Indexing

In this chapter, we will introduce indexing concepts, which are essential in file structuring to organize large data on disk. Indexing also helps in attaining efficiency in data access, data search, and memory allocation. This chapter will build the foundation of indexing, and cover various type of indexing, such as linear indexing, Indexed Sequential Access Method ( ISAM ), and tree-based indexing. The chapter introduces linear indexing and ISAM concepts (which are an improvement over linear indexing) using R. The current chapter will also cover advanced tree-based indexing data structures. The following is the list of topics that will be covered in detail:

2-3 trees B-trees

B+ trees

Linear indexing

Indexing is defined as the process of associating a key with data location. The basic field of a data index includes a search key and a pointer. The search key is set of attributes that is used to look up records from a file and the pointer stores the address of the data stored in memory. The index file consists of records, also known as index entries, of the form shown in Figure. 7.1 :

Figure 7.1: Example of index entries

Page 181

﻿

Indexing

Indexing helps in organizing a large dataset. A database has the following generic properties:

The records are in a structured tabular format

Records are searched using single or a combination of keys

Aggregation queries such as sum, min, max, and average are used to summarize the dataset

Indexing in databases is used to enforce a uniqueness into records, which helps in speedy access of data. A database can have several filesystems associated with it by using indexing. This is shown in Figure 7.2 using a store database example. The store database consists of three tables: the Customer table, which stores all customer-related information, the Order table, which stores transactional level information for all orders placed by customers, and the Employee table, which stores all employee-related information:

Figure 7.2: Example of store database

All tables in the store database can be mapped together using primary, secondary, or foreign keys. The columns in the database can be classified as follows:

Primary key : The primary key is the column which uniquely identifies each row in the table. For example, CustomerID , OrderID , and EmployeeID act as primary keys in the tables Customer , Order , and Employee respectively. Secondary key : The secondary keys are one or more columns which do not have a unique sequence. For example, FirstName and LastName in the Employee table are not able to uniquely represent each row of the table, and a second field will be required to make this column act as the primary key. Secondary keys can be used for indexing in M-dimensional feature space.

Foreign key : A foreign key is the column that points to the primary key in other tables. For example, CustomerID and EmployeeID in the table Order act as foreign keys. This key provides an interface for a smooth interaction with other tables in the database.

[ 170 ]

Page 182

﻿

Chapter 7

In linear indexing, the keys are stored in a sorted order, and the value of the key can point to the following:

A record stored in the filesystem Primary key of the dataset Values of the primary key

The indexes can be stored in the main memory or storage disk depending on the size of data and the keys required to map it. For example, a linear index generated based on sequence is shown in Figure 7.3 :

Figure 7.3: Example of linear indexing on sequence

A linear index contains the key field, and each key field has an associated pointer which links it with the actual dataset position. The sorting of the index allows an efficient search query using binary search. Binary search locates the pointers to the disk blocks or records in the file with specific key indices. As data size increases, storing the index in the main memory would not be feasible. To deal with the issue, one solution is to store the index in the hard disk. However, this would make search an expensive process, making the current indexing approach inefficient. The current issue could be addressed by using a multi-level linear index. Multi-level indexes utilize the sorted index property and computation property of binary search to minimize computation time. The computational property of binary search requires log 2 bi block accesses to search for an index with b i block. Each step performed during binary search reduces the search part of the index file by a factor of 2. Multi-level indexing utilizes this property to reduce the part of the index to be searched by a larger blocking factor, also known as fan-out ( fo , where fo is greater than 2), thus

improving the search to log fo bi . For fo equal to 2, there is no computational improvement due to multi-level indexing. An example of second-level linear indexing is shown in Figure 7.4 .

[ 171 ]

Page 183

﻿

Indexing

Figure 7.4 demonstrates a second-level linear indexing in which the first base level is the usual-ordered primary index with a distinct value for each key. Similarly, the second base level is a primary index for the first level. The second level is the block anchor, that is, it has a one entry for each block in the first level:

Figure 7.4: Example of a second-level linear index

The blocking factor or fan-out parameter for all levels is kept the same during indexing. Thus, if the first level has n 1 entries, then the blocking factor is bf , which is also the fan-out factor fo ; so, the first level needs (n 1 /fo) blocks, which is also the number of entries needed for the second level. Thus, with each addition of a level, the number of blocks is reduced by the fan-out factor. This approach can be repeated as many times to create a multi-level indexing, and is repeated until only one block is needed to fit all the indexes. Multi-level linear indexing can be used on any type of index, such as primary, secondary, or clustering, until the first-level index is represented with distinct keys. Linear indexing is quite efficient in structuring datasets. However, the main drawback is that any insertion or deletion operation would require a big change in the linear index, which would affect the computation effort significantly.

