HASHING SCHEMES FOR EXTENDIBLE ARRAYS

Extended Abstract

Arnold L. Rosenberg
Larry J. Stockmeyer
Mathematical Sciences Department
IBM Watson Research Center
Yorktown Heights, New York

ABSTRACT: The use of hashing schemes for storing extendible arrays is investigated. It is shown that extendible hashing schemes whose worst-case access behavior is close to optimal must utilize storage inefficiently; conversely, hashing schemes that utilize storage too conservatively are inevitably poor in expected access time. If requirements on the utilization of storage are relaxed slightly, then one can find rather efficient extendible hashing schemes. Specifically, for any dimensionality of arrays, one can find extendible hashing schemes which at once utilize storage well [fewer than $2p$ storage locations need be set aside for storing arrays having p or fewer positions] and enjoy good access characteristics [expected access time is $O(1)$, and worst-case access time is $O(\log \log p)$ for p- or fewer-position arrays]. Moreover, at the cost of only an additive increase in access time, storage demands can be decreased to $(1+\varepsilon)p$ locations for arbitrary $\varepsilon>0$. In fact, if one will abide a more drastic degradation of access efficiency, one can lower storage demands to $p+o(p)$ locations.

1. INTRODUCTION

Conventional schemes for storing arrays are not readily extendible. For instance, in two dimensions, the familiar store-by-row scheme admits easy adjunction of new rows but only cumbersome appendage of columns. Such asymmetry in extendibility is not inevitable: there are computed-access schemes for storing arrays which are readily extended in any direction [1]. (An array storage scheme uses computed access if it assigns an address to an array position as a displacement from the address assigned to position $\langle 1,\cdots,1\rangle$, the displacement being computed from the position's coordinates.) However, extendibility in array realizations does not come without cost. The most definitive illustration of the cost of extendibility is the study in [2] of the efficiency of storage utilization by extendible array realizations. It is shown in that paper that every d-dimensional extendible array realization must, for every integer p, "spread" some array having p or fewer positions over at least $O(p\cdot(\log p)^{d-1})$ storage locations. This bound suggests that the costs of unbridled extendibility are prohibitive; and it indicates that ways of overcoming the bound should be sought. Two avenues to a smaller lower bound are available: one can abandon the demand for unbridled extendibility, or one can abandon the restriction to computed-access realizations. With regard to the former alternative, it is shown in [2] that, by restricting the patterns of expansions of arrays and by focusing only on arrays which conform to the restrictions, one can improve storage "spread" to $O(p)$, irrespective of the dimensionality of one's arrays. With respect to the abandonment of computed access, two alternatives arise. If one's array-processing algorithms are predominantly traversal-oriented, then linked realizations of arrays ("orthogonal lists" in Section 2.2.6 of [3]) are an attractive alternative to computed-access realizations. Such realizations are at once easily extended and conservative of storage; only cp locations are needed for a p-position array, where c is a small integer. If, on the other hand, one's algorithms tend to access arrays by successive independent probes, then hashing represents a viable alternative to computed access, provided that hashing schemes can be found which are readily extended, conservative of storage, and not overly expensive to access. It is our purpose in this note to study the cost of extendibility in hashing schemes for extendible arrays.

Summary of Results. In Section 2, we describe how hashing schemes can be used to store extendible arrays; we use a simple bucket model of hashing schemes, with buckets organized as balanced search trees. We also define there the measures of efficiency that are the subject of our study. Section 3 is devoted to discussing a fundamental trade-off. For any d-dimensional hashing scheme, the product of total space requirements and maximum bucket size for p-position arrays must grow at least as $p\cdot(\log p)^{d-1}$; this generalizes the result in [2] about computed-access realizations (which can be viewed as hashing schemes whose buckets never contain more than one array position). Various extreme cases are considered in Section 4. Hashing schemes whose worst-case access time does not grow with the size of the array being stored (computed-access realizations are included here) must utilize storage very poorly: total storage needed for p- or fewer-position d-dimensional arrays grows as $p\cdot(\log p)^{d-1}$. Conversely, hashing schemes that do not use successively more buckets as the arrays being stored grow suffer very inefficient access: expected access time for p- or fewer-position arrays grows no slower than $\log p$. Surprisingly, similarly inefficient access plagues

hashing schemes that do use ever more buckets but do so in a very conservative way: included here are gap-free schemes which insist that, whenever bucket $b>1$ is used in storing array A, so also must be bucket $b-1$; here also are those schemes whose total storage demand for p- or fewer-position arrays is $p+O(1)$ locations. In Section 5 we relax the extreme restrictions on storage allocation that plagued the schemes of Section 4, and we seek schemes that are good in both utilization of storage and time of access: we study hashing schemes that use $O(p)$ buckets for p-position arrays. We show that any such linear hashing scheme must have worst-case access time of at least $O(\log\log p)$ for p- or fewer-position arrays (a dramatic improvement, if attainable, over Section 4's lower bound of $\log p$ for expected access time). The main message of Section 5, and, indeed, of the paper, is that this lower bound is achievable along with constant expected access time! Specifically, for every dimensionality d there is a d-dimensional extendible linear array-hashing scheme which, when storing arrays with at most p positions, has total storage demands of fewer than $2p$ locations,* has worst-case access time $O(\log\log p)$, and has expected access time $c(d)$ $(c(2) \le 3)$. The exact multiple of p in the expression for total storage demands is of little import here since, at the cost of only an additive increase in (both expected and worst-case) access time, this multiple can be brought below $(1+\varepsilon)$ for arbitrarily small $\varepsilon>0$. Indeed one can bring total storage demands down to $p+o(p)$ if one is willing to suffer a more drastic degradation of efficiency of access (but no worse than $O(\log\log p)$).

