Assignment 4 solutions

Q1

Consider a disk with average seek time of 10 ms, average rotational latency of 5 ms, and a transfer time of 1 ms for a 4KB block. The cost of reading/writing a block is the sum of these values (i.e. 16 ms). We are asked to sort a large relation consisting of 10,000,000 blocks of 4KB each. For this, we use a computer on which the main memory available for buffering is 320 blocks (*a bit small memory*). We begin as usual by creating sorted runs of 320 blocks each in phase 1. Then, we do 319-way merges. Determine the number of phases needed, and evaluate the cost of the Multi Phase Multiway Merge Sort.

Solution

We start by creating sorted sublists. We fill in the main memory (MM) with 320 blocks, sort them in MM and write the sorted sublist to disk (phase 1).

The number of sorted sublists at the end of phase 1 is: $\frac{10,000,000}{320} = 31,250$

Next we need to do merge. However, the number of sorted sublists is too big to allow a merge with only one pass. Since we need an output buffer, we can only merge 319 sorted sublists at a time. Therefore, we end up again with sorted sublists, but fewer this time.

The number of sorted sublists at the end of phase 2 is: $\frac{31,250}{319} = 98$

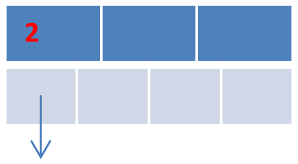
These sublists need to be merged in turn in another phase, phase 3.

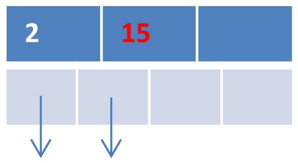
In each phase we read 10,000,000 blocks and write 10,000,000 blocks. So, the total number of I/Os is 3*2*10,000,000 = $6*10^7$. In terms of time, we multiply this number by 16 ms.

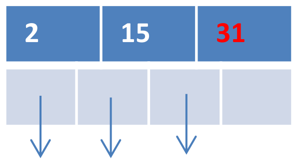
Note. We needed three phases because the available main memory is small, only 320*4K = 1.3 MB. However, it illustrates the point that sometimes we might need more than two phases (when memory is small and the file big).

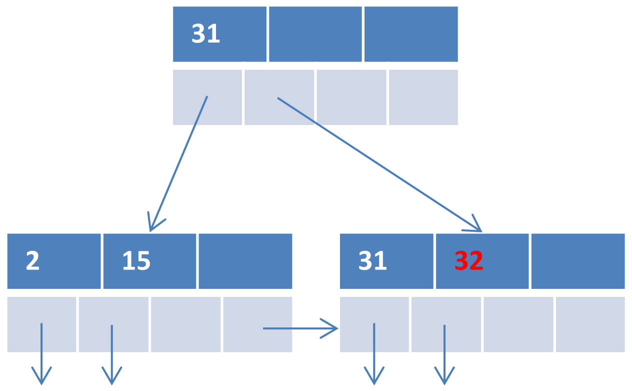
Q2

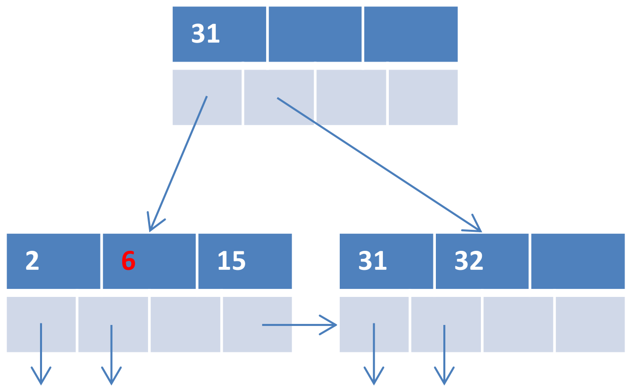
Build a B+ tree index with n=3 using the following sequence of keys:
2, 15, 31, 32, 6, 18, 19, 20, 3, 4, 5, 40, 41, 42
Redraw the tree each time an insertion is done.



