**File Organization Part 3 – Index**

|  |  |
| --- | --- |
| **Indexes**   * Most implementations separate index levels from the data level * Data level can be read sequentially * Direct access provided by traversing the index from the root to a pointer into the data level * The data level ordering options   + blocks ordered sequentially   + data clustered based on a particular key   + data not ordered by the index (although it might be ordered by another index) | **What happens when an index block is full and a new entry is inserted causing it to overlfow?**   * One approach (e.g., ISAM) is to write overflow to an overflow area * Another approach(e.g., B-Tree) is to split the block into two blocks and insert an entry into the block above |
| **Example 3-1: Sparse Index**  **Index has one entry for each block/track** | **Example 3-2: Dense Index**  **Index has an entry for each logical record** |
| **Data Level**  We already mentioned that the data level could be clustered based on an index. Now we will discuss how the data level blocks are represented.  Data blocks will contain a pointer to the next data block in sequential order.  Issues:   * Impact of Variable-sized columns (e.g., name, address):   + Storage space   + Data row (logical record) size   + Access to columns * Avoid changing an index pointer when row size changes * Uniqueness of each dense data level pointer (aka, RowID) | Suppose our data block contains student information.   * Student names could be as large as 80 bytes and as small as 8 bytes. These average 40 bytes. * Addresses could be as big as 300 bytes, but average 60 bytes. * Assume other attributes and overhead takes 32 bytes. * If the rows are of fixed length, each row is 80+300+32=412 bytes. With 20 bytes of overhead for a block (including a data block pointer to the next block), how many rows can we fit in a 4K data block?   + Max size used for data in a data block = 4096 – 20 = 4076   + Number of rows per data block is  4076 usable bytes per block  ---- = 9.9 -> 9 rows/block 412 bytes per row   With the fixed-length records, we can provide RowIDs which reference a combination of the data block and an offset to the row; however, we were able to only fit 9 rows per block.  How can we improve the rows per block by representing variable-sized columns differently? Since the row size of a particular row can change (e.g., name change, address change), we have to address the issues on the left. |
| **Access to Variable-Sized Columns**  We can place the variable-sized columns at the end of the row.  At the beginning of the row, include:   * row size in bytes * for each column after the first variable-sized column, include a 2 byte value representing the offset to the column.   Each variable-sized column will begin with a one or two byte size.  DBMSes have metadata tables that describe the contents of each table and index (e.g., DB2 has SYSTABLES, SYSCOLUMNS, SYSINDEXES, and SYSKEYS). | **Example** **3-3**: Using offset pointers and column sizes for variable-sized rows    For this row, we are now only taking 74 bytes plus additional row overhead (row size, row type).  However, now that rows can more easily shrink and grow when the data changes, the rows can move around in the block. Our RowId cannot be simply the combination of the block address and an offset to the row. |
| **Indexed Sequential Access method (ISAM)**  With ISAM, overflows of data blocks cause insertion into an overflow area.  ISAM keeps data in indexed order. | **Example 3-4**: ISAM with Overflow |
| **ISAM Overflow Insertion**  We attempt to insert 45 Callie in the first data block. Since it didn't fit, the highest logical record is placed in overflow.  We didn't yet have an overflow block for this data block so it allocated an overflow block for this purpose.  With a long overflow chain, it may take more retrievals to access a logical record.  Assuming none of the blocks are in memory, how many reads would it take to access:   * 90 Shane - 3 reads * 45 Callie – 3 reads * 240 Shelton – 4 reads * 250 Bridg - 5 reads * 50 Kris - 4 reads | **Example 3-5**: after inserting 45 Callie into the ISAM in example 3-5. |
