).
```

(b) Conversion from the DRAW-NF'

Figure 3: Logic-oriented knowledge base for the drawing database

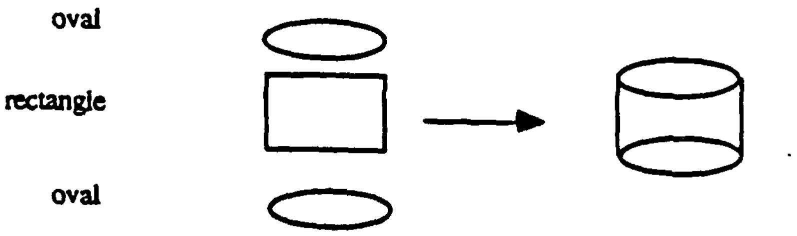


Figure 4: Disk object

```
disk([V-length,H-length], X-location, Y-location, Color, Linetype):-
    group([rectangle([V-length,H-length], X-location, Y-location,Color, Linetype),
          oval([X, H-length], X1, Y1, Color, Linetype),
          oval([X, H-length], X1, Y-location, Color, Linetype)]),
    X1 = X-location + H-length/2, Y1 = Y-location + V-length.
```

From the examples given above, we have showed that the logic-oriented knowledge base system is powerful enough to handle most classes of complex objects in a uniform way.

3 Basic Surrogate File Schemes

Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms. Examples of surrogate file schemes developed so far include Superimposed Code Word (SCW), Concatenated Code Word (CCW), and Transformed Inverted List (TIL). In this section we describe the construction of CCW.

The CCW of a tuple is generated by simply concatenating the binary representations (BR's) of all attribute values and attaching the unique identifier of the tuple. Consider, for example, p(a, b, c) and assume that the BR's for a, b, and c are 0100, 1100, and 1010 respectively. The corresponding CCW is

0100 | 1100 | 1010 | UID

The argument positions not specified in the query (i.e. variables) should be represented by don't care match indicators. The QCW for ← p(a,X,c) can be obtained by replacing the second argument position with don't care match indicators.

0100 | xxxx | 1010

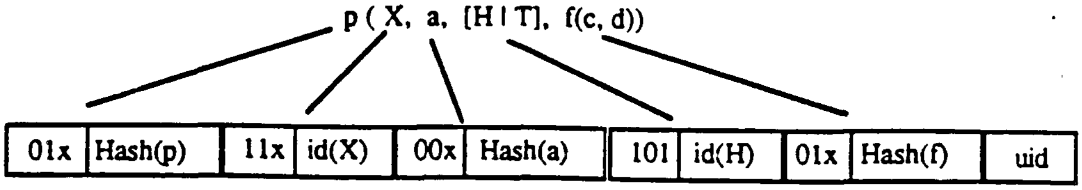


Figure 5: Extending CCW to general terms (CCW-2)

The matching condition is CCW = QCW provided that x (don't care match indicators) can match with both 1 and 0. A clear advantage of the CCW over SCW is that we can perform relational operations such as Join on the surrogate file itself rather than on the actual extensional database (EDB).

In the standard form of PARLOG[6], no structured term appears in the head of clauses. Thus, in this context, only pure variables and constants should be considered. We proposed a CCW method[7] (CCW-1) to handle the standard form of clauses. In this scheme, each CCW code corresponding to an argument has an one bit tag to indicate whether the argument is a variable. The tag bit is used for bidirectional don't care matches as a preliminary step of unification. CCW-1 provides an efficient mechanism in searching possible candidate clauses as well as in detecting binding conflicts among shared variables in early stages of execution. However, since this scheme is based on guarded Horn clauses and mode declarations, its application is somewhat limited to the parallel logic programming paradigm.

CCW-2[8] is basically the same structure as CCW-1, which can be constructed by concatenating transformed code words obtained from the arguments along with the predicate name of the head of a clause. Each code word is divided into two fields; tag field and value field. Unlike CCW-1, however, the tag field can represent any argument types including lists and structured terms as well as variables and constants. The value field contains the transformed representation of the corresponding argument according to the content of its tag field. For example, if the tag indicates structured term, then the value field contains the hashed value of the primary functor, while if the tag is for a variable, the corresponding value field represents the variable identification number. This scheme can be viewed as an augmentation of CCW-1 with the indexing scheme used in Warren's Abstract Prolog Instruction Set[9] where only the first argument is indexed. Table 1 shows the coding method of the CCW-2 scheme. An example of CCW-2 encoding is shown in Figure 5.

In contrast to CCW-1, CCW-2 can be used for current Prolog systems and does not

Argument Type	Tag Field	Contents of Value Field
Constant	00x	Hashed Value of the constant
Function	01x	Hashed Value of the Primary Functor/Arity
List	100	Hashed Value of the CAR constant
	101	Variable ID for the CAR variable
Variable	11x	Variable ID

Table 1: CCW-2 coding scheme

require mode declarations. Due to the type checking mechanism, false drops can be considerably reduced without sacrificing the compactness and uniformity of CCW. In addition, the uniform and compact data structure of CCW's allows the effective use of specialized hardwares such as associative memories.[7,8]

4 Extended Concatenated Code Word

The surrogate files proposed so far have fundamental limitations in handling complex objects. The SCW schemes[10,11] for complex objects cannot effectively support operations such as join. Furthermore, only very coarse clustering can be used for SCW's. For example, when 20-in-1 coding and transposed organization are used, 5 % of record descriptors need to be accessed. One of the distinctive features of the CCW scheme is its ability to perform relational operations without accessing actual data/knowledge bases. The previous CCW schemes, CCW-1 and CCW-2, however, do not support diverse operations for complex objects, although they can be used in searching. Furthermore, the previous CCW schemes lack the ability to support complex searching operations based on subarguments.

The surrogate files generally perform well in a parallel processing environment. One of the interesting examples is the use of SCW in document retrieval for the massively parallel Connection Machine.[12] One of the drawbacks of the surrogate file scheme (both SCW and CCW) is that every tuple descriptor should be compared to the given QCW, and thus its entire descriptor file should be retrieved from secondary storage. Although the size of descriptor file is small (about 20 % of the original file size[1]), loading entire descriptor files to the main memory. Stone[13] indicated that the use of one-level SCW for highly parallel machines can incur a serious I/O problem since all record descriptors should be loaded to data processors. He argued that using a cleaver indexing scheme for a sequential machine

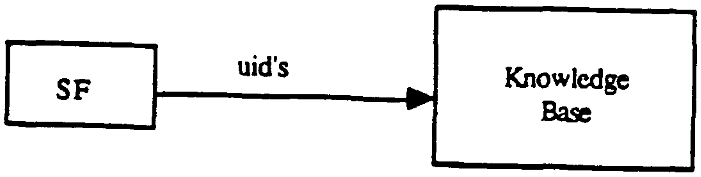


Figure 6: Top level organization of surrogate files

```
draw(uid1, 1, uid11, 10, 10).     circle(uid1, 10, black, 1).
draw(uid2, 1, uid21, 30, 40).     circle(uid2, 10, none, 1).
draw(uid3, 1, uid31, 10, 30).     rectangle(uid3, 10, 10, none, 1).
draw(uid4, 1, uid41, 13, 33).     rectangle(uid4, 10, 10, none, 1).
draw(uid5, 1, uid51, 30, 30).     rectangle(uid5, 25, 10, none, 1).
draw(uid6, 1, uid61, 40, 10).     vector(uid6, 135, 15, 1).
draw(uid7, 2, uid71, 10, 10).     circle(uid7, 10, none, 1).
draw(uid8, 2, uid81, 20, 10).     circle(uid8, 10, none, 1).
draw(uid9, 2, uid91, 10, 50).     circle(uid9, 10, none, 1).
draw(uid10, 2, uid101, 10, 30).   rectangle(uid10, 10, 10, none, 1).
draw(uid11, 2, uid111, 30, 50).   oval(uid11, 10, 20, none, 1).
```

Figure 7: Normalized complex objects

can outperform massive parallelism.

This section is concerned with the design of the Extended Concatenated Code Word (ECCW) to solve the problems of the previous surrogate files. Our surrogate file scheme is based on two heterogeneous storage models as shown in Figure 6. The surrogate file part (ECCW) is based on the normalized storage model so that partial match retrieval can be effectively supported. On the other hand, the actual knowledge base supports the direct storage model where an entire object can be easily accessed.

4.1 Normalization of complex objects

The first step in constructing ECCW codes is to transform complex objects to normalized forms. In the normalized form, every complex subobject is represented by a lower level "flat" object. The original position of the subobject is substituted by a unique identifier. Figure 7 shows the normalized complex objects for the drawing database given in Figure 3.

Argument Type	Tag Field	Contents of Value Field
Constant	00	Hashed Value of the constant
Nested Relation	01	Unique ID for the subrelation
List	10	Unique ID for the List
Variable	11	Variable ID

Table 2: ECCW coding scheme

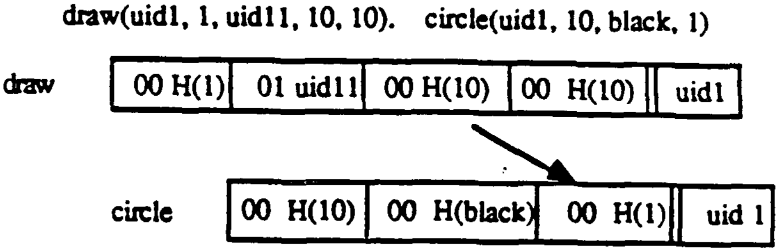


Figure 8: The logical representation of a complex object in ECCW

4.2 Tagged coding

The normalized "draw" relation has level number 0, representing an entire object. On the other hand, the relations such as "circle", "rectangle", "oval", and "vector" are regarded as objects at level 1. Since all the level 1 objects are the third arguments of the level 0 relation ("draw"), their addresses are 3 with respect to the level 0 relation. For each normalized form of objects, an ECCW code can be constructed by concatenating tagged binary descriptions of all the attributes. We use two bit tags to represent four different types of attribute values. Table 2 shows the tags and the corresponding values. This tagging scheme is a modification of the one developed for the CCW-2 scheme. Each object can be logically connected via the uid's as shown in Figure 8.

4.3 Directory structure of the ECCW

In order to reduce the search space for a given query, various techniques are explored. Note that, if an argument of a query is an atomic constant, the corresponding arguments of the qualified objects should also be constants. On the other hand, if the argument has a complex object type such as a list or a nested relation, then the corresponding argument type should be matched with that of the corresponding ECCW code. Since variables can be considered as "don't care" match indicators, the variables in ECCW should be considered for every query.

Thus, for a normalized relation with N arguments, we can partition the relation into 4^N subrelations by collecting tags of all attributes in a place. Consider, for example, a relation with three arguments. Since each attribute at level 1 can have 4 types, $4^3 = 64$ different collections of tags from 000000 to 111111 can be used for clustering. In the best case, when data are uniformly distributed without variables in ECCW, only $1/\ 3^3 = 3.7$ % of surrogate files need to be accessed. In the worst case, however, when every argument is a constant and the query is for exact match, all the ECCW codes need to be searched. Thus, we propose to use a data space partitioning scheme such as the grid file for the ECCW so that we can retrieve a desired code word with small number of disk accesses.

The overall organization of the ECCW is shown in Figure 9. As seen in the figure, the ECCW consists of three basic components:

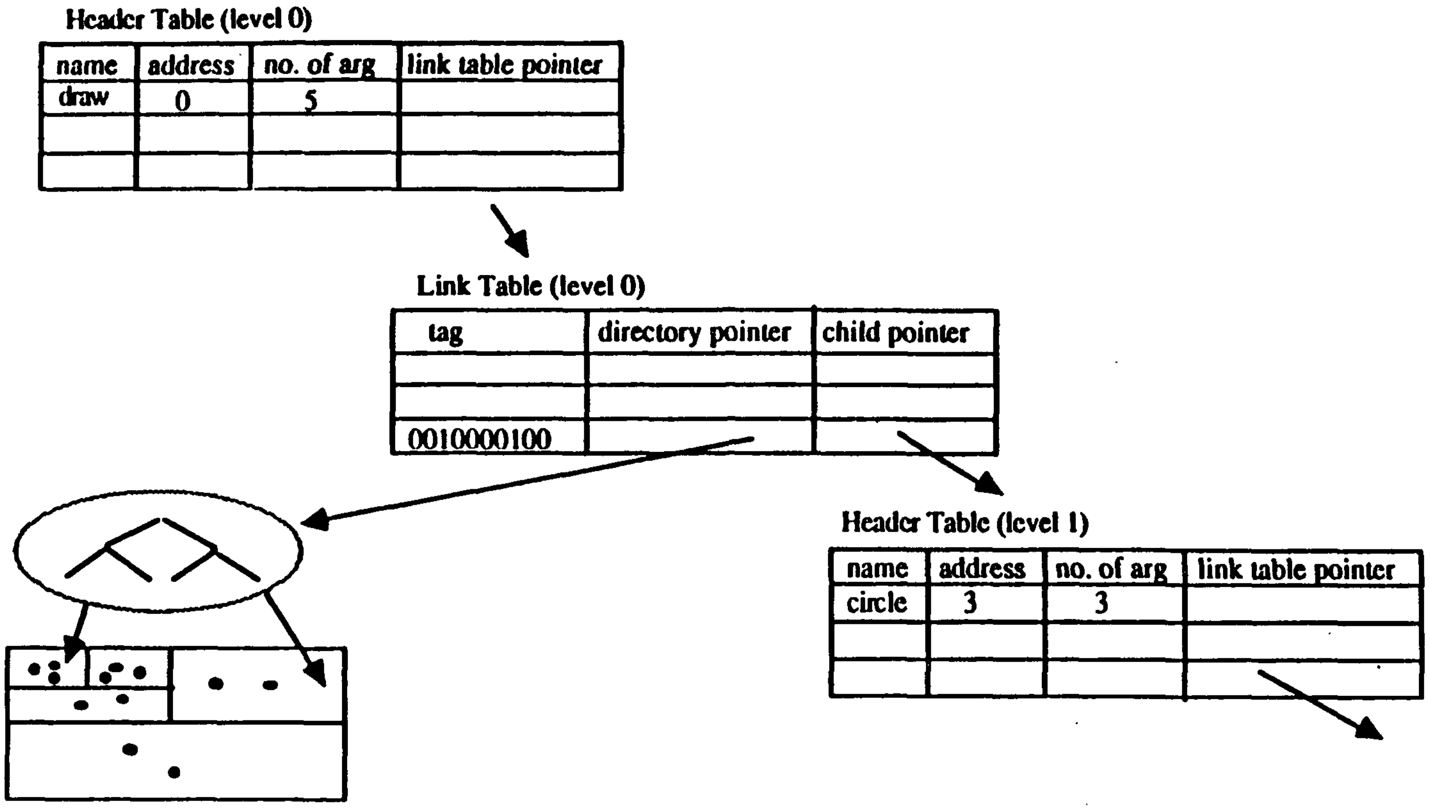
1. The ***header table*** for level k objects has four entries; relation names for level k relations (for example, "circle" for circle(10, none, 1)), argument positions (addresses) with respect to level 0 relation, the number of arguments of the relations, and pointers to the level k link tables.

2. The *link table* for level k objects has pointers to grid file directories to access the codes of level k normalized relations. Child pointers are used to indicate the locations of header tables at level k + 1.

3. The *grid file* for level k objects is used to store transformed data of level k objects, which is partitioned by using data space partitioning schemes described in section 3.3. Grid file directories allow easy access to a portion of partitions.

4.4 Surrogate file partitioning for parallel disks

Due to the limitations of both magnetic and optical technologies, the use of multiple disk systems in parallel such as the disk interleaving[14] has been considered. Low cost Winchester disks, which have the maximum data transfer rate of 1.5 megabytes/sec with the Small Computer System Interface (SCSI), make it feasible to use a very large number of disks for a computer system. The Connection Machine Model CM-2,[15] for example, can accommodate eight Data Vaults each of which consists of 39 individual Winchester disks



Header Table (level 0)

name	address	no. of arg	link table pointer
draw	0	5	

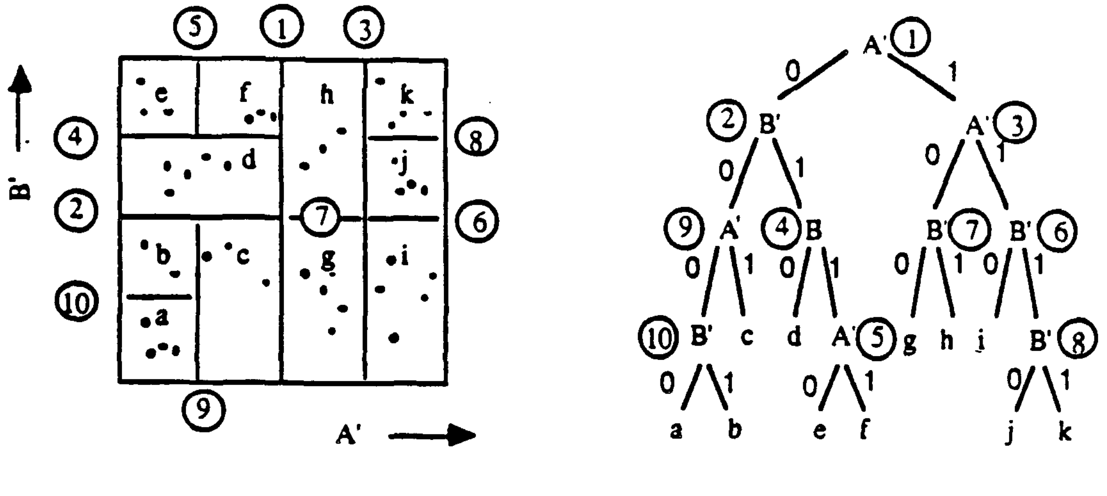
Link Table (level 0)

tag	directory pointer	child pointer
0010000100		

Header Table (level 1)

name	address	no. of arg	link table pointer
circle	3	3	

Figure 9: The organization of the ECCW surrogate file



(a) Two dimensional data space

(b) Trie directory

Figure 10: Trie directory for database partition

working in parallel. The Data Vault can transfer data at the rate of 40 M byte/sec and can hold up to 10 G bytes.

The ECCW adopts grid files[16] to partition large surrogate files into small manageable blocks. Suppose that we have a normalized subrelation R(A,B) and the corresponding *surrogate codes (without tags)* R'(A',B'). *A possible pair of* A' *and* B' *values can be plotted* in a two dimensional area as we have showed in Figure 10 (a). If a relation has k attributes, then k dimensional area should be considered. The surrogate file R' is now considered as a set of points scattered over this area. Since a code has a fixed length, a subrectangular can hold a fixed number of codes. Consider, for instance, a query R'(010010, 110011) and the trie directory[17] given in Figure 10 (b). For the query, only the bucket f needs to be accessed. As shown in the above example, we can divide searching into two processing levels, i.e. directory search and grid accesses. For exact match queries, two disk accesses are required. When buckets are ideally distributed and a large number of buckets should be accessed, the processing time can be reduced further by using parallel disk systems.

5 Conclusion and Future Work

The ECCW is designed to achieve high performance mainly for parallel processing environments. However, due to the index structures of the ECCW, we expect that, unlike previous surrogate file schemes, the ECCW can efficiently perform search operations in sequential environments as well. Furthermore, the storage requirement of the ECCW is not prohibitively large when compared to other schemes, although its performance is increased at the expense of more storage. We propose to compare the storage requirement of the ECCW with other schemes together with the retrieval performance in both sequential and parallel processing environments.

The main advantage of the ECCW stems from its ability to handle operations for complex objects. We are currently investigating the use of the ECCW for the following operations for both sequential and parallel processing environments.

1. *Unification/Term Matching.* We propose to develop a term matching and unification algorithm based on ECCW, since considerable speed-up is expected in handling complex terms if we can perform unification on surrogate files instead of actual knowledge bases. As in the case of relational operations on ground facts, it would be possible to delay accesses to actual knowledge bases until the processing of surrogate files is finished.

2. *Least Fixed Point Operation.* Since the LFP operation is basically repeated joins with equality check, it can be implemented as an extension of the parallel relational algebra.[8]

3. *Extended Relational Algebra (ERA).* Algorithms for ERA operations such as extended search/project[4,5] are relatively easy compared to the unification/term matching operations. But, in a parallel processing environment, the distribution of data to multiple storage units will play an important role to a high performance system. The use of ECCW for unification join[18] will be also investigated.

One of the important issues in the surrogate file processing is the effects of false drops. The false drops can be easily recognized for a search operation, but they may be hard to be detected for a complex operation. For example, after performing a number of join operations on conjunctions, it is very difficult to decide which portion of the answers resulted from false drops. Deciding the optimal point for accessing actual knowledge bases to remove false drops is one of the most difficult issues in the surrogate file processing. In other words, after performing a number of consecutive operations on the surrogate files, the detection of false drops might be difficult. However, checking false drops for every selection could degrade the system performance since it causes a lot of irrelevant secondary accesses. We will investigate efficient false drop detection schemes for the operations mentioned above.

Many details of the ECCW scheme such as the management of list objects and the directory scheme are left unspecified. We will evaluate various alternative methods to decide optimal structures for the ECCW, and continue to investigate the use of the ECCW for various operations to improve the performance of very large logic-oriented knowledge base systems.

6 References

1. P. B. Berra, S. M. Chung, and N. Hachem, "Computer architecture for a surrogate file to a very large data/knowledge base," *IEEE Computer* 20(3), 25–32 (1987).

2. P. Dadam et al., "A DBMS prototype to support extended NF^2 relations: an integrated view on flat tables and hierarchies," *Proceedings of SIGMOD '86*, 356–367 (1986).

3. F. Bancilhon and S. Khoshafian, "A calculus for complex objects," *Proceedings of the Fifth Symposium on Principles of Database Systems*, 53–59 (1986).

4. C. Zaniolo, "The representation and deductive retrieval of complex objects," *Proceedings of the 11th International Conference on VLDB*, 21–23 (1985).

5. C. Zaniolo. "Safety and compilation of non-recursive Horn clauses," *Proceedings of the First International Conference on Expert Databases*, 167–178 (1986).

6. K. Clark and S. Gregory, "PARLOG: parallel programming in logic," *ACM Transactions on Programming Languages and Systems* 8(1), 1–49 (1986).

7. D. Shin and P. B. Berra, "An architecture for very large rule bases based on surrogate files," *Proceedings of the 5th International Workshop on Database Machines*, 555–568 (1987).

8. P. B. Berra et al., "Computer architecture for very large knowledge bases," *Proceed-*

ings of IFIP WG 10.1 Workshop on C cepts and Characteristics of Knowledge-Based Systems (to appear), (1988).

9. D. H. D. Warren, *An Abstract Prolog Instruction Set.* Technical Report 306, SRI International (1983).

10. K. Ramamohanarao and J. Shepherd, "A superimposed codeword indexing scheme for very large Prolog databases," *Proceedings of the 3rd International Conference on Logic Programming*, 569–576 (1986).

11. M. Wada et al., "A superimposed code scheme for deductive databases," *Proceedings of the 5th International Workshop on Database Machines*, 569–582 (1987).

12. C. Stanfill and B. Kahle, "Parallel free-text search on the Connection Machine system," *Communications of the ACM* 29(12), 1229–1239 (1986).

13. H. S. Stone, "Parallel querying of large databases: a case study," *IEEE Computer* 20(10), 11–21 (1987).

14. M. Y. Kim, "Synchronized disk interleaving," *IEEE Transactions on Computers* C-35(11), 978–988 (1986).

15. Thinking Machines Co., *Connection Machines Model CM-2 Technical Summary*, Thinking Machines Technical Report HA87-4 (1987).

16. J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure," *ACM Transactions on Database Systems* 9(1), 38–71 (1984).

17. Y. Tanaka, "Massive parallel database computer MPDC and its control schemes for massively parallel processing," In *Database Machines*, A. K. Sood and A. H. Qureshi, ed., Springer-Verlag, 127–158 (1986).

18. H. Sakai et al., "A simulation study of a knowledge base machine architecture," *Proceedings of the 5th International Workshop on Database Machines*, 583–596 (1987).

Appendix 12-C

Computer architectures for logic-oriented data/knowledge bases [1]

Donghoon Shin
Schooi of Computer and Information Science
Syracuse University
Syracuse, NY 13244, USA

P. Bruce Berra
Dept. of Computer and Electrical Engineering
Syracuse University
Syracuse, NY 13244, USA

January 14, 1989

[1] This work was supported by the Air Force Systems Command, Rome Air Development Center at Griffiss Air Force Base in New York, and the Air Force Office of Scientific Research at Bolling AFB in Washington DC under Contract No. F30602-85-C-008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Abstract

In this paper we study a class of computer architectures proposed so far for logic-oriented knowledge bases by surveying specialized computer architectures supporting general clauses and terms in logic programming, together with important operations and algorithms considered in designing such machines. Our particular interest lies in handling large, complexly structured data/knowledge bases residing on secondary storage for data intensive applications.

1 Introduction

The Knowledge Base Management System (KBMS) is often considered one of the most important future computer applications, which can be viewed as an advanced database system augmented with an inferencing mechanism. One of the distinctive features of the KBMS's, when compared to other AI based systems, stems from its ability to handle large amounts of data efficiently. In other words, knowledge base systems should have all the functionalities of current database systems including efficient data access to secondary storage, while most AI systems such as production rule systems only concerns with manipulating data in fast primary memory.

Among various possible inferencing mechanisms, this paper concerns the first order logic or logic programming, as logic programming has been proven to be a useful tool in many knowledge-oriented applications such as expert systems, and can be well integrated with existing database management systems. The relational database system is well known to be congenial for the logic programming system as both systems are based on the first-order logic. Simple rules, except for rec rsive ones, of logic programming can be easily represented by the view definition of relational database systems. Even recursive queries can be processed by a sequence of relational algebra with equality checking. But the relational model only allows for normalized database, while logic programming deals with complex data structures called a *term*. On the other hand, the hierarchical model provides facilities to define a similar data structure to what logic programming deals with, but lacks the ability to support general rules. Thus, by introducing logic to a database system, the database system can gain two important functionalities; recursive view support by general rules and management complex objects defined by first order terms.

The major problem in interfacing logic programming systems with current database systems lies in that most traditional data models use some kind of data definition language to define the structure of data in advance, while logic programming systems can have arbitrary complex objects, and establish dynamic run-time data structures through unification. When secondary storage is involved, the traditional data access methods should be reconsidered, and the management of large persistent, complex objects can become a formidable task. Advances in hardware technology make it feasible to design specialized computer architectures for complicated applications requiring enormous processing loads.

This paper begins with basic definitions and principles used in the context of logic-oriented knowledge base systems. Section 3 presents a taxonomy of proposed knowledge base machines. We first classify the proposed machines into two large classies, and then describe the characteristics of those machines in each class. In the following sections, we will see how proposed machines are designed to deal with specialized operations required for these functionalities. Section 7 is a conclusion drawn from the survey.

2 Preliminaries

The basic data element that logic programming deals with is *term*. A *term* is defined as follows:

1. A variable is a term denoted by a capital letter such as X, Y, Z, ...

2. A constant is a term denoted by a lower case letter such as a, b, ...

3. If f is an n-ary function and $t_1, \cdots, t_n$ are terms, then f($t_1, \cdots, t_n$) is a term.

Terms are searched and manipulated through *unification*. Informally, the main purpose of unification is to make two or more terms identical by proper and the most general substitutions for logical variables existing in the terms. Figure 1 shows the original unification algorithm proposed by Robinson [51]. The disagreement set D of a set of terms S is the set of leftmost subterms consisting of different symbols. For example, the disagreement set of S ={p(a,g(X),b),p(a,r(Y),s(X))} is {g(X),r(Y)}. Robinson's algorithm requires exponential time with respective to the size of terms. The major processing load stems from 'occur checks' used to prevent variables from binding infinite terms. That is, when testing if a variable X unifies with a structured term t, a check should be done whether X occurs in t (i.e. {X/f(X)}). In current Prolog, the occur check is omitted for efficiency.

Ever since Robinson introduced the basic unification algorithm for the resolution principle, more efficient algorithms have been proposed. Paterson and Wegman developed the first linear unification algorithm based on Directed Acyclic Graph representation of terms [47].[1] The DAG representation of two terms, p(g(Y), h(X,Y)) and p(g(Z), h(Z, r(U,V))), and Paterson and Wegman's unification procedure on the DAG of the terms are illustrated in Figure 2. In the DAG representation of terms, common subexpressions are represented by a single subgraph (the variable Y in the above figure). The lines across two terms denotes the equivalence class.

[1] See also [7] for more discussion on Paterson and Wegman's algorithm. Another linear algorithm is developed by Martelli and Montanary [36].

```
/* S is a set of terms to be unified */

1. $k = 0; \sigma_0 = \phi$
2. If $|S\sigma_k| = 1$
      then mgu = $\sigma_k$
      else find the disagreement set $D_k$ of $S\sigma_k$
3. If there exist a variable v and a term t in $D_k$
      such that v does not occur in t (Occur Check)
      then $\sigma_{k+1} = \sigma_k\{v/t\}; k = k + 1$; go to 2;
      else S is not unifiable
```

Figure 1: Unification Algorithm

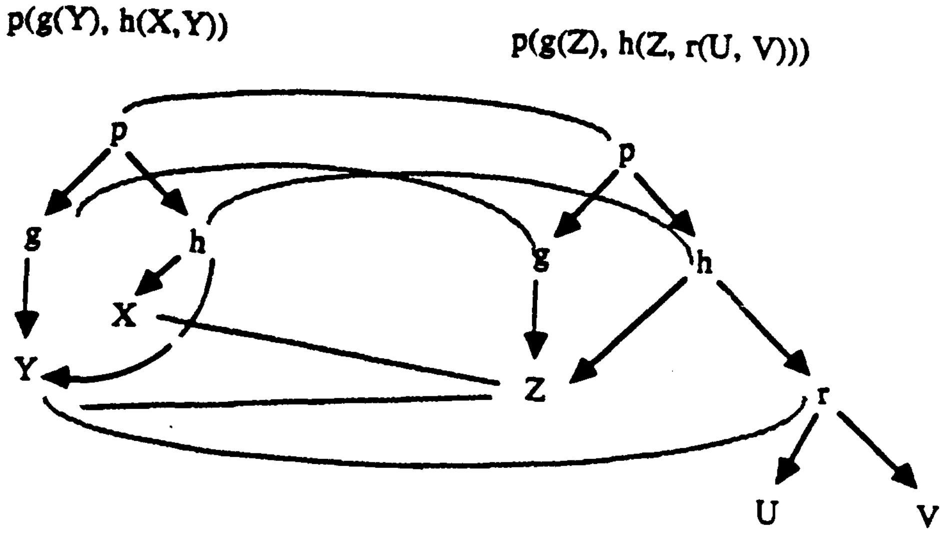


Figure 2: Unification on DAG

Since applying unification on large amounts of data requires a heavy processing load, exploiting parallelism in unification has been studied [25,83,90]. However, Dwork *et al.* [16] indicated that, since unification is inherently sequential, even parallel evaluation of a unification algorithm may not offer a considerable speed-up over a sequential one. On the other hand, term matching, which can be viewed as a special case of unification when either of the two terms to be unified is variable-free, has a very efficient parallel algorithm ($O(log^2N)$) [17]. Thus, unification can be more efficient if all unit clauses (facts) consist of variable free terms.

Parallel evaluation of logic programming is also an important issue in logic-oriented data/ knowledge base systems not only for performance but also for semantics, since the order of execution among several candidate clauses is very difficult to guarantee, especially when clauses reside in secondary storage. Conery [10] classified the inherent parallelism in logic programming into three major categories; low level parallelism, OR-Parallelism and AND- Parallelism. The low level parallelism is to overlap primitive operations required for logic programming evaluations so that the execution time can be reduced. For example, the Prolog Instruction Set called Warren Abstract Machine (WAM), proposed by Warren[86] can be executed in a pipelined fashion [80]. The sequential evaluation order in which alternative clauses are tried in sequential logic programming (Prolog) can be parallelized by trying all candidate clauses in parallel, which is called an OR-parallelism. In the context of very large knowledge bases, a special kind of OR-parallelism, called search parallelism, can enhance the performance of the system as finding candidate clauses is one of the most time consuming jobs. AND-Parallelism, on the other hand, concerns parallel evaluation of subgoals by replacing the strict left to right evaluation order of Prolog. A possible method of exploiting AND-parallelism is to restrict the parallelism by forcing subgoals with shared-variables to be executed sequentially through compile-time analysis, while executing the other literals in parallel by a simple run-time test [12]. Join on two relations can be viewed as an example of exploiting AND-parallelism in bottom-up processing.

Parallel logic programming languages have been developed to effectively exploit various forms of parallelism inherent in logic programming, and, in some cases, have been designed as the underlying language of logic-oriented data/knowledge bases. Examples of parallel logic programming languages include PARLOG [8], Concurrent Prolog [61], Epilog [87], and Guarded Horn Clauses [82]. For more complete surveys on parallelism in logic programming, readers may refer to [11,30,48]. As far as bottom-up (forward-chaining) set-oriented evaluation is concerned, parallelism may not be an important issue. But when top-down, tuple-by-tuple evaluation is to be used, the control of binding information among parallel processes becomes a critical requirement in the context of logic-oriented data/knowledge base systems.

2.1 Taxonomy of Logic-Oriented Data/Knowledge Base Machines

The major design objectives of logic-oriented knowledge base machines are to achieve high performance in two important tasks which can significantly influence on overall system performance; inferencing on large number of rules and management of a very large data space consisting of facts stored on secondary storage. In this paper, we classify logic-oriented data/knowledge base machines proposed so far into two large classes depending on how rules and facts are managed[2]:

1. *"combined" knowledge base machines* (C-KBM) consisting of a number of inference machines for handling rules and a number of database machines for managing facts.
2. *"integrated" knowledge base machines* (I-KBM) with capability of uniform management of both rules and facts.

The inference machine component of the C-KBM class can be viewed as a front-end of a database machine component, which is mainly responsible for the unification operation or for efficient management of memory space to maintain information about variable bindings for unification. The inference machines can be further classified into two subtypes based on the instruction level.

1. *unification machines* designed to speed up unification for logic programming interpreters.
    - *unification co-processors* which interact with the existing host by sharing memories or by interconnection networks.

[2] For broader surveys on AI machines, interested readers may refer to [28,81,85]. They use different classification schemes; in [28], the AI machines proposed so far are classified into three major categories, namely language-based machines, knowledge-based machines, and intelligent interface machines. On the other hand, Treleaven *et al.* [81] classifies computer architectures into 6 major categories including two AI-related categories logic computers and knowledge-based computers. The scope of this paper include those AI-machines based on logic and database machines in the logic paradigm. For surveys on database machines, readers may refer to [45,49]. The taxonomies of database machines are presented in [6,49].

- *unification filters* which function similar to database filters. They can perform unification on data streams from a large file store.

2. *Prolog machines* designed for compiled Prolog instruction sets. Most of them are based on the Prolog instruction set developed by Warren (Warren Abstract Machine: WAM) [86]. Since compilers' performance is far greater than that of interpreters, order of magnitude speed ups can be achieved by exploiting computer architectures tailored to the instruction set.

The database machine component of the C-KBM class provides knowledge base system with efficient access paths to objects stored on secondary storage, and supports diverse database operations such as join. In the context of knowledge base systems, we can consider two types of database machines.

1. *deductive database machines* mainly concerned with how to deal with rules in relational database environment. For example, the Least Fixed Point (LFP) operation [2] is considered to support recursive rules.

2. *database machines for unnormalized relations* designed to support non-relational models such as the hierarchical database model. As general terms and Horn clauses cannot be represented by the relational model, these machines can be used in dealing with general terms.

The knowledge base machines in the second class, the integrated knowledge base machines (I-KBM's), integrate data manipulation and inferencing into a single system. From the software point of view, this approach includes augmenting logic programming systems with a secondary storage access capability. We classified the I-KBM's into three categories:

1. *massively parallel ,¬achines* which can offer high performance required for integrated knowledge base machines. Since loading data to the processing level can be a major system bottleneck, a high data rate should be provided by I/O systems.

2. *machines with intelligent storage management* specially designed for logic-oriented knowledge bases. The principal enhancement of this approach over the combined approach stems from the capability of managing objects residing in secondary storage by intelligent storage management such as virtual memory and/or paging systems. Complex objects (terms) can be retrieved based on unification/term matching, and top-down evaluation (backward-chaining) of clauses is used.

3. *specialized associative processors* designed for logic-oriented knowledge base processing. By exploiting a clever encoding scheme, associative memories can be used for managing both rules and facts.

Many interesting physical data organization schemes and indexing schemes have been proposed with these machines, aiming at efficient hardware implementation.

3 Inference Machines

3.1 Unification Machines

3.1.1 Unification Co-Processors

Having found that most execution time of the UNSW Prolog interpreter is taken by unification operations, Woo proposed Hardware Unification Unit (HUU) as a back-end server of VAX11/780 to speed up the unification operations [88,89]. As seen in Figure 3, the HUU consists of six components:

1. *host interface* which receives input data (pointers to two terms to be unified) from the host and sends results to the host. It is also used by the host to access internal registers and the local memory of the HUU.

2. *registers* which store input data and intermediate results.

3. *ALU* which performs arithmetic operations (addition and comparison).

4. *local memory* used as a variable stack.

5. *microcontroller* which generates all control signals used in the HUU.

6. *clock generator* which generates clock pulses when the HUU is active.

Although the HUU can improve the performance of unification up to 100 times faster than the software 'unify' function of the UNSW Prolog, the HUU may not outperform Prolog compiler in terms of the overall performance. Dorby *et al.* indicated that the compilation to an optimized instruction for Prolog can gain factors of 20 over interpretation [14]. Even if we assume that the 'unify' function takes 70 % of the total execution time and the HUU can improve the performance of unification operations 100 times, the HUU can only achieve speedups of 3.3. The performance analysis of the Syracuse Unification Machine (SUM) showed a similar result (speedups of 2) [52].

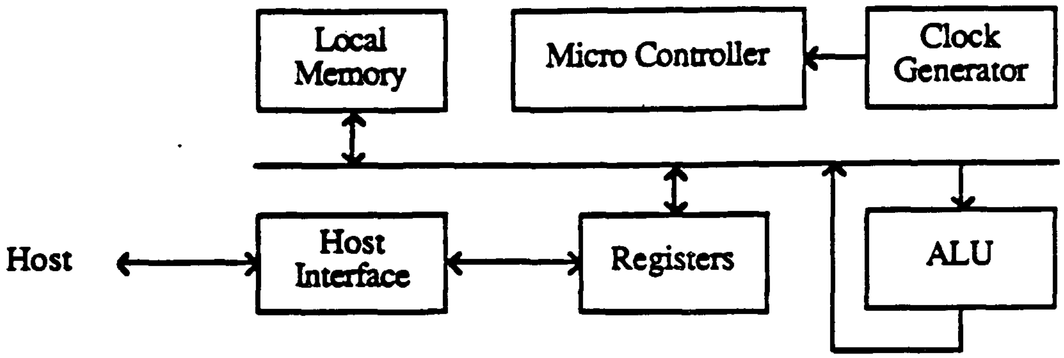


Figure 3: Hardware Unification Unit by Woo

The associative processor (AP) designed by Stormon [67] for unification is an enhanced version of the SUM. The Stormon's unification machine enhance the performance of the SUM by providing content addressability in stacks for managing binding environments. Unlike in the HUU, the host sends a stream of pairs of subterms to the associative processor since all data structures reside in the host. The maximum speed that can be achieved by the unification processor is predicted up to 500K LIPS (Logical Inference Per Second) which outperforms most Prolog machines.

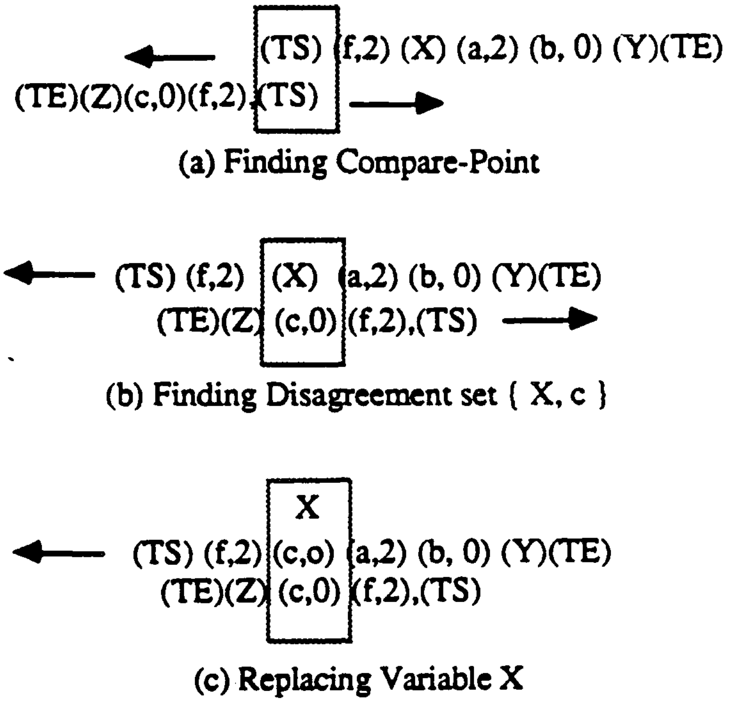
3.1.2 Unification Filter

Shobatake and Aiso [63] proposed a systolic-like method to implement a VLSI-oriented unification processor by using a linear organization of processing elements (cells) aiming at eliminating frequent data transfers between the host and the unification processor.

A cell consists of two identical pairs of registers — a bound variable register (BVR) and a symbol register (SYR). The SYR contains a functor with its arity, and the BVR is designed to keep the variable after unification is completed. It plays a similar role as the trail stack in Prolog interpreters. An input term to the cell array is composed of beginning mark(TS), functor name with arity followed by arguments represented also by functor name with arity number, and finally, ending mark (TE). This representation allows to store arbitrary complex terms. The other term to be unified with the given term is represented in a reverse order. The arities of variables and constants are assigned to zero. For example, a term f(X, a(b, Y)) can be represented as

(TS)(f,2)(X)(a,2)(b,0)(Y)(TE)

and the other term f(c,Z) to be unified can be represented as


(TS) (f,2) (X) (a,2) (b, 0) (Y)(TE)
(TE)(Z)(c,0)(f,2),(TS)
(TS) (f,2) (X) (a,2) (b, 0) (Y)(TE)
(TE)(Z) (c,0) (f,2),(TS)
X
(TS) (f,2) (c,o) (a,2) (b, 0) (Y)(TE)
(TE)(Z) (c,0) (f,2),(TS)

(a) Finding Compare-Point

(b) Finding Disagreement set { X, c }

(c) Replacing Variable X

Figure 4: Stream Unification

(TE)(Z)(c,0)(f,2)(TS).

Shobatake and Aiso's unification machine performs unification on the stream of two terms represented by the family order. Only the cell in the compare-point actually performs comparison. After variable is instantiated to a constant, the variable binding is broadcast to other cells to prevent binding conflicts among shared variables. When a variable is bound to a non-atomic term, the cell corresponding to the variable has the value of the functor, and a portion of a term is shifted to a direction to make an empty region. Then the empty region is copied from the subsequent subterm. Figure 4 illustrates the unification procedure in the unification processor.

One of the problem of the stream unification method expected in the implementation stage is the size of cell arrays. That is, the number of cells should be equal to or greater than the maximum number of symbols representable in a term. Since the data structures of Prolog can be arbitrary large (e.g. list), it would not be possible to construct a cell array large enough for all cases.

Tanabe and Aiso [72] proposed to use the structure sharing scheme, and designed pipelined unification processor (PUP) for stream processing to solve this problem. One of the major improvements of the PUP over the previous machine is that consistency checks on variable bindings are separated from comparisons among constants by using a pipelining technique. The PUU consists of 4 major components:

1. *STBV (Select Term Binding Variable) processor* which performs a comparison among constants to be unified (a preunification step). In this step, all the variables appearing in the terms are regarded as don't care match indicators.

2. *AVTC (Assign Variable To CTBV) processor* which finds binding conflicts for shared variables.

3. *CTBV (Charge Term Binding Variable) processors* which manage the actual bindings for variables. There are number of CTBV's each of which contains a variable binding. After a successful unification, the most general unifiers (mgu's) will be stored in CTBV's.

4. *CCTBV (Control CTBV) processor* which controls CTBV and AVTC.

The unification machines described above are not concerned with secondary storage. They are classified as unification filters due to their stream processing natures. The unification filter proposed by Sabbatel *et al.* [55,56,57] is a first unification machine that can find candidate clauses or unifiable terms from secondary storage. This machine exploits "on the fly" unification and set-oriented retrieval by using a pipelined method. The proposed hardware component functions similar to the database filters. The unification algorithm performed in the unification filter is separated into two stages; a preunification step and a consistency check. When a specialized secondary storage with very high data transfer rate is developed, the stream unification method will be very useful for logic-oriented data/knowledge bases.

3.2 Prolog Machines

Prolog machines are designed to enhance the performance of both sequential and parallel logic programming languages. Important techniques used in sequential Prolog machines include the pipelined execution of WAM instruction sets by exploiting special registers for passing arguments among procedures and the clause indexing scheme to find candidate clauses for the given goal. In WAM, clauses with the same predicate name are divided

Machine	Institution	KLIPS	Characteristics
PLM [15,14]	UC Berkley	425	Sequential, WAM
PPP [18]	UC Berkley	N/A	Parallel extension of PLM and WAM
HPM [41]	NEC	280	Sequential, WAM
IPP [1]	Hitachi	1000	Sequential, WAM Special clause Indexing
PSI [20]	ICOT	30	Sequential Interpreter (Structure Sharing)
PSI-II [20]	ICOT	100	Sequential, WAM
PIE [39,73]	U. of Tokyo	N/A	Parallel Interpreter (Structure Sharing)
PEK [71]	U. of Kobe	40	Sequential Interpreter
PARK [37]	U. of Kobe	N/A	Parallel
PIM-R [22,40]	ICOT	N/A	Reduction
PIM-D [29,22,40]	ICOT	N/A	Dataflow, GHC
ICM3 [43]	ECRC	530	WAM, Co-processor

Table 1: Prolog Machines

into several groups according to the types of the first arguments (i.e. variable, constant, list, structured term). Then, the selection of a candidate clause in a group is performed according to the hashed value of the first argument. This type of indexing is proven very useful for small logic programs residing in main memory, and allows programmers to enhance the performance by making clauses discriminated by the first arguments. Parallel Prolog machines also use diverse techniques such as parallelized WAM instructions, the use of AND/OR parallelism, and dataflow. Table 1 shows some Prolog machines proposed so far.

A representative Prolog machine based on the WAM is the Programmed Logic Machine (PLM) [14,15]. PLM exploits small scale parallelism and instruction pipelining with special registers and a microcoded instruction set. PLM consists of two main units as seen in Figure 5; the control unit and the execution unit. The control unit consists of a control store containing micro codes and a microsequencer for executing the micro codes, and is primarily responsible for directing the actions of the execution unit for its own command as well as for communicating with the host interface using buffers. The execution unit consists of two major components; a register file and an ALU. The register file consists of

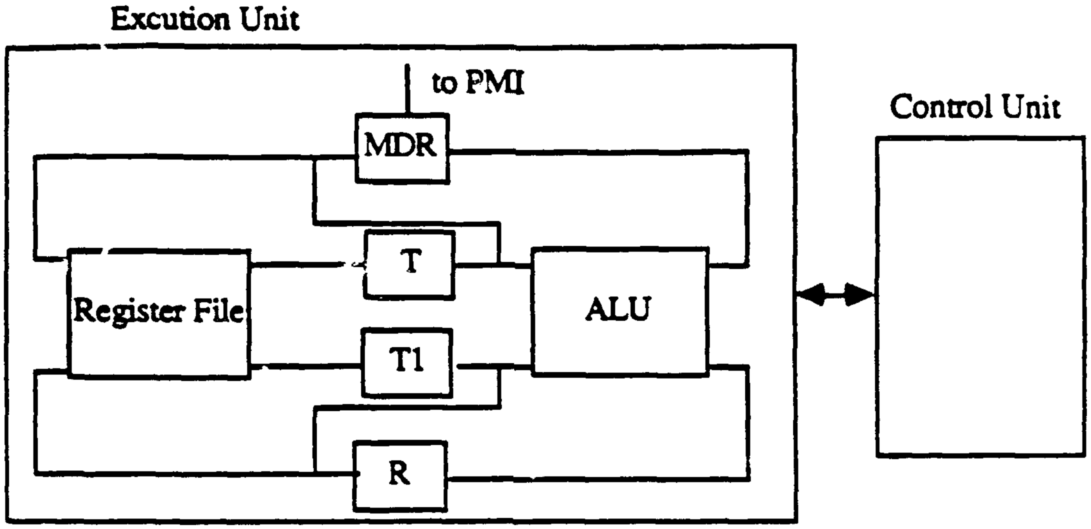


Figure 5: The Ar^`itecture of PLM

nine general registers to support the PLM instruction set. These registers are mainly used to store pointers to indicate the current state of computation and the previous state for backtracking. The other special registers - T1, T, MDR, and R - separate an operation into three stages for pipelining. In the first stage (C stage), the T and T1 registers are loaded with data. Then, at the next stage (E stage) the ALU performs corresponding commands and intermediate results are placed on the R and MDR registers. In the final stage (P stage), the results are transferred to the general registers in the register file. A typical micro routine involves one C stage operation, several E stage operations and one P stage operation. By pipelining these three stages, the system performance can be considerably improved. Parallel Prolog Processor (PPP) is a parallel extension of the PLM to exploit AND/OR parallelism [18]. A commercial version of PLM called X1 is also being developed by Xenologic [13,50].

The Integrated Prolog Processor (IPP) [1], which is one of the highest performance Prolog machines proposed so far, uses optimal arguments (up to 2) instead of the first argument for indexing. The optimal argument is the one on which clauses can be discriminated. Consider, for example, the following clauses:

$$p(X, r(Y,Z), [H \mid T]) \leftarrow \cdots$$
$$p(X, s(Y,Z), [H \mid T]) \leftarrow \cdots$$

The second arguments are optimal for indexing since the clauses differ each other only in the second arguments. Alternatively, an argument having the maximum number of constants is selected as the optimal argument. If there are more than two arguments that satisfy this condition, the left-most two optimal arguments are selected for indexing. If there are no such arguments satisfying above conditions, then the first argument is selected for indexing as in WAM. This indexing scheme contributes to the high performance of IPP.

4 Data Base Machines

Research and development on database machines have drawn a great attention in past decade as high-level data models such as the relational model prevails. Modern database systems should be supported by building multiple layers of software, mapping high-level commands into low-level storage representations, which have been considered as a cause of inefficiency under conventional computer systems. The main purpose of building a database machine is to reduce the gap between the semantic model and the actual internal representation of data [68]. In the context of logic-oriented knowledge base systems, we can consider two types of database machines; deductive database machines and non-relational database machines. The former are designed as a back-end of inference machines and mainly support the relational model by providing means of efficient relational operations for general rules. On the other hand, the machines in the latter category provides basic mechanism for handling complex objects. We investigate physical data organizations of these database machines since an efficient management of a large data space is one of the most critical requirements [5,66].

4.1 Deductive Database Machines

4.1.1 ILEX

The ILEX machine proposed by Li supports a typical example of knowledge base systems based on the combination of existing systems [32]. In order to avoid modifications on an existing logic programming system residing in the host and to provide an interface between a data manipulation language and a logic programming language, a meta-level procedure transforming a query written in the data manipulation language to the corresponding canonical logic programming form is used.

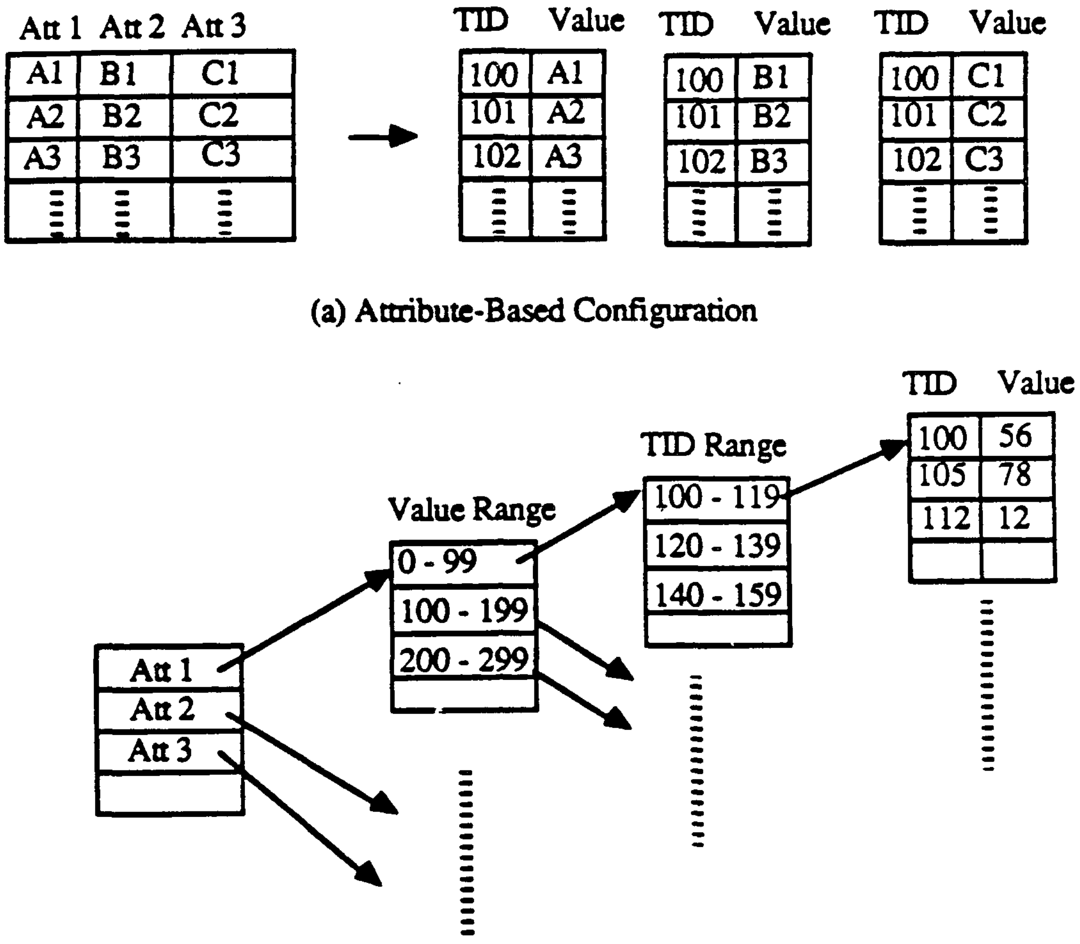
The hardware of ILEX consists of three major components; a PDP-11 host, Relational Associative Processor (RAP), and an information retrieval system called MEMEX. The host is mainly responsible for user interface operations and performs compilation and optimization of user queries. RAP sends transformed queries to MEMEX for searching, and then performs database operations such as join on the qualified data by using content addressable mechanism. The MEMEX is a specialized information retrieval system for searching text by using inverted lists (concordance). The KM-1 proposed by Kellog [31] takes a similar approach as ILEX in the sense that it uses existing systems rather than developing specialized hardwares (the Xerox 1100 Lisp machine as the front-end and the Britton-Lee IDM-500 relational database machine as the back-end).

4.1.2 DELTA

DELTA is a functionally distributed MIMD database machine supporting inference machines (PSI's) as a back-end [58]. It consists of two subsystems: the *relational database management supervisory/processing* (RSP) subsystem and the *hierarchical memory* (HM) subsystem. The RSP subsystem consists of four components each of which functions as follows:

- *interface processor* (IP) which is responsible for communications between hosts and the CP and between hosts and the HM subsystem.
- *control processor* (CP) which transforms queries into subcommands.
- *relational database engines* (RE's) which performs relational operations. Join operations can be performed in O(N) by using pipelined merge-sort algorithm and specialized hardwares.
- *maintenance processor* (MP) which monitors the status of the system.

The HM subsystem is used to store large databases, and provides access paths for efficient retrieval of qualified tuples. The HM has a two-layer structure; the lower level layer consists of moving head disks, and the upper level layer is a fast RAM-cache memory called database memory unit (DMU) which can be further divided into a buffer area, a disk cache area, a HM control program area and a working data area of the HM control program. The physical data organization scheme of DELTA is a fully decomposed storage organization, where a relation with N arguments is divided into N binary relations with a tuple identifier (TID) as the first attribute for all N binary relations (see Figure 6 (a)). This organization is well suited to partial match retrieval queries, especially for those queries having only



(a) Attribute-Based Configuration

(b) Two-Level Clustering Method

Figure 6: Physical Data Base Organization of DELTA

a small number of arguments (attributes) specified. To reduce the search space further, the data is clustered according to the TID numbers as well as the values. This two-level clustering method based on this configuration is shown in Figure 6 (b).

The rules and facts of knowledge bases are stored in PSI and DELTA respectively. Users write programs in a logic programming language to access external databases resided in DELTA's HM subsystem. Using the rules stored in PSI's, the PSI processes the given goal by converting it into DELTA commands and optimizes them to reduce the number of disk accesses. Additional information is attached to the commands to form a packet, and then the packet is sent to DELTA via a local area network.

