Multi-dimensional Sparse Matrix Storage*

Jiří Dvorský, Michal Krátký

Department of Computer Science, VŠB – Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava–Poruba
{jiri.dvorsky,michal.kratky}@vsb.cz

Abstract. Large sparse matrices play important role in many modern information retrieval methods. These methods, such as clustering, latent semantic indexing, performs huge number of computations with such matrices, thus their implementation should be very carefully designed. In this paper we discuss three implementations of sparse matrices. The first one is classical, based on lists. The second is previously published approach based on quadrant trees. The multi-dimensional approach is extended and usage of general multi-dimensional structure for sparse matrix storage is introduced in this paper.

Key words: sparse matrix, multi-dimensional data structure, quadrant tree, BUB-tree, R-tree

1 Introduction

Numerical computations represent serious problem for generations of mathematicians. There were not suitable device to make the computations, only human being. Development of computers gives to people power to perform computations, which were impossible in past. Many of these computations have matrix character. Thus one of the first task for computers was matrix and vector computations i.e. liner algebra. Although amount of memory in computers grows very rapidly, there are still matrices that are bigger than available memory. But many of these matrices are sparse, so that storage only non-zero values can solve the problem. Large sparse matrices play important role in industrial computations (e.g. FEM - Finite Elements Method), in computer science (indexing of class hierarchy [5]), and in many modern information retrieval methods. These methods, such as clustering, latent semantic indexing, performs huge number of computations with such matrices, thus their implementation should be very carefully designed.

This paper is organized as follows. Section 2 describe state-of-art in sparse matrix implementation. A previously published approach for sparse matrix storage [10] based on finite automata is given in Section 3. The multi-dimensional approach is extended and usage of general multi-dimensional structure for sparse

* This work was done under grant from the Grant Agency of Czech Republic, Prague No.: 201/03/1318

matrix storage is introduced in this paper. This storage method is described in Section 4. In Section 6 preliminary experimental results are shown. Finally, we conclude with a summary of contributions and discussion on future work.

2 Short survey of sparse matrix storage

Let $\mathcal{A}$ be a sparse matrix of order $n \times m$. The matrix $\mathcal{A}$ can be be efficiently processed, if the zero elements of $\mathcal{A}$ are not stored. There are many methods for storing the data (see for instance [1]). Here we will discuss Compressed Row and Column Storage.

2.1 Compressed Row Storage (CRS)

The Compressed Row Storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix $\mathcal{A}$, we create 3 vectors: one for floatingpoint numbers (val), and the other two for integers (col_{ind}, row_{ptr}). The val vector stores the values of the nonzero elements of the matrix $\mathcal{A}$, as they are traversed in a rowwise fashion. The col_{ind} vector stores the column indexes of the elements in the val vector. That is, if $val(k) = a_{i,j}$ then $col_{ind(k)} = j$. The row_{ptr} vector stores the locations in the val vector that start a row, that is, if $val(k) = a_{i,j}$ then $row_{ptr(i)} \leq k < row_{ptr(i+1)}$. By convention, we define $row_{ptr(n+1)} = n_{nz} + 1$, where n_{nz} is the number of nonzeros in the matrix $\mathcal{A}$. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2n_{nz} + n + 1$ storage locations.

The CRS format for this matrix is then specified by the arrays val, col_{ind}, row_{ptr} given in Table 1. If the matrix $\mathcal{A}$ is symmetric, we need only store the upper (or lower) triangular portion of the matrix. The tradeoff is a more complicated algorithm with a somewhat different pattern of data access.

2.2 Compressed Column Storage (CCS)

Analogous to Compressed Row Storage there is Compressed Column Storage (CCS), which is also called the *Harwell-Boeing sparse matrix format* [6]. The CCS format is identical to the CRS format except that the columns of $\mathcal{A}$ are stored (traversed) instead of the rows. In other words, the CCS format is the CRS format for $\mathcal{A}^T$.

The CCS format is specified by the 3 arrays val, row_{ind}, col_{ptr}, where row_{ind} stores the row indices of each nonzero, and col_{ptr} stores the index of the elements in val which start a column of $\mathcal{A}$. The CCS format for the matrix $\mathcal{A}$ in equation (1) is given in Table 2.

