Optimal Block Size for Set-Valued Attributes

Suchen H. Hsu and Richard Snodgrass

Keywords: temporal database, non-first-normal form database

1 Introduction

In a relational database management system, allowing tuples to be of variable length can complicate storage management considerably. As an example, consider an employee relation containing the employee's name, social security number, and salary. We could store the names of the employee's dependents in a separate relation, which would require an expensive join to access. As an alternative, we could add a set-valued attribute, called dependents, to the employee relation, with each tuple containing zero, one, or more values for that attribute. In non-first normal form relational databases, set-valued attributes are commonplace [3, 10, 11, 12, 13]. Variable-length tuples also arise in temporal databases, where sets of time intervals are associated with attributes or with tuples [1, 2, 5, 6, 8, 9].

One approach to representing such tuples on disk is to store the set-valued attribute values separately from the rest of the (now fixed-length) tuple. The sets can be stored as a linked list of fixed-size *blocks*, each storing one or several attribute values. If the sets exhibit a wide range of cardinalities, a linked list is an appropriate storage structure. The alternative of using a hash table to store the set of attribute values was rejected because such a table is also of variable size, complicating page space management.

The question we address is, What is the optimal number of attributes in a block? If one attribute per block is employed, we may waste space on pointers. On the other hand, if large blocks are used, the last block of each list may be largely unfilled. The objective of this paper is to obtain an expression identifying the optimal block size, to achieve the best space utilization.

In the next section, an analytical model is presented which can help determine the optimal block size. An example from temporal databases that exploits this model is described in Section 3. Section 4 provides a summary.

2 Assumptions and Model

Before we define the analytical model for determining the optimal block size, we make three assumptions about the storage structure. The first is that set-valued attributes are stored on the same (logical) page. Page overflows are stored in logically consecutive pages. Each page contains the disk address of its overflow page (e.g., [14]), and all overflow pages are brought into main memory along with the primary page. Because the address of the overflow page is always present, it will not enter into the tradeoff between large and small block sizes. Under this assumption, the time to follow a pointer (to get to the next block) and to increment an address (to access the next attribute value within a block) are about the same. A long list of small blocks does not yield poor time performance even though it requires traversing through many pointers.

The second assumption is that the length of each attribute value is fixed. Individual attribute values that are themselves of variable length are not considered in this paper.

Finally, the end of the list may be represented either with a count field or with a distinguished value for an attribute. Our model assumes a count field is used; this count field would be stored in the fixed portion of the tuple.

The storage structure for representing set-valued attributes on a database page is shown in Figure 1(a). In this arrangement, the fixed length portion of tuples is stored at the top of the page, growing downward. To link the sets with the tuple, one or more pointers are also stored in the tuple, referencing the first block of each list. Blocks, stored at the bottom of the pages, contain a fixed number of *cells*, each of which can hold a single attribute value, and a pointer referencing the next block, as shown in Figure 1(b). By traversing through the blocks, we can retrieve all of the attribute values in the set. Because the individual attribute values are of the same size, the block is also of a fixed size. The pointer can store either the physical address or the block number in the page. The latter representation requires a smaller pointer size to address the same maximum number of blocks.

Our model enables one to calculate the average number of bytes needed per attribute and the average number of bytes wasted per attribute. The percentage of wasted space may then be computed. A smaller percentage indicates higher space utilization achieved.

We first define some variables for the formulas.

