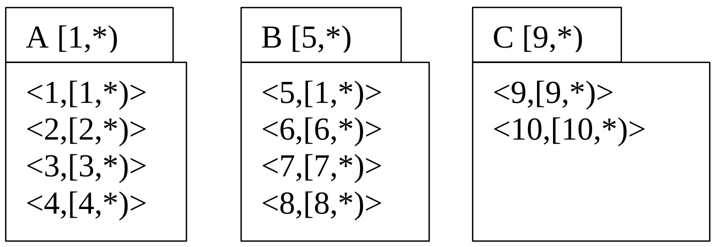
Boston University
CAS CS 562: Advanced Database Applications
Homework #2
Fall 2020

Example Solutions

Problem 1

a) We show after every 10 operations the blocks(pages) that store the records, the Access Forest, and the AT table (you can also show AT only at the end). For each block, we show the name and the lifespan of the block and the records stored there. For each record, we store the record id and the lifespan (insertion and deletion time). The * is used to represent now.

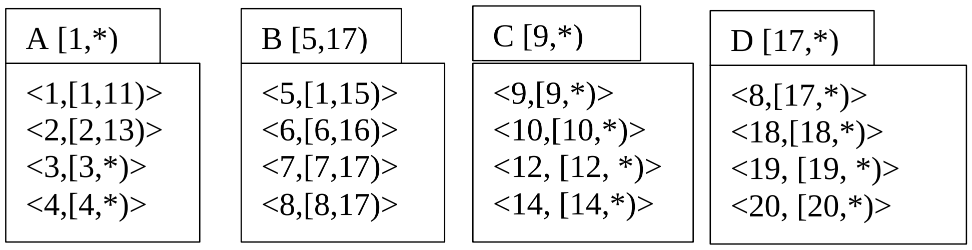
10 operations:



AF: $$\begin{tikzcd} SP \arrow[r, leftrightarrow] & A \arrow[r, leftrightarrow] & B \arrow[r, leftrightarrow] & C \end{tikzcd}$$

AT:	
1, A	
5, B	
9, C	

20 operations:



AF: $$\begin{tikzcd} SP \arrow[r, leftrightarrow] & A \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & C \arrow[r, leftrightarrow] & D \\ & B & & \end{tikzcd}$$

AT:	
1, A	
5, B	
9, C	
17, D	

30 operations:

A [1,*)
<1,[1,11)>
<2,[2,13)>
<3,[3,*)>
<4,[4,*)>

B [5,17)
<5,[5,15)>
<6,[6,16)>
<7,[7,17)>
<8,[8,17)>

C [9,*)
<9,[9,*)>
<10,[10,24)>
<12, [12, *)>
<14,[14,25)>

D [17,*)
<8,[17,*)>
<18,[18,28)>
<19,[19, 29)>
<20, [20,*)>

E [21,*)
<21,[21,*)>
<22,[22,*)>
<23, [23, *)>
<26, [26,*)>

F [27,*)
<27,[27,*)>
<30,[30,*)>

AF:
$$\begin{tikzcd}
SP \arrow[r, leftrightarrow] & A \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & C \arrow[r, leftrightarrow] & D \arrow[r, leftrightarrow] & E \arrow[r, leftrightarrow] & F \\
& B & & & &
\end{tikzcd}$$

AT:
1, A
5, B
9, C
17, D
21, E
27, F

40 operations:

A [1,*)
<1,[1,11)>
<2,[2,13)>
<3,[3,*)>
<4,[4,*)>

B [5,17)
<5,[5,15)>
<6,[6,16)>
<7,[7,17)>
<8,[8,17)>

C [9,32)
<9,[9,32)>
<10,[10,24)>
<12, [12, 32)>
<14,[14,25)>

D [17,31)
<8,[17,31)>
<18,[18,28)>
<19,[19, 29)>
<20, [20,31)>

E [21,*)
<21,[21,*)>
<22,[22,*)>
<23, [23,*)>
<26, [26,*)>

F [27,39)
<27,[27,37)>
<30,[30,38)>
<8,[31,39)>
<12,[32,39)>

G [33,*)
<33,[33,*)>
<34,[34,*)>
<35,[35,*)>
<36,[36,*)>

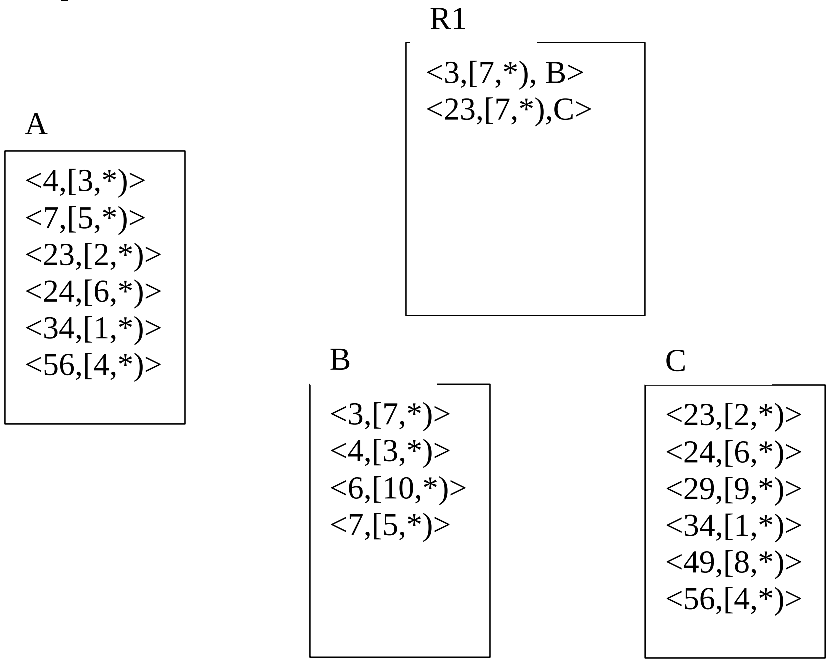
H [39,*)
<8,[39,*>
<40,[40,*)>

AF:
$$\begin{tikzcd}
SP \arrow[r, leftrightarrow] & A \arrow[r, leftrightarrow] \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & & E \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & G \arrow[r, leftrightarrow] & H \\
B \arrow[rr, leftrightarrow] & & C \arrow[d, leftrightarrow] & F & & \\
& & D & & &
\end{tikzcd}$$

AT: 1, A
5, B
9, C
17, D
21, E
27, F
33, G
39, H
(You can show array AT only at the end.)

b) We create the MVBT on the values attribute and we show the structure every 10 operations. I use the same way to represent the structure as in the paper. A leaf node contains entries of the form <value, [start_time, end_time)> and an internal node contains values <value, [start_time, end_time), pointer>, where pointer is the name of another node (block) in the tree.
Also, we assume that there is another structure (array) that stores the root of the logical B+-tree at different time instants (versions). So, if we need to search, we know from which node to start. We do not need to show this structure here.

10 operations:



Some points here.
i) Block (page) A is the initial block that is at the same time a leaf node and a root. When it is full and the next operation is an insertion, we need to do a Version split on A. So, at this point A becomes dead and we create a new page A*. However, A* has a strong version overflow (it has 6 entries) and we need to split based on the search key values. Thus we create two new pages B and C and a new root R1. The root R1 has references to B and C, not to A, since A is already dead. A was the root of the structure for the period [1,7) and this is stored in the array of roots that we mentioned before. So, we do not need to anything on A from now on.

ii) The order of entries inside each block is not very important. They can be ordered either by time or by search key. I used the search key here but if you use another order it is fine. The important thing is to have the correct entries in each block.