Example 1. As an example, consider the nonsymmetric matrix $\mathcal{A}$ defined by

$$\mathcal{A}=\begin{pmatrix}10000-2&0\\3900&0&3\\0787&0&0\\3087&5&0\\0809&9&13\\0400&2-1\end{pmatrix} \tag{1}$$

Table 1. The CRS format for the matrix $\mathcal{A}$ in equation (1)

val	10	-2	3	9	3	7	8	7	3	...	9	13	4	2	-1
col_{ind}	1	5	1	2	6	2	3	4	1	...	5	6	2	5	6

row_{ptr}	1	3	6	9	13	17	20

Table 2. The CCS format for the matrix $\mathcal{A}$ in equation (1)

val	10	3	3	9	7	8	4	8	8	...	9	2	3	13	-1
row_{ind}	1	2	4	2	3	5	6	3	4	...	5	6	2	5	6

col_{ptr}	1	4	8	10	13	17	20

2.3 Properties of CRS and CCS formats

The Compressed Row and Compressed Column Storage formats are general formats: they make absolutely no assumptions about the sparsity structure of the matrix, and they does not store any unnecessary elements.

On the other hand, these methods effectively support only part of matrix operations. While CRS can access any row vector in time $O(1)$, column vector can be selected in $O(m \times \log_2 \Delta)$, where $\Delta = row_{ptr(i)} - row_{ptr(i+1)}$ ie. number of nonzero elements in row i. Time complexity of these operations in CCS format is reverse. For example CRS format can effectively perform matrix - colum vector and CCS row vector - matrix multiplication. Any other matrix operation (eg. selection of submatrix) can be done with these formats, but time complexity is very high. Moreover the formats can be used only in the main memory of computer.

Aim of our work is to develop storage format for large sparse matrices. The format should support:

– random access to the matrix,
– effective selection of any submatrix
– persistence of the matrix (usage of secondary memory).

3 Sparse matrices and finite automata

Culik and Valenta [4] introduced finite automata for compression of bi-level and simple color images. A digitized image of the finite resolution $m \times n$ consists of $m \times n$ pixels each of which takes a Boolean value (1 for black, 0 for white) for bilevel image, or a real value (practically digitized to an integer between 0 and 256) for a grayscale image.

Sparse matrix can be viewed, in some manner, as simple color image too. Zero element of matrix corresponds to white pixel in bi-level image and nonzero element to black or gray-scale pixel.

Here we will consider square matrix $\mathcal{A}$ of order $2^n \times 2^n$ (typically $13 \leq n \leq 24$). In order to facilitate the application of finite automata to matrix description we will assign each element at $2^n \times 2^n$ resolution a word of length n over the alphabet $\Sigma = \{0, 1, 2, 3\}$ as its address. A element of the matrix corresponds to a subsquare of size 2^{-n} of the unit square. We choose ε as the address of the whole square matrix.

Its submatrices (quadrants) are addressed by single digits as shown in Figure 1(a). The four submatries of the matrix with address ω are addressed $\omega 0$, $\omega 1$, $\omega 2$ and $\omega 3$, recursively. Addresses of all the submatrices of dimension 4×4 are shown in Figure 1(b). The submatrix (element) with address 3203 is shown on the right of Figure 1(c).

In order to specify a values of matrix of dimension $2^n \times 2^n$, we need to specify a function $\Sigma^n \to R$, or alternately we can specify just the set of non-zero values, i.e. a language $L \subseteq \Sigma^n$ and function $f_{\mathcal{A}} : L \to R$.

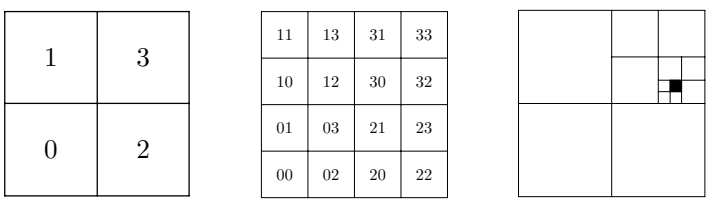

Fig. 1. The addresses of the submatrices (quadrants), of the subsmatrices of dimension 4×4, and the submatrix specified by the string 3203

This kind of storage system allows direct access to stored matrix. Each of elements can be accessed independently to previous accesses and access to each element has same, constant time complexity. Let $\mathcal{A}$ be a matrix of order $2^n \times 2^n$. Then time complexity of access is bounded by $O(\log_2 n)$. For detail information see [10].