[ 172 ]

Page 184

﻿

Chapter 7

ISAM

ISAM, promoted by IBM and Ingres DBMS, also uses the linear indexing philosophy of primary key sorting. In ISAM, files on disk are divided into cylinders on disk. The cylinders are tracks readable from a particular placement of the head on a multiple platter of disk drive. ISAM addresses the limitation of linear indexing of insertion or deletion to some extent due to its static structure, but it is only suitable for small changes. Thus, ISAM is usually applied in databases which are not frequently updated. Static index allocation makes frequent insertion and deletion an expensive process. However, it helps with concurrent access of records, which helps to scale the efficiency of the current data structure. The drawbacks of ISAM with regard to insertion/deletion was later addressed by B-tree based indexing. The ISAM approach was used before the adoption of tree-based indexing:

Figure 7.5: Illustration of ISAM structure

The ISAM data structure consists of a memory-resident cylinder index, cylinder index, and system overflow as illustrated in Figure 7.5 . The memory-resident cylinder keeps the highest-value key from each cylinder. Similarly, each cylinder index keeps the highest-value key for each block. Each cylinder in ISAM consists of the following:

Cylinder index : Each cylinder in ISAM contains an index that keeps updated the highest-valued key for each block.

Records : Data block which stores the records.

Overflow : Overflows are kept in cylinders to allow insertion of records. Based on the cylinder index, the correct overflow is identified for insertion of new records.

[ 173 ]

Page 185

﻿

Indexing

The system overflow in the ISAM structure is used if the cylinder overflow is completely utilized by insertion. This could drastically increase the search time:

Figure 7.6: ISAM node structure

The node structure of ISAM consists of n keys, which are associated with n+1 pointers, as shown in Figure 7.6 . An example of a node structure is shown in the following diagram:

Figure 7.7: Example of ISAM indexing

As records in system overflow increase due to insertion, the performance of ISAM

decreases. This is usually handled by database reorganization of records by minimizing the imbalance in records by updating the memory-resident cylinder index. The more efficient structure of ISAM is utilized in tree-based structures, as discussed in the next section.

Tree-based indexing

Linear indexing is efficient on static databases, that is, records from the database are rarely inserted or deleted. ISAM improves the performance of linear indexing, and can be used for limited updates of the database. As ISAM uses a two-level linear indexing schema, it would break down for a database where the top-level index is already too big to fit into the memory. Thus, as databases become large, we require better organization methods. One approach proposed in Chapter 6 , Exploring Search Options , is that a binary search tree could potentially be utilized for indexing to store the primary and secondary keys. The binary search tree provides an efficient structure to store duplicates, and to perform operations such as deletion and insertion given that sufficient memory is available. However, the only disadvantage with a binary search tree is that it could become unbalanced.

[ 174 ]

Page 186

﻿

Chapter 7

Unbalancing becomes an issue, especially in a scenario when the tree is stored in the disk, as any operation requires data to be loaded from the disk to the memory on the path to leaf node. Thus, to minimize operation time such as insertion or search, it is recommended to store each subtree in the same block, as shown in Figure 7.8 :

Figure 7.8: Example of breaking BST into blocks

In Figure 7.8 , the subtree in block is loaded; thus, in the current scenario, any operation requires two block load. The R script for the BST node structure is as follows:

bstnode <- function(key, value) {

node <- new.env(hash = FALSE, parent = emptyenv()) node$key <- key # Node key node$value <- value # Node Value

node$left <- NULL # left children key node$right <- NULL # Right children key class(node) <- "bstnode" return(node)

}

The load requirement could drastically increase if the tree is unbalanced, as shown in Figure 7.9 , unless the whole tree resides in the main memory, which will keep operation time restricted to O(log n) , where n is the tree depth:

Figure 7.9: Example of unbalanced tree

[ 175 ]

Page 187

﻿

Indexing

The two major challenges to be addressed in tree-based indexing are the following:

How to keep the tree balanced

How to minimize the path from the root node to the leaf node

Balancing the tree in BST is quite expensive as, usually, balancing requires reorganization of data. A 2-3 tree , discussed in next section, is an initial framework to balance a tree by keeping the leaves at the same level. The 2-3 trees are further extended to B-trees, which will be discussed later in the B-trees section.

