ON THE USE OF BIT MAPS FOR MULTIPLE KEY RETRIEVAL

Oscar Vallarino
University of Toronto

Abstract: The traditional file structures used to support fast response to complex user queries have been based on the inverted list organization, using either the pointer or bit string representation. In this paper, the use of bit maps for executing multiple key searches is studied. Bit maps turn out to be less precise inverted lists where the inversion is kept for a quantization of attribute domains and the objects referenced are blocks of data records. The goal is to reduce the total number of I/O accesses required to execute a retrieval based on a Boolean qualification. An evaluation of the method is given for both storage space and expected retrieval time under simplified assumptions. Key Words and Phrases: Multiple key retrieval, inverted lists, bit strings, bit maps, Boolean queries, data base management. CR Categories: 3.70, 3.71, 3.73, 3.74, 4.33, 4.34

1. INTRODUCTION

One of the most desirable features to have in large data bases is the ability to perform fact retrieval by means of queries to the system. In this paper we consider the problem of executing Boolean queries with the goal of reducing the I/O time required to retrieve the records that satisfy the request. The motivation behind executing Boolean queries efficiently is that, besides being a user's language in themselves, they might be produced as a result of interpreting a higher level query language such as SQUARE [10].

Inverted lists have been the traditional file organization used to support fast response to unanticipated queries and they are well covered in the literature [3,5,14,15,16,17,20]. Two basic representations for inverted lists have been employed in data retrieval systems: the pointer list representation and the bit string representation. In the former, for each attribute values (index term) present in the file a list of pointers (accession numbers of addresses of records) is kept which identifies all record occurrences in the file which have that attribute value. In the latter, a bit string is kept which contains one bit for each record in the file; the nth bit is set to 1 or 0 depending on the nth record having or not the value under consideration for the attribute. Davis and Lin [1] were the first to report the use of bit string inverted lists. Later, Thiel and Heaps [22,23] and King [21] have used bit strings for implementing inverted indexes in document retrieval applications, using counts of zero substrings for compression purposes. In this paper, a complementary inversion technique derived from the bit string representation is considered.

The collection of binary encoded inverted lists for the attribute domains of a file forms a Boolean matrix. In some cases, this matrix is quite sparse. This is due to the fact that, for some attributes, the number of different values present in the file is rather large, producing a relatively high frequency of low occurrence values. This sparseness suggests the idea of subdividing the matrix into submatrices and considering a second Boolean matrix which is a bit map that indicates which submatrices of the first contain a non-zero element. If a submatrix is non-zero it is assigned a one 1 bit in the second matrix, otherwise it is mapped to a 0 bit.

In section 4, the use of these bit maps in multiple key data retrieval is described. In section 5 we make an evaluation of the method for both storage requirements and number of I/O accesses needed. Both storage space and expected retrieval time are considered under the assumptions of random distribution of values in attribute domains, domain independence, and record occurrence independence within the file. Updating strategies are discussed in section 6.

In the paper, the terminology used by Codd in his relational model of data [8,9] is informally adopted. The terms relation, domain, and tuple correspond to the logical view of the data. We will assume that a relation is physically stored as a "flat file", and therefore the terms file, field, and record would be the corresponding to those of Codd's, and we will use them interchangeably The techniques are explained by means of examples.

2. THE PROBLEM

In on-line querying, I/O operations are the principal component accounting for retrieval time. Thus, any attempt to improve performance must aim at reducing the number of I/O accesses made. Also, an efficient data access strategy must recognize the operational characteristics of the mass storage devices presently available. These are (1) the access latency and (2) the data transfer rate. It

This research has been supported by the National Research Council of Canada. Author address: Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, M5S 1A7.

may take anywhere between 10 and 100 milliseconds for the I/O device to position before data transfer can be initiated; but once the device has been positioned data can be sent to the processor at very high transfer rates. Therefore, the file structure must take advantage of the high data rates possible by I/O devices by using large data blocks, and at the same time it must minimize the number of disk accesses. The data retrieval scheme described in the next sections has been designed with these considerations in mind.

