Efficient Processing of one and two dimensional Proximity Queries in Associative Memory *

K. L. Liu[1] G.J. Lipovski[2] C. Yu[3]
Naphtali Rishe[4]
1, 3: Department of EECS, University of Illinois at Chicago
Chicago, IL 60607
Tel : (312)-996-2318; FAX : (312)-413-0024
kliu, yu@dbis.eecs.uic.edu
2: Department of Electrical and Computer Engineering,
University of Texas at Austin
4: School of Computer Science, Florida International University
2, 3: members of Linden Technology

November 8, 1996

Abstract

Proximity queries that involve multiple object types are very common. In this paper, we present a parallel algorithm for answering proximity queries of one kind over object instances that lie in a one-dimensional metric space. The algorithm exploits a specialized hardware, the Dynamic Associative Access Memory chip. In most proximity queries of this kind, the number of object types is less than or equal to three and the distance d, within which object instances are required to locate to satisfy a given proximity condition, is small ($\lceil d/80 \rceil = 1$). The execution time for such queries is linearly proportional to the number of object types and is independent of the size of the database. This allows numerous concurrent users to be serviced. The algorithm is extended to process 2-dimensional proximity queries efficiently.

*This research is supported in part by NASA under NAGW-4080 and ARO under BMDO grant DAAH04-0024.

1 Introduction

There is a wide variety of proximity queries. We state below several common proximity queries.

- **Query I** : *(Bounded distance search)* Given a real number d and n points in a metric space, find all points that are at a distance less than or equal to d from a given query point.
- **Query II** : *(Fixed radius k-nearest-neighbor query)* Given a real number d, a positive integer k and n points in a metric space, find all groups of k points such that each point in a group is at a distance less than or equal to d from any other point in the same group.
- **Query III** : *(Multi-object type proximity query)* Given a real number d and n object types, instances of which are located in a metric space, find all groups, each containing an instance of each object type, such that each instance in a group is at a distance less than or equal to d from any other instance in the same group.

Below we give several examples of the above proximity queries.

(i) *Examples of Query I and Query II* : Assume that we have a database containing, among other information, the locations of villages in the countryside. The manager of a circus estimates that he would have enough audience for a show if there are k villages within a distance of d miles from the location of the show. To find out whether a site is suitable for holding a circus show, the manager would like to have an answer for the following instance of Query I : find the number of villages within d miles from the site. An instance of Query II is : find all groups of k villages such that each village in a group is situated within 2d miles from the other villages in the same group. An answer to this query would help the manager to select a location for a show. Information of this kind is also of value to the government to provide various kinds of services to the villagers.

(ii) *A one-dimensional example of Query III* : The keywords of a text document can be considered as object types with each occurrence of a keyword as an instance of an object type. The location of an instance of a keyword is the position of that instance in the text document, with the first word of the document starting at position 1. Thus, the text document can be viewed as a one-dimensional metric subspace with each instance of a word occupying an integer position in the subspace. Usually, in this type of applications, only the keywords are retained. For a text document, a common proximity query is : find all the locations of a phrase which consists of a set of keywords. In many situations, the keywords which make up a phrase are very close to each other, usually within 3 words apart. We can simulate the recognition of phrases by posing this kind of query to an IR (information retrieval) system [15]. Very often, we are interested in a variant of this query : determine for a document whether it has an occurrence of each keyword in a given set of keywords such that each occurrence of a keyword is within a distance of d words from the other occurrences. In other situations, the keywords we are interested in do not have to be very close to each other. For example, when we are interested in retreiving only those sections in a long text that are relevant to our needs [14], we may require the keywords to be within, say, a paragraph.

(iii) *A two-dimensional example of Query III* : A tourist in a city may wish to have lunch in a restaurant located near the places he would like to visit. Suppose he wants to visit an art museum and then see a movie after lunch. He would be interested in knowing the answer to the query : find the locations of a restaurant, an art museum and a theatre such that each is within d miles from the other.

Proximities queries of the forms as Query I or Query II have been treated extensively (see for example in [2, 3, 4, 5, 11]). Some algorithms for answering queries of type Query III were given in [1, 10]. In this paper, we concentrate on 1 and 2-dimensional queries of query type III. Specifically, we present an efficient algorithm for answering one dimensional example of Query III using associative memory. An important feature of our algorithm is that its running time is independent of the size of the database, which is not possible with any non-parallel algorithm. For small distance limit between object instances d, the running time of our algorithm is only linearly proportional to the number of object types n. A typical such query with $n \leq 3$ and $d \leq 80$ can be answered within 1.2 msec. This allows about 50,000 concurrent users, assuming that each user submits such a proximity query every minute. The speed of answering such queries is very important in applications involving numerous users. As the number of concurrent users in the Internet increases rapidly, it is critical that each query be answered efficiently so that there is no delay. Information retrieval systems utilizing main frame computers have been in service for users in the legal profession. It is known that unacceptable delays or even crashes occur whenever the number of users is very large. It is for these purposes that the specialized hardware and the algorithm given in this paper are being designed and implemented. The algorithm for answering 1-dimensional proximity queries is extended to the 2-dimensional case. It is found that such a typical 2-dimensional queries can be answered within 8.3 msec. Therefore, 7,200 concurrent users can be supported.

The rest of this paper is organized as follows. In section 2, we give a brief description of the architecture of a dynamic associative access memory chip. In section 3, we consider the one-dimensional version of Query III. We show how information is represented using the specialized hardware for the processing of a typical proximity query for text searching and present an algorithm to answer instances of such queries that have small distance limits between object instances. We also give a modified algorithm which can handle query instances having large distance limits. In section 4, we analyse the running times of these two algorithms. In section 5, the 1-dimensional algorithm is modified to be applicable to the 2-dimensional proximity queries. Time complexity analysis is also provided. Concluding remarks are given in section 6.

2 Architecture of a DAAM chip

A Dynamic Associative Access Memory (DAAM) chip [8] is modified from a dynamic random access memory (DRAM) chip. The concepts and the design of the DAAM chip is treated in [7, 9]. Other papers on DRAMs and associative memories include [12, 13, 18]. In this section, we give an overview of the architecture of the DAAM chip.

A DAAM chip is a dynamic random access memory chip which has been modified so that it can perform certain basic logical and arithmetical functions. In this paper we give, as an example, a 4-mega bit chip. It can be visualized as an array with 1024 rows and 4096 columns, with each entry corresponding to one bit (please refer to figure 1, p.7). Each row can be considered to be a processor, executing in parallel with other processors. The columns, for the sake of processing queries, can be considered as being partitioned into two sets. The first set contains $80 \times 6 \times 8$ bits and the second set contains the remaining 256 bits. The first set with all rows forms a region which is used to contain the data to be processed. This is called the data area. The second region formed from the second set with all rows is reserved to contain certain auxiliary data and working space to permit efficient answering of queries. This is called the working area. The bits in a row of the data set are organized as atoms. Each atom occupies 6 bytes. That is, each row contains 80 atoms. An instance of an object type under investigation is represented by an atom. Typical information stored in an atom are an identifier for an object type, the location of an instance of the object type and, possibly, its associated weight/significance. The kind of auxiliary data stored in the working area largely depends on the type of queries being processed. For example, if objects being represented by the atoms are words in a text document, the auxiliary data may include a bit indicating the end of a document and a count of the number of words, including non-content words, which are logically stored in that row. If more than one DAAM chip is needed, they are connected in such a way that they can logically be considered as a single long chip.

