**ADC Assignment 6**

Aditya Shekhar Camarushy, Bhavik Kollipara, Sai Prajwal Reddy, Srikar Devulapalli

**Q2)**

Table

Description automatically generated

Here we can see that as per the probability distribution defined initially in the Probability mass function (pmfSkill)

**Q3) (a)**

We can notice that as the size increases the amount of time taken for sorting and scanning

also increases.

We can also see that as the memory size is increased the amount of time taken for sorting

decreases. The reason that the time taken is high when the memory is less is because postgresql uses merge sort when the data does not fit in the memory. However, when the data fits in the memory postgresql uses quick sort and hence less time taken.

**(b)**

We can see that the average time that is required to sort is quite more than the average

time that is required for sorting.

We can also see that as we are increasing the memory, the amount of time that is required for sorting is becoming less as we increase the memory until a certain point after which the

time remains almost same.

However, the time required for scanning is almost the same even after increasing the memory.

In the case of 32mb and 256mb work memory, the time taken is almost same as the data fits in the memory in both the cases and hence quick sort is performed in both the cases and hence the almost same time.

**(c)** We can see from the tables that the amount of time taken to create indexes is higher than

the amount of time taken to perform sorting. We can also see that as the sizes increases, the

time taken is also higher for both creating and sorting.

However, when we compare the times for sorting using indexes to the times for sorting without indexes, we can see that the time taken for sorting using indexes is very less. This is

because sorting by index using B tree is faster compared to merge sort or quick sort

that was used by postgresql before indexing.

**Q4) (c)** We can see that initially for records below 100000, the amount of time taken to remove

duplicates using distinct and group by are almost the same. However, as the number of records

increase above that, we can see that the amount of time taken is less when distinct is used

compared to group by.

When we use explain analyze on both of them, we can see that the query plan that is used by

distinct and group by is essentially the same. We can however see that the time taken for

execution in this case is basically less in the case of group by than distinct. However, this

difference is very less.

Distinct is basically Group by that is performed without aggregation to remove duplicates and

hence in most cases, the time taken is approximately the same.

**Q7)**

1. Given Parameters –

block size = 8192 bytes

block-address size = 10 bytes

block access time (I/O operation) = 15 ms (micro seconds)

record size = 200 bytes

record primary key size = 8 bytes

N =

Let us calculate the number of keys in a node (n) –

In the best case scenario, the key would be the first element of the leaf node (hence the +1).

**Minimum time** = = **71.45 ms**

1. For maximum insertion time we assume that the height of the tree is maximum, this means that the branching factor is minimum which is 2.

At the root : 2

At non-leaf nodes : n/2 = ⌊454/2⌋ = 227

* **Maximum time** =

= \* 15 = 5\*15 = **75ms**

1. To hold the first two levels of B tree the main memory

We assume that the first 2 levels consists of the root node and the first level blocks

= 1 + 454 = **455 blocks**

The memory required to hold **first 2 levels** = 455 \* 8192 = **3,727,360 bytes**

Similarly, To hold the first 3 levels = 1 + 454 + 454 \* 454 = **206,571 blocks**

The memory required to hold **first 3 levels** = 206571 \* 8192 = **1,692,229,632 bytes**

**Q8)**

**(a)** Consider the following B+-tree of order 2 that holds records with keys 2, 8, and 11.

8

8 11

2

**Insert 5**

8

8 11

2 5

**Insert 7**

7 8

8 11

2 5

7

**Insert 10**

8

11

11

7

2 5

7

8 10

**Insert 4**

2 4

7

5

5 7

11

8

8 10

11

**Insert 10** ( No change in the B + - tree as 10 already exists record goes to datafile )

5 7

2 4

8 10

7

8

11

11

5

**Insert 12**

5 7

2 4

8 10

7

8

11

11

5

**Insert 1**

7

1 2

8 10

7

11

11 12

5

4

4

5 8

**(b)**

**The B + tree from part (a)**

1 2

8 10

7

11

11 12

5

5 8

4

7

4

**Delete 12**

4

7

4

5 8

5

11

11

7

8 10

1 2

**Delete 2**

1

8 10

7

11

11

5

5 8

4

7

4

**Delete 4**

1

8 10

11

11

8

5 7

7

5

**(c)**

**The B+ tree from part (b)**

1

8 10

7

11

11

5

8

5 7

**Delete 10**

**NOTE :** We delete 10 here, however it gets replaced by the duplicate 10 insert from part (a) and therefore 10 continues to remain in the index