| **B-Trees and B+Trees**  **Balanced Trees** (B-Trees, B+Trees) are the most popular indexing approach.  Each node in a Balanced tree is persisted as a index block.  When a **block overflows**, the block **splits** into two blocks and an entry is placed in the level above. If the root level splits, the number of levels in the Balanced Tree increases. Therefore, B-trees only grow at the top.  B-Trees contain keys, data, and pointers in each block. B+Trees separate the data into a data level. The index level contains keys and pointers.  **B-Tree**: A multi-level index of order ***n*** which has the following properties:   1. Every node contains at most 2***n*** keys. Corresponding to each key is data. Note that in a B+tree, only the leaf level nodes contain data. 2. Every node, except the root, contains at least ***n***keys. 3. Every node is either a leaf (has no descendants) or it has ***m+1*** descendants (assuming it has ***m*** keys). 4. All leaf nodes appear at the same level.   Assume that the keys are in ascending sequence. All pointers in the node with a subscript <= ***i*** point to nodes containing key values less than key***i.*** | **Example 3-6: B-tree of order N=2**  **B-Tree Node Structure:**    **The pointers will be Record Block Number (RBN) values for the b-tree nodes.**  **C structure:**  struct BtreeNode  {  int iNumEntries; // Number of entries in this node  Key keyM[4];  Data dataM[4];  long rbnM[5];  };  **Example B-Tree:**    Look at the node containing 10 and 20.   * In a B-Tree, that node would also contain the data for keys 10 and 20. * The pointer to the left of key value 10 will be to a node containing key values less than 10. * The pointer to the right of 10 and left of 20 will be to a node containing key values between 10 and 20. * The pointer to the right of 20 will be to a node containing key values greater than 20, but less than the ancestor key value 25.   The example has three levels of nodes. For n=2, what is the maximum number of entries at the bottom level?  Top level   * would have m = 2 \* n = 4 entries in 1 node * would have m+1 pointers to the level below; therefore 5 pointers   Middle Level would have   * 5 pointers from above so 5 nodes * So, 5 nodes \* m entries per node = m2 + m entries = 20 entries * So, 5 nodes \* (m+1) pointers per node = 25 pointers   Bottom level would have a maximum of how many entries?   * 25 nodes \* (m+1) |
| **Searching**  The searching algorithm on the right uses a general language. It returns the Relative Block Number (RBN) of the node containing the key or 0 if not found. If found, it also returns the data corresponding to the key through a parameter.  The search algorithm on the right is iterative. It is easier to implement the insertion and deletion using recursion. | The following uses a general language:  long Search (Btree btree, Key key, ByRef Data data):  Node node;  lRBN = btree.root;  outer: while lRBN <> 0:  rc = readNode(btree, lRBN, &node);  if rc <> 0:  errExit ...  for (i = 0; i < node.iNumEntries; i += 1):  rc = compare(key, node.keyM[i]);  select rc:  when EQUAL:  data = node.dataM[i];  return lRBN;  when LESS:  lRBN = node.rbnM[i];  continue outer; // continue at the top of the while  end select  end for;  // didn't find it <= any keyM, so it is higher  lRBN = node.rbnM[node.iNumEntries];  end while;  // didn't find it  return 0;  end Search; |
| **Insertion Algorithm - Fits in Leaf Node**  The insertion algorithm doesn't insert children below an existing leaf node. We will see that B-Trees grow levels above the root.  Step 1: In this algorithm, we should use recursion to recurse down to the leaf node that should contain a new entry *keyIns* and *dataIns*.  Step 2: If the leaf node contains less than 2n entries, insert the *keyIns/dataIns* entry in that leaf node. | Example 3-7: Insertion in a leaf node where it fits |
| **Insertion - Splitting**  Step 3: If the node already has ***2n*** keys (i.e., the node is full), the node must be **split** into two nodes (a new node is allocated). The ***2n+1*** keys (i.e., ***2n*** keys in the node and the one new key) must be equally distributed onto the existing node and a new node, and the middle entry needs to be moved up one level into the ancestor node.  We place the new node to the right of the existing node. | Example 3-8: Node is full, split it and insert the middle key/data in its ancestor. |