m : the maximum number of attribute values in a list

$P(i)$: the probability of a list containing i attribute values

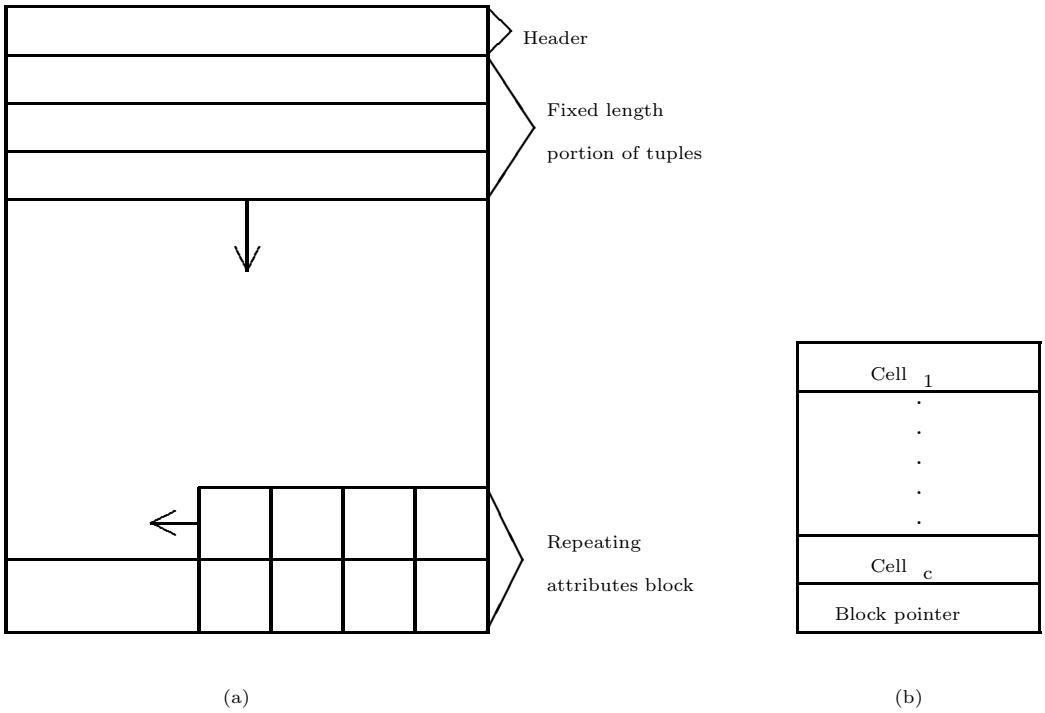

Figure 1: The Page and Block Structure

l : the size of a cell in bytes

s: the size of a pointer in bytes

c : the number of cells in a block

The expected total space for the blocks in a list representing a set is

$$(c \cdot l + s) \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i), \tag{1}$$

where $c \cdot l + s$ is block size. $\lceil i/c \rceil$ is the number of blocks needed to store i attributes. The sum of $\lceil i/c \rceil P(i)$ is the total number of blocks expected. Similarly, the expected number of attributes is $\sum_{i=1}^{m} i \cdot P(i)$. The division of (1) by this quantity yields the average number of bytes needed per attribute.

$$\frac{(c \cdot l + s) \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i)}{\sum_{i=1}^{m} iP(i)} \tag{2}$$

Adopting a similar approach, we can calculate the average wasted space per attribute. There are two aspects of space considered wasted: empty cells in blocks, and the space for storing pointers. To predict the number of bytes in empty cells, we must know how many empty cells are

in the last block of a list of attributes. This can be computed by $\lceil i/c \rceil - i/c$, since the $\lceil i/c \rceil$ is the number of blocks used and i/c is the actual number of blocks needed. The total number of empty bytes for all i under the given probability distribution is then

$$(c \cdot l) \sum_{i=1}^{m} \left(\left\lceil \frac{i}{c} \right\rceil - \frac{i}{c} \right) P(i). \tag{3}$$

The number of bytes needed for pointers is just the number of blocks used times s.

$$s \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i) \tag{4}$$

The total space wasted is the sum of (3) and (4). The average number of bytes wasted is

$$\frac{c \cdot l \sum_{i=1}^{m} \left(\left\lceil \frac{i}{c} \right\rceil - \frac{i}{c} \right) P(i) + s \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i)}{\sum_{i=1}^{m} iP(i)}. \tag{5}$$

The percentage of wasted space is (5) divided by (2) times 100, or

$$\frac{(c \cdot l) \sum_{i=1}^{m} \left(\left\lceil \frac{i}{c} \right\rceil - \frac{i}{c} \right) P(i) + s \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i)}{(c \cdot l + s) \sum_{i=1}^{m} \left\lceil \frac{i}{c} \right\rceil P(i)} \times 100. \tag{6}$$

Since the number of cells in a block is an integer, we can substitute various values of c, from 1 to m, and obtain the percentage of waste space. The number that gives the smallest percentage is the optimal cell number.

3 An Example: Chained Events and Intervals

We give an example to illustrate how to apply the model. In this example, we want to determine the optimal block size for a list of event or interval timestamps in a temporal relational database recording the history of an enterprise over time.

In a temporal database, there are two ways to associate timestamps with relations: *tuple timestamping* and *attribute value timestamping* [1]. If the timestamp is attached to a tuple each time the tuple has been updated, it is called tuple timestamping [2]. If the timestamp is attached to each attribute value to indicate the period of time that the attribute value is valid, it is called attribute value timestamping [5, 6, 9]. Both of these representations can be mapped to the data structure in Figure 1; they differ on how many lists are associated with each tuple. For tuple timestamping, each tuple would contain a pointer to the set of intervals during which the tuple was valid. For attribute timestamping, each tuple would contain a pointer to a set of intervals for each time-varying attribute.