Related Work. Knuth [3, Section 2.2.6] discusses both sequential (= computed-access) and linked allocation schemes for arrays; and he provides a comprehensive list of references to earlier work on such schemes. While scatter storage (= hashing) for arrays was undoubtedly considered and maybe even implemented in the 60's, we know of no reference predating [4] that discusses such schemes for storing arrays. The use of computed-access schemes for storing extendible arrays was considered first in [1]. The cost of such extendibility in terms of efficiency of storage utilization is studied in [2,5].

2. EXTENDIBLE ARRAY-HASHING SCHEMES

A. Notation

Let N denote the positive integers; and, for each $n\in N$, let N_n denote the set $N_n = \{1,\cdots,n\}$. For arbitrary $d\in N$, N^d is the set of d-tuples of positive integers; we ambiguously let $\varepsilon = \langle 1,\cdots,1\rangle$, relying on context to specify the dimensionality of any instance of ε. For $\pi\in N^d$, π_i $(i = 1,\cdots,d)$ is the i^{th} coordinate of π; thus, $\varepsilon_i = 1$ for all i. Finally, for any integer tuple π, $\Sigma(\pi) = \sum_i \pi_i$ and $\Pi(\pi) = \Pi_i \pi_i$.

*Each location is assumed capable of holding one datum, two pointers (to other locations in the same bucket), and a key used for searching.

B. Array Schemes and Hashing Schemes

In consonance with conventions for computed-access array realizations [1,3] and for the array-hashing functions of [4], we do not allow our hashing schemes to be data dependent; that is, we restrict attention to hashing functions that assign array positions to buckets using only a position's coordinates (and not its contents) to determine its bucket assignment. Accordingly, we can use as our notion of array the simple array schemes of [2,5] rather than any more elaborate representation of arrays.

(2.1) The d-dimensional array scheme (array, for short) of size $\langle n_1,\cdots,n_d\rangle$ $(d,n_1,\cdots,n_d\in N)$ is the set $A = N_{n_1}\times\cdots\times N_{n_d}$. Each $\pi\in A$ is called a position of A.

Graphically, we envisage an array scheme as being imbedded in the positive orthant of the appropriate dimensional space, with its positions laid on the lattice points. Extensions to arrays (adjoining new rows and/or columns, in two dimensions) can be viewed as extensions to the discrete rectangular solid formed by the imbedding. (Note that the "rectangularity" resides in A's being the cross-product of the coordinate sets N_{n_i}.)

We view our hashing schemes as operating in the following simple manner. There is a bucket function β that assigns each position of the array A being stored to a bucket; for simplicity, we view the buckets as being addressed by positive integers, so β is a function from A into N. Of course several array positions will usually reside in the same bucket (or else β would be a computed-access realization as in [1,2]); the collisions caused by β's being many-to-one are resolved by organizing all the residents of each bucket into a balanced search tree [7, Section 6.2.3] by sorting them according to some key such as the lexicographic ordering of their coordinates. (By so exploiting the order inherent in the keys, one can appreciably reduce both expected and worst-case access time in external [= bucket-employing] hashing schemes. It is not obvious that internal hashing schemes can benefit from such exploitation, but Amble and Knuth [8] show that internal schemes can be so improved.) Thus we imagine the storage available to our hashing scheme to be divided into a collection of bucket addresses and a pool of chainable storage for the trees. See Figure 1.

The specific problem we address in this paper is to gauge the cost of extendibility in array-hashing schemes. We lead up to our formal setting by considering the following scenario. We want to devise a bucket function β that will store any array of a given dimensionality d. We intend to use it as follows. We construct a system (say, a compiler or operating system) that includes an array-storage allocator based on β. As a customer (say, a program or a user) declares that his variable X will range over d-dimensional arrays having p or fewer positions, the system sets aside for X a block of $m+n$ locations, call them $\{1,\cdots,m, m+1,\cdots,m+n\}$; here $m = \max\{\beta(\pi) \mid \pi$ is a position of an array having p or fewer positions$\}$, and n is the largest number of locations the system will ever need for constructing trees.

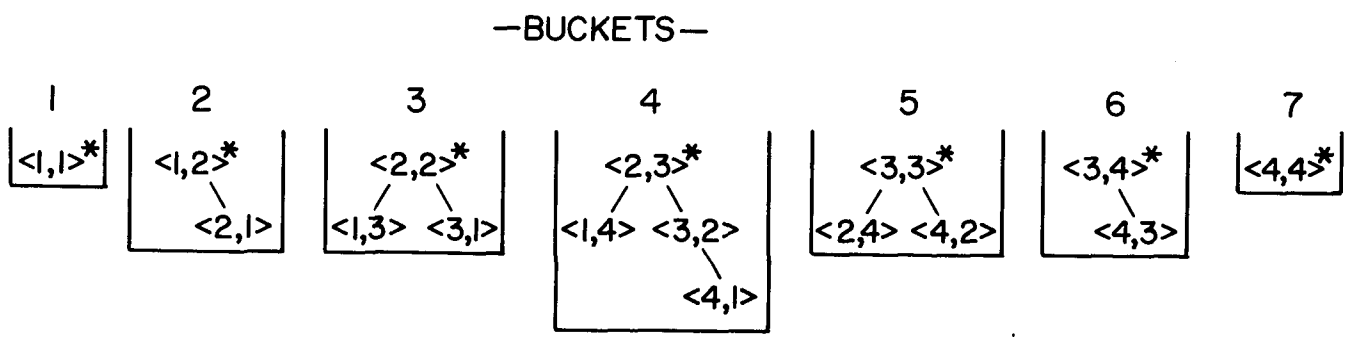

Figure 1

The Array Scheme $N_4 \times N_4$ Stored by the Hashing Scheme $\underset{\sim}{h}(i,j) = i+j-1$.
Starred position (= roots of trees) reside in "bucket" storage; the others reside in chainable storage