The basic operations with the DAAM chip include :

(i) *Comparison* : For example, compare if an identifier for an object type is equal to a set of identifiers stored in all rows under a given set of columns.

(ii) *Shift* : The set of numbers stored in all rows under a given set of columns are shifted to another set of columns. Numbers from one row can be shifted up or down to the next row, with all rows executing in parallel. The shifts can also be applied only to those rows which satisfy a comparison operation.

(iii) *Arithmetical and logical operations* : Add, subtract, multiply or divide a number to or from a set of numbers stored in all rows under a given set of columns. The arithmetic operations can also be applied on two set of numbers which are stored under two specified set of columns. Logical operations using *"and"*, *"or"* and *"not"* are permitted.

It is easy to see that the design is applicable to a chip of size 16 mega-bit, 64 mega-bit etc.

3 A parallel algorithm

In this section, we present a parallel algorithm that aims to answer queries of the same form as the one-dimensional version of Query III. For simplicity of illustration, we choose to describe our algorithm in terms of the following typical proximity query in text searching :

> Given a set of text documents and a set of n keywords $W_1, \ldots, W_n$, find all locations in each of the documents at which the n keywords are within d words of each other, i.e. $|loc(W_i) - loc(W_j)| \leq$ d for $1 \leq$ i, j $\leq$ n, where $loc(W_i)$ is the location of keyword W_i.

Each of these text documents is stored within a number of consecutive logical rows. The number of content words in each logical row is 80, with the possible exception of the last logical row of a document which may have fewer content words. Each content word is represented by an atom of a row in the data area of the DAAM chip. We use as many DAAM chips as necessary to store the entire set of documents. (Alternatively, a few DAAM chips with data pumped in from disks can be utilized to answer the query. We will not discuss this alternative in this paper.) An atom has two components : (*term#*, *location*), where *term#* is an identifier for the content word being represented and *location* is the position of the word from the beginning of the logical row (taking into account non-content words). Thus, if the location of a term is i, it is the i-th word from the beginning of that row. Note that two consecutive atoms in a row may have *location* values differ by more than one if there are non-content words lying between the two words which are represented by the atoms.

As each document is divided into a number of logical rows, the keyword instances of a group satisfying the proximity condition may not be located in the same row. In order to identify such groups, our approach requires that the locations of certain keyword instances present in other rows be known to each row i. These keyword instances are those that are located at distances of not more than d from the first word of row i in the document. We first present our algorithm for the case when the distance limit d is small, by which we mean that all the logical rows, except possibly the last row of a document (which may contain just a few words), have lengths greater than or equal to d. For this case, all the keyword information each row needs only pertains to those instances located in the previous row. Then, we will give the modifications needed to handle the other case when d is large, i.e. d is greater than the length of some logical row r and row r is not the last row. For ease of presentation, the algorithms presented are for a single DAAM chip. They can be generalized to handle the case when multiple DAAM chips are being used.

Case (a) : distance limit d is small

Our algorithm can be thought of as consisting of two phases : an information gathering and shifting phase and a phase for finding groups of instances satisfying the proximity condition. In the first phase, each processor searches for instances of keywords that are at distances of not more than d words away from the beginning of the next row and then transfers the keyword information obtained to the next row. In the second phase, each processor scans the atoms from left to right looking for groups of instances satisfying the proximity condition. In the working area of each row, we maintain two arrays *loc*[1...n], *flag*[1...n], a *MODIFY* bit and a *wordCOUNT* field. Each entry of *loc* is one byte in length and *flag* is a bit vector. *flag*[i] is set to 1 if an instance of keyword W_i is found; in that case, its location is placed in *loc*[i]. The *wordCOUNT* field in each row r stores the number of words, including non-content words, in the logical row r. This is done before the algorithm is executed and is not modified by the algorithm. All the bits of *flag* and the *MODIFY* bit are initialized to zero. The *MODIFY* bit is used only in the second phase. When the *MODIFY* bit in a row is 1, some keyword W_i has been found (in the second phase) in that row; it is reset to 0 whenever a group satisfying the proximity condition is reported by that row. The layout of the DAAM chip is depicted in figure 1.