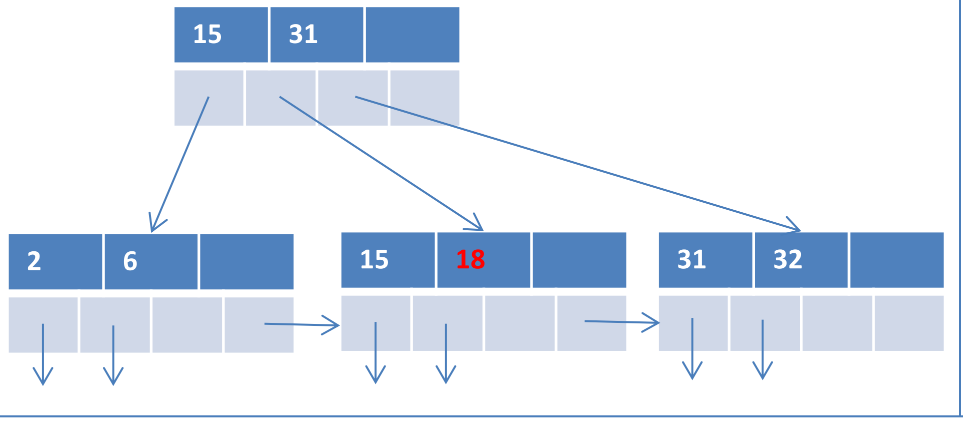


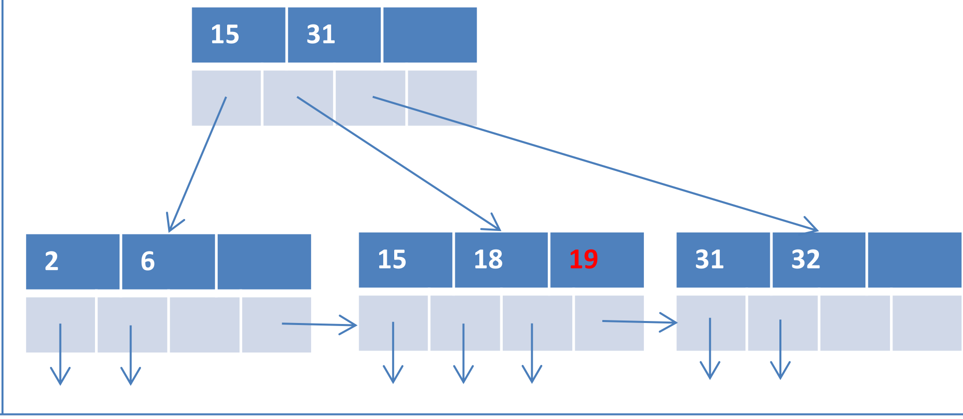


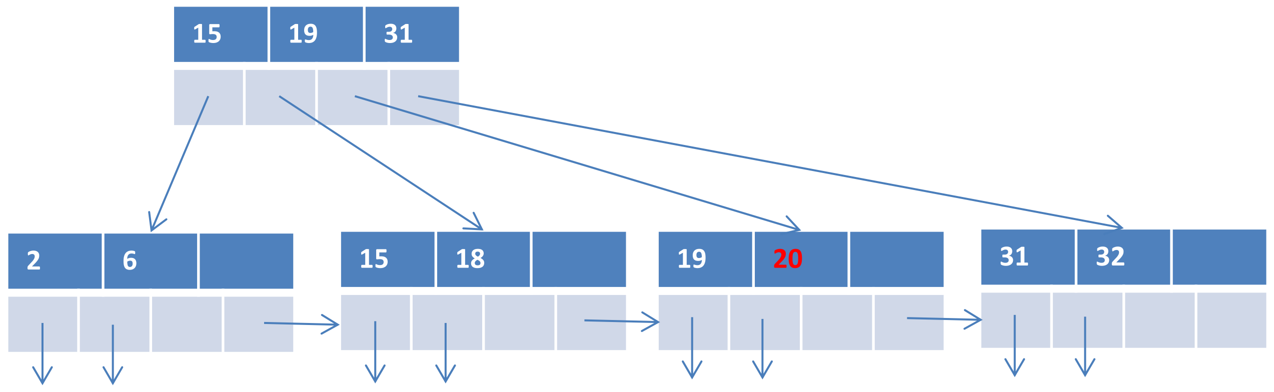


