**Question No. 1**

Suppose that a disk has the following parameters:

Seek Time s=10ms

rotational delay rd = 5ms

block transfer time brt = 1ms

block size B = 2048 btes

Ignore any inter block gap size for this question.

Suppose further that the PRODUCT file has the following fields:

PID (6 bytes)

Name (64 bytes)

Manufacture ID (6 bytes)

Series (4 bytes)

records r= 100,000

Given this, write appropriate formula to calculate the values for the following.

a. Calculate

1 the record size R

R= size of PID + size of Name + size of Manufacture ID + size of Series

R= 6 bytes + 64 bytes + 6 bytes + 4 bytes

**R= 80 bytes**

2 The blocking factor,bfr

BFR =

BFR =

**BFR = 26**

3 the total number of disk blocks, b

B =

B =

**B = 3907**

b. Calculate the average number of block accesses to search for an arbitrary record in the file using a linear scan.

Linear search scans through entire file, the average number of blocks would be half of the number of disk blocks.

Average block accesses =

=

= 1953.5

c. Calculate the worst case time in seconds to access an arbitrary record in the file using linear search.

The worst-time case would be seek time and rotational delay along with block transfer time for each block access. Since accessing half of the disk blocks:

Worst-case time = () \* (seek time + rotational delay + block transfer time)

Worst-case time = 1953.5 \* (10 ms + 5 ms + 1 ms)

Worst-case time = 1953.5 \* 16 ms

**Worst-case time = 31256 ms**

d. Suppose records are ordered by some key fields. Calculate the average number of blocks accessed and the average time needed to search for an arbitrary record in the field using binary search.

On average, binary search would require. Each comparison involves accessing a block. The average time for binary search can be calculated using same formula as for linear search, but with the number of block accesses replaced by the logarithm of the number of blocks.

Average block access for binary search =

Average block accesses for binary search = \* (seek time + rotational delay + block transfer time)

Question No. 2

**Assumptions for the Algorithms:**

* The hash function that is used gives a uniform distribution of keys in the hash buckets.
* Each bucket maintains a linked list to handle overflow condition using chaining.
* The hash file has a fixed size and cannot be resized dynamically.

Insertion:

|  |
| --- |
| 1. Compute address of the bucket using the hash function. 2. Access buckets in the hash function. 3. If the bucket is empty. 4. Create a new linked list node and insert records on its head. 5. If bucket is full, traverse in the linked list. 6. If same key record exists, update the data or perform handling remedies. 7. If it is not found, create a new linked list and insert record at the end of linked list. |

Deletion:

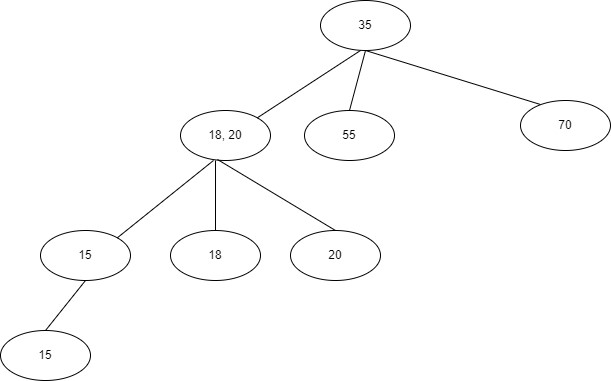
|  |
| --- |
| 1. Compute address of the bucket using the hash function. 2. Access buckets in the hash function. 3. In the buckets, traverse the linked list. 4. If targeted key record is found, remove that from the linked list. 5. After deletion if linked list becomes empty update the bucket. 6. If the record with targeted key is not found, perform remedies of static hashing. |

Update:

|  |
| --- |
| 1. Compute address of the bucket using the hash function. 2. Access buckets in the hash function. 3. In the buckets, traverse the linked list. 4. If targeted key record is found, update data with new values. 5. After deletion if linked list becomes empty update the bucket. 6. If the record with targeted key is not found, perform remedies of static hashing. |

**Question No. 3**

**B+- tree of order = 3 after inserting PID values: PID values: 70, 15, 20, 35, 18, 55, and 43:**



**Explanation:**

1. The Tree is empty initially.
2. First of all, 35 is inserted and it becomes the root.
3. After that 18, 20, and 15 are inserted in the tree. 35 has only one child because the order is 3, it can take 18 and 20 as its children and 15 is inserted as the leftmost child of 18.
4. 55 is inserted in the tree next. Since the root has space for one more child, 55 is added as right child of 35.
5. 70 is inserted in the tree finally. The root became full, it is split into two nodes, and 70 becomes the new root of the tree.

**µQuestion No. 4**

Q1:

Join EMPLOYE and DEPARTMENT on Dno and Dnumber.

Select Fname, Lname and Adress

where Dname = ‘Research’

Potential Query Trees:

1. Perform a Join between EMPLOYE and DEPARTMENT on Dno and Dnumber and then show rows where Dname = ’Research’ .
2. Show rows from DEPARTMENT where Dname = ‘Research’ and perform a Join with EMPLOYEE on Dno and Dnumber.

Q8:

Self-join EMPLOYEE table to match Super\_ssn with Ssn.

Select Fname and Lname.

Potential Query Tress:

1. Perform a self-join on EMPLOEE table on Super\_Ssn and select Fname and Lname.
2. Select Fname and Lname from EMPLOYEE and then self join the result on Super\_Ssn and Ssn.