2-3 trees

A 2-3 tree is a type of tree-based indexing where each internal node in the tree has either two nodes with one key or three nodes with two keys thus, it is classified as a balanced tree. Also, all the nodes in a 2-3 tree are at the same level of tree height. An example of a two- node structure is shown in Figure 7.10:

Figure 7.10: Example of a two-node tree structure

A two-node structure consists of one key and two children/subtrees. All the keys on the left side are smaller, and all the keys on the right subtree are bigger than the key. Similarly, three-node structure has two keys with three children/subtrees, as shown in Figure 7.11 :

Figure 7.11: Example of a three-node tree structure

[ 176 ]

Page 188

﻿

Chapter 7

In the three-node structure, the keys are arranged in a sorted order, the first key being the smallest. All keys on the left of the subtree are smaller than the first key. All the keys in the middle of the subtree are greater than the first key and smaller than the second key. All the keys on the right of the subtree are greater than the second key. The node structure for a 2-3 tree can be represented as follows:

tttnode <- function(lkey=NULL, lvalue=NULL, rkey=NULL, rvalue=NULL) {

node <- new.env(hash = FALSE, parent = emptyenv()) node$lkey <- lkey # left Node key node$lvalue <- lvalue # Node Value node$rkey <- rkey # right Node key node$rvalue <- rvalue # right Node Value node$left <- NULL # left children key node$center <- NULL # left children key node$right <- NULL # Right children key class(node) <- "tttnode" return(node)

}

The node structure for a 2-3 tree consists of two keys and a value pair with three children. The insertion in 2-3 nodes can be performed using steps listed next:

If the tree is empty, create a new node otherwise find a leaf where the value belongs. For example, let's try to add 70 to a new tree:

Figure 7.12(a): Insert a value into an empty tree

The insertion, as shown in the preceding image, can be obtained by creating a node using tttnode , with the following line of code:

extttree <- tttnode(70, 70)

[ 177 ]

Page 189

﻿

Indexing

If another value is to be inserted, it checks whether the root node has space for insertion, else finds the leaf node where the value is to be inserted. For example, adding a new value to extttree will check whether it's vacant, and add the element accordingly:

Figure 7.12(b): Insert second value to extttree

The empty tree can be checked by evaluating both key values, and if there is empty space, then the value will be inserted in the leaf. Before insertion, the value may need to be sorted:

# Function to check if node is empty check\_empty<-function(node){

ifelse((is.null(node$lkey) & is.null(node$rkey)), T, F)

}

The insertion script at the root is as follows:

# Function to insert if the node has empty space leaf\_insert<-function(node, key, val){

if(check\_empty(node)) return(tttnode(lkey=key, lvalue=val)) if(is.null(node$rkey)){

if(key>node$lkey){

node$rkey<-key

node$rvalue<-val

} else {

node$rkey<-node$lkey

node$rvalue<-node$lvalue node$lkey<-key

node$lvalue<-val

}

} else {

node$left<-tttnode(key, val)

}

return(node)

}

[ 178 ]

Page 190

﻿

Chapter 7

If there are three elements in the node, then the median of the node is promoted. For example, if element 80 is added to extttree , then the tree is updated as shown in Figure 7.12(c) :

Figure 7.12(c): Insert element 80 to extttree

The preceding step will be repeated to generate the tree. The generalized pseudocode for element insertion in a 2-3 tree is as follows:

ttinsert<-function(node=NULL, key, val){

if(check\_empty(node)) return(tttnode(lkey=key, lvalue=val)) if(is.null(node$left)) node<-leaf\_insert(node, key, val) ## Add element to internal nodes if(key<node$lkey){

subtree = ttinsert(node$left, key, val) if (identical(subtree, node$left)) {

return(node);

} else {

assign("left", subtree, envir = node) return(node)

}

} else if(ifelse(is.null(node$rkey), T, key<node$rkey)){

subtree = ttinsert(node$center, key, val) if(identical(subtree, node$center)) {

return(node)

} else {

assign("center", subtree, envir = node) return(node)

}

} else {

subtree = ttinsert(node$right, key, val) if(identical(subtree, node$right)) {

return(node)