20 operations:

A	R1
<4,[3,*)>	<3,[7,*), B>
<7,[5,*)>	<23,[7,12),C>
<23,[2,*)>	<23,[12,16),D>
<24,[6,*)>	<27,[16,*),F>
<34,[1,*)>	<29,[12,16),E>
<56,[4,*)>	

B	C	D	E	F
<3,[7,17)>	<23,[2,*)>	<23,[2,13)>	<29,[9,*)>	<27,[12,*)>
<4,[3,*)>	<24,[6,*)>	<24,[6,16)>	<49,[8,*)>	<29,[9,*)>
<6,[10,*)>	<29,[9,*)>	<27,[12,*)>	<55,[14,*)>	<49,[8,*)>
<7,[5,15)>	<34,[1,11)>		<56,[4,*)>	<55,[14,*)>
<8,[19,*)>	<49,[8,*)>			<56,[4,*)>
<19,[18,*)>	<56,[4,*)>			<122,[20,*)>

Some other points here. Notice that over different time instants (versions) the corresponding set of values that are associated with a block (for example B) can be different. However, this is not a problem since at different time instants the tree is also different. The important issue again is that the structure is correct and allow you to return the correct answer ta each time instant.

30 operations:

A	R1	R2
<4,[3,*)>	<3,[7,*), B>	<3,[7,27), B>
<7,[5,*)>	<23,[7,12),C>	<4,[27,*), I>
<23,[2,*)>	<23,[12,16),D>	<27,[21,*),G>
<24,[6,*)>	<27,[16,21),F>	<49,[21,*),H>
<34,[1,*)>	<29,[12,16),E>	
<56,[4,*)>		

```
B
<3,[7,17)>
<4,[3,*)>
<6,[10,24)>
<7,[5,15)>
<8,[19,*)>
<19,[18,28)>
```

```
C
<23,[2,*)>
<24,[6,*)>
<29,[9,*)>
<34,[1,11)>
<49,[8,*)>
<56,[4,*)>
```

```
D
<23,[2,13)>
<24,[6,16)>
<27,[12,*)>
```

```
E
<29,[9,*)>
<49,[8,*)>
<55,[14,*)>
<56,[4,*)>
```

```
F
<27,[12,*)>
<29,[9,*)>
<49,[8,*)>
<55,[14,*)>
<56,[4,*)>
<122,[20,*)>
```

```
G
<27,[12,*)>
<29,[9,*)>
<32,[22,*)>
<43,[21,*)>
<44,[23,*)>
```

```
H
<49,[8,*)>
<50,[30,*)>
<55,[14,25)>
<56,[4,*)>
<72,[26,*)>
<122,[20,*)>
```

```
I
<4,[3,*)>
<8,[19,29)>
<13,[27,*)>
<19,[18,28)>
```

40 operations:

```
A
<4,[3,*)>
<7,[5,*)>
<23,[2,*)>
<24,[6,*)>
<34,[1,*)>
<56,[4,*)>
```

```
R1
<3,[7,*), B>
<23,[7,12),C>
<23,[12,16),D>
<27,[16,21),F>
<29,[12,16),E>
```

```
R2
<3,[7,27), B>
<4,[27,*), I>
<27,[21,*),G>
<49,[21,33),H>
<49,[33,*),J>
```

```
B
<3,[7,17)>
<4,[3,*)>
<6,[10,24)>
<7,[5,15)>
<8,[19,*)>
<19,[18,28)>
```

```
C
<23,[2,*)>
<24,[6,*)>
<29,[9,*)>
<34,[1,11)>
<49,[8,*)>
<56,[4,*)>
```

```
D
<23,[2,13)>
<24,[6,16)>
<27,[12,*)>
```

```
E
<29,[9,*)>
<49,[8,*)>
<55,[14,*)>
<56,[4,*)>
```

```
F
<27,[12,*)>
<29,[9,*)>
<49,[8,*)>
<55,[14,*)>
<56,[4,*)>
<122,[20,*)>
```

```
G
<27,[12,39)>
<29,[9,32)>
<32,[22,*)>
<38,[40,*)>
<43,[21,*)>
<44,[23,*)>
```

```
H
<49,[8,*)>
<50,[30,*)>
<55,[14,25)>
<56,[4,*)>
<72,[26,*)>
<122,[20,31)>
```

```
I
<4,[3,*)>
<8,[19,29)>
<13,[27,37)>
<16,[36,*)>
<19,[18,28)>
<21,[34,*)>
```

```
J
<49,[8,*)>
<50,[30,38)>
<56,[4,*)>
<57,[35,*)>
<72,[26,*)>
<81,[33,*)>
```