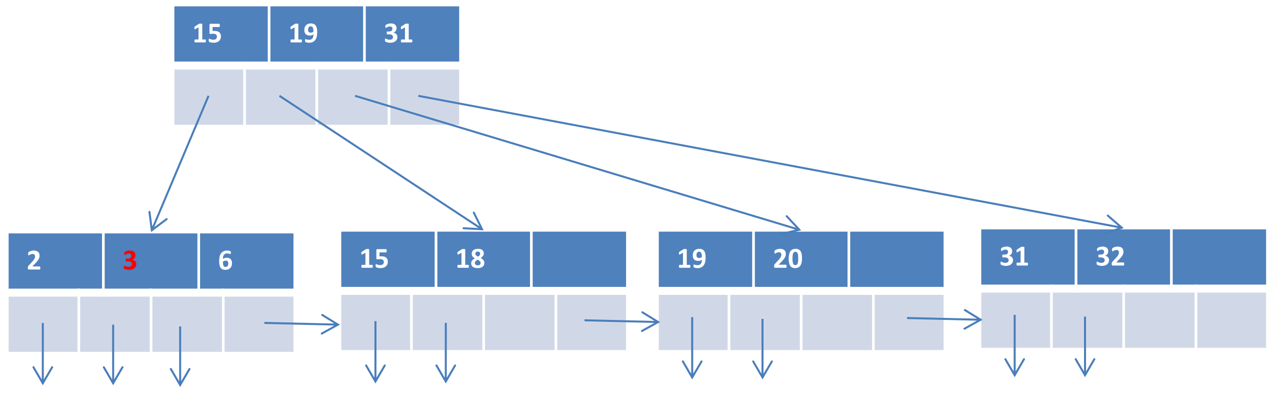


2, 15, 31, 32, 6, 18, 19, 20, 3, 4, 5, 40, 41, 42

