SEMANTIC PAGING ON INTELLIGENT DISCS*

G. Jack Lipovski
Department of Electrical Engineering
The University of Texas at Austin
Austin, TX 78712

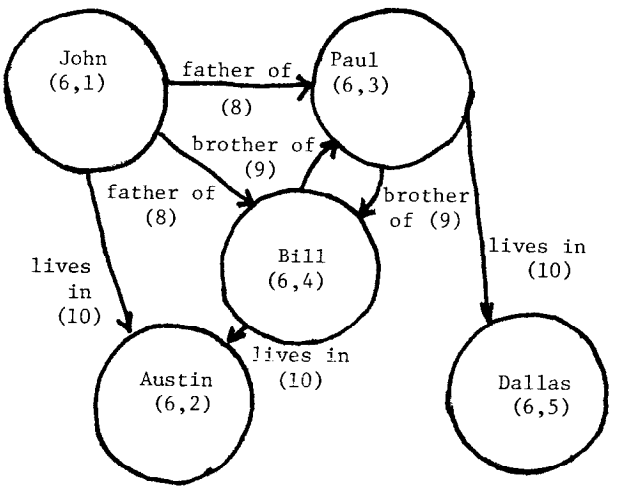
Abstract

In order to apply Artificial Intelligence techniques like inferencing to large (10^9-10^{12} bit) data bases, an intelligent secondary memory is proposed that is capable of extracting a subgraph from a graph stored in it for further processing in a conventional or special purpose computer. The "pointer transfer" technique developed for CASSM is used to efficiently mark and output a subgraph stored in one file, and a simple technique is proposed to link parts of the subgraph stored on different files. Besides proposing a useful secondary memory system for applying Artificial Intelligence techniques to large data bases, this paper attacks one of the most difficult problems in distributed data base systems, which is the generation of "homomorphic query fragments" that automatically carry the query from one file to other files needed to resolve the query.

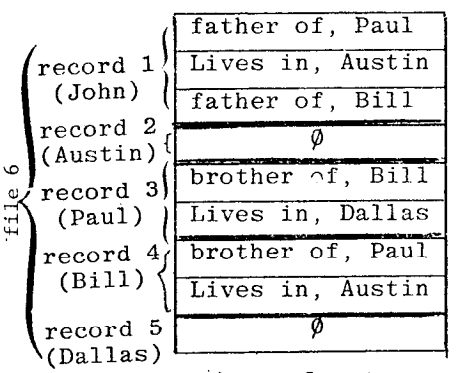
I. Introduction

The semantic network structure offers the potential of extending the power of data base management systems (DBMS) through techniques developed for artificial intelligence (AI) such as interpreting inferences. Whereas AI programs have had difficulty managing a million word data base in primary memory, and DBMS's require a few orders of magnitude more data, an intelligent secondary memory is indicated. Shallow structures can probably be analyzed on the disc using "demons." However, our experiences with CASSM [1] indicated some difficulty in analyzing deep structures met in AI techniques on a slow disc. We therefore propose to use an intelligent disc to "prefilter" the data, to automatically select a useful part of data stored on a disc for deep structure analysis in conventional computers. In this respect, the disc is used for a DBMS as in paging in virtual memories. However, rather than organizing data in pages in a random access memory to effect virtual memories, AI data should be stored directly in semantic networks, with a "page" defined in terms of a subgraph of the semantic network, that is specified at run time as the "page" is being moved. Intelligent hardware reduces the need for memory management software by searching for "pages" defined at run time, so that "pages" do not have to be predefined in the DBMS when the data is stored on the disc.

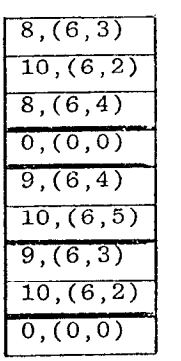
A semantic network model can be described for purposes of this article as a directed graph whose arcs have names. See Figure 1a. Note that arcs can be incident only between nodes, not to other arcs, and that non-directed edges have to be expressed as a pair of symmetric directed arcs.



a) Information Structure



b) Data Structure



c) Storage Structure

Figure 1. Semantic Network

* This paper was prepared with support from the NSF grant MCS77-15968.

