Batched Searching of Sequential and Tree Structured Files

BEN SHNEIDERMAN AND VICTOR GOODMAN
Indiana University

The technique of batching searches has been ignored in the context of disk based online data retrieval systems. This paper suggests that batching be reconsidered for such systems since the potential reduction in processor demand may actually reduce response time. An analysis with sample numerical results and algorithms is presented.

Key Words and Phrases: batched searching, information retrieval, database systems, data structures, sequential files, tree structures, B-trees, indexes, file management
CR Categories: 3.70, 3.74

1. INTRODUCTION

The technique of performing search and update requests on a file of records predates the existence of computers. Librarians and commercial tabulating equipment record keepers recognized the obvious advantages of accumulating a number of requests before performing a single pass through the data records during which all requests were satisfied. It was natural for programmers of business data processing applications to adopt this strategy for the periodic updating of a key sorted "old master tape file" from a key sorted "transaction tape file" to produce a "new master tape file." The impossibility of performing insertions and the difficulty of performing in-place updates in sequential tape files only reinforced the utility of the batch processing concept.

With the proliferation of disk based online systems and the demand for real-time response for individual retrievals, the use of batch processing has decreased. A few texts provide a general discussion of batching [1, 2], and Knuth's encyclopedic review of search techniques [3] gives a cursory treatment. A few other sources recognize the importance of batching [4–6], but there is no precise analysis of the benefits of batching. Early research assumed processing was always performed in batches while later research ignored this method altogether.

We present arguments for the reconsideration of batching, even in online real-time retrieval systems, and support these arguments with algorithms and mathematical analysis which demonstrate the practicality of batched retrievals in certain circumstances.

Authors' addresses: B. Shneiderman, Department of Information Systems Management, University of Maryland, College Park, MD 20742; V. Goodman, Department of Mathematics, Indiana University, Bloomington, IN 47401.

Developers of online retrieval systems argued that short response time was the primary goal and that each query should be dealt with as quickly as possible, even at the expense of additional memory requirements and inefficient retrieval techniques. This strategy eventually becomes counterproductive since the inefficiency of individual retrievals places a higher demand on the processor (or processors) and thereby potentially increases the response time for all queries. By using a somewhat more elapsed-time consuming but more machine efficient algorithm, i.e. batching queries, it may be possible to reduce the average response time for the entire batch of queries. As a crude example, imagine that the time to respond to a single query for a serial processing algorithm is 1 sec. Should ten queries arrive at a processor within the first second, the last of the responses will be removed from the queue and completed after 10 sec for an average retrieval time of approximately 5.5 sec. If the processor waits, say 1 sec, ten queries have been made before beginning the retrievals, and if the batch retrieval time is only 3 sec, then all the responses will be completed no later than 4 sec after the queries were entered, thus producing an improved average response time and reducing the demand on the processor.

The validity of this hypothetical situation must be ascertained, but it seems reasonable that if processor costs per query can be reduced system performance will be improved. The strength of our argument is increased by the appearance of very large online mass storage systems (10^9–10^{12} bytes) which have relatively long access times of up to 15 sec. If the number of accesses can be reduced, then there is a possibility of a substantial reduction in response time and an improvement in system performance.

While the hardware configuration strongly influences system characteristics, the data structures also play a central role. Of course, batching is most advantageous in sequential files, which are not the usual fundamental structure for online systems. However, sequential searches are often performed during searches of more complex structures, such as index blocks or hash table collision chains. Batching can also be used advantageously in searching tree structured files and multilevel indexes [7, 8]. In these cases, the upper levels need be traversed only once for the batch of queries, thereby reducing the average search cost for the batch.

For the remainder of this paper we will assume that files consist of records which are organized on the basis of a key unique to each record. We assume that there exists a linear lexical ordering of the keys. Queries, which are collected into a batch of size k, are made by simply specifying a key value. Key values may appear more than once in a batch. For simplicity of analysis we will assume that the key values in the batch are sorted and that the cost of sorting is negligible. This assumption is based on the fact that k is small (say 10 to 100) and that sorting of k queries can be accomplished in the high speed storage with no disk accesses. The search cost metric is the number of accesses required, not the number of comparisons.