We consider the problem of executing a Boolean query Q1 V Q2 V ... V Qt, where each Qi is a conjunction of relational conditions of the form C1∧C2∧...∧Chi. Inverted lists have traditionally been used for this data retrieval problem. In the usual inverted file organization, a collection of pointer lists is kept for each attribute domain, one list per each value present in the file for that attribute. We will reduce our analysis to queries on inverted attributes. Farley and Schuster [7] and Reardon [19] have considered the retrieval problem for Boolean queries when not all attribute domains are inverted.

If each of the Ci is an equality condition, then generally only one access is required to retrieve the inverted list corresponding to the value referenced. This would be the case, for example, if the data base consists of a file of technical articles and the index terms are keywords describing the articles. When some condition Ci in the query involves a range specification by means of the operators $<$, $\leq$, $>$, $\geq$, or $\neq$ a merge of several inverted lists is required. (The condition $(Ci \neq c)$ can be represented as $(Ci<c)$ V $(Ci>c)$.) In order to retrieve the data records that satisfy the conjunction, the resulting lists must then be intersected to determine the pointers to the data records desired. Finally, if the query is a disjunction of conjunctions an additional merge must be performed with the lists corresponding to conjunction results before record retrieval can be done. Alternatively, if the resulting inverted lists for each condition of the conjunction are of very different lengths, one can select the shortest one, retrieve each of the records it points to and evaluate the other conditions after each record has been retrieved. The trade-off here is that, in general, more records will be retrieved than those that satisfy the query. This strategy should be selected when the number of accesses to retrieve inverted lists that are saved is larger than the number of extra accesses made to retrieve data records.

When one deals with a large file and several attribute domains are inverted, the inverted lists begin to grow, both in number and in length, and processing them efficiently becomes another file problem in itself. Cardenas has discussed this problem in [20]. To cope with the large number of inverted values, usually a index tree is built on top of the domain values. The index permits to find quickly the inverted lists corresponding to the values referenced in the query. The inverted lists are also kept in sorted order by pointer values to speed up merging and intersecting (the price being paid, of course, in slower update).

Under these conditions, merging and intersecting become serial operations and can be performed in parallel. This intersecting/merging process insures that every record in the file which satisfy the query is retrieved and that multiple retrievals of records are not performed.

The total retrieval time required to execute a Boolean query by means of inverted lists following the merging/intersecting procedure discussed above depends directly on the number of attribute values referenced explicitly or implicitly by the query and on the number of data records retrieved. We can reduce the I/O time needed to retrieve the inverted lists by considering a more coarse inversion, and the I/O time due to record retrievals by retrieving large data blocks and by clustering the records within these blocks. Wong and Chaing have previously addressed this question in [18]; in the scheme they propose, inverted lists are kept only for canonical conjunctions of attribute/value pairs and records are clustered in the file according to the partitioning induced by this set of conjunctions. However, this file structure presents update difficulties and it may be expensive to construct if the number of distinct attribute/value pairs is large. We will describe next the use of bit maps to achieve the former objectives.

3. BIT MAPS

For clarity of exposition, we first introduce some notation. Let F denote our file, N denote the total number of records in F, and k be the number of different attributes (i.e., fields of F) inverted. We will denote the domain of the ith attribute by di; Mi will be the total number of distinct values present in F for di.

$\lceil \log(N) \rceil$ bits are required to address any record in F (all logarithms expressions throughout the paper are in base 2; the notation $\lceil x \rceil$ is used to represent the smallest integer greater than or equal to x). For a given domain di of F, the total space required to store the bit string inverted lists (ignoring the space required to store the domain values themselves for the accessing index) is $N*Mi$ bits, while the pointer list representation requires $N*\lceil \log(N) \rceil$ bits. Therefore, when $Mi<\lceil \log(N) \rceil$, the bit string representation is more economical in terms of space.