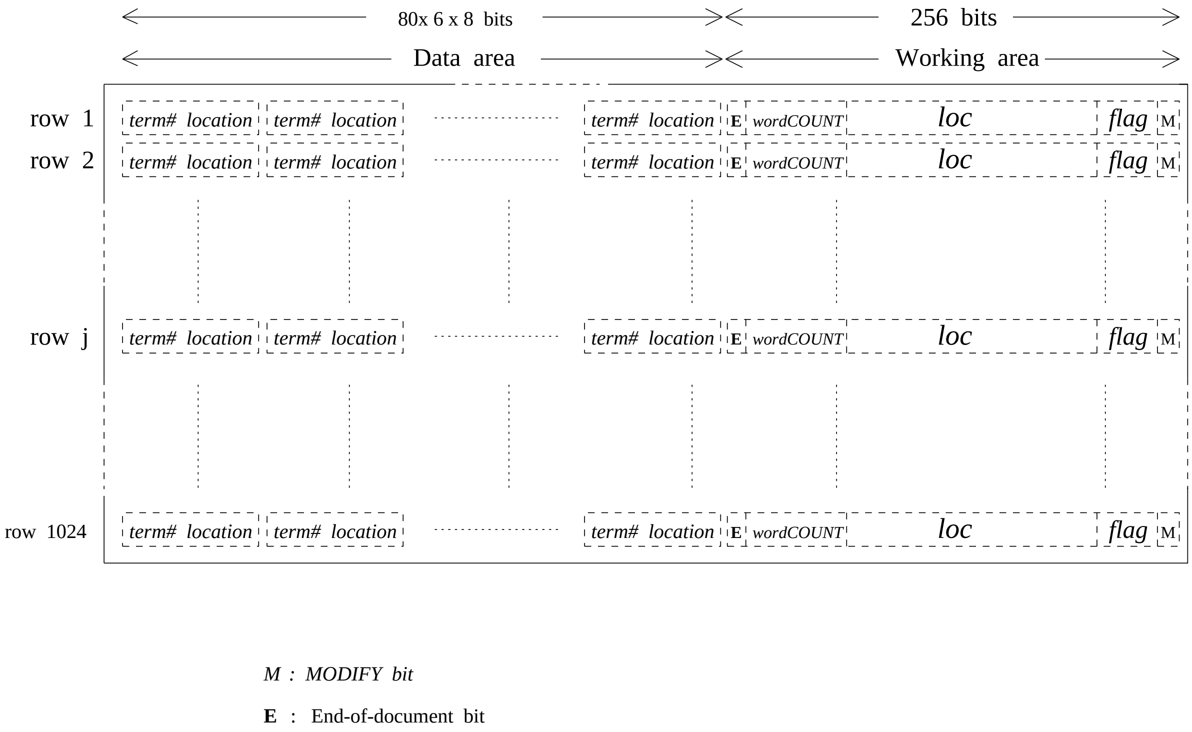


Figure 1

The following is our algorithm in pseudocode. The first two arguments d and n are respectively the distance limit and the number of keywords specified in the query. The third argument *TERM* is the set of keywords in the query and is represented as an array, the k-th entry of which contains the identifier for keyword W_k. The fourth argument m is the maximum of (81-d) and 1. In each row, the m-th atom is the first atom in that row that could represent a keyword instance lying at a distance of not more than d words from the beginning of the next row. atom[i] and row[j] denote respectively the i-th atom of a row and the j-th row.

```
PROX_SEARCH_I(d, n, TERM, m)
/* Steps 1-3 are for information gathering and shifting; step 4 is for finding and reporting groups
of instances satisfying the proximity condition. */

1. for each processor j, in parallel                    /* Initialization */
       do   { MODIFY = 0;
              for  k = 1 to n
                  flag[k] = 0;
            }

2. for each processor j, in parallel
       do   for  i = m to 80
                for  k = 1 to n
                     if (atom[i].term# == TERM[k])
                          { flag[k] = 1;
                            loc[k] = atom[i].location;
                          }

3. /* shift potentially useful information to the next row */
   for each processor j, in parallel
     do   { for  i = 1 to n
                 if (flag[i] == 1)
                     if (wordCOUNT - loc[i] ≤ d)
```

```
            row[j+1].loc[i] = loc[i] - wordCOUNT;
          else    flag[i] = 0;

      for  i = 1 to n
          row[j+1].flag[i] = row[j].flag[i];
    }

4. for each processor j, in parallel          /* check for proximity condition */
   do   for  i = 1 to 80
        { for  k = 1 to n
             if (atom[i].term# == TERM[k])
                     { flag[k] = 1;
                       loc[k] = atom[i].location;
                       MODIFY = 1;
                     }
          if (flag == "11..1" and MODIFY == 1)
               { for r = 1 to n
                    if (atom[i].location - loc[r] > d)
                              flag[r] = 0;
                 if (flag == "11..1")
                    { MODIFY = 0;
                      report (TERM[t], loc[t]), t = 1, ..., n;
                    }
               }
        }
```

The gathering of keyword information begins in step 2 of algorithm *PROX_SEARCH_I*. Each processor scans, from left to right, from the first atom that could contain useful keyword information for the next row. This atom is the m-th one, where m, given by the fourth argument, is the maximum of (81-d) and 1. Whenever an instance of W_i is found, its location is copied to *loc*[i] and the bit *flag*[i] is set to 1. Then, in step 3, for each row, we shift the needed keyword information to the next row as follows. For each row, if *flag*[i] is 1 ($1 \leq i \leq n$), indicating an instance of W_i has been found, we subtract the number of words in the logical row, stored in the *wordCOUNT* field, from *loc*[i], which contains the location of the rightmost instance of W_i if the row contains multiple instances of W_i. This is a negative number and this operation changes the location reference of the instance from the beginning of the row to the beginning of the next row. If this value indicates that this instance of W_i is at a distance of not more than d words away from the beginning of the next row, we transfer it to the *loc*[i] of the next row. Otherwise, we reset the bit *flag*[i] to 0. Before proceeding to the second phase, the contents of *flag* of each row are propagated to *flag* of the next row. Having obtained all the required information, each processor scans its row looking for groups of instances that satisfy the proximity condition (step 4) starting from the first atom. Whenever an instance of W_i is found, in addition to copying its location to *loc*[i] and setting *flag*[i] to 1, we also set the *MODIFY* bit to 1. When the *MODIFY* bit and all the bits of *flag* are 1's, indicating that a group containing an instance of each given keyword has been located, a test for the satisfaction of the proximity condition is made. The proximity condition is checked by determining whether each instance in the group is at a distance of not more than d words from the rightmost instance, which is the one most recently found, of the group. It is easy to see that if this is the case, all the instances in the group are within d words from each other, i.e. they satisfy the proximity condition. During the test, if an instance of keyword W_i in the group is found to be located at a distance greater than d words from the rightmost instance in the group, then the bit *flag*[i] is reset to 0. If no bit of *flag* was reset to 0 during the test, we report the locations of these n instances and set the *MODIFY* bit to 0. The searching ends when each processor has scanned all the atoms in its row.

Note that when an instance of, say W_i, is found, we assign *flag*[i] a value 1 and reset it to 0 only after the instance is found to be at a distance of more than d words to the left of a keyword instance represented by the currently examined atom. Suppose there is a group of instances located between (L-d) and L satisfying the proximity condition. As none of these instances is more than d words from any other instances of this group, all the bits in *flag* and the *MODIFY* bit will be 1's after the processor finds the rightmost instance of this group and sets the corresponding bit in *flag* and the *MODIFY* bit to 1. The existence of the group will then be detected.

Case (b) : distance limit d is large

In algorithm *PROX_SEARCH_I*, we assume that all logical rows, except possibly the last row, have lengths greater than or equal to the distance limit d, which we expect to be the case for the majority of proximity queries. When some logical row r has length less than d and it is not the last row, instances from more than two logical rows may satisfy the proximity condition. In this case, information concerning the instances found in a row may need to be shifted to the next few rows and we need to modify step 3 of *PROX_SEARCH_I* so that enough information is propagated to each row. Our modification to step 3 consists of maintaining for each row another bit vector *tmpFG*[1...n] in the working area. *tmpFG*[i], i = 1, ..., n, of each row is initialized to 0. Its value is 1 if the location information of an instance of W_i has been shifted to the next row and 0 otherwise. That is, the contents of *tmpFG* of each row reflect what keyword information has been passed to the next row. Below, we give, in pseudocode, step 3', which is a replacement of step 3 in *PROX_SEARCH_I*.

