Reading a Set of Disk Pages*

Bernhard Seeger Per-Åke Larson Ron McFadyen

Abstract

The problem studied in this paper is as follows. Consider a file stored in contiguous space on disk. Given a list of pages to be retrieved from the file, what is the fastest way of retrieving them? It is assumed that adjacent pages on disk can be read with a single read request. The straightforward solution is to read the desired pages one by one. However, if two or more pages are located close to each other it may be faster to read them with a single read request, possibly even reading some intervening "empty" pages. It is shown that finding an optimal read schedule is equivalent to finding the shortest path in a certain graph. A very simple approximate algorithm is then introduced and (experimentally) shown to produce schedules that are close to optimal. The expected cost of schedules produced by this algorithm is derived. It is found that significant speed-up can be achieved by the simple mechanism of using additional buffer space and issuing "large reads" whenever it is advantageous to do so.

1 Introduction

Reading some subset of the pages of a file is a frequent operation in database systems. This occurs, for example, when a secondary index is used for the evaluation of a selection query. The simplest way of performing this operation is to scan the index and, for each qualifying entry in the index, retrieve the required page from the file. This has the drawback that the same page may be retrieved more than once. An improvement on this scheme is to first create a list of which pages have to be retrieved, eliminate duplicates from this list, and then retrieve the required pages one at a time. If two or more required pages happen to be located close to each other, (for example, on the same disk track), total retrieval time may be reduced if all of them are read with a single read request instead of issuing multiple requests, each one reading a single page. This requires additional buffer space, of course.

The problem studied in this paper is as follows. Given a list of pages to be retrieved from a file, what is the fastest way of retrieving them? It is assumed that adjacent pages on disk can be read with a single read request (also called set-oriented I/O [Wei89]). The cost of reading a page consists of two parts: positioning time and transfer time (time to transfer a page from secondary storage to main memory). Typically, (average) positioning time is much higher than transfer time. The idea is to reduce the number of read requests (positioning operations) by attempting to read multiple target pages with one request. To achieve this, we may even read some pages that are not absolutely required if this reduces the overall retrieval time. Assuming concurrent read requests, this policy might however result in a lower throughput of the disk, i.e. the expected number of satisfied requests per second may decrease.

The rest of the paper is organized as follows. In section two, we define the problem more precisely and introduce our cost model. In section three, we show that finding an optimal read schedule is equivalent to finding the shortest path in a certain graph. In section four, we present a very simple algorithm for reading a set of pages and (experimentally) show that its performance is close to optimal. In section five,

*This work was supported by grants from Deutsche Forschungsgemeinschaft, Germany, the Natural Science and Engineering Research Council, Canada and the Information Technology Research Centre, Canada.
Authors' address: Bernhard Seeger, Institut für Informatik, Universität München, Leopoldstr. 11b, D-8000 München, Federal Republic of Germany; Per-Åke Larson, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1; Ron McFadyen, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

we analyze the expected cost of the read schedules produced by this algorithm. We begin with two special cases and derive simple closed formulas for the expected cost. These two cases provide upper and lower bounds on the expected cost. We then analyze the general case and derive a recurrence relation which can be used to numerically compute the expected cost. Section six extends the model to vector reads (scatter-reads). Section seven summarizes the results and concludes the paper.

We have not found any papers dealing with the exact problem studied here. However, several related problems have been studied previously. Several authors considered the problem of estimating the number of target pages given the number of records in the response set. [Yao77] and [WWS 83] are the most important references on this problem. Related problems are discussed in several other papers, such as prefetching [Smi 78], batching of queries [Pal 85] and buffering [Pal 88]. The work by Weikum [Wei89] and Hutflesz et al. [HSW 88] is most closely related to the work reported in this paper. In [Wei89] reading complex objects using "large" requests was experimentally shown to pay off in comparison to the traditional approach. In order to support range queries efficiently, a multidimensional access method that preserves the ordering of data on disk was proposed in [HSW 88] (see also further references in that paper). Pages are stored close together on disk when their data is close together in the data space (domain). This property results in reducing the positioning operations of a range query. However, only target pages are read.

2 Problem Definition

Consider a file F consisting of a contiguous sequence of pages numbered $1, \ldots, N$. Pages are of fixed size and each page can store a maximum of c records (page capacity). A query selects some subset of the records stored in the file (the response set of the query). A page containing at least one record in the response set is called a *target page* and the set of all target pages is called the *target set*. A page containing no records in the response set (and thus not in the target set) is called an *empty page*. To compute the result of the query, every target page must to be read. We assume that the complete target set is known before actual retrieval of the pages begins. This situation occurs relatively frequently in query processing. Retrieval by means of an index is the most typical example but there are other situations where the target set is known before retrieval begins.

The (elapsed) time for a read request consists of positioning time and transfer time. The positioning time is the time to move to the right position in the file (seek time and rotational delay). The transfer time is simply the time to transfer the requested pages from secondary storage into main memory. We make the simplifying assumption that the positioning time is constant. We take the time required to transfer a page as the cost unit and express the cost in terms of page transfers. Let P denote the ratio of positioning time to transfer time. The cost of a read request transferring f pages is then $P + f$ (transfer units).

F	target file
N	number of pages in the file
c	capacity of a page (in records)
Q	target set (pages to retrieve)
b	number of target pages
α	$= \frac{b}{N}$
m	maximum gap size
p	number of buffer pages
P	ratio of positioning time to transfer time

Table 1: List of symbols

Whenever a sequence of pages is read into main memory, a sufficiently large buffer area must be available. We assume that buffer space for at most p pages, $p \geq 1$, is available. The problem then is how to minimize the overall cost of reading the required pages into main memory. An obvious way of reducing the cost is as follows: whenever there is a contiguous sequence of target pages (at most p pages), read all of them into main memory with a single request. If the transfer time is significantly less than the positioning time, it may be worthwhile reading some empty pages if this reduces the number of read requests. For example, consider a situation where a target page is followed by an empty page which is followed by a target page. Overall cost is (almost always) reduced if all three pages are read with a single request, instead of reading the two target pages using separate requests.