} else {

assign("right", subtree, envir = node)

[ 179 ]

Page 191

﻿

Indexing

return(node)

} } }

The implementation involves recursively determining the location where the insertion has to be made. There are two major parts to the process of insertion:

Inserting the key and value in the root node

Inserting the key and value in the internal node; the node is the current tree or subtree where insertion will be performed, and key and val are keys and the values/records to be inserted to the tree respectively

The aforementioned insertion process is repeated, and nodes are splitted and promoted as required. For example, let's try and add the values 95 and 99 in two steps. The tree will be updated as shown in Figure 7.12(d) and Figure 7.12(e) :

Figure 7.12(d): Insert element 95 to extttree

Figure 7.12(e): Insert element 99 to extttree

Another critical aspect is searching for values using the key in a 2-3 tree. The search pseudocode within a 2-3 tree is shown as the following R function:

search\_keys<-function(node, key){

if (is.null(node)) return(NULL) # empty node if (node$lkey== key) return(node$lvalue)

if(!is.null(node$rkey) & node$rkey==key) return(node$rvalue) if(key<node$lkey) {

sort\_keys(node$left, key)

} else if(is.null(node$rkey)){

sort\_keys(node$center, key) } else if(key<node$rkey) {

[ 180 ]

Page 192

﻿

Chapter 7

sort\_keys(node$center, key)

} else {

sort\_keys(node$right, key)

} }

Another important aspect is deletion. Deleting a key in 2-3 tree (pretty much in all tree- based data structures) requires deleting the key only in the leaf; thus, the easiest scenario for deletion is when the leaf only contains two keys. However, if deletion is required in an internal node, then deletion occurs at the leaf, and then the deletion effect is possibly propagated up the tree. The details of deletion in tree-based data structure will be covered in Chapter 8 , Graphs , while discussing the generalization of B-trees.

The B-tree is the other most widely used data structure within databases. B-trees are a generalization of the 2-3 trees which address the data retrieval issue from storage devices, as discussed in the next section.

B-trees

Memory efficiency is an important aspect to be considered while designing data structure and algorithms. Memory can be broadly classified into two types:

Main memory (RAM)

External storage such as hard disk, CD ROM, tape, and so on

Data stored in main memory (RAM) has minimal access time thus preferred by most of the algorithms, whereas, if the data is stored in external drives then access times become critical, as it usually takes much longer to access data from external storage. Also, as the data size increases, retrieval become an issue. To deal with this issue data is stored in chunks as pages, blocks, or allocation units in external storage devices and indexing is used to retrieve these blocks efficiently. B-trees are one of the popular data structures used for accessing data from external storage devices. B-trees are proposed by R. Bayer and E. M. McCreight in 1972 and are better than binary search trees, especially if the data is stored in external memory.

B-trees are self-balancing (or height-balanced) search trees, where each node corresponds to a block on an external device. Each node of a B-tree stores data items and the address of successors blocks. The properties of B-trees include the following:

The tree will have a single root node and each node may have one record and two children, if the tree is empty

[ 181 ]

Page 193

﻿

Indexing

Non-leaf nodes of B-tree will have values between d/2 and d-1 sorted records with d/2 +1 and d children, where d is ordered

Keys in the i th subtree of a B-tree node are less than the (i+1) th key and greater than the (i-1) th key (if they exist)

B-trees are self-balancing, that is, all leaf nodes are at the same level

B-trees address all the issues related to data access from external memory, which include the following:

B-trees minimize the number of operations required while performing read/write access from external devices

B-trees keep similar keys on the same page, thus minimizing the access required B-trees also maximize space utilization by distributing data efficiently B-trees are self-balancing, that is, all leaf nodes are at the same level

An example of a B-tree with order four is shown in Figure 7.13 , where the order of the tree is defined by the maximum number of children that each non-leaf node supports:

Figure 7.13: Example of a fourth-order B-tree

A B-tree of the fourth order will have three keys, and internal nodes can have up to four children. A B-tree is a generalization of the 2-3 tree at the d th order, where the order is decided to fill the disk block. The search in a B-tree is a generalization of the 2-3 search strategy by using binary search on the nodes to find whether the key is present. Similarly, B-tree insertion is a generalization of 2-3 insertion by just checking whether a node can be inserted at the key. If there is space for insertion, then the key is inserted, as the node needs to be split. The insertion requires finding an appropriate leaf node where insertion will be performed – if there is space in the leaf node, then it is straightforward and the value is inserted in the leaf node. In cases where the current leaf is already full, then it must be split into two leaves with one storing the current value. The parents are updated to store the new key and the child pointer. If the parents are full, then a ripple effect takes place, and updates are performed till the B-tree properties are satisfied. The ripple effect can go up to root node at the most. In cases where the root node is also full, a new root is created and the current root node is split into two.