3'. /* s is the length of the shortest logical row of a document, not including the last row; N_r is the number of logical rows of the longest document. */

$\mu = \min\{\lceil d/s \rceil, N_r\}$;

/* μ is the maximum number of information transfers from each row to the next; the algorithm can be modified slightly to ensure that information is never shifted across documents, but the details are not given here. */

```
for each processor j, in parallel
    do μ times    /* The statment to be executed μ times is referred to as statement (+) */
        { for i = 1 to n
            if (flag[i] == 1 and tmpFG[i] == 0)
                { row[j+1].loc[i] = loc[i] - wordCOUNT;
                  tmpFG[i] = 1;
                }
```

```
for i = 1 to n
    row[j+1].flag[i] = tmpFG[i];
}
```

In each execution of the body of the loop of step 3', location information is passed from each row to the next. We refer below to one such information passing as a transfer. In step 3', the body of the loop, *statement (+)*, is repeated μ times, where μ is the minimum of $\lceil d/s \rceil$ and N_r, s being the length of the shortest logical row of the document, not including the last row, and N_r being the number of logical rows of the longest document. In other words, each processor performs μ transfers in step 3'. Note that although a row needs keyword information from preceding rows, the number of preceding rows providing potentially useful information to the row is bounded by μ. During each of the μ transfers, for each row, if *flag*[i] equals 1 and *tmpFG*[i] equals 0, i = 1, ..., n, we assign the difference *loc*[i] - *wordCOUNT* (changing the location reference to the beginning of next row) to *loc*[i] of the next row and set *tmpFG*[i] to 1. That is, the location information of an instance of W_i is passed to the next row, if the row has the location information and has never passed such information to the next row in any previous transfer. As a result of these two conditions for the propagation of location information, once a row has received the location information of an instance of a keyword W_i from a preceding row, it will never receive such information regarding keyword W_i in any later transfer (since *tmpFG*[i] of the preceding row is now 1 and will never be reset to 0 in step 3'). To see how the location information is shifted, suppose there are multiple instances of W_i present in the rows preceding some row r and each of these instances is within a distance of d words from the beginning of row r. Let the instance of W_i, I_i, closest to row r occur in row r', $r' < r$. When *statement (+)* in step 3' is executed the first time, the condition $COND$: (*flag*[i] == 1 and *tmpFG*[i] == 0) is satisfied by row r'. As a result, the location of I_i is passed to row (r'+1) from row r' and *flag*[i] in row (r'+1) is set to 1. If (r'+1) is r, then row r has now received the location information of I_i from row r'. Otherwise, as row r' is the closest row to row r having an instance of W_i, *tmpFG*[i] in row (r'+1) remains 0 after the first execution of *statement (+)*. When *statement (+)* is executed the second time on row (r'+1), the condition $COND$ is satisfied, resulting in the location of I_i, which has been received from row r', passed to row (r'+2). In addition, *flag*[i] in row (r'+2) is set to 1. Subsequent execution of *statement (+)* on row (r'+2), row (r'+3) etc.

causes the location of I_i to pass to row (r'+3), row (r'+4) etc.. Thus, if row r is no more than μ rows away from row r', then row r will receive the location of I_i from row r'.

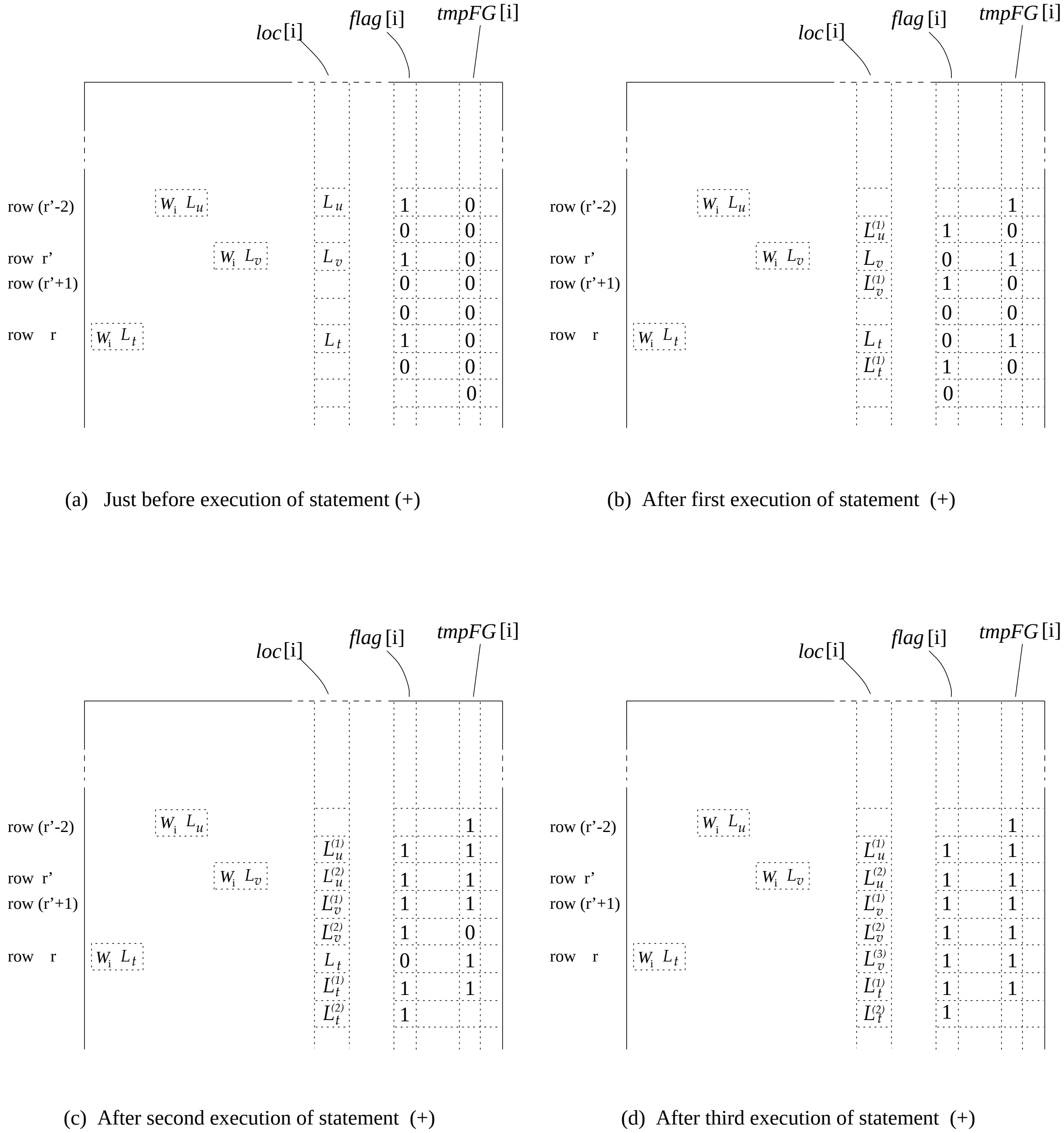
The algorithm will not permit the location of an instance of W_i from row r", $r" < r' < r$, to pass to row r for the following reason. When *statement (+)* is executed on row r' the first time, *tmpFG*[i] in row r' is set to 1. Since this variable is not reset to zero, the condition *COND* is not satisfied by row r' in the second and later executions of *statement (+)*. Thus, no location information on W_i can pass from row r" to rows beyond r'. Figure 2 illustrates how *loc*[i], *flag*[i] and *tmpFG*[i] change for 8 rows for 3 executions of *statement (+)*. Note that the location information of the instance of W_i in row (r'-2) is not shifted from row r' to row (r'+1) in the third execution.