To use the model derived in the previous section, four variables need to be defined: $P(i)$, l, s and m. Interesting distributions for $P(i)$, the probability of a list of i intervals, are uniform, normal, exponential, and gamma.[2] We normalize the normal distribution according to m and shift the center of normal distribution to $m/2$. The probability of a list of i attributes can then be obtained over the range of 1 to m. We don't normalize the exponential and gamma distributions, because the normalized distributions are very flat. There is not much difference between $P(1)$ and $P(m)$, if m is large. The approach that we use is to evenly spread m points in the range of (0..8) for exponential distribution and in the range of (0..10) for gamma distribution. The reasons are first, both $p(8)$ in the exponential distribution and $p(10)$ in the gamma distribution are almost equal to zero and second, the distributions put more weight on small numbers as opposed to large numbers that are more common in practical use. The probability of i is computed by $P(8 * i/m)$ and $P(10 * i/m)$, respectively. Although the integration of the new probabilities is greater than 1, the total space and wasted space are divided by the expected number of attributes, so the average is still correct.

The timestamp representation we use occupies eight bytes with the granularity of one second for low resolution and one microsecond for high resolution [4]. A high resolution timestamp can represent 28 thousand years and a low resolution timestamp can represent thirty billion years. This representation is adequate for most applications. An interval, which requires two timestamps to represent the delimiting events, occupies sixteen bytes. Another way to represent intervals is to store the start of interval and the span of the interval. Because most intervals represent a short period of time (e.g., within 100 years), the representation of such spans can require less space than storing a second timestamp. For example, using a four byte span, which represents 136 years with low resolution, the alternate interval representation takes twelve bytes. We examine three cell sizes: an eight-byte cell for one timestamp representing an event, a sixteen-byte cell representing an arbitrary interval, and a twelve-byte cell as an alternate interval representation.

We have studied two pointer sizes. The first, one byte long, can reference up to 256 blocks (assuming the interval is referenced by block number). This restricts the number of overflow pages. The smallest block contains eight bytes for an event (i.e., one cell) and one byte for a pointer. If pages were 512 bytes long, an overflow chain would be no longer than about four pages for the blocks, depending of course on the size of the fixed-length portions. The other pointer size of two bytes can reference a 64K byte space. This can reference up to 4096 intervals, which should be

[2]The gamma distribution used is $P(x) = \frac{x^{\alpha-1}}{\Gamma(\alpha)\beta^{\alpha}} e^{-(x/\beta)}$. We use $\alpha = 2$ and $\beta = 1$, i.e., $P(x) = \frac{x}{\Gamma(2)} e^{-x}$.

Figure 2: Percentage of wasted space per block

sufficient for our purposes. Since the limit of a one byte pointer is 256, the m (maximum number of attributes) should not be bigger than 256. We experiment with m from 16 to 256 in steps of 8.

There are 8 combinations of distributions and pointer sizes. Samples of the results are shown in Figure 2(a) for one byte pointers and a gamma distribution and in Figure 2(b) for two byte pointers and a normal distribution; the remaining distributions and pointer sizes are given elsewhere [7]. Generally, as would be expected, small blocks are preferable for short lists of intervals; large blocks are better for long lists of intervals. One-cell blocks are always better for a small m. Then a two-cell block becomes preferable. The rate of change depends on the distribution. For the uniform and normal distributions, the rate is about the same. One-cell blocks are the best choice for m less than 16; a six-cell block is best when m is close to 256. The rate of change is much slower for the exponential and gamma distribution. In fact, there is not much difference between two-cell blocks and four-cell blocks when m is large because those distributions put more weight on small list lengths. Note that the percentage of wasted space for one-cell blocks is fixed. The reason is that the wasted space is simply $s/(l+s)$, which is independent of m and $P(i)$.

We depict the suggested minimum number of attributes for each block size in Tables 1 and 2 for one-byte and two-byte pointers, respectively, assuming eight-byte event timestamps. For example, if the block has a one-byte pointer and the maximum number of attributes of a list is less than 17, then the best block size is one cell for all of the four distributions. If the maximum is between 42 and 124 and the distribution is gamma, a two-cell block is the best choice. According to these

8 Bytes Per Cell

distribution	*number of cells per block*					
	1	2	3	4	5	6
$N\ U$	1	17	49	98	161	248
γ	1	42	124	242	—	—
E	1	70	200	—	—	—

Table 1: Suggested minimum number of attributes for each block size for one-byte pointers

distribution	*number of cells per block*							
	1	2	3	4	5	6	7	8
$N\ U$	1	9	25	49	85	121	163	228
γ	1	23	63	123	203	—	—	—
E	1	36	104	202	—	—	—	—

Table 2: Suggested minimum number of attributes for each block size for two-byte pointers

Where N is normal; U is uniform; γ is gamma; E is exponential.

tables, we can easily determine the optimal block size for different combinations of distributions and list length, and analogously for the other combinations of parameters [7].