[ 182 ]

Page 194

﻿

Chapter 7

An example of insertion in a B-tree is shown in the following diagram:

Figure 7.14: Example of insertion order B-tree (element 105 is inserted in leaf node in a B-tree which propagated the update in non-leaf node as leaf node is completely ﬁlled)

Similarly, deletion in a B-tree can be performed as shown in Figure 7.15 . The deletion makes the tree unbalanced at one node where the number of elements is less than d/2 ; thus, readjustment is performed to balance the tree.

The generalization provided by a B-tree makes it a stronger data structure than 2-3, and most of the databases currently use B-tree or its variants. Since the development of B-tree, many variants have been proposed. The two most-used variants are B\* tree and B+ tree. The B\* tree variant of B-tree ensures that every node is at least two-thirds full (instead of half full); thus, in an overflow situation during insertion, B\* tree applies a local redistribution scheme which delays splitting until the nearby nodes are also filled. It usually splits into three nodes instead of two. The other major variation of B-tree is B+ tree, which is the most utilized variant of B-tree, and will be discussed in detail in the next section:

[ 183 ]

Page 195

﻿

Indexing

Figure 7.15: Example of deletion in B-tree

B+ trees

Queries are executed quickly if the data is stored in a sorted order as a linear indexing strategy. Thus, they are very good for static data. However, when it comes to adding operations such as insertion and deletion, they are not good, as it requires rewriting the whole data. When it comes to dealing with stored datasets, B-trees are good in indexing data in external storage devices as they read data in blocks, although any insertion and deletion could potentially lead to a lot of empty space. B+ trees generalize the B-trees to address this issue by keeping all the values in the leaf and the internal nodes only contain the keys. All the leaves which store values are linked, and the internal nodes only help to guide the operations on the leaves. A B+ tree stores up to d references to children and up to d-1 keys. An example of a B+ tree is shown in Figure 7.16 :

Figure 7.16: Example of B+ tree

[ 184 ]

Page 196

﻿

Chapter 7

The B+ tree requires all leaves to be equidistant from the root node. Thus, in the example shown in Figure 7.16 , searching for any value will require three nodes to be loaded from the disk – root node, second-level, and a leaf. In practice, depth d will be a number as large as it takes to fill the block in the disk. For example, if a block size is 6 KB, our block is an 8-byte integer, and each reference is a 4-byte offset, then d is selected as the maximum value by using the equation 8(d-1) + 4d ≤ 8192 , which comes out to be 682. The following properties need to be maintained in a B+ tree:

If a node has more than one reference, then it has keys. All leaves are at the same distance from the root node.

Every N th non-leaf node has k number of keys. All keys in the first child's subtree are less than N 's first key, and all keys in the i th child's subtree (2 ≤ i ≤ k) are between the (i − 1) th key of n and the i th key of n . The root has a minimum of two children.

Every non-leaf, non-root node has children between floor(d/2) and d . Each leaf contains at least floor(d/2) keys.

Every key from the table appears in a leaf in a left-to-right sorted order.

The pseudocode for a B+ tree node using R is shown as follows:

bplusnode<-function(node=NULL, key, val){

node <- new.env(hash = FALSE, parent = emptyenv()) node$keys<-keys node$child<-NULL node$isleaf<-NULL node$d<-NULL

class(node) <- "bplustree" return(node)

}

The child node is of type doubly linked list to have a connection as shown in Figure 7.11 :

dlinkchildNode <- function(val, prevnode=NULL, node=NULL) {

llist <- new.env(parent=create\_emptyenv()) llist$prevnode <- prevnode llist$element <- val llist$nextnode <- node

class(llist) <- " dlinkchildNode" llist

}

[ 185 ]

Page 197

﻿

Indexing

The above function can be utilized to create a doubly linked list. In the case of a new object, a new environment is created as shown below while adding a value 1 to a new doubly linked list:

The B+ tree uses the copy-up and push-up approach for insertion. Let's take an example: Initially, B+ tree will have a single node during tree creation, and as the node overflows, the B+ tree will split the node into two. The new node is generated as the new root. The first key in the right node is copied up to the new root, as shown in Figure 7.17 :

Figure 7.17: Example of B+ tree creation

The preceding example can be used to set up the insertion process for a B+ tree, which follows a schema very similar to the 2-3 tree insertion, and is stated in the following steps:

Insert a key at the root if the node is empty

If the node is full, then split the node into two, distributing the keys evenly between the two nodes

For the leaf node, copy the minimum value in the second of these two nodes, and recursively repeat this to insert it into the parent node

For the internal node, exclude the middle value during the split, and repeat the insertion algorithm to insert this excluded value in the parent node

[ 186 ]

Page 198

﻿

Chapter 7

Similar to the B-tree, the B+ tree supports exact match queries, that is, it finds the value for the given key. Furthermore, B+ trees are also very efficient for range queries, that is, finding all the values within a defined range. For an exact match query, the B+ tree follows a single path from the root node to leaf node, as depicted in Figure 7.18 , which shows the search path for key 56 :

Figure 7.18: Exact query match example in B+ tree

B+ trees also very efficiently support range queries, that is, finding all the objects within a defined range. This is due to the fact that all the leaf nodes are sorted and linked together. If we want to search for all objects lying within a defined range of values, this can be achieved by performing an exact match for the lower-value key, and then following the sibling leaf using connection. An example of a range query is shown in Figure 7.19 :

Figure 7.19: Range query example in B+ tree

The preceding diagram shows an example of a range query for all objects between [56, 97]. The B+ tree first locates the lowest value within the tree, and then transverses through the nodes to find all the values till 97. The pseudo script in R for a range query is written as follows:

### Function for range queries

querry\_search<-function(node, key1, Key2){

## Function to get values within leaf node using link list search\_range<-function(child, key1, key2, val=NULL){

if(child$element>key1 & child$key2){

val<-c(val, child$element)

search\_range(child$nextnode, key1, key2, val)

[ 187 ]

Page 199

﻿

Indexing

} else {

return(val)

} }

if(key1>key2){

temp<-Key2 key2<-key1 key1<-temp

}

child<-search\_lower\_key(node, key1) # search lower leaf rangeVal<-search\_range(child, key1, key2) # Return Range return(rangeVal)

}

The search\_lower\_key function uses a structure similar to the search\_key function in 2-3 node. Once the leaf with the lower key is identified, the search\_range function transverses through the interconnected leaves to find all the values till the higher limit is hit. To delete an object from a B+ tree, the path from the root to the leaf is identified, and then the object is removed from the leaf. After removal, if the leaf is more than half-full, then nothing needs to be done. However, if, after removal, the leaf size decreases to less than half, then the algorithm redistributes values from the neighbors to underflowing nodes. In cases where redistribution is not possible, the underflowing node is merged with a neighbor. An example of deletion is shown in Figure 7.20 :

Figure 7.20: Deletion in B+ tree when leaf has suﬃcient number of objects after deletion

In Figure 7.19 , the deletion of key 68 has not unbalanced the tree. However, if another element 67 is deleted from the tree, the tree readjusts as shown in Figure 7.21 , and 79 is pushed up the node:

[ 188 ]

Page 200

﻿

Chapter 7

Figure 7.21: Deletion in B+ tree when leaf has an insuﬃcient number of objects after deletion

In cases where neighboring nodes are not able to satisfy, then the nodes are merged and readjusted accordingly.

The B+ tree also supports standard of aggregation queries due to its efficiency with range queries such as count, sum, min, max, average, and so on. One approach to performing aggregation queries using a B+ tree is to keep a temporary aggregate value; starting with an initial default value, the aggregator keeps updating with every value found in the tree. On completion of the search, the aggregator's final value is returned. However, this approach is not efficient as it will require a computational effort of O(log b n + t/b) , where t is number of keys and b is the tree average branching factor. The approach is also not suitable for large queries, as it requires a lot of disk pages to be accessed while aggregation. Another way to implement the B+ tree with reduced computation while aggregation is to store the aggregated values of the subtrees. Thus, when any query is executed, the local aggregates are used, and it avoids browsing the whole subtree.