Observe that if row r has an occurrence of W_i, then its location will be overwritten by the location of the instance I_i of W_i in row r'. However, during step 4, the location information from row r will be recovered when the row is scanned.

In step 3', we do not check whether the distance of a keyword instance, whose location is to be shifted, is less than or equal to d from the beginning of the row requiring the keyword. After the execution of step 3', a row may receive keyword information not useful to it. Note that in step 4, for each row r, we check a group of n instances for the satisfaction of the proximity condition only when the *MODIFY* bit is 1, that is, this group must contain an instance of a required keyword which is originally present in logical row r. In other words, the redundant keyword information which a row may receive does not lead to the row reporting a group consisting entirely of instances present in preceding rows. Any such information will be discovered and discarded when the row checks for the proximity condition in step 4.

4 Analysis

We assume that the *term#* component of each atom has length 4 bytes. Let C_1 be the cost of an arithmetical comparison of a one-byte field with a constant, C_2 be the cost of a logical comparison of a four-byte field with a constant, C_3 the cost of copying the contents of one byte to another byte of the same row, C_4 the cost of shifting the contents of one byte up or down one row, C_5 the cost of an arithmetical operation, C_6 the cost of a bit comparison, C_7 the cost of setting or resetting a bit, C_8 the cost of shifting a bit up or down one row, and C_9 the



(a) Just before execution of statement (+)

(b) After first execution of statement (+)

(c) After second execution of statement (+)

(d) After third execution of statement (+)

* When the location of an instance is propagated to the next row, its location reference is changed to the beginning of the next row. $L_v^{(i)}$ (i = 1, 2, 3), for example, are the location values of the same instance present in row r'.

Figure 2

cost of reporting a group of instances satisfying the proximity condition. When the distance limit d is small, keyword information needs to be shifted from each row to the next only once. This information shifting, by step 3, takes at most n(C_1 + C_4 + C_5 + C_6 + C_8) time. The initialization takes (n+1)C_7 time. The worst case running time of step 2 is 80n(C_2 + C_3 + C_7), while that of the last step is 80(n$\times C_1$ + n$\times C_2$ + n$\times C_3$ + n$\times C_5$ + (2n+1)C_6 + (2n+1C_7 + C_9). Hence, for a given set of n keywords, the worst case running time of *PROX_SEARCH_I* is estimated to be 81n$\times C_1$ + 160n$\times C_2$ + 160n$\times C_3$ + n$\times C4$ + 81n$\times C_5$ + (161n + 80)C_6 + (241n + 81)C_7 + n$\times C_8$ + 80$\times C_9$, linearly proportional to n and independent of the size of the database. With current technology, the refresh cycle of a DAAM chip is about 60 nanoseconds. The costs C_i, i = 1, ..., 9, can be determined. If a query has 3 keywords (i.e. n = 3), the estimated time to answer it is about 1185.52 μsec. In other words, about 843 such queries can be processed per second. If each user submits one such proximity query every minute, the DAAM chip can serve about 50,000 concurrent users.

For large distance limit d, keyword information needs to be shifted from each row to the next more than once and *PROX_SEARCH_II* needs to be used. The running time of step 3' of *PROX_SEARCH_II* is μ(2n$\times C_6$ + n($C_4 + C_5 + C_7 + C_8$)). Since *PROX_SEARCH_II* uses an additional field *tmpFG*, the initialization time is increased to (2n+1)C_7. Thus, the worse case running time for *PROX_SEARCH_II* is estimated to be 80n$\times C_1$ + 160n$\times C_2$ + 160n$\times C_3$ + 80n$\times C_5$ + 80(2n+1)C_6 + (242n + 81)C_7 + 80$\times C_9$ + μ(2n$\times C_6$ + n($C_4 + C_5 + C_7 + C_8$)). Suppose such a query is to find a set of 10 content words within a paragraph and each paragraph has no more than 500 content words. Then, $\mu \leq \lceil 500/s \rceil \leq \lceil 500/80 \rceil = 7$. We estimate that the execution time for this query is about 4,050 μsec.

Although our algorithm will find all the locations where there are groups of instances satisfying the proximity condition, it may not identify all the different groups that fall within the same or nearly the same locations. As the atoms are being scanned from left to right, when an instance of W_i is found, its location is copied to *loc*[i] overwriting any location information originally there. When the processor finds that the proximity condition is satisfied, it reports only the most recently found instance of each keyword. Suppose that there are two instances of a keyword W_j that are very close to one another and that each of them together with the same other (n-1) instances forms a distinct group satisfying the proximity condition. If both instances are not the rightmost one in their respective groups, when either of them is being scanned, the processor has not yet found all the n instances and therefore cannot detect the two groups. But then after the second instance of W_j has been processed, its location information overwrites that of the first one. Thus, the processor will not be able to detect the group containing the first instance of W_j. However, for situations in which the groups of object instances do not overlap, our algorithm will find all the groups that satisfy the given proximity condition. For example, suppose we are interested in finding the locations of a given phrase, say "database management". It would be very unusual to find two or more occurrences of "database" or two or more occurrences of "management" satisfying a proximity condition of no more than 3 words apart, while having an occurrence of "database" and an occurrence of "management" satisfying the proximity condition. Thus, for most applications of the type under consideration, the algorithm given above is sufficient.

5 Extension to two-dimensional proximity query

A visitor to a city may want to have lunch at an Italian restaurant. Afterwards, he may want to visit a science museum and then go to a concert. For convenience (or to save time), he would like all these places (an Italian restaurant, a science museum and a concert hall) not very far apart. We have indicated in the introduction section that the query for which he wishes to have an answer is an instance of a two-dimensional version of Query III. An algorithm for answering this kind of two-dimensional query is somewhat complicated. Rather than providing such an algorithm, we give one for answering a modified version of the query that requires only that the instances be within a rectangular region. The query we shall concern ourselves with is as follows :

> Given two non-negative real numbers d_x and d_y and n object types O_t, $1 \leq t \leq n$, find the locations of instances such that for each object type O_t, 1 ≤ t ≤ n, there is an instance I_t satisfying : for all I_i, I_j, $|X_i - X_j| \leq d_x$ and $|Y_i - Y_j| \leq d_y$, where (X_i, Y_i) and (X_j, Y_j) are the cartesian coordinates of I_i and I_j respectively.