**Q1B:**

* Similar to Q1 with aliasing tables.
* Potential queries similar to Q1 with aliasing.

Q27:

Join PROJECT, WORKS\_ON and EMPLOYEE tables on Pnumber, Pro, and Essn.

Group by Pnumber and Pname and count the rows.

1. Join PROJECT, WORKS\_ON and EMPLOYEE tables together and then perform GROUP By on Pnumber and Pname, followed by counting rows.

b.

It would be necessary to take into consideration the particular execution technique and optimization tactics that were utilized in order to draw optimized and original query trees for each query executed. In light of the fact that I am unable to directly generate SQL query execution plans, I will instead present a generalized optimization procedure:

Initial Query Tree:

Creation of the first query tree should be done by making use of the logical order of operations that is provided by the SQL query.

Optimization Steps:

In order to improve the query tree, you should implement modifications such as predicate pushdown, join reordering, index utilization, and other similar modifications.

You can achieve maximum efficiency while simultaneously processing a less amount of data if you take into consideration elements such as access paths, join order, and selection order.

Final Query Tree:

The query tree that has been optimized by following the steps that have been given should be displayed.

c. Comparing queries (part a) and Initial Query trees (part b) requires analyzing how well the initial query trees shows the logical operations that are specifies by SQL queries and how final tress enhance performance through optimization techniques. Comparing them evaluates the effectiveness of the selected query trees optimization strategies in executing the SQL queries effectively.

Bonus Questions:

16.47: Extensible Hashing:

Data management is dynamic approach with extensible hashing. This hashing algorithm involves flexibility as a pivotal factor. This approach is flexible to adjust the hashing function based on the situation and data type.

Algorithm:

Directories and buckets are key to this strategy. Directories store pointers to buckets with hashed data. Each directory has a Unique Id.  
The algorithm works as follows:

|  |
| --- |
| Initialize bucket depths and global depths of directories. Convert data to binary.  Consider data's "global depth" least significant bits (LSBs).  Map data according to directory ID.  Look for these circumstances if a overflows:  Global depth == bucket depth: The bucket should be divided in half, and both the global depth and the depth of the buckets should be increased. It is necessary to re-hash the components that were found in the division bucket.  Global > bucket depth: Only the depth of the bucket should be increased once the bucket has been divided in half. It is necessary to re-hash the components that were found in the division bucket.  Repeat all these steps for each element. |

Example:

1: Data = {28, 4, 19,1, 22, 16, 12, 0, 5, 7}

Bucket limit = 3

2: Convert data into binary.

Data = {28, 4, 19,1, 22, 16, 12, 0, 5, 7}

Binary = {11100, 00100, 10011, 00001, 10110, 10000, 01100, 00000, 00101, 00111}

3: Consider LSBs according to global depth. It is 1 in this case.

LSB(1) = 0: Map to Bucket 0

LSB(0) = 0: Map to Bucket 0

LSB(3) = 1: Map to Bucket 1

LSB(1) = 1: Map to Bucket 1

LSB(6) = 0: Map to Bucket 0

LSB(0) = 0: Map to Bucket 0

LSB(4) = 0: Map to Bucket 0

LSB(5) = 1: Map to Bucket 1

LSB(7) = 1: Map to Bucket 1

4: Check Overflow

Bucket0: {28, 4, 2, 16, 12, 0, 5}

Bucket1: {19, 1, 7}

Global depth = 1

Bucket depth = 1

No Overflow condition exists.

5: Only the depth of the bucket should be increased once the bucket has been divided in half. It is necessary to re-hash the components that were found in the division bucket.

6: Directory: [Bucket0, Bucket1, Bucket2]

Bucket0: {28, 4, 22, 16, 12, 0, 5}

Bucket1: {19, 1, 7}

Bucket0: {} (Empty initially).

**Question 17.26**

Consider redistribution between leaf nodes is feasible in order to alter the B+-tree insertion method to take redistribution into account before generating a new level. Rather of dividing the leaf node, redistribute if that is feasible. A high-level overview of the updated algorithm is shown as:

Check for Overflow:   
If a leaf node gets full after adding a key, consider redistribution among nearby leaf nodes before splitting.   
When possible, redistribute the leaf node instead of separating it.   
If redistribution fails, separate the leaf node.   
Redistribution:   
In left redistribution, keys can be moved from the right to the left neighbor to balance the burden. The left neighbour must have enough room.   
The right redistribution mechanism moves keys from the left neighbour to the right neighbour to equal out the distribution, if the right neighbour has enough space.   
Splitting:   
Split the leaf node as usual and update the parent node if redistribution is not possible.   
Update Parent Nodes:   
Update the parent nodes and propagate the updates to the root when a leaf node splits.

Question 18.17

Algorithm is needed to modify to add non-matching rows from the left table in the result to extend the sort-merge join algorithm for the implementation of LEFT OUTER JOIN operation on data.

|  |
| --- |
| 1. Sort the input tables by both the left and right tables on the join attribute. 2. Execute a merge operation same as the standard sort-merge join, and compare join attribute values from the left and right relations. 3. For matching attributes, output the joined row. 4. For non-matching attributes, output the left table row with NULL values for the right table columns. 5. If there are remaining rows in the left table after the merge output these rows with NULL values for the right table columns. |