```
/* R: result; T: knowledge base; T': previous result; $R_\Delta$:new result
   $\sigma$:selection; $\pi$: projection; $\bowtie$: join /*
$R \leftarrow \sigma, \pi(T)$;
repeat
   begin
      /* T is assumed to be a binary relation */
      $T' \leftarrow R$;
      $R_\Delta \leftarrow \pi(T'.2 \bowtie T.1)$;
      $R \leftarrow R \bigcup R_\Delta$
   end until $T' = R$
```

Figure 7: Basic LFP Algorithm for a Simple Transitive Clauses

4.1.3 DDC

When no recursive rule exists, any rule can be translated to a number of conjunctions wchich can be solved by a series of relational algebra operations such as selections, projections, and joins. The series of relational operations required to solve a goal is called a plan. For example, the conjunction of EDB predicates p(a, X), q(X, b) can be solved by two selection operations on p and q based on the given constant values *a* and *b* respectively followed by a join operation between the second attribute of p and the first attribute of q. However, unlike the view definitions of relational database systems, the rules in logic-oriented knowledge base systems can be recursively defined. In this case, the plan generator cannot recognize how many plans it must generate to find all solutions for a given goal. Therefore, the system is required to inform the plan generator of the termination point by monitoring the intermediate results. This operation is called a *Least Fixed Point* (LFP) operation [2,24]. The LFP is basically performed by union and equality-check between previous results and newly generated ones. The basic LFP algorithm presented in Figure 7.

The Delta Driven Computer (DDC) [21] uses the Alexander method [53] to efficiently perform LFP operations. DDC consists of a set of PCM (Processor, Communication Device, Memory) nodes interconnected via a bus without a shared memory, and supports three language levels as follows:

- a high level language such as a logic programming language or a functional programming language
- an intermediate level language called the Virtual Inference Machine (VIM) which is

a production rule system without functional symbols.

- a DDC machine language called DDCL.

The recursive rules can be eliminated by transformation. Consider for example, the following rules and a goal.

← r(a,W).
R1: p(X,Y) → r(X,Y).
R2: p(X,Z), r(Z,Y) → r(X,Y).

By using the Alexander method, the above rules are translated to

R1.1: problem_r(X),p(X,Y) → solution_r(X,Y).
R2.1: problem_r(X),p(X,Y),r(Y,Z) → solution_r(X,Y).
R2.2: problem_r(X),p(X,Y) → problem_r(Y),cont(X,Y).
R2.3: solution_r(Y,Z), cont(X,Y) → solution_r(X,Z).

where the recursive rules have been removed. Then, the DDC compares previously generated results with newly generated ones. This mechanism is called the Delta Driven Execution Model (DDEM). Then a VIM rule is transformed to a number of *delta rules*. For example, a VIM rule p,q → r leads to two delta rules

$W\Delta p$,q' → $B\Delta$r
$W\Delta q$,p' → $B\Delta$r

where p' and q' are the current representations of p and q in the database. Here $B\Delta$ and $W\Delta$ represent the deduced facts (virtual relation) and the newly generated results respectively. When no more $W\Delta$ is generated, the process stops. At DDCL level, both existing facts and deduced facts are viewed as relations. A large number of tuples are distributed to multiple processing elements by using a hashing method.

4.1.4 MPDS/MPDC

Tanaka proposed a data-stream database computer called Multiple Processor Direct Search (MPDS) which uses the encoding/ decoding of databases to reduce the size of large databases [74]. The use of database encoding makes variable length values mapped to fixed length codes. It not only reduces the data transfer time, but also makes data structure more uniform so that a specialized hardware such as an associative memory can be exploited. The storage organization scheme of MPDS is based on transposed relations which can be viewed as inverted files of the encoded database. That is, each attribute has a transposed binary relation whose first attribute represents the values and the second attribute stores the tuple identifier of the relation.

The Massive Parallel Database Computer (MPDC) [75] is a successor of the MPDS. The MPDC uses the grid file [42] as the basic physical storage structure to partition large data space into manageable small buckets. MPDC consists of two subsystems; the control subsystem and the data subsystem. The control subsystem and the data subsystem are responsible for partition search and partition processing respectively.

In the grid file, tuples with n attributes are represented as points of n dimensional hyper space $(D_1, D_2, \ldots, D_n)$ where D_i is the domain of the ith attribute. The hyper space is partitioned according to the data distribution. Figure 8 shows the partition of data space for two dimensional data (a) and three dimensional data (b). In Figure 8 (a), the numbers in circle represent the order of data space partitioning. We assume that a bucket can hold up to 5 records.

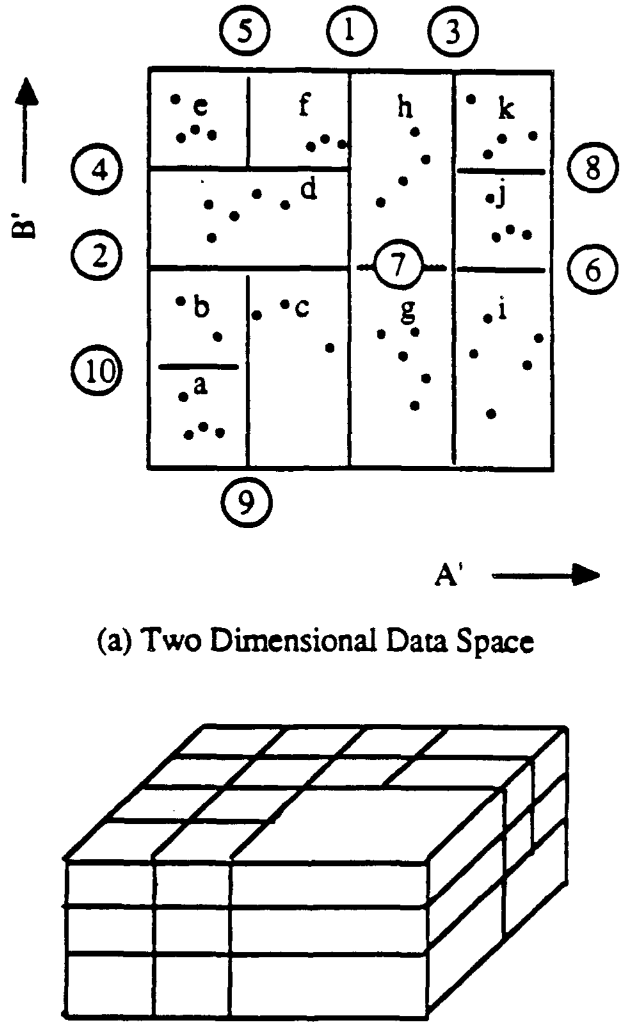
The variations of grid file schemes depend on a number of issues such as the splitting policy of a overflow bucket, the merging policy of two or more buckets in case of deletion, and the implementation of a grid directory. Directory schemes proposed for grid files can be classified into three types:

1. Scale based directory [27]

2. Interpolation based directory [19,46]

3. Trie directory [44,75]

The MPDC adopts the trie directory scheme, where the attribute used in splitting is maintained in the corresponding node of the trie directory. Suppose, for example, that we have a relation R(A,B) and the corresponding encoded relation R'(A',B'). A possible pair of A' and B' values can be plotted in a two dimensional space as shown in Figure 8(a). If a relation has k attributes, then k dimensional area should be considered. The encoded relation R' can be represented as a set of points scattered over this area. Assume that a rectangular region represents a bucket with capacity of 5 encoded tuples. Consider, for instance, a query R'(010010, 110011). For the query, only the bucket f needs to be accessed. Figure 9 shows the corresponding colored binary trie directory for the grid files shown in



(a) Two Dimensional Data Space

(b) Three Diemensional Data Space

Figure 8: Grid File Based on Data Space Partitioning

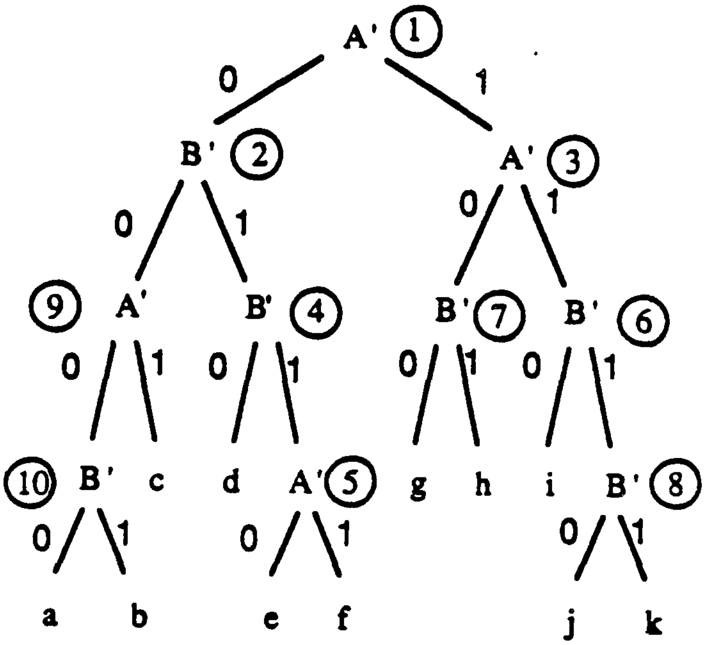


Figure 9: The Physical Data Organization of MPDC (Colored Binary Trie)

Figure 8(a). Since each node can be represented by a small amount of data, the directory itself can be resident with main memory in most existing computer systems.

As shown in the above example, by using grid files, searching can be divided into two processing levels i.e. directory search and grid accesses. For exact match queries, only two disk accesses will be required.

4.2 Database Machines for Unnormalized Relation

Most database machines developed so far are designed for the relational model. The data in the relational model strictly follows the first normal form restriction where each attribute value must not be decomposable. Many researchers become aware of this limitation of the relational model in the context of knowledge base systems and in semantic modeling, and thus have investigated the use of non-first normal form (NFNF) relations [54]. In this section, we consider a database machine CASSM[70] which is designed for the hierarchical model, and B-LOG machine for data/knowledge bases having network-like structures.

4.2.1 CASSM

Context Addressed Segment Sequential Memory (CASSM) is one of the earliest database machines [70], which supports hierarchically structured relations [69]. The CASSM has

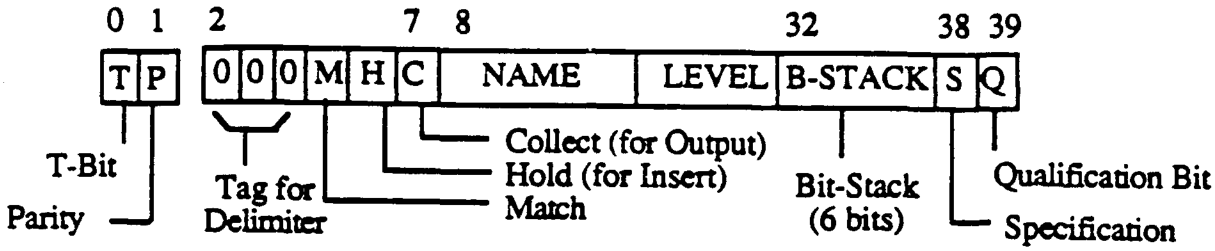


Figure 10: The Word Organization of CASSM (Delimiter Type)

facilities for searching complexly structured data types such as sets, trees, and directed graphs. Another distinctive feature of the CASSM is that it can fetch compiled programs from disks. This concept is similar to that of logic programming systems, that is, program and data are uniformly managed.

The CASSM contains a linear array of cells each of which consists of a processing element and a rotating memory device. A hierarchical record is linearized in a top-down and left-right order (level-order), and is physically stored in a linear vector consisting of 40-bits words. A word is composed of a 8-bit tag and 32-bit data. According to the tag bits, eight data types can be identified, including delimiters, name-value, pointer, string, instruction, operand, protect-lock, and erased data. Figure 10 shows the data structure of the delimiter type. Since CASSM contains data and program represented by hierarchical trees, each node of a tree has a unique level number. As seen in the figure, the tag consists of a bit stack (6 bits), S bit (Specification Bit), and Q bit (Qualification Bit). The bit stack is used for storing intermediate results. The Q bit is used to mark a context to search a specific subtree, while the S bit is used to find nodes satisfying the specification given in the query. Due to its associative processing nature, processing nodes in a lower level is as easy as processing higher level nodes.

4.2.2 B-LOG Machine

Lipovski and Hermenegildo [35] proposed a parallel "branch and bound" algorithm called B-LOG to efficiently evaluate logic programs stored in secondary storage. A parallel computer exploiting a special secondary storage named a Semantic Paging Disk (SPD) [34] is proposed to implement the B-LOG scheme. The SPD is specially designed to support pointer-based searches for databases with a network structure, and consists of one or more search processors (SP). Each SP has a read-write head corresponding to one or more tracks, a random access memory able to hold a track's data, and a special logic. The logic is designed to perform search and mark operations by traversing pointers. The basic task of SPD is thus to provide processors with appropriate subset of data needed for further processing.

Since SPD works effectively on complexly structured data linked by pointers and stored on secondary storage, rules and facts are represented by a network-like model. A block representing a rule or a fact contains pointers to other blocks of subgoals, and weights (bounds) to guide the search procedures for finding optimal path. The search strategy adopted by the B-LOG machine is the best-first search combined with a branch-and-bound algorithm. Although the best-first search has advantages over depth-first or breadth-first search in the context of parallel processing, it is not an easy task in the evaluation of logic programs to assign a weight to each arc. In the B-LOG approach, weights are added to each branch of OR-tree. The AND nodes, the conjunction of subgoals, are assumed to be executed sequentially. A heuristic success probability is introduced as a basis for branch-and-bound algorithm.

5 Integrated Knowledge Base Machines

In this section, we consider integrated knowledge base machines which can manage both facts and rules in a uniform way. For very large knowledge bases, these machines should support unification-based retrieval of clauses from secondary storage, and thus require enormous processing loads. They generally exploit the top-down strategy on which most logic programming languages are based, rather than the databases' conventional bottom-up strategy.

5.1 Massively Parallel Machines

Many massively parallel machines have designed aiming at high performance AI systems. But when large amounts of data are involved, transferring data from secondary storage to the processing level can be a serious bottleneck in these machines. We present three massively parallel machines which can be used for logic-oriented knowledge bases, and assess the issue of I/O bottleneck.

5.1.1 DADO

DADO consists of a large number of identical processing elements (PEs) connected by using a binary tree topology, and was originally designed to provide high performance in the execution of large production rule systems [65]. A Control Processor (CP) is attached to the root of the DADO tree, which is mainly responsible for broadcasting data to PEs.

Taylor *et al.*[77,78] proposed an implementation technique for OR-parallel execution of logic programs by exploiting the parallel pattern matching capability of DADO. For logic program evaluation, clauses are initially distributed across DADO as the following:

- the bodies of rules are stored in the CP,
- clause heads are stored across each PE, and
- an index is associated with each clause head and the corresponding body (no index for facts).

The proposed parallel evaluation procedure is as follows:

1. The CP broadcasts the given goal to all PEs.
2. Each PE performs unification between the broadcast goal and stored heads.
3. The CP polls each PE to collect bindings first from unit clauses, and then from rule heads.
4. The collected binding sets are merged. For example, {a(Y)/X, b/Y} is reduced to {a(b)/X}.
5. The conjunction is first solved independently in a left-to-right manner. Then join is performed for shared variables.

In [48], Ponder and Patt argues the correctness and soundness of DADO's parallel evaluation scheme. But, the major problem of the DADO architecture for data intensive applications lies in slow data transfer time; the data should be read from the host and should be broadcast from the root, resulting in high overheads for large knowledge bases.

5.1.2 NON-VON

Shaw proposed a massively parallel database machine called NON-VON [62]. Although NON-VON was originally designed for efficient relational operations, it can also support rule-based systems [26]. NON-VON consists of two major components; a primary processing subsystem and a secondary processing subsystem. The primary processing subsystem is made of many small processing elements (SPE) and large processing elements (LPE). The SPE's are interconnected by a tree network like in DADO, but leaf nodes are also connected via a mesh connection. Each SPE has a very small amount of local memory where data are stored. On the other hand, a LPE is a general purpose microcomputer having a large random access memory to hold programs, which is connected to a SPE located near the root of the tree. The operation of LPEs is primarily in a MIMD mode, while SPEs perform instructions broadcast by LPEs in a SIMD mode. Thus, NON-VON can operate in a multiple-SIMD mode. The secondary processing subsystem consists of a large number of disk systems each of which is connected to a LPE. Associated with the secondary processing subsystem is the intelligent head unit capable of on-the-fly data filtering. Thus, NON-VON can offer higher I/O bandwidths from secondary storage than DADO.

5.1.3 Connection Machine

The Connection Machine Model CM-2 [79] is a massively parallel back-end computer consisting of a very large number (8K - 64K) of data processors, a number of sequencers, a Nexus switch, and I/O systems. Each data processor has a 8 K bytes of bit-addressable local memory and an arithmetic logic unit. A data processor can communicate with any other data processor via two communication methods. Sixteen data processors are fabricated in a chip, each of which is connected by a cube topology. Associated with a chip is a router which allows accesses to the local memory of other processors. First, the router can be used for communication. Second method is the use of the NEWS grid which allows processors to communicate in a rectangular pattern. That is, a data processor in a grid pattern such as $256 \times 256, 1024 \times 64, 8 \times 8192, \ldots, 8 \times 8 \times 4 \times 8 \times 8 \times 4$ can pass its data to neighbor processor. A data processor does not directly execute parallel instructions. Instead, they are processed by a sequencer. Processors can be divided into up to four independent sections each of which can have its own sequencer, router, and NEWS grid. Up to four front-ends can be connected by the Nexus which is a programmable, bidirectional switch.

The CM-2's I/O structure allows data to be moved into or out of the parallel processing unit at peak rate as high as 320 megabytes/second. I/O is done in parallel by allowing as much as 2 K processors to simultaneously perform I/O related operations when the maximum number of I/O channels are available. The Data Vault, CM-2's mass storage system, can be connected to each I/O channel, which can transfer data at the rate of 40 megabyte/sec and can hold data up to 10 gigabytes. A Data Vault unit consists of 39 individual disks working in parallel. Thus to get the maximum bandwidth, data is spread across drives and disk cache is also exploited. In [4], Boral proposed a similar I/O system consisting

Many interesting applications have been developed for the Connection Machine using the high I/O systems. One of the interesting example is the document retrieval system based on the superimposed code word (SCW) [64]. By using a special database encoding scheme called a concatenated code word (CCW), we have implemented an experimental logic-oriented knowledge base system on the Connection Machine [3].

5.2 Intelligent Virtual Memory Support/Disk Cache

One of the approaches in realizing a logic-oriented very large knowledge base is to provide a logic programming system with facilities to access secondary storage. Intelligent virtual memories and special disk caches have been proposed for this purpose.

5.2.1 MANIP-2

MANIP [84] is designed to solve combinatorial searching problems in parallel by a branch-and-bound algorithm with the best-first search strategy. The branch-and-bound algorithm partitions a large problem into many smaller subproblems until a solution is obtained or infeasibility is found. The best-first search strategy selects the most feasible subproblem based on heuristic values assigned to subproblems. As the best-first search requires more memory spaces, especially in parallel processing environment, a special care must be taken for efficient management of the memory space. MANIP exploits a special virtual memory for this purpose [92].

After the problem is decomposed into many small subproblems, the subproblems stored in secondary storage is organized as a B^+-tree [9] whose leaf node represents a page containing one or more subproblems. The main memory maintains a heap of pointers to a partial list of subproblems also residing in main memory. When a subproblem is generated.

it is added to the list until the main memory space is run out. Then, the subproblems are inserted to the B^+-tree.

MANIP-2 [33] is designed for parallel heuristic evaluation of logic programming with a similar architecture to MANIP except for the absence of the secondary storage redistribution network which exists in MANIP between secondary storage and the subproblem memory controller shared by a number of processors. The global data resister in MANIP is also changed to the global broadcast bus in MANIP-2. The SLD resolution can be represented by a OR-tree as in the B-LOG, where a node denoting a subproblem can be a goal, or by a AND/OR tree where each node can be either a goal or a subgoal. Due to the AND/OR tree representation, the calculation of heuristic values of MANIP-2 is more complex than that of B-LOG since weights should be added to both AND and OR node.

5.2.2 MPPM

Yokota and Itoh [91] proposed *Relational Knowledge Base Model* which can accommodate unrestricted Horn clauses (general rules). Formally, a tuple d in a relational database is represented as follows:

$$d \in R \subset D_1 \times D_2 \times \ldots \times D_n$$

Here D_i is a set of constants and R is called a relation. On the other hand, a tuple t in a relational knowledge base has two attributes of the form:

$$t \in T \subset H \times B$$

where H is a general term representing the head of a clause, and B is a conjunction of terms representing the body of a clause. Here, T is called a *term relation*.

The basic operation proposed by Yokota and Itoh includes unification-restriction which is based on unification on data stream [38], and unification-join which is actually a parallelized, top-down resolution procedure. The algorithm of the proposed procedure is given in Figure 11.

The first step of the retrieval-by-unification is to find candidate tuples in a term relation for a given goal. This is done by unification-restriction operation. The unification-restriction operation is invoked for the term relation T by the condition

$$T(Head) \diamond goal$$