For k domains inverted, the total number of bits required for the inverted lists stored as bit strings is $N* \sum_{i=1}^{k} Mi$, of which only $k*N$ bits are 1, therefore the proportion of non-zero bits is only $k/\sum_{i=1}^{k} Mi$. So, in the case of large files, if we store the inverted lists as bit strings, we will be incurring in extensive space overhead because the resulting Boolean matrix will be largely sparse. On the other hand, when dealing with Boolean queries, the bit string representation makes the operations of merging and intersecting very efficient since we can use the Booleans OR and AND directly, and computers usually provide instructions to execute these operations. Furthermore, the

implementation of the ≠ operator becomes trivial, since it reduces to a complementation on a bit string. This approach would be attractive if we could reduce the space overhead. To this end, we will consider the use of a bit map to compress the Boolean matrix corresponding to the collection of bit string inverted lists for all the attributes inverted in F.

The idea of using a single bit to represent the presence of absence of a certain property or object is well known and many applications of bit maps have appeared in the literature. Pooch and Nieder [11] have surveyed the use of bit maps techniques for sparse matrix manipulation. Hardgrave [12] has proposed the use of bit maps and bit strings for representation of large sparse sets. More recently, the use of binary strings and superimposed coding for multiple key retrieval has been also discussed by Knuth [5]. Casey [4] has described a tree structure implementation that also makes use of bit mappings and superimposed coding.

We will denote the Boolean matrix corresponding to the collection of bit string inverted lists maintained for the file by B0. Let us now consider a subdivision of this matrix into a set of small rectangular submatrices of similar shape, and let us form another Boolean matrix B1 that represents a bit map of B0. Each bit of B1 will correspond to a "rectangle" in the partition of B0. If this rectangle is made of all 0's, the corresponding bit in B1 is set to 0, otherwise it is set to 1. We will present the scheme with an example. Figure 1 shows a sample file called CARS with five attributes: CAR# (which is the key that uniquely identifies each record), MAKE, MODEL, and MILES.

CARS	(CAR#	MAKE	MODEL	MILES)
	324	FORD	75	23
	350	VW	68	152
	363	CHEVROLET	70	121
	412	FIAT	69	94
	445	VOLVO	72	82
	467	FORD	71	27
	504	FORD	75	47
	527	CHEVROLET	74	39
	539	CHEVROLET	68	136
	548	CHRYSLER	74	42
	570	CHRYSLER	73	83
	582	VOLVO	75	15
	630	CHRYSLER	70	89
	638	TOYOTA	74	72
	652	FORD	70	116
	673	DATSUN	73	54
	739	CHEVROLET	73	33
	741	VW	72	130
	750	CHEVROLET	75	8
	761	VOLVO	70	90
	817	DATSUN	73	77
	822	FORD	74	31
	837	FORD	70	142
	854	CHEVROLET	71	64

Figure 1. A sample file

Figure 2 illustrates a bit map for the Boolean matrix corresponding to the collection of inverted lists of the file CARS, where rectangles of size 2x2 were mapped to one bit. The attributes inverted are MAKE, MODEL, and MILES. For the attribute MILES the inversion is provided for a subdivision of the domain into 16 equal intervals. Record keys are indicated to the right.

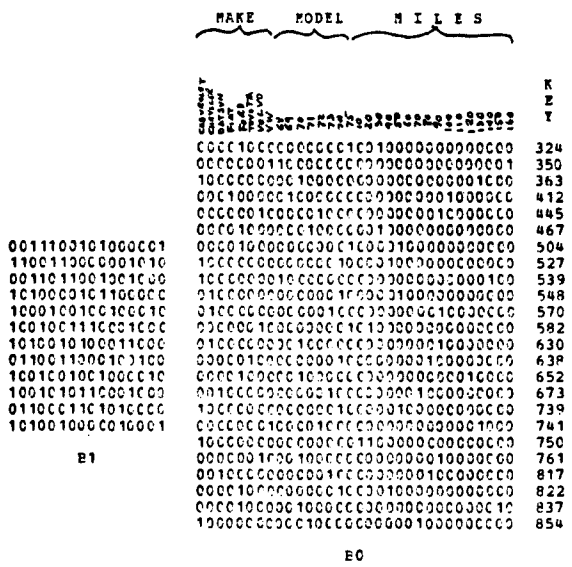

Figure 2. Bit map and matrix of inverted lists for file CARS.