(Precisely, $n = \max\{q-\#\underset{\sim}{h}(A) \mid A$ is a $(q \le p)$-position array$\}$.) One can now store by hashing any array in X's domain, using the storage set aside by the system and using $\underset{\sim}{h}$ as the bucket function. The "extendibility" in the described situation resides in the facts that (1) the same function $\underset{\sim}{h}$ is used as the bucket function irrespective of what integer p appears in the declaration binding X; and (2) the scheme generated by the allocator will properly store any array in the domain of the variable X. The described scheme exhibits two types of stability: first, the computation required to store and/or access array positions does not change as the array being processed is extended; second, bucket assignments never change because of changes to the stored array. In fact, even if the declared value of p is changed dynamically, relatively minor rechaining within buckets can restore a stable configuration (especially if the pool of chainable storage is accessed from the high-address side). It is such stability under alterations to the stored array that we equate with "extendibility" of the hashing scheme. An alternative to our scenario might depict a different function $\underset{\sim}{h}$ being used for each declared bound p. It is hard to see how any such system would exhibit the uniformity of scheme-generation or of extension and the algorithmic simplicity enjoyed by our system. The major question raised by this alternative is how much our uniformity costs. Theorems 5.3 and 5.4, our main results, suggest that the schemes generated by our imaginary system can attain a practical level of efficiency in both time of access and utilization of storage.

For simplicity, we henceforth identify a hashing scheme with its bucket function.

(2.2) A d-dimensional extendible (array-) hashing scheme is a total function $\underset{\sim}{h}: N^d \to N$ such that $\underset{\sim}{h}(\varepsilon) = 1$.

The normalizing assumption "$\underset{\sim}{h}(\varepsilon) = 1$" simplifies certain computations in what follows but is not indispensable. (Cf. (2.4(a)) in the absence of this assumption.) Henceforth we shall study only $(d \ge 2)$-dimensional extendible hashing schemes.

C. Measures of Efficiency

The efficiency of any hashing scheme is measured in terms of the scheme's demands on two resources, time and storage. However, deciding on what to measure is only half the battle; we must decide on how to estimate the consumption of these resources.

Access Time. We view the cost of computing a bucket function $\underset{\sim}{h}$ as being negligible compared to the cost of searching even a modest size tree. (To make this assumption reasonable, we have tried to use only very simple bucket functions when establishing upper bounds.) Accordingly, we gauge the cost of accessing position π of array scheme A when A is stored by $\underset{\sim}{h}$ as being the log of the population of π's bucket; precisely,

$$\underset{\underset{\sim}{h}}{\text{Access}}(\pi;A) = \lceil \log(\#(A \cap \underset{\sim}{h}^{-1}(\underset{\sim}{h}(\pi))) + 1) \rceil .$$

The justification for the logarithmic cost is that each bucket is organized into a balanced search tree. Positions added to an array via extensions are to be stored at the time of their first access and so stay within our logarithmic cost. The base of the logarithm used is immaterial in almost all of our results because of their "big-Oh" assertions. In the few cases (e.g., Theorems 5.3 and 5.4) where constants are crucial to the development, we use the base 2 logarithm appropriately dilated. (See Theorem A of [7, Section 6.2.3].)

Now that we have a way of gauging an extendible hashing scheme's accessing characteristics on a single array, we can formulate the first two functions of interest, which reflect a scheme's access behavior on arrays having no more than p positions. Focus on an extendible hashing scheme $\underset{\sim}{h}$.

(2.3)(a) Worst-Case Access Time

$\alpha(p;\underset{\sim}{h})$ = (the worst-case time needed by $\underset{\sim}{h}$ to access a position of an array having at most p positions)

$= \max\{\underset{\underset{\sim}{h}}{\text{Access}}(\pi;A) \mid A$ has at most p positions, and $\pi \in A\}$.

(b) Worst Expected Access Time

$\bar{\alpha}(p;\underset{\sim}{h})$ = (the average time needed by $\underset{\sim}{h}$ to access a position of that p- or fewer-position array having worst expected access time under $\underset{\sim}{h}$)

$= \max\{(1/q)\sum_{\pi \in A} \underset{\underset{\sim}{h}}{\text{Access}}(\pi,A) \mid$ A has $q \le p$ positions$\}$.

Storage Requirements. Our view of extendible hashing schemes demands that we segregate the storage used for chaining (for building the trees that make up the bucket interiors) from the storage used for bucket headers (= the roots of those trees). If we did not partition storage in this way, a series of expansions of a stored array might demand for use as the header of bucket k a location already being used in the tree of bucket $\ell \neq k$. Such partitioning also is consonant with the scenario motivating our view of extendible hashing schemes. Finally, this division of storage makes the analysis of proposed hashing schemes or classes of hashing schemes immeasurably simpler than it would be if we used some more sophisticated method of collision resolution (e.g., linear probing or secondary clustering [8]). We formalize this division of storage by our method of measuring efficiency of storage utilization. Again, let h be an extendible hashing scheme.