Let $a_1, a_2, \ldots, a_p$ and $b_1, b_2, \ldots, b_q$ be respectively the distinct x-coordinates and y-coordinates, in ascending order, of the coordinates of all the object instances located in the area of interest. Note that the units of measurement for the x and y axes may not be the same. In a city where streets are laid out in the form of a grid, we may use, for example, a block as a unit of measurement. We assume that the units of measurement are chosen so that the differences $(a_{i+1} - a_i)$ and $(b_{j+1} - b_j)$ are at least 1 (i=1,..., p-1; j=1, ..., q-1). Each atom is associated with one location (a_r, b_s) (r = 1, ..., p; s = 1, ..., q). The atoms of the first $\lceil p/80 \rceil$ rows are associated with the locations having y-coordinate b_1. Associated with the i-th atom of the (k+1)-st row of this group of rows is $(a_{(i+80k)}, b_1)$, where i+80k $\leq p$ (i = 1, ..., 80; k = 0, ..., $\lceil p/80 \rceil$-1). The atoms of the u-th group of $\lceil p/80 \rceil$ rows are similarly associated with the p locations having y-coordinate b_u (u = 1, ..., q). Note that the choice of reference axes is arbitrary. In order to reduce the number of rows representing locations having the same y-coordinate, we choose the x and y axes so that $p \leq q$. As all the atoms in the same row are associated with locations having the same y-coordinate, for the storage of this common value, we maintain a field, called *ORDINATE*, for each row in the working area. Each atom has two components : *TYPE* and *X_COORD*. The *X_COORD* component contains the x-coordinate of the location associated with the atom. If there is an object instance at that location, the *TYPE* component of the atom indicates the type (say, for example, an Italian restaurant) of the instance; otherwise, the *TYPE* component is given a special value which indicates the absence of an instance of any object type of interest at that location. In addition to the *ORDINATE* field, in the working area, for each row, we maintain two arrays *loc*[1 ...n] and *flag*[1 ...n], a *MODIFY* bit and two fields *iTP* and *Y_COORD*. *loc*[i] is an ordered pair (x, y) for the storage of the coordinates of an instance of object type O_i (i = 1, ..., n). *flag* is a bit vector. Both *flag* and *MODIFY* serve the same purposes as before. *iTP* and *Y_COORD* are used respectively for the propagation of object type and y-coordinate information in the second stage of our algorithm. If p, the number of distinct x-coordinates, is greater than 80, our algorithm will have two stages. The first stage propagates location information. For this stage, the field *iTP*, which each row uses for the propagation of the object type information in the second stage, is used for the propagation of the x-coordinates, and the field *Y_COORD* for the propagation of y-coordinates. In figure 3 where a layout of the DAAM chip is shown, we assume that p is 161. The first entry in the working area in each row is the *ORDINATE* field and contains the y-coordinate of the locations associated with the atoms of that row. In the *X_COORD* component of each atom is the x-coordinate of the location associated with the atom. Consider a real life example. In downtown Chicago, the total number of restaurants, museums and theaters is about 1,200. The number of distinct x-coordinates is about 200 and the number of distinct y-coordinates is about

250. The number of rows having the same y-coordinate is $\lceil 200/80 \rceil$ = 3. Thus, about 250×3 = 750 rows are needed to represent the location information of the restaurants, museums and theaters in this area.

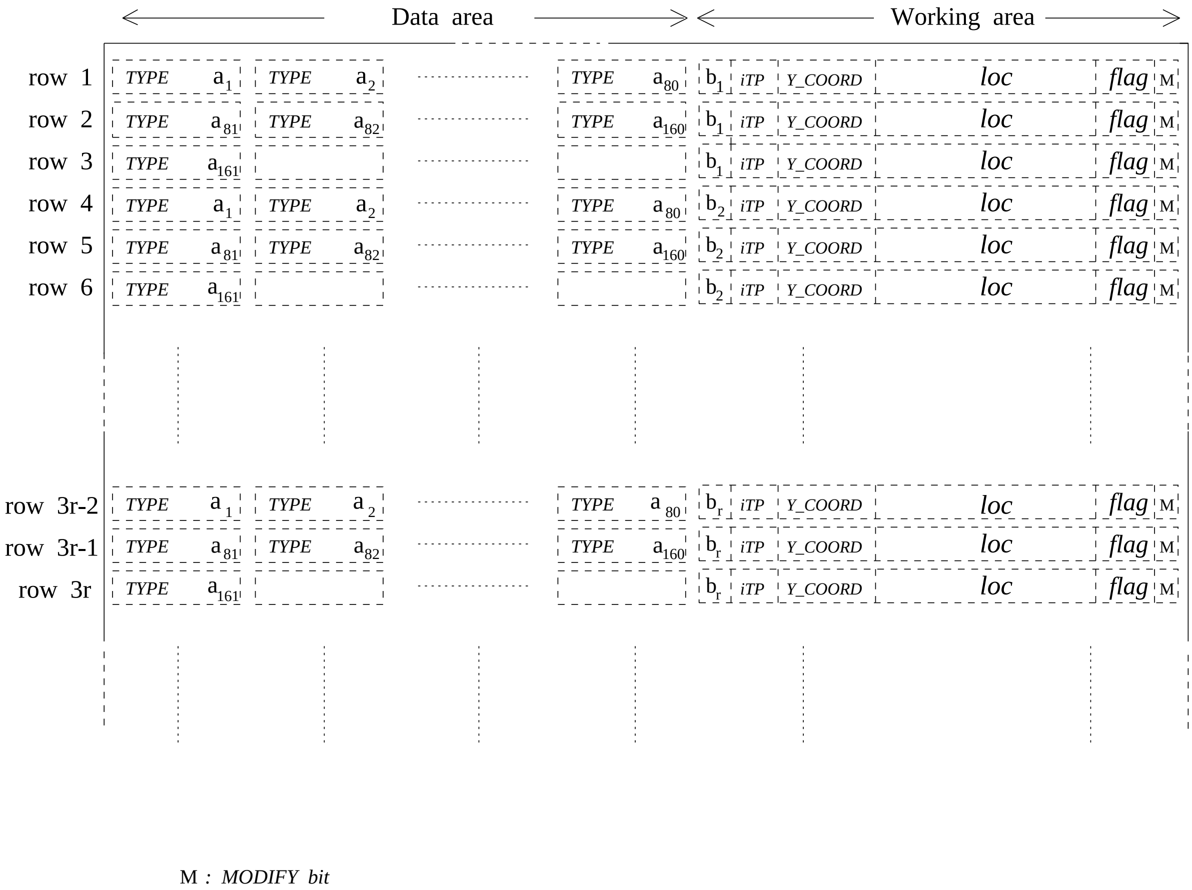


Figure 3

For simplicity of presentation, we assume that the value of d_x is small, i.e. the x-coordinates associated with the first atoms of any two adjacent rows differ by more than d_x. This is analogous to case (a) of section 3. The case for large d_x is analogous to case b in section 3 and will not be presented here. In the following discussion, if two locations (a, b) and (p, q) are such that 0 $\leq$ (a-p) $\leq$ d_x and 0 $\leq$ (b-q) $\leq$ d_y, we say that (a, b) *covers* (p, q).