For example, a node, like "John," has an arc to "Paul" that is named "father of." A semantic page is defined by (Q,d), as a set of nodes S that satisfy a query Q, and the subgraph of all nodes and arces which are within Hamming distance d of any node in S in the directed graph. For example, Q may specify the set of all nodes having arcs "father of" to "Paul" and "lives in" to "Austin." Then, (Q,2) would be the subgraph consisting of all nodes in S (specified by Q), all nodes at the head of an arc are incident from these nodes, and all arces incident from all these nodes. The object of the intelligent secondary memory is to store a large (10^9-10^{12}) bit semantic network, and be able to insert, modify, replace, delete, and read out semantic pages defined by any given (Q,d) over the entire data base.

In the remainder of this paper, we will present the hardware and data structure of an architecture for semantic paging on intelligent discs. We discuss the key problem of transferring tokens over pointers in the same file, then over different files. Then we consider the problem of data integrity in multiple file updates, and then we present our conclusions.

II. Hardware and Data Structure

Following the approach used in CASSM RAP [2] and LEECH [6] the hardware consists of an ensemble of head-per-track discs with a "microprocessor" on each head, and a large backup tertiary store and a rather conventional computer to schedule and control queries, as well as buffer data generated by queries. The data base is actually stored in the tertiary memory or distributed over a network, and is divided into rather large files (10^9 bits) such that a file can be effectively loaded into and searched on the ensemble of head-per-track discs. A file could be the data on a cylinder of each of an ensemble of IBM 3330-type discs and the file being searched would be on the cylinder under the heads of each 3330. Alternatively, a file could be the data at a node in a computer network that can be searched on the head-per-track discs or under the moving heads of 3330's, at that site. The file is further divided into equal length segments, each of which is stored on a track of the disc, and is searched by a "microprocessor" on the head, such that one search is conducted over the entire segment in one disc revolution. See the right half of figure 2. Thus, in each disc revolution, one entire file is searched according to some search instruction.

Independent of the hardware segmentation of the file, the file is composed of a 1 dimensional array of variable length records of fixed length words. The words are divided into three fields. See the left half of figure 2. A node of the semantic network is stored as a record, such that its words correspond to arcs incident out of the node. For simplicity, each word is an ordered pair $(a,(b_1,b_2))$ where a is a code word for the name of the arc and (b_1,b_2) is a logical address of the node at the head of the arc. b_1 identifies the file and b_2 identifies the record in the file that stores the node. (Actually, in a practical implementation, b_1 can be deleted if it is the number of the same file that this word is on.) Additionally, the logical address of the record implies a code word name for the node. That is, (b_1,b_2) is both the code word for the node and the location of the node, since the node is the b_2th record from the top of the b_1th file. See Figure 1b. The node John has words corresponding to arcs "father of" "Paul" and "lives in " " Austin." Since the node is stored on, for instance, the first record in the sixth file, the node has a code word representation (6,1). If "father of" is coded as code word 8, and "Paul" is a node that is stored in the 6th file, and is the second record from the top of the sixth file, then the arc is represented by (8(6,2)). See figure 1c. Significantly, the record that represents a node may overlap two or more segments (tracks), and more than one record can be stored in a segment (but a node is all stored on one file.) Such records can be processed by the user in a CASSM type system as if they were on the same segment, using intelligent hardware to link the segments together.

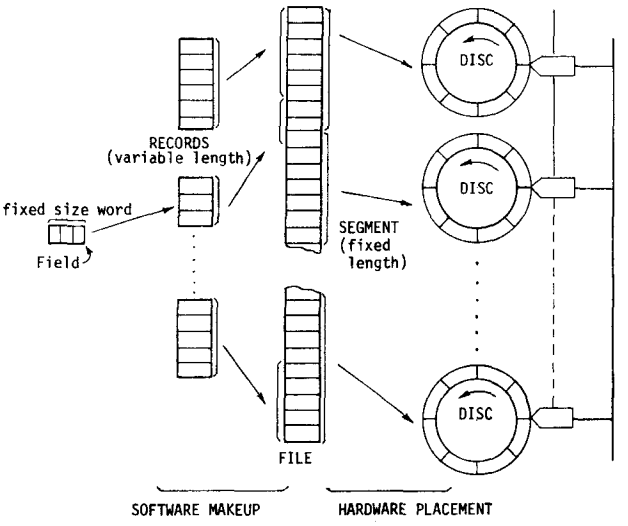


Figure 2

III. Token Transfer Across Pointers within the Same File

A typical query will normally contain a code word (i.e. a logical address) for a node. For example, it will instruct: "mark node (6,2)." Or it might be qualified by a set of arcs incident out of the node, such as mark all nodes having arcs (8,(6,3)), to identify one or more nodes. More complex queries will define a set of nodes by a subgraph with some unknowns as in the ASP language [3]. See Figure 3. Some typical instructions would be: "set membership" instruc-