Example 2. Let $\mathcal{A}$ be a matrix of order 8×8.

$$\mathcal{A} = \begin{pmatrix} 20000000 \\ 04001000 \\ 00300609 \\ 00010000 \\ 00001000 \\ 00000500 \\ 00000090 \\ 00000007 \end{pmatrix}$$

The language $L \subseteq \Sigma^3$ is now

$$L = \{111, 112, 121, 122, 211, 212, 221, 222, 303, 310, 323\}.$$

Then function $f_{\mathcal{A}}$ will have following values (see Table 3).

Table 3. Positions in matrix $\mathcal{A}$ and corresponding values – function $f_{\mathcal{A}}$

$x \in L$	$f_{\mathcal{A}}(x)$	$x \in L$	$f_{\mathcal{A}}(x)$
111	2	221	9
112	4	222	7
121	3	303	6
122	1	310	1
211	1	323	9
212	5		

Now automaton that computes function $f_{\mathcal{A}}$ can be constructed (see Figure 2). The automaton is 4-ary tree, where values are stored only at leaves. This knowledge leads to multi-dimensional sparse matrix storage and usage of the quadrant tree.

4 Multi-dimensional sparse matrix storage

In order to a general multi-dimensional data structure can be used for the sparse matrix storage, the following definitions must be introduced.

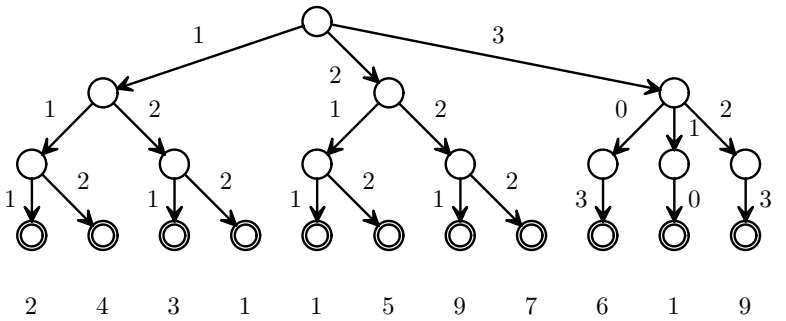

Fig. 2. Automaton for matrix $\mathcal{A}$

Definition 1 (A matrix as tuples of 2-dimensional space).

Let $\mathcal{A}$ be a matrix of order $n \times m$ and $\Omega_{MT} = D_N \times D_M$ be an 2-dimensional discrete space (called matrix space), where $D_N = \{0, 1, \ldots, 2^{l_N} - 1\}$, $D_M = \{0, 1, \ldots, 2^{l_M} - 1\}$. It holds $n \leq 2^{l_N} - 1$, $m \leq 2^{l_M} - 1$. For all $a_{i,j} \in \mathcal{A}$ there is mapping $\alpha : \mathcal{A} \rightarrow \Omega_{MT}$ such that $\alpha(a_{i,j}) = (i, j)$.

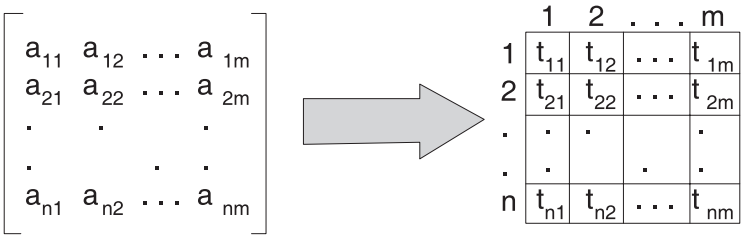

Fig. 3. A matrix as tuples of 2-dimensional space.

The mapping α transforms elements of matrix $\mathcal{A}$ to 2-dimensional space Ω_{MT}. The matrix space can be seen in Figure 3. The matrix space seems to be 3-dimensional. However, indices of matrix elements have to be indexed. The value of element is stored as non-index data. Consequently, only two coordinates must be indexed, so that the matrix space is only 2-dimensional.

4.1 Retrieving of a sub-matrix

A sub-matrix is retrieved using the range query.

Definition 2 (Range query).