Each bit of matrix B1 has two functions: one is to indicate whether the corresponding rectangle in B is or not all-zero, and the other is to indicate the presence or absence of a range of attribute values in a group of records. Thus the matrix B1 represents in fact a collection of less precise inverted lists where the objects "inverted" are groups of records, and each column corresponds to an interval of values in the corresponding attribute domain. In the scheme described, data records are assumed to be stored in large blocks (or buckets) and a bit map descriptor is provided for each block (this bit descriptor would correspond to a row of matrix B1.) In the above example, records would have been stored in blocks of 2 records.

4. DATA RETRIEVAL

How can we use the bit map in data retrieval? Since the bit map represents a collection of less precise inverted lists it can only direct the retrieval of a block of records. Furthermore, we now have retrieval precision less than 1 since some of the blocks could be accessed due to spurious bit combinations and may not contain any record that satisfies the query.

Let's consider the processing required to execute the general Boolean query Q using bit maps. The bit map can be processed either by rows or by columns. In the second method, we proceed as if processing normal bit string inverted lists. (Now, for the operator ≠, the Boolean operation to be used should be "set to 1" instead of complementation.) A set of bit string "accumulators" is kept in core. For each condition of a conjunction, we OR together the columns involved; then we AND the obtained result columns which yields the result corresponding to the conjunctions, and finally OR this to the result columns of other conjunctions to obtain the final result for the disjunction. In order to find the blocks that "satisfy" the query, one then must

scan the accumulator with the final result performing a counting operation to determine the index of the 1 bits in it. Since the resulting bit vector represented by the accumulator will contain mostly 0's, a fast scan can be made by means of logical masking instructions (considering, say, 256 bytes at a time) and then shifting through a register when the resulting substring is non-zero.

In the method of processing by rows, we must test whether each row has at least one 1 bit in the columns corresponding to the attribute values referenced in the query. When this happens, the block corresponding to that row may contain a record that satisfies the query and should be retrieved. The method of processing by rows was first used by Burke and Rickman [13] for attribute oriented searches but without considering the possibility of attribute values groupings. Unless the bit map fits entirely in core, scanning by rows is impractical because it implies searching the entire bit map. However, the method of processing by rows is interesting because it leads to the idea of clustering those records with the same attribute values in the same block. This is further discussed in section 6.

In either method, since a block retrieved may or may not contain a record that satisfies the query, once the block is in core a sequential search must be performed evaluating the query on each record in the block.

5. PERFORMANCE EVALUATION

Are these bit maps or coarse inverted lists represented as bit strings better than conventional inverted lists? Assuming that we can perform unambiguously all required operations, to evaluate the alternative file structures we must compare

(a) the storage requirements, and
(b) the efficiency of performing the operations of retrieval, deletion, modification, and addition of records.

5.1 Storage requirements

In order to formulate the problem analytically, we will make the following simplifying assumptions about the file:
(1) all attribute domains inverted are independent of one another,
(2) record occurrences in the file are independent of one another,
(3) values present in the file for any given attribute domain are randomly distributed.

