Algorithms for Massive Data Sets

Student name and id: Alexander R. Johansen (s145706)

Collaborator name(s) and id(s): Jonas (s142957)

Hand-in for week: 12

1 Tree Layout in External Memory

The 1.1: $TransposeMatrix(n*m)$ Problem can be solved in $O(\frac{n*m}{B})$ I/O's.
The 1.2: The $TransposeMatrix(n*m)$ assumes $M \geq B^2$.
The 1.3: $InPlaceTransposeMatrix(n*n)$ Problem can also be solved in $O(\frac{n*n}{B} * log_4(\frac{n*n}{M}))$ I/O's.

1.1 TransposeMatrix(n*m)

To transpose the matrix the algorithm loads an index $index_{i,j}$ from the given matrix $n*m$ and outputs it into a new matrix at position j, i in an $m*n$ matrix. To make it I/O efficient several index'es are loaded simultaneous through blocks. The amount of I/O's are defined by the following recursion.

$$IO(n*m) = \begin{cases} O(1) & \text{if } n*m \leq B \\ 2*IO(min(n,m)*\frac{max(n,m)}{2}) & \text{if } n*m > B. \end{cases}$$

The recursion works by cutting the matrix in half(at the largest dimension) at each step, this allows edge cases such as vectors to exists.

Recursion-tree

Looking at the recursion, only the base case has a cost in it. Thus the recursion tree works out as follows.

1. no IOs
2. no IOs

...

i. $O(1)$ for each leaf

As the floor of the tree determines the amount of I/O's I need to calculate the depth and degree. Degree is 2 at every level, thus the amount of leaves are exponentially growing at every level - 2^{depth}.

The size of the matrix is halved at every step, until $n*m \leq B$. Such halving is going to give a depth of $depth = log(n*m) - log(B) = log(\frac{n*m}{B})$.

Thus the amount of leaves is $2^{log(\frac{n*m}{B})} = \frac{n*m}{B}$ Having a cost of $O(1)$ per leaf, I can conclude that the amount of I/O's required is $O(\frac{n*m}{B})$.

1.2 Assumption from 1.1 TransposeMatrix(n*m)

As the blocks are not assumed a square size, a block from the first matrix might cross several blocks in the output matrix and could overwrite information. To avoid such, it needs to be assumed that the memory can span the size of the blocks in both in the lateral and horizontal direction, such that no cross-overs can happen, Thus.
$M \geq B^2$ (appendix A figure related).

1.3 InPlaceTransposeMatrix(n*n)

The algorithm proposed works by dividing the problem into smaller problems, solve these smaller problems and have the small problems swap places to recursively solve the transpose problem. This approach leads me to the following recursion.

$$IO(n*n) = \begin{cases} O(\frac{n*n}{B}) & \text{if } n*n \leq M \\ 4*IO(\frac{n*n}{4}) + O(\frac{n*n}{B}) & \text{if } n*n > M. \end{cases}$$

I am using M in this case, seeing as it leads to less recursions having to be travelled. As the matrix is assumed square, I can further use a split of 4 instead of 2(as the resulting matrices are also going to be square).
Recursion-tree
As opposed to the previous recursion, this recursion has cost at every level(why I tuned it with 4 splits and M). The recursion tree works out as follows.
1. $O(\frac{n*n}{B})$
2. $4*O(\frac{n*n}{4*B}) = O(\frac{n*n}{B})$
...
i. $4^{i-1}*O(\frac{n*n}{4^{i-1}*B}) = O(\frac{n*n}{B})$
Thus with equal cost at every level, the depth will be a muliplicative factor to how costly this algorithm will be. The size of the matrix is cut to a fourth at every step, until $n*n \leq M$. The cut to a fourth will cause a reduction of log w/ base 4 and the stop at M will cause a logarithmic subtraction, resulting in $depth = log_4(n*n) - log_4(M) = log_4(\frac{n*n}{M})$.
Thus with $O(\frac{n*n}{B})$ at every level and $log_4(\frac{n*n}{M})$ levels the resulting IO's will be $O(\frac{n*n}{B} * log_4(\frac{n*n}{M}))$

2 Appendix

2.1 A

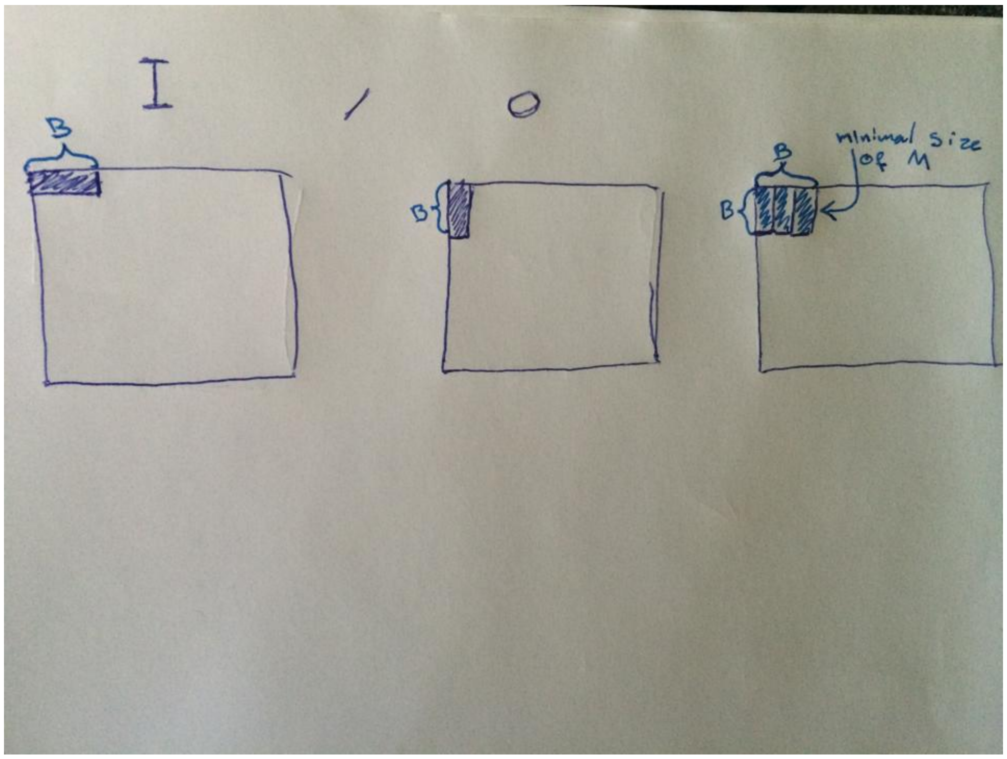

Figure 1: 1.2 Assumption