(2.4)(a) Bucket Storage Requirements

$\beta(p;h)$ = (the number of storage locations over which h "spreads" the bucket headers [roots of search trees] when storing arrays having at most p positions)

$= \max\{h(\pi) \mid \Pi(\pi) \leq p\}$.

Note: π is a position of a p- or fewer-position array iff $\Pi(\pi) \leq p$.

(b) Chainable Storage Requirements

$\sigma(p;h)$ = (the amount of storage that is needed to store the bucket interiors [the nonroot nodes of the search trees] when storing arrays having at most p positions)

$= \max\{q - \#h(A) \mid A \text{ has } q \leq p \text{ positions}\}$.

(c) Total Storage Requirements

$\tau(p;h)$ = (the total amount of storage that must be set aside to store arrays having no more than p positions)

$= \beta(p;h) + \sigma(p;h)$.

The aim of this paper is to discover relationships among the five efficiency measures of (2.3) and (2.4). In stating results, we shall fix the hashing scheme h by either quantification or instantiation; we shall then be free to consider our five functions as functions of p alone, as in "For all hashing schemes h, $\sigma(p;h) = 0(p)$," or "$\overline{\alpha}(p;h_0) = 0(1)$." [As an aside, the former assertion is obviously true; and we shall present a scheme in Section 5 for which the latter assertion is true.]

The following lemma will be used in our investigation.

(2.5) For all dimensionalities $d \in N$ and $p \in N$,

$S_d(p) = \{\pi \in N^d \mid \Pi(\pi) \leq p\}$.

Lemma 2.1 [2]. (a) A position $\pi \in N^d$ resides in $S_d(p)$ iff it is a position of some array having p or fewer positions. (b) For all $d \in N-\{1\}$, there exist constants $k, \ell > 0$ such that

$k \cdot p \cdot (\log p)^{d-1} < \#S_d(p) < \ell \cdot p \cdot (\log p)^{d-1}$

for all $p \in N-\{1\}$.

3. A FUNDAMENTAL TRADEOFF

As a vehicle for introducing the reader to extendible hashing schemes, and as a prologue to a challenging research problem, we present a fundamental tradeoff in storage use by extendible hashing schemes. The tradeoff involves the total storage function $\tau(p;h)$ and the function now defined.

(3.1) Size$(p;h)$ = (the maximum size any bucket attains as h stores p- or fewer-position arrays)

$= \max\{\#(A \cap h^{-1}(h(\pi))) \mid A \text{ has } p \text{ or fewer positions, and } \pi \in A\}$.

We prepare the reader for the tradeoff statement via some examples.

(3.2) A Computed-Access Realization. If $h:N^d \to N$ is one-to-one, then h is an extendible array realization in the sense of [1,2,5]. It follows from Lemma 2.1, then, that $p \cdot (\log p)^{d-1} = 0(\tau(p;h))$; and it is tautologous that Size$(p;h) \equiv 1$.

(3.3) A Linked-Allocation Scheme. If $h(\pi) \equiv 1$, then h is a pure chaining scheme. It is immediate, then, that $\tau(p;h) =$ Size$(p;h) = p$.

(3.4) Define $h:N^d \to N$ by $h(\pi) = \Sigma(\pi)-d+1$; the two-dimensional ($d=2$) version of h is illustrated in Figure 1. It is not difficult to verify that $\beta(p;h) = p$, and $\sigma(p;h) = p-dp^{1/d}+d-1$, so that $\tau(p;h) = 2p-dp^{1/d}+d-1$. It is also easily seen that Size$(p;h) = \binom{n+d-1}{d-1}$ where $n = p^{1/d}$, so that $p^{1-1/d} = 0(\text{Size}(p;h))$.

(3.5) Define $h:N^d \to N$ by $h(\pi) = \Pi(\pi)$. Obviously, $\beta(p;h) = p$ so that $p \leq \tau(p;h) < 2p$. (By definition $\sigma(p;h) < p$ for any h.) Moreover, by considering the cases when p is a power of 2, one verifies easily that $(\log p)^{d-1} = 0(\text{Size}(p;h))$.

Note that in all of the examples just presented, the product $\tau(p;h) \cdot$ Size$(p;h)$ grows (in order of magnitude) at least as fast as $p \cdot (\log p)^{d-1}$. It is our conjecture that this is a universal phenomenon; however, we are able to establish only an "infinitely often" version of this conjecture.

Theorem 3.1. Let $h:N^d \to N$ be an extendible hashing scheme. There is a constant $c>0$ such that, for infinitely many $p \in N$,

$\tau(p;h) \cdot \text{Size}(p;h) > c \cdot p \cdot (\log p)^{d-1}$.

4. EXTREME CASES

It is not uncommon that a seemingly innocuous restriction on one of our complexity measures forces another measure to behave as badly as possible (i.e., to grow as fast as one could ever force it to). In this section we present a number of such situations. To lend the reader a point of reference, we note

minimax lower bounds on three of our complexity measures. (Recall that, by definition, $\sigma(p;\mathfrak{h})$ cannot be forced to exceed p.)

