Information Processing Letters 36 (1990) 141–145
North-Holland

1 November 1990

# THE OPTIMUM EXECUTION ORDER OF QUERIES IN LINEAR STORAGE

John G. KOLLIAS
*National Technical University of Athens, Greece*

Yannis MANOLOPOULOS
*Aristotelian University of Thessaloniki, Greece*

Christos H. PAPADIMITRIOU
*University of California, San Diego, CA, USA*

Communicated by D. Gries
Received 26 March 1990

Consider a file that resides in a linear storage device with one read head. Suppose that several queries on the file must be answered simultaneously with no prespecified order. To satisfy the $i$th query the head must be located at point $L_i$ of the file and traverse the file up to point $R_i$ without interruptions, where $1 \leqslant L_i < R_i \leqslant N$ denote positions in the file. We wish to find the execution order that minimizes the total time to service all queries, measured as the total distance traversed by the head. Although this is obviously a special type of traveling salesman problem, we show that the optimum sequence can be determined by a simple algorithm in $O(n \log n)$ time. The case in which the head may traverse a file in reverse is similarly solved.

*Keywords*: Databases, analysis of algorithms, batched disk queries

## 1. Introduction

The optimal execution of a set of batched queries in a disk is an old and honored problem (see, e.g., [2,3,12,15]). Its significance declined with the replacement of file systems by more sophisticated database technology. More recently, however, the problem has reemerged in different contexts and guises, as an important optimization problem *within* advanced database systems. For example, it arises as the final, physical level, optimization phase in the implementation of conjunctive queries (one of the most basic optimization problems in databases and AI, see [4,1,12,11]). Also, it is one of the basic optimization problems arising in the efficient implementation of the rule satisfaction subsystem (sometimes called *inference engine*) of logic databases and knowledge bases [14]. It is therefore quite timely that the corresponding computational problem—which a priori seems quite hard—is completely and satisfactorily solved in the present paper.

The problem can be posed as follows: Consider a file that resides in $N$ consecutive locations of a linear storage medium such as a disk or a tape. We identify each location by a number $i$, $1 \leqslant i \leqslant N$; it may correspond to a block (for files in magnetic tape), bucket, page, or cylinder (for files in a disk). We must service a batch of $n$ queries on this file; the queries are of various origins and degrees of complexity, e.g. queries based on single or range primary key values, secondary key retrievals, etc.

Each query $q_i$ is satisfied by a set of records residing at a subset of the $N$ locations. In fact, most file organization schemes provide mechanisms for determining quickly the set of locations

containing the records satisfying the query. Thus, for each query $q_i$ we can assume that we know the first (leftmost) location that satisfies it, call it $L_i$, as well as the rightmost, $R_i$. We assume that, in order to answer $q_i$, the head must traverse all locations between $R_i$ and $L_i$ without interruptions and from left to right. In other words, we rule out techniques that would interleave searches for more than one query, or that may traverse the records in reverse order (we later relax this latter assumption). We assume that in the beginning the head is positioned at location 0, and it need not return after the end (actually, the problem in which the head returns to position 0 after processing all queries can be solved in a similar—in fact, slightly simpler—manner). The problem is, how to sequence the queries so as to optimize the travel time of the head? Naturally, there is no way to save the travel time from $L_i$ to $R_i$ for each query $q_i$; however, the precise order of answering the queries can have a dramatic effect on the total travel time *between* queries.

For example, take the six queries shown in Table 1. Satisfying them in the order $q_1$, $q_2$, $q_3$, $q_4$, $q_5$, $q_6$ would require the head to travel between answering queries from location 0 to $L_1$ (remember, the head resides initially in location 0), from $R_1$ to $L_2$, from $R_2$ to $L_3$, from $R_3$ to $L_4$, from $R_4$ to $L_5$, and from $R_5$ to $L_6$, which would result in a total travel time between queries of $12 + 85 + 13 + 33 + 30 + 8 = 181$. The optimal order, $q_2$, $q_4$, $q_3$, $q_1$, $q_5$, $q_6$ (which we shall explain how to find easily) requires only $5 + 10 + 3 + 41 + 20 + 8 = 87$ travel time between queries. The *query sequencing problem* is to find the sequence that results in the smallest travel time.