Section 2 contains the analysis for sequential searches, and Section 3 the analysis for multiway trees. Section 4 contains a summary and suggestions for further work.

2. SEQUENTIAL SEARCHES

In the following analysis we assume that retrieval requests are made randomly for records in a sorted sequential file of N records, so that each record is equally likely

to be requested. Then if a sequential search is performed, the search length or "cost" is a random numerical quantity. We will use the following fundamental formula for the mean or expected value of a bounded random integral quantity X.

$$\text{mean} \equiv E(X) = \sum_{-\infty<x<\infty} x\text{Prob}\{X = x\}. \tag{1}$$

A partial summation argument shows that also

$$E(X) = \sum_{x \in \text{range of } X} \text{Prob}\{X \geq x\}. \tag{2}$$

For example, if X denotes the random length of a simple sequential search, then

$$\text{Prob}\{X \geq j\} = (N - j + 1)/N$$

and hence

$$E(X) = \sum_{j=1}^{N} (N - j + 1)/N = N + 1 - (1/N)\sum_{j=1}^{N} j = (N + 1)/2.$$

One additional formula for $E(X)$ involves conditional means. If there are two random quantities, X, Y (not necessarily independent!), then the occurrence of the event $Y = c$ determines new probabilities for X since knowledge of Y may furnish additional information. Using the new probabilities in (1) we obtain the conditional mean of X given that $Y = c$; this quantity will be denoted by $E(X \mid Y = c)$. The following formula holds:

$$E(X) = \sum_{-\infty<c<\infty} E(X \mid Y = c)\text{Prob}\{Y = c\}. \tag{3}$$

Our objective is to compute the expected cost savings in batching k requests as opposed to performing k sequential searches. Instead of computing each expected cost separately, we consider the difference between the number of accesses required for k batched random requests and k sequential searches. The expected value of this random quantity will be denoted by $S(k, N)$, the average savings from batching k requests against a sequential file of N records. Clearly,

$$S(0, N) \equiv 0, \quad S(1, N) \equiv 0, \quad S(k, 1) = k - 1.$$

Suppose k requests are distributed randomly among $N + 1$ keys. We introduce the random quantity $Y = \#$ requests for key 1. Then an elementary combinatorial argument gives

$$\text{Prob}(Y = j) = \binom{k}{j}(N^{k-j}/(N + 1)^k).$$

Also, since $Y = j$ is the condition that $k - j$ requests are distributed among the keys $2, \ldots, N + 1$, we have $E(\text{savings} \mid Y = j) = k - 1 + S(k - j, N)$. Hence, from (3) we obtain the recursive formula

$$S(k, N + 1) = \sum_{j=0}^{k} (k - 1 + S(k - j, N))\binom{k}{j}(N^{k-j}/(N + 1)^k),$$

which reduces to

$$S(k, N + 1) = (1/(N + 1)^k) \sum_{r=1}^{k} S(r, N)N^r\binom{k}{r} + k - 1. \tag{4}$$

One may verify directly (see Appendix) that the solution of (4) is

$$S(k, N) = (N + 1)(\tfrac{1}{2}k - 1) + \sum_{r=1}^{N} (r/N)^k. \tag{5}$$

An accurate closed form estimate is obtained by estimating the last expression in (5). Since

$$1/(k + 1) = \int_0^1 x^k \, dx < (1/N) \sum_{r=1}^{N} (r/N)^k < \int_0^1 x^k \, dx + 1/N = 1/(k + 1) + 1/N,$$

we have

$$0 < S(k, N) - [(\tfrac{1}{2}k - 1)(N + 1) + N/(k + 1)] < 1.$$

A lower estimate of the average relative savings of batching k requests over k sequential searches is then

$$((\tfrac{1}{2}k - 1)(N + 1) + N/(k + 1))/(\tfrac{1}{2}k(N + 1)) = 1 - 2/(k + 1) - 2/(k(k + 1)(N + 1)).$$

Interpretation of these results leads to some interesting points. Clearly when k is large this analysis yields approximately the same results as the cruder technique, which assumes that a batched search requires the entire sequential file be scanned. For small batch size this analysis reveals that the entire file need not be scanned to satisfy the batch of queries. For a batch size of two, approximately the first two-thirds of the file needs to be traversed; for a batch size of three, approximately the first three-fourths of the file needs to be traversed.

