# Quick- instead of Merge-sort for pipelined systems

Ahmad H. Khalaila, Frank Eliassen
Department of Computer Science
The University of Tromsø
9037 Breivika, Norway
{ahmad,frank}@cs.uit.no

May 1997

**Abstract**

This paper proposes an external sorting algorithm for large data as an alternative to the widely used merge-sort algorithm. The algorithm we present is an application of the widely known quick-sort algorithm to large sequences of data stored externally on a disk device.

The problem with the merge-sort algorithm is not its time complexity, but the large amount of time it requires to output its first results. This is a serious problem in the context of pipelined processing, since the operations consuming its result will have to wait all that time before they can start their processing, thus limiting the degree of vertical parallelism achievable by pipelined processing.

Using quick-sort instead of merge-sort for external sorting in pipelined data processing systems results in an optimization in the order of $logN$ (where N is the size of the data sequence to be sorted) for the entire query pipeline where the sorting operation is involved.

## 1 Introduction

The sort operation imposes a given order on a sequence of data tuples using some comparison predicate defined on some attributes of these tuples. In order to figure out the location of a data object within the sequence of data objects, one has to traverse the whole sequence. In order to sort the entire sequence most algorithms take an amount of time proportional to $O(N^2)$, where $N$ is the length of the sequence. Many other widely-known algorithms such as heap-sort and merge-sort, take a time proportional to $O(NlogN)$.

A very interesting algorithm which is widely applied to sequences of data residing in main-memory, is the quick-sort algorithm. This algorithm has a worst-case complexity of $O(N^2)$, and an average-case complexity of $O(NlogN)$.

Merge-sort is the sorting algorithm normally applied in case the data sequence is very large and therefore stored on disk.

In this paper, we argue that a quick-sort algorithm for external sort (i.e. sorting a large data sequence that is stored on an external device, in this case the disk) is more suitable for pipelined processing than the merge-sort algorithm.

## 2 External quick-sort

An outline of the external quick-sort algorithm is presented below.

**Algorithm 2.1** An external quick-sort algorithm

```
$equick\_sort\ (file, first, last) \equiv$
        Var:
            $pivot : integer;$
            $file1, file2 : SEQ[T];$
        Program:
                If $(first = last)$
                    $output(quick\_sort(file, first));$
                Else
                    $pivot \leftarrow pick\_pivot(file, (first + last)/2);$
                    $partition(file, pivot, file1, file2);$
                    $equick\_sort(file1, 1, |file1|);$
                    $equick\_sort(file2, 1, |file2|);$
```

The above specification of the quick-sort is self-explanatory; *pickpivot* is the function that picks a value for pivot from the middle of the file. This value is used by the *partition* routine which partitions $file$ into $file1$ and $file2$. $File1$ will contain all the data items having a value less than or equal to the value of *pivot*, and $file2$ all the items having a value greater than that of *pivot*.

### 2.1 Complexity of the first output

Although the complexity of this external quick-sort algorithm has a worst case scenario of $O(N^2)$, this is very bad compared to the worst case of merge-sort. The quick-sort algorithm will most often start to output sorted packets of data much earlier than the merge-sort algorithm. It is this property of the external quick-sort algorithm that makes it very appealing in the context of pipelined data-processing systems.

The merge-sort starts to output data after carrying an amount of work equal to $O(N(logN-1))$, i.e. after it has computed the first $(logN-1)$ merge rounds.

The best case behavior of quick-sort occurs when the value chosen for pivot is the value at the middle of the sequence. In such a case, the first call to the quick-sort routine will partition the sequence in half, the second call will partition a half of the sequence into two halves, and so on. There will be $logN$ calls to the routine before the first data block is output. In the first call the input sequence has length $N$, in the second call $N/2$, in the third call $N/4$, and so on. In the $i^{th}$ call the sequence has length $N/2^i$, and in the last call before the first data block is output a length of $N/2^(logN)$.

Thus, the quick-sort algorithm will in its best case output data after it has computed an amount of work that has a time complexity $T$ where follows:

$$\begin{aligned} T &= N + N/2 + N/4 + \cdots + N/2^k \\ &= \Sigma_{i=0}^{k} N/2^i \\ &= N\Sigma_{i=0}^{k} 1/2^i \\ &= N\tfrac{1-(1/2)^{k+1}}{1-1/2} \\ &= N(2-1/2^k) \\ &= 2N-1 \end{aligned}$$

where $k = log_2 N$.

The above equation tells us that quick-sort will output its first block of data after $2N-1$ steps.

However, the worst case is in fact quadratic (i.e. $O(N^2)$) for the first block in order to be output, and it is associated with the following scenario. The first call partitions the input sequence into a subsequence of length $N-1$, and another of length 1, the second call partitions the sequence of $N-1$, into a subsequence of $N-2$, and another subsequence of length 1, and so on. The computational cost before the first data block is output is equal to $N + (N-1) + (N-2) + \cdots + 1 = N(N+1)/2$.

Notice that what we call the best behavior of quick-sort is not the one associated with the least amount of time before the first data block is output, but the best behavior of the quick-sort algorithm in general. That is because the best case for the least amount of work to be carried out before first data block is output is $N$, and it occurs when the first call partitions the input sequence into a subsequence of length 1, having all the items less than or equal to the value of pivot, and another subsequence of length $(N-1)$ having all items greater than pivot.

We also know that the average behavior of quick-sort has a complexity of $NlogN$. The average amount of work performed before the first data block is output (i.e. $2N-1$) is associated with this general average behavior of quick-sort.

Comparing the complexity of the first output of quick-sort with that of merge-sort, we find that quick-sort is on the average superior in the order of $logN$.

## 3 Conclusion

Our conclusion is that using quick-sort for external sorting instead of merge-sort may very often result in an optimization in the order of $logN$ for the entire query pipeline in which the sorting operation is engaged.

Although our idea of using quick-sort instead of merge-sort for external sorting is based on a theoretical analysis only, it remains to be tested by some experiments.