5 7

8

5

11

11

7

8 10

1

**Delete 7**

1

8 10

11

11

5

8

5

**Delete 1**

5

8

5

11

11

8 10

**Delete 5**

11

11

8 10

**Q9)**

**(a)**

i = 0

i = 1

1

0

1. Insert 9 (1001) and 4 (0100)

1001

i = 1

0100

i = 1

i = 1

1

0

1. Insert 1 (0001) and 2 (0010)

i = 2

i = 2

i = 1

1001

i = 2

0100

0010

0001

10

01

11

00

1. Insert 8 (1000) and 3 (0011)

|  |
| --- |
| 000 |
| 001 |
| 010 |
| 011 |
| 100 |
| 101 |
| 110 |
| 111 |

0100

0011

0010

0001

1000

1001

i = 1

i = 3

i = 3

i = 2

i = 3

1. Insert 6 (0110) and 7 (0111)

|  |
| --- |
| 000 |
| 001 |
| 010 |
| 011 |
| 100 |
| 101 |
| 110 |
| 111 |

i = 1

1001

1000

0110

0111

i = 3

i = 3

0100

i = 3

0010

0011

0001

i = 3

i = 3

i = 2

i = 2

0111

0110

1000

1001

i = 1

01

i = 2

0010

0011

**(b)**

1. Delete records with keys 4 (0100) and 1 (0001)

00

10

11

01

1. Delete Records with keys 9 (1001) and 6 (0110)

i = 2

01

i = 2

00

0111

i = 2

1000

i = 1

0011

0010

10

11

1. Delete Records with keys 7 (0111) and 3 (0110)

i = 1

01

i = 1

00

0010

i = 1

1000

**Q 10)**

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 10000 | 0.651 | 0.223 |
| 100000 | 5.450 | 2.301 |
| 1000000 | 50.126 | 24.527 |

**Q 11)**

1. Small Range Execution:

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 100000 | 23.297 | 1.409 |
| 1000000 | 193.031 | 13.125 |
| 10000000 | 1141.897 | 198.567 |

1. Intermediate Range Execution:

We can see a small discrepancy in the following where in when the number of total records is 1000000, the average time taken without index is less than the average time taken with index.

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 100000 | 32.706 | 24.716 |
| 1000000 | 263.306 | 266.025 |
| 10000000 | 2361.667 | 2323.625 |

1. Maximum Range Execution:

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 100000 | 27.060 | 9.210 |
| 1000000 | 246.500 | 93.820 |
| 10000000 | 1757.513 | 1207.124 |

**Q 12)**

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 100000 | 4.011 | 0.033 |
| 1000000 | 72.061 | 0.033 |
| 10000000 | 214.805 | 0.059 |

**Q 13)**

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 10000 | 2.108 | 1.799 |
| 100000 | 21.341 | 17.976 |
| 1000000 | 238.002 | 192.713 |

**Q 14)**

We can see a small discrepancy in the following where in when the number of total records is

10000, the average time taken without index is less than the average time taken with index and also when the number of records is 1000, the time is almost the same in both cases.

|  |  |  |
| --- | --- | --- |
| Number of Total Records | Avg Exec. Time without Index: | Avg. Exec. Time with Index: |
| 10 | 0.337 | 0.053 |
| 100 | 0.500 | 0.159 |
| 1000 | 1.937 | 1.951 |
| 10000 | 19.058 | 23.449 |