Let Bil denote the set of bit map columns corresponding to the domain di. If, for domain i, we map "rectangles" of size pxqi to one bit, the matrix Bil will consist of $\lceil N/p \rceil * \lceil Mi/qi \rceil$ bits. Therefore, the total space required by the bit map Bl is $\lceil N/p \rceil \sum_{i=1}^{k} \lceil Mi/qi \rceil$ bits. This has to be compared with the space required by inverted pointer lists, which is $k*N*\lceil Log(N) \rceil$ bits. p will be an integer representing the blocking factor of records in buckets of secondary storage; in general Mi≤N, and therefore, 1≤qi≤p. (If p=1 and qi=1 for all i, then bit maps reduce to bit string inverted lists.) If we choose qi=1, i=1,...,k, then bit maps will require less space than inverted pointer lists when (approximately)

$$p \geq \lceil (\sum_{i=1}^{k} Mi)/(k*\lceil Log(N) \rceil) \rceil .$$

However, if we let p and the gi be too large the bit map will be saturated with 1's and would be of little value for data retrieval. These parameters must be selected in each particular case according to the file characteristics and query usage. This problem is further discussed in the next section.

We now derive a probability expression which will be needed later. Let's consider a row of Bil. Any one bit in this row represents p values of di corresponding to record occurrences in F. Given our mapping definition, a row of Bil consists of $\lceil Mi/qi \rceil$ bits. Each of these bits represents a rectangle of pxqi bits in BO. Therefore, a row of Bil represents $\lceil Mi/qi \rceil$ rectangles among which the possible p attribute values will be distributed. What is the probability that any one of these rectangles is made of all zeros? Under our previous assumptions, any given value is assigned to a given rectangle is equal to the probability that all values be assigned to the others, that is:

$$vi = ((\lceil Mi/qi \rceil - 1)/\lceil Mi/qi \rceil)^p , \quad i=1,...,k.$$

Thus, the probability of a rectangle having at least one 1 bit, and therefore, the probability of being mapped a 1 bit in Bl is 1-vi.

5.2 Retrieval time

What is the retrieval performance to be expected from bit maps? We will develop our analysis for the simplified case of a conjunctive query Q = C1∧C2∧...∧Ch, where mi values are referenced in each condition Ci, i=1,...,h, h≤k.

A block will satisfy the query if at least one of the p possible records in it satisfies the query. Therefore, the probability of a successful access, Ps, will be

Ps = Prob{block contains at least one record which satisfies the query}
= 1 - Prob{none of the p records satisfies the query}
= 1 - (Prob{one record in block does not satisfy the query})^p
= 1 - (1 - Prob{a record in the block satisfies the query})^p
= 1 - (1 - Prob{each group of mi bits in record descriptor (in BO) has at least one 1})^p
= 1 - (1 - $\prod_{i=1}^{h}$ Prob{group of mi bits has at least one 1})^p
= 1 - (1 - $\prod_{i=1}^{h}$ (1 - Prob{group of mi bits if made of all 0's}))^p
= 1 - (1 - $\prod_{i=1}^{h}$ (1 - ((Mi - 1)/Mi)^{mi}))^p.

The probability that a block is accessed, that is, the probability that the bit map descriptor of the block indicates that the block contains a record that "satisfies" the query, is

Pa = Prob{each of the substrings of mi bits in the query descriptor (in B1) has at least one 1}

$$= \prod_{i=1}^{h} \text{Prob\{substring i of mi bits has at least one 1\}}$$

$$= \prod_{i=1}^{h} (1 - \text{Prob\{all mi bits are 0\}})$$

$$= \prod_{i=1}^{h} (1 - (vi)^{mi})$$

$$= \prod_{i=1}^{h} (1 - (((\lceil Mi/qi \rceil - 1)/\lceil Mi/qi \rceil)^{p})^{mi}).$$

Hence,

pd = Prob{false drop}
= Prob{access is made} - Prob{the block retrieved contains at least one record which satisfies the query}
= Pa - Ps

$$= \prod_{i=1}^{h} (1 - (((\lceil Mi/qi \rceil - 1)/\lceil Mi/qi \rceil)^{p})^{mi}) + (1 - \prod_{i=1}^{h} (1 - ((Mi - 1)/Mi)^{mi}))^{p} - 1.$$

Note that when p=qi=1, the probability of false drop becomes 0.