In many operating systems, in particular several variants of UNIX, there are two suitable operations for implementing read requests. An *ordinary read* transfers a contigious sequence of pages from the disk into a contigious area of main memory. A *vector read* can be used to transfer the pages into several noncontiguous buffers. The advantage of a vector read is that all empty pages of a read request can be assigned to the same position in the buffer. Thus, at most one page of the buffer is used for collecting the empty pages of a read request. We first analyze the case of ordinary reads in sections 2-5. The case of vector reads is analyzed in section 6.

Definition 2.1 *Let Q be a subset of the set $F = \{1, \ldots, N\}$ (the file) and p, $p \geq 1$, an integer (representing the buffer capacity). Let the tuple (s, t) denote* *a read request reading t pages beginning from page s. Then a sequence $\delta = ((s_1, t_1), \ldots, (s_m, t_m))$ is a* read schedule *for Q, if*

1. *$t_i \leq p$ for every $i \in \{1, \ldots, m\}$*
2. *for every $q \in Q$, there exists a tuple (s_i, t_i) in δ such that $s_i \leq q < s_i + t_i$*

The read schedule is ordered *, if*

3. *$s_i < s_{i+1}$ for every $i \in \{1, \ldots, m-1\}$*

Example: Consider the file and target set illustrated below. One indicates a target page and zero indicates an empty page.

1	0	1	0	0	1	1	0	1	0	0	0	1	1	0	1

Assuming a buffer with 4 pages, ((1,3), (6,2), (9,1), (13,4)) is an example of an ordered read schedule. This schedule is interpreted as follows: the first read request reads pages 1,2 and 3, the second reads pages 6 and 7, and so on.

Definition 2.2 *Let $C(\delta)$ denote the cost of a read schedule δ and Δ the set of all possible read schedules for a given target set Q and buffer size p. The* subset reading problem *is then to find a read schedule δ_{opt} such that*

$$C(\delta_{opt}) = \min_{\delta \in \Delta} C(\delta) \quad (1)$$

Up to section 6, we assume that the cost of a read schedule $\delta = ((s_1, t_1), \ldots, (s_m, t_m))$ is computed as

$$C(\delta) = \sum_{i=1}^{m} (P + t_i). \quad (2)$$

This cost function is admittedly simplistic, but better than simply counting the number of pages read. A more detailed cost model would have to consider the geometry of the disk, the actual layout of the file on the disk, the seek times and rotational delays incurred, and the time to process the records on a page.

3 Optimal Read Schedules

In this section, we show that an optimal read schedule can be found by computing the shortest path in an appropriately constructed graph. The graph is acyclic with positive edge weights and any standard shortest-path algorithms can be used. We first state two lemmas which show that only a restricted class of read schedules need be considered. Note, however, that the lemmas do not necessarily hold under a different cost model.

Lemma 3.1 *Any optimal read schedule, $\delta_{opt} = ((s_1, t_1), \ldots, (s_m, t_m))$, has the following two properties:*

1. *$s_j + t_j \leq s_i$ or $s_i + t_i \leq s_j$ for every $i, j \in \{1, \ldots, m\}$, $i \neq j$*
2. *for every $i \in \{1, \ldots, m\}$, $s_i \in Q$ and $s_i + t_i - 1 \in Q$*

In other words, an optimal read schedule must have non-overlapping reads and every read request must begin and end with a target page. Both properties are rather obvious so we will only outline the proof.

Proof: To prove that the first property must be satisfied, assume that a schedule contains two overlapping reads: (s_i, t_i) and (s_j, t_j). If (s_i, t_i) is a subset of (s_j, t_j) (or vice versa), the cost can be reduced by eliminating (s_i, t_i) from the schedule. If the two reads overlap, but neither is a subset of the other, the transfer cost is reduced if the common pages are eliminated from one of the reads. It follows that a read schedule containing overlapping reads cannot be optimal.

If a read request (s_i, t_i) begins with an empty page, we can reduce the transfer cost simply by changing it to $(s_i + 1, t_i - 1)$. The same applies if a read request ends with an empty page. It follows that an optimal schedule must satisfy property two. □

Lemma 3.2 *Let δ be a read schedule satisfying the properties of the previous lemma and δ' be the equivalent ordered schedule, that is, containing exactly the same read requests but listed in ascending order on the first component (s_i). Then $C(\delta) = C(\delta')$.*

Proof: The proof follows immediately from the observation that $C(\delta')$ is simply a reordering of the terms in $C(\delta)$. □

An ordered schedule satisfying the two properties of Lemma 3.1 will be called a *regular schedule*. The two lemmas guarantee that we need only consider regular schedules. To find an optimal schedule, we create a *schedule graph* from which all regular read schedules can be determined. The schedule graph is created as follows:

1. There is one node for each member (page) of the taget set Q. The node corresponding to member (page) i is labeled i. There is one (initial) node labeled 0.
2. Let i denote a node and j the node with the next higher node label. For every node i, there is an edge
   (a) from node i to node j. The weight of the edge is $P + 1$.

Figure 1: Target set and corresponding schedule graph

(b) from node i to every node k such that $k - j < p$, $j < k \leq N$. The weight of the edge is $P + (k - j + 1)$.

3. There are no other nodes and edges.

Let M denote the maximum node label occurring in the graph. Every path from node 0 to node M represents a read schedule and each edge in the path represents a read request. Consider an edge from a node i to a node k. If there are no nodes between i and k, that is, no nodes with labels in the range $i+1$ to $k-1$, then the edge represents the read request $(k, 1)$. Otherwise, the edge represent the read request $(j, (k-j+1))$ where j denotes the node with the next higher label after i. In other words, each edge points to the last page of a read request. The sum of the edge weights of a path is equal to the cost of the read schedule.

An example target set and schedule graph are shown in Fig. 1. The file consists of 16 pages and there are 8 target pages. The buffer is assumed to have a capacity of 4 pages ($p = 4$). If, for example, a read request ends at page 3, there are three possibilities for the next request. First, we can read page 6 only, requiring the transfer of one page. Second, we can read pages 6 and 7, requiring the transfer of two pages. Third, we can read pages 6,7,8 and 9, requiring the transfer of four pages. Note that this request reads page 8 although it is an empty page. The shortest path for $P = 2$, and thus the optimal schedule, is indicated by bold edges.

Theorem 3.3 *The shortest path from node 0 to node M in the schedule graph defines an optimal read schedule.*

Proof: To prove the theorem, we must first prove that (a) every path from node 0 to node M represents a regular schedule and (b) every regular schedule is represented in the graph. Part (a) follows directly from the construction of the graph. Hence, we need only show that every regular read schedule is represented by a path in the graph.