The object instances of a group satisfying the proximity condition may be represented by atoms in different rows. Among these rows, we choose the processor of the row with the largest *ORDINATE* value to detect and report such a group. Because of this choice, the information of an instance found in a row needs to be propagated down to those rows containing at least an atom whose associated location *covers* the location of the instance. If $p > 80$, two or more rows of atoms are needed to represent those instances having the same y-coordinate. Let v be the number of such rows (i.e. $v = \lceil p/80 \rceil$). Consider a group satisfying the proximity condition. Suppose this group is composed of instances having x-coordinates less than or equal to a_{80N} as well as x-coordinates greater than a_{80N}, where N is an integer such that $0 < 80N < p$. As we assume d_x is small, those instances having x-coordinates greater than a_{80N} are represented by atoms close to the left ends of some rows of the DAAM chip while those having x-coordinates less than or equal to a_{80N} by atoms close to the right ends of some other rows. Because we scan the atoms of a row from left to right, some object instances of this group will be found first. By the time those instances (of the same group) represented by atoms near the right ends are discovered, it is possible that the location information of those object instances found early in the scan are overwritten and, as a result, this group is never discovered. We have encountered a similar problem in the one-dimensional case in section 3. In order to identify such groups, we use the same strategy as before. Before we start searching for groups satisfying the proximity condition, for each row r, information concerning those instances that are represented by atoms near the right end of the row is found and propagated down to those rows that will need the information. To gather the needed information, for each row r, we scan the atoms starting from the m-th one, where m is the maximum of 1 and (81 - $\lfloor d_x \rfloor$). Since we assume that the difference $a_{i+1} - a_i$ is at least 1 (i=1,..., p-1), this atom is the first atom of row r whose associated location may be *covered* by the location associated with the first atom of some row s. It is evident that if row r is not the last row in a group of v rows having the same $ORDINATE$ value, one such row s is row (r+1). The other rows that need to obtain instance information from row r are those rows (r+1 + v×N), where N is a positive integer such that the $ORDINATE$ value of row (r+1 + v×N) differs from that of row r by not more than d_y. To see this, it suffices to note that the corresponding atoms of row (r+1) and row (r+1+ v×N) have the same *X_COORD* value.

Our algorithm consists of two stages. The first stage is invoked if p is greater than 80. In this stage, we first scan the atoms of each row from left to right starting from the m-th one (m = max(1, 81-$\lfloor d_x \rfloor$)). Each time an atom has been scanned, if it is found to represent an instance of object type O_k, the bit *flag*[k] is set to 1, the contents of the atom's *X_COORD* component are copied to *loc*[k].x and the *ORDINATE* value to *loc*[k].y. This is repeated for the j-th atom,

$m \leq j \leq 80$, in each row. Then, for each row r, the gathered information is propagated down to those rows s ($s = r + 1 + v \times N$) satisfying the constraint on the y-coordinate. This was discussed in the last paragraph. Note that the location associated with the first atom of the first row of each group of v rows having the same $ORDINATE$ value lies on the leftmost boundary of the region of interest. No instances located to its left are represented in the DAAM chip. For these rows, any information propagated to them in the first stage is useless. We therefore set all the entries of $flag$ to 0 for these rows before going to the second stage. In the second stage, the processor of each row r scans the atoms from left to right starting from the first one. Immediately before each atom is being scanned, the field iTP is assigned a value 0. If an instance of object type O_k is found, in addition to copying its location information to loc[k] and setting $flag$[k] to 1 as in the first stage, we also set the $MODIFY$ bit to 1 and assign the value k to iTP. Unlike the first stage which propagates the instance information after all the atoms have been scanned, each time an atom of row r has been scanned, information represented by the atom is propagated. Since we assume d_x is small, the rows that may need the information from row r in this second stage are those rows s beneath it, where $s = r + v \times N$. As the corresponding atoms of rows r and $r + v \times N$ have the same X_COORD value, there is no need to propagate this value. The information that needs to propagated from each row r is the y-coordinate associated with the atom, which is given by the $ORDINATE$ field of row r and the atom's $TYPE$ value, which can now be obtained using the value in iFP. To avoid destroying the contents of the field $ORDINATE$, we use the field Y_COORD for the propagation of the y-coordinate. Thus, for each row, we first copy the $ORDINATE$ value to Y_COORD before shifting the contents of these two fields iTP and Y_COORD to the corresponding fields of the next row repeatedly. Whenever the information from each row r has shifted down a multiple of v rows to row s, we examine the iTP value propagated from row r. If its value is positive and equals to, say t, implying that the atom in row r represents an object instance I of type O_t, then the constraint on the y-coordinate, i.e. row[s].$ORDINATE$ - row[s].$Y\_COORD \leq d_y$, is checked. (Note that at this point, the value in Y_COORD of row s is the same as the value in $ORDINATE$ of row r.) If it is satisfied, then in row s, the location of this instance I (found in row r) is copied to $loc[t]$ if either $flag[t]$ is 0 or its x-coordinate is greater than $loc[t]$.x, i.e. the instance whose information is now propagated to row s is located to the right of the instance to which the information in $loc[t]$ pertains. Note that because of the constraint on the y-coordinate, the maximum number of rows that the information from each row needs to reach is $v \times \lfloor d_y \rfloor$. That is, the maximum number of downward shifting operation by each row is $v \times \lfloor d_y \rfloor$. Whenever all the entries of *flag* and the *MODIFY* bit are all 1's indicating that a new group containing an instance of each object type has been detected, we do the checking and, if the proximity condition is satisfied, report as before. Below we give only the psuedocode for the second stage.

```
/* Otype[1...n] is an array of object types. Otype[i] contains the i-th object type O_i, i = 1,
..., n.   */
for each processor j, in parallel
        do   for i = 1 to 80
               { iTP = 0;
                 for k = 1 to n           /* scanning the i-th atom */
                     if (atom[i].TYPE == Otype[k])
                        { MODIFY = 1;
                          flag[k] = 1;
                          loc[k].x = atom[i].X_COORD;
                          loc[k].y = ORDINATE;
                          iTP = k;                          }
                 CHECK(i);        /* checks the constraint on x-coordinate */

                   /* propagate the instance information represented by the i-th atom
                        to those rows that may require it */
                 Y_COORD = ORDINATE;

                 for s = 1 to ⌊d_y⌋
                   { for t = 1 to v          /* v = ⌈p/80⌉ */
                         { row[t+1].Y_COORD = row[t].Y_COORD;
                           row[t+1].iTP = row[t].iTP;
                         }
                     if (iTP > 0 and ORDINATE - Y_COORD ≤ d_y)
                       if (flag[iTP] == 0 or loc[iTP].x < atom[i].X_COORD)
```

```
{ MODIFY = 1;
  flag[iTP] = 1;
  loc[iTP].x = atom[i].X_COORD;
  loc[iTP].y = Y_COORD;
}
CHECK(i);
}
}
```

```
procedure CHECK(i)
  { if (flag == "11...1" and MODIFY == 1)
      { for r = 1 to n
          if (atom[i].X_COORD - loc[r].x > d_x)
            flag[r] = 0;

        if (flag == "11...1")
          { MODIFY = 0;
            report (Otype[i], loc[i]), i = 1, ..., n;
          }
      }
  }
```