```
/* R: result; T: knowledge base; $T_i$: temporary relations;
   $\sigma$: selection; $\pi$: projection; $\diamond$: unification; $\bowtie$: join */
$R := \phi$;
$T_0 := \sigma_{head \diamond goal}(T)$;
$i := 0$;
while $T_i \neq \phi$ do
   begin
      $R := \sigma_{body=[]}(T_i) \cup R$;
      $T_{i+1} := \pi_{T_i.head, T.body}(T_i \; body \overset{\bowtie}{\diamond} head \; T)$;
      $i := i + 1$
   end;
```

Figure 11: The Algorithm of Retrieval-By-Unification

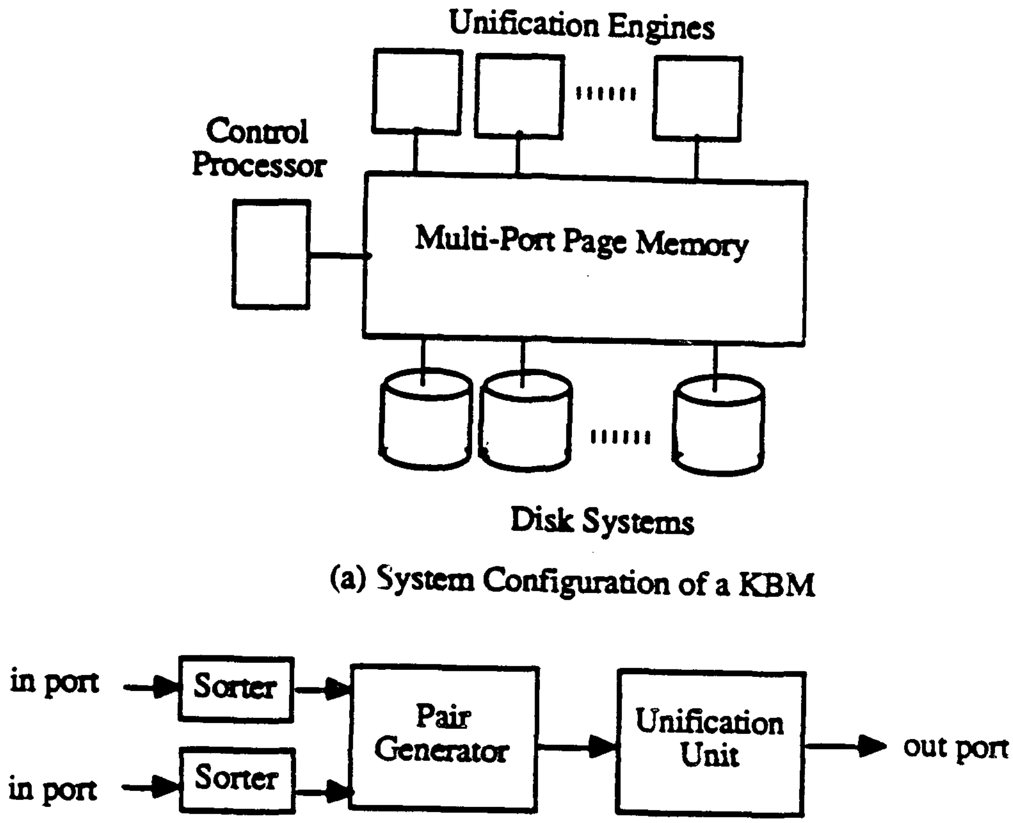
and makes a new term relation T_1. Here the symbol $\diamond$ denotes unification. Then, the unification-join operation between the second attribute of T_1 and the first attribute of T is invoked. The new relation, T_2 is then produced as a result of unification-join and projection. This procedure continues until no more new results are generated.

The advantage of the term relation stems from the uniform management of both rules and facts. However, efficient *indexing schemes* would be very difficult to develop since the entire knowledge base can be viewed as one large relation. In addition, the top-down (backward-chaining) processing strategy may result in a poor performance for large knowledge bases.

For the relational knowledge base model, an integrated knowledge base machine with specialized unification units is designed. The proposed knowledge base machine consists of a number of unification engines (UE), a control processor, a multi port page memory, and a number of disk systems. Figure 12 shows the basic architecture of MPPM.

One of the distinct features of the proposed knowledge base machine is the use of multi port page memory (MPPM) designed by Tanaka [76] which provides very high bandwidth among unification engines and disk systems. The MPPM consists of a set of I/O ports, a set of memory banks, and a switching network (Omega network) connecting the ports with the memory banks. The Unification Engine (UE) has three channels connected to the MPPM, and processes data streams from the MPPM in a pipelined fashion. The control processor is responsible for the parallel control of the UE's and the disk systems.

The unification-join can also be used for bottom-up processing like the equi-join opera-



(a) System Configuration of a KBM

(b) Unification Engine (UE)

Figure 12: MPPM with Multiple Unification Engines

tion in a relational database [59]. In this case, the equality check of the equi-join is replaced by unification. Zaniolo proposed different general term handling methods called Extended Relational Algebra (ERA) for bottom-up processing of relations of general terms [93,94]. In ERA, accesses to the subcomponents of terms are allowed, but recursive rules are not considered.

5.3 Associative Processors

During past decade, associative processors and associative memories have played very important roles in database machines. To mange both facts and rules by associative processors, effective internal representations of rules and facts in associative memories should be developed.

5.3.1 HAS

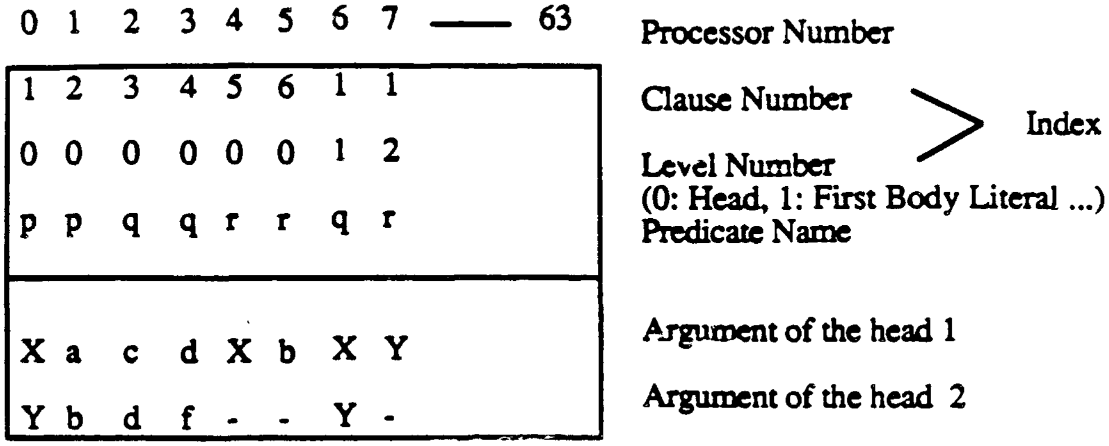
The Hybrid Associative Store (HAS) consists of 64 processing elements each of which is associated with a large number of memory cells [23]. The resulting architecture can be viewed as a quasi-associative processor, since all the PE's work synchronously by a central control unit, that is, all the PE's simultaneously access the same locations of memory cells. Each PE has an *ALU*, a *number* of registers (A,B, and register file C), and a mask register M.

To illustrate the management of rules in HAS, consider, for example, the following rules and facts:

1. p(X, Y) ← q(X, Y), r(Y).
2. p(a, b).
3. q(c, d).
4. q(d, f).
5. r(X) ← s(X, X).
6. r(b).

Figure 13 shows the basement store for the above logic program. Suppose that a goal ← p(a, Z) is given. Each literal in the program is stored together with the line number and the level number (i.e. head = 0, first subgoal = 1, second subgoal = 2, ...). The evaluation process is as follows:

1. Find heads of clauses which begin with $< *, 0 >$. Then, unify the heads with the given goal. In our example, the results are $< 1, 0 >$ *and* $< 2, 0 >$.



0	1	2	3	4	5	6	7	—— 63	Processor Number
1	2	3	4	5	6	1	1		Clause Number
0	0	0	0	0	0	1	2		Level Number (0: Head, 1: First Body Literal ...)
p	p	q	q	r	r	q	r		Predicate Name
X	a	c	d	X	b	X	Y		Argument of the head 1
Y	b	d	f	-	-	Y	-		Argument of the head 2

Figure 13: Data Organization of HAS

2. Find literals starting with < 1, 1 > and < 2, 1 > for the first subgoals.

3. Bind variables. Variables are substituted after all possible candidates are searched.

Although associative searching can be very effective for parallel evaluation of logic programs, the size limitation of associative memories makes it difficult to be used for arbitrarily complex objects.

5.3.2 ASSIP-T

ASSIP-T consists of a master processor, two associative memories (CAM1 and CAM2), multiple inference processors and unification processors [60] (Figure 14). The main principle of ASSIP-T is to separate unification operations from resolution by maintaining two graphs called a ***deduction plan*** and ***a unification graph with constraints*** (UwC). Nodes of a deduction plan denote goals, and edges are classified into two types, SUB and RED, according to the type of inference performed on the resolution step. Associated with each edge of the deduction plan is a set of constraints of the form $\{(t_1, s_1, \{n\}), \ldots, (t_m, s_m, \{n\})\}$, assuming that the starting edge is of the form p$(t_1, \ldots, t_m)$, the ending edge is p$(s_1, \ldots, s_m)$, and the edge number (i.e. resolution step) is n. The corresponding UwC is constructed by forming a labeled undirected graph from the constraint. That is, for the constraint (t, s, d), the nodes t and s is connected by a edge labeled d. Unification failure is detected when a variable node in the UwC leads to different constant values. When a unification failure

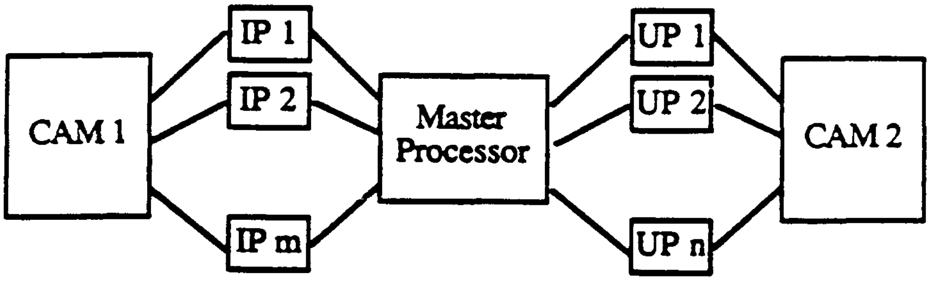


Figure 14: The Architecture of ASSIP-T

occurs, a minimal conflict set of edges are found and by removing one of the edges in the minimal conflict set, a successful unification can be obtained.

The master processor gets the constraints from the inference processors and distributes them to the unification processors where the conflict sets are calculated. The CAM2 stores each term for the UwC with term names, labels, and the adjacent node in the UwC.

6 Conclusion

Knowledge base systems require considerably more processing power than traditional database systems or AI systems do. The management of general rules and persistent complex objects is the most important task in these systems, which requires an enormous processing load. But advances in hardware technology make it feasible to design specialized computer architectures suitable for knowledge base processing to achieve high performance. In this paper, we surveyed computer architectures designed for logic-oriented data/ knowledge bases based on the taxonomy presented in section 3. We classified knowledge base machines into two large classes, investigated the characteristics of the machines in each class. and then presented special features of various proposed architectures. The operations that can be efficiently supported by these machines' specialized hardwares are also identified.

In order to support general rules in data intensive applications. database machines need to support the LFP operations. The machines used as front-ends which are responsible for generating commands for back-end database machines using internal rules are either general purpose or simple inference machines based on logic programming interpreters. Almost none of high performance Prolog machines designed for optimized compilers or unification machines are considered as part of knowledge base machines for data intensive applications. although these machines can provide order of magnitude speed ups over general purpose computers. Massively parallel architectures and associative processors are proposed as integrated knowledge base machines where rules as well as facts can be uniformly handled. However, for theses machines, transferring data from secondary storage to a processing level can be a serious bottleneck. NON-VON and CM-2 have specialized disk systems to solve this problem.

Another important issue in knowledge base machines is the management of a large number of complex objects stored on secondary storage. The traditional file organizations would not be well suited for the following reasons:

- Retrieving the desired tuples of knowledge bases can be viewed as an extension of the multiple-key attribute partial match retrieval problem because any subset of argument position can be specified in a query. Furthermore, attribute values are decomposable, and hence a query can be based on subcomponents of an argument (e.g. f(r(a, X), Y)). For example, if a fully inverted list is to be used, all the subcomponents should be indexed along with the position in a term. It could result in substantial amount of index data.

- The ordering among general terms cannot be decided due to the logical variables. Thus, we cannot use an ordered file organization such as the B-tree or the indexed sequential file.

- Most traditional data organization schemes do not lend themselves easily to parallel processing.

Thus, one of the important issue in designing a computer architecture for data intensive applications is to develop an effective physical data organization tailored to available hardware components to reduce data transfer rate as seen in CASSM and MPDC.

References

[1] S. Abe, T. Bandoh, S. Yamaguchi, K. Kurosawa, and K. Kiriyama. High performance Integrated Prolog Processor IPP. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 100–107, 1987.

[2] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, pages 110–117, 1979.

[3] P. B. Berra et al. *Implementing Knowledge Base Management Systems Based on Surrogate Files.* Technical Report, Syracuse University, October 1988. Unpublished Report.

[4] H. Boral. Design considerations for 1990 database machines. In *COMPCON Spring '86*, pages 370–373, 1986.

[5] H. Boral and D. J. Dewitt. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Proceedings of 3rd International Workshop on Database Machines*, pages 166–187, 1983.

[6] H. Boral and S. Redfield. Database machine morphology. In *Proceedings of 11th International Conference on VLDB*, pages 59–71, 1985.

[7] D. Champeaux. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences*, 32:79–90, 1986.

[8] K. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[9] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121 - 137, June 1979.

[10] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, Massachusetts, 1987.

[11] J. Crammond. A comparative study of unification algorithms for OR-parallel execution of logic languages. *IEEE Transactions on Computers*, C-34(10):911–917, October 1985.

[12] D. DeGroot. Restricted AND-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478, November 1984.

[13] T. Dorby. A coprocessor for AI: LISP, Prolog and data bases. In *COMPCON Spring '87*, pages 396–402, February 1987.

[14] T. P. Dorby, A. M. Despain, and Y. N. Patt. Performance studies of a Prolog machine architecture. In *Proceedings of the 12th Symposium on Computer Architectures*, pages 180–190, June 1985.

[15] T. P. Dorby, Y. N. Patt, and A. M. Despain. Design decisions influencing the microarchitecture for a Prolog machine. In *Micro 17 Proceedings*, pages 217–231, 1984.

[16] C. Dwork, P. Kanellakis, and J. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[17] C. Dwork, P. Kanellakis, and L. Stockmeyer. Parallel algorithms for term matching. In *Proceedings of the 8th Conference on Automated Deduction*, pages 416–430, 1986.

[18] B. S. Fagin and A. M. Despain. Performance studies of a parallel Prolog architecture. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 108–116, 1987.

[19] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of SIGMOD '87*, pages 260–269, 1987.

[20] K. Fuchi and K. Furukawa. The role of logic programming in the fifth generation computer project. In *Proceedings of the Third International Conference on Logic Programming*, pages 1–24, July 1986.

[21] R. Gonzalez-Rubio, J. Rohmer, A. Bradier, and B. Bergsten. DDC: a deductive database machine. In *Proceedings of the Fifth International Workshop on Database Machines*, pages 116–129, 1987.

[22] A. Goto and S. Uchida. Toward a high performance parallel inference machine – the intermediate stage plan of PIM –. In *Proceedings of the Advance Courses on Future Parallel Computers*, pages 299–320, June 1986.

[23] K. Hahne, P. Pilgram, D. Schuett, H. Schweppe, and G. Wolf. Associative processing in standard and deductive databases. In D. J. DeWitt and H. Boral, editors, *Database Machines - Fourth International Workshop*, pages 1–12, Springer-Verlag, 1985.

[24] J. Han. *Pattern-Based and Knowledge-Directed Query Compilation for Recursive Data Bases.* PhD thesis, University of Wisconsin, 1985.

[25] J. Harland and J. Jaffar. On parallel unification for Prolog. *New Generation Computing*, 5:259–279, 1987.

[26] B. K. Hillyer and D. E. Shaw. Execution of OPS5 production systems on a massively parallel machine. *Journal of Parallel and Distributed Computing*, 3:236–268, 1986.

[27] K. Hinrichs. Implementation of the grid file: design concept and experience. *BIT*, 25:569–592, 1985.

[28] K. Hwang, J. Ghosh, and R. Chowkwanyun. Computer architectures for artificial intelligence processing. *IEEE Computer*, 20(1):19–27, January 1987.

[29] N. Ito et al. The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 149–156, 1986.

[30] P. C. Kanellakis. Logic programming and parallel complexity. In *Proceedings of the International Conference on Database Theory*, pages 1–30, September 1986.

[31] C. Kellog. From data management to knowledge management. *IEEE Computer*, 19(1):75–84, January 1986.

[32] D. Li. *A Prolog Database System.* Research Studies Press, London, 1984.

[33] G. Li and B. W. Wah. MANIP-2: a multicomputer architecture for evaluating logic programs. In *Proceedings of the International Conference on Parallel Processing*, pages 123–130, 1985.

[34] G. J. Lipovski. Semantic paging on intelligent disks. In *Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing*, pages 30–34, 1978.

[35] G. J. Lipovski and M. V. Hermenegildo. B-LOG: a branch and bound methodology for the parallel execution of logic programs. In *Proceedings of International Conference on Parallel Processing*, pages 123–130, 1985.

[36] A. Martelli and U. Montanary. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[37] H. Matsuda et al. Implementing parallel Prolog system on multiprocesor system PARK. In *Proceedings of the Fifth International Workshop on Database Machines*, pages 640–653, 1987.

[38] Y. Morita, H. Yokota, and H. Itoh. Retrieval-by-unification operation on a relational knowledge base. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 52–59, August 1986.

[39] T. Moto-oka et al. The architecture of a parallel inference engine - PIE -. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 479–488, November 1984.

[40] K. Murakami et al. Research on parallel machine architecture for fifth-generation computer systems. *IEEE Computer*, 18(6):76–92, June 1985.

[41] R. Nakazaki et al. Design of a high-speed Prolog machine (HPM). In *Proceedings of the 12th International Symposium on Computer Architectures*, pages 191–197, June 1985.

[42] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[43] J. Noye and J.-C. Syre. ICM3: design and evaluation of an inference crunching machine. In *Proceedings of the 5th International Workshop on Database Machines*, pages 1–14, October 1987.

[44] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[45] E. Ozkarahan. *Database Machines and Database Management*. Prentice– Hall, 1986.

[46] E. A. Ozkarahan and C. H. Bozsahin. Join strategies using data space partitioning. *New Generation Computing*, 6:19–39, 1988.

[47] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[48] C. G. Ponder and Y. N. Patt. Alternative proposals for implementing Prolog concurrently and implications regarding to their respective micro architecture. In *Micro 17 Proceedings*, pages 192–202, 1984.

[49] G. Z. Qadah. Database machines: a survey. In *National Computer Conference*. pages 212–223, 1985.

[50] R. L. Ribler. The integration of the Xenologic X-1 artificial intelligence coprocessor with general purpose computers. In *COMPCON Spring '87*, pages 403–407, February 1987.

[51] J. A. Robinson. Machine oriented logic based on resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[52] P. Robinson. The SUM: an AI coprocessor. *Byte*, 10(6):169–180, June 1985.

[53] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4:273–285, 1986.

[54] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: a query language for $\neg$1*NF* relational databases. *Information Systems*, 12(1):99–114, 1987.

[55] G. B. Sabbatel and W. Dang. Search strategy for Prolog data bases. In *Proceedings of the 5th International Workshop on Database Machines*, pages 654–667. October 1987.

[56] G. B. Sabbatel and W. Dang. Unification for a Prolog data base machine. In *Proceedings of the 2nd International Logic Programming Conference*, pages 207–217. 1984.

[57] G. B. Sabbatel, J. C. Ianeselli, and G. T. Nguyen. A Prolog data base machine. In *Proceedings of the 3rd International Workshop on Database Machines*, pages 267 – 276, 1983.

[58] H. Sakai et al. Development of Delta as a first step to a knowledge base machine. In A. K. Sood and A. H. Qureshi, editors, *Database Machines*, pages 159–181, Springer-Verlag, 1986.

[59] H. Sakai, S. Shibayama, H. Monoi, Y. Morita, and H. Itoh. A simulation study of a knowledge base machine architecture. In *Proceedings of the 5th International Workshop on Database Machines*, pages 583–596, October 1987.

[60] H.-A. Schneider and W. Dilger. Information processing with associative processors. In *Proceedings of the Conference on Algorithms and Hardware for Parallel Processing*, pages 222–229, September 1986.

[61] E. Shapiro. Concurrent Prolog: a progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[62] D. Shaw. Relational query processing on the Non-Von supercomputer. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 248–258, Springer-Verlag, 1985.

[63] Y. Shobatake and H. Aiso. A unification processor based on a uniformly structured cellular hardware. In *Proceedings of the 13th International Symposium on Computer Architectures*, pages 140–148, 1986.

[64] C. Stanfill and B. Kahle. Parallel free-text search on the Connection Machine system. *Communications of the ACM*, 29(12):1229–1239, December 1986.

[65] S. J. Stolfo. Initial performance of the DADO 2 prototype. *IEEE Computer*, 20(1):75–83, January 1987.

[66] H. S. Stone. Parallel querying of large databases: a case study. *IEEE Computer*, 20(10):11–21, October 1987.

[67] C. D. Stormon. *An Associative Processor and Its Application to Logic Programming Computation*. Technical Report 8611, Syracuse University-CASE Center, October 1986.

[68] S. Y. W. Su. *Database Computers - Principles, Architectures, and Techniques*. McGraw-Hill, 1988.

[69] S. Y. W. Su and A. Emam. CASDAL: CASSM's DAta Language. *ACM Transactions on Database Systems*, 3(1):57–91, March 1978.

[70] S. Y. W. Su, L. H. Nguyen, A. Emam, and G. J. Lipovski. The architectural features and implementation techniques of the multicell CASSM. *IEEE Transactions on Computers*, C-28(6):430–445, June 1979.

[71] N. Tamura et al. Sequential Prolog machine PEK. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 542–550, November 1984.

[72] M. Tanabe and H. Aiso. The unification processor by pipeline method. In *Proceedings of the 5th International Workshop on Database Machines*, pages 668–680, October 1987.

[73] H. Tanaka. A parallel inference machine. *IEEE Computer*, 19(5):48–54, May 1986.

[74] Y. Tanaka. A data-stream database machine with large capacity. In D. K. Hsiao, editor, *Advanced Database Machine Architecture*, pages 168–202, Prentice-Hall, 1983.

[75] Y. Tanaka. Massive parallel database computer MPDC and its control schemes for massively parallel processing. In A. K. Sood and A. H. Qureshi, editors, *Database Machines*, pages 127–158, Springer-Verlag, 1986.

[76] Y. Tanaka. A multiport page-memory architecture and a multiport disk-cache system. *New Generation Computing*, 2:241–260, 1984.

[77] S. Taylor et al. Logic programming using parallel associative operations. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 58–68, February 1984.

[78] S. Taylor, C. Maio, S. J. Stolfo, and D. E. Shaw. *Prolog on the DADO Machine: A Parallel System for High-Speed Logic Programming*. Technical Report CUCS-46-83, Columbia University, 1983.

[79] *Connection Machines Model CM-2 Technical Summary*. Thinking Machines Co., April 1987. Thinking Machines Technical Report HA87-4.

[80] E. Tick and D. H. D. Warren. Toward a pipelined Prolog processor. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 29–40, February 1984.

[81] P. C. Treleaven and A. N. Refenes. Computer architecture for artificial intelligence. In *Proceedings of the Advance Courses on Future Parallel Computers*, pages 416–492, June 1986.

[82] K. Ueda. Guarded Horn Clauses. In *Proceedings of the 4th Conference on Logic Programming*, pages 168–179, 1985.

[83] J. S. Vitter and R. A. Simons. New classes for parallel complexity: a study of unification and other complete problems for *P*. *IEEE Transactions on Computers*, C-35(5):403–418, 1986.

[84] B. W. Wah, G. Li, and C. F. Yu. The status of MANIP – a multicomputer architecture for solving combinatorial extremum-search problems. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 56–63, 1984.

[85] B. W. Wah and G.-J. Li. A survey on special purpose computer architectures for AI. *SIGART Newsletter*, 96:28–46, April 1986.

[86] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 306, SRI International, October 1983.

[87] M. J. Wise. Experimenting with EPILOG: some results and preliminary conclusions. In *Proceedings of the 13th Symposium on Computer Architectures*, pages 130–139, 1986.

[88] N. S. Woo. The architecture of the hardware unification unit and an implementation. In *Micro 18 Proceedings*, pages 89–98, 1985.

[89] N. S. Woo. A hardware unification unit: design and analysis. In *Proceedings of the 12th International Symposium on Computer Architectures*, pages 198–205, June 1985.

[90] H. Yasuura. On parallel computational complexity of unification. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 235–243, 1984.

[91] H. Yokota, K. Sakai, and H. Itoh. Deductive database system based on unit resolution. In *Proceedings of the 2nd Data Engineering*, pages 228–235, 1986.

[92] C. F. Yu and B. W. Wah. Virtual memory support for branch-and-bound algorithms. In *COMPSAC*, pages 618–626, 1983.

[93] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings of the 11th International Conference on VLDB*, pages 21–23, August 1985.

[94] C. Zaniolo. Safety and compilation of non-recursive Horn clauses. In *Proceedings of the First International Conference on Expert Databases*, pages 167–178, April 1986.

Appendix 12-D

Key-Sequential Access Methods for Very Large Files Derived from Linear Hashing

Nabil I. Hachem

P. Bruce Berra

Dept. of Electrical and Computer Engineering
Syracuse University
Syracuse, NY 13244-1240
(315) 423-4445
hachem@sutcase.syr.edu
berra@sutcase.syr.edu

ABSTRACT

In this paper a new class of order preserving dynamic hashing structures is introduced and analyzed. The access method is referred to as dynamic random-sequential access method (DRSAM) and is derived from linear hashing. With respect to previous methods DRSAM presents the following characteristics: 1) the structure captures the hashed order in consecutive storage areas so that order preserving schemes result in performance improvements for range queries and sequential processing. 2) It adapts elastic buckets [LOM87] for the control of file growth. This approach outperforms the partial expansion method previously proposed by Larson [LAR82]. The file structure is also extended with proper control mechanisms to cope with non-uniform distributions. The outcome is a multi-level trie stored as a two-level sequentially allocated file.

Index Terms: dynamic file structures, order preserving hashing, access method, random and sequential files, consecutive retrieval, searching, management of very large files.

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

1. Introduction

Advances in hashing methods led to the development of new file structures to handle volatile files [LAR78, FAG79, LIT80, LAR82, RAM82]. Referred to as dynamic hashing schemes, these methods are suitable for the management of very large files, with no prior or future knowledge of their sizes. The objective was the development of file organizations that, with reasonable insertion costs, keep their near optimal performance characteristics for random access even when the file size constantly changes.

An additional characteristic of file organizations, uncommon to hashing methods, is the sequential processing or range query capabilities as found in indexed-sequential files and B-trees [BAY72]. Based on linear hashing (LH), different order preserving hashing (OPH) methods were proposed: Orenstein [ORE83] used the mirror image of the leftmost bits with LH to achieve local order preserving while Burkhard [BUR83] proposed a multi-key retrieval order preserving method based on the shuffle order.

In this paper, a new class of files derived from linear hashing is presented. It is referred to as the dynamic random-sequential access method (DRSAM). The main characteristic of this access method over previous ones is the *sequential allocation property* which leads to the natural adaptation of *elastic buckets* [LOM87] to directoriless organizations. DRSAM shows performance improvements over previous methods for both direct access and range queries performance.

We begin in Section 2 with the file design objectives and the approach which is followed with DRSAM. In Section 3 an implementation of DRSAM is described and illustrated. In Section 4 we present the concept of elastic buckets (EB) as a technique to control file growth and for the management of overflow chains. In Section 5 we evaluate DRSAM's performance characteristics: first the performance with elastic buckets is analyzed followed by the effect of the insertion method on unsuccessful search cost. Then we compare the use of elastic buckets with respect to Larson's partial expansions method and discuss range queries with DRSAM files. Section 6 discusses possible extensions of DRSAM to cope with non-uniform distributions. Finally, in Section 7 we summarize the work and introduce future research topics.

2. Objectives and Methodology

Similarly to other dynamic hashing methods the primary objectives of DRSAM are to 1) achieve the near optimal random access performance and, 2) be able to easily and naturally adapt to dynamic environments with low update overhead (insertion, deletion, change). Additionally a third objective that we consider essential is for DRSAM to provide for efficient range queries and sequential processing. The basic idea with DRSAM is to map logical nodes to consecutive physical buckets to achieve, for range queries, the consecutive retrieval (C-R) property introduced by Ghosh [GHO86]. This assures a minimized disk arm movement and is referred to as the *consecutive or sequential allocation property in key-order.*

Previous OPH schemes have used the linear hashing expansion sequence. For example the idea in [ORE83] is to use the mirror image of the leftmost bits to achieve order preserving with linear hashing. This leads to the mapping of Figure 2.1. As the file grows the physical locations of buckets containing records in key-order are known but remotely located and range queries that access multiple buckets will require extra disk seeks for each retrieved bucket.

In Figure 2.2 we illustrate how DRSAM maintains physical ordering in the mapping from logical to physical structure. To achieve this mapping, one needs a *dynamic, sequential allocating* and *order preserving hashing* function (OPH_i). Using *Prefix(Key,i)* as the leftmost i bits of a key, a simple OPH_i for DRSAM is provided with:

$$OPH_i(Key) = Prefix(Key,i).$$

With fl as the home hash level, the logical address of a Key ,for the home hash function, is found by:

$$log\_add_{fl}(Key) = OPH_{fl}(Key)$$

and for the split function:

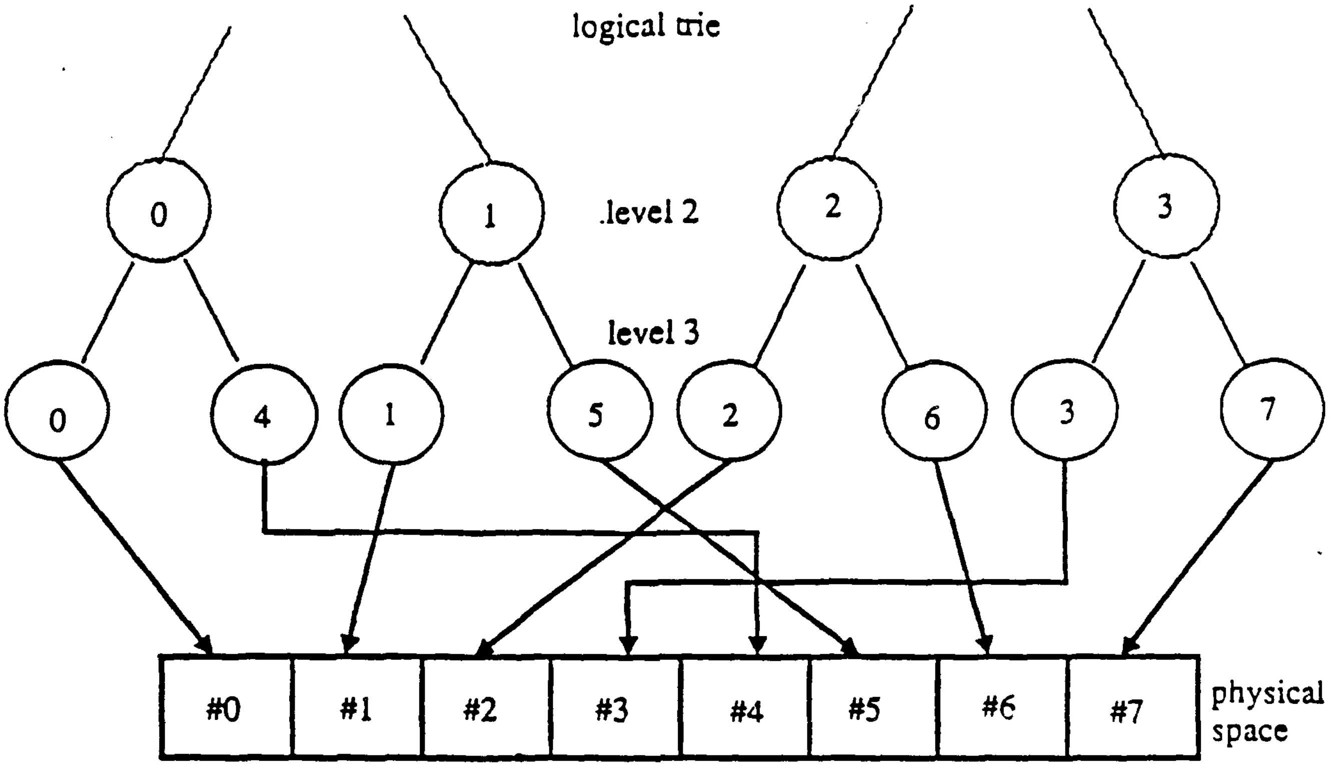


Figure 2.1 Linear hashing logical to physical mapping

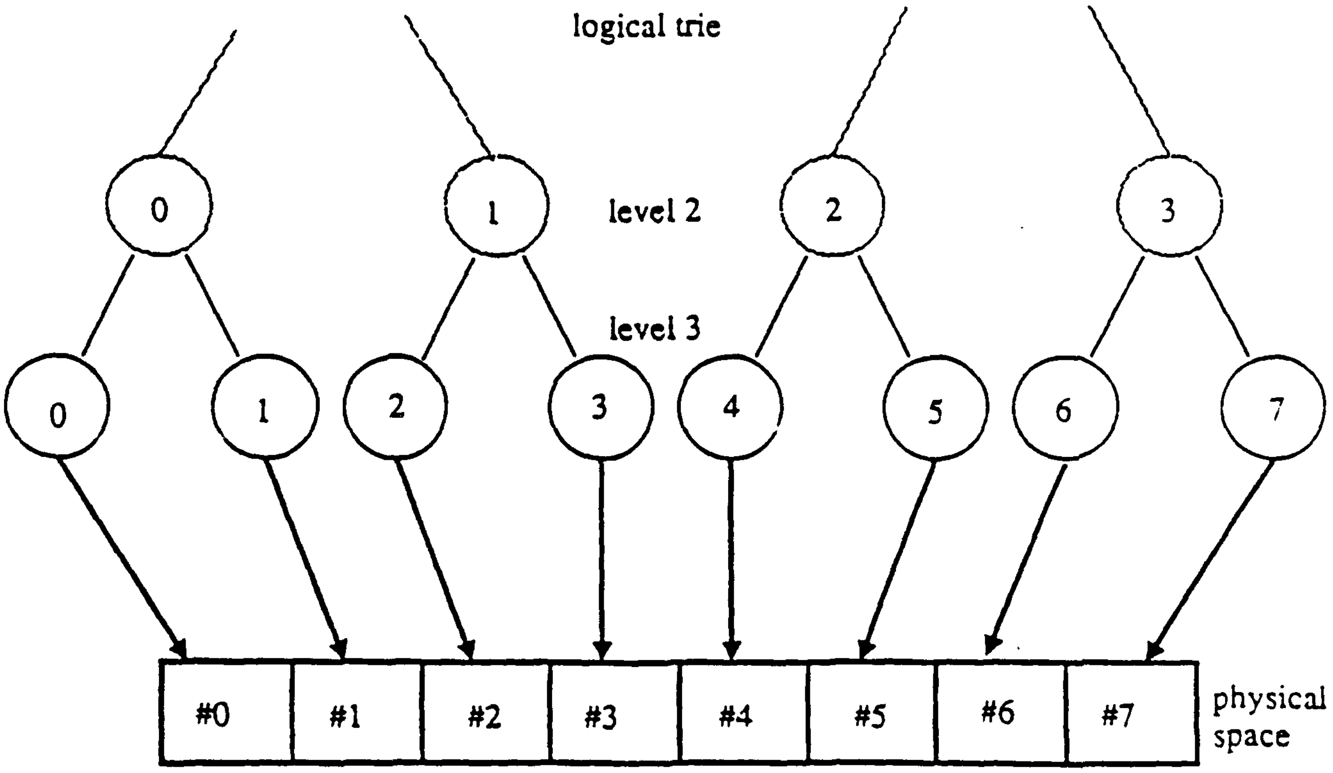


Figure 2.2 DRSAM logical to physical mapping

$$log\_add_{fl+1}(Key) = OPH_{fl+1}(Key)$$

The prefix function is chosen for convenience but is not generally considered to be a good randomizing function. We discuss non-uniform distributions in Section 6.

In linear hashing and DRSAM the file is logically viewed as a two-level flattened trie structure with its leaves at level *fl* or *fl+l*. We will denote a logical node as the tuple (x,y); where x is the node number and y its trie level. The mapping of Figure 2.2 is optimal for sequential processing in the sense that it tends to minimize disk arm movement. DRSAM files try to achieve this mapping and we illustrate the concept in the next section.

3. The Dynamic Random-Sequential Access Method

We designed and analyzed two DRSAM schemes, namely variant 0 and 1 [HAC88b]: DRSAM variant 0 relies on the same file management characteristics as linear hashing but with a modified expansion/contraction sequence to approach the mapping of Figure 2.2. Independently to our work, Hutflesz et al. introduced a similar idea for multidimensional OPH [HUT88]. We will only consider variant 1 in this paper and use DRSAM to refer to that file structure. As for linear hashing, the splitting bucket is not necessarily the one that receives the inserted record. Thus an overflow resolution procedure is necessary. In the illustrations of this section overflow chains are not shown.

DRSAM uses the same logical expansion/contraction sequence as linear hashing. To avoid physical bucket overwriting we slightly extend the operating system file management features. For an expansion, the basic idea is to relocate the logical bucket on a newly allocated larger storage space while the previous space is freed. This operation is irreversible and the freed bucket cannot be recorvered for a subsequent merge operation. Like linear hashing, the storage area is only allowed to expand (contract) from one of its boundaries. The extension is that the other boundary is allowed to move to return contiguous storage space to the operating system. For the design of DRSAM, this feature is an essential but realistic characteristic of the operating system.

3.1. Storage Areas

We define two storage areas: the type 1 area (T1), generally used for the expansion of the file and the type 2 area (T2) used concurrently with the contraction of the file. A T1 storage space can grow or shrink from its right boundary with its left boundary allowed to return storage to the operating system secondary storage pool while the T2 area operates in the reverse mode. The T1 area is illustrated in Figure 3.1. It shows the leftmost bucket expanding onto the rightmost (0,3) and (1,3) buckets.

(-,-)	(1,2)	(2,2)	(3,2)	(0,3)	(1,3)

Figure 3.1 An illustration of a type 1 storage area.

In Figure 3.2 we illustrate a T2 area after contracting the rightmost 2 buckets onto the leftmost bucket (1,1). File movement in both directions is not permitted.

(1,1)	(0,2)	(1,2)	(-,-)	(-,-)

Figure 3.2 An illustration of a type 2 storage area.

A storage area is always scanned from left to right on secondary storage. The left boundary in each area is the lowest logical bucket stored in the area and the physical addresses within a T1 (T2) area are defined with respect to its left (right) boundary. This will enable us read the home storage area in key-order through a sequential scan with minimum disk head movement.

3.2. The Mechanism for DRSAM

At a given time, the primary storage space of a DRSAM file can consist of a T1 and a

T2 area: the T1 storage area can consist of 2 subareas, the home (HL) and the expansion (E) areas while the T2 storage area can consist of the home (HR) and contraction (C) areas. In Figure 3.1 the HL area consists of buckets (1,2), (2,2) and (3,2) while (E) consists of buckets (0,3) and (1,3) and in Figure 3.2 the HR area consists of buckets (0,2) and (1,2) while (C) consists of bucket (1,1). Though not required, the subareas are assumed to be contiguously located on secondary storage.

As with LH we attach an expansion pointer (*ep*) which points to the logical address of the next bucket to expand. If we expand (contract) the file *ep* is *logically incremented* (decremented) by 1. We also need a *skew counter* (*sc*) to determine the logical boundary between the HL and HR areas. The expansion/contraction rules are designed to guarantee that the (E) and (C) areas cannot coexist. The rules are defined as:

Expansion

```
if "C exists" then ($e_1$) "expand from C to HL"
else if "HR exists" then ($e_2$) "expand HR to E"
    else ($e_3$) "expand HL to E"
```

Contraction