Assume that there exists a regular read schedule which is not represented by any path in the graph. Then the schedule must contain at least one read request for which there is no corresponding edge in the graph. Assume that this read request begins with page j and ends with page k, $k \geq j$. Because the schedule is regular (property 2 of Lemma 3.1), pages j and k must be target pages and consequently the graph also contains a node j and a node k. Let i denote the node immediately preceding node j. An edge from node i to node k would represent the read request and we must show that such an edge exists. There are two case to consider: $j = k$ and $j < k$. For the case $j = k$, the existence of the edge follows from 2(a) in the definition of the graph. For the case $j < k$, we note that $k - j < p$ must be true. Otherwise the schedule would be invalid. From this observation and point 2(b) of the definition of the graph, it follows that there exists an edge from node i to node k. This contradicts the assumption that the read request is not represented in the graph.

The construction of the graph guarantees that there is always an edge between two adjacent nodes. Hence, at least one path from node 0 to node M always exists. It follows that the shortest path from node 0 to node M defines an optimal read schedule. $\square$

Once the graph has been constructed, we can use any shortest-path algorithm to find an optimal read schedule. However, it is questionable whether it is worthwhile in practice to compute an optimal schedule. The model ignores many factors, for example, queuing delays and time to process records. Hence, a theoretically optimal schedule may not in practice be optimal. Furthermore, we cannot estimate the performance of schedules produced by this algorithm (without knowing the exact layout of the target set), something which is needed for query optimization purposes. In the next section, we present a simple algorithm which produces read schedules that are very close to optimal.

4 Simplified Algorithm

The basic idea of the algorithm is simple: start reading from the next target page, stop reading either at the last target page that fits into the buffer area or at the last target page before a long stretch of empty pages. This algorithm is based on the observation that it is often cheaper to read a few empty pages than to skip them. A sequence of empty pages is called a gap. Let m denote the maximum sequence of empty pages that will be read, that is, when a gap of $m+1$ or more empty pages is encountered, the read request ends with the last target page before the gap.

```
Algorithm ReadSubset(F: File; Q: TargetSet; B: Buffer;
                     p: BfrSize; m: GapSize);
BEGIN
  end := 0;
  REPEAT
    start := NextTargetPage( F, Q, end);
    prev := start;
    LOOP
      next := NextTargetPage( F, Q, prev);
      IF (next > m + 1 + prev) OR (next ≥ p + start)
        OR (next > N) THEN
        end := prev; EXIT
      END;
      prev := next;
    END;
    ReadIntoBuffer(F, B, start, end);
    Process records in B;
  UNTIL (next > N);
END ReadSubset;
```

The function *NextTargetPage(F,Q,j)* is assumed to compute the position of the first target page after page j. If none exists, it returns a value greater than N. The procedure *ReadIntoBuffer(F, B, start, end)* reads pages $start, start+1, \ldots, end$ from file F into buffer B. The first and the last page of a read request are always target pages. The algorithm adds pages to the read request until one of three conditions is satisfied: a gap of $(m+1)$ or more empty pages is found, the next target page is past the end of the buffer, or the end of the file has been reached.

This simple algorithm does not guarantee optimal read schedules. We have performed extensive simulation experiments which indicate that the schedules produced are close to optimal. The results of one set of experiments are listed in Table 2. The results are for a file with 100,000 pages and a target set of 10,000 (randomly chosen) pages. The figures are averages of 20 experiments. The table lists the cost per target page of schedules produced by *ReadSubset*, the cost of optimal schedules per target page and the relative difference. The cost of schedules produced by *ReadSubset* depends on the value of the maximum gap size (m). For each buffer size (p), the value of m was

p	*ReadSubset*	Optimum	Diff.(%)
2	10.079	10.079	0.0000
4	8.883	8.866	0.1948
6	8.206	8.153	0.6527
8	7.818	7.715	1.3372
10	7.585	7.454	1.7552
12	7.415	7.293	1.6660
14	7.299	7.184	1.5967
16	7.209	7.105	1.4669
18	7.144	7.046	1.3847
20	7.090	7.004	1.2371
24	7.019	6.948	1.0190
28	6.971	6.914	0.8242

Table 2: Cost (per target page) of read schedules produced by *ReadSubset* compared with cost of optimal schedules. ($P = 10, \alpha = 0.1, N = 100,000$)

chosen so as to produce the best schedule (minimal cost). As shown in the table, the (best) read schedules produced by algorithm *ReadSubset* were within 2% of the optimum. Similar results were obtained from other experiments.

The behavior of the algorithm depends on the buffer size (p) and maximum gap size (m). Five different cases are discussed below. We illustrate the discussion using the example file and target set shown in Figure 4. The value of P is assumed to be 2.

1. $p = 1$:
   Setting $p = 1$ produces schedules reading one target page at a time, that is, the traditional approach. The cost of the schedule produced for the example file is 24 $(8*2+8)$.

2. $m = 0$, $p = \infty$:
   An unlimited amount of buffer space is assumed. This parameter setting results in schedules where each request reads a contiguous sequence (cluster) of target pages. The algorithm takes advantage of whatever clustering there is in the file but never reads an empty page. The cost of the resulting schedule is 20 $(6*2+8)$. This case was analyzed in [McF 90].

3. $0 < m < \infty$, $p = \infty$:
   This version also assumes an unlimited amount of buffer space. To reduce the number of positioning operations, (short) gaps of empty pages are read instead of skipped. The value of m affects the cost of the read schedules produced. Setting $m = 2$ produces a read schedule with cost 17 $(2*2+13)$ for our example file.

4. $m = \infty$, $1 < p < \infty$:
   A buffer of limited size is used but there is no

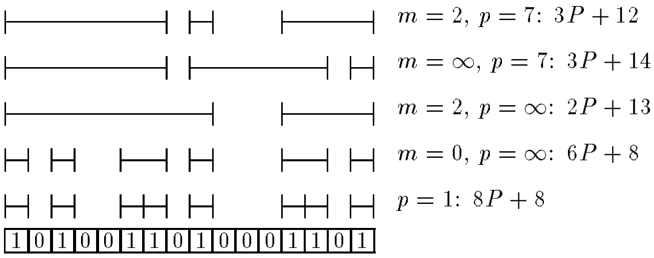

Figure 2: Example file and read schedules for different parameter settings

restriction on the length of gaps. In other words, a request reads every page up to and including the last target page covered by the buffer. For $p = 7$, we obtain a schedule with cost 20 ($3 * 2 + 14$) for our example file.