tions such as "mark all records (nodes) that have code word (b_1,b_2)," "mark all records that have an arc $(a,(b_1,b_2))$," "unmark all records that do not have an arc $(a,(b_1,b_2))$," or "token transfers across arc" instructions such as "from all marked records, move the mark across the arc incident out of those named "a" to mark the records (b_1b_2)," or "from all marked records, move the mark across all arcs incident out of them and mark the records at the head of the arcs." Generally, each instruction above will take one disc revolution although marking a node with a specified code word takes no disc revolutions and token transfer instructions may occasionally take extra disc revolutions to sort out memory contention conflicts as discussed by Bush [4]. Thus, if the semantic page resides entirely in a file, a typical query Q may take a small number of instructions (disc revolutions), depending on the number of conditions in the query, and the semantic page (Q,d) will take d additional instructions (disc revolutions) of the last type mentioned above. The semantic page will be output directly during the last d revolutions.

The problem is quite simple if the semantic page (Q,d) resides entirely within one file, and the basis for doing this is essentially already implemented in CASSM. The mechanism called "pointer transfer" [1] or "transferring a token across a pointer" [5] is used to find nodes at the head of selected marks and "set search" [1] is used, to identify nodes by code words. The token transfer mechanism can be sped up by using three one bit wide random access memories RAM_0, RAM_1 and RAM_2 having as many bits as there are nodes in the file. In one cycle RAM_0 is cleared

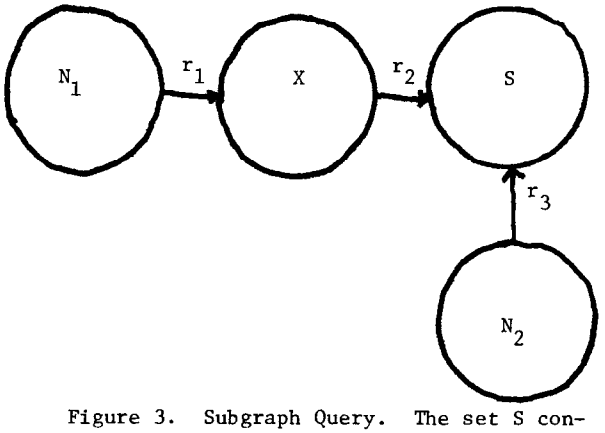


Figure 3. Subgraph Query. The set S consists of all nodes of distance 2 from node N_1 and of distance 1 from N_2 (X is a don't care)

while RAM_1 is written into, and RAM_2 is read from. In the next cycle, RAM_0 is written into, RAM_1 is read from and RAM_2 is cleared. This generation is repeated, cyclically clearing RAM_2 then RAM_0 then RAM_1 and so on. Such cyclical operation permits pipelining so that tokens can be transferred across all arcs from marked records in effectively one disc revolution. Basically, the instruction "from all marked records follow the arc named a to mark records (b_1b_2)" works this way in the cycle when RAM_0 is being cleared. The ith record is initially considered "marked" if the ith bit of RAM_2 is 1. This is "marked" by the query Q. For instance, "mark" node (6,2) would set the second bit of the RAM. Marking can also be done by set search or token transfer instructions. As the ith record passes over the disc head (determined simply by a counter) if bit i is 1 and a word with arc name "a" is encountered, and b_1 corresponds to this file, then the b_2th bit of RAM_1 is set. In the next revolution the ith record is marked if bit i of RAM_1 is set so that another instruction "from all marked words" can be executed in that cycle, and so on. The contents of the nodes so marked are output concurrently. The other instructions of this machine are similar. Deletion can use automatic hardware garbage collection as in the CASSM system [7]. While this discussion works efficiently for a single cell, efficient operation of a multiple cell machine follows from the approach discussed in Bush et. al. [4].

IV. Token Transfers Across Pointer Between Files

Modifications to CASSM, of particular interest to distributed data base processing, concern the problem of extracting and updating semantic pages that overlap files. However, the modifications are really quite simple. These are discussed in the context of files stored on cylinders of IBM 3330-type moving head discs. This discussion can obviously be extended to distributed data bases where files are on different network nodes. The concept of saving intermediate information in memory is augmented by sending this intermediate information through the network. The principles and techniques are the same, although the cost parameters would be different.

