Ali Nawaz Maan
S4468881

Efficient data structures designed to work on data in External Memory

Data which is relatively large and cannot fit in the machine's internal memory is hard to handle. Large datasets in larger applications are too massive to fit entirely inside the machine's memory. The input/output operations between internal and external memory is a real bottleneck of having the large datasets in external memory and do operations on that data by loading a small chunk in internal memory.

This problem poses a demand for such data structures which are efficient in doing operations such that I/O costs are not much of a problem or can reduce the overhead cost of I/O. External memory is divided into blocks and each block contains some data. Each block also has a memory location reference attached to it which lets the system to access that block in a constant time – excluding seek time and rotational delays. For this reason, it seems best to have data structures which store relative memory addresses of the related blocks to efficiently accessing related data.

(a,b) Trees and B-Trees

The best way to store data in external memory and perform operations on the data, we have to do better than logarithmic time operations keeping in account the block transfer times. (a,b) trees are multiway search trees which store data and reference to the memory location of the block as a map. This is a balanced generalization of (2,4) tree where each node has between a and b children and stores between a-1 and b-1 entries. (Goodrich, Tamassia, & Goldwasser, 2014)

B-Tree is the extension of (a,b) search trees where each node correspond to a specific memory block where some data is stored. Each node stores many data items and has many accessors which store addresses of successor blocks.

The general explanation of B-tree must have one root and it can have as few as two children nodes. If the tree is of order d, each leaf node will then have between d/2 and d children. From the practical considerations, each node represents a block of memory and some content, so the size is usually power of 2 and the number of records in the node is usually even. Also, the order of the tree is usually odd.

Search in B-Tree

The search for a record key k in the B-tree is usually a binary search. The algorithm describes as search the root for item x, Either it is found or is between 2 children keys. This way, corresponding subtree should be searched.

Insertion in B-Tree

In order to insert an item in the tree, we have to maintain its balance. The general algorithm goes as first search the node with a particular item with appropriate location to insert, if the node is not full, insert the item into the node in order. If the node is full, we need to split the node and reorder the tree.

Splitting of a node is somewhat tricky in this case, we have to find middle value in the node corresponding to existing items and new item to be inserted. Keep items smaller than the middle in existing node but put greater items into a new node. Pushing middle up into the parent node will balance the sub tree. If this operation makes the parent node full, split it as well and push the middle to its parent. It involves recursive steps until either some space is found in an ancestor node or a new node is created.

Deletion of items is similar to (2,4) trees and is done by handling underflows. Replace the item to be removed by its in-order successor. Then remove the successor from its leaf node. If this causes and underflow, it can be balanced either by fusion or by transfer.

Dictionary or Hash maps

Another well-known data structure designed for using in external memory is Dictionary or Hash Map or Hash table. This concept provides accessing to a particular data item in constant time. By this implementation, both internal and external space usage can be improved by using extensible hashing. Hash function is implemented using the extension of (a,b) trees I.e. the buffer trees. (Meyer & Sanders, 2003)

The idea is to implement a dictionary which supports predecessor queries in a key set. Each key value contains information about its successor or related keys to aid in reducing seek time and rotational delay and hence supports fast data transfers from external to internal memory. The keys are stored in a linked list in this sense. For each key in the linked list, smallest hash value is stored in predecessor dictionary and a pointer to that relevant block. Searching is then performed for items in that block.

Insertion and deletion is done by inserting or deleting the key to linked list and making constant number of updates to the predecessor dictionary.

Bibliography

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures & Algorithms in Java* (6th ed.). USA: John Wiley & Sons.

Meyer, U., & Sanders, P. (2003). *Algorithms for Memory Heirarchies.* Germany: Springer-Verlag Berlin Heidelberg .