From the above analysis it follows that the expected number of false drop accesses made in executing the query Q is $\lceil N/p \rceil * Pd$, and the expected number of accesses made for retrieval of data blocks is $\lceil N/p \rceil * Pa$. If we further assume that the mi attribute values referenced by each condition of the query are all consecutive (as would be the case in a condition of the form (Ci<c)), then the number of accesses made to retrieve bit map columns is $\sum_{i=1}^{h} \lceil mi/qi \rceil$, and therefore the total expected number of accesses, E(Q), due to retrieval of bit map columns and data blocks for the query under consideration is $E(Q) = \sum_{i=1}^{h} \lceil mi/qi \rceil + \lceil N/p \rceil * Pa$.

The expressions for the probabilities Pa, Ps, and Pd, can be easily extended to the case of a disjunctive query by considering the union of non-disjoint sets; for example, if Q = Q1 V Q2 and Q1 and Q2 do not involve mutually exclusive conditions, then Pa(Q) = Pa(Q1) + Pa(Q2) - Pa(Q1∧Q2).

We now return to the question of determining the value of the parameters p and qi. The optimum value of p and qi will be those that minimize the number of I/O accesses made, that is E(Q). Minimizing E(Q) analytically is difficult even for the simplified query being considered, unless unwarrantable assumptions are made about the parameters defining the file, Mi and N, and those defining the query structure, mi and h. The problem is further complicated by the effect of clustering because clustering invalidates the assumptions of uniform distribution, and attribute and record independence in the data base. (Clustering is discussed in the next section.)

Choosing the parameters p and qi involves the traditional time/space trade-off: we would want to set p and qi as large as possible to reduce the space required by the bit maps. However, as p and qi increase so will the probability of false drop and therefore the number of accesses made. To get a feel of the behavior of E(Q) consider the following practical example: N=100000; h=4; M1=10, M2=200, M3=1000, M4=2000; m1 = 1, m2 = 3, m3 = 200, m4 = 10; q1 = q2 = 1. We assume that $q^3 = q^4 = q$, and p are the parameters to be considered as variables. Figure 4 shows E(Q) as a function of p and q.

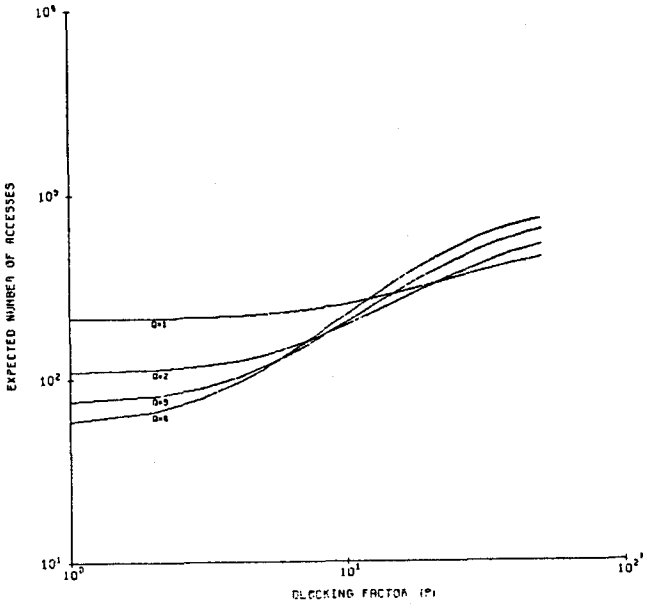

Figure 4. Effect of blocking factor and attribute domain quantization on the number of accesses made.

This and other examples that were computed show that E(Q) seems to be more sensitive to the qi than to p, thus allowing a larger range of possible values for p. In the example, for p=10, q=4 would give better performance than q=1 requiring only 1/4 as many bits to be stored for these attributes. For p>10 performance deteriorates rapidly due to the increased number of false drop accesses being made. In general, if Mi << N, then qi=1 should be chosen. Additional experimentation is needed to determine the optimum values for the qi and p as a function of the file characteristics and query structure.