In another variant of the problem, we allow the head to answer a query by traversing its range in reverse order. In other words, we do not care if the records of a query are retrieved in reverse order, as long as they are not intermixed with records from other queries. In this case, a better traversal sequence is possible, namely $q_2$, $q_1$, $q_6$, $q_5$, $q_3$, $q_4$ (of which only the first two are traversed in the forward direction, and all others backwards) with extra travel of 49. We call this the *two-way query sequencing problem*.

Table 1
An Example

| Query | $L_i$ | $R_i$ |
|---|---|---|
| $q_1$ | 12 | 90 |
| $q_2$ | 5 | 30 |
| $q_3$ | 43 | 53 |
| $q_4$ | 20 | 40 |
| $q_5$ | 70 | 80 |
| $q_6$ | 72 | 82 |

These problems have been studied extensively in the past (see, for example, [2,3] and the survey paper by Wong [15]). The thrust of previous work was to devise simple motion rules for the head (example: "Service next the query with starting record closest to the current head position.") whose performance is good *on the average*, under the assumption that $R_i$'s and $L_i$'s are somehow uniformly distributed. It is easy to see that these rules, although efficient on the average, are in general suboptimal with respect to worst case (this observation motivated our work). In fact, if the queries originate from many different families, such as range queries, individual queries, batches of primary key queries, secondary key queries, etc., it should be expected that their statistical parameters may differ significantly from the uniformly distributed case. The main asset of the rules analyzed in the literature, besides their good average-case performance under probabilistic assumptions, has been their simplicity. In this paper, we show that finding the *exact optimum processing sequence* does not require much more sophisticated algorithms for head motion; the most complex operations of our algorithm involve sorting the $L_i$'s and $R_i$'s and manipulating in an elementary way a balanced tree containing the $R_i$'s and $L_i$'s.

The algorithm that we propose is of a sort rather common in the traveling salesman literature, sometimes called *subtour patching* (see [9, Chapter 4]). The basic idea is to find the optimal assignment (which may be disconnected), and then "patch" the pieces in an optimal way. This technique has been used in the past in order to solve other variants of the traveling salesman problem, including one that is superficially very similar to ours: Optimize head motion in a *circular storage medium*, in which record $N$ coincides with record

0 [5,6]. However, the technique in these papers does not apply to the *linear* storage medium problem, which we discuss and solve in this paper.

## 2. The algorithm

The query sequencing problem can be formulated as a traveling salesman problem [9]: We create a "city" $c_i$ for every query $q_i$, and an extra city $c_0$ (for the initial position of the head at location 0). The distance from $c_i$ to $c_j$, $i, j \geq 1$, $i \neq j$, is taken to be $|R_i - L_j|$—the amount of extra travel between queries that results if query $q_j$ is serviced immediately after $q_i$. The distance from $c_0$ to $c_i$ is $L_i$—the initial extra travel—and the distance between $c_i$ and $c_0$ is 0—no charge for ending up at $R_i$. Notice that the distances are not symmetric. We wish to find the shortest *tour* starting from $c_0$, visiting all cities, and ending in $c_0$. The shortest tour is obviously the query sequence that induces the least extra travel time.

A tour can be considered as a *connected* directed graph with all nodes having indegree one and outdegree one. A directed graph with all nodes having indegree and outdegree one (but not necessarily connected) is called an *assignment*. As with many algorithms for special cases of the traveling salesman problem (see [9, Chapter 4]) our algorithm first finds an optimal assignment, and then enforces connectivity in an optimal way. There is a polynomial-time algorithm for finding the shortest assignment (see, e.g., [10]), but in our special case it can be done much easier as follows: We sort the $L_j$'s and the $R_j$'s in increasing orders separately, and call the two resulting permutations $\lambda$ and $\rho$: $i \leqslant j$ implies $R_{\rho(i)} \leqslant R_{\rho(j)}$ and $L_{\lambda(i)} \leqslant L_{\lambda(j)}$. We let $\rho(0) = \lambda(n+1) = c_0$.