```
if "E exists" then ($c_1$) "contract from E to HR"
else if "HL exists" then ($c_2$) "contract HL to C"
    else ($c_3$) "contract HR to C"
```

For expansions, e_1 is used first until all the buckets in the C area are exhausted, then e_2 until HR is fully expanded and finally HL buckets are sequentially expanded. The same scheme holds for file contraction. Within a storage area we sequentially increase *ep*. It is easy to show that the "lower" logical buckets are in the C area, followed by the ones in the HR, then the HL and finally E areas. For a sequential scan of the file, if C exists then we scan HR, followed by HL and then C; else we scan E followed by HR then HL.

Consider the DRSAM file of Figure 3.3 with only a home area HL (sc=0) composed of 8 buckets with the expansion pointer pointing to logical node 0 (fl=3 and ep=0). Assuming 8 bit keys we include the different bucket hash ranges to show the key sequential order of the file (values in []). Through the addition of records, let us assume that a split condition occurs with the file. We use rule e_3 and follow the sequence of linear hashing. The leftmost bucket HL(0,3) splits and we relocate its contents using the split hash function as the first two buckets E(0,4) and E(1,4) of a newly created T1 expansion area (E). Bucket HL(0,3) is freed as shown in Figure 3.4.a and ep and sc are incremented by 1.

A 2nd expansion splits HL(1,3) by relocating its contents onto E(2,4) and E(3,4). The empty bucket is returned to the storage pool, ep and sc are incremented and we have the file status of Figure 3.4.b. Continuing with the expansion process, the file will be at level 4 after 8 expansions with the (E) area relabeled as (HL).

Referring to Figure 3.4.b let us assume that due to successive deletions, a merge is necessary. Referring to Figure 3.5.a buckets E(2,4) and E(3,4) are merged back together and relocated onto HR(1,3) in a newly created T2 home area (HR). Pointer (ep) is decremented by 1 while sc stays the same; sc actually points to the minimum logical bucket address in the HL area and thus $sc-1$ points to the maximum logical bucket address of the HR area. Rule c_1 was used here.

A second merge relocates the contents of buckets E(0,4) and E(1,4) onto HR(0,3) as shown in Figure 3.5.b. The file is at level 3 and at the beginning of the expansion cycle (ep=0). The existence and size of HR is determined by the skew pointer (sc=2). This file is logically equivalent to the one in Figure 3.3 though physically different (different sc values). With the knowledge of ep, sc, the home level fl, and the existence or non existence of the areas C and E, we can uniquely determine the physical and logical status of the file.

Following a similar example, we can show how the file can use a contraction area (C) with rules c_2 and c_3. The difference between the algorithms of DRSAM and linear hashing

(HL)

(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
[0-31]	[32-63]	[64-95]	[96-127]	[128-159]	[160-191]	[192-223]	[224-255]

Figure 3.3 File with only the HL storage area at level 3

(HL)

(-,-)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
	[32-63]	[64-95]	[96-127]	[128-159]	[160-191]	[192-223]	[224-255]

(E)

(0,4)	(1,4)
[0-15]	[16-31]

Figure 3.4.a After one expansion from Figure 3.3.

(HL)

(-,-)	(-,-)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
		[64-95]	[96-127]	[128-159]	[160-191]	[192-223]	[224-255]

(E)

(0,4)	(1,4)	(2,4)	(3,4)
[0-15]	[16-31]	[32-47]	[48-63]

Figure 3.4.b After two expansion from Figure 3.3

(HL)

(-,-)	(-,-)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
		[64-95]	[96-127]	[128-159]	[160-191]	[192-223]	[224-255]

(E)

(0,4)	(1,4)
[0-15]	[16-31]

(HR)

(1,3)
[32-63]

Figure 3.5.a After one merge from Figure 3.4.b

(HL)

(-,-)	(-,-)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
		[64-95]	[96-127]	[128-159]	[160-191]	[192-223]	[224-255]

(HR)

(0,3)	(1,3)
[0-31]	[32-63]

Figure 3.5.b After two merges from Figure 3.4.b

are in the mapping of logical buckets onto physical storage space. These algorithms are straight forward and we implemented them for file expansion. Details are found in [HAC88b].

4. Generalizing DRSAM

Elastic buckets (EB) were first proposed by Lomet for indexed file organizations [LOM87] and further studied within the context of B^+-trees by Baeza et al. [BAE87]. In this section we use them to control DRSAM file growth and extend the concept to the management of the overflow area of DRSAM.

Partial Expansions

With partial expansions the size of the file is doubled over a full expansion cycle (FEC) composed of a number of partial expansion cycles (PEC). Within a PEC, Larson's partial expansions technique (LPE) uses a sequence of intermediate hashing functions to distribute the contents of a group of "x" buckets onto "x+1" buckets [LAR82]. The "x" buckets are reused with the addition of one "remote" bucket at the end of the file. A similar scheme was also proposed in [RAM82].

From the expansion mechanism of DRSAM, we found that elastic buckets are a natural extension to this file structure. Denote the number of PEC in an FEC by r and elastic logical nodes (ELN) with a triplet $(lba, fl, pexp)$; where lba is the node number at level fl and having completed PEC $pexp$. Figure 4.1.a through 4.1.d show the different partial expansion steps of a logical node "x" at level "i" for the case when r=3: initially, in Figure 4.1.a the node consists of 3 (r) partial physical buckets, and during the 1st and 2nd partial expansion, the node elastically expands by one *partial physical bucket* (pb). This is shown in Figures 4.1.b and 4.1.c.

For the last partial expansion, adding a new partial physical bucket doubles the capacity of the node with respect to its initial state of Figure 4.1.a. In this case, using the split hash function the bucket forks into two new child buckets "2x" and "2x+1" at level "i+1" and with the minimum capacity of 3 partial buckets. Over a PEC each bucket is elastically expanded and is relocated to a new physical address. This address is within the expansion area (E) if it exists or in the HL storage area if C exists.

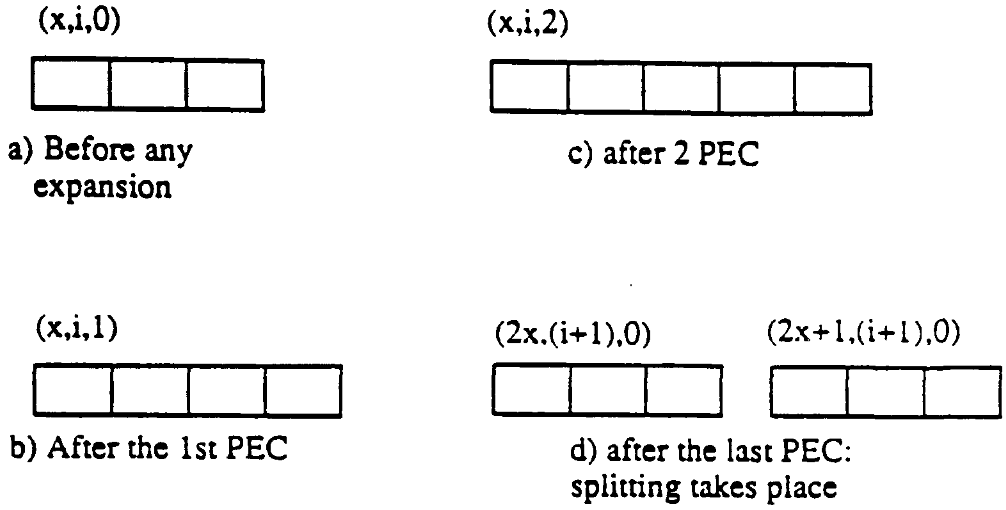
Two main differences between EB and LPE are observed: 1) elastic buckets are suitable for file expansions on physically contiguous locations while Larson's PE is devised for buckets that expand by relocating their contents onto non contiguous physical locations. 2) Elastic buckets do not require the use of an intermediate hashing function. Splitting the expanding bucket occurs during the last PEC of a FEC.

Elastic Overflow Chaining

The assumption of contiguous allocation for file expansion is easily extended to deal with overflow chains. The technique is referred to as *elastic overflow chains* and is a generalization of the usual overflow chaining used for collision resolution. The method assumes an overflow storage pool with capacities ob_0 to $ob_{el-1} = el \times ob_0$ records in increments of ob_0 records ($el \geq 1$). When an overflow bucket of capacity $ob_{i-1} = ob_0 \times i$ becomes full (i.e. overflows) its contents, with the inserted record, are written onto an overflow bucket of capacity $ob_i = ob_0 \times (i+1)$. This process is continued until we reach the capacity of ob_{el-1}. Then a new bucket of capacity ob_0 is attached to the chain thus increasing its length by one. An elastic overflow bucket for $el=3$ is illustrated in Figure 4.2. The usual overflow chaining method corresponds to the case where $el=1$.

5. Performance Evaluation of DRSAM

In this section we discuss the results of a performance evaluation of DRSAM with elastic buckets. We target the application of DRSAM to secondary key retrieval, like transformed inverted lists [BER87, HAC88a]. Thus we study the effect of ordered insertions on the performance characteristics of DRSAM. This is followed by a performance comparison of LPE and EB for partial expansions. Finally, sequential and range query processing with DRSAM files



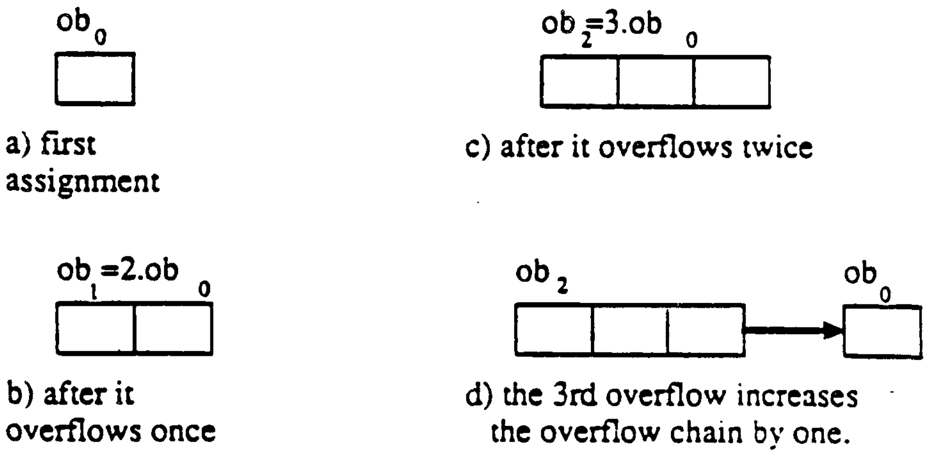
a) Before any expansion

c) after 2 PEC

b) After the 1st PEC

d) after the last PEC: splitting takes place

Figure 4.1 ELN during expansion (elasticity r= 3)



a) first assignment

c) after it overflows twice

b) after it overflows once

d) the 3rd overflow increases the overflow chain by one.

Figure 4.2 Elastic overflow node with el= 3

are discussed and compared to similar OPH methods.

5.1. System Model

We developed an analytical model for very large DRSAM files and performed extensive trace driven simulations of the file structure. The results from the analytical model closely match those from the simulations. As we are interested in secondary storage systems, the performance measures are in terms of disk access cost; where disk seek time is the predominant component in the cost measure. The details of the performance analysis can be found in [HAC88b].

The model assumes a uniform distribution over the hashed domain. Furthermore, main memory buffer storage is large enough to hold an accessed bucket with its overflow chain. For a random insertion one needs to perform at least one disk access to read the home bucket and subsequent reads to traverse its potential overflow chain. It is followed by a disk access to write back the bucket after inserting the new record at the end of the chain. With ordered insertions, the record may be inserted at any location in the chain with equal probability. This implies that, in general, one needs to write back more than one bucket. Elastic overflow chains are used and the cost to expand or create a new chain is always equal to two disk accesses (operating systems overhead not included).

With lfc as the number of record insertions between file expansions, the model follows the load factor control mechanism described in [RAM82]. The file is expanded by the addition of one partial bucket after every lfc inserted records. The load factor (lf) is almost constant and is computed as $\frac{lfc \times r}{hb}$ [*]; with hb as the minimum home bucket capacity. For an expansion operation we need at least one disk read to retrieve the bucket undergoing the expansion. This is followed by a disk access to write the generated blocks into consecutive

[*] The load factor is defined as: $lf = \frac{\#\ of\ records\ inserted}{home\ storage\ space\ in\ records}$.

locations on secondary storage. Obviously, the existence of overflow buckets increases the number of disk accesses accordingly. The performance parameters which were determined are the average of the storage utilization factor (*auf*), the average costs for a successful search (*acss*), unsuccessful search (*acus*), random insertion (*acri*) and ordered insertion (*acoi*).

5.1.1. Results and Discussion

In this section, we report some results from the analytical model for a home bucket capacity of hb=48 records.

Effect of Elastic Buckets

We illustrate the effect of the number of partial expansions (r) on storage utilization in Figure 5.1. With a fixed average load factor (lf = 1.25), the utilization factor increases with increasing r from an average of 0.91 at r=1 to 0.94 at r=3. To evaluate the retrieval performance, we chose the basic overflow bucket capacity (ob_0) such that the average storage utilization factor is around auf=0.91 for the different values of r. Then we plotted the successful and unsuccessful search costs in Figure 5.2 and 5.3 respectively. It is clear that elastic buckets improve the random access performance (*acss* and *acus*) of DRSAM.

A higher elasiticity for the overflow buckets does not affect the storage utilization of the file, but it results in a decrease of the overflow chain length. This tends to improve all performance parameters as reported in [HAC88b]. Figure 5.4 illustrates the effect of increasing the elasticity el of the overflow chains on successful search cost. For a sustained average storage utilization of 93%, the average successful search cost improves with an increase in the elasticity of the overflow chains. The variations of the performance curves over a FEC decrease as well.

With a storage utilization factor around 94%, hb=48, r=3 and el=4, our results show that the performance of DRSAM is excellent with $acss$ - 1.20, $acus$<2.00 and $acri$<3.25. Considering that the performance of file organizations degrade very fast with increasing

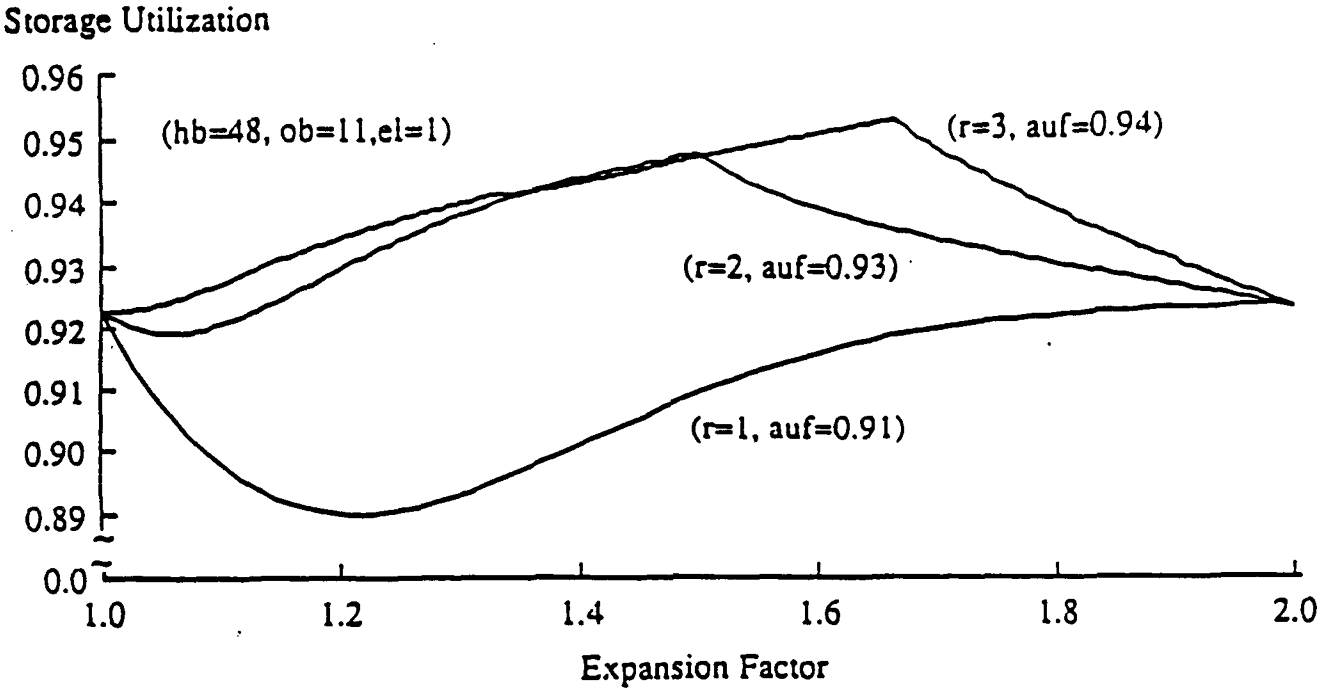


Figure 5.1 DRSAM storage utilization (r=1 to 3)

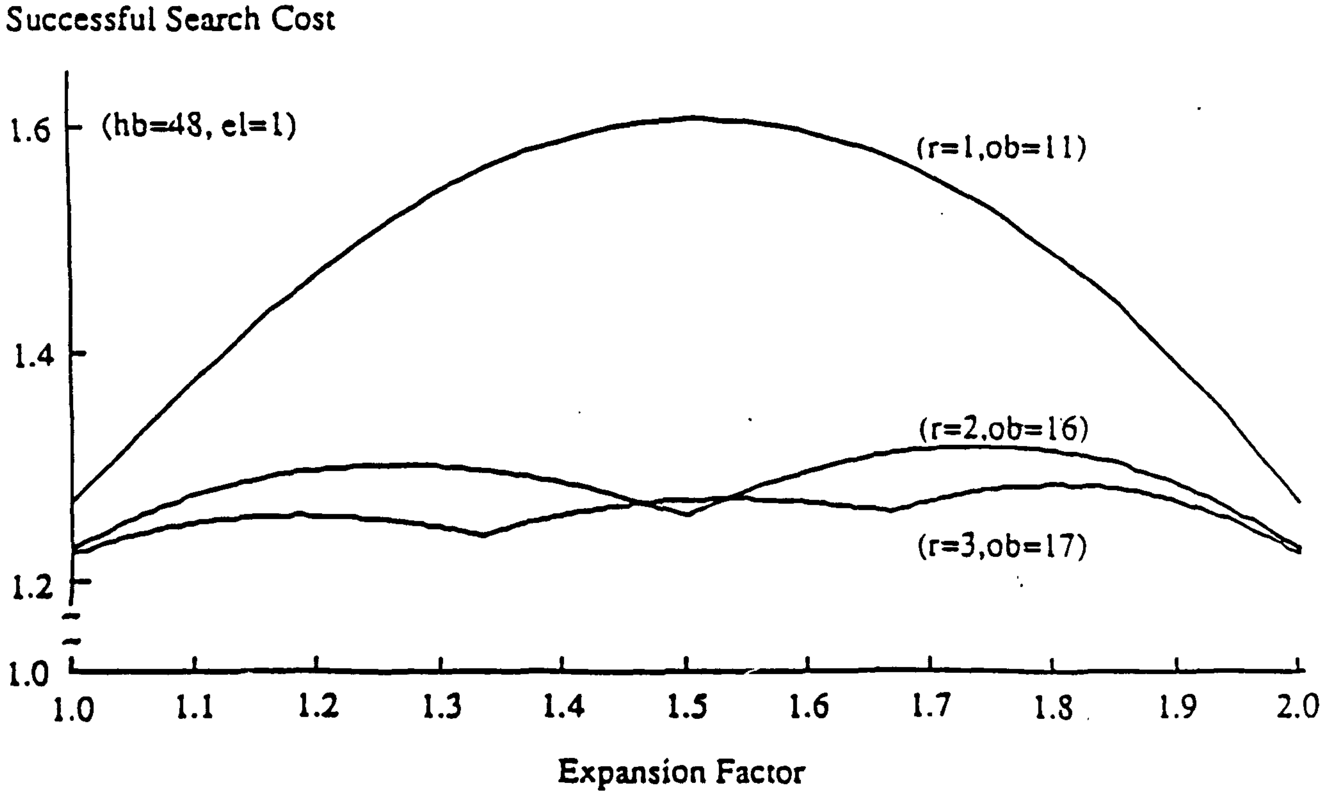


Figure 5.2 DRSAM successful search cost (auf = 0.91)

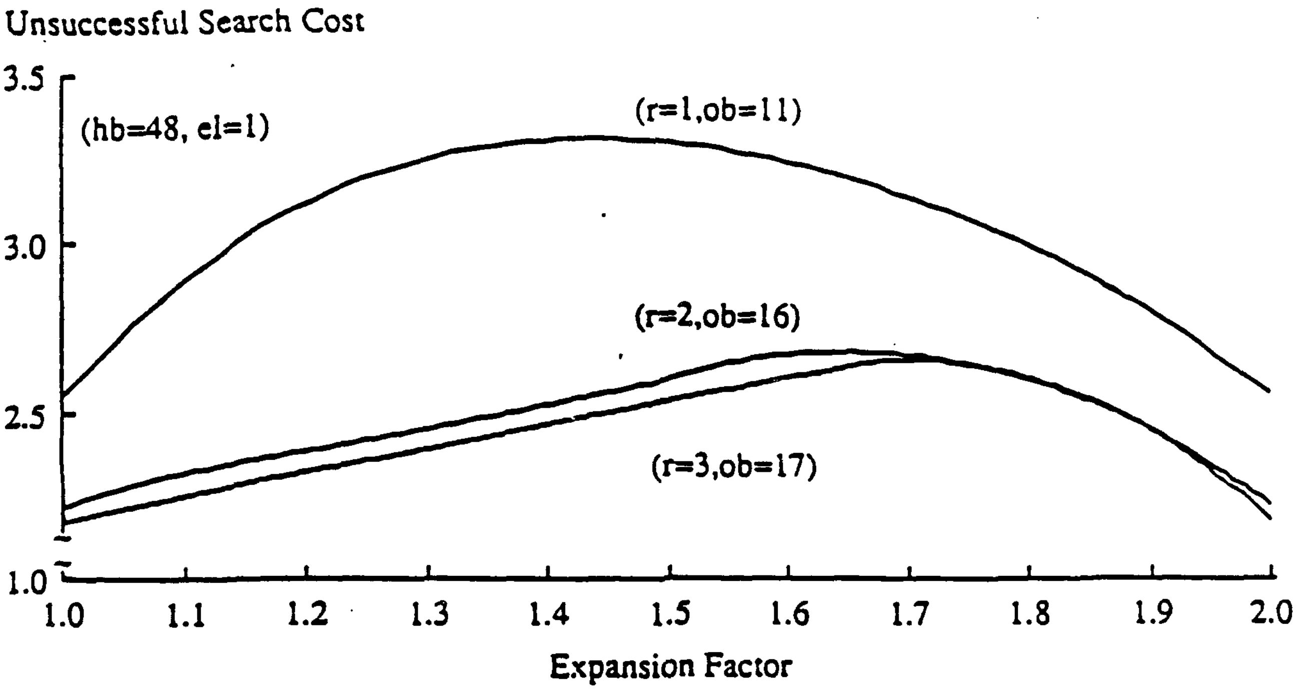


Figure 5.3 DRSAM unsuccessful search cost (auf = 0.91)

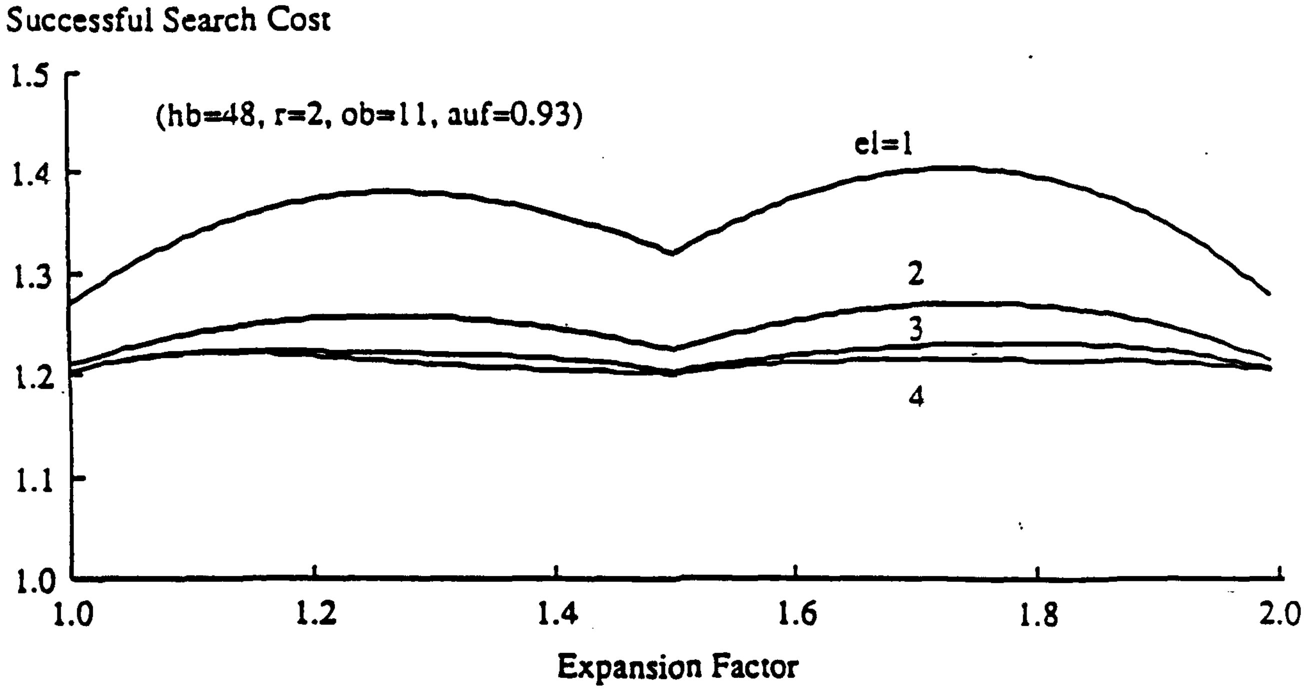


Figure 5.4 DRSAM successful search cost (el= 1 to 4)

utilization factors makes the elastic buckets techniques an important extension to DRSAM file structures and to non indexed dynamic hashing schemes in general.

Effect of Insertion Methods on the Unsuccessful Search

We chose the case where hb=48 records and r=3 to compare the random and ordered insertion methods. In Table 5.1 we report the average insertion and average unsuccessful search costs for el=1 and el=4. With el=4 ordered insertions result in $acus$ = 1.196 disk access; while it is equal to 1.934 if random insertions are used. This is an improvement in performance of 38%. On the other hand, the average insertion cost increases by 20%. The cumulative effect of ordered insertions and elastic overflow buckets is to improve unsuccessful search cost from 2.966 to 1.196 (a factor of 2.48).

hb=48, ob_0=11, r=3, lfc=20, auf=0.94				
insertion	el=1		el=4	
method	acus	insertion cost	acus	insertion cost
random	2.966	4.372	1.934	3.227
ordered	1.312	6.006	1.196	3.883

Table 5.1 Effect of the insertion method on the cost of an insertion and unsuccessful search

5.2. Elastic Buckets Versus Larson's Partial Expansion

To compare Larson's partial expansion (LPE) with elastic buckets (EB) we used DRSAM as the basic file structure. The test case was for hb=48 records, r=2, el=1 and ob_0=11 records; lfc was adjusted to result in the same average storage utilization for both methods (auf ~0.93). Table 5.2 shows the averages for LPE and EB.

$hb=48$, $ob_0=11$, r=2, el=1				
	auf	*acss*	*acus*	*acri*
LPE	0.932	1.567	3.496	4.820
EB	0.933	1.359	2.949	4.253

Table 5.2 Comparison of LPE and EB for partial expansions

From the results in Table 5.2, it is clear that elastic buckets outperform Larson's partial expansions scheme. However, the cost paid is a wider fluctuation in storage utilization factor. This is clear from Figure 5.5 where the *auf* for EB has a peak fluctuation of – 1.5% around the average while the *auf* for LPE is almost constant. Figure 5.6 compares the successful search cost for EB and LPE. LPE required a higher average load factor in order to achieve an equivalent average storage utilization to EB. This causes lengthier overflow chains to exist and explains the poorer retrieval performance.

It is noted that the performance measure is in terms of random disk access cost and does not account for differences in data transfer time. For EB the transfer time is higher due to the use of variable bucket capacities. But the difference in performance which is observed in Figure 5.5 and Table 5.3 is large and the unaccounted transfer time becomes irrelevant.

5.3. Sequential Processing with DRSAM

In this section the efficiency of DRSAM files in handling range queries and sequential processing is discussed and compared with other relevant order preserving linear hashing techniques. The analysis covers home (primary) buckets and does not account for the overflow chains which have to be scanned as well.

In Section 3.2, it was pointed out that for the sequential scan of the file, one traverses the storage areas in sequence: the E and C areas cannot coexist simultaneously. Then the storage areas which are traversed are either E then HR followed by HL, or HR then HL followed by

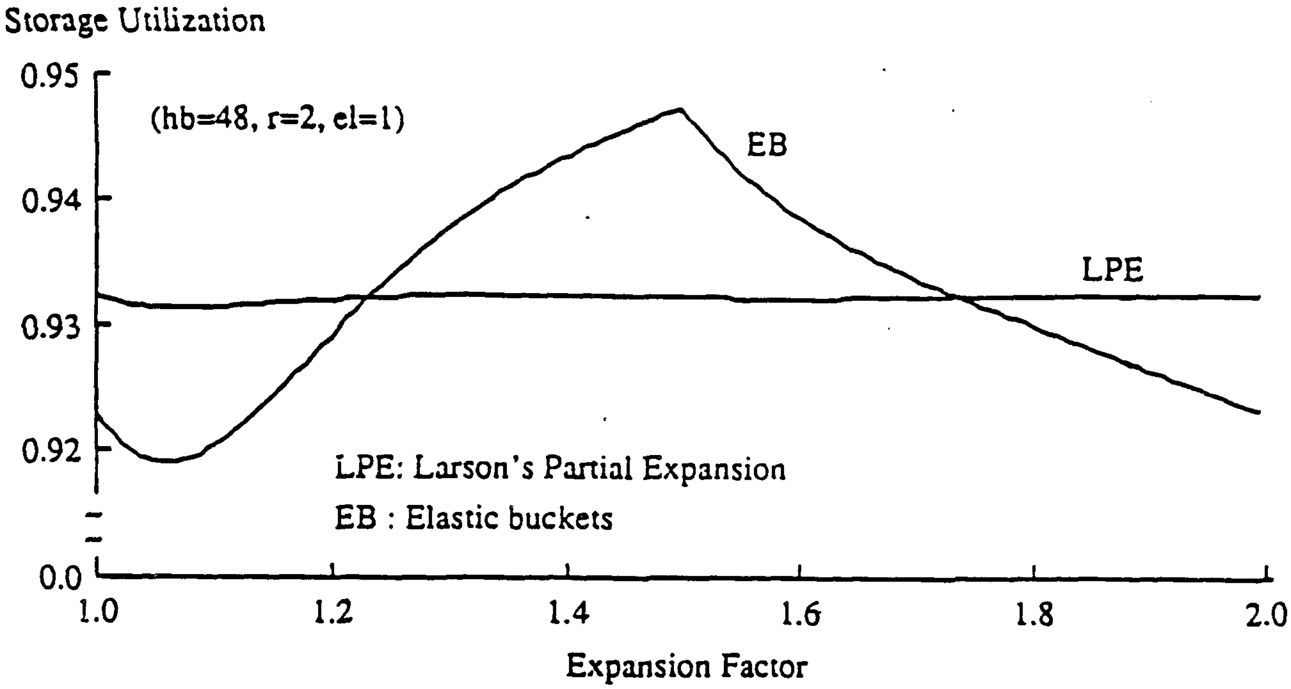


Figure 5.5 LPE versus EB: storage utilization (auf=0.93)

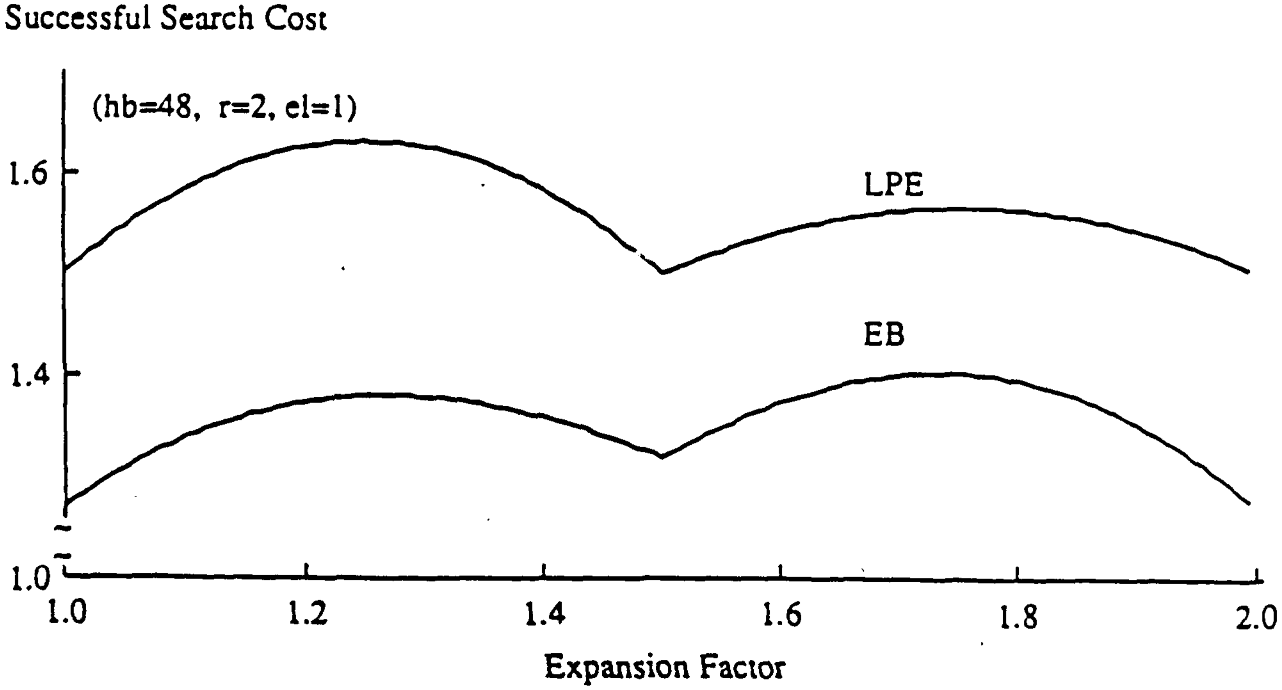


Figure 5.6 LPE versus EB: successful search (auf=0.93)

C. Thus, if a range query is specified the number of storage area boundaries to be traversed is at most three. Therefore, we can theoretically scan the primary storage area in O(1) disk seeks. In general, if enough buffer space is available, a range query which overlaps over two or more primary buckets would require one disk seek (excluding overflow access).

Considering other proposed order preserving linear hashing schemes [ORE83, BUR83]) and from Section 2, it is clear that a range query spanning two buckets will require two disk seeks to the primary storage area. Furthermore, these methods cannot make use of the availability of a large buffer space. As the number of retrieved buckets increases the improvement in range query performance increases with the use of DRSAM file structures. This improvement is based on the locality of data on consecutive home buckets and results from the sequential allocation property.

The existence of the overflow area means that the complete sequential scan of a DRSAM file would still require O(N) disk seeks. But, because of the consecutive allocation property, we showed that DRSAM outperforms previous methods. Furthermore, the use of elastic overflow bucket chains follows the same concept and implies further improvement of the performance of DRSAM files for range queries. While achieving an O(1) disk seeks is still an open problem, a promising approach is to apply the concept of recursive hashing [RAM84]. With DRSAM files, recursive hashing would lead to file structures that are near optimal for both sequential processing and random access.

If robust solutions for randomizing OPH functions become available, range queries and sequential processing performance using DRSAM are expected to be comparable to indexed sequential files and better than B-trees structures.

6. Extended DRSAM

Possible solutions for robust OPH file structures were proposed with the "quantile" methods [BUR84, KRI87], the statistic based approach [ROB86] and the indexed bounded disorder method [LIT87]. Also to be mentioned is the linear piecewise method [GAR86] based on distribution dependent hashing. For DRSAM we propose similar control mechanisms to alleviate the problems incurred by non uniform distributions and low selectivities of key values in secondary indices. These mechanisms rely on the sequential allocation property to maintain the overall performance of the file, and the resulting structure is referred to as extended DRSAM (EDRSAM). Two control strategies are discussed: 1) global file control with an index and 2) local control with a multi-level trie embedded as a sequentially allocated structure. The dynamic administration of EDRSAM with the proposed strategies is discussed in [HAC88b].

6.1. Global Control with an Index

The global control mechanism consists of a two level file structure similar to TIL1 [HAC88a]. As shown in Figure 6.1 the first level is a memory resident index table. The table entries are such that the domain of a key is partitioned into intervals with quasi-uniform distributions. The effect of the index table entries is to digitize the probability density function or equivalently create a piecewise linear approximation of the cumulative distribution. Each digital level implies the creation of a DRSAM storage area pointed to by an entry in the index table. This area grows according to the estimated cumulative distribution over the interval. Each storage area is independently managed resulting in a multi-level DRSAM structure. This approach is suitable to a distributed environment where storage is allocated in quanta of contiguous physical blocks.

The use of the index table is essential, as generally good randomizing OPH functions are not available and non-uniform distributions are detrimental for hashed files. We estimated that a table size of 4 Kbytes can accommodate around 200 index entries. A file size of 2^{24} elastic buckets can be achieved. These are more values than one expects to use for very large files. If the table cannot fit in main memory an additional disk access would be required for the index.

But this situation is not expected to arise unless highly irregular and biased distributions are considered. Paging algorithms can be used in this case. Details of the index table entries are found in [HAC88b].

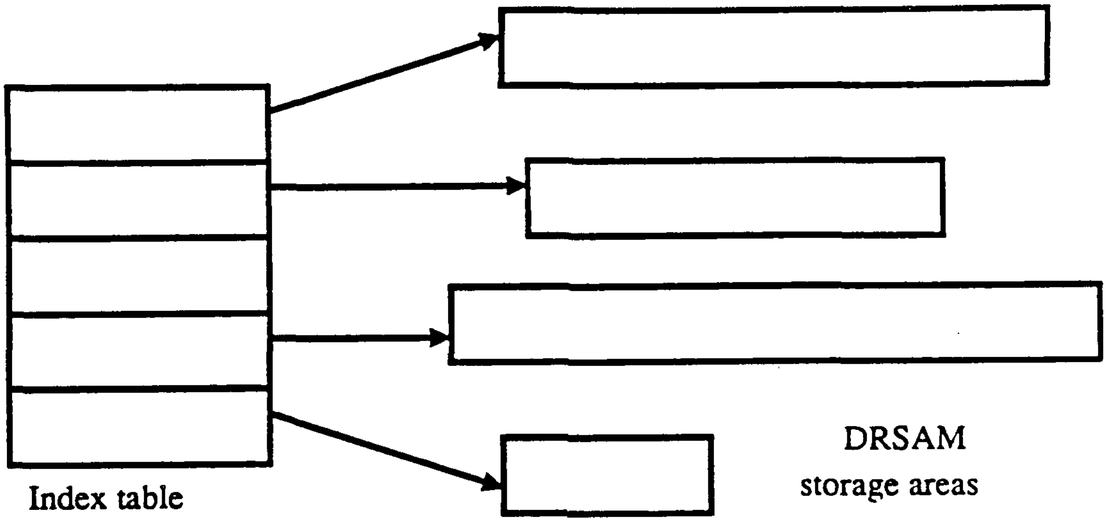


Figure 6.1 General File Model

6.2 Local Control

The DRSAM storage areas in Figure 6.1 are said to have a quasi-uniform distribution of keys. As the file dynamically evolves excessive unbalance due to non-uniform hashing will appear and degrade the performance within a storage area of the file. Proper local control for fine tuning is needed to adjust to localized unbalanced hashing. We briefly introduce two such schemes, namely *sub-hashing* and *super-hashing.*

Sub-Hashing

The idea of sub-hashing is illustrated in Figure 6.2. Assume that two brother buckets at level "i" are such that one is highly loaded and the other is sparse. The highly loaded bucket will have a lengthy overflow chain while the sparse brother will be almost empty. This situation leads to a degradation in the retrieval performance as well as in the storage utilization of

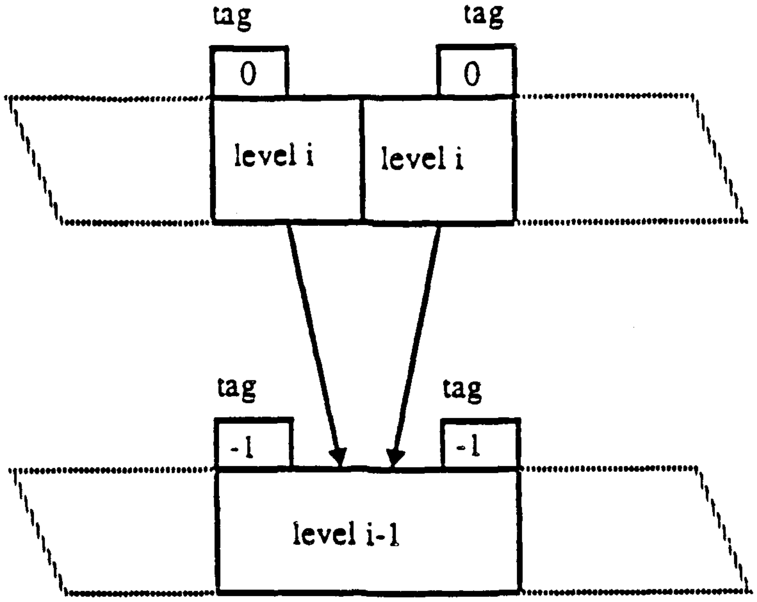


Figure 6.2 Two buckets regrouped with sub-hashing

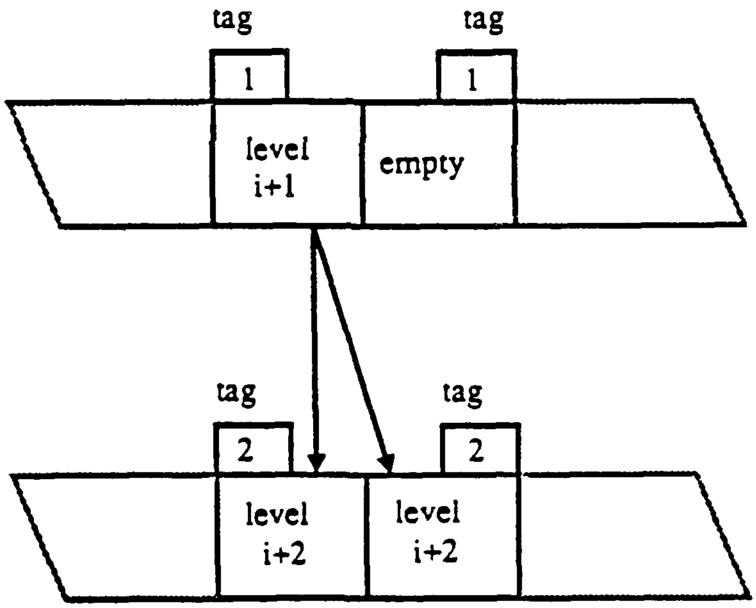


Figure 6.3 A bucket uses empty space with super-hashing

the file. We resolve this situation by merging the contents of the two buckets at a sub-level "i-1".

Tag entries are used to determine the relative level with respect to the home level of the storage area. A tag value of 0 refers to the home hash level while a 1 is used for the split hash level. In Figure 6.2 the grouped buckets are at sublevel "i-1". This concept follows the same idea introduced by Orenstein for MLOPLH [ORE83]. With *bc* as the bucket capacity, the physical contiguity is used and the resulting group can use a capacity of 2×*bc* records. Overflow is thus handled more efficiently and the problem arising from a low selectivity on some secondary keys is naturally resolved. This characteristic stems out of the sequential allocation property.

Super-Hashing

Super-hashing is illustrated in Figure 6.3 and is the counterpart of sub-hashing. When all records in a bucket at level "i" have the same (i+1)-th bit, one of the resulting child buckets is empty. we need to hash with a higher level to differentiate the keys. Tag information is also used to identify relative super-hashing depth. In Figure 6.3, the split hash level "i+1" is used first and results in the empty bucket on the right. One additional hashing depth is required to differentiate the contents of the left brother bucket. The resulting buckets are at level "i+2" and have a tag value of 2.

Sub and super-hashing can be applied recursively resulting in a embedded multi-level and sequentially allocated trie structure. With a 4 bit tag per bucket, 16 different levels of local control can be accommodated.

6.3. Some Performance Implications

For the local control mechanisms, we observed some similarities with Orenstein's MLOPLH [ORE83]. But the two mechanisms we propose, result in a file structure with its own characteristics: the use of tag information induces a maximum of one additional access. Furthermore due to bucket contiguity this access will not need disk head movement. In comparison, Orenstein's method could require a maximum of $\log_2 fl$ disk seeks.

If enough buffering is provided, the random access performance is degraded by the increased transfer time overhead. In the case where buffering is provided for one single chain, the additional overhead would be measured in disk rotations. This is compared with other order preserving linear hashing methods which require one disk seek for each home bucket access. Furthermore, larger disk buffers cannot be used by such schemes.

The control mechanisms do not assume an underlying probability distribution of the keys over the key domain. They are designed so that DRSAM files adapt to most key distributions. The dynamic administration of these methods rely on heuristic strategies which will be evaluated with real data fil.s. Details of these methods are found in [HAC88b].

7. Conclusion

In this paper, we proposed a dynamic random-sequential access method; DRSAM. Our simulation and analytical results show performance improvements for this file structure over other well known order preserving dynamic hashing techniques.

We adapted the concept of elastic buckets to directoriless dynamic hashing schemes and extended their use to overflow chains. These techniques improve the performance characteristics of DRSAM, especially for applications requiring high storage utilization. These situations are typical of very large data/knowledge bases. From our work, we recommend that ordered insertions be followed if DRSAM files are used for secondary key retrieval. Finally, we found that elastic buckets outperform Larson's partial expansion method.

As order preserving hashing leads to non-uniform distributions, we extended DRSAM with special control mechanisms. The resulting structure, EDRSAM, is designed to handle a wide range of applications without prior knowledge of the key distributions. EDRSAM can be viewed as embedding multi-level trie hashing in a sequentially allocated physical structure. In depth evaluation of EDRSAM by simulation is forthcoming.

Our long term plan is to gain more insight to DRSAM files and apply them to inverted surrogate files [BER87]. Based on the inverted surrogate files model and EDRSAM we are working on the development of a parallel back end architecture [HAC88a] for the management of very large data/knowledge bases.

References

[BAE87] Baeza-Yates R.A, Larson P.-A, *Analysis of* B$^+$ - trees with Partial Expansions, Research Report CS-87-04, Department of Computer Science, University of Waterloo, Waterloo, Canada, February 1987, pp 20.

[BAY72] Bayer R, McCreight E, *Organization and Maintenance of Large Ordered Indexes*, Acta Informatica by Springer-Verlag, Vol. 1, 1972, pp 173-189.

[BER87] Berra P.B, Chung S.M, Hachem N.I, *Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base*, IEEE Computer, March 1987, pp 25-32.

[BUR83] Burkhard W.A, *Interpolated-Based Index Maintenance*, BIT 23, 1983, pp 274-294.

[BUR84] Burkhard W.A, *Index Maintenance for Non-Uniform Record Distributions*, ACM SIGACT-SIGMOD 3rd Symposium on Principles of Database Systems, Waterloo, Ontario, April 2-4 1984, pp 173-180.

[FAG79] Fagin R, Nievergelt J, Pippenger N, and Strong H.R, *Extendible Hashing-A Fast Access Method for Dynamic Files*, ACM Transactions on Database Systems, Vol. 4, No. 3, September 1979, pp 315-344.

[GARG86] Garg A.K, Gotlieb C.S, *Order Preserving Key Transformations*, ACM Transactions on Database Systems, Vol. 11, No. 2, June 1986, pp 213-234.

[GHO86] Ghosh S.P, *Data Base Organization For Data Management*, 2nd Edition, Computer Science and Applied Mathematics Series, Academic Press, 1986.

[HAC88a] Hachem N.I, Berra P.B, *Back End Architecture based on Transformed Inverted Lists, A Surrogate File Structure for a Very Large Data/Knowledge Base*, Proc. 21st Hawaii International Conference on System Sciences, Vol. I, January 1988, pp 10-19.

[HAC88b] Hachem N.I, *Key-Ordered File Structures for the Random and Sequential Access to Very Large Data/Knowledge Bases*, Ph.D. Dissertation, Electrical and Computer Engineering Department, Syracuse University, Syracuse, NY, June 1988.

[HUT88] Hutflesz A, Six H-W, Widmayer P, *Globally Order Preserving Multidimensional Linear Hashing*, Proceedings of the International Conference on Data Engineering, 1988, pp 572-579.

[KRI87] Kriegel H-P, Seeger B, *Multidimensional Dynamic Quantile Hashing is Very Efficient for Non-Uniform Record Distributions*, Proceedings of the International

Conference on Data Engineering, 1987, pp 10-17.

[LAR78] Larson P.-A, *Dynamic Hashing*, BIT 18, No. 2, 1978, pp 184-201.

[LAR82] Larson P.-A, *Performance Analysis of Linear Hashing with Partial Expansions*, ACM Transactions on Database Systems, Vol. 7, No. 4, December 1982, pp 566-587.

[LIT80] Litwin W, *Linear Hashing: A New Tool for File and Table Addressing*, Proc. Sixth International Conference on Vary Large Databases, 1980, pp 212-223.

[LIT87] Litwin W, Lomet D.B, *A New Method for Fast Data Searches with Keys*, IEEE Software, March 1987, pp 16-24.

[LOM87] Lomet D.B, *Partial Expansions for File Organizations with an Index*, ACM Transactions on Database Systems, Vol. 12, No. 1, March 1987, pp 65-84.

[ORE83] Orenstein J.A, *A Dynamic Hash File for Random and Sequential Accessing*, Proc. Ninth International Conference on Very Large Databases, 1983, pp 132-141.

[RAM82] Ramamohanarao K, Lloyd J.W, *Dynamic Hashing Schemes*, The Computer Journal, Vol. 25, No. 4, 1982, pp 478-485.

[RAM84] Ramamohanarao K, Sacks-Davis R, *Recursive Linear Hashing*, ACM Transactions on Database Systems, Vol. 9, No. 3, September 1984, pp 369-391.

[ROB86] Robinson J.T, *Order Preserving Linear Hashing Using Dynamic Key Statistics*, 5-th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1986, pp 91-99.

Appendix 12-E

AN OPTICAL SYSTEM FOR FULL TEXT SEARCH

Pericles A. Mitkas P. Bruce Berra

Electrical & Computer Engineering Department
Syracuse University
Syracuse, NY 13210-1240
Tel. (315) 443-4445

ABSTRACT

In this paper we propose a full text search system based on optics. The storage and processing of the textual data are performed by an optical back-end system to an electronic computer. In this way we can take advantage of the speed and parallelism of digital optical processing. Using the proposed çonfiguration we show how one might implement a set of text processing operations using lasers, spatial light modulators and photodetectors.

1. INTRODUCTION

Over the past three decades advancing technology and parallelism have had a profound effect on modern electronic digital computing. This impact has come largely from the development of new microcircuit technology and the use of parallelism at all levels. While these dramatic advances are expected to continue in the foreseeable future, optics will play an ever increasing role primarily because of its inherent speed and parallelism. Over the past few years optical storage has gained considerable prominence primarily because of its high storage densities. Communicating via optical fibers is now the method of choice primarily because of its high speed and large bandwidth. Analog optical processing has been with us for years but only recently has there been increased emphasis on digital optical processing and the increased interest in optical storage and communications will promote additional interest in optical processing.

One of the major problem areas in the information storage and retrieval field concerns full text search [HAS83]. When the text database contains newspaper articles, case histories or legal briefs, one must often resort to full text search of the data. Even with the most sophisticated computing equipment this process may be time consuming and expensive [HOL83]. Because of its inherent speed and bandwidth we believe that optics may offer some future solutions to these problems.

In this paper we propose an electro-optical system for performing full text operations and show how one might perform a rich set of full text operations using this optical system.

2. BACKGROUND

2.1 Optical Text Processing

Shown in Figure 1 is a high level block diagram of an electro-optical system for performing full text search operations. The textual data are stored on optical storage devices and are removed in large quantities for transport to an optical text processor. As will be discussed in the next section, optical disks have slower access times than magnetic disks but the potential exists for massive data transfer rates through multi-track reads. Rates on the order of 300 megabytes per second appear to be feasible [BER87]; this is a full two orders of magnitude greater than current magnetic disks. Obviously, with these data rates current electronic computers would have difficulty in accepting the data since they are designed for much slower rates.

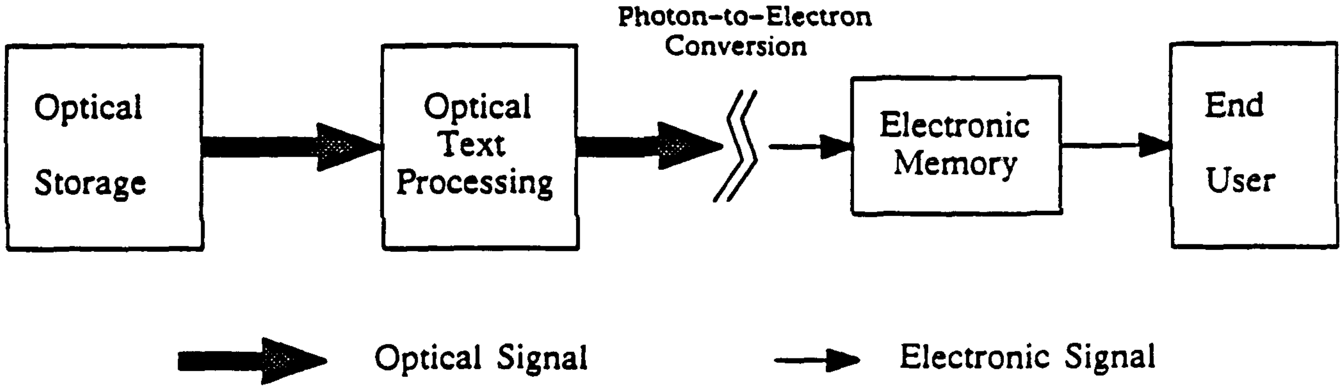


Figure 1. Optical Text Processing System.

It is logical then to consider passing the data via optical fibers to an optical text processor. The optical text processor performs a variety of text operations on the light beams based upon user queries. The resulting data that satisfy the queries are transformed from photons to electrons for further processing by the electronic computer or presentation to the user. In this way the superior speed and bandwidth of optics can be used to advantage. Additionally, the resulting output data rate will be much lower and within the capabilities of the electronic computer. Thus, the optical text processing system serves as a back-end system to an electronic computer.

2.2 Optical Disks

The need for large capacity and high bandwidth secondary storage will be satisfied by using optical disks which can hold up to 10 GBytes per platter (14-inch diameter). Currently, access times of optical disks are larger than those of magnetic disks [CAR86]. The reason is that the focusing optics are bulkier than the "flying" miniature heads of magnetic disks. Data rates are comparable, with potential for improvement since optical disk technology is relatively new.

Optical storage [ALT86, CAR86, DAV87] has its origin on the first video disk systems which were introduced in the late '70s and the overwhelming success of the compact audio disks that followed. The replacement of the earlier gas laser heads by new solid-state Laser Diodes gave birth to the first CD-ROMs [CHE86] with a capacity of hundreds of Megabytes.

Optical storage technology offers ultra high recording density in the order of hundreds of Mbits/cm2 on a medium that can be replicated at low cost and high speed. The protective plastic shield on the recording surface considerably prolongs media life, eliminates the super clean environment requirements and makes optical disks more compact and easier to use. The relatively large (order of millimeters) Head/Medium gap enables the optical head to focus into very small spots, eliminates surface wear and makes head crashes virtually impossible. In addition, it makes optical disks removable and interchangeable. Efficient encoding techniques ensure low uncorrected bit error rate which is further suppressed by error correction mechanisms. Finally, optical disks allow the integration of video and audio signals along with data in a medium directly accessible and interactable with the computer.

Optical disks can be classified into three categories: a) Read-only (ROM), b) Write-Once-Read-Many (WORM) and c) Erasable/Rewritable. The best-known representative of the first type is the CD-ROM, a compact disk with a standardized format. All the information is prerecorded by the manufacturer and cannot be altered by the user. Optical disks of the second category are not standardized and come in different sizes. As their name suggests, data can be written only once by the user and cannot be erased afterwards. They are suitable for storage of less volatile data, such as text data bases, or applications that require a complete history of updates.

Research efforts for the development of erasable optical disks have been focused primarily into three different approaches: a) Phase change of the material (amorphous to crystalline, b) Plastic deformation and c) Magneto-optic combination. The latter approach is the most advanced and appears to be the most viable one. In fact, the first magneto-optic erasable disks were introduced in late 1988 but as of yet no standards have been established. Magneto-optical disks have the potential to dominate the secondary memory market. Predictions for tenfold increase in storage capacity and transfer rates over the next few years are no longer considered optimistic. Future improvements include faster drives and smaller optical heads along with more efficient encoding and error correcting techniques. We anticipate that optical disks will steadily replace their magnetic counterparts in applications with very large storage requirements.

2.3 Increasing The I/O Bandwidth From Optical Disks

While we believe that single beam read optical disks will improve in performance, we also believe that massive data rates can be obtained from optical disks and we present two such approaches here.

a) Multiple-Beam Read

Solutions to the problem of relatively low access times and transfer rates can be provided by using more than one laser beams simultaneously. The non-interfering nature of light and the relatively large distance between the optical head and the disk surface make parallel multi-beam data access feasible. In fact, commercially available CD-ROM drives can read many tracks (i.e. ± 32) without head movement. A single laser beam can be expanded to multiple parallel coherent beams and each of them can be focused on a different track on the disk. The readout beams can be further separated and directed to an array of photodetectors. This technique (Fig. 2.a) can provide data rates in the order of hundreds of MBytes/sec but it is not suitable for parallel recordings since independent modulation of a particular beam is not possible.

b) Transmissive Optical Disks

The design shown on Figure 2.b uses a transmissive instead of a reflective optical disk [MOS87]. The unfocused laser beam illuminates a large number of tracks simultaneously, scanning millions of spots at a time and transferring that information to an array of photodetectors underneath the disk. According to the Faraday effect the polarization of a laser beam passing through a magnetized medium suffers a rotation determined by the direction of the magnetic field of the material. Therefore, magneto-optical disks can be operated in the transmissive mode if appropriate consideration is given to the elimination of diffraction effects.The potential transfer rates of this approach are enormous. Theoretically, it seems possible that the entire disk can be read in a single revolution if the beam diameter is large enough, although it is questionable whether such vast amounts of data could be handled in the output.

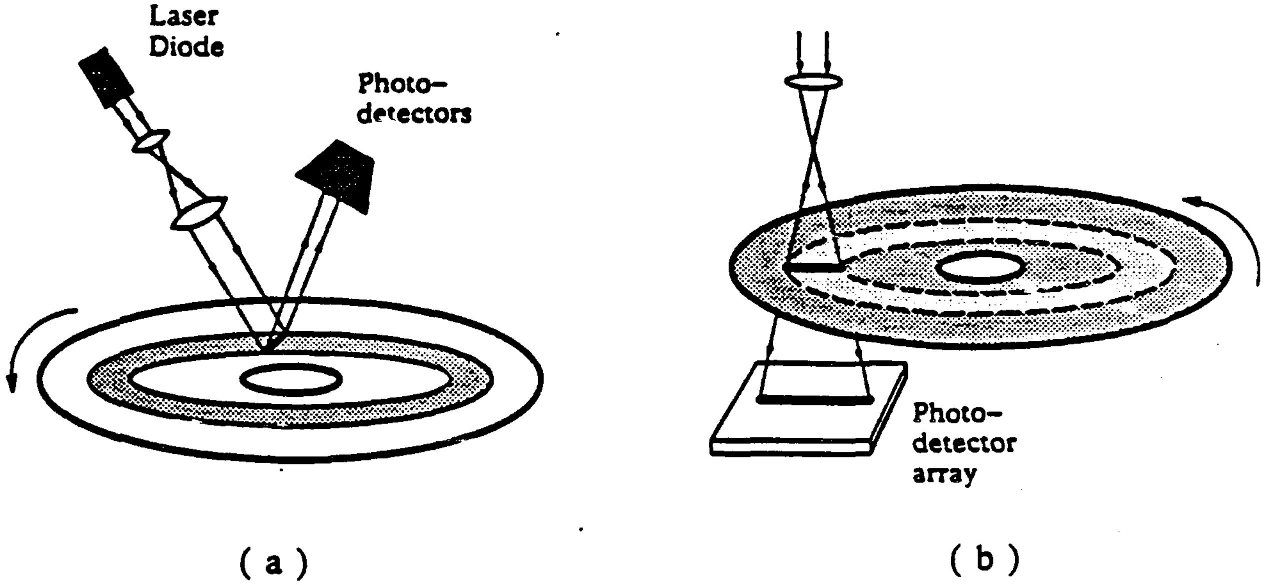


Figure 2. (a) Multiple–beam read from optical disks and (b) Transmissive optical disks.

A more practical solution would be the use of multiple heads, each one of them accessing a specific number of tracks. Five to ten such heads could cover the entire area of the disk, which means that the only moving part would be the optical disk. One lens for each head would focus the output data to a single array of detectors. The situation becomes even better if there is no need for photon–to–electron conversion. Instead, the reflected or trasmitted laser beams could be guided through optical waveguides directly to optical processors, thus eliminating any contention or saturation problems. Extending these ideas, the next step is to use memory technology that requires no moving parts.

2.4 Optical Comparisons

Following an approach taken by Guilfoyle [GUI86, GUI88] optical comparisons can be performed based on the exclusive–or (EX–OR) primitive using dual–rail logic. Two n–bit words A and B are equal if $A_i\bar{B}_i + \bar{A}_iB_i = 0$ for each pair of corresponding bits A_i and B_i. Since both the value of the bit and its complement are needed for the comson, each n-bit word will be represented by 2n light beams (i.e. the 4-bit word 1011 will become 10-01-10-10). Using this method a 00 combination corresponds to a "don't-care" character while the 11 combination always produces a "Not-Equal" result. The coding scheme for the two logical values, 1 and 0, can be either light and no light or horizontal and vertical polarization respectively.

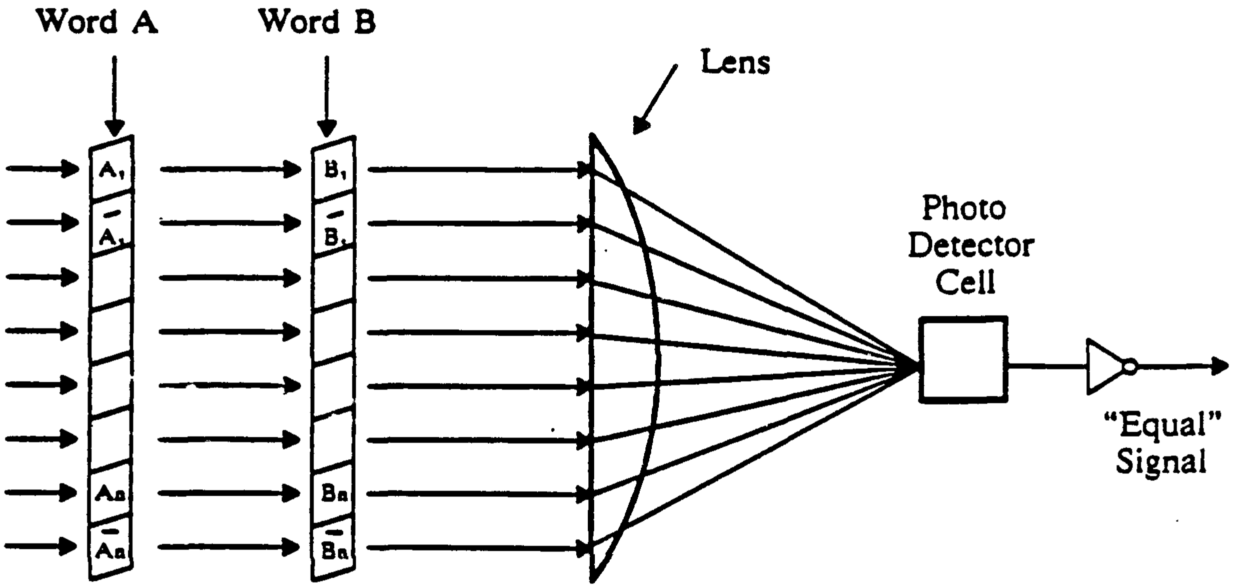


Figure 3. Optical comparison of two n-bit words based on the EX-OR primitive.

As shown in Figure 3, the light beams are superimposed bit-wise and are focused by means of a convex lens on a single photodetector which performs the logical OR (or summation) of all the beams. If no light is detected the two words are equal while any level of light intensity other than zero indicates that the two words differ in at least one bit. The output of the photodetector is electronic.

Multiple word comparisons can be performed in parallel if two 2-dimensional arrays, A and B, are employed each having 2n rows and m columns. At any instance in time the word at the i-th column of the A array is compared to the word at the i-th column of the B array. The result (equal or not equal) is recorded on the i-th cell of a row of photodetectors. This configuration, depicted on Figure 4, allows for m comparisons of n-bit words to take place simultaneously.

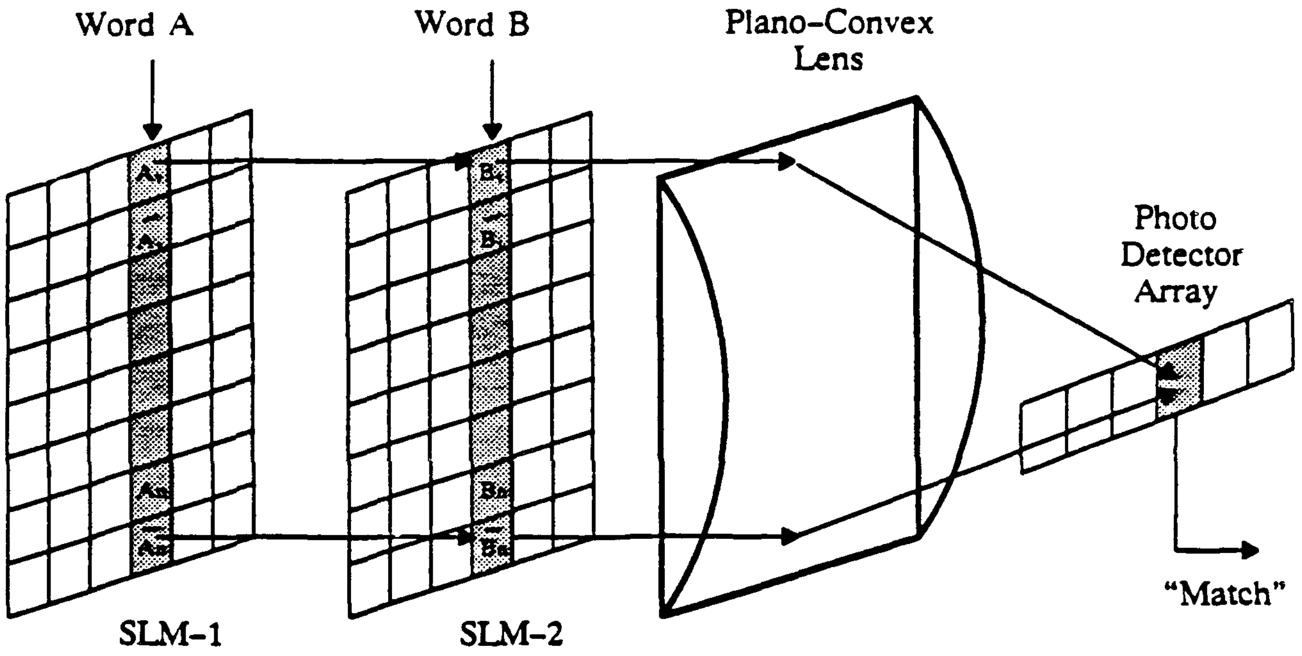


Figure 4. The two-array configuration for multiple-word optical comparisons.

The information on each array has to be recorded on two-dimensional spatial light modulators (SLMs). SLMs constitute an essential part of any optical data processing system [WAR87, PEN86, KNI81, CAS77]. These active optical devices have the ability to: a) store on a one- or two-dimensional array information encoded in an input (write) pattern, and/or b) spatially modify or amplify some of the optical characteristics (phase, amplitude, intensity, polarization) of a readout light distribution as a function of space and time. SLMs may operate in either transmissive or reflective mode. They can be electrically or optically addressed according to the nature of the write signal. Different versions can process optical signals in 1-D, 2-D or 3-D formats. A figure of merit for an SLM can be given by its *Time-Space Bandwidth*, in pixel operations per second, which offers a good approximation to the modulator's processing power. Time-Space Bandwidths of 10^{10} bit-operations/sec, which compare favorably to electronic processors have been experimentally demonstrated [LEE88].

3. AN ARCHITECTURE FOR OPTICAL TEXT PROCESSING

3.1 A Hybrid Opto-Electronic Text Processor

Figure 5 illustrates the design of an architecture for optical text processing. Documents are stored in banks of large optical disks. The goal of the design is to process data optically "on-the-fly" taking full advantage of the parallel nature and high speed of optical processing and retain in the memory only useful information.

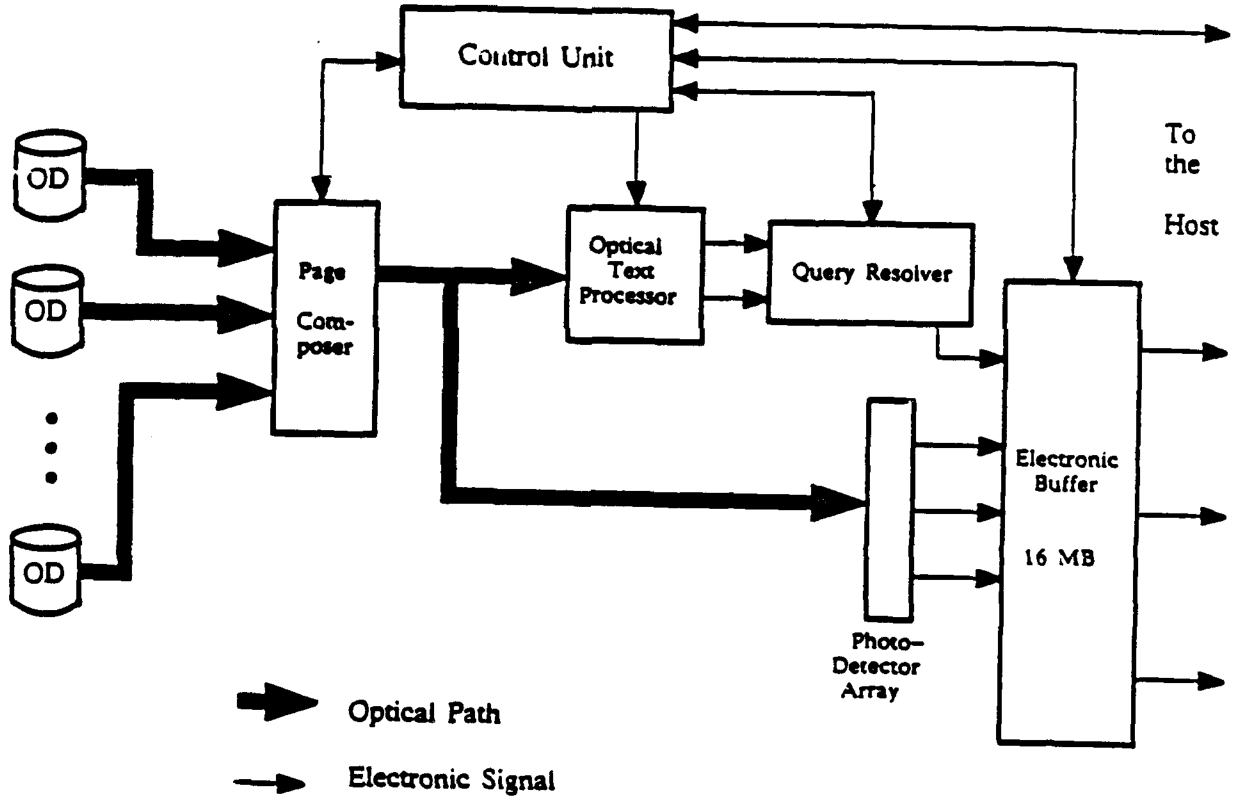


Figure 5. An architecture for text processing.

When a transaction is issued by the user, the participating data are retrieved from the optical disks and sent to the Optical Text Processor (OTP) via the Page Composer. The Query Resolver accepts electronic control signals from the OTP and the Control Unit, decides which documents satisfy the particular query and marks them for permanent storage in the Electronic Buffer. The actual contents of the documents arrive at the Buffer through an alternate optical path and are converted into electronic data by a Photodetector Array. We now describe each part of the system in more detail.

The Optical Text Processor

The Optical Text Processor (Fig. 6) is based on the two-array configuration described in the previous section. The first SLM is reflective, optically addressed and receives the light beams generated by the Page Composer. The second SLM is transmissive, electronically addressed and receives its input from the Control Unit. Optical data are written (one 16-bit character per column) on the first spatial light modulator while the appropriate search argument is loaded on the second SLM. The two optical patterns are superimposed and the result is detected on a row of photodetector cells, one for each character position.

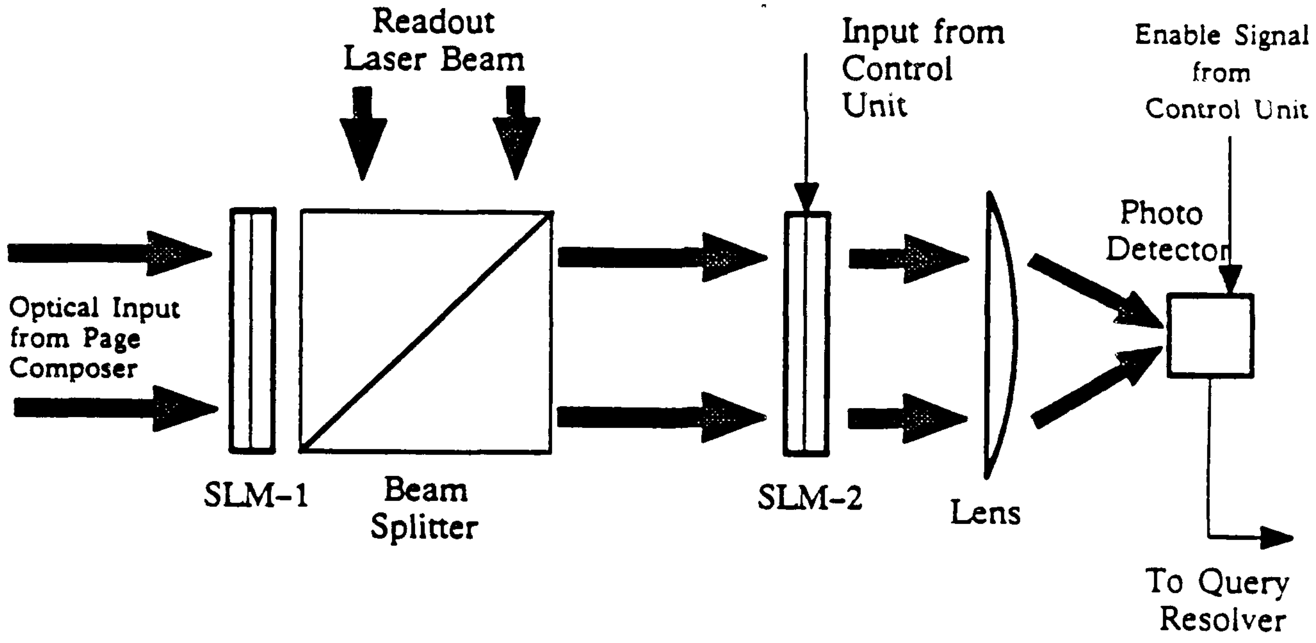


Figure 6. The Optical Text Processor

The optical pattern on the first SLM is shifted one character at a time and the string that is contained in the horizontal dimension of the array is compared to the search argument on the second array. The detector records the positions of the characters that match and sends the corresponding "hit" signals to the Query Resolver. The first few character positions on SLM-2 will permanently contain the character strings for "New-Paragraph", "New-Sentence" and "New-Word" (the "New-Word" character will be indicated as □). In addition, some positions may be reserved for the detection of special characters such as control characters etc, as shown in Figure 7.a. The remaining length will hold the search arguments(s). The total length of the array depends on the size of the search strings.

Each photodetector cell performs the logical OR of the light beams emerging from the 16-bit positions corresponding to a single character. Each cell can be individually controlled (enabled or disabled) by the Control Unit (Fig. 7.b). The decision on whether a match has occurred is taken by checking only selected groups of enabled detector cells, that correspond to the array columns on which the search arguments are stored.

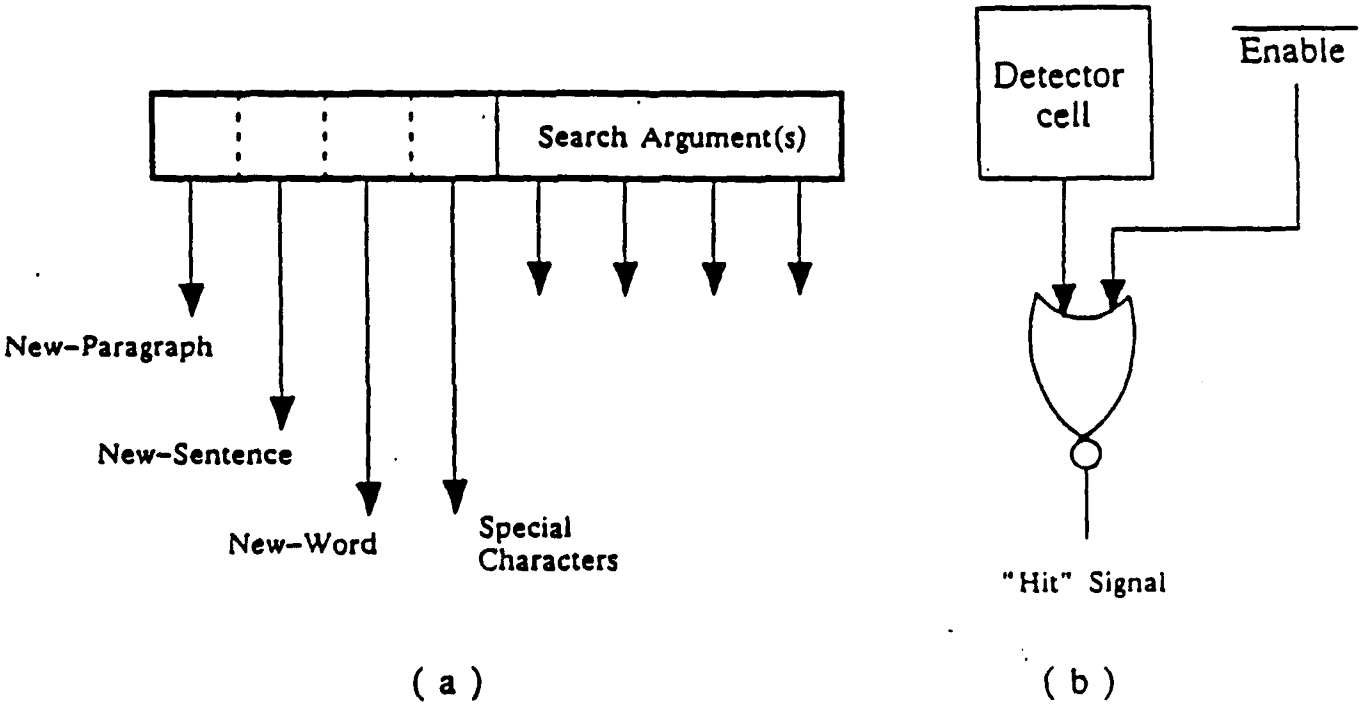


Figure 7. (a) Arrangement and (b) Individual control of the photodetector cells

The Page Composer

The Page Composer contains the necessary optical elements to generate "on-the-fly" the input optical patterns for the OTP. The main element is a fast optical scanner which will convert the sequential input from the disks to a two-dimensional pattern. The speed of this optical scanner is faster than the transfer speed from an optical disk, thus eliminating any contention problems at the Composer's input and the resulting need for buffering. The output of the Page Composer is sent to both the Optical Text Processor and, via a beam splitter, the Phctodetector Array.

The Query Resolver

The Query Resolver is an electronic sequential circuit that contains various binary counters to count the number of occurrences of particular components (paragraphs, sentences, words or even specific characters). It also contains a set of flags that can be set by the "hit" signals arriving from OTP. The necessary logic is employed to resolve ambiguous results and inform the Control Unit about successful queries. In addition, the Query Resolver maintains the number (or the name) of the document currently under process.

The Electronic Buffer

Following an alternative optical path, data from the disks arrive at another two-dimensional photodetector array where they are converted to electronic signals and stored into the Buffer. The Buffer, is a fast semiconductor memory of 16 MBytes, which operates as a ring-type queue. A document remains in the Buffer for further processing or output to the user only if it is marked by the Query Resolver as such. Otherwise, it is overwritten by subsequent data streams. Since the system operates as a back-end machine connected to a host computer, there are multiple parallel channels between the Buffer and the front-end.

The Control Unit

The Control Unit maintains the identifier of the document being processed in the OTP and issues control signals to the optical disks and the page composer. It generates the appropriate search arguments which consist of strings of characters, spaces and "don't-cares" (a "don't-care" will be indicated as $\star$) and enables the detector cells corresponding to the positions of these strings. The Control Unit also monitors the contents of the Electronic Buffer.

3.2 Performing Text Operations

The proposed architecture is capable of performing many different kinds of text search operations. The optical part constantly performs comparisons while the electronic part is responsible for writing the necessary search arguments, manipulating and interpreting the signals generated by OTP, and passing the results to the user. The following is a description of the implementation of various text operations.

a) Count the number of occurrences of word A *in a document.*

The search argument, which will be □A□, is loaded on SLM-2 of the OTP. The Query Resolver is notified about the arrival of a new document at the OTP and resets a counter to 0. Every new occurrence of A will cause an increment of the counter until the end of the document is reached.

b) Search for word A *in a document, paragraph or sentence.*

The search argument will be □A□. As soon as a match is detected the current component will be marked and kept in the buffer for further processing. The arrival of a new component will be detected by keeping track of matches on the reserved photodetector cells.

c) Search for string A *and/or string* B *in a document, paragraph or sentence.*

The search argument will be □A□★★□B□ (the number of don't-care characters in the middle is arbitrary but has to be at least 1). A match over all the detector cells corresponding to the string A and/or string B will signal a success to the Query Resolver. If the end of the component is reached before the occurrence of the second string (for the "and" case) the process will start all over again.

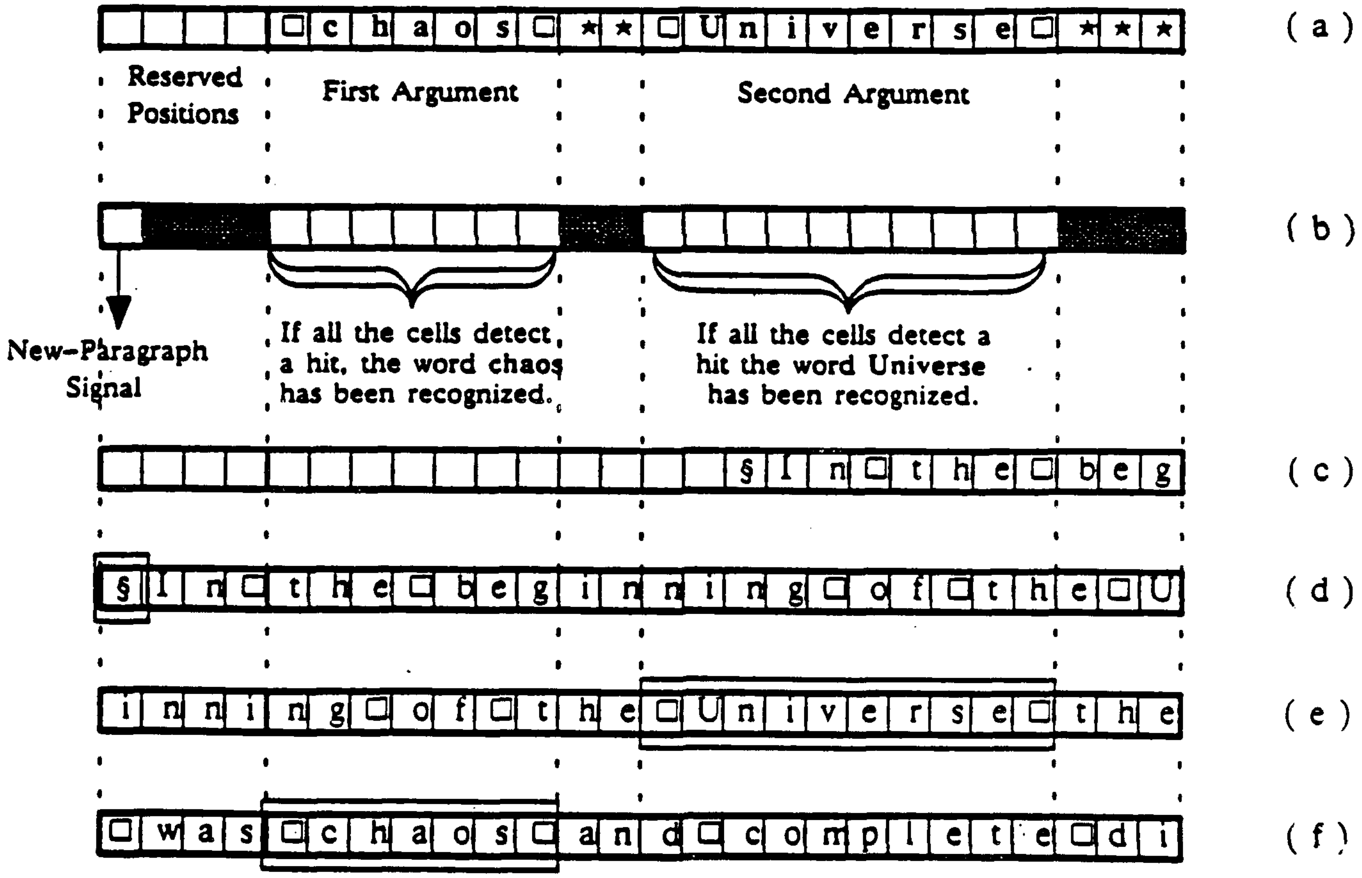
The following example will help illustrate the procedure. Suppose we want to find all the documents that contain the words "**chaos**" and "**Universe**" in the same paragraph and that in one such document a particular paragraph starts like this: "**In the beginning of the Universe there was chaos...**". As previously mentioned, the search argument: □**chaos**□★★□**Universe**□ will be loaded in SLM-2 (Fig. 8a) and the appropriate detector cells will be enabled (Fig.8b). The separation of the two arguments by "don't-cares" is necessary to avoid ambiguities in the detection of each string. The characters of the document begin to slide one-by-one into SLM-1 and are compared to the search arguments.

The query will be satisfied when:

1) a new paragraph is detected, and

2) the presence of strings "chaos" and "Universe" are detected —not necessarily in that order— before the end of the paragraph.

When a new paragraph begins, a flag F_1 is set to 1 and two others, F_2 and F_3, are reset to 0 in the Query Resolver. If one of the words is found, the second flag is set. Only when the second word is detected and the three flags register 1, has a match occurred. The paragraph is marked in the buffer and the three flags are reset. The process will be repeated with the arrival of a new paragraph. Figures 8c–8f show four different instances of SLM-1 namely : the entrance of the paragraph into SLM-1, the detection of the new paragraph, the detection of word "Universe" and finally, the detection of the word "chaos" which signals the success of the operation.


□ c h a o s □ * * □ U n i v e r s e □ * * *
(a)
Reserved Positions
First Argument
Second Argument
(b)
New-Paragraph Signal
If all the cells detect, a hit, the word chaos has been recognized.
If all the cells detect a hit the word Universe has been recognized.
§ I n □ t h e □ b e g
(c)
§ I n □ t h e □ b e g i n n i n g □ o f □ t h e □ U
(d)
i n n i n g □ o f □ t h e □ U n i v e r s e □ t h e
(e)
□ w a s □ c h a o s □ a n d □ c o m p l e t e □ d i
(f)

(a) The contents of SLM-2, (b) the grouping of the photodetector cells and (c–f) the contents of SLM-1 in four different cycles.

Figure 8. Search for the words "**chaos**" and "**Universe**" in the same paragraph.

This query is case-sensitive because it will detect only **Universe** and not **universe** or **UNIVERSE**. It is possible, however, to modify the software so that it automatically inputs all these variations as additional search arguments in SLM-2. Then, by adding some more flags in the Query Resolver and slightly changing the decision logic we can perform case-insensitive queries.

d) Search for words **A** *and* **B** *with arbitrary number of words in between.*

Once more the search argument will be □A□★★□B□. When the presence of word **A** or **B** is detected a flag in the Query Resolver is set and the process continues until the other word is detected or the end of the document is reached. Again, if a match has occurred the component is marked for further processing.

e) Search for words **A** *and* **B** *with exactly* **n** *words in between.*

The search argument will be □A□B□. As soon as the word **A** is detected the word counter is reset to 0 and starts counting using the signal from the space character position between **A** and **B**. When its content becomes equal to **n** t ıe values of the detector cells corresponding to the **B** string are examined for a possible match. If there is no match the process is repeated. With some additional logic and more word counters available, cases of multiple occurrences of **A** followed by multiple occurrences of **B** can be resolved.

f) Search for the string **X★★Y** *(fixed length embedded don't care).*

The search argument will be □X★★Y□. To avoid accepting the given pattern when it does not belong in the same word the detection of a space between **X** and **Y** (when it reaches the leading space position of the search argument) will cancel the match signal.