For an unsorted sequential file the average batched search length is the same as for a sorted sequential file if we assume that all keys in the batch of queries occur in the file. Should there be any unsuccessful queries, then the entire file must be scanned and the analysis is trivial. In the case of unequal request probabilities or of probabilistic sequential search the analysis would have to be repeated beginning from (3).

Table I shows the number of physical accesses saved and the percentage savings of batching as opposed to multiple sequential searches.

Table I. Average Number of Accesses Saved by Batching Requests in a Sequential File and the Percentage Saving

Batch size	Number of records in sequential file					
	100		1000		10000	
2	33.3	33.0	333.3	33.3	3333.3	33.3
5	168.1	66.6	1668.2	66.6	16668.2	66.6
10	413.1	81.8	4094.9	81.8	40913.1	81.8
20	913.8	90.5	9056.6	90.5	90485.2	90.5
50	2426.0	96.1	24043.6	96.1	240220.1	96.1
100	4950.0	98.0	49658.9	98.0	490148.0	98.0

3. BATCHING IN j-ARY SEARCH TREES

We assume that retrieval requests are made randomly from a tree with l levels where each node contains $j-1$ keys and j pointers. We also assume that the search cost within a node is zero. The expected cost savings from batching k requests against a j-ary tree with l levels is denoted by $S_j(k, l)$. Since j will be fixed in the following discussion, we omit the subscript.

We consider the expected cost saving due to those batch requests which appear in the subtree determined by the first node on level 1 (see Figure 1), where the tree has $l+1$ levels. Given that there are n requests in this subtree, the expected savings is $S(n, l)$. The number of keys in a j-ary tree with l levels is given by $\sum_{n=0}^{l-1} (j-1)j^n = j^l - 1$. Therefore,

$$\text{Prob(a request falls within first subtree)} = (j^l - 1)/(j^{l+1} - 1) \equiv P_l,$$

and so

$$\text{Prob(exactly } n \text{ requests fall within first subtree)} = \binom{k}{n} P_l^n (1 - P_l)^{k-n}.$$

Then from (3), the expected cost savings due to the batched requests in the first subtree is given by

$$\sum_{n=0}^{k} S(n, l)\binom{k}{n} P_l^n (1 - P_l)^{k-n}.$$

Due to the symmetry of the tree, each subtree at level 1 has the same expected savings and thus we obtain the recursive equation

$$S_j(k, l+1) = k - 1 + j \sum_{n=1}^{k} S_j(n, l)\binom{k}{n} P_l^n (1 - P_l)^{k-n}. \tag{6}$$

Clearly $S_j(1, l) \equiv 0$ and $S_j(k, 1) = k - 1$.

The algorithms for searching j-ary trees are more challenging. An elegant algorithm for batched searching of binary trees draws on the partitioning scheme used in quicksorting [9]. The root of the tree is used to partition the unsorted array of key requests so that the left-hand portion of the array contains values less than the root and the right-hand portion values greater than the root. Then by recurring down the left side of the tree with the left-hand portion of the array and down the right side of the tree with the right-hand portion of the array, the search can be carried out completely. As key requests are located in the tree, appropriate information can be returned. This algorithm can be generalized for j-ary trees.