(4.1) A bound on β. In [2] it is proved that, if $\mathfrak{h}:N^d \to N$ is one-to-one, then $p\cdot(\log p)^{d-1} = O(\beta(p;\mathfrak{h}))$. Moreover, for each $d\in N$, there is a one-to-one scheme $\mathfrak{h}:N^d \to N$ for which $\beta(p;\mathfrak{h}) = O(p\cdot(\log p)^{d-1})$.

(4.2) Bounds on α and $\bar{\alpha}$. By its very definition (cf. (3.1)), $\mathrm{Size}(p;\mathfrak{h})$ can obviously not exceed p. It follows that $\alpha(p;\mathfrak{h}) = O(\log p)$ for any hashing scheme $\mathfrak{h}$; a fortiori, $\bar{\alpha}(p;\mathfrak{h}) = O(\log p)$ for all $\mathfrak{h}$. If $\mathfrak{h}(\pi) \equiv 1$ so that $\mathfrak{h}$ is a linked-allocation scheme, then all three of these worst cases are realized simultaneously.

We begin our survey of harmful restrictions by exhibiting two restrictions, either of which guarantees poor storage utilization.

Theorem 4.1. Let $\mathfrak{h}$ be a d-dimensional extendible hashing scheme. If either $\alpha(p;\mathfrak{h}) = O(1)$ or $\sigma(p;\mathfrak{h}) = O(1)$, then $p\cdot(\log p)^{d-1} = O(\beta(p;\mathfrak{h}))$.

Hint: The proof hinges on the fact that the array scheme $(N_p)^d$ contains the union $S_d(p)$ of all p- or fewer-position arrays. See Figure 2 and Lemma 2.1.

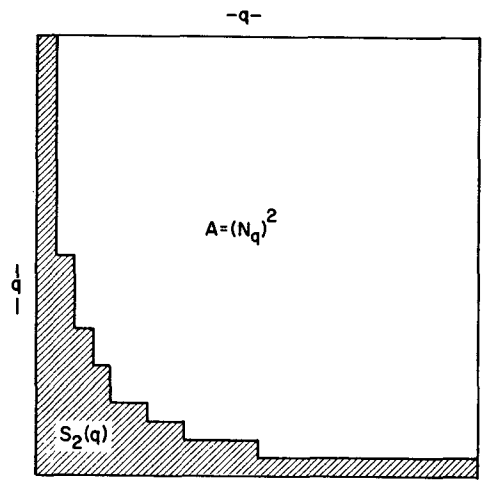

Figure 2

Every q-position array is a subset of the $q \times q\times\cdots\times q$ array

It is worth noting that Theorem 4.1 cannot be strengthened by replacing α by $\bar{\alpha}$: in Section 5, we show that a constant $\bar{\alpha}$ can coexist with a β that is linear in p.

Expected access time must be poor for schemes that do not use increasing numbers of buckets as array size grows. Such schemes would typically use linked allocation or hashing by division (i.e., modular arithmetic as in $\mathfrak{h}(\pi) = \Sigma(\pi) \pmod m$).

Theorem 4.2. Let $\mathfrak{h}$ be an extendible hashing scheme. If $\beta(p;\mathfrak{h}) = O(1)$, then $\log p = O(\bar{\alpha}(p;\mathfrak{h}))$.

Hint: Such schemes behave approximately as linked-allocation schemes (cf. (4.2)).

Finally, we note two restrictions on efficiency of storage management which, like bounded bucket demands, guarantee poor access time. The first restriction is that $\mathfrak{h}$ assign buckets to arrays in a gap-free manner; that is, for every array A, there is an integer k for which $\mathfrak{h}(A) = \{1,\cdots,k\}$. (In other, words, bucket $\ell-1$ is used whenever bucket ℓ is.) The second restriction, which includes special cases of bounded bucket demands, is that total storage demands grow slowly, for instance, $\tau(p;\mathfrak{h}) = p + O(p^{1/2-\varepsilon})$. The fact that these restrictions give rise to worst-possible access time (in the sense discussed earlier) is somewhat more surprising than the fact that the bounded bucket restriction does. The desired results emerge as corollaries of the following theorem.

(4.3) Let $\mathfrak{h}$ be a d-dimensional extendible hashing scheme, and let $f:N \to N\cup\{0\}$ be a nondecreasing function. We say that f bucket-bounds $\mathfrak{h}$ if, for all $p\in N$ and all array schemes $A \subset N^d$ having p positions,

$$\max \mathfrak{h}(A) - \#\mathfrak{h}(A) \le f(p).$$

Theorem 4.3. Let $\mathfrak{h}$ be a d-dimensional extendible hashing scheme. If the function f bucket-bounds $\mathfrak{h}$, then

$$(1-1/d)\cdot\log p - \log(f(p)+1) = O(\bar{\alpha}(p;\mathfrak{h})).$$

Hint: If f bucket-bounds $\mathfrak{h}$, then

$$\mathfrak{h}(\pi) \le (\Sigma(\pi)-d+1)(f(\Pi(\pi))+1)$$

[by induction on $\Sigma(\pi)$].

Corollary 4.1. If the extendible hashing scheme $\mathfrak{h}$ is gap-free, then $\log p = O(\bar{\alpha}(p;\mathfrak{h}))$.

Hint: $\mathfrak{h}$ is gap-free iff it is bucket-bounded by $f(p) \equiv 0$.

Corollary 4.2. Let $\mathfrak{h}$ be a d-dimensional extendible hashing scheme. If $\tau(p;\mathfrak{h}) \le p + f(p)$ where $f:N \to N\cup\{0\}$ is nondecreasing, then

$$(1-1/d)\cdot\log p - \log(f(p)+1) = O(\bar{\alpha}(p;\mathfrak{h})).$$

Hint: The hypothesized bound on τ's rate of growth implies that f bucket-bounds $\mathfrak{h}$.