g) Search for the patterns **?X** *or* **X?** *(variable length prefix or suffix don't care).*

The search argument will be **X**□ or □**X**, respectively.

h) Search for the pattern **X?Y** *(variable length embedded don't care).*

The search argument will be □X□Y□. When □X is detected a flag is set at the Query resolver and the detector cells corresponding to the pattern □Y□ are enabled. If Y□ is detected and no space has occurred in between, a match has taken place. If a space is detected before the occurrence of Y□, the flag is reset and the process is repeated.

i) Count the number of sentences and/or paragraphs and/or words in a document.

A different binary counter is reset for each component in the Query Resolver and its input is linked with the corresponding detector cells. At the end of the document the contents of the counters hold the results.

4. MULTIPLE-PAGE AND MULTIPLE-DOCUMENT SEARCH

Spatial light modulators can achieve spatial resolutions in the order of tens of line-pairs per millimeter, which means that a 16-bit character can be stored in an area roughly 1 mm long and 0.05 mm wide. The effective area of most SLMs is in the order of 10-30 cm², large enough to accommodate about 10^5 characters resulting in 10^5 parallel comparisons in a single step.

It would be extremely inefficient to allow all this processing power to be wasted with only one input data string arriving at SLM-1. Therefore, the initial design can be modified in order to employ multiple stream input to the first SLM as well as multiple search arguments (not necessarily the same) on the SLM-2. Each input stream will correspond to a different page of the document or even to a different document.

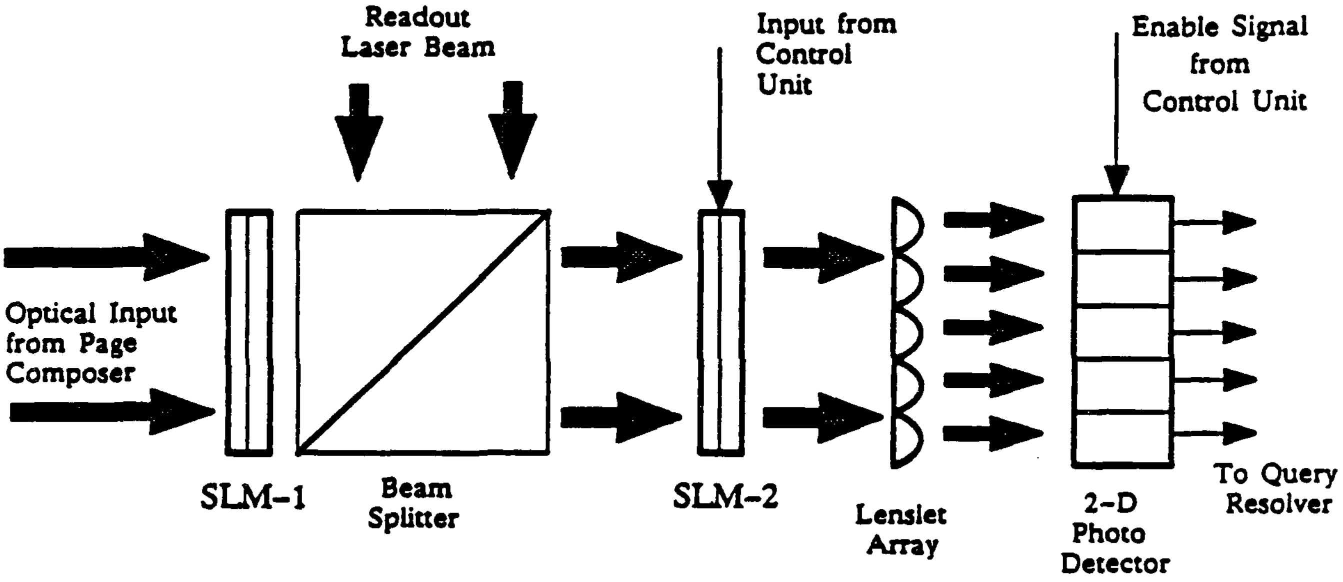


Figure 9. The Optical Text Processor for multiple-page text search.

Using the previously discussed techniques different pages can be stored on different tracks of the disk and retrieved in parallel by multiple laser beams. This complex optical pattern is imaged on SLM-1 with the help of the page composer. In this case a two-dimensional photodetector array is needed with more complicated control mechanisms. Instead of a single plano-convex lens, a lenslet array (an array of multiple miniature lenses) can be incorporated to focus the output of SLM-2 to the photodetectors. The new design is depicted in Figure 9.

Obviously, there will be a significant increase in the system's throughput not, however, without a considerable increase in the hardware complexity of the Control Unit and the Query Resolver.

5. CONCLUSION

In this paper we present an initial design of an optical full text processor. We show how one can perform a full set of text operations ranging from simple searches to various types of variable length "don't-care" searches. Because of the speed and parallelism of optics we believe that the performance of this system would be extremely good. However, our next step is to conduct detailed mathematical and simulation analyses to obtain a quantitative measure of possible performance improvement that such a system could attain.

REFERENCES

[ALT86] W.P. Altman, G.M. Claffie, M.L Levene, "Optical Storage for High Performance Applications in the Late 1980's and Beyond", RCA Engineer Magazine, 31-1, Jan./Feb. 1986.

[BEL86] T. E. Bell, "Optical Computing: A Field in Flux", IEEE Spectrum, Vol. 23, Aug. 1986, pp 34-57.

[BER87] P. B. Berra and N. Troullinos, "Optical Techniques and Data/Knowledge Base Machines", IEEE Computer, Vol. 20, Oct. 1987, pp 59-70.

[CAR86] W. Carpenter and G. Herman, "Comparative Analysis of Online Data Storage Technologies", Tech. Report MTR-86W117, MITRE Corp., Oct. 1986.

[CAS77] D. Casasent, "Spatial Light Modulators", Proc. of IEEE, Vol. 65, Jan. 1977, pp 143-157.

[CHE86] P. P. Chen, "The Compact Disk ROM: How it Works", IEEE Spectrum, Vol. 23, April 1986, pp 44-49.

[GUI86] P. S. Guilfoyle, W. Jackson Wiley, "Combinatorial Logic Based Optical Computing," Proceedings SPIE, Vol. 639-17, April 1986.

[GUI88] P. S. Guilfoyle, W. Jackson Wiley, "Combinatorial Logic Based Optical Computing Architectures", Applied Optics, Vol. 27, May 1988, pp 1661-1673.

[HAS83] R. L. Haskin and L. A. Hollaar, "Operational Characteristics of a Hardware-Based Pattern Matcher", ACM Transactions on Data Base Systems, Vol. 8, Mar. 1983, pp 15-40.

[HOL83] L. A. Hollaar, K. F. Smith, W. H. Chow, P. A. Emrath and R. L. Haskin, "Architecture and Operation of a Large Full-Text Information-Retrieval System", in *Advanced Data Base Machine Architectures* (edited by D. K. Hsiao), Prentice-Hall, Englewood Cliffs, NJ, 1983, pp 256-299.

[KNI81] G. R. Knight, "Interface Devices and Memory Materials," in Optical Information Processing-Fundamentals, Topics in Applied Physics, Vol. 48, Springer-Verlag, 1981, pp 111-180.

[LEF88] J. N. Lee and A. D. Fisher, "Development issues for MCP-Based Spatial Light Modulators", in Spatial Light Modulators and Applications, 1988 Technical Digest Series, Vol. 8, S. Lake Tahoe, NV, Jun. 1988, pp 60-63.

[MOS87] Y. S. Abu-Mostafa, D. Psaltis, "Optical Neural Computers," Scientific American, March 1987, pp 88-95.

[PEN86] W. Penn, "Liquid Crystals and Spatial Light Modulators", Tut. T24, SPIE 30-th ITS on Optical and O-E ASE, August 1986.

[WAR87] C. Warde and A. Fisher, "Spatial Light Modulators: Applications and Functional Capabilities", in *Optical Signal Processing* (edited by J. Horner), Academic Press, 1987, pp 478-524.

Appendix 12-F

AN OPTICAL ARCHITECTURE FOR PERFORMING RELATIONAL DATA BASE OPERATIONS

Pericles A. Mitkas P. Bruce Berra

Electrical & Computer Engineering Department
Syracuse University
Syracuse, NY 13210-1240
Tel. (315) 443-4445

ABSTRACT

In this paper we propose a data base machine architecture based on optics. The storage, transport and processing of the data are performed by an optical back-end system connected to an electronic computer. In this way we can take full advantage of the speed and parallelism of digital optics prior to transforming from photons to electrons. Using the proposed configuration we show how one might implement selection, projection and equi-join operations using lasers, spatial light modulators and photodetectors.

1. INTRODUCTION

During the past two decades data bases have become an inseparable part of our every day life. From a simple bank transaction to complex weather prediction, there is an enormous demand for the maintenance and effective manipulation of large volumes of data. Data bases with on the order of gigabytes are very common and their expansion to the terabyte range in the near future is a conservative prediction. In addition, fast response is often critical, especially for real-time applications.

The relational data model [COD70] has been widely used. The key to flexibility in a relational data base lies in the ease with which relations can be manipulated. The application will often need data from many different relations in order to solve a particular problem. The Data Base Management System must be able to derive the desired relations. For this. several relational operations have been developed with which the original relations can be manipulated to achieve the desired representation for the application.

The most common relational operations are: Union, Set Difference, Intersection, Cartesian Product, Selection, Projection, Quotient and Join. Three of them, namely, Selection, Projection and Join, are particularly useful for processing queries but are time-consuming, especially the Join. A fast and reliable implementation of these operations is important to the performance of a data base system. With the combined requirement of large storage and real time processing, electronic computers can be hard pressed to meet data base system needs.

Digital optics is a relatively new technology that may be able to help solve these problems by replacing electronic signals with light beams. Photons have some very attractive properties, such as high speed, massive parallelism and non-interference. The idea of optical computing is not new. Optical (especially analog) technology has been used for years in applications such as: image processing using spatial filtering of coherent light, pattern recognition using matched filters, signal processing using acoustooptic devices and matrix-vector multiplication using discrete optical processors. However, in recent years, with the maturing of laser technology, research results have indicated that digital optical processing is feasible and may offer considerable benefits in terms of speed and parallelism.

The first commercial optical products such as videodisks or compact audio disks have become impressingly successful. Optical fibers are extensively used in modern telecommunications and the use of holographic memories for multiple interconnection networks has already begun. There are numerous areas that can benefit from optical technology, Very Large Data/Knowledge Bases being one of them [BER87].

In this paper we examine some of the aspects of using optics to deal with data base problems. We begin with a section on data base and optics in which we discuss optical storage, transport and processing of data. We then present an initial design of an optical data base machine and indicate how to perform a subset of relational operations, namely selection, projection and equi-join.

2. OPTICS AND DATA BASES

2.1 Data Base Machines

A common approach to deal with the problems associated with the processing of enormous amounts of data in very large data bases is the incorporation of a Data Base Machine (DBM). A DBM with multiple storage units, multiple processors and the appropriate interconnection network will operate as a back-end machine to a host computer undertaking a large part of the total transaction load. DBMs must have very large storage capacity, high degree of parallelism to ensure acceptable data rates and specialized processing units such as sorting pipes, data filters, relational operators or inference mechanisms. Many electronic DBMs have been proposed but only a few have been implemented and even fewer have become commercially successful [NEC84, NEC86, BRI88]. While electronic processing is acceptably fast, the Input/Output process remains a bottleneck particularly when a transaction requires a global search through the entire data base.

High-capacity optical disks, high-bandwidth reliable optical interconnection networks and optical processing elements can offer an alternative solution. Here, we are concerned with the design of a back-end optical Data Base Machine capable of performing a set of relational operations, namely selection, projection and join. The goal of the system is to preprocess data "on-the-fly" in their optical form and pass to the electronic host only "useful" information. The photon-to-electron conversion takes place only at the final stage of the machine.

2.2 Optical Data Base Machines

Shown in Figure 1 is an overall block diagram of an electro-optical data base machine. The current data rates that can be sustained from current optical disks are in the order of one megabyte per second with the rates from magnetic disks being about 3 megabytes per second. However, as discussed in a subsequent section, it appears to be feasible to generate data rates in the area of 300 megabytes per second through multibeam reads and multiple heads. Assuming that these rates are feasible then the data could be input directly into an electronic computer, shown as interface A on Figure 1.

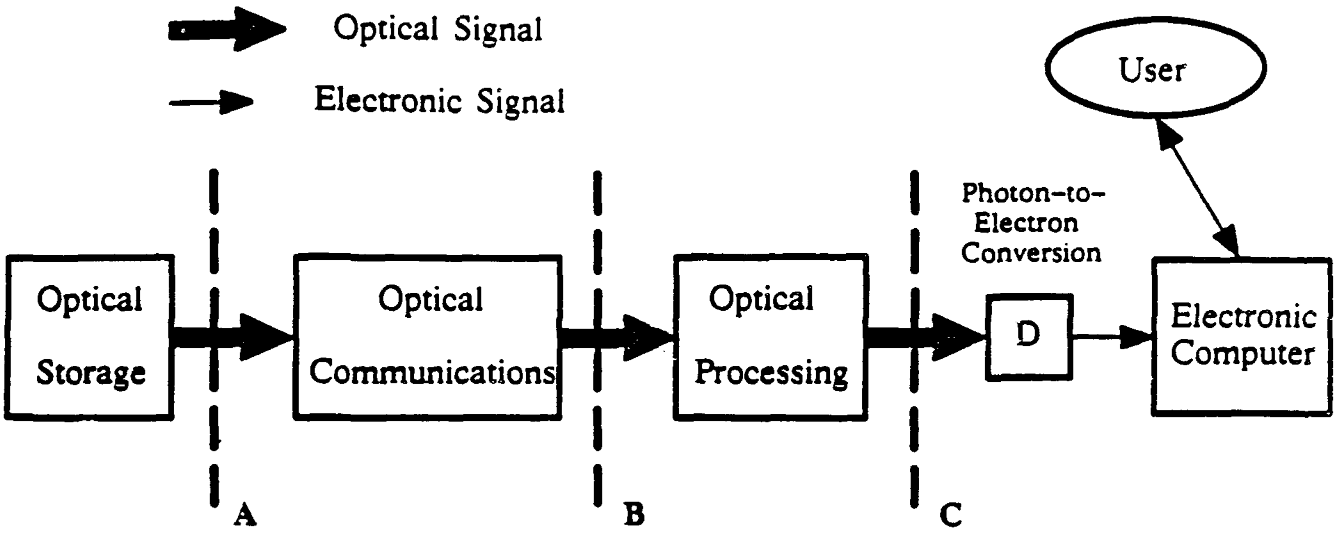


Figure 1. General block diagram of an electro-optical data base machine.

However, these data rates would overwhelm current electronic digital computers since they have been designed for magnetic disk rates. An alternative approach is to feed the data from the disks into optical fibers and distribute them to remote locations. This is shown as interface B cn Figure 1. While this approach takes advantage of the superior speed and parallelism of optical communications the receiving computers will again have difficulty with such high rates. Finally, interface C seems to offer the best approach. In this case the data in its optical form are processed by an optical processor prior to conversion and presentation to the electronic computer. The electronic computer is then better able to accept a reduced data rate; one that will be richer in content.

2.3 Optical Storage

Large capacity secondary storageis available using optical disks which can hold up to 10 GBytes per platter. Currently, access times are larger than those of magnetic disks [CAR86]. The reason is that the focusing optics are bulkier than the "flying" miniature heads of magnetic disks. Data rates are a factor of three slower but with potential for improvement since optical disk technology is relatively new.

Optical storage technology offers ultra high recording density in the order of hundreds of Mbits/cm^2 on a medium that can be replicated at low cost and high speed. The protective plastic shield on the recording surface considerably prolongs media life, eliminates the super clean environment requirements and makes optical disks more compact and easier to use. The relatively large (order of millimeters) head-medium gap enables the optical head to focus into very small spots, eliminates surface wear and makes head crashes virtually impossible. In addition, it makes optical disks removable and interchangeable. Efficient encoding techniques ensure low uncorrected bit error rate which is further suppressed by error correction mechanisms. Finally, optical disks allow the integration of video and audio signals along with data in a medium directly accessible and interactable with the computer.

The major disadvantage of the optical disks is the low Input/Output transfer rate (sustained) due to high access times. Solutions to this problem can be provided using three different techniques: a) multiple-beam read, b) transmissive disks and c) multiple-head read.

The non-interfering nature of light and the relatively large distance between the optical head and the disk surface make parallel multi-beam data access feasible. In fact, commercially available CD-ROM drives [CHE86] can read many tracks without head movement. A single laser beam can be easily expanded to multiple parallel coherent beams and each of them can be focused on a different track on the disk. The readout beams can be further separated and directed to an array of photodetectors. This technique (Fig. 2.a) can provide data rates in the order of hundreds of MBytes/sec but it is not suitable for parallel recordings since independent modulation of a particular beam is not possible.

The design shown in Figure 2.b uses a transmissive instead of a reflective optical disk [MOS87]. The unfocused laser beam illuminates a large number of tracks simultaneously, scanning millions of spots at a time and transferring that information to an array of photodetectors underneath the disk. According to the Faraday effect the polarization of a laser beam passing through a magnetized medium suffers a rotation determined by the direction of the magnetic field of the material. Therefore, magneto-optical disks can be operated in the transmissive mode if appropriate consideration is given to the elimination of diffraction effects. The potential transfer rates of this approach are enormous. Theoretically, it seems possible that the whole disk can be read in a single revolution if the beam diameter is large enough. However, it is questionable whether such vast amounts of data could be handled in the output.

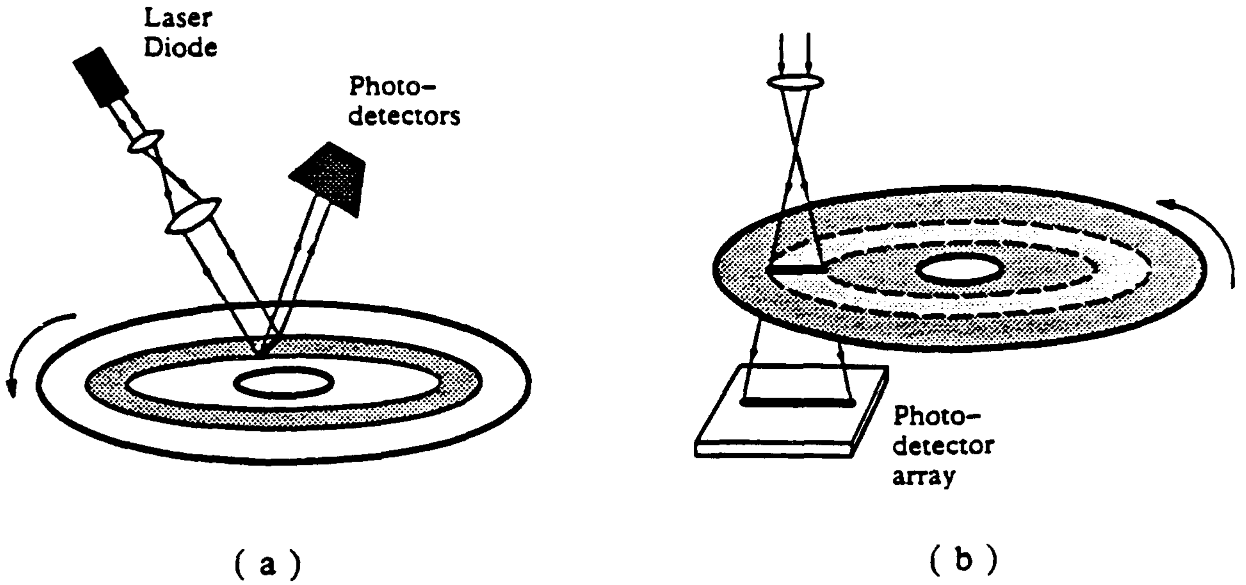


Figure 2. (a) Multiple–beam read from optical disks and (b) Transmissive optical disks.

A more practical solution would be the use of multiple heads, each one of them being able to access a specific number of tracks. Five to ten such heads could cover the entire area of the disk, which means that the only moving part in the whole device would be the optical disk rotating at a fixed speed. One lens for each head would focus the output data to a single array of detectors. The situation becomes even better if there is no need for photon–to–electron conversion. Instead, the reflected or trasmitted laser beams could be guided through optical waveguides directly to optical processors, thus eliminating any contention or saturation problems.

Beyond these approaches lie other possibilities including the complete elimination of mechanical movement. Considerable research effort has been devoted to holographic memories. Search of fixed format data, such as the indices of Very Large Data/Knowledge Bases, could make effective use of optical content–addressable memory which can be implemented by multiplexing a large number of holograms in a thick recording material like lithium niobate [GAY85, BER88].

2.4 Optical Processing

Optical comparisons can be performed based on the exclusive-or (EX-OR) primitive which calls for dual-rail logic [GUI86, GUI88]. Two n-bit words A and B are equal if $A_i\bar{B}_i + \bar{A}_iB_i = 0$ for each pair of corresponding bits A_i and B_i. Since both the value of the bit and its complement are needed for the comparison, each n-bit word will be represented by 2n light beams (i.e. the 4-bit word 1011 will become 10-01-10-10). Using this method a 00 combination corresponds to a "don't-care" character while the 11 combination always produces a "Not-Equal" result. The coding scheme for the two logical values, 1 and 0, can be either light and no light or horizontal and vertical polarization respectively.

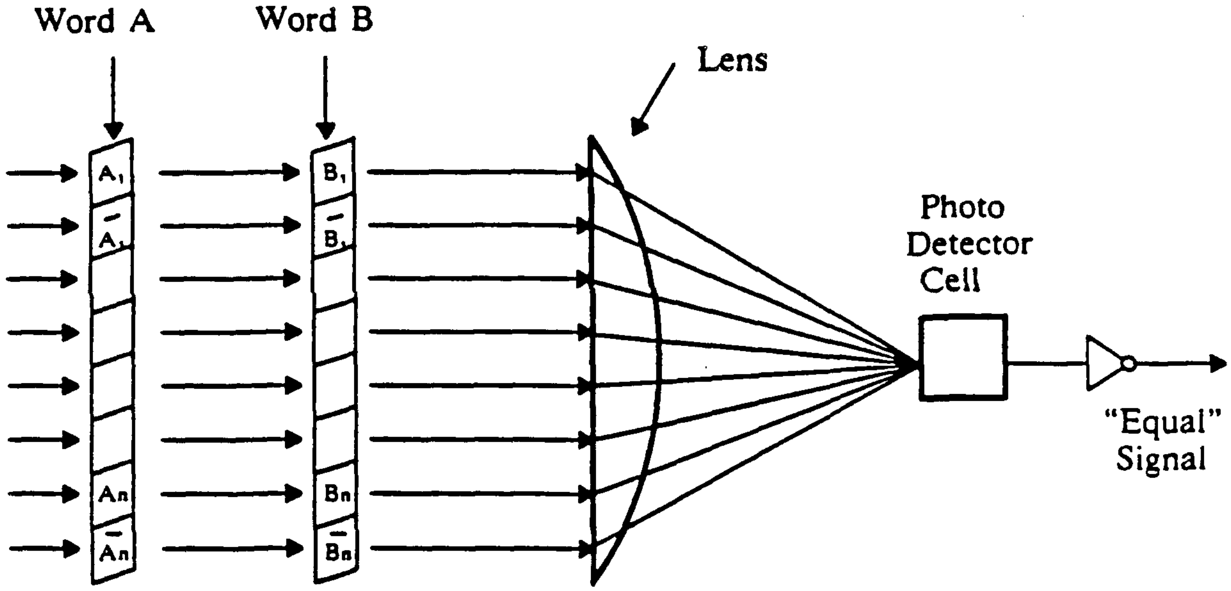


Figure 3. Optical comparison of two n-bit words based on the EX-OR primitive.

As can be seen in Figure 3, the light beams are superimposed bit-wise and are focused by means of a convex lens on a single photodetector which performs the logical OR (or summation) of all the beams. If no light is detected the two words are equal while any level of light intensity other than zero indicates that the two words differ in at least one bit. The output of the photodetector is electronic.

Multiple word comparisons can be performed in parallel if two 2-dimensional arrays, A and B, are employed each having 2n rows and m columns. At any instance in time the word at the i-th column of the A array is compared to the word at the i-th column of the B array. The result (equal or not equal) is recorded on the i-th cell of a row of photodetectors. This configuration, depicted on Figure 4, allows for m comparisons of n-bit words to take place simultaneously. The operation is concluded in as much time as it takes for a laser beam to propagate from a source through the two arrays and reach the photodetectors.

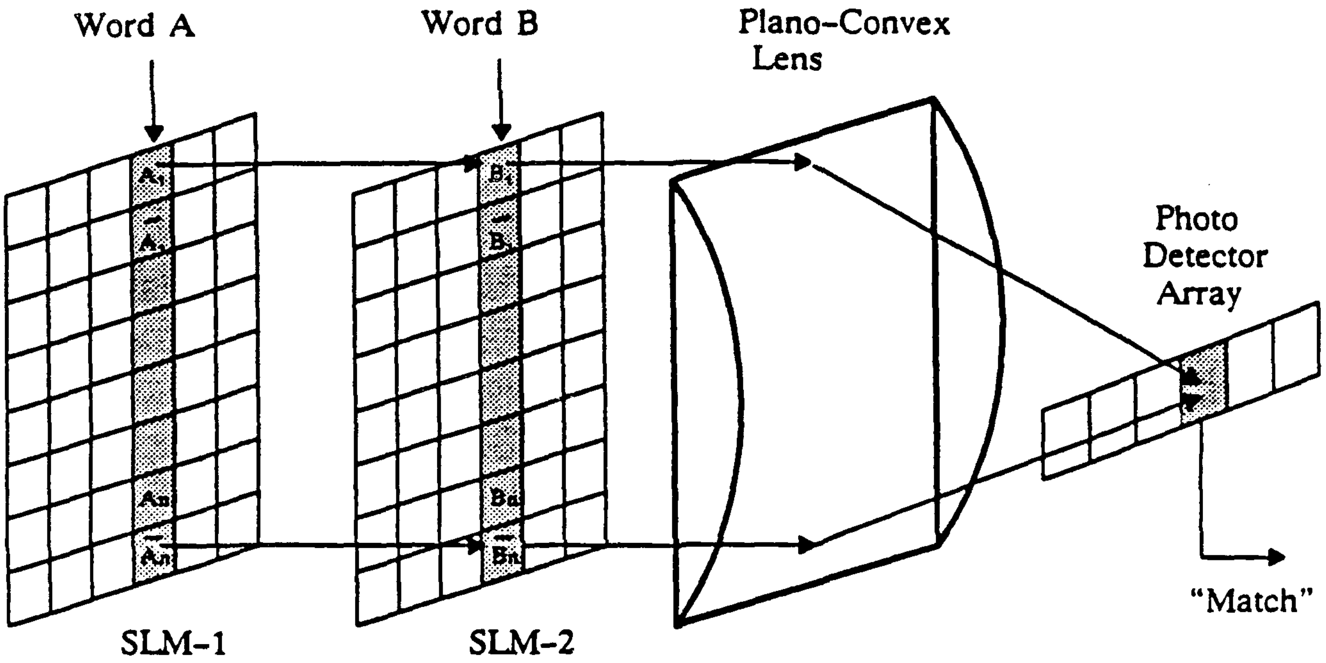


Figure 4. The two-array configuration for multiple-word optical comparisons.

The information on each array has to be recorded on two-dimensional spatial light modulators (SLMs). SLMs constitute an essential part of any optical data processing system [PEN86, KNI81, CAS77]. These active optical devices have the ability to: a) store on a one- or two-dimensional array information encoded in an input (write) electrical or optical pattern, and/or b) spatially modify or amplify some of the optical characteristics (phase, amplitude, intensity, polarization) of a readout light distribution as a function of space and time.

SLMs may operate in either transmissive or reflective mode. They can be classified as electrically or optically addressed SLM (E-SLM and O-SLM respectively) according to the nature of the control or write signal. Different versions can process optical signals in 1-D, 2-D or 3-D formats. Another classification [WAR87] divides SLMs into three classes on the basis of their functional capabilities: a)Signal-Multiplication and Amplification devices, b) Self-Modulating devices and c)Self-Emissive devices.

In a typical amplitude-modulation application the amplitude at each point in the output image is determined by the product of the input signal at a corresponding point on the device and the readout image amplitude. The input write image generates a distribution pattern of electric fields which in turn cause the light-modulating material to modify the polarization, phase and/or amplitude of the readout light.

A number of physical characteristics of an SLM are critical to its performance. *Spatial Resolution*, in line pairs per mm, has to be as high as possible because it determines the number of individually addressed pixels or bits on the array. *High Framing Speed*, defined by the time needed for a complete Write/Read/Erase cycle, is the most desirable feature of an SLM. A figure of merit for an SLM can be given by its *Time-Space Bandwidth*, in pixel operations per second, which offers a good approximation of the modulator's processing power. Time-Space bandwidths of 10^{10} operations/sec, which compare favorably to electronic processors, have been experimentally demonstrated [LEE88]. *Storage Time*, varies considerably with different designs. Applications that use SLMs as storage devices require long storage times while real-time processing needs can be satisfied with subsecond storage times. *Exposure Sensitivity* must be high to minimize power requirements and heat dissipation problems. *Contrast Ratio* can be critical in some image processing applications but is not as important for binary information processing.

The spectrum of SLM applications is impressingly broad. Acoustooptic modulators have been used in a variety of real-time operations such as convolution, matched-filter correlation, pattern recognition, spectrum analysis and radar detection. SLMs are widely used for optical analog and digital computing [RHO84, GUI88]. Numerical applications include matrix-vector [CAS82] and matrix-matrix products, matrix inversion, binary addition, multiplication [CHA86] and division. All Boolean logic functions can be implemented between multiple beams. Computations are performed using two's-complement or residue arithmetic and achieve high numeric accuracy. Optical systolic array architectures and linear algebraic processors [CAS84] have been proposed.

2.5 Relational Operations

The performance of a data base machine with a high degree of parallelism depends largely on the efficiency of the interconnection network. Therefore, the capabilities and limitations of the interconnect technology utilized in realizing a computational or signal processing unit are essential in determining the speed and flexibility of the operations that can be achieved by that unit. Optical signals need not obey the nearest neighbor interconnection law. They can flow through three-dimensional space to achieve the required interconnect pattern between elements of a two-dimensional data array before executing the desired operation between them [GOO84]. Using SLMs with space-bandwidth of 256×256 we can achieve total interconnect gate densities in the area of 10^9. Optical systems with clock speeds reaching 10^8 show potential interconnect bandwidth in the order of 10^{17} [GUI88]. To examine these advantages more closely, four categories of operations must be considered.

Only projection belongs in the first category since it is the only operation that does not require comparisons (except possibly in a second stage where any duplicates have to be removed). Each element of a one- or two-dimensional array is dropped or retained based only on its position in the array and not its value.

The second category contains single element operations like selection and text retrieval. In such computations, each element in a one- or two-dimensional array is processed independently from the rest of the array elements. The interconnectivity required by these operations is the loading and unloading of data to a processor array. Clearly, optical interconnections have the advantage of being able to input an entire data array in parallel using the third dimension for data propagation. On the other hand, in an electronic associative processor, data can be input and output only along the edges of a two-dimensional array, one row-column at a time. Optics have a lot to offer to Data/Knowledge Base (D/KB) systems where single-element operations are common.

Another category of operations is that of sorting, which is especially important in D/KB systems. Computations of this type require global interconnections between all the elements of the input array. That is, every element of the output array is dependent on all the entries in the input array. The structure of the sorting problem suggests an efficient algorithm in which computations grow as $O(N\times\log N)$. In order to achieve these computational savings, complex interconnect configurations are necessary among the input elements of the array. Additionally, these interconnections have to be changed during the different stages of the computation. The requirement for dynamic interconnections can be exploited by employing the perfect shuffle function configuration. The perfect shuffle can be applied repeatedly at each stage of the computation to produce the currently desired interconnect pattern, presumably at the expense of extra time required to complete the interconnections. Optics offer an efficient realization of the perfect shuffle function, hence its use in hardware sorting units would lead to improvements in system throughput.

The fourth category includes space and time variant operations like equi- and theta-join. The input relations can form two one-dimensional arrays and each element of the first array must be compared to all the elements of the second array. The interconnectivity pattern for these operations varies in space and time. Furthermore, the various interconnections are data dependent, making it impossible to predict in advance the appropriate interconnection patterns required at the different stages of the computation. The throughput of a parallel machine implementing this type of operations is critically affected by the availability of a dynamic and global interconnect network. Many processors could be idle for a significant number of cycles waiting for data to be properly routed to them. The overhead associated with the supervision of a controller in such a multiprocessor environment lacking space and time variant interconnection network may severely degrade all the advantages of parallel processing. Optics again offer great interconnection flexibility.

The last three categories of operations involve extensive comparing of the input values to some reference either fixed (selection, text retrieval) or constantly changing (join, sorting). As a result, the design must be capable of performing multiple parallel comparisons rapidly. Optical comparisons can be achieved efficiently using the AND-OR-INVERT (Exclusive-Or) logic primitive which was discussed in the previous section.

3. AN OPTICAL DATA BASE MACHINE

Figure 5 illustrates the design of a hybrid opto-electronic architecture capable of performing relational data base operations. The data base resides in banks of large optical disks. Binary information is read from some optical storage medium using a single or multiple laser beams and is available in its optical form for further processing.

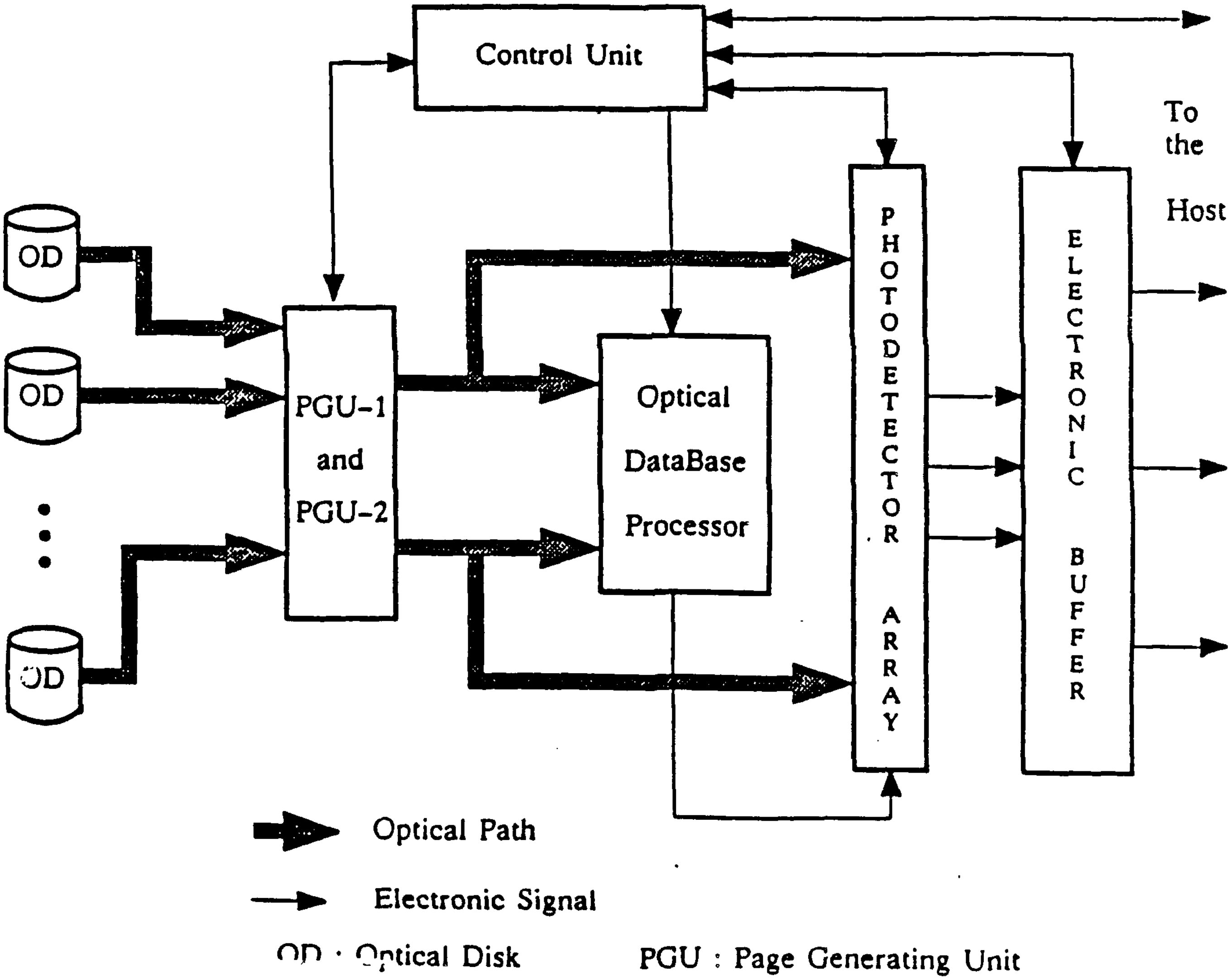


Figure 5. An architecture for relational data base operations.

The way in which records of a certain relation are retrieved is not a major issue since the distribution of light beams can be spatially modified by means of various optical elements (lenses, mirrors, beam splitters etc). However, every relation has to reside in a different storage volume thus allowing for simultaneous access to any subset of the data base relations.

When the host issues a request for a transaction to the data base the participating data are located on the optical disk units, retrieved and an appropriate beam distribution is formed in the Pattern Generating Unit. The optical data are processed, if needed, by the Optical Data Base Processor (ODBP) and the result is recorded on a photodetector array. The ODBP is described in the next section.

The Pattern Generating Unit (PGU) contains the necessary optical elements to generate "on-the-fly" the input optical patterns for the ODBP. There will be one PGU for each optically addressed SLM. The main element of a PGU is a fast optical scanner which will convert the sequential input from the disks to a 2-D pattern, as shown in Figure 6.

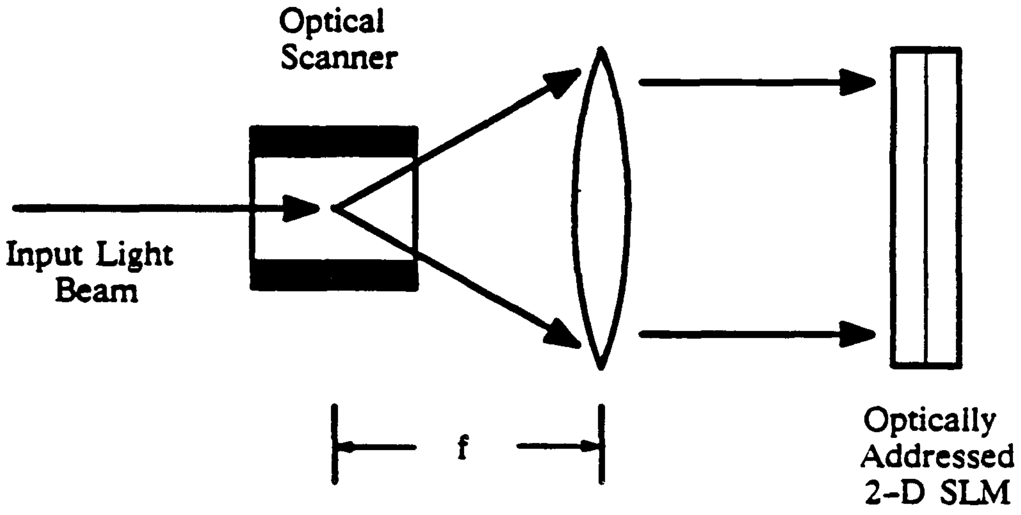


Figure 6. The use of the optical scanner for the conversion of the sequential input to a two-dimensional pattern.

Such optical scanners are capable of deflecting a laser beam at any point in an area of tens of square centimeters in about 4 nanoseconds. This speed is faster than the transfer speed from an optical disk, thus eliminating contention problems at the PGU input. The output of the PGU can be sent either directly to the SLM or, via a beam splitter, to the detector array. An active optical device is required to produce the readout optical beam necessary to read the information stored on the light modulators.

The final result of a transaction is recorded on an array of photodetector cells which transforms the information from photons to electrons. The resulting electronic pattern is stored in a fast semiconductor buffer and eventually passed to the host. The size of the Photodetector Array (PA) depends on the length of the records in the data base and must be large enough to accommodate the answer to any type of query. In any case it has to be no longer than the sum of the sizes of the two longest records in the data base. Each element of the array can be active or inactive according to control electronic signals issued either by the Control Unit or the ODBP.

The Control Unit accepts a transaction request from the host and translates it to the appropriate control signals to the disks, the PGUs, the ODBP and the detector array. It generates the necessary arguments for the Selection operation and passes them to the ODBP. The Control Unit also monitors the contents of the electronic buffer and informs the host when the result is available.

3.1 The Optical Data Base Processor

The Optical Data Base Processor (Figure 7) is based on the two-array configuration described in the previous section and performs only comparisons. There are three spatial light modulators. SLM-1 and SLM-2 operate in the reflective mode and are optically addressed by the light beams generated in the Page Generating Units. SLM-3 is transmissive, electronically addressed and receives its input from the Control Unit.

Optical data are written, one record per column, on the first two modulators while the appropriate selection arguments are loaded on SLM-3. For our purposes we need two-dimensional spatial light modulators on which binary information can be recorded and used in a fast and reliable way. A resolution in the order of 1000×1000 pixels or better will be adequate if the total time needed to write and erase a frame (framing speed) is kept in the order of 10^{-4} seconds and lower. These numbers bring the potential time-space bandwidth of the device to 10^{10} operations/sec with roughly 10^{7} records being processed every second.

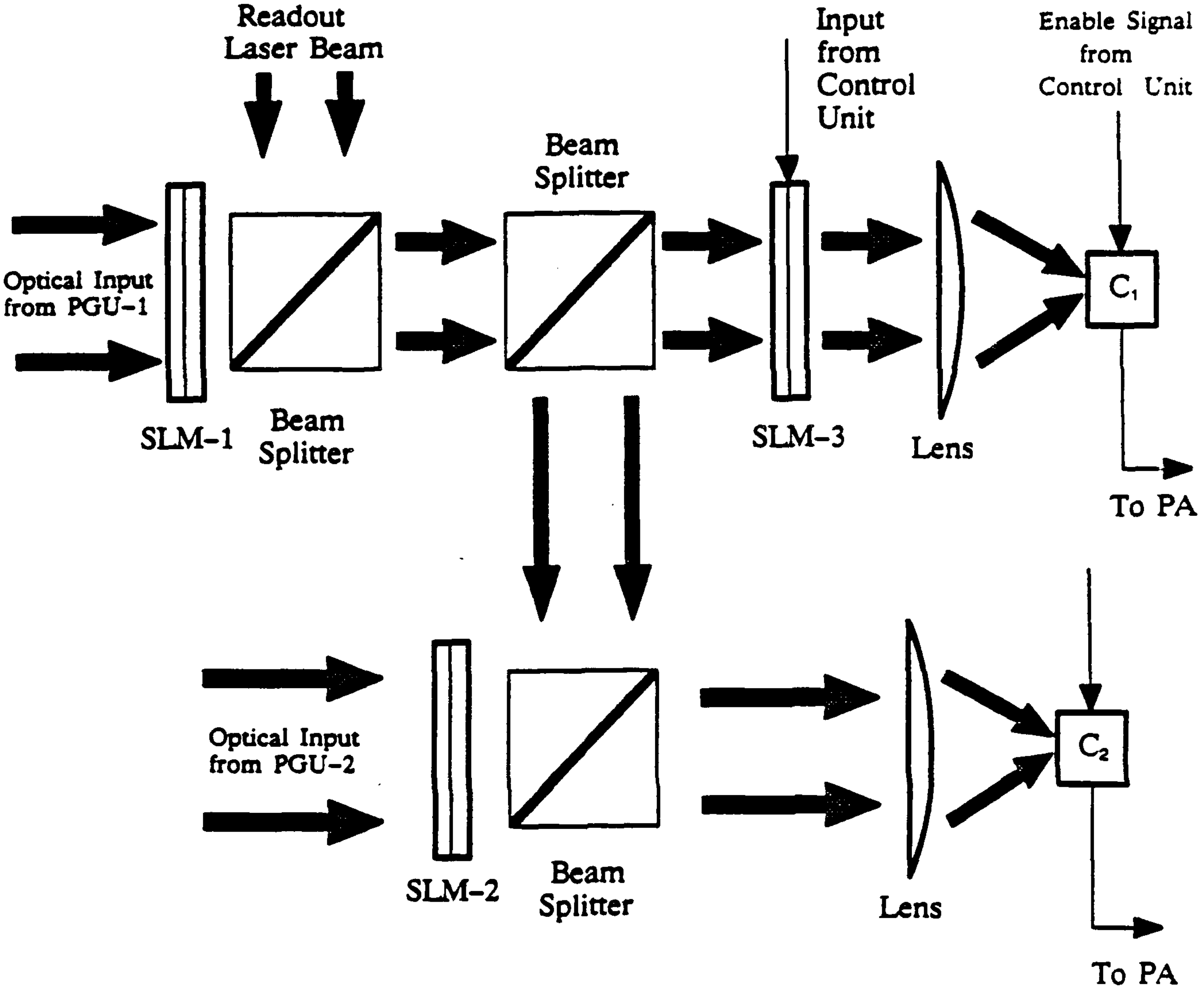


Figure 7. The Optical Data Base Processor.

There are two sets of photodetector comparator cells, C_1 and C_2, only one of which is enabled at any given instance depending on the type of operation being performed. The two optical patterns on SLM-1 and SLM-2 or SLM-1 and SLM-3 are superimposed and the result is detected on the comparators C_1 and C_2 respectively. Each photodetector cell performs the logical OR of the light beams emerging from the bit positions corresponding to a tuple. Each cell can be individually controlled (enabled or disabled) by the Control Unit (Figure 8). The decision on whether a match has occurred is taken by checking only selected groups of enabled detector cells.

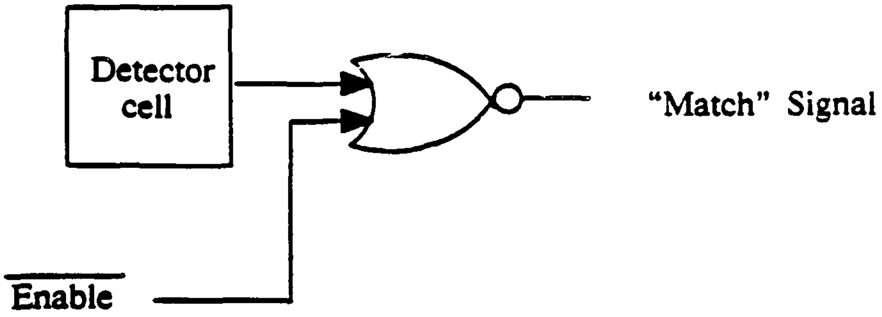


Figure 8. Individual control of the photodetector cells.

The "match" signals from the Comparators are used to select and enable the appropriate rows and/or columns of the Photodetector Array so that only these data, which satisfy the query, will be converted to electronic signals and passed to the buffer.

The realization of the relational operations projection, selection and join is discussed in the next section.

4. PERFORMING RELATIONAL OPERATIONS.

4.1 Projection

Projection can be easily performed without the use of the Optical Data Base Processor when it is the only or the last operation required by a certain transaction. All that is needed is the deactivation of the rows or columns of the photodetector array that correspond to the data fields to be masked out. As a result, only the useful part of each record is passed to the buffer. Since the result of a projection may contain multiple similar entries, the removal of the duplicates will be performed electronically.

However, projection is frequently followed by additional operations. In this case the necessary mask out will take place "on-the-fly" inside the pattern generating unit and the remaining data fields will be spatially compressed before they are written on one or two SLMs for further processing.

The maximum throughput of a projection operation is bounded only by the framing speed of the SLMs and the size of the tuples of the relation.

4.2 Selection

Selection is based mostly on comparison of a number of data fields, usually 1, to the selection argument. The records of the participating relation are written, one per column, on SLM-1. An optical pattern consisting of the constant value of the selection argument is placed at the bit positions corresponding to the data field(s) to be compared and "don't-cares" for the remaining bit positions. N comparisons take place simultaneously, where N is the number of resolvable pixels in the horizontal dimension of the SLMs. If the entry in a data field of a record is equal to the constant value a match is detected at the optical comparator C_1 and the corresponding column of the photodetector array is activated to record the qualified tuple which follows the alternate optical path shown on Figure 6. During the next cycle another N records are loaded to SLM-1 and compared. Therefore, the service time, T_{Sel}, for a selection operation on a relation R with N_R records is given by:

$$T_{Sel} = T_{su} + \frac{N_R}{N} \times T_{Fr}$$

where T_{su} is the initial set-up time and T_{Fr} is the time needed to input an entire frame to the SLM.

If the size of a record is larger than the number of rows in the SLM then two or more columns of the array can be used. Also, projection will be performed beforehand to drop the undesired data fields, if any.

4.3 Equi-Join

In the equi-join operation the fields of two records belonging to two different relations, R_1 and R_2, are concatenated only if the entries in one or more common data fields are equal. In our system (Figure 9) the records of relation R_1 are loaded on SLM-1 as described before while those of R_2 are written on SLM-2 in such a way that the common data field(s) occupy corresponding bit positions. The decision at the optical comparator C_2 is based only on the comparison of these bit positions.

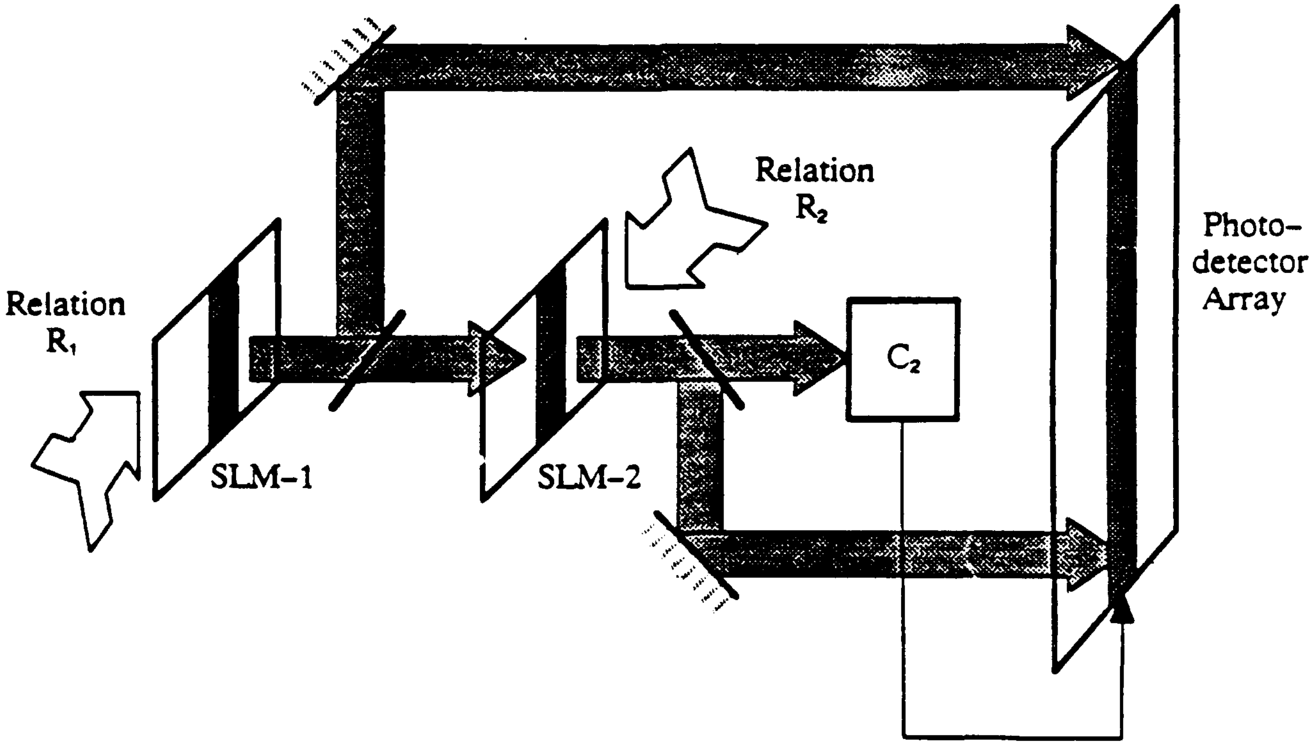


Figure 9. Performing the equi-join of two relations.

If a match is detected during the i-th cy and the respective photodetector columns are activated, the tuples from both modulators are physically concatenated on the detector array. During the (i+1)-th cycle the entries in SLM-1 are shifted (actually the entire array is erased and rewritten) one column to the left and N new comparisons take place while in the (i+2)-th cycle the entries in SLM-2 are shifted in the opposite direction for another N comparisons.

Using a slightly different approach, which may prove more efficient when the number of records of a relation is larger than N, the pattern on SLM-2 remains fixed while the tuples of R_1 slide through SLM-1. When they are exhausted a new pattern corresponding to the next N tuples of R_2 is written on SLM-2 and the process is repeated. An upper limit of the service time, T_{Join} , for this operation is given by:

$$T_{Join} = T_{su} + \left[T_{Fr} \times \left(1 + \frac{N_{R_1}}{N} \right) \times \frac{N_{R_2}}{N} \right]$$

where T_{su} is the set-up time, T_{Fr} is again the time needed to input an entire frame to the SLM and N_{K_1} and N_{R_2} are the numbers of records in relations R_1 and R_2, respectively.

4.4 Logic Operations

The proposed architecture can perform filtering of ground clauses in a logic-based knowledge base environment. Selection on a conjunction of exact-match criteria is simply accomplished by incorporating all of them in the reference pattern. Disjunction-based selection could be done by using concatenated search patterns if the total length is less than N (and matching on a subset of the detectors), or by connecting more than one optical matcher in a pipeline. Optical inference engines [CAU85, WAR86] should be more efficient than their electronic counterparts because the parallel searching operation eliminates the need for backtracking through the knowledge base.

4.5 Some Future Research Considerations

There is a number of modifications that can improve the overall efficiency of the original system. One such modification will allow the output of SLM-2 to be imaged on SLM-3 also. In this way, the throughput of a selection operation could be doubled since there will be two independent optical streams from SLM-1 and SLM-2 arriving, in an interleaved manner, at SLM-3 which holds the common selection argument(s). The optical flow is controlled by adjustable beam splitters.

Another possibility involves a feedback optical path from the SLMs to the PGUs to allow for optical processing in a loop. This approach is feasible when the binary encoding technique uses different polarization angles so that the original information is not destroyed after passing through the second SLM. The alternative design is illustrated in Figure 10.

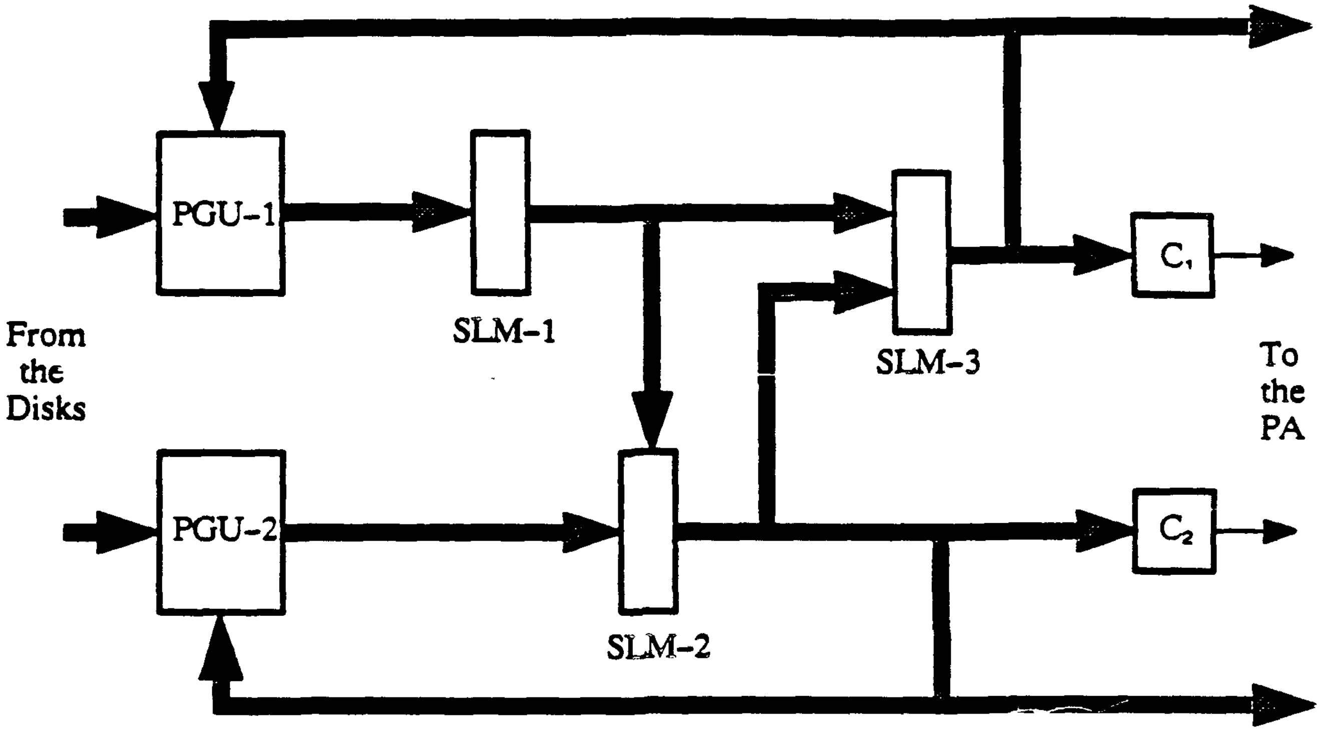


Figure 10. An alternative design of the Data Base Processor.

5. CONCLUSION

In this paper we present an initial design of an optical data base machine. We show how one can perform relational data base operations including selection, projection and equi-join. Because of the speed and parallelism of optics we believe that the performance of this system will be extremely good. However, our next step is to conduct detailed mathematical and simulation analyses to obtain a quantitative measure of possible performance improvement that such a system could attain.

REFERENCES

[BER87] P. B. Berra and N. Troullinos, "Optical Techniques and Data/Knowledge Base Machines", IEEE Computer, Vol. 20, Oct. 1987, pp 59–70.

[BER88] P. B. Berra and S. J. Marcinkowski, "Optical Content Addressable Memories for Managing an Index to a Very Large Data/Knowledge Base", Data Engineering Bulletin, Vol. 11, No 1, March 1988.

[BRI88] Britton Lee Corporation, — Latest Product Literature on the IDM 500 Database Machine, Los Gatos, CA, 1988.

[CAR86] W. Carpenter and G. Herman, "Comparative Analysis of Online Data Storage Technologies", Tech. Report MTR-86W117, MITRE Corp., Oct. 1986.

[CAS77] D. Casasent, "Spatial Light Modulators", Proc. of IEEE, Vol. 65, Jan. 1977, pp 143–157.

[CAS82] D. Casasent, "Acoustooptic Transducers in Iterative Optical Vector-Matrix Processors", Applied Optics, Vol. 21, May 1982, pp 1859–1865.

[CAS84] D. Casasent, "Acoustooptic Linear Algebra Processors: Architectures, Algorithms and Applications", Proc. of IEEE, Vol. 72, Jul. 1984, pp 831–849.

[CAU85] H. J. Caulfield, "Optical Inference Machines", Optical Communications, Vol. 55, Sep. 1985, pp 259–260.

[CHA86] V. Chadran, T. E. Krile and J. F. Walkup. "Optical Techniques for Real-Time Binary Multiplication", Applied Optics, Vol. 25, Jul. 1986, pp 2272–2276.

[CHE86] P. P. Chen, "The Compact Disk ROM: How it Works", IEEE Spectrum, Vol. 23, April 1986, pp 44–49.

[COD70] E. F. Codd, "Relational Model for Large Shared Data Banks", Comm. ACM, Vol. 13, Jun. 1970, pp 377–387.

[GAY85] T. K. Gaylord, M. M. Mirsalehi, C. C. Guest, "Optical digital truth-table look-up processing," Optical Engineering, Vol. 24, Jan./Feb. 1985, pp 48–58.

[GOO84] J. Goodman, F. Leonberger. S. Y. Kung, R. A. Athale, "Optical interconnections for VLSI systems," Proc. of IEEE, Vol. 72, July 1984, pp 850–866.

[GUI86] P. S. Guilfoyle, W. Jackson Wiley, "Combinatorial Logic Based Optical Computing," Proceedings SPIE, Vol. 639-17, April 1986.

[GUI88] P. S. Guilfoyle, W. Jackson Wiley, "Combinatorial Logic Based Optical Computing Architectures", Applied Optics, Vol. 27, May 1988, pp 1661-1673.

[KNI81] G. R. Knight, "Interface Devices and Memory Materials," in Optical Information Processing - Fundamentals, Topics in Applied Physics, Vol. 48, Springer-Verlag, 1981, pp 111-180.

[LEE88] J. N. Lee and A. D. Fisher, "Development issues for MCP-Based Spatial Light Modulators", in Spatial Light Modulators and Applications, 1988 Technical Digest Series, Vol. 8, S. Lake Tahoe, NV, Jun. 1988, pp 60-63.

[MOS87] Y. S. Abu-Mostafa, D. Psaltis, "Optical Neural Computers," Scientific American, March 1987, pp 88-95.

[NEC84] P.M. Neches, "Hardware Support for Advanced Data Management Systems," IEEE Computer, Nov. 1984, pp. 29-43.

[NEC86] P.M. Neches, "The Anatomy of a Data Base Computer — Revisited", CompCon Proceedings, IEEE, Spring 1986.

[PEN86] W. Penn, "Liquid Crystals and Spatial Light Modulators", Tut. T24, SPIE 30-th ITS on Optical and O-E ASE, August 1986.

[RHO84] W. T. Rhodes and P. S. Guilfoyle, "Acoustooptic Algebraic Processing Architectures", Proc. of IEEE, Vol. 72, Jul. 1984, pp 820-830.

[WAR86] C. Warde and J. Kottas, "Hybrid Optical Inference Machines — Architectural Considerations", Applied Optics, Vol. 25, March 1986, pp 940-947.

[WAR87] C. Warde and A. Fisher, "Spatial Light Modulators: Applications and Functional Capabilities", in Optical Signal Processing (edited by J. Horner), Academic Press, 1987, pp 478-524.

Appendix 12-G

Implementing Knowledge Base Management Systems Based on Surrogate Files[1]

P. B. Berra A. Ghafoor
S. Marcinkowski
Department of Electrical and Computer Engineering
Syracuse University

D. Shin
School of Computer and Information Science
Syracuse University

January 14, 1989

[1] This work was supported by the Air Force Systems Command, Rome Air Development Center at Griffiss Air Force Base in New York, and the Air Force Office of Scientific Research at Bolling AFB in Washington DC under Contract No. F30602-85-C-008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Abstract

We have implemented two prototypes of knowledge base management systems to demonstrate the use of surrogate files in two different environments. The first system is developed for a combined environment where an existing database system, INGRES, is used as a back-end to a Prolog interperter. Here surrogate files serve as an alternative indexing scheme to the traditional ones such as B-tree or Hashing. The second system is for an integrated environment where rules and facts are handled uniformly, which is implemented in *Lisp on the Connection Machine having 32 K processors. This second system shows that the surrogate file technique lends itself well to the parallel processing environment, yet special care must be taken for the possible I/O bottlenecks. Future work involves the use of surrogate files in handling more complex objects such as unrestricted Horn Clauses and multimedia databases.

1 Introduction

In the near future one of the major computer applications will involve Knowledge Base Management Systems (KBMS), which can be viewed as a technology extension of current database systems and expert systems. To realize a KBMS, two approaches have generally been considered. The first combines an existing database with some types of inference engine, where the inferencing mechanism is clearly separated from the management of large data. A typical example of this approach is to combine a relational database system with a Prolog Interpreter. The corresponding system architecture can be constructed by combining two components; an inference machine and a database machine. This system is not concerned with the extension of the underlying data model. That is, only existing database models supporting normalized relations are considered. The second approach integrates data manipulation and inferencing into a single system. The principal enhancement of this approach over existing expert systems lies in the capability to manage data residing on an external device; this capability includes access methods which are typical of database technology and are integrated in the logic programming framework. This system can offer a more extended data model by allowing complex objects in attribute values to be retrieved based on unification.

In terms of physical organization, knowledge base management systems differ greatly from database systems in the ways in which queries are specified and the internal data are organized. Most queries in knowledge base systems are partial match queries; any subset of attribute values can be specified in a query. That is, we cannot take advantage of the primary key in indexing. As general terms from Horn clauses are allowed as attribute values, the file organization schemes and indexing schemes which have been used in current database systems should be totally changed. In addition, recent advances in massively parallel machines raise another issues in physical knowledge base design; most file organization schemes developed so far were designed for sequential processing, and hence did not easily lend themselves to the parallel processing environment. Due to their uniform structures, surrogate files have recently been given much attention in the parallel processing environment [1,2]. This paper concerns the practical aspects of surrogate files. We implemented two prototypes of knowledge base systems each of which represents two different approaches in realizing knowledge base systems, namely the combined approach and the integrated approach.

The rest of this paper is organized as follows. In the next section, the basic concepts of

Key	BR	Codewords (12-in-2)
p	0010	0010 0000 0100
f	1011	0000 0110 0000
g	1000	0001 0000 1000
a	0100	0100 1000 0000
b	1100	0010 0100 0000
c	1010	0001 0000 0010
d	1001	0010 0000 0010

Table 1: Sample Values for Codewords

surrogate files are presented. Section 3 concerns an experimental surrogate file processing system based on the first approach; combination of existing technology (i.e. INGRES and Prolog). Described in Section 4 is a prototype of the second approach; an integrated system where rules and facts are managed together. We implemented this experimental system on the Connection Machine with 32K processors. Future work regarding both approaches are presented in the last section.

2 Construction of Surrogate Files

Surrogate files are constructed by transformed binary codes where the transform is performed by well chosen hashing functions on the original terms. Surrogate file schemes developed so far involve Superimposed CodeWord (SCW), Concatenated CodeWord (CCW), and Transformed Inverted List (TIL). In this section we describe the construction of SCW and CCW. Readers are referred to [3,4,5] for complete analysis of diverse surrogate files including TIL.

In a superimposed coding scheme, a fixed length of bit-string, called a record descriptor or a SCW, is associated with each tuple in the knowledge base. The SCW is formed by superimposing (bitwise OR-ing) each codeword corresponding to each attribute of the tuple. A codeword is said to be *m-in-n encoding* if the length of the codeword is m bits and the number of bits to be set to 1's are n. The number n is often called the *weight* of the codeword. One possible way to obtain a codeword is first hashing the individual attribute values to obtain a Binary Representation (BR) and then use the BR as a seed for a random number generator which generates n positions to be set to 1's. Consider, for example, a tuple p(a,b,c) and the codewords given in Table 1. The SCW for p(a,b,c) is formed as follows:

$$\begin{array}{lcl}
\mathrm{CW(a)} & \rightarrow & 0100\ 1000\ 0000 \\
\mathrm{CW(b)} & \rightarrow & 0010\ 0100\ 0000 \\
\mathrm{CW(c)} & \rightarrow & 0001\ 0000\ 0010 \\
\hline
 & & 0111\ 1100\ 0010 \mid \mathrm{UID}
\end{array}$$

The unique identifier (UID) is attached as shown. For a given query, a Query CodeWord (QCW) is constructed by the same method. Then the record descriptors, rather than the record themselves, are searched against the QCW. Matching condition can be determined by AND-ing the QCW and the SCW for each tuple; the AND operation should result in the same bit string as the given QCW for a matching tuple. That is, the matching condition is QCW = QCW .AND. SCW. For the matched SCW's, the corresponding UID's are collected to be used as pointers to the actual tuples. Consider, for example, a query ← p(a,X,c). The corresponding QCW can be obtained as follows:

$$\begin{array}{lcl}
\mathrm{CW(a)} & \rightarrow & 0100\ 1000\ 0000 \\
\mathrm{CW(c)} & \rightarrow & 0001\ 0000\ 0010 \\
\hline
 & & 0101\ 1000\ 0010
\end{array}$$

By AND-ing the QCW with the SCW of the p(a,b,c), since the same bit-string as the given QCW obtained, the UID is used to gain access to the actual tuples.

$$\begin{array}{lcl}
\text{QCW for } \mathtt{p(a,X,c)} & = & 0101\ 1000\ 0010 \\
\text{SCW for } \mathtt{p(a,b,c)} & = & 0111\ 1100\ 0010 \mid \mathrm{UID} \\
\hline
 & & 0101\ 1000\ 0010
\end{array}$$

Due to the superimposing of codewords, some tuples which do not match the query can have the SCW which is the superset of the QCW. For example, the QCW for ← p(d,X,Y) is 0010 0000 0010 which matches with the SCW for p(a,b,c) since

$$0010\ 0000\ 0010 = 0010\ 0000\ 0010\ .\mathrm{AND}.\ 0111\ 1100\ 0010$$

Such a match is called a false drop. Some false drops are caused by not discriminating the argument positions. For example, the QCW's for ← p(X,a,Y) would match with the SCW of p(a,b,c). Due to the false drops, actual tuples should be checked after they are retrieved. However, only a small portion of actual tuples will need to be retrieved based on the primary key, that is, UID.

The CCW of a tuple is generated by simply concatenating the binary representations (BR's) of all attribute values and attaching the unique identifier of the tuple. Consider, for example, p(a,b,c) and the BR's given in Table 1. The corresponding CCW is

0100 | 1100 | 1010 | UID

The argument positions not specified in the query (i.e. variables) should be represented by don't care match indicators. The QCW for ← `p(a,X,c)` can be obtained by replacing the second argument position with don't care match indicators.

0100 | xxxx | 1010

The matching condition is CCW = QCW provided that x (don't care match indicators) can match with both 1 and 0. A clear advantage of the CCW over SCW is that we can perform relational operations such as Join on the surrogate file itself rather than on the actual extensional database (EDB). One of the drawbacks of the surrogate file scheme (both SCW and CCW) is that every tuple descriptor should be compared to the given QCW, and thus entire descriptor file should be retrieved from secondary storage. Although the size of descriptor file is small (about 20 % of the original file), loading entire descriptor files to the main memory requires many disk accesses when the actual relation is very large. We consider some solutions to this problem in a later section.

3 The Combined Approach

As the size of knowledge based systems grow, increased demands will be placed on the management of their knowledge bases. The management of the intensional database (IDB) of rules will become a large and formidable task in itself, but the major management activity will be in the access, update and control of the extensional database (EDB). The volume of facts is expected to be in the gigabyte range. In this section we present the design of a demonstration system implementing the surrogate file concept for CCW and SCW for knowledge bases. This system is currently implemented on the VAX8800 and uses the approach of combining Logic Programming through an interface with a Data Management System as seen in Figure 1. The system demonstrates that the surrogate file concept can be used to access, update, and control the extensional database (EDB), which contains the facts that the intensional database (IDB) can use. The volume of facts is expected to be in the gigabyte range, and we can expect to have general EDB's that serve multiple inference mechanisms.

Let us set the stage for the problem that we are interested in by considering the following simple logic programming problem:

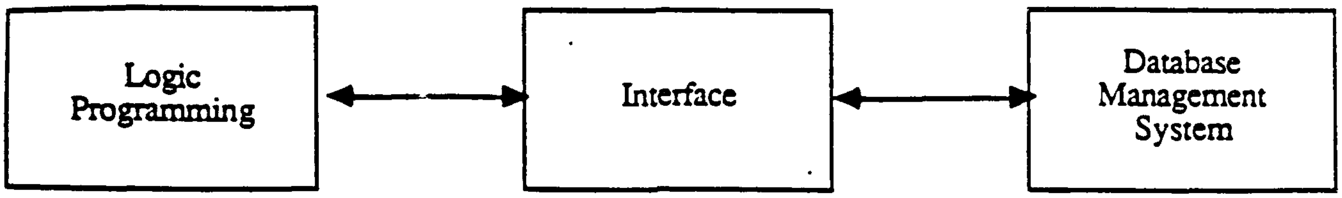


Figure 1: Combining Logic Programming with Data Management System