5. $0 < m < \infty$, $1 < p < \infty$:
   This is the most general case: a buffer of limited size is available and the maximum gap size is also limited. For $p = 7$ and $m = 2$, the resulting schedule has a cost of 18 ($3 * 2 + 12$).

5 Analysis

In this section, we analyze the expected cost of read schedules produced by algorithm *ReadSubset*, first for two special cases and then the general case. The analysis is structured in this way because (a) simple closed formulas can be derived for the two special cases but not for the general case and (b) the cost formulas for the two special cases are upper and lower bounds for the general case. The analysis is asymptotic, that is, for $N, b \rightarrow \infty$ and keeping $\alpha = b/N$ constant. The cost is expressed as the expected cost per target page. Target pages are assumed to be randomly distributed over the file. The probability of a page being a target page is α.

5.1 Unlimited Gaps, Limited Buffers

We begin by considering the case $m = \infty$ and $p < \infty$. We first derive the expected number of read requests and then the expected number of pages transferred per request.

The expected number of *target* pages transferred by a read request is $1 + (p-1)\alpha$. The first page is always a target page. Each one of the remaining $p - 1$ pages covered by the buffer is a target page with probability α. The number of read requests per target page is then simply

$$\frac{1}{1 + (p-1)\alpha}$$

A read request transfers some number of pages (target pages and empty pages). The number of pages transferred equals p minus the number of empty pages located at the end of the buffer. The probability u_i that there are exactly i empty pages at the end of the buffer is given by

$$u_i = \begin{cases} \alpha(1-\alpha)^i & \text{for } i < p-1 \\ (1-\alpha)^{p-1} & \text{for } i = p-1 \end{cases}$$

The expected number of empty pages at the end of the buffer is then

$$\begin{aligned} E &= \sum_{i=0}^{p-1} i u_i \\ &= (p-1)(1-\alpha)^{p-1} + \sum_{i=1}^{p-2} \alpha(1-\alpha)^i i \\ &= (p-1)(1-\alpha)^{p-1} + \\ & \quad \frac{(1-\alpha)}{\alpha}(1 - (1-\alpha)^{p-2} - (p-2)\alpha(1-\alpha)^{p-2}) \\ &= \frac{1-\alpha}{\alpha}(1 - (1-\alpha)^{p-1}) \end{aligned}$$

The expected number of pages actually transferred is $p - E$. Combining the expected number of read requests and the number of pages transferred, we obtain the following formula for the expected cost per target page:

$$cost(\alpha, p, \infty) := \frac{P + p - \frac{1-\alpha}{\alpha}(1 - (1-\alpha)^{p-1})}{1 + (p-1)\alpha} \quad (3)$$

In Figure 3 the expected cost is plotted as a function of the buffer size. The positioning cost and transfer cost are also shown to illustrate the behavior of the two components of the cost function. As expected, the positioning cost decreases (fewer requests) and the transfer cost increases (reading more empty pages) with the buffer size. The cost function has a global minimum. In particular, very large buffers result in a higher overall cost because the number of empty pages read increases.

In Figure 4, the expected cost is plotted for three different buffer sizes ($p = 1, 5, 10$). The figure clearly shows the benefit of using a larger buffer. For $\alpha = 0.2$, increasing the buffer size from one page to 10 pages, reduces the expected cost by 50%.

For the case illustrated in Figure 3, the expected cost has a minimum. The optimal buffer size cannot be derived analytically but can be computed numerically quite easily. Table 3 shows the optimal buffer

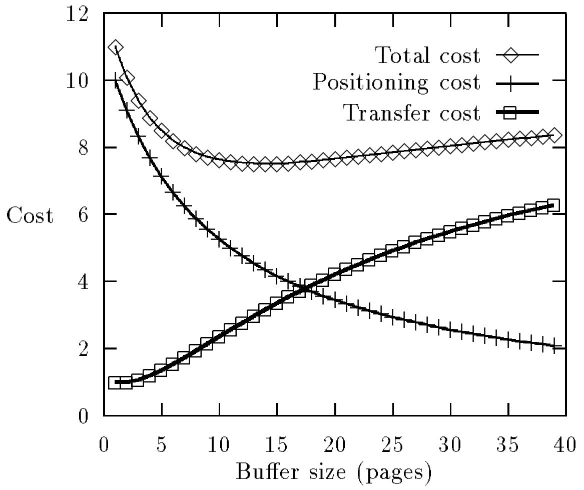

Figure 3: $cost(0.1, p, \infty)$ in page transfers as a function of buffer size ($P = 10$)

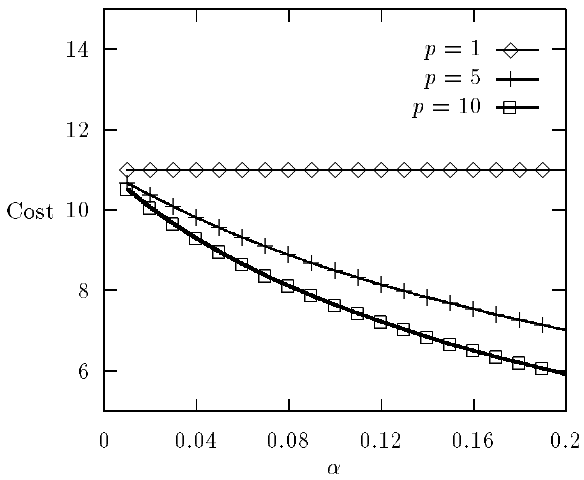

Figure 4: $cost(\alpha, p, \infty)$ in page transfers as a function of α, ($p = 1, 5, 10$, $P = 10$)