The interest of Corollary 4.2 for the material in this section is that it implies that a very slowly-growing τ can signal very inefficient access behavior. For instance, if $\tau(p;\mathfrak{h}) = p+O(p^c)$ for some $c < 1-1/d$, then $\log p = O(\bar{\alpha}(p;\mathfrak{h}))$.

5. LINEARLY-GROWING BUCKET SPACE

The results in Section 4 indicate that for us even to hope to find an extendible hashing scheme that is efficient in both speed of access and utilization of storage, we must relax our stringent demands on the bucket function. Fortunately, we needn't relax our demands so much as to admit fast-

growing bucket functions: one can find efficient hashing schemes h with $\beta(p;h) = p$. For the sake of generality that will not occasion any technical encumbrance, we study here hashing schemes with $\beta(p;h) = O(p)$ (i.e., we allow h to use more than p buckets); we call such h's linear hashing schemes. (Restricting attention to such schemes is really only a mild relaxation of our earlier demands since $\beta(p;h) = p$ even for the d-dimensional gap-free scheme $h(\pi) = \Sigma(\pi) - d + 1$.) The reader who balks at our allowing $\beta(p;h)$ to exceed p should note that, for any integer k, the scheme $h^{(k)}(\pi) = \lceil h(\pi)/k \rceil$ has the following properties. (1) $\beta(p;h^{(k)}) = \lceil \beta(p;h)/k \rceil$; (2) $h^{(k)}$ is more conservative of total storage than h; (3) $h^{(k)}$ has only slightly worse access time than h (see Theorem 5.4). Therefore, we shall not be using the "big Oh" to hide large multiples that can refute our claims of efficiency.

A. Lower Bounds on Performance

The restriction to linearly-growing bucket space imposes floors on certain other efficiency criteria. We present two such lower bounds.

First we note that worst-case access time for linear hashing schemes must grow as log log p. Two points merit note. First, this bound is dramatically lower than the log p bounds on access time established in Theorem 4.2 and Corollaries 4.1 and 4.2 (this last with $f(p) = O(p^c)$ for $c<1-1/d$). Second, whereas the cited results bound expected access time, the present result deals with worst-case access time.

Theorem 5.1. For any linear extendible hashing scheme h, $\log \log p = O(\alpha(p;h))$.

Hint: The array scheme $(N_p)^d$ contains the set $S_d(p)$ (cf. Lemma 2.1).

Our second lower bound asserts that σ must grow linearly with p whenever β does. The import of this result is that, if one wishes to replace a given linear scheme h (for which $\tau(p;h) = O(p)$ automatically) by a linear scheme h' for which $\tau(p;h') = p + o(p)$, one cannot hope to do it simply by lowering interior bucket demands so that $\sigma(p;h') = o(p)$. The desired savings in storage can come only by constructing h' so as to assure that β grows slowly. (Cf. the discussion in the latter half of Section 5C on lowering storage demands.)

Theorem 5.2. For any linear extendible hashing scheme h, $p = O(\sigma(p;h))$. Specifically, if $\beta(p;h) \le c \cdot p$ for almost all p, then $\sigma(p,h) \ge \delta \cdot p$ for almost all p, where $\delta = \exp(-(c+2))$.

B. An Efficient Hashing Scheme

(5.1) For each dimensionality $d \in N-\{1\}$, define the d-dimensional extendible hashing scheme $h_d: N^d \to N$ by the following recursion.

(a) For $\pi \in N^2$, $h_2(\pi) = \pi_2 + (\pi_1 - 1)2^{\lfloor \log \pi_2 \rfloor}$.

(b) For $\pi \in N^d$, $d > 2$,

$$h_d(\pi) = h_2(\pi_1, h_{d-1}(\pi_2, \cdots, \pi_d)).$$

Informally, one computes $h_d(\pi)$ by taking the minimum-length binary representation of $\pi_1, \cdots, \pi_d$, removing the leading 1 from all but that of π_1, and concatenating the remaining strings in the appropriate order; thus the binary representation of $h_d(\pi)$ assumes the form $1\xi_1\xi_2\cdots\xi_d$, where $1\xi_j$ is the binary representation of π_j.

1	2	3	4	5	6	7	8	9	10
2	4	5	8	9	10	11	16	17	18
3	6	7	12	13	14	15	24	25	26
4	8	9	16	17	18	19	32	33	34
5	10	11	20	21	22	23	40	41	42
6	12	13	24	25	26	27	48	49	50
7	14	15	28	29	30	31	56	57	58
8	16	17	32	33	34	35	64	65	66
9	18	19	36	37	38	39	72	73	74
10	20	21	40	41	42	43	80	81	82

Figure 3

A Schematic View of $h_2((N_{10})^2)$.

Theorem 5.3. For each dimensionaltiy $d \in N-\{1\}$, the extendible hashing scheme h_d has the following properties. (a) $\beta(p;h_d) = p$. (b) There is a constant $c > 0$ such that $cp < \sigma(p;h_d) < p$. (c) $\alpha(p;h_d) \le (d-1) \log \log p + O(1)$. (d) $\bar{\alpha}(p;h_d) = O(1)$; in particular $\bar{\alpha}(p;h_2) < 8.5$.