4 Summary

Supporting set-valued attributes complicates storage management. Because the cardinality of the sets can vary considerably, a linked list structure is preferable over preallocating space. The list structure that we studied consists of blocks, each of which contains a fixed number of cells, each of which stores an attribute. The storage structure described in Section 2 stores the blocks in the same page or in logically consecutive pages.

In order to achieve the best space utilization, the block size should be carefully chosen. A long list of small blocks wastes space on pointers; however, large blocks may contain substantial segments of unfilled space. In this paper, we showed how to obtain the optimal block size in terms of the number of attributes. The variables that affect the space utilization include the maximum number of attributes in a list, the cell size, the distribution of lists of different lengths, and the size of pointers. We provide two formulas that compute the average number of bytes needed per attribute value and the average number of bytes wasted per attribute value. We then determine the optimal block size according to the percentage of wasted space; the smallest percentage indicates the best space utilization.

We applied this model to obtain the optimal block size to minimize the space needed for sets of events or intervals in a temporal database. We showed that the longer the list is, a bigger optimal block size is obtained. However, the optimal block size is still rather small: in all cases we studied, the optimal block size is less than nine cells [7]. Another result is that the ratio of pointer size to the cell size is also an important indicator for choosing optimal block size. If the ratio is high, the wasted space on pointers is also high.

5 Acknowledgements

This research was partially supported by National Science Foundation Grant IRI-8902707 and partially by IBM Contract #1124. The comments of Curtis Dyreson and Michael Soo are appreciated.

6 Bibliography

[1] Ahn, I. "Towards an Implementation of Database Management Systems with Temporal Support," in *Proceeding of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 374–381.

[2] Clifford, J. and A. Croker. "The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans," in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1987, pp. 528–537.

[3] Dadam, P., K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor and G. Walch. "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies," in *Proceedings of the ACM International Conference on Management of Data*. Washington, DC: ACM Press, May 1986, pp. 356–367.

[4] Dyreson, C.E. and R.T. Snodgrass. "Timestamp Semantics and Representation." Technical Report TR 92-16. Department of Computer Science, University of Arizona. July 1992.

[5] Gadia, S.K. "A Homogeneous Relational Model and Query Languages for Temporal Databases." *ACM Transactions on Database Systems*, 13, No. 4 (1988), pp. 418–448.

[6] Gadia, S.K. "A seamless generic extension of SQL for querying temporal data." Technical Report TR-92-02. Computer Science Department, Iowa State University. May 1992.

[7] Hsu, S.H. and R.T. Snodgrass. "Optimal Block Size for Repeating Attributes." TempIS Technical Report No. 28. Department of Computer Science, University of Arizona. Dec. 1991.

[8] McKenzie, E. and R. Snodgrass. "Supporting Valid Time in an Historical Relational Algebra: Proofs and Extensions." Technical Report TR–91–15. Department of Computer Science, University of Arizona. Aug. 1991.

[9] McKenzie, E. and R. Snodgrass. "An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases." *ACM Computing Surveys*, 23, No. 4 (1991), pp. 501–543.

[10] Özsoyoğlu, G., Z.M. Özsoyoğlu and F. Mata. "A Language and a Physical Organization Technique for Summary Tables," in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 3–16.

[11] Özsoyoğlu, G., Z.M. Özsoyoğlu and V. Matos. "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions." *ACM Transactions on Database Systems*, 12, No. 4 (1987), pp. 566–592.

[12] Roth, Mark A., H.F. Korth and A. Silberschatz. "Extended Algebra and Calculus for Nested Relational Databases." *ACM Transactions on Database Systems*, 14, No. 4 (1988), pp. 389–417.

[13] Schek, H.-J., Scholl, M.H. "The Relational Model with Relation-valued Attributes." *Information Systems*, 11, No. 2 (1986), pp. 137–147.

[14] Stonebraker, M., E. Wong, P. Kreps, and G. Held. "The Design and Implementation of Ingres." *ACM Transactions on Database Systems*, 1, No. 3 (1976), pp. 189–222.