to copy data units, move them, and delete them if they are not used. Using these operations, our goal is to minimize EST while keeping the number of redundant copies as low as possible. After constructing a cache oblivious layout of the data set to get an initial ordering of data units, we copy one data unit to another location, and reassign one or more of the access requirements that uses the old copy of the data unit to the new copy, such that the EST is reduced. If all the access requirements that used the old copy, now use the new copy of the data unit, then the old copy is deleted. We repeat this copying and possible deletion of individual data units until our redundancy limit has been reached.

Blocks to Copy: Note that the span of an access requirement does not change by moving an interior data unit to another interior location. Cost can be reduced only by moving the data units that are at the either ends of the access requirement. This observation greatly reduces the search space of data units to consider for copying. Additionally, for the sake of simplicity of the algorithm, we operate on only one data unit at a time.

Location to Copy: Based on the above observation, given an access requirement, we can possibly move the beginning or the end data units of an access requirement to its interior. This will reduce its span, thus reducing the EST for the layout. However, if the new location of the data unit is in the span of other access requirements, such as location 11 in Figure 1, it increases the span of those accesses by one unit. Let j be the new location for the start or end data unit of an access requirement i. Let ΔA_i denote the change in the span of the access requirement i by performing this copying operation. ~~With the access requirements whose span overlaps j, their span would be increased by one.~~ Let k_j denote the number of access requirements whose span overlaps at location j. The reduction in EST by performing this copying operation is given by

$$\Delta EST_C(i,j) = \Delta A_i - k_j$$

where C denotes *copying* the data unit for access requirement i to the location j. We find the location j ~~that is~~

$$argmax_j(\Delta EST_C(i,j))$$

~~which signifies where the start or end data unit of the access requirement i needs to be copied, using a simple linear search through the span of i.~~

Assignment of Copies to Access Requirements: The above operation would result in two copies of the same data unit, say d_{old} and d_{new}. Clearly the new copy d_{new} in location j will be used by the access requirement i. But d_{old} could be accessed by multiple access requirements. All other access requirements that accesses d_{old} can either continue to use d_{old} or use d_{new} depending on the overall effect on their span. Let S be the set of access requirements whose span does not increase by using d_{new} instead of d_{old}. Now the total benefit by copying the data unit d_{old} of the access requirement i to the new location j is

$$\Delta EST_C(i,j) = \Delta A_i - k_j + \sum_{s \in S} \Delta A_s. \quad (1)$$

Moving versus Copying: Let T be the set of access requirements whose span will increase by accessing d_{new} instead of d_{old}. Further, let k_{old} be the number of access requirements in whose span d_{old} is. If we force all the access requirements that uses d_{old} to use d_{new} and then delete d_{old} – in other words, if we move d instead of copying – then the benefit of this move would be given by

$$\Delta EST_M(i,j) = \Delta A_i - k_j + \sum_{s \in S} \Delta A_s + \sum_{t \in T} \Delta A_t + k_{old}$$

$$\Delta EST_M(i,j) = \Delta EST_C(i,j) + \sum_{t \in T} \Delta A_t + k_{old}$$

where $\Delta EST_M(i,j)$ gives the benefit of *moving* a start or end data unit of the access requirement i to position j. Note that each of ΔA_t is negative. Hence the benefit of moving might be more or less than the benefit of copying depending on the relative values of $\sum_{t \in T} \Delta A_t$ and k_{old}. But the main advantage of moving instead of copying is that this operation does not increase the redundancy thus it keeps the storage requirement the same. So we perform moving instead of copying as long as $\Delta EST_M(i,j)$ is positive.

Data Unit processing order: We now need to figure out how to use this information to decide in what order the copying and moving should be done. We will make two heaps: E_M and E_C. The E_M heap will organize the move operations and consist of the values of $\Delta EST_M(i,j)$ for the start and end data units for all access requirements i where the units are put in their optimal location j. The E_C heap will be the same ~~thing~~ except it will organize the copy operations and consist of the values of $\Delta EST_C(i,j)$.

~~You empty~~ the E_M heap first as long as the top of the heap is positive. After each removal and processing, ~~you update~~ ΔEST_M and ΔEST_C of the affected access requirements and ~~update~~ the corresponding heaps. If there are no more data units where ΔEST_M is positive, then ~~process~~ the top of E_C. After processing ~~a copied~~ data unit from the top of heap E_C, the heaps E_C and E_M are again updated with new values for the affected access requirements. If this introduces an element in the top of E_M heap with positive values, the E_M heap is processed again. This process gets repeated until ~~we run out of space for more redundant data units~~. As a summary, the psuedo-code of this algorithm is shown as algorithm 1.

```
Input: Data units and their access requirements (AR) ;
for start and end unit of each AR i do
    Find optimal location j for copy;
    Calculate ΔEST_M(i,j) and insert into E_M ;
    Calculate ΔEST_C(i,j) and insert into E_C ;
end
while true do
    while top element of E_M is positive do
        Pop top element and move the data unit to its destination ;
        Update E_M and E_C ;
    end
    if there is more space for redundancy then
        Pop top element and copy the data unit to its destination ;
        Update E_M and E_C ;
    else
        break
    end
end
```

Algorithm 1: Pseudo-code for our algorithm

4 Complexity Analysis

We now analyze the running time and storage requirements of our algorithm. Let N be the number of data units and A be the number of access requirements. We will use k as the average span of a single access requirement. The variable Q will represent the number of data units that can be copied as specified by the user. Let r be the amount of redundancy if we have a single-seek layout [Jiang et al. 2013]. The average number of overlapping spans for a single data unit ends up being $O(rN)$.