```
1.  grandfather(X,Y) ← father(X,Z), parent(Z,Y)
2.  parent(X,Y) ← father(X,Y)
3.  parent(X,Y) ← mother(X,Y)
4.  father(henry,cathy)
    father(don,louise)
                         ⋮
5.  mother(zofia,cathy)
    mother(lisa,tiffany)
                         ⋮
6.  ← grandmother(X,joan)
```

The first three clauses form the IDB of rules for this problem, the next two sets form the EDB of facts, and the last statement is the goal. To solve the problem (that is, to satisfy the goal), we must find the names of the grandfathers of joan. For this we search the mother and father facts on the second argument position, finding values for the first argument position that can be used later. Thus, we need to find joan's mother and father before finding her grandfathers. Consider the following general goal statement of a logic programming language:

$$\mathrm{r}(X_1, X_2, \ldots, X_n).$$

In this case, values for some subset of the X_i's will be given in the process of trying to satisfy its goal. Since the subset of the X_i's is not known in advance and can range from one to all of the values, this places considerable requirements on the relational database management system that supports the logic programming language. In fact, in order to insure minimum retrieval time for the relational database all of the X_i's must be indexed. With general indexing the index data could be as large as the actual EDB. In order to considerably reduce the amount of index data yet provide the same capability, we have considered surrogate files. Obviously if not all of the X_i's can take part in goal satisfaction then the indexing strategy will change, however here we assume the most general case in which all of the X_i's are active. Retrieving the desired rules and facts in this context is an extension of the multiple-key attribute partial match retrieval problem, because any subset of the argument positions can be specified in a query and matching between terms consisting of variables and functions as well as constants should be tested as a preunification step.

3.1 Demonstration System for SCW and CCW

In this section the demonstration system supporting the surrogate file concept for CCW and SCW is presented. The functional architecture of this system is shown in Figure 2. As is noted from this figure the overall architecture is a collection of subsystems interfaced together mainly by a relational database management system, which is INGRES. The function of each subsystem is described below.

3.1.1 Logic Programming and Interface

For the operating environment of the proposed system, it is assumed that a user submits his/her queries using a logic programming language such as Prolog [6]. Upon receiving a Prolog based query, the system passes it to an INGRES interface handler. The main function of this interface is to transform this query into a relational query with arguments and format acceptable to INGRES. This is achieved by the use of specially designed buffers for parameter passing. The interface itself is written in C and Prolog [7]. The detail discussion about its internal structure is given in section 3.3.

3.1.2 INGRES

INGRES is the conventional relational database management system and provides all the data management functions such as retrieval, manipulation, integrity control, relational operations, and so on. A query posed by a user is first transformed to a corresponding

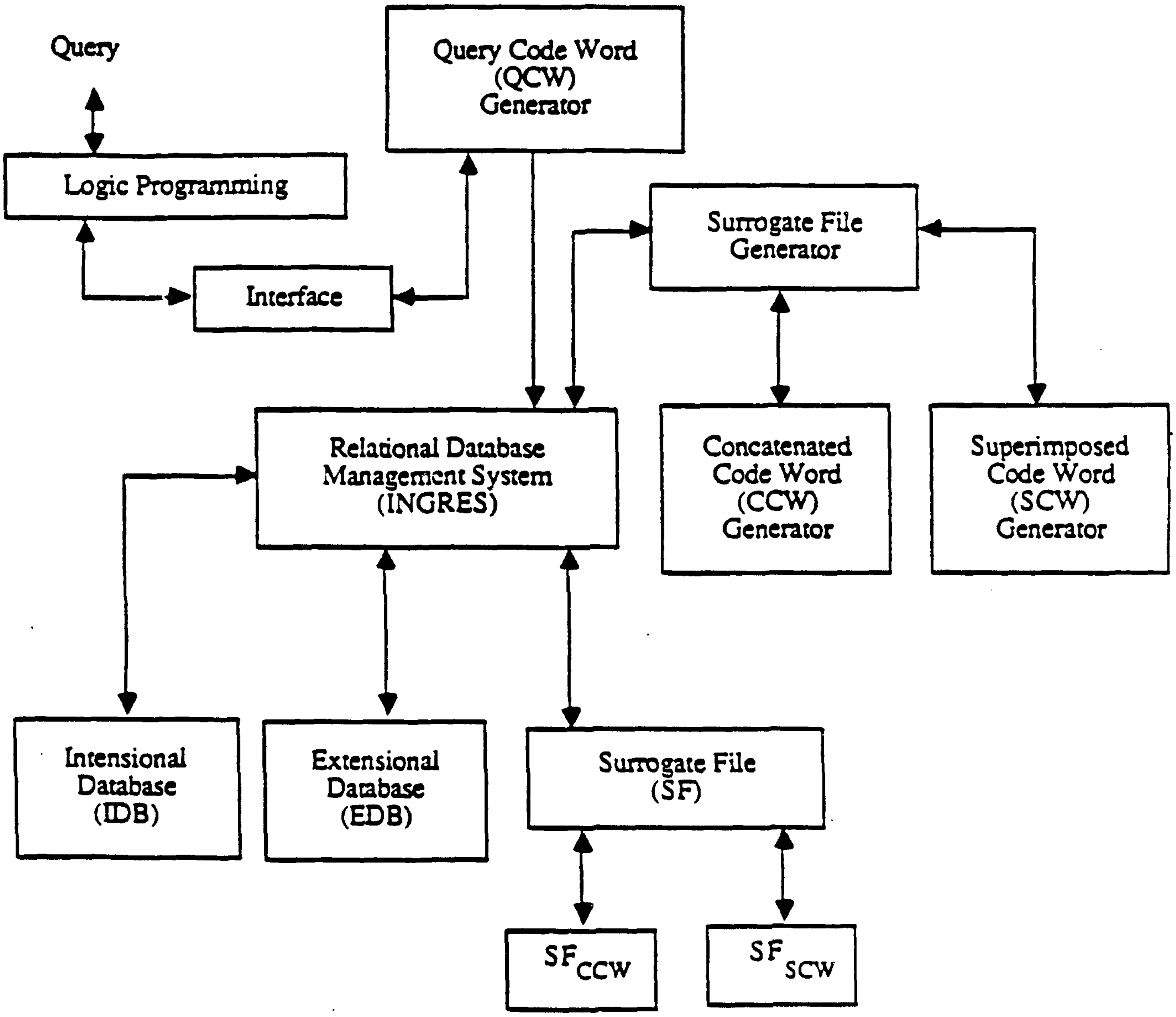

Query
Query Code Word (QCW) Generator
Logic Programming
Surrogate File Generator
Interface
Relational Database Management System (INGRES)
Concatenated Code Word (CCW) Generator
Superimposed Code Word (SCW) Generator
Intensional Database (IDB)
Extensional Database (EDB)
Surrogate File (SF)
SF_{CCW}
SF_{SCW}

Figure 2: Demonstration System for Surrogate Files

query codeword (QCW). Then the QCW is used as the search key for surrogate files which are also maintained by INGRES. Upon receiving the QCW, INGRES first accesses the surrogate file to retrieve the unique identifiers of qualified facts. Then the unique identifiers are used as primary keys of the actual knowledge bases and false matches are removed.

In this demonstration system, since surrogate files as well as the actual knowledge base are maintained by the INGRES, the latency through INGRES increases overhead, and thus the full advantage of the surrogate file cannot be realized in the demonstration system.

3.1.3 Query Codeword and Surrogate File Generators

In order to retrieve a fact, based upon one or more arguments, each argument is passed to the QCW Generator. The argument is hashed and a QCW is generated. The type of hashing employed depends on whether a CCW or SCW surrogate file is used. The surrogate file generator forms the CCW and SCW surrogate files. When a new fact is entered through INGRES, it is passed to the surrogate file generator, where it is hashed in order to generate a CCW or SCW. At the same time a unique identifier (UID) is generated. Both parameters are then passed to INGRES, which in turn stores them in the surrogate file.

3.2 Demonstration System Implementation

This section describes the detail implementation of the demonstration system. The system has been built using an ancestral database described earlier. The system employees the Columbus Prolog for logic programming and EQUEL as the database language. EQUEL is a combination of C and QUEL which is the language used to interact with INGRES. The combination of C with QUAL is used for the implementation of the interface to the database. The complete implementation of the system consists of two major tasks; the implementation of the Surrogate File, and interfacing the Surrogate File system with Logic Programming. The first task was achieved through the implementation of all the modules associated with the surrogate file, without taking into account the use of Logic Programming. Having completed the surrogate file portion of the system, the next task was to integrate Logic Programming with the Surrogate File subsystem. The detailed discussion of these tasks is given below.

3.2.1 Surrogate Files Using INGRES

The first major task was to write the code which would handle the surrogate files. First of all, the surrogate files had to made for each relation. In other words, the Surrogate File Generator had to be implemented. An interface using EQUEL was written so as to allow the user to build up the extensional database. The interface was designed to be user friendly through the use of menus. The sequence that a user needs to follow is described below. At the start of the session the user is presented with the following options:

```
enter the number corresponding to the task you wish to perform.
   1. Append tuples to a relation
   2. Find the children of a parent
   3. Find the parent of a child
   9. To Exit
enter choice:
```

To add tuples the user would enter the number 1 for the choice. Having done this the user is then prompted for the relation name, followed by another prompt to supply the relevant information. The following session illustrates the process of adding a tuple to the maternal relation in the ancestral database. The objective is to add the information that zofia is the mother of cathy.

```
     enter the relation name (maternal, paternal):  maternal
     maternal relation chosen
     enter mother's name:  zofia
     enter child's name:  cathy
  zofia is the mother of cathy
```

The data entered is first processed by the Surrogate File Generator, where it is hashed according to the mechanism employed for CCW (or SCW) surrogate files. The information is then stored in the appropriate surrogate file, CCW/SCW using INGRES. Also, INGRES stores the original input in the EDB. Figure 3 shows how the information is stored in the surrogate file and in the EDB.

3.2.2 Testing Surrogate File

Having developed the facility to generate the surrogate file, the next step was to test the system by submitting queries to the system. The queries at first did not come from a

maternal relation		
uid	mother	child
1	zofia	winston
2	zofia	peter
3	mary	bruce
4	vera	mary
5	eva	teresa
6	teresa	james
7	cindy	yvonne
8	zofia	cathy
9	eva	lorie
10	eva	paul
11	carmela	mary
17	carrie	john
18	joyce	peter
19	laura	vera

msurro relation		
mh	mc	uid
8023910	7825774	1
8023910	7366004	2
7168370	6451829	3
7759218	7168370	4
6649441	7628146	5
7628146	6971757	6
6515054	7960175	7
8023910	6513012	8
6649441	7106418	9
6649441	7364918	10
6513010	7168370	11
6513010	6975336	17
6975353	7366004	18
7102837	7759218	19

Figure 3: Storage of information in Surrogate File (CCW) and in the EDB

logic programming environment so as to test only the surrogate file part of the system. Later logic programming was interfaced to allow queries to come from a logic programming environment. Testing the surrogate file part of the demonstration system was performed by choosing the appropriate menu item. By choosing item 2 we can for instance find the children of zofia; such a query can be written in Prolog as follows, `mother(zofia,X)`. By choosing item 3 from the menu we can query the database to find for instance the mother of cathy, the equivalent query in Prolog would be `maternal(Y,cathy)`. INGRES does not accept a query directly based on Prolog. A query based on Prolog must first be processed so as to format it according to the requirements of INGRES. As mentioned earlier , this is done by an interface, which takes the query and processes it make it understandable to INGRES. This will be discussed in more detail in the next section. The following are examples of how the surrogate file concept was tested.

```
enter choice : 2   (Finding the children of a parent)
to find the children of a father enter --- father
to find the children of a mother enter --- mother
enter your choice:  mother
```

```
   looking for a child's mother
enter the mother's name to find her children:  zofia
looking for the children of zofia
   winston
   peter
   cathy
-----------------------------------------------------
enter choice : 3  (Finding the parent of a child)
to find the father of a child enter --- father
to find the mother of a child enter --- mother
enter your choice:  mother
  looking for the mother of a child
enter the child's name to find its mother:  cathy
looking for the mother of cathy
   zofia
-------------------------------------------------------
```

```
 range of c is maternal
 range of d is msurro
      retrieve(parents=c.mother)
          where c.uid = d.uid and d.mc = hashed query
------------------------------------------------------
```

3.3 Interfacing Logic Programming

In order to incorporate the Logic Programming Environment, an interface linking Prolog and INGRES together [7] has been employeed. The interface converts the logic query into QUEL language based relational query, which is then submitted to INGRES. For example, the logic query `mother(zofia,X)`, is translated by the interface into the following QUEL based format:

```
queryDB([relation-name(field1, field2)]).
```

The query `mother(zofia,X)` is actually processed by the following way.

```
mother(Name,Child) ← hash(Name,Newname),
                     queryDB([msurro(A,Newname,Uid)]),
                     queryDB([maternal(Uid,Name,Child)]).
```

The first subgoal hashes the query, or in other words the Query Codeword is generated. The second subgoal accesses the appropriate surrogate file through INGRES to find the Unique Identifier (UID). Then, the UID is passed to the EDB, where the appropriate facts are retrieved. Furthermore a check for false drops needs to be made because the hashing function may not be perfect. The complete process from the time a logic query is submitted until the final answer is produced, is illustrated as follows. For this purpose the sample run of `mother(zofia,X)` is presented.

```
> prolog db

     Starting up DBMachine
     resetting the machine
     COLUMBUS Prolog Unix Version 2.9 (DB version)
?- initDB(logic)     /* specifies the database to be used */

     yes.
?- consult(demopro).  /* consults the logic programming rules */
     Consulting 'demopro'...
     'demopro' consulted

     yes.
?- mother(zofia,X).  /* query to find children of zofia */
   X = 'winston';
   X = 'peter';
   X = 'cathy';
   no.
```

In the above example, the 'prolog db' initiates the Columbus Prolog interpreter together with the INGRES interface. The command 'initDB(logic)' tells INGRES that the database name to be used is 'logic'. Then, the rest of procedures are the same as those of conventional Prolog interpreters.

4 The Integrated Approach

Presented in this section is a succinct description of the experimental implementation of the surrogate file processing method for relational algebra on the Connection Machine using 32

K processors with conventional disk systems. This system is run under Ultrix supporting both Connection Machine models CM-1 and CM-2, and is implemented in *Lisp. We also developed a small Prolog interpreter that directly accesses surrogate files as well as transforms queries into conjunction of goals. This interpreter, however, is restricted in the sense that neither recursive queries nor complex objects are supported. Some modifications of this system should support these functionalities, which require further study. Some empirical results are presented in order to identify the possible sources of speedups and bottlenecks in processing very large data/knowledge bases. Complete source listing is given in the Appendix.

4.1 Parallel Algorithms for Relational Algebra

Described in this section are parallel algorithms for two representative relational algebra operations based on surrogate files, selection and equi-join, in a massively parallel processing environment. The experimental system is based on the following assumptions:

- The entire surrogate file for a relation can be resident with in the local memory of the Connection Machine. We consider several ways to improve the surrogate file loading time in subsequent sections.

- Each processing element (PE) stores a CCW from a relation. That is, with 32 K processors, we can handle a relation with up to 32 K tuples. Each PE may accommodate k CCW's from k different relations. The value k depends on the number of attributes for the relations since we assume the use of a fixed number of bits for all attributes.

The parallel selection based on surrogate files can be divided into two main parts. First, surrogate files are processed against the query codeword (QCW) of the given query, and the UID's corresponding to the matched tuples are collected. Then the actual EDB are accessed using the UID's as primary keys.

The query that requires joins is of the form $\leftarrow Q_1, Q_2, \cdots, Q_n$. For join operations, n selection operations are independently performed for the n conjunctions $Q_1, Q_2, \cdots, Q_n$. That is, the shared variables (join variables) are not considered in selection operations. All PE's have flag bits corresponding to $Q_i's$ called Mark(i) ($i = 1, \ldots, n$). The selection operation for Q_i sets the flag bit (Mark(i)) to 1 in each PE's local memory when the PE contains a CCW such that CCW = QCW. Then the number of matches N_i, are calculated for each relation by adding the corresponding flags for all the PE's, and then sorted in ascending order. The Q_i with the lowest N_i is used to form a new set of queries for other Q_i's. Consider, for instance, a query of the form R(a,X), P(X,Y,c). Suppose that the number of tuples satisfying the subquery R(a,X) is 3 with results {X | c, d, f } and the cardinality of the relation P is greater than 3. The number ot matched tuples can be easily calculated using parallel sum functions after setting a flag to 1 for all processors where matchings occurred. Three new queries are formed for the relation P after selections are completed. They are P(c,Y,Z), P(d,Y,Z) and P(f,Y,Z). For each newly formed queries, selections are performed and the corresponding flag is logically OR-ed with the flags resulted from other newly formed queries and logically AND-ed with the previous flag resulted from the initial selection. Since we do not explicitly produce a new relation, the operation can be viewed as an implicit join based on a parallel nested-loop algorithm. Figure 4 presents the join algorithm described in this section in a more formal manner. In the next section we consider some empirical results obtained from this experiment.

4.2 Some Empirical Results

In order to measure the performance against large databases, a procedure `make-edb` is developed. This procedure can make an arbitrary length of test EDBs by generating random numbers. For example, `make-edb('R 2 8000)` creates an EDB R with 2 attributes and 8000 tuples. This function also creates the surrogate file for the relation. The simple Prolog interpreter developed as a front-end translates user queries into conjunctions whose literals are all EDB predicates. The variables appearing in the query can be viewed as don't care match indicators, which are represented by symbols starting with ?. This notation is based on [8,9]. The variables given in the users' query are considered as level 0 variables. The level is increased thereafter for all the unbound variables whenever unification is performed with non EDB predicates. Consider, for example, the following simple IDB:

```
((grandfather ?x ?y)(father ?x ?z)(parent ?z ?y))
((parent ?x ?y)(father ?x ?y))
((parent ?x ?y)(mother ?x ?y))
```

where father and mother are EDB predicates. The query `(grandfather ?x a)` is first translated to

```
(grandfather (?x.0) a).
```

```
/* Let Q be a query of the form $\leftarrow Q_1, Q_2, \cdots, Q_n$.
$Q_i$ is of the form $Q_i(X_1, \cdots, X_{k_i})$.
n : the number of subqueries in the query.
$N_i$ : the number of selected tuples for $Q_i$ */
begin
    for i = 1 to n do
    begin
        /* Do selection in parallel for $Q_i(X_1, \cdots, X_{k_i})$. */
        Mark(i) := $(CCW_j = QCW)$ for each processor j $(j = 0, ..32K - 1)$;
        $N_i$ := Sum(Mark(i));
    end {for}
    $Q'_i$ := Sort $Q_i$ according to $N_i$;
    for i = 2 to n do
    begin
        Form a new query set using $Q'_1, \cdots, Q'_{i-1}$;
        for k = 1 to $N_i$ do
        begin
            New-Mark(i) := New-Mark(i) .OR. $(CCW_j = QCW)$;
            Mark(k) := New-Mark(i) .AND. Mark(k);
        end {for}
    end {for}
    Access EDB based on UID (the PE number);
    Checking False Matches;
end {program}
```

Figure 4: The Parallel Join Algorithm for the CM

Then by the unification with the first clause

```
(father (?x.0) (?z.1))(parent (?z.1) a).
```

Then, this is transformed to two independent conjunctions;