6. UPDATE OPERATIONS

We consider first the case of additions of new records. First we must select the bucket where the record is going to be placed. If we consider variable length buckets, then we can assume that every bucket has enough spare space for one or more additional records. For fixed length buckets, when a bucket is full we could allocate a new bucket at the end of the file and start a new bit map descriptor for it. However, in this case, an additional bit map column would have to be kept to identify those buckets that are full.

Let us assume first that the bit map fits entirely in core. This is not as unlikely as it would appear. For example, if the file has 10,000

records, and it is stored as 1,000 blocks of 10 records each, for each of which a 96-bit descriptor is kept, the bit map would occuply only 12K bytes, which would fit in a disk track and could be read into core with a single access.

In this case, the bit map would be processed by rows. When a new record X has to be added to the file, we generate the bit map descriptor of X and use a clustering approach to decide where to place X. The new record should be stored within the block with the most "similar" bit map descriptor. Let x be the bit map descriptor of record X. We select the block corresponding to bit map row y where the new record is to be placed such that d(x,y) is minimum, where d(x,y) is the number of 1's in (x XOR y), that is, the Hamming distance between x and y. In the case that more than one such block y is found, one of them could be selected at random, or the bucket with most space remaining could be chosen.

This approach for placing new records in the file has two advantages: (1) it would cluster the additions in the bit maps which will reduce the probability of false drops in future retrievals, and (2) it would cluster the records in the buckets on disk, which will result in fewer accesses during retrieval since the records will tend to be grouped together according to the values inverted.

If the bit map is too large to fit entirely in core then processing by rows is too expensive and we must store and process it by columns. In this case, the update strategies suggested by Casey [4] can be applied. In order to add a new record X to the file we generate and execute a query with X's values for the domains inverted, and retrieve the bit map columns corresponding to the attribute values associated with X. (In the case that the new record to be added involves a value for an attribute not presently in the file, the selection must of course be based only on the remaining attributes.) In general, upon executing the query, we would obtain a set of candidate blocks where the new record could be placed, any one of which could be chosen. The clustering effectiveness will now be much less than before because whole block descriptors are not being considered. Note that in order to make available a whole bit map row, another copy of the bit map would have to be stored by rows which would imply doubling the space requirements and update overhead.

As with additions, to delete record X we generate a query with X's values for the domains inverted. We execute the query to find the location of X in the file. Since one or more blocks may be retrieved, the record to be deleted must be further identified by means of the primary key attribute in the file, and then flagged as deleted. The bit map descriptor for the block containing the deleted record must then be recomputed again as a function of all the records in the block. If many attributes are inverted this may be time consuming; a "garbage collection" approach could be adopted in which the bit map is not updated immediately but periodically. Under this strategy all blocks updated are flagged (possibly by using an additional bit map column) and their corresponding bit map descriptor recomputed during the garbage collection pass. This strategy will tend to degrade clustering effectiveness (and therefore retrieval performance). This might require that garbage collection be executed rather frequently, possibly as a function of the percentage of flagged blocks..

Modifications to the file are best treated as a deletion of the old record followed by the addition of the modified record. This will ensure the clustering effect and preserve the precision of the bit maps.

From the above discussion it follows that, unless the bit map is small enough to fit in core, update operations are fairly expensive. However, the possibility of dynamic clustering offered by the method could produce a definite improvement in performance during retrievals. Further research lies ahead to investigate this interesting possibility.

7. CONCLUSION

The use of bit maps for multiple key retrieval has been discussed and some simple expressions relating to space and time performance have been derived. Bit maps represent less precise inverted lists where the inversion corresponds to a quantization of the attribute domain and the objects are groups of data records.

Bit maps have the advantage that less space will be generally required than in the normal pointer representation of inverted lists. The merging and intersecting process required to execute complex Boolean queries are greatly facilitated since they reduce to Boolean operations on bit strings. Update operations are expensive but the required CPU processing is simple because bit map columns are of fixed length.

The main disadvantages of the method is its retrieval precision less than 1, i.e., the possibility of useless accesses, due to the problem of false drops. However, this problem can be minimized by using the clustering strategy for record addition and modification. The possibility of dynamic clustering provided by the scheme would also improve retrieval performance, especially when the average number of records that satisfy the query is large.