Proof sketch. (a) First, $h_d(N_p \times \{1\}^{d-1}) = N_p$, so $\beta(p;h_d) \ge p$. On the other hand, $h_2(\pi) \le \pi_2 + (\pi_1 - 1) \cdot \pi_2 = \pi_1 \cdot \pi_2$ for all $\pi \in N^2$; this generalizes by induction to the assertion that $h_d(\pi) \le \Pi(\pi)$ for all $\pi \in N^d$.

(b) The upper bound on σ is immediate by definition. The lower bound follows directly from Theorem 5.2 in conjunction with part (a).

(c) Say that 1ω ($\omega \in \{0,1\}^*$) is the binary representation of $h_d(\pi)$ for some $\pi \in S_d(p)$. By part (a), $h_d(\pi) \le p$. There are, therefore, at most $\lceil \log(p+1) \rceil^{d-1}$ ways that the string ω can be broken (or parsed) into the form $\omega = \xi_1 \cdots \xi_d$, each $\xi_i \in \{0,1\}^*$. In other words, when h_d stores an array having p or fewer positions, no bucket gets more than $\lceil \log(p+1) \rceil^{d-1}$ positions. Part (c) is now immediate by definition of Access and of α.

(d) We consider only the case $d = 2$. The analysis of the general case is similar. Say that h_2 stores the array scheme A of size $\langle m,n \rangle$ having $p = mn$ position. For $z \in N$, let $\ell(z) = \lfloor \log z \rfloor + 1$ denote the length of the binary representation of z. Let $t = \ell(m) + \ell(n)$.

View the computation of h_2 as concatenation of binary representations as described in (5.1). It is clear that if bucket z is used when h_2 stores A, then $\ell(z) \le t-1$. Consider any bucket z with $\ell(z) = t-1$. It is not hard to see that there is at most one way z can be "parsed" (cf. part (c)) into integers π_1 and π_2 with $\pi_1 \le m$ and $\pi_2 \le n$; therefore, the population of such buckets is at most 1. There are at most 2^{t-2} such buckets. Similarly, the 2^{t-3} buckets z with $\ell(z) = t-2$ can each be parsed in at most two ways, the 2^{t-4} with $\ell(z) = t-3$ in at most three ways, and so on. This yields the bound

$$\overline{\alpha}(p;h_2) \le (1/p) \sum_{k=1}^{t+1} k \cdot \lceil \log(k+1) \rceil \cdot 2^{t-k-1}.$$

Noting that $p \ge 2^{t-2}$, this sum can be bounded above by 8.5.

Remark. A more careful analysis in part (d) proves that $\overline{\alpha}(p;h_2) < 3$; and computer simulation reveals that $\overline{\alpha}(2500;h_2) > 2$. Of course, the average access time in practice will depend on $\overline{\alpha}$ dilated by a constant that reflects the particular data structure used to store buckets and the particular algorithm used to access positions within buckets; see [7, Section 6.2].

Even though our definition of Access assumes some variant of balanced search tree as the basic data structure, it is interesting to note that the average access cost of h_2 remains bounded (by 8) if one stores buckets in linear lists, charging average cost $(k+1)/2$ to access a position in a bucket of size k. Linear lists may be more attractive for "small" arrays, since the accessing algorithm is simpler for linear lists than for trees. However, the worst-case access time of h_2 becomes roughly $\log p$ if linear lists are used.

C. Tradeoffs between Access Time and Storage Utilization

Theorem 5.3 does not specify precise values for $\overline{\alpha}(p;h_d)$ or $\tau(p;h_d)$, since by means of (often easily-effected) changes to a hashing scheme, one can trade off access time and storage utilization: at the cost of increasing $\beta(p;h)$ by a factor of k, one can decrease $\overline{\alpha}(p;h)$ by roughly $\log k$; conversely, at the cost of increasing $\overline{\alpha}(p;h)$ by $\log k$, one can decrease $\tau(p;h) - p$ by the factor k.

Lowering access time. Any change to h that causes a k-fold decrease in the size of buckets will reduce $\overline{\alpha}(p;h)$ by $\log k$, albeit at the cost of a k-fold increase in the number of buckets used. It is difficult to discuss such alterations in the abstract, since the effectiveness of a given bucket-shrinking ploy appears to depend in an essential way on the particular h in question. We limit our discussion, therefore, to a simple example in two dimensions.

Let $h(\pi) = \pi_1+\pi_2-1$; this is a gap-free hashing scheme (cf. Corollary 4.1). Define $h':N^2 \to N$ by $h'(\pi) = k \cdot h(\pi) - k + [\pi_1 \pmod k]$. Now compare the behaviors of both h and h' on an array $A \subset N^2$. If h assigns m array positions to a given bucket i, then h' assigns at most $\lceil m/k \rceil$ positions to each of buckets $ki-k,\cdots,ki-1$. The cumulative time required to access these positions is, therefore, at most $m\lceil \log(m+1) \rceil$ for h and $m\lceil \log(\lceil m/k \rceil+1) \rceil$ for h'. Since $\log(\lceil m/k \rceil+1) \le \log(m/k+2) < \log(m/k) + 1$ for almost all m, our estimates on the time to access all of bucket i lead us to conclude that the average access time for A under scheme h exceeds the average access time under h' by at least $\log k - 2$ providing only that A has at least 4k rows and columns. Since $\overline{\alpha}$ is monotonic in p, we see that $\overline{\alpha}(p;h')$ approaches $\overline{\alpha}(p;h) - \log k$ for sufficiently large p.