α (in %)	p^*	*Records in the response set (in %)*			
		$c = 5$	$c = 10$	$c = 20$	$c = 40$
1.0	12	0.2010	0.1005	0.0503	0.0251
2.0	12	0.4041	0.2020	0.1010	0.0505
4.0	12	0.8164	0.4082	0.2041	0.1021
6.0	12	1.2375	0.6188	0.3094	0.1547
8.0	13	1.6676	0.8338	0.4169	0.2085
10.0	14	2.1072	1.0536	0.5268	0.2634
12.0	15	2.5567	1.2783	0.6392	0.3196
14.0	18	3.0165	1.5082	0.7541	0.3771
15.0	20	3.2504	1.6252	0.8126	0.4063
16.0	25	3.4871	1.7435	0.8718	0.4359
17.0	∞	3.7266	1.8633	0.9316	0.4658

Table 3: Optimal buffer size (p^*) as a function of α, ($P = 10$)

size as a function of the fraction of target pages. We have also listed, for a few different page capacities, the fraction of records in the response set that corresponds to each fraction of target pages. These results were obtained using Yao's formula [Yao77]. At first, the optimal buffer size increases slowly with increasing α. However, when α increases to 17%, the lowest cost occurs for $p = \infty$. This simply means that for large α the best policy is to read the whole file with one read request. In practice, this translates to reading the file sequentially using (very) large buffers.

The results are somewhat surprising if we consider the fraction of records in the response set needed to exceed this critical point. For example, when $c = 20$, our model indicates that the cheapest way to answer a query is to scan the whole file even when the response set contains as little at 1% of the records.

5.2 Limited Gaps, Unlimited Buffer

Next we consider the case $p = \infty$ and $0 < m < \infty$. A gap is a contiguous sequence of empty pages delimited on the left and the right by a target page. A cluster is a contiguous sequence of pages with the following properties:

- the first and the last page of the sequence are target pages
- the sequence does not contain any single gap longer than m
- the sequence is delimited on the left and the right by gaps strictly longer than m

Because $p = \infty$, each read request will read exactly one cluster. To calculate the expected cost of a read schedule, we calculate the expected number of clusters and their expected length.

Consider an arbitrary page in the file. This page begins a cluster if it is a target page and to the left of it is a gap longer than m. Hence, the probability of a page beginning a cluster is

$$\sum_{j>m} \alpha(1-\alpha)^j \alpha = \alpha(1-\alpha)^{m+1}$$

The expected length of a cluster, including the gap separating it from the next clusters, is then $1/(\alpha(1-\alpha)^{m+1})$. The next step is to compute the expected length of the gap separating two cluster. The probability of the gap being of length $m+1+j$, $j \geq 0$, is $\alpha(1-\alpha)^j$. The expected length of the gap is therefore

$$\sum_{j\geq 0}(m+1+j)\alpha(1-\alpha)^j = m + 1/\alpha$$

The pages which are part of the gap will not be read. The expected length of an m-cluster, counting only the pages read, is therefore

$$\frac{1}{\alpha(1-\alpha)^{m+1}} - m - \frac{1}{\alpha}$$

The expected cost per page in the file is then

$$\alpha(1-\alpha)^{m+1}(P + \frac{1}{\alpha(1-\alpha)^{m+1}} - m - \frac{1}{\alpha})$$

which can be simplified to

$$\alpha P(1-\alpha)^{m+1} + 1 - (1-\alpha)^{m+1}(1+m\alpha)$$

Finally, by dividing by α, we obtain the expected cost per target page

$$\begin{aligned} cost(\alpha, \infty, m) &= P(1-\alpha)^{m+1} + \\ & \frac{1}{\alpha}(1-(1-\alpha)^{m+1}(1+m\alpha)) \end{aligned} \quad (4)$$

In Figure 5, the expected cost has been plotted as a function of m. Positioning and transfer costs are also plotted to show their contribution to the total cost. The cost function has a minimum at about $m = 9$ for the case shown in the figure.

For the purpose of finding the minimum of the function $cost(\alpha, \infty, m)$, (for a given value of α), we can treat m as begin defined over the real numbers. The value of m that minimizes the function, can then be determined by taking the derivate of the function with respect to m. If m^* denotes the real value minimizing the function, the optimal integer value is then either $\lceil m^* \rceil$ or $\lfloor m^* \rfloor$.

$$\begin{aligned} m^* &= P - \frac{1}{\alpha} - \frac{1}{\ln(1-\alpha)} \\ &= P - \frac{1}{2} - \frac{1}{12}\alpha - O(\alpha^2) \end{aligned}$$

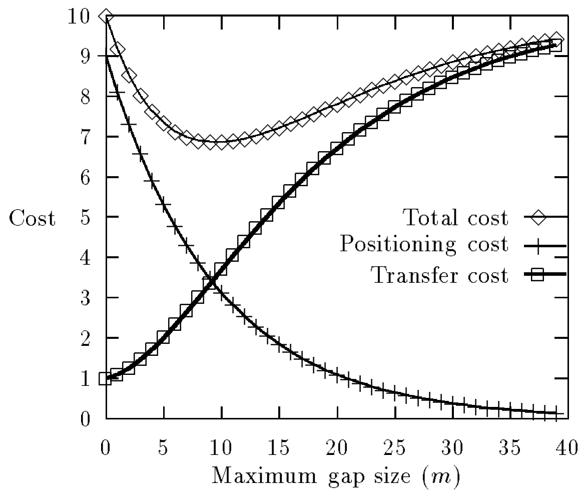

Figure 5: $cost(0.1, \infty, m)$ as a function of m, $(P = 10)$

Figure 6 shows the expect cost as a function of α, for four different values of m. For $P = 10$, the integer minimum for m is either 9 or 10. The graph corresponding to $cost(\alpha, \infty, 20)$ is clearly above the one for $cost(\alpha, \infty, 9)$.