Let Ω be an n-dimensional discrete space, $\Omega = D^n$, $D = \{0, 1, \ldots, 2^{l_D} - 1\}$, and points (tuples) $T^1, T^2, \ldots, T^m \in \Omega$. $T^i = (t_1, t_2, \ldots, t_n)$, l_D is the chosen length of a binary representation of a number t_i from domain D. The range query *RQ is defined by a* query hyper box *(*query window*) QB which is determined by two points $QL = (ql_1, \ldots, ql_n)$ and $QH = (qh_1, \ldots, qh_n)$, QL and $QH \in \Omega$, ql_i and $qh_i \in D$, where $\forall i \in \{1, \ldots, n\} : ql_i \leq qh_i$. This range query retrieves all points $T^j = (t_1, t_2, \ldots, t_n)$ in the set $T^1, T^2, \ldots, T^m$ such as $\forall i : ql_i \leq t_i \leq qh_i$.* ■

Let be $\mathcal{A}_{i_1 j_1 i_2 j_2}$ a sub-matrix of matrix $\mathcal{A}$. Sub-matrix is retrieved from the matrix space using the range query $(i_1, j_1) : (i_2, j_2)$. A column vector and row vector are special kind of the sub-matrix. Consequently, the column vector $c_i^{\mathcal{A}}$, $1 \leq i \leq m$, is retrieved using the range query $(1, i) : (n, i)$, the row vector $r_j^{\mathcal{A}}$, $1 \leq j \leq n$, is retrieved using the range query $(j, 1) : (j, m)$. Such range query is called the *narrow range query*. Next Section describes some multi-dimensional data structures, especially a multi-dimensional data structure for efficient processing of the narrow-range query.

5 Multi-dimensional data structures

Due to the fact that a matrix is represented as a set of points in 2-dimensional space in the multi-dimensional approach, we use multi-dimensional data structures for their indexing, e.g., paged and balanced multi-dimensional data structures like UB-tree [2], BUB-tree [7], R-tree [8], and R*-tree [3].

(B)UB-tree data structure applies *Z-addresses* (*Z-ordering*) [2] for mapping a multi-dimensional space into single-dimensional. Intervals on *Z-curve* (which is defined by this ordering) are called *Z-regions*. (B)UB-tree stores points of each Z-regions on one disk page (tree leaf) and a hierarchy of Z-regions forms an index (inner nodes of tree). In Figure 4(a) we see two-dimensional space with 8 points (tuples) and Z-regions dividing the space. Figure 4(b) denotes schematically a BUB-tree indexing this space.

In the case of indexing point data, an R-tree and its variants cluster points into *minimal bounding boxes* (*MBB*s). Leafs contain indexed points, super-leaf nodes include definition of MBBs and the other inner nodes contain hierarchy of MBBs. (B)UB-tree and R-tree support *point* and *range queries* [11], which are used in the multi-dimensional approach to sparse matrix storage. The range query is processed by iterating through the tree and filtering of irrelevant tree nodes, i.e. (super)Z-regions in the case of (B)UB-tree and MBBs in the case of R-tree, which do not intersect a query box.

The range query often used in the multi-dimensional approach is called *narrow range query*. Points defining a query box have got some coordinates the same, whereas the size of interval defined by other coordinates near to the size of space's domain. Notice, regions intersecting a query box during processing of a range query are called *intersect regions* and regions containing at least one point of the query box are called *relevant regions*. We denote their number by

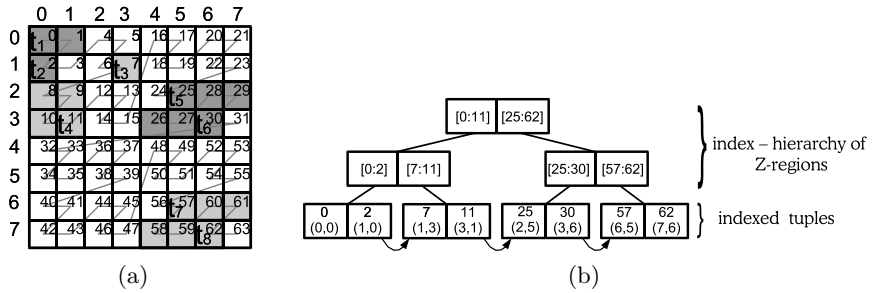

Fig. 4. (a) 2-dimensional space 8×8 with points t_1 – t_8. These points define partitioning of the space to Z-regions [0:2],[7:11],[25:30],[57:62] by capacity of BUB-tree's nodes 2. (b) BUB-tree indexing this space.