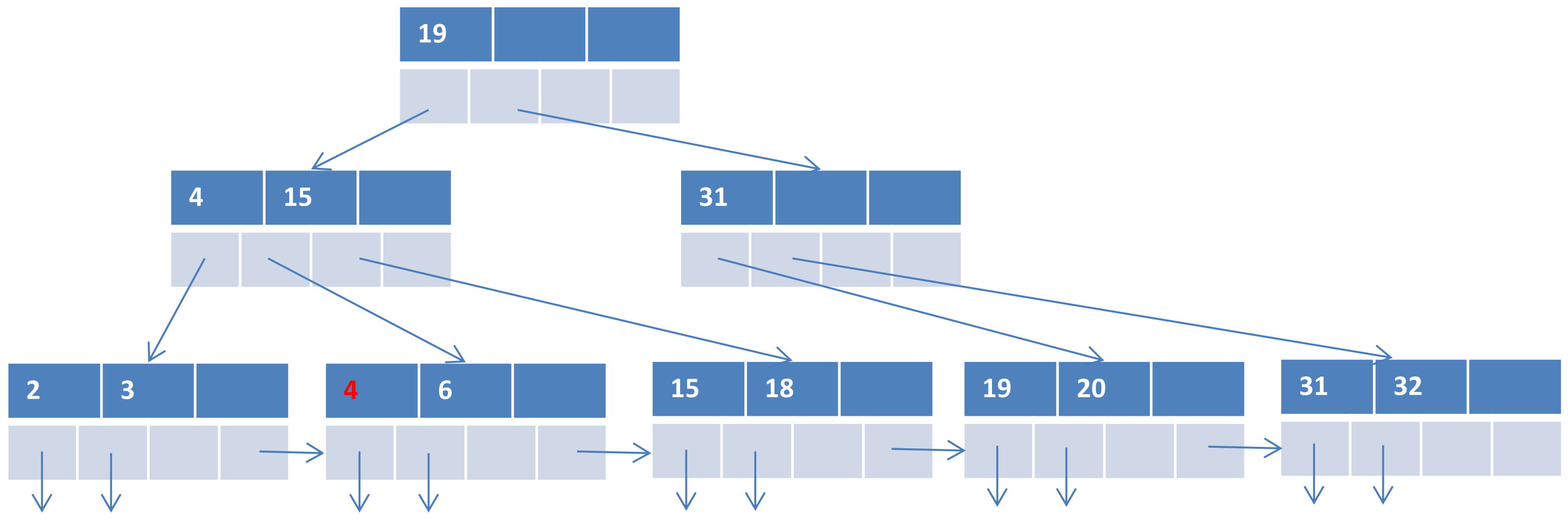


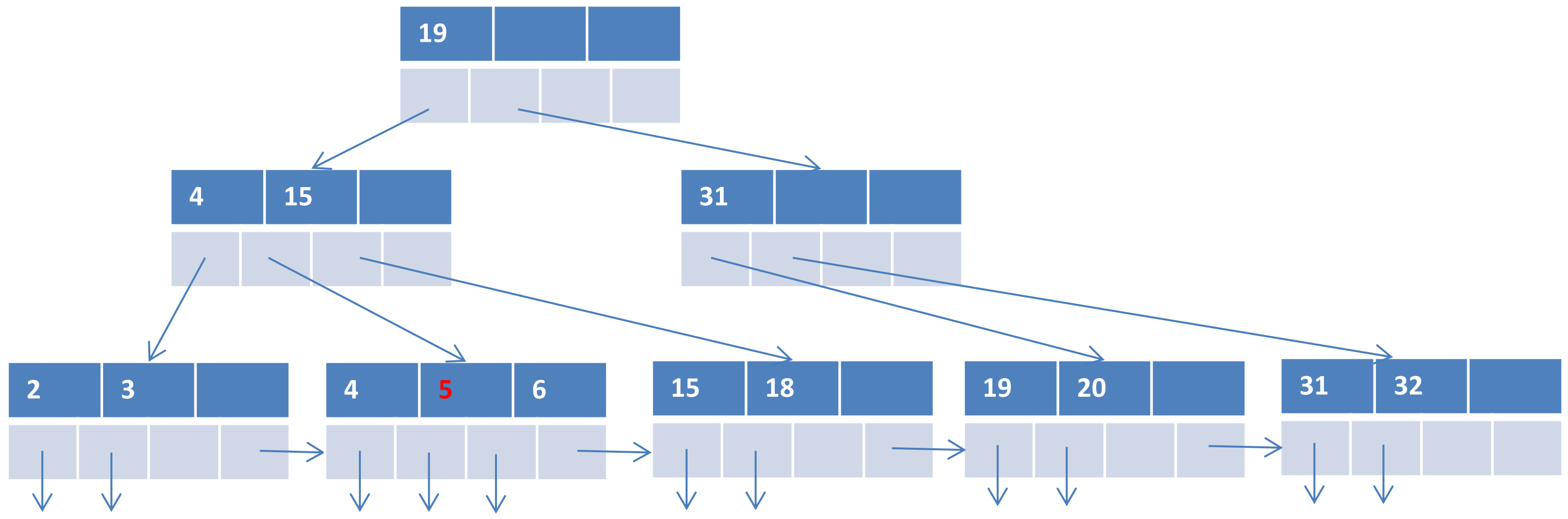




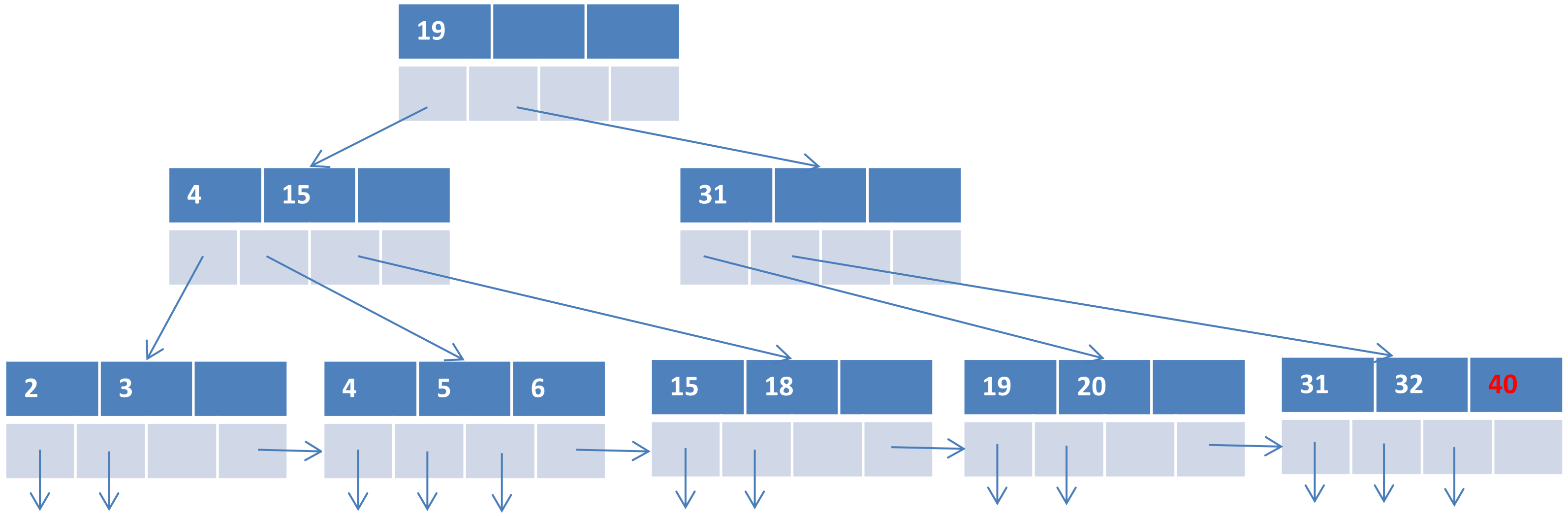