Again, we assume that there is an auxiliary structure that keeps the root blocks over time (you did not have to show it). This structure will contain: (A,[1,7)), (R1[7,21)), (R2,[21,*)).

c) We will use the Snapshot Index to answer question 1) and the MVBT to answer question 2) (If you answered the questions using also the other index it is also fine).

1) We use the AT array to find the acceptor block at time 30. It is block F. Then we use the algorithm from the slides (or paper) to search the Access Forest. We have to visit: E, A, C, B, D, SP. Actually, SP is a virtual page (block) and we do not have to visit it. So, the pages that we visit are: F, W, A, C, B, and D.

2) We need to find the root block at time instant 15. It is block R1. We read R1 and we check the entries that are alive at time 15. Then we follow the pointers that intersect the query and we retrieve the pages: D and E. So, the pages that we need to visit are: R1, D, and E.

Problem 2

The answer to this question is discussed in the Snapshot Index paper. The main idea is to keep a pointer from each block to the next acceptor block after this block. We can use the next pointer since this is not used after a block becomes dead. So, we have a linked list of the acceptor blocks.

When we have a query [t_s, t_e], first we use the Snapshot Index to find all the pages that were alive during the time ts. For each of these pages, we read all the records inside and we report the records that were alive during the query period. After that, we find the last page in this set, say X, and we use the next pointer to find the next block. We continue retrieving the next blocks one by one using the next pointer. These blocks contain records that were created after t_s and we report all these records. We stop when we find a block that has records that were inserted after the t_e. From this block, we report only the records that satisfy the query.

The complexity of this algorithm is optimal and is proved in Theorem 2 of the Snapshot Index paper. In particular, the space remains linear since we still use O(N/B) pages to store the index and we report the answer using: O($\log_B$(N/B) + r/B) I/Os, where r is the size of the answer.

Problem 3

We use the definitions of DTW and LCSS and the Dynamic programming algorithms to find the answer. Next, we show the DP matrix and the final answer:

DTW

(5,5)	19	11	20	20	18	22
(4,4)	13	7	18	17	18	24
(4,5)	9	5	16	18	18	23
(3,4)	4	2	16	18	20	27
(2,3)	1	2	16	18	22	31
A / B	(2,2)	(3,3)	(15,2)	(4,3)	(4,5)	(6,8)

DTW(A,B)= 22

LCSS

(5,5)	1	2	2	3	4	4
(4,4)	1	2	2	3	3	3
(4,5)	1	2	2	2	3	3
(3,4)	1	2	2	2	2	2
(2,3)	1	1	1	1	1	1
A / B	(2,2)	(3,3)	(15,2)	(4,3)	(4,5)	(6,8)

LCSS(A,B) = 4

Problem 4

(a) When the lower bounding lemma does not hold, it means that GEMINI cannot provide the correct answer all the time and cannot give a guarantee on how many correct answers will miss.

Notice that the framework can still be used as a heuristic and if the distance in the feature space is mostly (even though not always) lower bounding, then we can still get some results that will be close to the correct answer. On the other hand, if the distance in the feature space is completely independent of the actual distance (this means that the transformation is not really relevant), then the results that we will get will be very different than the correct answer.

(b) There a couple of approaches to do that. One is to modify the distance in the feature space and divide it by 2. That is, use the following distance when you search in the feature space:

$D'(F(x), F(y)) = D_{feature}(F(x), F(y))/2$

Another approach is: for the RangeQuery, replace ε with 2ε in the search. For the K_NNQuery, replace εmax with 2εmax.