Lowering storage demands. We can discuss techniques for improving storage utilization with somewhat more authority than those for improving access time, for here we find general techniques that work for all h.

(5.2) Let h be an extendible hashing scheme. For $k \in N$, $h^{(k)}$ is the extendible hashing scheme defined by $h^{(k)}(\pi) = \lceil h(\pi)/k \rceil$.

If the linear hashing scheme h has total storage demand $\tau(p;h) = (1+c)p$ for $c > 0$, then the (again linear) scheme $h^{(k)}$ has demand not exceeding $(1+c/k)p + 1$. Therefore, by choosing k sufficiently large, one can reduce storage demand below $(1+\varepsilon)p$ for any $\varepsilon > 0$ by replacing h with $h^{(k)}$. Such a replacement has an adverse effect on access efficiency, but only an additive one.

Theorem 5.4. Let h be an extendible hashing scheme. (a) $\tau(p;h^{(k)})-p \le 1+(\tau(p;h)-p)/k$.
(b) $\overline{\alpha}(p;h^{(k)}) \le \overline{\alpha}(p;h) + \log k + 1$.

Hint: Consider how $h^{(k)}$ coalesces h's bucket assignments.

The following important corollary of Theorem 5.4 is immediate.

Corollary 5.1. If $\overline{\alpha}(p;h) = O(1)$, then the same is true of $\overline{\alpha}(p;h^{(k)})$ for any k.

We now have linear hashing schemes whose storage requirements grow as $(1+\varepsilon)p$ and whose expected and worst-case time grow as a constant and $\log \log p$, respectively. (The last remark is immediate and left to the reader for verification.) The only material improvement one might hope for is to reduce storage demands to $p+o(p)$ without excessive deterioration of access time. We have very little to report with regard to such improvements.

Proposition 5.1. Let h be a linear hashing scheme. The hashing scheme h^* defined by $h^*(\pi) = \lceil h(\pi)/\log \Pi(\pi) \rceil$ has the following properties.
(a) $\tau(p;h^*)-p \le 1+(\tau(p;h)-p)/\log p$.
(b) $\alpha(p;h^*) = O(\alpha(p;h) + \log \log p)$.

If one is willing to use only worst-case behavior to measure efficiency of access, Proposition 5.1

yields a technique for attaining both good storage utilization and good access characteristics. However, this technique inevitably destroys efficient expected access time.

Proposition 5.2. For any extendible hashing scheme $\mathcal{h}$, $\overline{\alpha}(p;\mathcal{h}) \geq \log(p/\beta(p;\mathcal{h}))$.

This proposition indicates that, for any linear $\mathcal{h}$, $\log \log p = O(\overline{\alpha}(p;\mathcal{h}^*))$ -- since, for some $c > 0$ $\beta(p;\mathcal{h}^*) \leq \lceil cp/\log p \rceil$.

We are thus left with the following open question.

Problem. Does there exist an extendible hashing scheme $\mathcal{h}$ such that $\tau(p;\mathcal{h}) = p+o(p)$ and $\overline{\alpha}(p;\mathcal{h}) = O(1)$?

Our results do not resolve this issue, but they do narrow down the search materially. Specifically we know that the desired $\mathcal{h}$ (if it exists) must satisfy the following efficiency bounds.

(5.5) (a) For some $k,k' > 0$, for almost all p,

$$kp \leq \beta(p;\mathcal{h}) \leq k'p.$$

(b) For some $\ell > 0$, for almost all p,

$$\ell p \leq \sigma(p;\mathcal{h}) < p.$$

The upper bound on β follows from our desire to keep storage demands low; the lower bound follows from Proposition 5.2 and our desire to keep expected access time down. The upper bound on σ follows by definition; the lower bound on σ follows from Theorem 5.2 in the presence of the linearity of $\mathcal{h}$, i.e., (5.5(a)).

Another remaining challenge, of a materially different nature, is to "fine tune" the results in this section in an actual computer environment, that is, select the necessary data representations, investigate the cost of bookkeeping details that are not visible in any abstract investigation, and study the real effect of the tradeoffs mentioned. We suspect, on the basis of Theorems 5.3 and 5.4 and the remarks on lowering $\overline{\alpha}$, that extendible hashing schemes that attain a practical level of efficiency exist.

ACKNOWLEDGMENT

We are grateful to Shmuel Winograd for constructive suggestions in the early stages of this research.

REFERENCES

1. A. L. Rosenberg, Allocating storage for extendible arrays. JACM, 21 (1974) 652-670.

2. A. L. Rosenberg, Managing storage for extendible arrays. SIAM J. Comput., to appear.

3. D. E. Knuth, The Art of Computer Programming I: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

4. E.v.d.S. de Villiers and L. B. Wilson, Hash coding methods for sparse matrices. Tech. Rpt. 45, Univ. of Newcastle-upon-Tyne (Computing Lab.), May, 1973. See also, Hashing the subscripts of a sparse matrix. BIT, 14 (1974) 347-358.

5. L. J. Stockmeyer, Extendible array realizations with additive traversal. IBM Report RC-4578, 1973.

6. A. L. Rosenberg, Computed access in ragged arrays. in Information Processing 74 (J. Rosenfeld, ed.) North-Holland, Amsterdam, 1974, pp. 642-646.

7. D. E. Knuth, The Art of Computer Programming III: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

8. O. Amble and D. E. Knuth, Ordered hash tables. Comput. J., 17 (1974) 135-142.