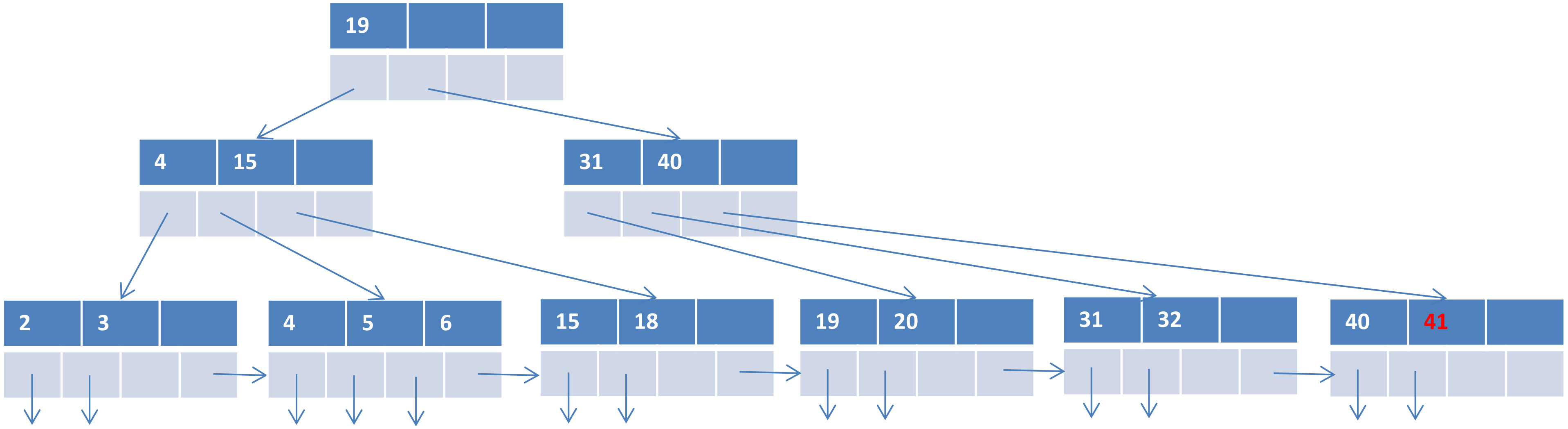
2, 15, 31, 32, 6, 18, 19, 20, 3, 4, 5, 40, 41, 42