Acknowledgements

The author is grateful to S.A. Schuster and K.C. Sevcik for their critical reading of this paper.

8. REFERENCES

[1] Davis, D.R., and Lin, A.D. "Secondary key retrieval using an IBM 7090-1301 system". CACM 8, 4 (April 1965), 243-246.

[2] Fraser, W.D. "A proposed system for multiple descriptor data retrieval", in Some Problems in Information Systems, M. Kocken (Ed.). Scarecrow Press, New York, 1965, pp. 187-205.

[3] Lefkovitz, D.
File structures for on-line systems. Spartan Press, New York, 1969.

[4] Casey, R.G.
"Design of tree structures for efficient querying". CACM 16, 9 (September 1973), 549-556.

[5] Knuth, D.E.
The art of computer programming V3: sorting and searching. Addison-Wesley, Reading, Mass., 1973, pp. 550-567.

[6] Rothnie, J.B., and Lozano, T.
"Attribute based file organization in a paged environment". CACM 17, 2 (February 1974), 63-69.

[7] Farley, G.H.J., and Schuster, S.A.
"Query execution and index selection for relational data bases." Technical report CSRG-53, Computer Systems Research Group, University of Toronto, 1975.

[8] Codd, E.F.
"A relational model for large shared data banks". CACM 13, 6 (June 1970), 377-387.

[9] Codd, E.F.
"Relational completeness of data base sub-languages". Courant Computer Science Symposia 6, in Data Base Systems, Randall Rustin (Ed.), Prentice Hall, New Jersey, 1972, pp. 65-98.

[10] Boyce, R.F., Chamberlin, D.D., King III,W.F., and Hammer, M.M. "Specifying queries as relational expressions: SQUARE". IBM Technical Report RJ1291, IBM Res. Labs., San Jose, California, October 1973.

[11] Pooch, U.W., and Nieder, A.
"A survey of indexing techniques for sparse matrices". ACM Computing Surveys 5, 2 (June 1973), 109-133.

[12] Hardgrave, W.T.
"The prospects of large capacity set support systems imbedded within generalized data base management systems". Proc. International Computing Symposium 1973, Davos, Switzerland. North-Holland Publishing Co., Amsterdam, 1974, pp. 549-556.

[13] Burke, J.M., and Rickman, J.T.
"Bit maps and filters for attribute-oriented searches". International Journal of Computer and Information Sciences 2,3 (1973), 187-200.

[14] Martin, J.
Computer Data-Base Organization. Prentice-Hall, New Jersey, 1975, pp. 379-200.

[15] Hsiao, D., and Harary, F.
"A formal system for information retrieval from files". CACM 13, 2 (February 1970), 67-73.

[16] Vose, M.R., and Richardson, J.S.
"An approach to inverted index maintenance". The Computer Bulletin 16, 5 (May 1972) 256-262.

[17] Inglis, J.
"Inverted indexes and multilist structures". The Computer Journal 17 (February 1974), 59-63.

[18] Wong, E., and Chaing, T.C.
"Canonical structure in attribute based file organization". CACM 14, 9 (September 1971), 593-597.

[19] Reardon, B.C.
"An adaptive information retrieval system using partial file inversion". Information Storage and Retrieval 2, 10 (February 1974), 49-56.

[20] Cardenas, A.F.
"Analysis and performance of inverted data base structures". CACM 5, 18 (May 1975), 253-263.

[21] King, D.R.
"The binary vector as a basis of an inverted index file". Journal of Library Automation 7, 4 (December 1974), 307-314.

[22] Thiel, L.H. and Heaps, H.S.
"Program design for restrospective searches on large data bases". Information Storage and Retrieval 8 (1972), 1-20.

[23] Heaps, H.S. and Thiel, L.H.
"Optimum procedures for economic information retrieval". Information Storage and Retrieval 6 (1970), 137-153.