CONTENTS

Selin Cansu Akbaş
191180005

1. B+ TREE

We start by revisiting the concept of the B+-tree structure and the invariants it has to abide by in order to provide a solid foundation for the discussion. First of all, let's take a look at what the B+ Tree is. A B+ Tree is just an extended B Tree with values or pointers stored at the leaf node level, which makes using it for various tasks much simpler. The B-tree is one variant that enables effective sequential file processing while preserving the desired logarithmic cost for locate, insert, and delete operations [1].

1.1. Definition Of B+ Tree Deletion

The B+ tree has three procedures for deleting elements: searching, deleting, and balancing. The delete algorithm's objective is to eliminate the desired entry node from the tree structure. Until no node is identified, we repeatedly invoke the delete procedure on the corresponding node. We traverse along for each function call, using the index to find the node, remove it, and then climb back to the root. B+ trees, like other trees, can be shown as a grouping of three different node types: internal, leaf, and root [2].

1.2. How Deletion Process Is Done For B+ Tree?

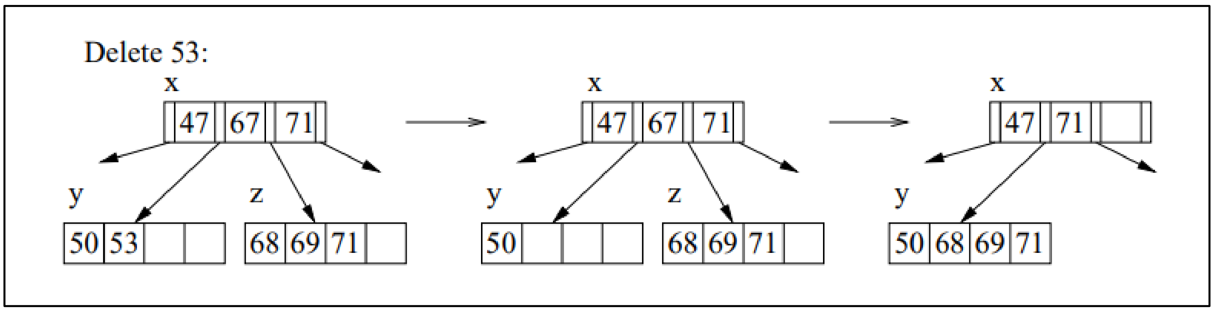
Deletion in a B+ tree is a two-step process:

1- Search for the key to be deleted. This involves traversing the tree and finding the leaf node that contains the key.
2- Once the leaf node is found, the key is deleted from the node. If the key is the only one in the node, the node is deleted as well. If the node still has other keys after the deletion, it is not deleted.

If the node that is being deleted is the root node, then the tree is modified so that the next node becomes the root.

If the node being deleted is not the root and has fewer than the minimum number of keys required (also known as the minimum degree of the tree), then the tree may need to be rebalanced. This can be done by borrowing a key from a sibling node or by merging the node with a sibling [3].

It's important to note that B+ trees are designed to allow efficient searches, inserts, and deletes even when the tree is large, so the deletion process may involve multiple levels of the tree and may require several steps to complete. (Picture 1).



Picture 1. Deletion example is done for B+ tree

As x is a non-leaf node, we must continue the rebalancing procedure there if the parent x has fewer offspring after fusing than t. The router value between the pointers to the two non-leaf nodes in the parent is taken out of the parent and added to the fused node when fusing is applied to them. The worst-case scenario requires that every node along the route from the fused leaf to the root be fused. The condition where fusing begins after one key has been taken from the leaf on the left is depicted in the figure below [4].

1.3. Algorithm Of B+ Tree Deletion

B+ tree deletion consists of 5 steps:

Step 1- Find the leaf node that contains the key value using the input from a key-value pair.

Step 2- Remove the entry from the leaf if the key is discovered. If the leaf satisfies the "Half Full condition," it is finished; if not, it still needs some data additions.

Step 3- If the right leaf sibling is allowed to enter. Then transfer the tiniest entry to the leaf's right sibling. If the leaf's left sibling cannot accept an entry, then transfer the smallest node to that leaf's left sibling. Merge both a leaf and a sibling if it doesn't match the aforementioned two requirements.

Step 4- Recursively deleting the entry that points to a leaf or sibling from the parent is done during merging.

Step 5- The tree's height might alter if there is a merger [5].

2. DYNAMIC HASHING

Dynamic hashing is a method used in computer programming to dynamically increase or decrease the size of a hash table in a hash-based data structure. It allows for the hash table to adapt to changes in the number of elements being stored, ensuring that the table remains balanced and efficient. In dynamic hashing, the hash table is divided into a number of "buckets," each of which is assigned a unique range of hash values. When a new element is added to the table, it is placed in the appropriate bucket based on its hash value. If the number of elements in a particular bucket exceeds a certain threshold, the bucket may be split into two or more smaller buckets, or it may be merged with another bucket to create a larger bucket. This allows the hash table to adjust to changes in the number of elements being stored, ensuring that the table remains balanced and efficient. [6].