5.3 Limited Gaps, Limited Buffer

In this section we analyze the general case of the algorithm, that is, the case $m < \infty$ and $p < \infty$. Consider a read request filling some number of pages in the buffer. Let $Q(i,j)$, $1 \leq i,j \leq p$, denote the probability that page j in the buffer receives the i-th target page read by this request. Since page j in the buffer can receive at most the j-th target page, it follows $Q(i,j) = 0$ for $i > j$. Furthermore, the first page is always a target page. Hence, for $i = 1$ we have

$$Q(1,1) = 1 \quad and \quad Q(1,j) = 0 \;\; for \;\; 2 \leq j \leq p$$

Now consider the case $i > 1$ and $j \leq i$. Assume that the $(i-1)$-th target page is in position $j-k$ ($k \geq 1$, $1 \leq j-k \leq m+1$). The conditional probability that the next target page is in position j is then $(1-\alpha)^{k-1}\alpha$. Consequently, the probability that the i-th target page is in position j can be computed by summing over all possible positions for the $(i-1)$-th target page. The $(i-1)$-th target page cannot be to the left of page $j-(m+1)$ and $j-k > 0$ must always hold. It follows that for $j \geq i$ $Q(i,j)$ can be computed by the recurrence relation

$$Q(i,j) = \sum_{k=1}^{\min(j-1,m+1)} Q(i-1, j-k)\alpha(1-\alpha)^{k-1}$$

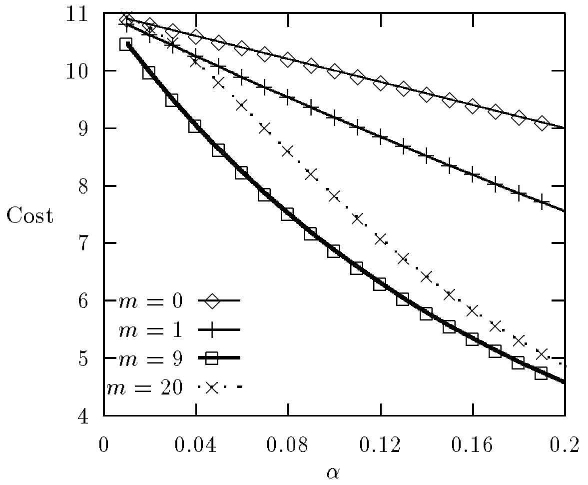

Figure 6: $cost(\alpha, \infty, m)$ as a function of α, $(P = 10)$

Note that the number of target pages per read request is s

Let $Q_{stop}(j)$, $1 \leq j \leq p$ denote the probability that exactly j pages are transferred by a read request. There are two cases to consider. If $p - j > m$, a gap of length $m + 1$ (or more) follows page j. This occurs with probability $(1 - \alpha)^{m+1}$. Otherwise, that is $p - j \leq m$, $p - j$ empty pages follow and the end of the buffer is reached. This occurs with probability $(1 - \alpha)^{p-j}$. Combining the two cases, we obtain

$$Q_{stop}(j) = \begin{cases} (1-\alpha)^{p-j} & \text{if } p - j \leq m \\ (1-\alpha)^{m+1} & \text{if } p - j > m \end{cases}$$

The probability of a page being a target page is independent of its position in the file. For $j > 1$, the probability $Q(i, j)$ is independent of the probability $Q_{stop}(j)$. Therefore, the probability that exactly i target pages are contained in a buffer is given by

$$\sum_{j=1}^{p} Q(i, j) Q_{stop}(j)$$

The expected number of target pages per read request can then be computed as

$$E_{target} = \sum_{i,j \geq 1}^{p} Q(i, j) Q_{stop}(j) i$$

and the expected number of pages transferred as

$$E_{total} = \sum_{i,j \geq 1}^{p} Q(i, j) Q_{stop}(j) j$$

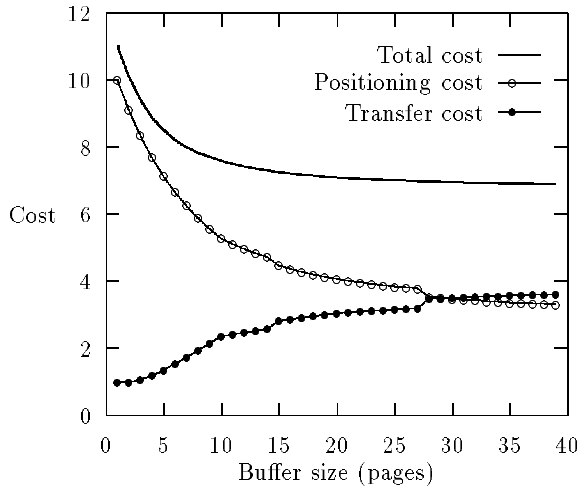

Figure 7: $cost(0.1, p, m^*)$ as a function of p, $(P = 10)$

Finally, the expected cost per target page is given by

$$cost(\alpha, p, m) = \frac{1}{E_{target}} (P + E_{total}) \qquad (5)$$

The expected cost is plotted in Figure 7. The positioning and transfer costs are shown separately. Note that the results are not for a fixed value of m. For each buffer size p, the best value of m was chosen.

It can be proved that the functions $cost(\alpha, p, \infty)$ and $cost(\alpha, \infty, m)$ are indeed special cases of the general function $cost(\alpha, p, m)$ and that the two simple functions are upper and lower bounds on $cost(\alpha, p, m)$. However, the proofs are complicated and therefore not included here. The three functions $cost(\alpha, p, \infty)$, $cost(\alpha, \infty, m)$ and $cost(\alpha, p, m)$ are plotted in Fig. 8. The lowest curve plots the function $cost(\alpha, p, \infty)$. The simpler closed formulas are fairly good approximations of the most general case.

6 A cost model for vector reads

So far we have assumed that ordinary reads are used, i.e. a contiguous area of the disk is copied into a contiguous area of the buffer. Thus, the buffer might contain a high fraction of empty pages. However, many operating systems (in particular several variants of UNIX) offer another operation, called vector read, for reading multiple pages with a single request. A vector read transfers a contiguous sequence of pages from secondary storage into a non-contiguous collection of buffer pages. In particular, a single buffer page may receive several pages. This property can be used for assigning all empty pages of a read request to the

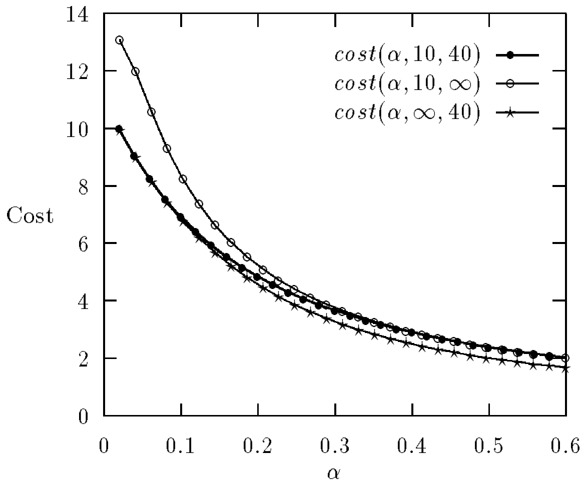

Figure 8: Comparison of the cost functions $cost(\alpha, p, m)$ for $(m, p) = (10, \infty)$, $(10, 40)$, $(\infty, 40)$, $P = 10$

same buffer page. Then, at most one buffer page is sacrificed for receiving empty pages. In the following, we address the problem of finding optimal read schedules under the assumption that a read request is implemented as a vector read. First, we briefly introduce the modified problem definition and a modified algorithm.