| **Insertion - Splitting causing ancestor to also split**  Step 4: The insertion in the ancestor may cause it to split.  In the worst case, splits can propagate all the way to the root.  If the ancestor is the root, the number of levels increases by 1. Instead of growing by creating children of leaves, Balance Trees grow by splitting the root. This is why they are always balanced. | Example 3-9: Split a node causing a split in the ancestor. |
| **Exercise 1-Part 1:**  Assume we start with an empty B-Tree.  a. Insert 20  b. Insert 40, 10, 30  c. Insert 15 |  |
| **Exercise 1-Part 2 continue from above:**  d. Insert 35, 7, 26, 18, 22  e. Insert 5 |  |
| **Exercise 1-Part 3 continue from above:**  f) 42, 13, 46, 27, 8, 32  g) 38, 24, 45, 25 |  |
| **Deletion**  The deletion algorithm is complicated by node **underflow** where a deletion can cause a non-leaf node to have less than **n** entries.  Let's assume our key to delete is called *delKey.*  **Step D1**: Use a recursive algorithm to find the node which contains *delKey*.  **Step D2**: If *delKey* is in a leaf node:   * Remove the entry (removing the key and data) and shift the entries down. * If the number of keys in that leaf node becomes less than **n** and the leaf isn't the root, property #2 is violated. This is underflow. (See **Balancing for Underflow**.) | **Example 3-10: Deleting in a leaf node which doesn't cause underflow** |
| **Deletion - non-leaf node**  **Step D3:** If *delKey* is in a non-leaf node:   * Replace the *delKey* entry with its successor (the next key in key sequence) which must be on a leaf node. * Remove the successor from the leaf. * If the number of keys in that leaf node becomes less than **n** and the leaf isn't the root, property #2 is violated. This is underflow. (See **Balancing for Underflow**.) | **Example 3-11: Deleting in a non-leaf node replacing it with its successor.** |
| **Deletion - Balancing for Underflow using a Borrow**  Assume leaf node, *leaf,* underflows (it has less than **n** keys).  **Step U1:** Select an immediately adjacent sibling node, *sib*, of node *leaf* having more than **n** keys. (For consistency, we will try the right sibling first unless there isn't one.)  **Step U2**: If there is a sibling, *sib*, which has more than **n keys,** evenly distribute between *sib* and *leaf.* The ancestor entry between *sib* and *leaf* must be placed in *leaf*, and an entry from *sib* must replace the ancestor entry. This is a **borrow**. | **Example 3-12: Borrowing to handle an underflow.** |
| **Deletion - Balancing for Underflow using a Merge**  **Step U3:**  If there isn't a sibling which has more than ***n*** keys, **merge** a sibling with *leaf*. One of the nodes is freed and the other will contain the entries from *leaf,* the entries from the sibling, and the ancestor entry between them. This is the inverse of an insertion split. Removing the ancestor entry from the ancestor node may cause the ancestor node to underflow. This could propagate up to the root. If the number of keys in the root is reduced to zero, it is deleted; thereby, causing a reduction in the height of the b-tree. | **Example 3-13: Merging to handle an underflow.** |
| **Another Example**  For this example, we will start with the following B-Tree:    a. Delete 7  b. Delete 12  c. Delete 32 | **Example 3-14: Several deletions**  a**.** No underflow    b. What happens? S underflows, borrows from T, 30 placed in S, 31 becomes ancestor key.    c. What happens? T under flows, merge S,T, and 31. |
| **Example 3-15 Continued**  d. delete 6, 13, 31  e. delete 9 | **Example 3-14 continued**  d. simple deletions    e. What happens? R underflow, cant borrow, merge R,S, and 10 causeing new root. |
| **Exercise 2-P1: deletion**  Show the resulting B-Tree after the specified deletions.    a. Delete 25, 45 | **Exercise 2-P1:**  a. |
| **Exercise 2-P2: deletion**  b. Delete 42 | **Exercise 2-P2:**  **b.** |
|  |  |

**©2020 Larry W. Clark,** UTSA CS students may make copies for their personal use