B-tree analysis

B+ trees have received a lot of attention, and have been used widely in databases for indexing. Before we get into an analysis of the B+ tree, let's introduce some of its practical aspects. The minimum number of occupancy in B+ tree is 100. Thus, the fan-out parameter for a B+ tree will be between 100 and 200. From a practical perspective, an average page capacity in a B+ tree is around 66.7%.

[ 189 ]

Page 201

﻿

Indexing

Thus, a page with a fan-out parameter of 200 will contain 200\*0.667 = 133 elements. Thus, the relationship between the height and the number of objects that a typical B+ tree can hold can be evaluated as shown in Table 7.1 :

Table 7.1: Deletion in B+ tree when leaf has an insuﬃcient number of objects after deletion

The initial levels of a B+ tree have very few number of pages. For example, if a disk page is 4 KB large, then the first two levels will hold 4\*134=536 KB space on the disk, which is small enough to be stored in-memory, and need to be scanned to go down the leaf while searching.

In B-trees and B+ trees, operations are asymptotic with I/O cost of O(log b n) , where n is the total number of records in the tree and base b is the tree average branching factor. The operation time for insertion, deletion, and search is the same in both B-trees and B+ trees as both data structures follow the path from the root to the leaf for an operation. Let's assume that the time spent on each node is O(d) in the main memory. As B-tree ensures every node is at least half-full, the average branching factor will be d/2 where d is the order. Thus, operations in a B+ tree are asymptotic with O(log d/2 n) . The search, insert, and delete operations will take O(d log d/2 n) time. To further reduce the computation, especially while inserting a lot of records, the B+ tree uses bulk load approaches, which sorts in the input, and fills the leaf nodes in a sequential order in block of page size. If the keys are sorted, then the bulk load method reduces the insertion time by O(n/S) , where 2S is the number of keys stored in a leaf.

[ 190 ]

Page 202

﻿

Chapter 7

Exercises

1. Consider that a secondary-level index with 10,400 blocks needs to be represented

using multi-level linear indexing. The blocking factor is 52 entries per block, which is also the fan-out factor for multi-level indexing:

How many levels are required for indexing?

How many blocks are required at the second level?

What would be the minimum blocking factor to fit the dataset using a second-level linear index?

2. Write the code for deletion of an element from a 2-3 tree. Also, prove that the number of leaf nodes in a 2-3 tree with height h will be between 2 h-1 and 3 h-1 . 3. Assume a computer system with disk blocks of 8,192, and that you want to store records with 16-byte keys and 64-byte fields. What will be the greatest number of records that can be stored in a file if a linear index of size 4 MB is used?Assume the records are sorted and packaged sequentially into the disk file.

4. In the balanced binary tree given in the figure that follows, how many nodes will

become unbalanced when inserted as a child of the node g ?

Figure 7.22: Balanced binary tree

[ 191 ]

Page 203

﻿

Indexing

5. Illustrate what the insertion of 1, 3, 6, 2, 7, 9, 10 (in same order) into a B+ tree will

look like.

6. Suppose N is an interior node whose capacity is m keys and m+1 leaves. Let m be assigned the m+2 leaf because of leaf splitting below which leads to the creation of a new node m which will be siblings to immediately to its right of m .

7. Write a program to create and manage a B+ tree. The program should implement

the following operations: creation, insertion, deletion, and tree display.

Summary

The current chapter covered the fundamentals of indexing. The chapter built on the fundamentals of linear indexing, and extended it to ISAM. The linear indexing method is a good approach for static datasets which do not change over time; however, if any updates are required, they come at a very high computation cost. To address this issue, the ISAM indexing approach has been introduced, which tries to address the updating issue of databases. But it is still suitable for a few updates only.

The chapter also covered tree-based indexing structures which utilize the binary search tree-based structure to minimize search and updates. Multiple tree-based indexing approaches were also covered. The most primitive version is a 2-3 tree which uses the two- key and three-child strategy. The 2-3 tree indexing approach is a good starting point for tree-based indexing approaches, but retrieval of data from disk-based storage is slow. The approach is further generalized as a B-tree which ensures that the tree is balanced, and is a suitable indexing structure for disk storage as well. The B+ tree, an enhanced version of B-tree which stores data only in the leaf (B+ tree) is discussed in later part of the chapter. The B+ tree stores data only in leaves, and all leaves are interconnected, which allows multiple types of aggregation queries to be efficiently executed. The next chapter will introduce graph-based data structures, which are highly useful in understanding relationships between objects.

[ 192 ]