Definition 6.1 *Let Q be a subset of the set $F = \{1, \ldots, N\}$ and p, $p \geq 1$, an integer. Let the tuple (s, u, v) denote a read request reading $u + v$ pages beginning from page s where u and v denotes the number of empty and target pages, respectively. Then a sequence $\delta = ((s_1, u_1, v_1), \ldots, (s_m, u_m, v_m))$ is a* v-read schedule *for Q, if*

1. $v_i < p$ *for every* $i \in \{1, \ldots, m\}$ *with* $u_i \neq 0$
2. $v_i \leq p$ *for every* $i \in \{1, \ldots, m\}$ *with* $u_i = 0$
3. *for every $q \in Q$, there exists a tuple (s_i, u_i, v_i) in δ such that $s_i \leq q < s_i + u_i + v_i$*

Instead of using cost formula 2, we assume that a cost of a read schedule is computed as

$$C(\delta) = \sum_{i=1}^{m} (P + u_i + v_i). \qquad (6)$$

Similar to our previous cost model presented in section 2, the solution of computing an optimal v-read schedule can be reduced to solving a shortest-path problem in an acyclic graph. The graph can be constructed in a similar way to the graph described in section 2. However, we are primarily interested in a simple approximative algorithm which produces v-read schedules close to the optimum and whose cost can be easily computed. In order to support vector reads, algorithm *ReadSubset* requires only a few modifications. The modified algorithm *VReadSubset* follows.

```
Algorithm VReadSubset(F: File; Q: TargetSet;
                        B: Buffer;
                        p: BfrSize; m: GapSize);
BEGIN
  end := 0;
  REPEAT
    adr[1] := NextTargetPage( F, Q, end);
    ones := 1; zero_flag := 0;
    LOOP
      adr[ones+1] := NextTargetPage( F, Q, adr[ones]);
      IF (adr[ones+1] > m + 1 + adr[ones]) OR
        (ones ≥ p - zero_flag) OR (adr[ones+1] > N)
      THEN
        end := adr[ones]; EXIT
      END;
      IF adr[ones+1] > adr[ones] + 1 THEN
        zero_flag := 1;
      END;
      ones := ones+1
    END;
    VRead(F, B, adr, ones);
    Process records in B;
  UNTIL (adr[ones+1] > N);
END VReadSubset;
```

The procedure *VRead(F,B,adr,ones)* reads pages with addresses $adr[1]$, $adr[1]+1$, .., $adr[2]-1$, $adr[2]$, $adr[2]+1$, .., $adr[ones]$ from file F into the buffer B. Note that the target pages $adr[1]$, $adr[2]$, .., $adr[ones]$ are assigned to the first *ones* pages in the buffer. All empty pages are read into the p-th buffer page. The algorithm adds pages to the read request until one of three conditions is satisfied: a gap of $(m + 1)$ or more empty pages is found, the next target page causes an overflow of the buffer, or the end of the file has been reached. Let us mention that the algorithm *VReadSubset* indeed produces schedules which are close to optimal.

Most interesting is the analysis of the algorithm *VReadSubset* and a comparison of *VReadSubset* with *ReadSubset*. Note that both algorithms produce the same schedule if the buffer is unlimited. In the following, we restrict our considerations to the most general case, i.e. we assume limited gaps and limited buffer. The following analysis is similar to the one presented in section 5.3. Consider a read request that reads pages labeled $1, 2, 3, \ldots$ into the buffer with p pages. Let $R(i, j)$, $1 \leq i \leq p$, $j \geq 1$ denote the probability that page j is a target page and is assigned to the i-th buffer page. Note that $R(i, j) = 0$ for $i < j$.

Since page 1 has to be always a target page, it will be always the first page in the buffer. Hence for $i = 1$ we obtain

$$R(1,1) = 1 \quad and \quad R(1,j) = 0 \;\; for \;\; j \geq 2$$

Furthermore, the p-th page of the buffer will be filled with a target page, only if all of the other buffers are filled up with target pages and none of the pages read from the file was an empty page. Therefore, for $i = p$ we have

$$R(p,p) = 1 \quad and \quad R(p,j) = 0 \;\; for \;\; j \geq 1, j \neq p$$

Now consider the case $1 < i < p$ and $j \geq i$. Assume that the $(i-1)$-th target page has the label $j-k$ ($k \geq 1$, $1 \leq j-k \leq m+1$). The conditional probability that the next target page has the label j is then $(1-\alpha)^{k-1}\alpha$. Consequently, the probability that the i-th target page has label j can be computed by summing over all possible positions for the $(i-1)$-th target page. The $(i-1)$-th target page cannot be to the left of page $j-(m+1)$ and $j-k > 0$ must always hold. It follows that for $j \geq i$ $R(i,j)$ can be computed by the recurrence relation

$$R(i,j) = \sum_{k=1}^{\min(j-1,m+1)} R(i-1, j-k)\alpha(1-\alpha)^{k-1}$$

Let $R_{stop}(i,j)$, $1 \leq i \leq p, j \geq i$ denote the probability that exactly j pages are transferred by a read request and that i of the j pages are target pages. There are four cases to consider. If $i = p$, the buffer is completely filled and thus, no more pages can be read. If $i = p-1$ and $i < j$, $p-1$ target pages are transferred and at least one of the buffer pages is reserved for the empty pages. Then, the buffer is already full. If $i = j = p-1$, it is possible to read the p-th page into the buffer. However, this page is not read, if the p-th page is an empty page. This occurs with probability $(1-\alpha)$. Otherwise, that is $i < p-1$, a gap of length $m+1$ (or more) follows after the j-th page. This occurs with probability $(1-\alpha)^{m+1}$. Combining the four cases, we obtain

$$R_{stop}(i,j) = \begin{cases} 1 & \text{if } i = p \\ 1 & \text{if } i = p-1, i < j \\ (1-\alpha) & \text{if } i = j = p-1 \\ (1-\alpha)^{m+1} & \text{otherwise} \end{cases}$$