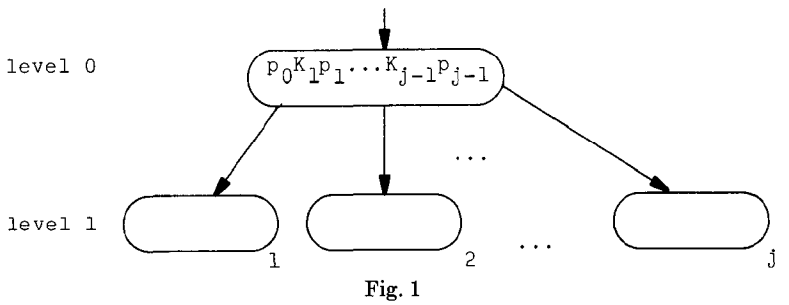

Fig. 1

Table II. Average Number of Accesses Saved by Batching Requests Against a Binary Tree and the Percentage Savings

Batch size	Number of levels in binary tree							
	2		5		10		20	
2	1.2	36.6	1.8	21.2	2.0	11.0	2.0	5.3
3	2.6	51.9	4.0	32.1	4.6	17.1	4.6	8.2
5	5.6	67.2	9.4	45.1	11.2	24.8	11.3	11.9
10	13.7	82.2	25.3	60.9	31.8	35.3	32.4	17.0
20	30.3	91.0	61.6	74.0	82.2	45.6	84.5	22.2
50	80.3	96.4	180.2	86.6	265.0	58.8	276.9	29.1

Table III. Average Number of Accesses Saved by Batching Requests Against an 11-ary Tree and the Percentage Savings

Batch size	Number of levels in 11-ary tree							
	2		3		4		5	
2	1.1	28.1	1.1	18.9	1.1	14.1	1.1	11.2
3	2.2	38.7	2.3	26.2	2.3	19.6	2.3	15.6
5	4.7	49.1	4.9	33.7	4.9	25.2	4.9	20.1
10	11.8	61.4	12.5	43.1	12.6	32.4	12.6	25.8
20	28.3	73.7	30.1	53.0	31.2	40.0	31.3	31.9
50	84.0	87.6	95.2	65.6	97.6	50.0	97.9	40.0

Table IV. Average Number of Accesses Saved by Batching Requests Against a 101-ary Tree and the Percentage Savings, Assuming Root Node Is Kept in Storage

Batch size	Number of levels in 101-ary tree			
	2		3	
10	.4	2.1	.4	1.4
50	10.2	10.3	10.5	7.0
100	35.7	18.0	36.8	12.3
150	70.6	23.6	72.8	16.2

Tables II and III show the numbers of accesses saved and the percentage savings for batched searches as opposed to multiple serial searches, for binary and 11-ary trees, respectively.

In any realistic implementation of a tree structured file, the root node would usually be kept in the high speed storage at all times, thereby reducing the advantage of batched searching. This implementation assumption would eliminate the $k-1$ term in (6) and reduce the values in Tables II and III. Table IV shows the number of accesses saved and the percentage savings for a 101-ary tree, assuming that the root node is always in the high speed storage.

As the number of levels l in the tree increases, the number of accesses increases,

since the initial shared paths are longer, but the percentage saved decreases. As the degree of the tree j increases, the benefit of batched searching decreases since there is a smaller probability that two queries will follow the same path in the tree. As the batch size k increases the advantage of batched searching increases.

If the probabilities of request for the nodes are not equal, then more searches will be concentrated in one portion of the tree and batched searching will be more appealing. The equal probability of request assumption is the least favorable environment for batched searching.

4. SUMMARY

Batching of search or update requests can produce substantial savings over multiple serial searches when the number of accesses is the criterion for performance. Sequential files and tree structured files provide the most obvious cases for analysis, but other data structures should be studied as well. Although insertions can clearly be batched for sequential files, tree structured files present a greater challenge when insertion is considered. This is particularly true in the case of B-trees where the node splitting strategy complicates the problem.

This paper focuses on complete trees where each node requires a disk access. We also assume that each key is equally likely to be requested. Additional analyses are required for arbitrary j-ary trees, for the case of nonequal request probabilities and for other storage strategies.