We consider the problem of outputting a semantic page, although the mechanisms for deleting, creating, or rewriting a semantic page are similar. Mechanisms for preserving the integrity of the data base for deletion and rewriting are considered later. At the outset, the code words for nodes are already explicitly encoded as a pair (b_1,b_2) so that b_1 indicates which file contains the node, and implies therefore which file is loaded from tertiary memory to the intelligent secondary memory or over which cylinder of a 3330 type disc to position the heads. If a query contains several codes words with different values for b_1, then the query must be separately executed in "fragments" on such files as are indicated by these values. Moreover, in following an arc from all marked records to mark all records at the head of these arcs, if the arc is on a different file than that currently residing in secondary storage, it cannot be marked until that file is brought in other heads are moved. A "fragment" has to be generated here too. Both concepts, called "homomorphic query fragments", are discussed more fully as a product of a search on a file in the next paragraph. The scheme, simply put, is to store such "fragments" in a buffer in the supporting computer, until the file can be brought in or the heads are moved. Some reflection on

the need to avoid thrashing files in and out of the secondary store indicates that several pending queries will be fragmented at the outset, such that query fragments of several queries will be executed on the same file. Other fragments will also be generated as some query fragments are executed on files. Therefore the supporting computer will buffer these fragments and schedule the loading of files or movement of the heads and execution of query fragments on them in order to minimize thrashing. (Equivalently, in a network, fragments are separated for each node and collected at each node for processing there.)

In order to find a semantic page that is split among two or more files, buffers will be created and the instruction i and operand b_2 will be kept in a buffer for executing query Q on file b_2. As a query token transfer instruction is being executed on a file, when a pair $(b_1 b_2)$ is encountered as a node at the head of the transfer and b_1 is not the file currently on the secondary storage, then a buffer for executing query Q on file b_1 will be created and an instruction i with operand b_2 will be put there where i is supplied by the query program stored in the support computer and (b_1,b_2) are the values output from the disc. In both cases, since instructions in a fragment are executed sequentially without interrupts, the buffer will store these in the same order to generate a homomorphic query fragment automatically. We mean, by "homomophic" that the generated query has the same form as the original. For instance, suppose the input query was: (1) in file b_1 mark all nodes having arc $(a,(b_1,b_2))$, (2) follow all arcs incident out of them whose name is c, (3) follow all arcs incident out of them whose name is d, (4) follow all arcs incident out of them. Suppose an arc $(c(e_1,e_2))$ emerged from file b_1 to file e_1 after instruction 2 was executed. The resulting query would be generated for file e_1: (1) mark node e_2, (2) follow all arcs incident out of them whose name is d, (3) follow all arcs incident out of them. The resulting query has the same form, and should be able to be mechanically generated from the original query. A query is then completed and the semantic page is completely output when all the fragments it generates are completely executed. This can be determined by the support computer. A similar mechanism will be used for updating, deleting or inserting a semantic page.

V. Date Integrity Across Files

Assuming no deletions or updates are allowed, the discussion of section III makes semantic paging very attractive on single file (10^9 bit) data bases, and the discussion of section IV makes it plausible on multiple file (10^{12} bit) data bases if data can be clustered on files so as to prevent excessive query fragmentation. However, when deletions or modifications are allowed, they require attention to data integrity. The two problems are consistency of code words and file locking.

It should be noted that the code word for a node is its logical address, such as code word (6,3) represents the third record from the top of the sixth file. This technique assigns code words compactly to records, as opposed to hash coding used in LEECH [6], to insure that the RAM's can be kept small. However, the logical address is not its physical (ie segment) address, since files can overlap segment boundaries in a CASSM type system. This means that words can be added to or deleted from records and automatic hardware garbage collection will move words between segments to adjust the file to accommodate these modifications without changing the code words, as discussed by De Martinis [7]. Nevertheless a whole records may not be delected or inserted, except at the end of a file, because the logical address of all records below it in that file would be decremented or incremented, and these addresses may be present on other files throughout the data base. Therefore, if a record is deleted, we delete all words in it but do not delete the record itself. This changes the record to a null record. If a record is to be inserted, it is inserted into a null record, or if none exist, it is inserted at the end of the file. When files are just inserted or deleted, this technique will preserve the code words which are the linkages between files. However, if it is desirable to move a node (record) from one file to another file such as when the former overflows, each pointer to that node requires a back pointer stored in that node so that when it is moved, all references to it can be adjusted.