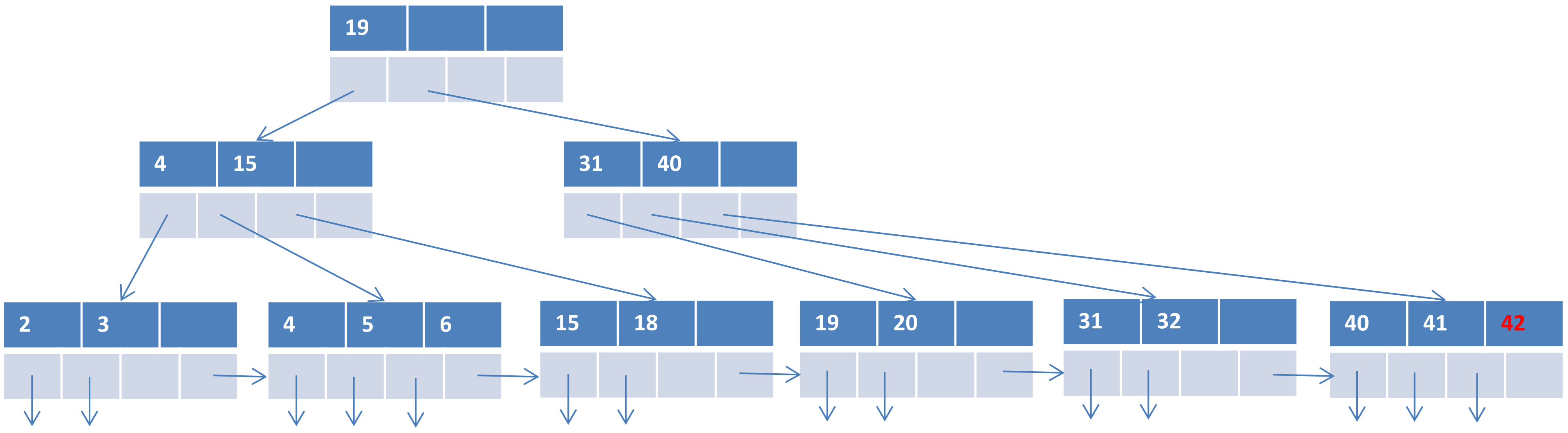


2, 15, 31, 32, 6, 18, 19, 20, 3, 4, 5, 40, 41, 42





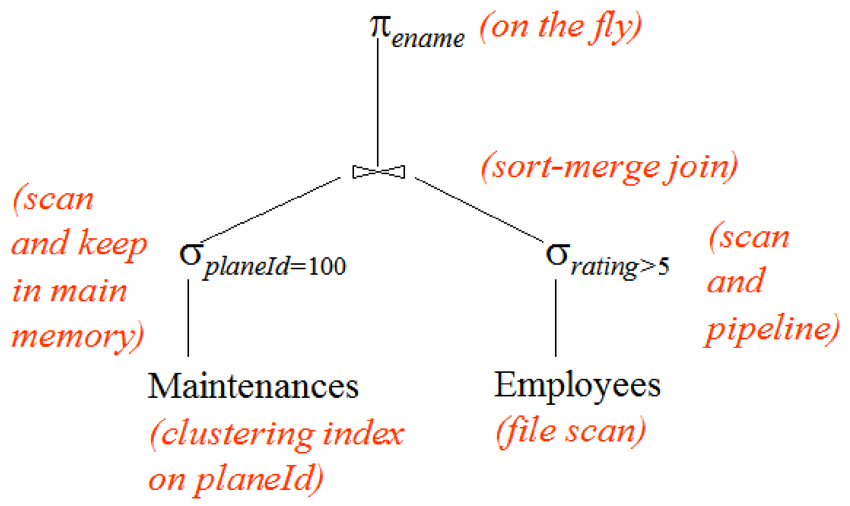
2, 15, 31, 32, 6, 18, 19, 20, 3, 4, 5, 40, 41, 42



Q3

Consider the following query plan.

What is the cost in term of number of I/Os for this plan?



Notes. The result of the left selection, being small, is kept in main memory, where it is sorted. The result of the right selection is pipelined to the join operator, i.e. the generation of the sorted sublists for the first phase of sort is done on the fly. Do not count the I/Os for writing the final results (after projection). Consult queryeval.pdf for the table statistics.

We need 12 I/Os for computing the selection *planeId=100* on Maintenances. The result of the selection is 10 blocks, which we assume fits in main memory (where it is also sorted).

We need to scan Employees in order to compute the selection *rating>5*. This costs 50 I/Os. The result, 25 blocks, is pipelined to the join. Thus the first phase of sorting, the generation of sorted sublists, will only need 25 I/Os (not 2*25).

The second phase of sorting will need 2*25 = 50 I/Os.

Then we do the join. It will take 25 I/Os.

The last operation (projection) is performed on the fly.

In total we have: 12 + 50 + 3*25 + 25 = 162 I/Os.