**Lemma 1.** *The shortest assignment contains the arcs* $(\rho(i), \lambda(i+1))$, $i = 0, \ldots, n$.

**Proof.** See, e.g., [8]. □

We shall be constructing the optimum tour of the cities, as a set of arcs $(c_i, c_j)$, where we say that $(c_i, c_j)$ is in the tour if we visit city $c_j$ immediately following city $c_i$. From the optimum tour, we can derive immediately the optimum head motion: Initially, the head goes to $L_i$, where $(c_0, c_i)$ is in the tour. After this, if the head is on $L_i$, then it should move to $R_i$ answering query $q_i$. If the head is in $R_i$, then it should next go to $L_j$, where $(c_i, c_j)$ is in the tour. Finally, if $(c_i, c_0)$ is in the tour, the process is over, and the head may stay at $R_i$. Up to now, the "tour" under construction contains the arcs in the optimum assignment.

Recall that the optimum assignment may consist of several disconnected cycles, denoted $K_1, \ldots, K_k$. In the example of Table 1 the optimum assignment consists of the arcs $(c_0, c_2)$, $(c_2, c_1)$, $(c_1, c_0)$, $(c_4, c_4)$, $(c_3, c_3)$, $(c_5, c_6)$, $(c_6, c_5)$, and thus has four cycles, namely $(c_0, c_2, c_1)$, $(c_3)$, $(c_4)$, and $(c_5, c_6)$. Our next task is to "patch" these cycles together in an optimal way, to obtain the shortest tour. The cycle involving $c_0$ ($(c_0, c_2, c_1)$ in our example) is called the *basic* cycle, and is denoted $K_1$; it also contains $c_{\lambda(1)}$ and $c_{\rho(n)}$ (the queries with the leftmost and rightmost endpoints). Our algorithm will repeatedly merge nonbasic cycles among themselves and with the basic cycle, until there is only one cycle, namely the optimum tour.

The merging is done in two phases. In the first phase we perform mergings that add nothing to the cost of the final solution, and in the second phase we perform mergings that increase the cost. Define the *span* of a nonbasic cycle $K_j$ to be the interval between the leftmost $L_i$ and the rightmost $R_j$ in the cycle. The span of the basic cycle is a set of intervals, namely the union of all intervals $[R_i, L_j]$ for each consecutive pair of cities $c_i, c_j$ in $K_1$ (this special treatment of $K_1$ reflects the fact that there is no travel from $R_{\rho(n)}$ back to location 0).

In the first phase we repeat the following step: Find a pair of cycles with intersecting spans, and merge them at no extra cost. The merging is done by finding two arcs $(c_i, c_j)$ and $(c_k, c_l)$, one in each cycle, such that the intervals $[L_j, R_i]$ and $[L_l, R_k]$ intersect (this is always possible for cycles with intersecting spans). We then merge the two cycles by replacing the two arcs $(c_i, c_j)$ and $(c_k, c_l)$ by $(c_i, c_l)$ and $(c_k, c_j)$. For example, we notice that the span of cycle $(c_4)$ overlaps with the span of the basic cycle, and thus we replace arcs $(c_2, c_1)$

and $(c_4, c_4)$ by $(c_2, c_4)$ and $(c_4, c_1)$; the two cycles are thus merged in one (basic) cycle, $(c_0, c_2, c_4, c_1)$. Obviously, this results in no extra cost. Repeating this as many times as possible, we end up with an optimal assignment whose cycles have disjoint spans. (Notice that we postpone the description of the efficient way to implement these manipulations until the detailed analysis of the running time of the algorithm.) In the first phase, therefore, we do nothing more than finding a particular optimal assignment (the one that has the fewest possible cycles).

In the second phase we find the optimum tour by repeatedly merging cycles with disjoint spans. Define the *distance* of two cycles to be the smallest distance between two points of the corresponding spans. For example, the distance between cycles $(c_3)$ and $(c_5, c_6)$ is 17, and the distance between $(c_3)$ and the basic cycle is 3. We consider thus the cycles as nodes with these distances, and find their minimum spanning tree [8,10]. Finally, for each two cycles that are connected in the minimum spanning tree, we find two arcs, $(c_i, c_j)$ and $(c_k, c_l)$, one in each cycle, such that the points $R_i$ and $L_l$ realize the distance of the cycles. We then merge the two cycles by replacing the two arcs $(c_i, c_j)$ and $(c_k, c_l)$ by $(c_i, c_l)$ and $(c_k, c_j)$. In our example, the shortest distance 3 will definitely be in the spanning tree, and thus we replace $(c_4, c_1)$ and $(c_3, c_3)$ by $(c_4, c_3)$ and $(c_3, c_1)$, thus merging the two cycles at a cost of 6, twice the distance. We continue merging the cycles that are connected in the shortest spanning tree, until a single cycle results.