It would be useful to analyze batched searching for structures which are maintained entirely in the high speed storage. In this case the cost should be estimated by the number of comparisons required. Batching requests against a binary searched array of keys would be especially important since this strategy is so frequently used.

The batching of requests can be generalized to other searchable structures such as digital trees, digital tries, hash tables, etc. Another application is in the batching of substring match requests against a search string. This idea has been applied by Aho and Corasick [10] to scan journal article titles for occurrence of any one of a number of keywords.

APPENDIX

A verification that the solution of (4) is

$$S(k, N) = (N + 1)(\tfrac{1}{2}k - 1) + \sum_{n=1}^{N} (n/N)^k.$$

The formula clearly holds when $N = 1$ or $k = 1$. The substitution of this expression into the right-hand side of (4) gives

$$((N + 1)/(N + 1)^k) \sum_{r=1}^{k} \binom{k}{r} [\tfrac{1}{2}rN^r - N^r] + ((N + 1)/(N + 1)^k) \sum_{r=1}^{k} \binom{k}{r} \sum_{n=1}^{N} n^r + k - 1.$$

From the identity $(1 + a)^k = \sum_{r=0}^{k} \binom{k}{r} a^r$, we obtain

$$\sum_{r=1}^{k} \binom{k}{r} r a^r = a \frac{d}{da}(1 + a)^k = ak(1 + a)^{k-1}.$$

The expression in (4) may then be written as

$$(1/(N+1)^{k-1})[\tfrac{1}{2}kN(1+N)^{k-1} - (1+N)^k + 1] + (1/(N+1)^k) \sum_{n=1}^{N} \sum_{r=1}^{k} \binom{k}{r} n^r + k - 1$$

$$= \tfrac{1}{2}kN - (N+1) + (1/(N+1)^{k-1}) + 1/(N+1)^k \sum_{n=1}^{N} [(1+n)^k - 1] + k - 1$$

$$= \tfrac{1}{2}k(N+2) - N - 2 + 1/(N+1)^{k-1} + (1/(N+1)^k) \sum_{n=0}^{N} (1+n)^k$$

$$- 1/(N+1)^k - N/(N+1)^k$$

$$= (N+2)(\tfrac{1}{2}k - 1) + (1/(N+1)^k) \sum_{n=0}^{N} (1+n)^k.$$

Thus the formula is valid.

ACKNOWLEDGMENTS

We wish to thank Dan Friedman and Andrew Lenard for their valuable discussions and suggestions. Ken Shafer implemented several batching algorithms connected with this work. The insightful comments of the referees substantially improved the quality of this paper.

REFERENCES

1. Meadow, C.T. *The Analysis of Information Systems.* Melville Pub. Co., Los Angeles, 2nd ed., 1973, pp. 300–303.
2. Salton, G. *Automatic Information Organization and Retrieval.* McGraw-Hill, New York, 1968, pp. 243ff.
3. Knuth, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
4. Stocker, P.M., and Dearnley, P.A. A self-organising data base management system. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 337–348.
5. Nijssen, G.M. Efficient batch updating of a random file. Proc. 1971 ACM-SIGFIDET Workshop—Data Description, Access and Control, pp. 173–186.
6. Hwang, K., and Yao, S.B. Parallel processing of multiway search trees. Proc. 1975 Conf. on Computer Graphics, Pattern Recognition and Data Structures, IEEE Computer Soc., May 14, 1975, pp. 170–176.
7. Shneiderman, B. A model for optimizing indexed file structures. *Internat. J. of Computer and Information Sciences 3*, 1 (1974), 93–103.
8. Bayer, R., and McCreight, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1* (1972), 173–189.
9. Friedman, D.P., and Wise, D.S. An environment for multiple-valued recursive procedures. Tech. Report 40, Comput. Sci. Dep., Indiana U., Bloomington, Ind., Oct. 1975.
10. Aho, A.V., and Corasick, M.K. Efficient string matching: An aid to bibliographic search. *Comm. ACM 18*, 6 (June 1975), 333–340.

Received January 1976; revised April 1976