Q4

For each of the schedules of transactions T1, T2, and T3 below:

1. r1(A); r2(B); r3(C); r1(B); r2(C); r3(D); w1(A); w2(B); w3(C);
2. r1(A); r2(B); r3(C); r1(B); r2(C); r3(A); w1(A); w2(B); w3(C);

do each of the following:

i. Insert shared and exclusive locks, and insert unlock actions. Place a shared lock immediately in front of each read action that is not followed by a write action of the same element by the same transaction. Place an exclusive lock in front of every other read or write action. Place the necessary unlocks at the end of every transaction.

Tell what happens when each schedule is run by a scheduler that supports shared and exclusive locks.

ii) Insert shared and exclusive locks in a way that allows upgrading. Place a shared lock in front of every read, an exclusive lock in front of every write, and place the necessary unlocks at the ends of the transactions.

Tell what happens when each schedule is run by a scheduler that supports shared locks, exclusive locks, and upgrading.

iii) Insert shared, exclusive, and update locks, along with unlock actions. Place a shared lock in front of every read action that is not going to be upgraded, place an update lock in front of every read action that will be upgraded, and place an exclusive lock in front of every write action. Place unlocks at the ends of transactions, as usual.

Tell what happens when each schedule is run by a scheduler that supports shared, exclusive, and update locks.

Q4, i.1

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$xl_1(A); r_1(A)$		
	$xl_2(B); r_2(B)$	
		$xl_3(C); r_3(C)$
$sl_1(B);$ denied		
	$sl_2(C);$ denied	
		$sl_3(D); r_3(D)$
		$w_3(C); u_3(C); u_3(D);$
	$sl_2(C); r_2(C)$	
	$w_2(B); u_2(B); u_2(C);$	
$sl_1(B); r_1(B)$		
$w_1(A); u_1(A); u_1(B)$		

Q4, i.2

$r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

T_1	T_2	T_3
$xl_1(A)$; $r_1(A)$		
	$xl_2(B)$; $r_2(B)$	
		$xl_3(C)$; $r_3(C)$
$sl_1(B)$; denied	$sl_2(C)$; denied	
		$sl_3(A)$; denied

Deadlock!

Q4, ii.1

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$sl_1(A); r_1(A)$		
	$sl_2(B); r_2(B)$	
		$sl_3(C); r_3(C)$
$sl_1(B); r_1(B)$		
	$sl_2(C); r_2(C)$	
		$sl_3(D); r_3(D)$
$xl_1(A); w_1(A); u_1(A); u_1(B)$		
	$xl_2(B); w_2(B); u_2(B); u_2(C);$	
		$xl_3(C); w_3(C); u_3(C); u_3(D);$

Q4, ii.2

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(A); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$sl_1(A); r_1(A)$		
	$sl_2(B); r_2(B)$	
		$sl_3(C); r_3(C)$
$sl_1(B); r_1(B)$		
	$sl_2(C); r_2(C)$	
		$sl_3(A); r_3(A)$
$xl_1(A)$; denied		
	$xl_2(B)$; denied	
		$xl_3(C)$; denied

Deadlock!

Q4, iii.1

$r_1(A); r_2(B); r_3(C); r_1(B); r_2(C); r_3(D); w_1(A); w_2(B); w_3(C);$

T_1	T_2	T_3
$ul_1(A); r_1(A)$		
	$ul_2(B); r_2(B)$	
		$ul_3(C); r_3(C)$
$sl_1(B);$ denied		
	$sl_2(C);$ denied	
		$sl_3(D); r_3(D)$
		$xl_3(C); w_3(C); u_3(C); u_3(D);$
	$sl_2(C); r_2(C)$	
	$xl_2(B); w_2(B); u_2(B); u_2(C);$	
$sl_1(B); r_1(B)$		
$xl_1(A); w_1(A); u_1(A); u_1(B)$		

Q4, iii.2

$r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

T_1	T_2	T_3
$ul_1(A)$; $r_1(A)$		
	$ul_2(B)$; $r_2(B)$	
		$ul_3(C)$; $r_3(C)$
$sl_1(B)$; denied		
	$sl_2(C)$; denied	
		$sl_3(A)$; denied

Deadlock!