```
(father (?x.0) (?z.1)) (father (?z.1) a)
(father (?x.0) (?z.1)) (mother (?z.1) a)
```

The shared variables act like join variables and searching is performed based on the constant. We used 16 bit binary representations (BR's) for each attribute. The selection time, S_{cm} is estimated as 0.01 second for the BR, which does not depend on the number of tuples selected, or number of tuples resident in local memory of the processing elements. That is, most of the time is spent in forming new queries and the parallel selection time performed in the Connection Machine is negligible. As can be seen in the previous section, in the average case, our join algorithm has the parallel complexity of $O(k \times n^2)$ where n is the minimum selectivity among k sub goals. For an equi-join between two relations, the join time is measured approximately n x S_{cm} where n is the minimum selectivity for the initial selection and S_{cm} is the selection time including all the overheads in the front-end and the Connection Machine. Once surrogate files are loaded to the local memory of the Connection Machine, relational algebra can be performed very efficiently. Surrogate file processing is very good choice in a massively parallel processing environment (SIMD type) due to its uniform data structure. However, we observed some problems in this method mainly due to the low transfer rate from secondary storage.

Since the Connection Machine used for this experiment does not have any specialized disk systems, it has to load data from its front-end having conventional disk systems. Figure 5 shows the actual surrogate file loading time including the overhead required to distribute the loaded data to parallel variables. This figure shows that with conventional disk systems surrogate file processing incurs high communication overhead.

Another time consuming task in our implementation is the EDB access time using UID's as primary keys. We used a block with the capacity of 50 tuples for actual EDBs to randomly access the qualified tuples. A UID is assumed to give the block number as well as relative position in the block for the corresponding tuple. In Figure 6, we plotted the actual EDB access time in the front-end. We cannot measure exact time for a large number of matched tuples, since, when that is the case, the system spends a lot of time in

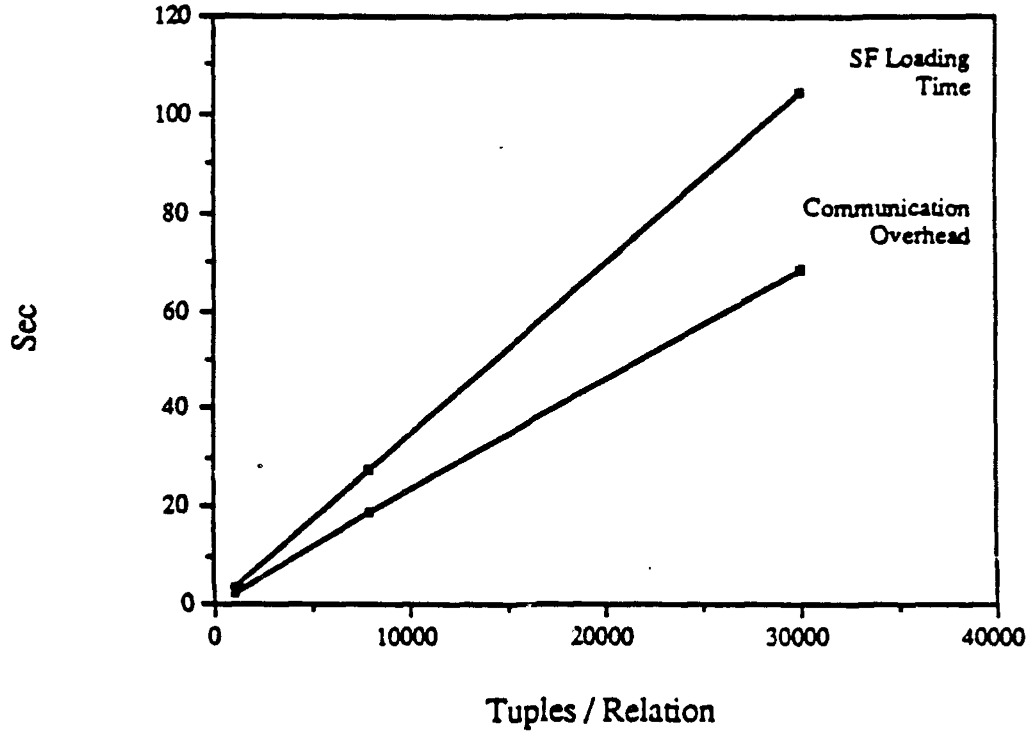


Figure 5: Surrogate File Loading Time

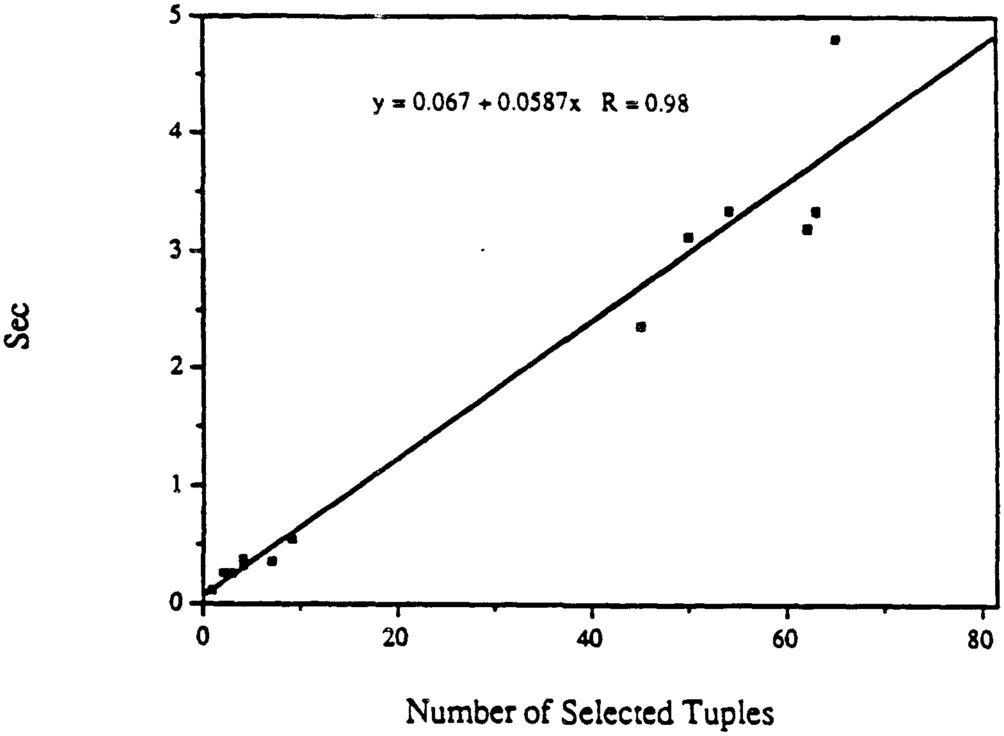


Figure 6: Actual EDB Access Time with Conventional Systems

garbage collection. But, we can estimate the EDB access time by using simple regression.[1]

4.3 Enhancing I/O Performance

One of the drawbacks for the algorithms described in previous sections is that the entire surrogate file should be read and distributed to all processing elements. This section concerns efficient surrogate file loading schemes and describes some methods which can reduce the surrogate file loading time.

4.3.1 Data Vault

The Connection Machine Model CM-2, unlike the CM-1, allows fast data transfer using multiple I/O controllers [10]. Input/Output can also be done in parallel, with as many as

[1] When a large portion of a relation is qualified in surrogate file processing, the actual EDB processing time can be reduced. Since reading a tuple takes almost the same amount of time as reading 50 tuples in the same block. But the EDB access time is several orders of magnitude larger than the surrogate file searching time in the CM.

2K processors able to send and receive data at the same time. The data processors send and rece:ve data via I/O controllers, each of which can control a set of 8 K processors. An I/O controller treats its 8K physical processors as two banks of 4K, and can be connected to a Data Vault. The Data Vault (DV) is the Connection Machine mass storage system, which provides large blocks of data with very high transfer rate. A DV can transfer data at a rate of 40 megabytes per second. Since there can be up to 8 I/O controllers for a fully configured Connection Machine, the maximum transfer rate of 320 megabytes per second can be achieved. The 8 DV's can accommodate up to 80 gigabytes of data. Thus, when 8 DV's are used, the 512 megabytes of surrogate files can be loaded in 2 seconds. Each DV stores its data in an array of 39 individual disk drives working in parallel. The DV can provide enough data transfer rate for the massively parallel environment. However, this solution is not cost effective since 8 DVs use 312 disks, each of which has the capacity of about 250 megabytes.

4.3.2 Disk Interleaving

A group of disk units is interleaved if each data block is stored in such a way that succeeding portions of the block are on different disks. This technique has been implemented for many existing computer systems [11]. By interleaving data on multiple disks, the data may be accessed in parallel reducing data transfer time by a factor of 1/n where n is the degree of interleaving. Previously proposed database machines exploited the idea of interleaving at the level of track rather than disk. Disk interleaving is more cost effective than track interleaving since conventional moving head disk can be used. Since data is interleaved among a number of disk systems, conversion is necessary in order to assemble parallel data streams coming from disks into a serial data stream and vice versa. Figure 7 shows the conversion buffer used for this purpose. As the buffer is filled, the columns are converted into the rows of a single data stream. Kim indicated that as the amount of data to be transferred is increased, the performance of disk interleaving is considerably better than parallel disk systems without interleaving. Thus, disk interleaving can enhance the performance of surrogate file loading.

4.3.3 Surrogate File Clustering

In contrast to the parallel disk systems which concerns increasing data transfer rate, clustering techniques are used to decrease the amount of data to be loaded. The increasing

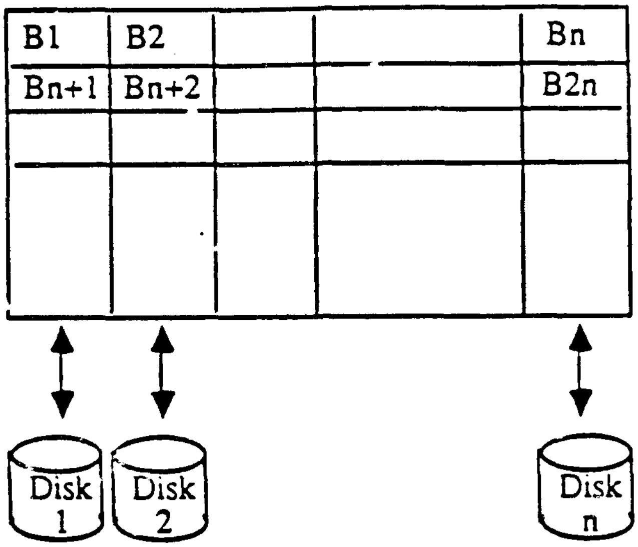


Figure 7: Conversion Buffer for Synchronized Parallel Disks

usage of databases and integrated information systems has encouraged the development of file structures specifically suited to partial match queries. Inverted file were among the earliest such file structures. However, inverted files were originally designed for single key retrieval, and hence manifest various deficiencies in particular for partial match queries. Furthermore, maintenance costs of inverted files tend to be very high when compared to other indexing methods. We describe a way to cluster surrogate files using a proposed file organization called a grid file with a binary trie directory [12] to illustrate the idea. A file structure designed to manage a disk allocates storage in units of fixed size called disk buckets. The structure used to organize the set of buckets is the heart of a file system. Suppose that we have a relation R(A,B) and the corresponding surrogate file R'(A',B'). A possible pair of A' and B' values can be plotted in a two dimensional area as shown in Figure 8. If a relation has k attributes, then k dimensional area should be considered. The surrogate file R' is now considered as a set of points scattered over this area. A sub rectangular represents a bucket with capacity of 5 codewords. Consider, for instance, a query R'(010010, 110011). For the query, only the bucket f needs to be accessed. Figure 9 (a) shows the corresponding colored binary trie directory for the grid files shown in Figure 8. Figure 9 (b) represents the data structure for a node in the colored binary trie. Since each node can be represented by a small amount of data, the directory itself can be resident with in main memory in most existing computer systems. As shown in the above example, clustering divides searching into two processing levels i.e. directory search and grid accesses. For exact match queries,

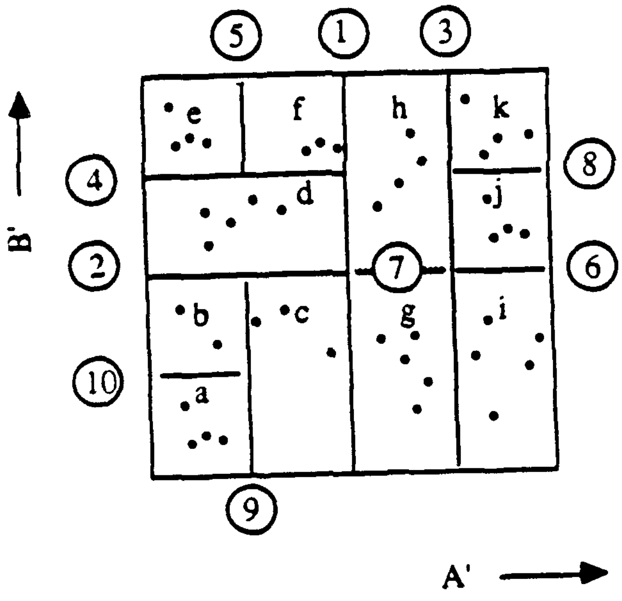
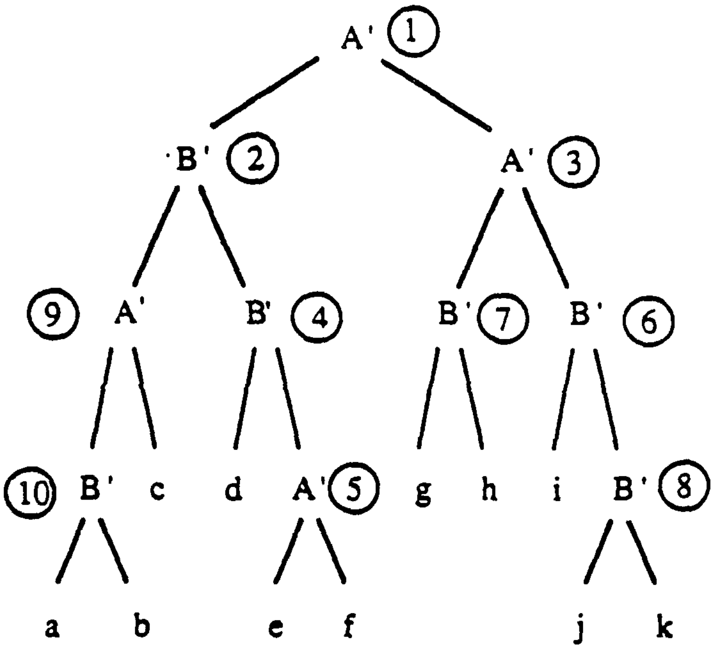


Figure 8: Surrogate File Clustering

such as used in actual EDB accesses, two disk accesses are necessary.

5 Conclusion and Future Work

In the context of knowledge base management systems, logic and databases have been extensively investigated, as both fields have much to offer the other. First order logic can enhance the functionality and representation power of current database systems exploiting general rules and allowing complex attribute values. Hence, knowledge base systems should be able to handle more complex operations than database systems; such examples include least fixed point (LFP) operation [13], unification-based retrieval [14], extended projection/selection [15], unification-join [16] and so on. In this context, we are currently developing surrogate file schemes suited to general terms and rules, and parallel algorithms for the operations mentioned above. An initial approach regarding this research is introduced in [17]. We have implemented an experimental deductive database system on the Connection Machine to test the surrogate file schemes for various partial match queries and implicit join operations. This experiment reveals that surrogate file loading time can be a major bottleneck in parallel processing environments since it not only involves the surrogate file reading time from slow secondary storage but also incurs a high communication



(a)

0	Arg Number	Left Child Pointer	Right Child Ponter

Internal Node

1	Physical Bucket Address

Leaves

(b)

Figure 9: Colored Binary Trie for the Example

overhead which is required to broadcast the codewords to a large number of processing elements. One possible solution to this problem is exploiting parallel I/O techniques such as the disk interleaving [11] or the use of specialized parallel disk systems such as the data vault used in the Connection Machine model CM-2. An effective segmentation technique is also being investigated to reduce the search space. We will continue this effort to see how well the surrogate file concept would work for very large knowledge base systems.

References

[1] C. Stanfill and B. Kahle. Parallel free-text search on the Connection Machine system. *Communications of the ACM*, 29(12):1229–1239, December 1986.

[2] D. Waltz, C. Stanfill, S. Smith, and R. Thau. Very large database applications of the Connection Machine system. In *National Computer Conference*, pages 159–165, 1987.

[3] P. B. Berra, S. M. Chung, and N. Hachem. Computer architecture for a surrogate file to a very large data/knowledge base. *IEEE Computer*, 20(3):25–32, March 1987.

[4] N. I. Hachem and P. B. Berra. Back end architecture based on transformed inverted lists, a surrogate file structure for a very large data/knowledge base. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 10–19, January 1988.

[5] S. M. Chung and P. B. Berra. A comparison of concatenated and superimposed code word files for very large data/knowledge bases. In *Proceedings of the International Conference on Extending Database Technology*, pages 364–387, 1988.

[6] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1984.

[7] K. Hughes. Interfacing Prolog with a relational database system. 1986. Syracuse University.

[8] M. Nilsson. The world's shortest Prolog interpreter? In J. A. Campbell, editor, *Implementations of Prolog*, pages 87–92, Ellis Horwood, 1984.

[9] M. Carlsson. On implementing Prolog in functional programming. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 154–159, February 1984.

[10] *Connection Machines Model CM-2 Technical Summary.* Thinking Machines Co., April 1987. Thinking Machines Technical Report HA87-4.

[11] M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[12] Y. Tanaka. Massive parallel database computer MPDC and its control schemes for massively parallel processing. In A. K. Sood and A. H. Qureshi, editors, *Database Machines*, pages 127–158, Springer-Verlag, 1986.

[13] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *ACM Symposium on Principles of Programming Languages*, pages 110–117, 1979.

[14] Y. Morita, H. Yokota, and H. Itoh. Retrieval-by-unification operation on a relational knowledge base. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 52–59, August 1986.

[15] C. Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings of the 11th International Conference on VLDB*, pages 21–23, August 1985.

[16] H. Yokota, K. Sakai, and H. Itoh. Deductive database system based on unit resolution. In *Proceedings of the 2nd Data Engineering*, pages 228–235, 1986.

[17] D. Shin and P. B. Berra. An architecture for very large rule bases based on surrogate files. In *Proceedings of the 5th International Workshop on Database Machines*, pages 555–568, October 1987.

```
/*                                              */
/*           Demonstration System Program                  */

/* The following is the listing of the demonstration system implementing */
/* the surrogate file concept for CCW and SCW.                  */

/* Any statements preceede by # or ## are EQUEL statements.         */

main()
{
# include <stdio.h>
## int menuflag,q;
## char choice[2];

FILE *fopen(),*fp; /* file used to keep track of the last UID used */

## ingres logic   /* the name of the INGRES database */


/* The following code displays the initial menu, prompting the user to    */
/* make a choice.                              */

menuflag = 0;

do {
  printf("enter the number corresponding to the task you wish to perform.\n");

  printf(" 0 initialize the uid to 0 . \n");

  printf(" 1 append tuples to a relation. \n");

  printf(" 2 find the children of a parent. \n");

  printf(" 3 find the parent of a child. \n");

  printf(" 9 to exit. \n");

  printf("enter choice : ");
  scanf("%s",choice);
  printf("\n \n");
```

```
/* Choice zero reinitializes the UID to zero; the file that keeps track of */
/* the last UID used must be opened and a zero written to it and then     */
/* closed. This choice was used only during the initial stages of programming.*/
/* Choice one calls the subroutine to add tuples to the database and thus */
/* building up our extensional database and surrogate files. Choice two   */
/* allows the user to find the child(ren) of a parent, this option enabled */
/* us to check the workings of our surrogate files before interfacing the  */
/* system with logic programming. Choice three allows the user to find     */
/* of a child. The fourth choice labled option nine exits the program.    */


  switch (choice[0]) {
     case '0' :
           q = 0;
           fp = fopen("unique","w");
           fprintf(fp,"%d\n",q);
           fclose(fp);
           break;
         case '1' :
           addtuple();
       break;

     case '2' :
           parentque();
           break;

         case '3' :
           childque();
           break;

     case '9' :
           menuflag = 1;
           break;

         default :
           printf("invalid choice entered \n");
           break;
                     }
  } while (menuflag == 0 );

## exit
}


/* The following is a simple hash function used to derive the CCW. This */
/* hash subroutine can be made more sophisticated as the system grows. */

hash(s)
char s[];
{
## int ccw;

ccw= 0;


ccw = s[0] * 256 * 256;
ccw = ccw + s[1] * 256;
ccw = ccw + s[2];
printf("\n%d\n",ccw);
return(ccw);
}


/* The following subroutine adds tuples to our extensional database,    */
/* including the associated surrogate files. The user is provided with   */
/* a menu for ease of use. The user is prompted for the relation name.   */
/* Once the relation has been chosen the variables are set to their      */
/* proper values for that particular relation.                           */


addtuple()
{
## char fa[16],ch[16];
## char name[16];
## char relative[11];
## char surrogate[16];
## Char parent[11];
## char hpa[3],hch[3];
## int mx,cnt,key,flag,i,ccw1,ccw2,rname;
## int uniden;



FILE *ffp;

ffp =(FILE *) fopen("unique","r");

fscanf(ffp,"%d",&uniden);
fclose(ffp);


printf(" enter the relation name : ");
scanf("%s",name);


if(strcmp(name,"paternal")==0) {
            rname = 1 ; }

if(strcmp(name,"maternal")==0) {
            rname = 2 ; }

## range of n is name


   /* The variables are set to the appropriate values, reflecting the */
   /* choice made.                                                     */

switch (rname) {
 case 1 :
   printf("\n paternal relation chosen \n");
   strcpy(relative,"father");
   strcpy(surrogate,"psurro");
   strcpy(parent,"father");
   strcpy(hpa,"hf");
   strcpy(hch,"hc");
   break;
 case 2 :
   printf("\n maternal relation chosen \n");
   strcpy(relative,"mother");
   strcpy(surrogate,"msurro");
   strcpy(parent,"mother");
   strcpy(hpa,"mh");
   strcpy(hch,"mc");
   break;

 default :
   printf("\n invalid relation name entered \n");
   break;
}



    /* This is the while loop that allows tuples to be added, it is */
    /* terminated when quit is detected as an input.             */

flag = 0;

while(flag == 0)
  {

            printf("\n enter %s name : ",relative);
            scanf("%s",fa);

            if (fa[0]=='q' && fa[1]=='u' && fa[2]=='i' && fa[3]=='t') {
            flag = 1;
      break;
  }

/* The hash function is called to hash the parent's name.          */

            ccw1 = hash(fa);

            printf("\n enter child's name :");
            scanf("%s",ch);
```

```
/* The hash function is called to hash the childs name.          */

          ccw2 = hash(ch);

          printf("\n %s is %s  of %s \n",fa,relative,ch);

/* The UID is incremented and the new value is written out to a file. */

uniden++;

ffp=(FILE *) fopen("unique","w");
fprintf(ffp,"%d\n",uniden);
fclose(ffp);


/* The following are EQUEL statements that append the enetered tuples to */
/* the database and to the surrogate files.                    */

## append to name(uid=uniden,parent=fa,child=ch)
## append to surrogate(hpa=ccw1,hch=ccw2,uid=uniden)
## print paternal
## print psurro
## print maternal
## print msurro
}
return;
}


/* The following subroutine will find the children of a particular father
or mother. The user is presented with a menu to facilitate usage.
Once the apropriate choice has been made the variables are set to the
values that will make the search possible.


parentque()
{
## char findpar[11];
## char fpsurrogate[11];
## char fprel[11];
## char baby[16];
## char hpar[5];
## char pronoun[4];
## char pquery[16];
## int part;
## int fqh;
## int flag1;

printf("to find the children of a father enter ----  father \n");
printf("to find the children of a mother enter ----  mother \n");
printf("enter your choice : ");
scanf("%s",findpar);


if(strcmp(findpar,"father")==0) {
        part = 1; }

if(strcmp(findpar,"mother")==0) {
        part = 2; }


   /* The variables are set to the approriate values, reflecting the */
   /* choice made.                                                   */

switch(part) {
 case 1 :
        printf("\n looking for a child's father \n");
        strcpy(pronoun,"his");
        strcpy(fpsurrogate,"psurro");
    strcpy(fprel,"paternal");
    strcpy(hpar,"hf");
        break;

 case 2 :
        printf("\n looking for a child's mother \n");
        strcpy(pronoun,"her");
        strcpy(fpsurrogate,"msurro");
        strcpy(fprel,"maternal");
        strcpy(hpar,"mh");
        break;
 default :
          printf("\n invalid choice entered \n");
          break; }


  /* The following defines what relations of the database will be accessed. */
  /* What relations are accessed depends on the values of 'fpsurrogate' and */
  /* 'fprel', which were set according to whether we are dealing with a    */
  /* maternal or paternal query. */


## range of s is fpsurrogate
## range of r is fprel



  /* This is the loop that prompts the user for the name of the parent   */
  /* for which the names of its children are to be found. If quit is     */
  /* detected the loop is terminated.                                    */


flag1 = 0;

while(flag1 == 0)
 {
          printf("\n enter the %s's name to find %s children:",findpar,pronoun);
          scanf("%s",pquery);

   if (pquery[0]=='q' && pquery[1]=='u' && pquery[2]=='i' & pquery[3]=='t') {
          flag1 = 1;
      break; }


/* The hash function is called to hash the query.  */

          fqh = hash(pquery);

printf("%d\n",fqh); printf("looking for the children of %s \n",pquery);
```

```
    /* These are the actual EQUEL statements which retrieve the desired */
    /* information.                                          */

## retrieve (baby=r.child) where r.uid = s.uid and s.hpar = fqh
## {
          printf(" %s \n",baby);
## }

}
return;
}

/* This subroutine finds the particular parent (mother or father) of a    */
/* particular child. The user is presented with a menu. The menu asks     */
/* him/her whether he/she is intersted in finding the mother or the father */
/* of a particular child. Having answered this question the variables are  */
/* set with the appropriate values reflecting the answer given.        */

childque()
{
## char findch[11];
## char chsurrogate[11];
## char chrel[11];
## char parents[11];
## char hach[5];
## char prnoun[4];
## char chquery[16];
## char partype[8];
## int kid;
## int chqh;
## int flag2;


printf("to find the father of a child enter ---- father \n");
printf("to find the mother of a child enter ---- mother \n");
printf("enter your choice : ");
scanf("%s",findch);

if(strcmp(findch,"father")==0){
```

```
          kid = 1; }

if(strcmp(findch,"mother")==0) {
          kid = 2; }


   /* The variables are set to the appropriate values, reflecting the
   /* choice made. */

switch(kid) {
 case 1 :
          printf("\n looking for the children of a father \n");
          strcpy(prnoun,"his");
          strcpy(chsurrogate,"psurro");
          strcpy(chrel,"paternal");
          strcpy(hach,"hc");
          strcpy(partype,"father");
          break;


 case 2 :
          printf("\n looking for the children of a mother \n");
          strcpy(prnoun,"her");
          strcpy(chsurrogate,"msurro");
          strcpy(chrel,"maternal");
          strcpy(hach,"mc");
          strcpy(partype,"mother");
          break;

 default :
          printf("\n invalid choice enetered \n");
          break;
}


  /* The following defines what relations of the databse will be accessed. */
  /* What relations are accessed depends on the values of 'chsurrogate'    */
  /* and 'chrel', which were set according to whether we are dealing with  */
  /* a maternal or a paternal quuery.                                  */
```

```
## range of d is chsurrogate
## range of c is chrel


   /* This is the while loop that prompts the user for the childs name, if */
   /* quit is detected then the loop terminates.                          */

flag2 = 0;

while(flag2 == 0)
  {
          printf("\n enter the child’s name to its %s : ",findch);
          scanf("%s",chquery);

   if(chquery[0]==’q’ && chquery[1]==’u’ && chquery[2]==’i’ && chquery[3]==’t’) {
    flag2 = 1;
    break;  }


          chqh =    hash(chquery);

printf("%d\n",chqh);



    /* These are the actual EQUEL statements that access the desired    */
    /* information.                                      */

## retrieve (parents=c.partype) where c.uid = d.uid and d.hach = chqh
## {
          printf(" %s \n",parents);
## }
}
return;
}
```

```
;;; ==================================================================
;;; Integrated Knowledge Base System Based on Surrogate Files
;;; File Management for Very Large Knowledge Bases Based on
;;; Surrogate Files (Concatenated Code Words).
;;; ==================================================================


;;; ------------------------------------------------------------------
;;; Global constants and variables
;;; ------------------------------------------------------------------

;;; MAXIMUM-CONJ: the maximum number of conjunctions allowed in a query

(defconstant MAXIMUM-CONJ 10)

;;; surrogate-pvar: array of parallel variables to store surrogate files
;;; surrogate-selected: bit-map resulted from surrogate file selection
;;; temp-selected: a temporary  variable for 'surrogate-selected'

(defvar surrogate-pvar (make-array MAXIMUM-CONJ))
(defvar surrogate-selected (make-array  MAXIMUM-CONJ ))
(defvar temp-selected (make-array  MAXIMUM-CONJ))

;;; Initializing parallel variables

(dotimes (k  MAXIMUM-CONJ)
  (setf (aref surrogate-selected k) (allocate!! nil!!))
  (setf (aref surrogate-pvar k) (allocate!! (!! 10000)))
  (setf (aref temp-selected k) (allocate!! nil!!)) )

,,, *array-available* is the number of surrogate files stored in the CM
;;; pvar-in-use contains EDB predicate names corresponding to surrogate-pvar

(defvar *array-available* 0)
(defvar pvar-in-use (make-array 4 :initial-element nil))
```

```
;;; idb: contains IDB clauses
;;; *edb*: the EDB predicate names
;;; no-of-clauses: the number of clauses in the current program

(defvar *idb* nil )
(defvar *edb* nil )
(defvar no-of-clauses 0)


;;; environment: variable bindings represented by bit-maps
;;; temp-result, intermediate-result: temporary storage for selection
;;; *mask*: masking some fields for selection
;;; Note:
;;; *mask* is served as the mask bits in associative memories.
;;; If a field of the given querv is a variable, the corresponding *mask*
;;; field is set to 10000000...0000, otherwise 1111111 .. 111 (i.e. normalized).

(defvar environment (make-array  MAXIMUM-CONJ))
(*defvar temp-result nil!!)
(*defvar intermediate-result nil!!)
(*defvar *mask* (!! 0))

;;; no-of-predicate-in-conjunction: the number of predicate in the current
;;;                         conjunction
;;; proved-var-list: the list of variables that have been instantiated.

(defvar no-of-predicate-in-conjunction)
(defvar proved-var-list)


;;; -----------------------------------------------------------------
;;; SIMPLE PROLOG INTERPRETER
;;; Special Commands : "load" loads new IDB
;;;              "listing" displays the current program
;;;              "halt" terminates the execution of this program
;;; Note: IDB file names should be capital
;;;     EDB predicates should be declared with arities
;;; -----------------------------------------------------------------
```

```
(defun toplevel ()
 (progn
   (format t "Surrogate File Demonstration System - 1")
   (terpri)
   (do ((goal nil))
      ((equal goal 'halt) 'END-OF-EXECUTION)
      (progn
        (setq goal (and (print '?- )(read)))
        (cond ((equal goal 'load ) (initialize-system))
             ((equal goal 'listing ) (listing))
             ((listp goal)  (solve (list goal )))
        ))
   )))


;;; Initialization of some global variables

(defun initialize-global-vars ()
  (progn
   (setq  *array-available* 0)
   (setq  no-of-clauses 0)
   (dotimes (j (length surrogate-pvar))
    (*all
      (*set (aref surrogate-pvar j) (!! 0)))
   )
 ))


;;;  'Consulting' a new set of IDB and load surrogate files  specified
;;;  by the IDB to the pvars in the CM.
;;;  EDB should be declared as a part of the program

(defun initialize-system()
 (let ((fname nil))
  (initialize-global-vars)
  (format t "IDB FILE NAME :   ")
  (setq fname (read))
  (cond ((probe-file fname)
             (progn
           (setq *idb* nil)
```

```
          (setq *edb* nil)
          (with-open-file ( ifile fname :direction :input)
            (do ( (k 0 (+ k 1))
                (edb nil)
                (clause (read ifile nil) (read ifile nil)))
                ((eq clause nil) (setq no-of-clauses k))
                 (setq  *idb* (append *idb* (list clause)))
                (if (eq (caar clause) 'edb )
                 (progn
                   (setq edb (cdar clause))
                  (cm:time
                   (dotimes (k (length edb))
                     (load-surrogate (car (nth k edb)) (cadr (nth k edb)))
                     (setq *edb* (append *edb* (list (car (nth k edb)))))
                   ))))))))
       (t (format t "~S does not exist" fname)))
  ))


;;; Listing the clauses including the EDB predicates

(defun listing ()
  (progn
    (format t "IDB")
    (dotimes (k no-of-clauses)
         (print (nth k *idb*)))
    (terpri)
    (format t "EDB")
    (dotimes (k *array-available*)
         (print (aref pvar-in-use k)))
  ) )

;;; -----------------------------------------------------------------
;;; HASHING
;;; -----------------------------------------------------------------

;;; Constants
;;;   FIELD-LEN is the length of each code word
;;;   FIELD-VAL is the upper bound of a code word
;;;   NORM-VAL is the minimum value for a code word
;;;  CONS-MASK is mask for a constant argument
;;;              (the maximum value for the code word)
;;;  VARS-MASK is mask for a variable argument

(defconstant FIELD-LEN 16)
(defconstant FIELD-VAL (expt 2 FIELD-LEN))
(defconstant NORM-VAL (expt 2 (1- FIELD-LEN)))
(defconstant CONS-MASK (1- FIELD-VAL))
(defconstant VARS-MASK NORM-VAL)

;;; A simple hashing function. It returns a normalized integer value between
;;; NORM-VAL and FIELD-VAL. Normalization is required for logical AND and OR
;;; operations

(defun hash (a-str)
 (let ((hash-value 0))
  (dotimes (k (length a-str))
    (setq hash-value (+ (* 26 hash-value)
                     (- (char-code (char a-str k)) 64) )))
  (setq hash-value (mod hash-value FIELD-VAL))
  (cond ((< hash-value NORM-VAL) (+ hash-value NORM-VAL))
      (t hash-value))
  ))

;;; ------------------------------------------------------------------
;;; BASIC SURROGATE FILE HANDLING ROUTINES:
;;; ------------------------------------------------------------------

;;; make-surrogate: create a surrogate file
;;;             file-name is actually a EDB predicatename.
;;;             Note: If the given file-name is FATHER, its corresponding
;;;                 surrogate file name is FATHER.SUR.

(defun make-surrogate ( file-name )
  (with-open-file (ofile (concatenate 'string (symbol-name file-name)  ".SUR")
             :direction :output :if-exists :supersede
             :element-type ‘(mod ,FIELD-VAL) )
    (do ((clause (read ifile) (read ifile))
       (cl nil) (hashed-arg 0))
      ((null clause))
```

```
        (setq cl clause)
        (dotimes (i (length cl))
           (setq hashed-arg  (hash (symbol-name (nth i cl))))
           (write-byte hashed-arg  ofile)
         ))
     ))


;;; load-surrogate: loading surrogate files to surrogate-pvar
;;;               returns the number of tuples in the 'filename'

(defun load-surrogate ( filename  arity )
  (with-open-file (ifile (concatenate 'string (symbol-name filename)  ".SUR")
                 :direction :input :element-type ‘(mod ,FIELD-VAL) )
   (progn
    (do ((k 0 (+ 1 k))
        (code-value 1)
        (ll 0))
       ((= code-value 0) k)
       (dotimes (i arity)
         (setq ll (read-byte ifile nil 0))
         (setq code-value (+ (* code-value (* i FIELD-VAL )) ll)))
         (setf (pref (aref surrogate-pvar *array-available* ) k) code-value)
       )
    (setf (aref pvar-in-use *array-available*) filename)
    (setq *array-available* (1+ *array-available*))
   )))


;;; -----------------------------------------------------------------
;;; MAKING A QUERY CODE WORD
;;; -----------------------------------------------------------------


;;; Making a query code word

(defun generate-qcw (argument-list)
 (let ((arity (length argument-list))
     (qcw 0)
     (mask-field 0)
     (argument nil))
   (dotimes (kk arity qcw)
      (setq argument (nth kk argument-list))
      (setq code (cond ((and (listp argument)
                             (variable-symbol-p (car argument)))
                          VARS-MASK)
                        ((stringp argument)  (hash argument))
                        ((symbolp argument) (hash (symbol-name argument)))
                  ))
      (setq qcw (+ (* qcw FIELD-VAL) code))
    )
 ))


;;; Detect variable symbols
;;; Variable names start with ? [Nilsson 84][Carlsson 84]

(defun variable-symbol-p (x)
  (and (symbolp x) (eq #\? (char (symbol-name x) 0)))
 )


;;; ------------------------------------------------------------------
;;; PARALLEL QUERY PROCESSING BASED ON SURROGATE FILES
;;; ------------------------------------------------------------------

;;;; Solves a conjunction of literals

(defun solve-conjunction (edb-predicates)
   (setq no-of-predicate-in-conjunction
     (cond ((symbolp (car edb-predicates)) 1)
          (t (length edb-predicates))))
  ; constructing the list of boolean pvars
  (terpri)(format t "Surrogate File Selection Time")
  (cm:time
  (dotimes (k no-of-predicate-in-conjunction)
     (setf (aref environment k) (nth k edb-predicates))
     (*set (aref surrogate-selected k)
          (selection-on-surrogate (nth k edb-predicates)))))
  (cond ((*null) (terpri)(format t "No (by surrogate file selection)"))
        ; immediate failure
        ((= (length edb-predicates) 1)
              (terpri) (format t "Actual EDB Loading time")
              (time (simple-selection (car edb-predicates) 0)))
        ; when the conjunction has only one predicate
        (t ; reducing the size of virtual relations
          ; false-drop elimination from virtual relations
         (terpri)(format t "Join on SF")
         (cm:time (solving-conjunction-surrogate))
          ; sets global boolean pvars
          (cond ((*null) (terpri)(format t
                 "No (by relational operations on surrogate files)"))
                ; immediate failure after restriction
               (t (terpri)(format t "Actual EDB Loading Time")
                     (time (relational-operations)))
          ))
        ))


;;; Selection on the surrogate files
;;; Returns boolean pvar
;;; BUGS: it can't check p(x, x), i.e. shared variables in a predicate.

(*defun selection-on-surrogate (goal)
  (=!! (!! (generate-qcw (cdr goal)))
      (logand!! (!! (set-mask-field (cdr goal)))
              (aref surrogate-pvar (find-surrogate-file (car goal))))
  ))


;;; According to the predicate name, access the appropriate surrogate file
;;; Returns the index

(defun find-surrogate-file (predicate-name)
 (let ((sf-index 0))
  (dotimes (k *array-available* sf-index)
    (if (equal (aref pvar-in-use k) predicate-name) (setq sf-index k))
  ) ))
```

```
;;; reducing the size of selected tuples

(defun solving-conjunction-surrogate ()
  (let ((sort-key (sort-by-selectivity))
        (temp-env (make-array  no-of-predicate-in-conjunction
                        :initial-element nil)))
  ; reodering the goals by selectivity
   (dotimes (k no-of-predicate-in-conjunction)
    (setf (aref temp-env k)
     (aref environment (car (nth k sort-key))))
    (*set (aref temp-selected k)
          (aref surrogate-selected (car (nth k sort-key)))))
   (dotimes (k no-of-predicate-in-conjunction)
     (setf  (aref environment k) (aref temp-env k) )
     (*set (aref surrogate-selected k) (aref temp-selected k)))
   (setq proved-var-list nil)
   (dotimes (k (1- no-of-predicate-in-conjunction))
     (equijoin-surrogate k))
  ))


;;; Nested Loop Join
;;; index can be used for goal in environment & selected tuples
;;; in surrogate-selected

(defun equijoin-surrogate (index)
; i is used to indicate ith argument of (aref environment index)
 (let ((arg (cdr (aref environment index)))
      (s1 (find-surrogate-file (car (aref environment index))))
      (s2 0) (cell nil)
      (arg-to-prove nil) (rest-goal-args nil))
   (dotimes (i (length arg))
    (setq arg-to-prove (nth i arg))
    (cond ((or (symbolp arg-to-prove)
               (member arg-to-prove proved-var-list :test #'equal))  nil)
          ; if the corresponding argument is constant or already proven
          ; variable cell then do nothing
          (t
            ; k is used to indicate rest goal positions in the environment
           (setq proved-var-list (append proved-var-list (list arg-to-prove)))
      (do ((k index))
         ((= k (1- no-of-predicate-in-conjunction)) )
          (setq k (1+ k))
          (setq rest-goal-args (cdr (aref environment k)))
          (setq s2 (find-surrogate-file (car (aref environment k))))
          ; j is used to indicate the argument in target goal
          (dotimes (j (length rest-goal-args))
            (setq cell (nth j rest-goal-args))
            (cond ((not (equal arg-to-prove cell)) nil)
                 (t (refine-selected-set s1 index i s2 k j))))
     )))) ))


;;; Parallel Part of equijoin-surrogate

(defun refine-selected-set (s1 index-1 arg-pos-1 s2 index-2 arg-pos-2)
  (let ((tqcw-index nil)
      (tqcw-list nil)
      (qcw (set-mask index-2 arg-pos-2)))
   (*when (aref surrogate-selected index-1)
     (do-for-selected-processors (k)
    ; making qcw
      (setq tqcw 0)
      (dotimes (i (length (cdr (aref environment index-2))))
        (setq tqcw (+ (* tqcw FIELD-VAL)
             (cond ((= i arg-pos-2)
                    (mod (truncate (/ (pref (aref surrogate-pvar s1) k)
                                   (expt 2 (* FIELD-LEN
                                     (1- (- (length
                                     (cdr (aref environment index-1)))
                                          arg-pos-1))))
                                   ))
                           FIELD-VAL))
                   (t VARS-MASK)))))
       (setq tqcw-index (cons k tqcw-index))
       (setq tqcw-list (cons tqcw tqcw-list))
      ))
  (*all
   (*set intermediate-result nil!!)
   (*set *mask* (!! qcw))
```

```
     (dotimes (i (length tqcw-index))
       (*set temp-result (=!! (!! (nth i tqcw-list))
                             (logand!! *mask* (aref surrogate-pvar s2))))
       (if (= 0 (selectivity temp-result))
          (setf (pref (aref surrogate-selected index-1)
                  (nth i tqcw-index)) nil)
            (*set intermediate-result (or!! intermediate-result temp-result))
            )
      )
     (*set (aref surrogate-selected index-2)
          (and!! (aref surrogate-selected index-2)
                            intermediate-result))
    )))


(defun *null ()
 (let ((is-fail t))
   (dotimes (k no-of-predicate-in-conjunction )
      (cond ((= (selectivity (aref surrogate-selected k)) 0)
                (setq is-fail t) (return))
            (t  (setq is-fail nil))))
    is-fail))

 (defun set-mask (environment-index arg-pos-index)
  (let ((qcw-mask 0) (arg (cdr (aref environment environment-index))))
    (dotimes (k (length arg) qcw-mask)
      (setq qcw-mask (+ (* FIELD-VAL qcw-mask)
                       (cond ((= k arg-pos-index) CONS-MASK)
                             ((and (listp (nth k arg))
                              (variable-symbol-p (car (nth k arg)))) VARS-MASK)
                             (t CONS-MASK))))))
    )

(defun set-mask-field (arg)
   (let ((qcw-mask 0))
    (dotimes (k (length arg) qcw-mask)
      (setq qcw-mask (+ (* FIELD-VAL qcw-mask)
                       (cond ((and (listp (nth k arg))
                              (variable-symbol-p (car (nth k arg)))) VARS-MASK)
                             (t CONS-MASK))))))
  )

;;; Primary False Drop Detection; checking constants argument
;;; and Creates Virtual relations
;;; Immediate return when creating null virtual relations

(defun relational-operations ()
 (let ((virtual-relations nil))
 (dotimes (k no-of-predicate-in-conjunction)
   ; Creating Virtual Relation
   (setq virtual-relations
     (append virtual-relations
       (list (simple-selection (aref  environment k) k))))
 )
))

;;; Sorts the global variables environment and surrogate-selected
;;; based on the selectivity
;;; returns the sorted index

(defun sort-by-selectivity ()
 (let ((sort-key nil))
  (dotimes (k no-of-predicate-in-conjunction)
    (setq sort-key (append sort-key
      (list (cons k (selectivity (aref surrogate-selected k)))))))
  (sort sort-key  #'< :key #'cdr)
 ))


;;; Determines how many surrogates are selected

(defun selectivity (x)
  (*when x (*sum (!! 1))))

;;; Accessing the actual EDB (optimized version)
;;; After surrogate file processing has been completed, actual EDB tuples are
;;; loaded to main memory.

(defun simple-selection (goal index)
 (let ((result nil) (old-block-no -1)
    (old-rec-no 0) (block-number 0)
    (rec-no 0) (ifile nil)
    (tuple nil) (filename (car goal))
    (arg (cdr goal)))
    (*when (aref surrogate-selected index)
     (do-for-selected-processors (i)
       (setq block-number (truncate (/ i BLOCK-SIZE)))
       (setq rec-no (mod i BLOCK-SIZE))
       (cond ((= block-number old-block-no)
            (dotimes (k (1- (- rec-no old-rec-no)))
               (read ifile))
            (setq tuple (read ifile))
            (setq old-block-no block-number)
            (setq old-rec-no rec-no))
        (t
            (setq new-file-name
             (merge-pathnames *default-pathname-defaults*
              (make-pathname :name filename :version block-number)))
             (if (streamp ifile) (close ifile))
             (setq ifile (open new-file-name :direction :input))
             (dotimes (k  rec-no)
               (read ifile))
              (setq tuple (read ifile))
            (setq old-block-no block-number)
            (setq old-rec-no rec-no)
              ))
        (if (filter arg tuple)
            (setq result (append result (list (project arg tuple)))))
    ))
    (close ifile)
    (print (cons (var-filter arg) result))
))

;;; Eliminates constant arguments from the argument list

(defun var-filter (arg-list)
 (mapcan #'(lambda (X) (and (listp x) (variable-symbol-p (car x)) (list x)))
   arg-list))

;;; False Drop Detection
```

```
(defun filter (argument result-tuple)
 (every
   #'(lambda (x y) (or (and (listp x) (variable-symbol-p (car x)))
                  (equal x y)))
   argument result-tuple)
 )

;; Eliminating the common constants from the result tuple

(defun project (arg tuple)
  (mapcan #'(lambda (x y) (and (and (listp x) (variable-symbol-p (car x)))
                      (list y))) arg tuple)
  )


;;;------------------------------------------------------------------
;;; PROGRAMS FOR TRANSLATING GOALS TO EDB PREDICATES
;;;------------------------------------------------------------------

;;; Variable
;;; *var-level* : used to avoid name conflict between variables.
;;;           Level 0 variables are those appeared in the original
;;;           goal specified by users

(defvar *var-level* 0)


;;; Constructing conjunctions of edb predicate and solving each conjunction
;;; Assume that the variables in the goal should be shared among each literal
;;; No recursive clause and function argument are considered.

(defun solve (goal-list)
 (let ((conjunction-list nil)
      (new-goal-list nil))
     (setq *var-level* 0)
     (setq new-goal-list (free-var-substitute goal-list))
     (setq conjunction-list (unify-subgoal new-goal-list))
     ;; Solve conjunctions
     (dotimes (k (length conjunction-list))
      (format t "Solving ... ")
      (print (nth k conjunction-list))
     (cm:time (solve-conjunction
        (cond ((symbolp (car (nth k conjunction-list)))
             (list (nth k conjunction-list)))
            (t (nth k conjunction-list))))
    ))))


;;; Making a list of candidate clauses.

(defun find-candidate ( goal)
 (mapcan #'(lambda (x) (and (unifiable goal x) (list x))) *idb*))

(defun unifiable (goal clause)
 (and (equal (car goal) (caar clause))
   (= (length goal) (length (car clause))) ))


;;; Unify a subgoal

(defun unify (goal )
 (let ((unified nil)
    (candidate-clause (find-candidate goal)))
 (dotimes (k (length candidate-clause) unified)
   (setq unified (append (unify-subgoal (substituting-candidate goal
                      (nth k candidate-clause))
                        ) unified)))
 ))


;;; Instantiate free variables by unification

(defun substituting-candidate (goal clause)
 (let ( (binding nil)
     (new-clause nil))
   (setq binding (find-binding (cdr goal) (cdar clause)))
   (setq new-clause (mapcar  #'(lambda (x) (sublis binding x)) (cdr clause)))
   (setq *var-level* (1+ *var-level*))
   (setq new-clause (free-var-substitute new-clause))
 ))

(defun find-binding (goal-arg head-arg)
 (mapcar #'cons head-arg goal-arg)
)



;;; Resolving a conjunction to edb predicates.
;;; This function calls unify recursively.

(defun unify-subgoal (body)
 (let ((resolved nil))
  (cond ((null body) nil)
       (t
        (dotimes (k (length body) resolved)
         (if (member (car (nth k body)) *edb*)
          (setq resolved (cartesian-product resolved (list (nth k body))))
          (setq resolved (cartesian-product resolved (unify (nth k body))))
         )))
)))


;;; Substituting the free variables in the body ( conjunctions )
;;; by using *var-level* to avoid variable name conflicts

(defun free-var-substitute (body)
  (map 'list #'(lambda (y) (map 'list #'sub y))  body))

;;; Substituting variable symbol with variable cell

(defun sub (x)
  (cond ((variable-symbol-p x) (cons x *var-level*))
       (t x)))

;;; Appending the alternative clauses to make conjunction

(defun cartesian-product (list1 list2)
  (let ((result nil))
    (setq result nil)
```

```
   (cond ((null list1) (setq result list2))
      ((null list2) (setq result list1))
      (t
          (dotimes (k (length list1))
            (dotimes (j (length list2))
              (setq result (append result (list
                 (cons-1 (nth k list1) (nth j list2)))))))))
   result ))


(defun cons-1 (a b)
 (cond ((and (symbolp (car a)) (symbolp (car b))) (list a b))
      ((symbolp (car a))  (cons a b))
      ((symbolp (car b))  (append a (list b)))
      (t (append a b))))


;;; ---------------------------------------------------------------------
;;; CREATING SAMPLE EDBS BY GENERATING RANDOM NUMBERS
;;; ---------------------------------------------------------------------

;;; Constants.
;;; BLOCK-SIZE:  the maximum number of tuples that can be stored in a file.
;;;            If make-edb generates more than BLOCK-SIZE tuples,
;;;            create-block creates subsequent blocks
;;;            (e.g. FATHER#0, FATHER#1 ........ ).
;;; MAX-FIELD + 1: the maximum number of characters that can appear in an
;;;              argument.

(defconstant BLOCK-SIZE 50)
(defconstant MAX-FIELD 5)


;;; Creating an EDB and its surrogate file by generating random numbers
;;;  filename : an EDB predicate name
;;;  fields : The number of arguments in the predicate
;;;  cardinality : The number of tuples to be generated

(defun make-edb (filename fields cardinality)
 (let ( (file-tag 0) (buffer (make-array BLOCK-SIZE :initial-element nil))
    (cnt 0) (tuple nil) (tmp 0) (hashed-arg 0) (arg nil)(k 0))
   (with-open-file (ofile
           (concatenate ’string (symbol-name filename)  ".SUR")
           :direction :output :if-exists :supersede
           :element-type ‘(mod ,FIELD-VAL) )
    (dotimes (i cardinality )
     (setq tuple nil)
     (dotimes (j fields)
      (setq k (+ (random MAX-FIELD) 1))
      (setq arg nil)
      (dotimes (l k)
       (setq tmp (+ 65 (random 26)))
       (setq arg (concatenate ’string arg (list (code-char tmp)))))
      (setq hashed-arg  (hash arg))
      (write-byte hashed-arg  ofile)
      (setq tuple (append tuple (cons arg nil)))
    )
    (setf (aref buffer (mod i BLOCK-SIZE)) tuple)
    (cond ((= (mod i BLOCK-SIZE) (1- BLOCK-SIZE))
        (setq new-file-name
          (merge-pathnames *default-pathname-defaults*
            (make-pathname :name filename :version file-tag)))
        (setq cnt BLOCK-SIZE)
        (create-block new-file-name cnt buffer)
        (setq file-tag (+ file-tag 1)))
       ((= i  (1- cardinality))
        (setq new-file-name
         (merge-pathnames *default-pathname-defaults*
           (make-pathname :name filename :version file-tag)))
        (setq cnt (1+ (mod i BLOCK-SIZE)))
        (create-block new-file-name cnt buffer)
       ))
   ))))


;;; Creating a subfile
;;; The main purpose of dividing the files into subfiles is for efficient
;;; access to secondary storage. We are using sequential file access

(defun create-block (filename count buffer)
(with-open-file (ofile filename :direction :output
                :if-exists :supersede)
  (dotimes (k count)
   (princ (aref buffer k) ofile))))
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.