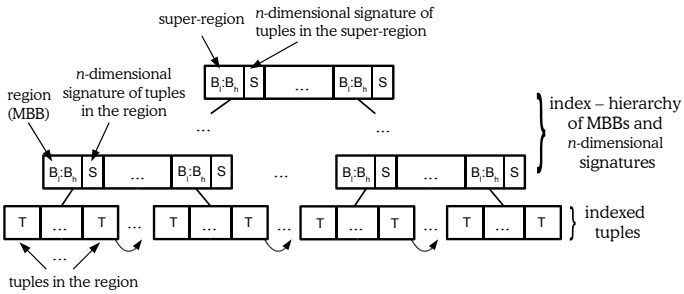

Fig. 5. A structure of the Signature R-Tree.

N_I and N_R, respectively. Many irrelevant regions are searched during processing of the narrow range query in multi-dimensional data structures. Consequently, a ratio of relevant and intersect regions, so called *relevance ratio* $c_R \ll 1$ with an increasing dimension of indexed space. In [9] Signature R-tree data structure was introduced. This data structure enables efficient processing of the narow range query. Items of inner nodes contain a definition of (super)region and n-dimensional signature of tuples included in the (super)region (see Figure 5). A superposition of tuples of coordinates by operation `OR` creates the signature. Operation `AND` is used for better filtration of irrelevant regions during processing of the narrow range query. Other multi-dimensional data structures (e.g. (B)UB-tree) are possible to extend in the same way.

6 Experimental results

In our experiments[1], we used a randomly generated sparse matrix $10^7 \times 10^6$. The matrix contains 5×10^6 of non-zero values. The BUB-tree was used for our test. The index size is 80 MB (compare to 38MB of CRS matrix storage). In Table 4 a characterization of the BUB-tree for storage of the matrix is shown.

Table 4. A characterization of BUB-tree used for sparse matrix storage

Dimension	2	Utilisation	68.1%
l_N, l_M	24	D_N, D_M	$2^{24} - 1$
Number of tuples	5,244,771		
Number of inner nodes	20,351	Number of leaf nodes	249,297
Inner node capacity	19	Leaf node capacity	30
Item size	12 B	Node size	308 B

In the test, randomly generated column and row vectors were retrieved from the BUB-tree. The average number of result tuples (items of a sub-matrix), searched leaf nodes (Z-regions), disk access cost (DAC), and time were measured. A ratio of the searched leaf nodes and all leaf nodes is shown in square brackets. Table 4 shows the result of our tests.

Table 5. Experimental results of the multi-dimensional sparse matrix storage

Number of result tuples	**Number of searched leaf nodes**	**DAC**	**Time** [s]
199	101 [0.041%]	285	0.04

We see that very small part of the index was searched and time of searching was low as well. Experiments prove the approach can serve as efficient sparse matrix storage. The index size is lager than in the case of classical CRS or CCR sparse-matrix storage, but a arbitrary sub-matrix may be retrieved in our approach.

7 Conclusion

In this contribution the multi-dimensional approach to indexing sparse matrix was described. Previously published approach [10] using the quad tree was de-

[1] The experiments were executed on an Intel Pentium ®4 2.4Ghz, 512MB DDR333, under Windows XP.

scribed and a general multi-dimensional approach was introduced. Our experiments prove the approach can serve as efficient sparse matrix storage. In our future work, we would like further to test our approach over a real matrix and to compare the approach with other sparse matrix storage approaches.

References

1. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
2. R. Bayer. The Universal B-Tree for multidimensional indexing: General Concepts. In *Proceedings of WWCA'97, Tsukuba, Japan*, 1997.
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331.
4. K. Culik and V.Valenta. Finite automata based compression of bi-level and simple color images. In *Computer and Graphics*, volume 21, pages 61–68, 1997.
5. P. Dencker, K. Drre, and J. Heuft. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems*, 6(6):546–572, 1984.
6. I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.
7. R. Fenk. The BUB-Tree. In *Proceedings of 28rd VLDB International Conference on VLDB*, 2002.
8. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD 1984, Annual Meeting, Boston, USA*, pages 47–57. ACM Press, June 1984.
9. M. Krátký, V. Snášel, J. Pokorný, P. Zezula, and T. Skopal. Efficient Processing of Narrow Range Queries in the R-Tree. In *Submitten at VLDB 2004*, 2003.
10. V. Snášel, J. Dvorský, and V. Vondrák. Random access storage system for sparse matrices. In G. Andrejková and R. Lencses, editors, *Proccedings of ITAT 2002*, Brdo, High Fatra, Slovakia, 2002.
11. C. Yu. *High-Dimensional Indexing*. Springer–Verlag, LNCS 2341, 2002.