The running time of our algorithm is dominated by the running time of the second stage. The worst case running time for the second stage is estimated to be $80[10+54n+\lfloor d_y \rfloor(50v+66+26n)]$ refresh cycles. For $v > 1$, the worst case running time is estimated to be $812 + 4384n + 23n \times min(80, d_x) + \lfloor d_y \rfloor(5280 + 4000v + 2137n + 50vn)$ refresh cycles. Consider the visitor example given at the beginning of this section. In a city, people tend to judge the distance along a street in terms of the number of shops while the distance in the direction perpendicular to the street in terms of the number of blocks. For an application to answer the visitor's query, we therefore use "a shop" as a unit of measurement for the x-axis and "a block" for the y-axis. We assume that the streets are laid out (roughly) in the form of a grid. Suppose the visitor does not want to walk more than 50 shops along a street and more than 5 blocks away. Then, the value for d_x is 50 and that for d_y is 5. Assume that the values for n and v are both 3. The worst case running time for the query is estimated to be 8.3 msec. If, on the average, each user submits a query every minute, the number of concurrent users that can be supported is about 7,200.

6 Conclusion

We have presented a simple yet efficient algorithm to answer one-dimensional queries using DAAM, a modified dynamic random access memory. To answer such a query having number of object types less than or equal to three and small distance limit, the execution time is estimated to be 1185 μsec. This permits the servicing of around 50,000 concurrent users, each submitting, on the average, one such query every minute. The same architecture can be used to find documents which are similar to a given query, using, for example, the COSINE function [15]. (The algorithm to compute the similarities can be shown to be very efficient, but is beyond the scope of this paper.) The algorithm to process 1-dimensional proximity queries is extended to handle 2-dimensional proximity queries. The execution time for a typical 2-dimensional proximity query is 8.3 msec, permitting 7,200 users to be serviced concurrently.

There are two advantages for using the DAAM chip. First, the cost of the DAAM chip is projected to be only slightly above the corresponding standard dynamic random access memory (DRAM). As the size of DRAM increases from 4 megabit to 16 megabit, 64 megabit etc., the power of the DAAM chip increases proportionally, as the number of processors also increases proportionally. The technology of modifying static memories, which are more expensive than dynamic memories, to associative memories has recently been demonstrated to be successful [6]. It is expected that the DAAM chip is more cost effective. Second, the traditional way to answer proximity queries in textual databases is by the use of inverted lists. It is known that the cost for maintaining such lists is very high [17], as the insertion of new documents causes changes for many such lists. In contrast, the cost of inputting new documents into the DAAM architecture is negligible, as all existing documents are not affected and there is no inverted list to maintain. The Connection Machine [16] is a well known parallel machine. One of its application areas is text searching. However, we are not aware of its performance on proximity searching. Therefore, we cannot give a comparison in this regard.

The DAAM chip can be very efficient in processing text queries, in 3D rendering in computer graphics and in MPEG II decoding. It is our hope that it will be fully utilized in other real life applications. The chip is being designed and will be in production in approximately nine months. The cost of a 4-mega bit DAAM chip is projected to be $30. For a database of one gigabytes, the number of 4-mega bit chips needed is approximately 2,000. Thus, the cost to contain one gigabytes of data is about $60,000. Although this is high for conventional machines, its performance is much faster. The cost of DAAM follows the standard DRAM memory. When the price of DRAM drops, we expect the cost of DAAM to drop proportionally.

References

[1] W.G. Aref, D. Barbara, S. Johnson and S. Mehrotra. *Efficient Processing of Proximity Queries for Large Databases.* IEEE Data Engineering pp.147-154. 1995.

[2] J.L. Bentley and J.H. Friedman. *Data structures for range searching.* ACM Comp. Surveys, 11(4) pp.397-408, Dec. 1979.

[3] L.S. Davis and N. Roussopoulos. *Approximate pattern matching in a pattern database system.* Info. Sys., 5(2) pp.107-119, 1980.

[4] D. Dobkin and R.J. Lipton. *Multidimensional searching problems.* SIAM J. Comp., 5(2) pp.181-186, June 1976.

[5] C. Faloutsos, M. Ranganathan, Y. Manolopoulous. *Fast Subsequence matching in Time-Series Database.* ACM SIGMOD 94.

[6] M. Gokhale, B. Holmes and K. Iobst. *Processing in Memory: The Terasys Massively Parallel PIM Array.* IEEE Computer, 1995.

[7] G.J. Lipovski *Dynamic Systolic Associative Memory Chip.* Proc. Application Specific Array Processors. Princeton, NJ. pp.481-492, 1990.

[8] G.J. Lipovski. *Dynamic Memory with Logic-in-Refresh.* Patent No. 4,989,180. Jan. 29, 1991.

[9] G.J. Lipovski. *A Four Megabit Dynamic Systolic Associative Memory Chip.* Journal of VLSI Signal Processing, 4. Boston: Kluwer Academic pp.37-51, 1992.

[10] U.Manber and R. Baeza-Yates. *An algorithm for string matching with a sequence of don't cares.* Info. Proc. Ltrs., 37(3) pp.133-136, Feb. 1991.

[11] F. Preparata and M. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, NY, 1985.

[12] I. Robinson. *Chameleon: A Pattern Matching Memory System.* Hewlett Packard Tech. Report HPL-SAL089-24, April 19, 1989.

[13] I. Robinson. *The Pattern Addressable Memory: Hardware for Associative Processing*, in (J.G. Delgado-Piras and W.R. Moore, eds.) VLSI for Artificial Intelligence. Boston: Kluwer Academic pp.119-129, 1989

[14] G. Salton, J. Allan and C. Buckley. *Approaches to Passage Retrieval in Full Text Information Systems.* ACM SIGIR 93, pp.49-56.

[15] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* New York: McGraw-Hill, 1983.

[16] C. Stanfill and B. Kahle. *Parallel Free-text Search on the Connection Machine Systems.* CACM, 1986.

[17] A. Tomasic, H. Garcia-Molina and K. Shaeus. *Incremental Updates of Inverted Lists for Text Document Retrieval.* ACM SIGMOD, 1994.

[18] J.P. Wade and G.S. Sodini. *A Ternary Content Addressable Search Engine.* IEEE Journal of Solid-State Circuits, vol. 24, pp.1003-1013, 1989.