Resource locking is required to allow updating or modification. If the data base is static, permitting no modifications, there is no need to lock read-only resources. However, if part of the data base is modifiable and modification requires unique control over the part being modified, we must prevent deadlock where two queries Q_1 and Q_2 modify two parts P_1 and P_2 and Q_1 has P_1 but needs P_2 while Q_2 has P_2 and needs P_1. Moreover, the response to a given query ought to correspond to some complete, consistent, instance of the data base. That is, if Q_1 modifies the data base and Q_2 is outputting data, the data Q_2 outputs should either be the data before Q_1 modified it or after Q_1 modified it, but not part of the data before Q_1 modified it and part of the data after Q_1 modified it [9]. Both problems require locking part of the data. However, we hope to lock the smallest part of the data base to prevent strangling the entire system by locking out a large part of it. Therefore, individual records (nodes) should be locked. Lock bits can be stored on the disc with the node in some field. An intelligent secondary memory greatly eases the problem of locking at the record level.

Some locking strategies require knowledge of all the required resources to insure freedom from deadlock. This is impractical in this technique because query fragments requiring further resources may be discovered after some files are searched. Therefore we need a strategy that does not depend on resource identification numbers. Stucki et. al. [9] have just provided us with a preprint of their paper that is presented at this workshop, and their scheme has the desired property because locking depends on the unique I.D. of the query itself. For outputting pages, we would output semantic pages as

their first phase is locking resources, but if the first phase fails to complete we would discard all the inputs and start the query over. The second phase would not do anything and the third phase would release them. Note that, if no locking problem exists, a semantic page is output quickly. For modifying pages, we would lock them in phase 1, modify them in phase 2 and output them in phase 3. This appears to be a feasible solution to the deadlock problem and the consistency problem for semantic paging across multiple files.

VI. Conclusions

This paper fits into two interesting themes, which are discussed in other papers at the same workshop. It provides a secondary memory for Artificial Intelligence Systems such as the LISP processor to be discussed by Williams [10]. As such, it may expand the size of data bases treated by AI programs to 10^9 or 10^{12} bits. This should lead to significant new uses for both DBM systems and AI systems. Moreover, it fits into Korfhage's model for distributed systems [11]. The homomorphic query fragments presented here are hardware generated "windows" that visit his "data base processors". This may unlock many of the doors to efficient distributed data base processing.

References

[1] Lipovski, G.J., "Architectural Features of CASSM: A Context Addressed Segment Sequential Memory", pp. 31-38, Proc 5th Symp. Comp. Arch., April, 1978.

[2] Ozkarahan, E.A., Schuster, S.A., and Smith, K.C., "A Data Base Processor" TR CSRG-43, University of Toronto, Nov., 1974.

[3] Savitt, D.A., Love, H.H. and Troop, R.E., "Association Storing Processor Study," Defense Documentation Center, AD 488438, June, 1966.

[4] Bush, J., Lipovski, G.J., Su, S.Y.W., Watson, J.K., and Ackerman, S.J., "Some Implementations of Segment Sequential Functions" 3rd Am. Symp. Comp. Arch., Clearwater, Fla. January 1976, pp. 178-185.

[5] Lipovski, G.J., "On Imaginary Fields, Token Transfers, and Floating Codes in Intelligent Secondary Memories" Proc. 3rd Workshop on Computer Architecture for Non-Numeric Processing, Computer Architecture News, Vol. 6 No. 2, May 1977, pp. 17-22

[6] McGregor, D.R., Thompson, R.G., and Dawson, W.N., "High Performance Hardware for Data Base Systems" Systems for Large Data Bases (Lockemann and Newhold eds.) North Holland, Amsterdam, 1976, pp. 103-116.

[7] De Martinis, M., Lipovski, G.J., Watson, J.K., and Su, S.Y.W., "A Self-Managing Secondary Memory" Proc 3rd Ann. Symp. Comp. Arch., pp. 186-194, January, 1976.

[8] Chandi, K.M., Private Communications

[9] Stucki, M.J., Cox, J.R., Roman, G.C., and Turcu, P.N, "Coordinating Concurrent Access in a Distributed Database Architecture", Proc. 4th Workshop on Computer Architecture for Non-Numeric Processing (these proceedings) 1978.

[10] Williams, R., "A Multiprocessing System for the Direct Execution of LISP", Proc 4th Workshop on Computer Architecture for Non-Numeric Processing (these proceedings) 1978.

[11] Korfhage, R.R., Day, W.H.E., Beck, L., and Appelbe, W.F., "Data Physics--An Unorthodox View of Data and its Implications in Data Processors", Proc. 4th workship on Computer Architecture for Non-Numeric Processing (these proceedings) 1978.