In analogy with section 5.3, the expected number of target pages per read request can then be computed as

$$V_{target} = \sum_{i=1}^{p} \sum_{j \geq i} R(i,j) R_{stop}(i,j) i$$

and the expected number of pages transferred as

$$V_{total} = \sum_{i=1}^{p} \sum_{j \geq i} R(i,j) R_{stop}(i,j) j$$

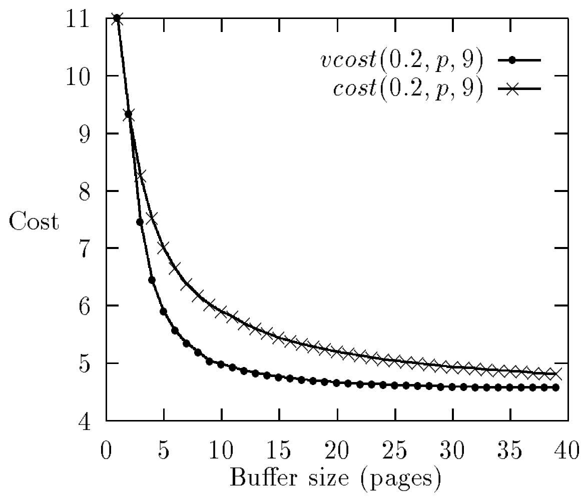

Figure 9: $vcost(0.2, p, 9)$ and $cost(0.2, p, 9)$ as a function of p, $(P = 10)$

Finally, the expected cost per target page is given by

$$vcost(\alpha, p, m) = \frac{1}{V_{target}}(P + V_{total}) \qquad (7)$$

The expected cost $vcost(0.2, p, 9)$ is plotted in Figure 7. Additionally, the expected cost $cost(0.2, p, 9)$ is shown to be higher than $vcost(0.2, p, 9)$, particularly for a small number of buffers.

If pages are always transferred exactly in the order specified in the read request, a separate buffer page for empty pages is not needed. We can let the last target page overwrite the last buffer which was used for collecting empty pages.

7 Discussion

In this paper, we investigated how to rapidly read a (known) set of pages from a file stored in a contiguous area on disk. The basic idea is to reduce the number of read request (positioning operations) by making use of additional buffer space and "large reads", that is, read request transferring multiple adjacent pages. If it is advantageous to do so, a read request may include some gaps. A gap is a contiguous sequence of empty pages, that is, pages not containing any required records.

We showed that an optimal read schedule can be found by computing the shortest path in a certain graph. Then a simplified algorithm was proposed and it was found (experimentally) that it produces close-to-optimal read schedules. We derived three cost functions for the algorithm. For two special cases simple functions in closed form were found. For the most general case, the cost function must be computed numerically by a simple recurrence relation. Additionally, we presented an algorithm that exploits vector reads and analyzed it performance. A vector read assigns each of the empty pages of a read request to the same position in the buffer.

The idea of using 'large reads' is not new, of course. The main contribution of this paper is in the analysis. Our analysis quantifies the benefits of "large reads" but, unfortunately, under a somewhat simplistic cost model. The improvement depends on the number of target pages, the size of the buffer, how well the file is clustered on disk and the architecture of the underlying disk. In order to make the analysis tractable, we had to make several simplifying assumptions. Most important for our analysis are the following three assumptions: the file is stored contiguously on secondary storage, the target pages are uniformly distributed in the file, and the cost of a positioning operation is fixed.

The first assumption can be relaxed. The fraction α of target pages can also be used to model the situation of non-contiguous files as long as the target pages are uniformly distributed in an contiguous area of the disk. The area can be additionally populated by pages of other files and by "free" pages of the disk manager. If we know the actual layout of the file on disk, our scheduling algorithms can still be used. The pages on disk can be brought into a linear order (e.g. according to their cylinders, tracks, and positions on tracks) and the algorithms for finding read schedules can then be applied to this sequence.

The second assumption (target pages are uniformly distributed) can be observed in many applications. Such an access pattern is likely, for example, when a secondary index is used for performing a selection query on a file.

Our cost model disregards the partitioning of disks into cylinders and tracks and the existence of multiple read/write heads. We simply charge a fixed cost for each position operation. Although the position cost varies greatly, this assumption is very common in practice, e.g. I/O-time of a query is usually measured in the number of required disk accesses. It is clearly a blatant simplification. However, relaxing this assumption makes the analysis much harder and is left as an open problem.

The reader may have noticed our implicit assumption that the order in which pages are read (and processed) is unimportant. Changing this assumption leads to an interesting (and open) problem. Even if target pages have to be processed in a certain order, we may still read them in a different order provided that enough buffer space is available to store a page until it is processed. This provides some freedom that can be exploited to find a better read schedule than the one dictated by the processing order.

References

[HSW 88] Andreas Hutflesz, Hans-Werner Six, Peter Widmayer: 'Globally order preserving multidimensional linear hashing', Int. Conf. on Data Engineering 1988, 572-579.

[McF 90] R. McFadyen: 'Sequential access in files used for partial match retrieval', Ph.D. thesis, University of Waterloo, 1990.

[Pal 85] Prashant Palvia: 'Expressions for batched searching of sequential and hierarchical files', ACM TODS, Vol. 10, No. 1, 1985, 97-106.

[Pal 88] Prashant Palvia: 'The effect of buffer size on pages accessed in random files', Information Systems, Vol. 13, No. 2, 1988, 187-191.

[Smi 78] A. J. Smith: 'Sequentiality and Prefetching in Database Systems', ACM TODS, Vol. 8, No. 3, 1976, 223-247.

[WWS 83] Kyu-Young Whang, Gio Wiederhold, Daniel Sagalowicz: 'Estimating block accesses in database organization: a closed noniterative formula', CACM, Vol. 26, No. 11, 1983, 940-944.

[Wei89] G. Weikum: 'Set oriented disk access to large complex objects', Proc. Int. Conf. on Data Engineering 1989, 426-433.

[Yao77] S. B. Yao: 'Approximating block accesses in database organizations', CACM, Vol. 20, No. 4, 1977, 260-261.