That the algorithm described above yields an optimal tour can be shown as a consequence of [9, Chapter 4, Theorem 15]. However, we shall give an independent proof.

**Lemma 2.** *The tour resulting from repeatedly merging the cycles of the optimum assignment as described above is optimum.*

**Proof.** Given any tour, consider the spans of the cycles. Since the tour is connected, there are arcs of the form $(R_i, L_j)$ going in and out from each span. Consider then the portions of the arcs within the span. Since at each span boundary the number of $L_i$'s and $R_i$'s to its left is balanced (and thus to its right also), these portions will make up an assignment of the cities involved, and therefore a tour. Also, again because of the balanced $L_i$'s and $R_i$'s, there is an arc leaving and an arc coming into each span, both from the same boundary. It follows that any tour can be decomposed into tours of the cycles of the assignment, plus pairs of arcs connecting the spans. Since the constructed tour optimizes both parts (it consists of the optimal assignment, plus a shortest spanning tree of the cycles), it is optimum. □

It remains to describe how we can implement all these steps efficiently. Sorting the $L_i$'s and $R_i$'s can be done in $O(n \log n)$ time. We can search for cycles with intersecting spans by inserting the endpoints of the spans in a balanced search tree one after the other, and merging each time an intersection is detected. Insertion and detection takes $O(\log n)$ time, and there are at most $2n$ of these. The resulting tree will also contain the spans from left to right, and thus information concerning the leftmost and rightmost points and the distances is also immediately available. Phase two can be similarly carried out in $O(n \log n)$ time.

**Theorem 3.** *The query sequencing problem can be solved exactly in $O(n \log n)$ time.*

The two-way query sequencing problem can also be solved by a very similar algorithm, which we outline below. In this case we find not the optimal assignment, but the optimum *matching* of the $L_i$'s and $R_i$'s, now considered as members of one set. We then merge the resulting cycles exactly in the same way. The details, very similar to those above, are omitted.

**Theorem 4.** *The two-way query sequencing problem can be solved in $O(n \log n)$ time.*

Finally, as an open problem, we propose the intermediate case, in which certain queries *must* be answered in the forward direction, and the rest in the backward direction. The techniques used above do not appear to generalize to this variant.

## References


[1] A.V. Aho, Y. Sagiv and J.D. Ullman, Equivalence among relational queries, *SIAM J. Comput.* (1978).

[2] J. Bitner and C.K. Wong, Optimal and near-optimal algorithms for batched processing in linear storage, *SIAM J. Comput.* **8** (4) (1979).

[3] F.W. Burton and J.G. Kollias, Optimizing disk head movements in secondary key retrievals, *Comput. J.* **22** (3) (1979).

[4] A. Chandra and C. Merlin, in: *Proc. 1977 STOC Conference* (1977).

[5] S.H. Fuller, An optimal drum-scheduling algorithm, *IEEE Trans. Comput.* **21** (1972) 1153–1165.

[6] R.S. Garfinkel, Minimizing wallpaper waste, Part I: A class of traveling salesman problems, *Oper. Res.* **25** (1977) 741–751.

[7] J.G. Kollias, An estimate of seek time for batched searching of random or indexed sequential files, *Comput. J.* **21** (2) (1978).

[8] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (1977).

[9] E. Lawler, J.K. Lenstra, A. Rinooy Kan and D. Shmoys, eds., *The Traveling Salesman Problem* (Prentice-Hall, Englewood Cliffs, NJ, 1986).

[10] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

[11] B. Schneiderman, Reduced combined indexes for efficient multiple attribute retrieval, *Inform. Systems* **2** (2) (1977) 149–154.

[12] B. Schneiderman and V. Goodman, Batched searching of sequential files of tree-structured files, *ACM Trans. Database Systems* **1** (3) (1978) 268–275.

[13] D.E. Smith and M.R. Genesareth, Ordering conjunctive queries, *Artificial Intelligence* **26** (1985) 171–215.

[14] J.D. Ullman, Principles of knowledge-based systems, Manuscript (1988).

[15] C.K. Wong, Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems, *Comput. Surveys* **12** (2) (1980).