Q5 (Undo Log), Q6 (Redo Log)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

Suppose that we begin a nonquiscent checkpoint immediately after one of the following log records has written (in memory):

a) <S,A,60>
b) <T,A,10>
c) <U,B,20>
d) <U,D,40>
e) <T,E,50>

For each, tell:

When the <END CKPT> record is written, and

For each possible point at which a crash could occur, how far back in the log we must look to find all the possible incomplete transactions.

Q5.a (<S,A,60>)

<START S>
<S,A,60>
<START CKPT (S)>
<COMMIT S>
<END CKPT>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

For a crash
after <END CKPT>:
search back to
<START CKPT(S)>

For a crash
prior to <COMMIT S>:
search back as far as
<START S>

Q5.b (<T,A,10>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START CKPT (T)>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<END CKPT>
<V,B,80>
<COMMIT V>

For a crash
after <END CKPT>:
search back to
<START CKPT(T)>

For a crash
prior to <COMMIT T>:
search back as far as
<START T>

Q5.c (<U,B,20>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<START CKPT (T,U)>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<END CKPT>
<V,B,80>
<COMMIT V>

For a crash
after <END CKPT>:
search back to
<START CKPT(T,U)>

For a crash
prior to <COMMIT T>:
search back as far as
<START T>

Q5.d (<U,D,40>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<START CKPT (T,U,V)>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>
<END CKPT>

For a crash
after <END CKPT>:
search back to
<START CKPT(T,U,V)>

For a crash
prior to <COMMIT T>:
search back as far as
<START T>

For a crash
prior to <COMMIT V>
but after <COMMIT T> :
search back as far as
<START V>

Q5.e (<T,E,50>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<START CKPT (T,V)>
<COMMIT T>
<V,B,80>
<COMMIT V>
<END CKPT>

For a crash
after <END CKPT>:
search back to
<START CKPT(T,V)>

For a crash
prior to <COMMIT T>:
search back as far as
<START T>

For a crash
prior to <COMMIT V> but
after <COMMIT T> :
search back as far as
<START V>

Q6.a (<S,A,60>)

<START S>
<S,A,60>
<START CKPT (S)>
<END CKPT>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

<END CKPT> occurs right away because there aren't any dirty buffers yet (buffers modified by committed transactions)

If the crash occurs after <END CKPT>, but before <COMMIT S>:
search back as far as <START CKPT(S)>
If the crash occurs after <COMMIT S>:
search back as far as <START S>

Q6.b (<T,A,10>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START CKPT (T)>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

<END CKPT> can occur at any point after <START CKPT(S)>

If the crash occurs after <END CKPT>:
we restrict ourselves only to committed transactions that were listed in <START CKPT(..)>, i.e. T, and those that started after this point.
If the crash occurs in between, then, for this example, we have to consider all the log.

Q6.c (<U,B,20>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<START CKPT (T,U)>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

Similar explanation to Q6.b

Q6.d (<U,D,40>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<START CKPT (T,U,V)>
<V,F,70>
<COMMIT U>
<T,E,50>
<COMMIT T>
<V,B,80>
<COMMIT V>

Similar explanation to Q6.b

Q6.e (<T,E,50>)

<START S>
<S,A,60>
<COMMIT S>
<START T>
<T,A,10>
<START U>
<U,B,20>
<T,C,30>
<START V>
<U,D,40>
<V,F,70>
<COMMIT U>
<T,E,50>
<START CKPT (T,V)>
<COMMIT T>
<V,B,80>
<COMMIT V>

Similar explanation to Q6.b

Q7

For each of the following relation schemas and sets of FD's:
R(A,B,C,D) with FD's $AB \rightarrow C$, $B \rightarrow D$, $CD \rightarrow A$, $AD \rightarrow B$.
R(A,B,C,D) with FD's $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow A$.
Indicate all the BCNF violations. Decompose the relations, as necessary, into collections of relations that are in BCNF.

Solution.
a)
A+=A, B+=BD, C+=C, D+D
AB+=ABCD
BC+= BCDA
CD+= CDAB
AD+= ADBC

$B \rightarrow D$ is a BCNF violating FD, so we decompose R into R1(B,D) and R2(A,B,C).

b)
A+=ABCD
B+= ABCD
C+= ABCD
A+= ABCD.
Thus, the left sides of all these FDs turn out to be keys. Therefore all other combinations on the left side would be super-keys. No BCNF violation.