2.1. Definition Of Dynamic Hashing Deletion

Dynamic hashing deletion is the process of removing an element from a dynamic hash table. In dynamic hashing, the hash table is able to adjust its size and structure as elements are added or removed to maintain an efficient distribution of data. When an element is deleted from a dynamic hash table, the table may need to adjust its size or structure to maintain this efficiency. This process may involve rehashing elements or altering the size of the table.

2.2. How Deletion Process Is Done For Dynamic Hashing?

To delete a key value,

- Locate it in its bucket and remove it,
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table),
- Merging of buckets can be done (can merge only with a “*split image*” bucket),
- Decreasing bucket address table size is also possible,

- Note: Decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table [7].

2.3. Algorithm Of Dynamic Hashing Deletion

1- First, search for the key to be deleted in the hash table using the hash function.
2- If the key is not found in the table, return an error message.
3- If the key is found, delete it from the table and decrease the count of elements in the table.
4- If the number of elements in the table is less than or equal to the lower load factor limit, rebuild the table using a smaller hash table size.
5- If the key being deleted is the last element in a bucket, adjust the overflow list by deleting the key and updating the pointers.
6- Return a message indicating the key has been successfully deleted [7].

3. LINEAR HASHING

Linear hashing is a method of distributing data across a database by using a hash function to assign a data item to a specific slot in a hash table. The hash function takes the data item and converts it into a numerical value, which is then used to determine the slot in the table where the data item will be stored. The hash function is chosen so that it evenly distributes the data across the table, and the table is resized as needed to ensure that the data is evenly distributed. Linear hashing is often used in database management systems to provide fast access to data and to ensure that data is distributed evenly across the database [8].

3.1. Definition Of Linear Hashing Deletion

Linear hashing deletion is a process in which a data element is removed from a linear hashing table. This is typically done by setting the corresponding hash key to null or deleting the entire record associated with the key. Linear hashing deletion is used to remove data elements that are no longer needed or to make room for new data elements to be added to the table. It is an important part of the linear hashing algorithm, as it helps to maintain the structure and efficiency of the table.

3.2. How Deletion Process Is Done For Linear Hashing?

The deletion process for linear hashing involves the following steps:

- Search for the key to be deleted in the hash table.
- If the key is found, delete it from the hash table by setting the corresponding element to NULL or some placeholder value.
- If the key is not found, return an error message or handle the case appropriately.
- Check if the number of elements in the hash table is below the minimum fill ratio. If it is, then reduce the number of buckets in the hash table by using the overflow bucket as the new bucket.
- Rehash all the elements in the hash table to their new locations based on the new number of buckets.
- Repeat step 4 and 5 until the number of elements in the hash table is above the minimum fill ratio or there are no more overflow buckets to use [9].

3.3. Algorithm Of Linear Hashing Deletion

1- Begin by identifying the bucket that the item to be deleted is located in.

2- Check if the item to be deleted is the only item in the bucket. If it is, simply remove the item and return.

3- If there are other items in the bucket, find the index of the item to be deleted and remove it from the bucket.

4- Check if the number of items in the bucket is less than the minimum number of items allowed in the bucket according to the linear hashing algorithm. If it is, perform a split and redistribute the items in the bucket.

5- Return [9].

REFERENCES

1. Comer, D. (1979). Ubiquitous B-tree. ACM Computing Surveys (CSUR), 11(2), 121-137.

2. Ramakrishnan R. & Gehrke J. (2011). Database management systems (3rd ed. Intern. ed. 2003 14. [print.]). McGraw-Hill.

3. Maelbrancke, R., & Olivié, H. (1995). Optimizing Jan Jannink's implementation of B+-tree deletion. ACM Sigmod Record, 24(3), 5-7.

4. Jannink, J. (1995). Implementing deletion in B+-trees. ACM Sigmod Record, 24(1), 33-38.

5. Pollari-Malmi K. Soisalon-Soininen E. & Ylönen Tatu. (1996). Concurrency control in b-trees with batch updates. Ieee Transactions on Knowledge & Data Engineering 975–984.

6. Zou X. Wang F. Feng D. Zhu J. Xiao R. & Su N. (2022). A write-optimal and concurrent persistent dynamic hashing with radix tree assistance. Journal of Systems Architecture.

7. Enbody, R. J., & Du, H. C. (1988). Dynamic hashing schemes. *ACM Computing Surveys (CSUR)*, *20*(2), 850-113.

8. Larson, P. Å. (1982). Performance analysis of linear hashing with partial expansions. ACM Transactions on Database Systems (TODS), 7(4), 566-587.

9. Litwin, W., Moussa, R., & Schwarz, T.J. (2005). LH*RS---a highly-available scalable distributed data structure. ACM Trans. Database Syst., 30, 769-811.