Exercise 4: TDT4145 - Data Modelling, Databases and Database Management Systems

Alexander J. Ohrt, Martin Olderskog, Jim Totland

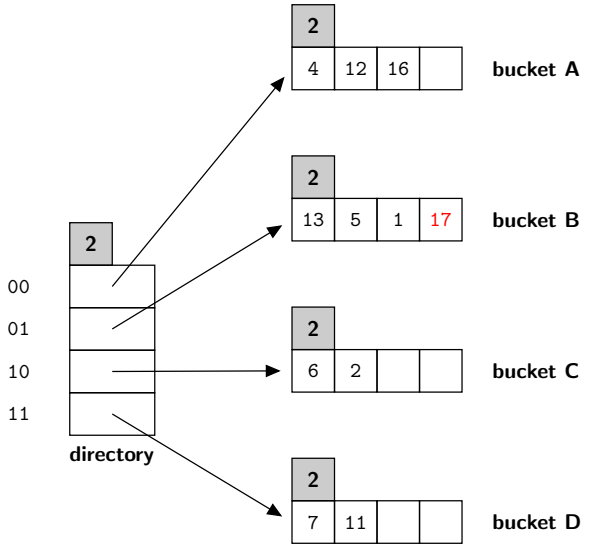
April 2021

Task 1: Extendible Hashing

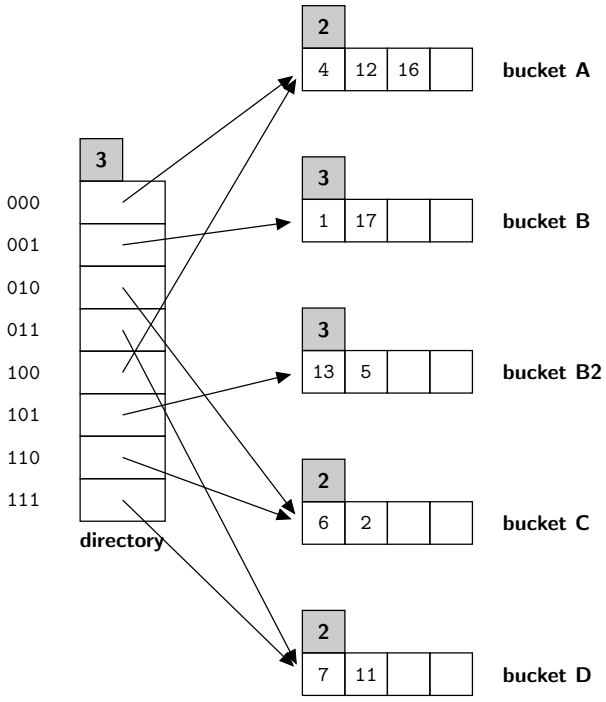
We assume a hash function h(K) = K mod 32.

Key	Binary
7	00111
4	00100
13	01101
12	01100
11	01011
5	00101
6	00110
1	00001
2	00010
16	10000
17	10001

When inserting into an extendible hashing structure, we begin with a directory with 4 slots and 4 data blocks (buckets). Both the global and local depth is equal to 2. Each block has space for three keys. We start by inserting each key by looking at their binary representation.

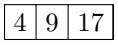


The first directory doubling occurs as block B overflows when we try to insert the last key (17). The solution to this problem is to split the block. Since the local depth cannot be higher than the global depth, we also have to increase the size of our directory. After performing directory doubling and adding another block (B2) the final hashing diagram is

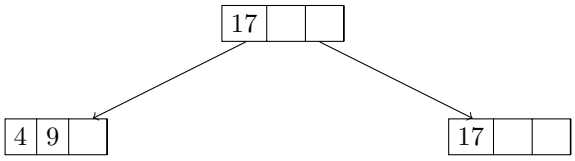


Task 2: Construction of B+ tree

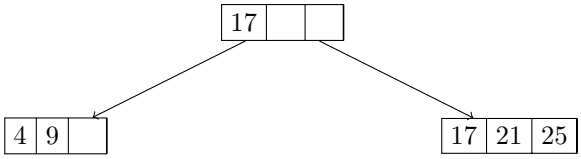
We are asked to insert keys 9, 17, 4, 21, 25, 26, 8, 10, 30, 22 and 32, in the given order, into an empty B+ tree where each block can store 3 records and blocks that are not leaves can store up to 4 block identifiers. Note that the mutual pointers between each neighboring node in each level are not shown in the figures, since these are not of main interest in the task.
After we insert the first three keys (9, 17, 4) we get the following "tree"



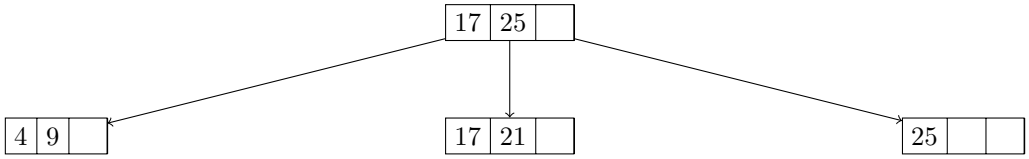
As the next key to be inserted is 21, we will have to split the node at 17, as the only leaf node has been filled. This gives the following B+ tree



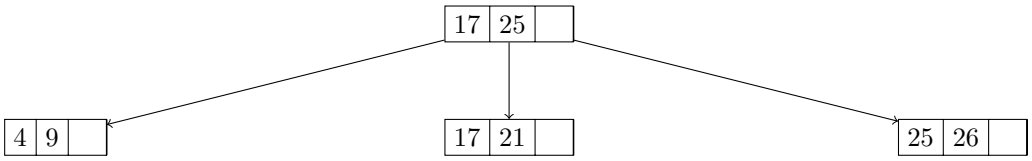
Then, we insert the keys 21 and 25. We cannot yet insert 26, as the rightmost leaf block is full.



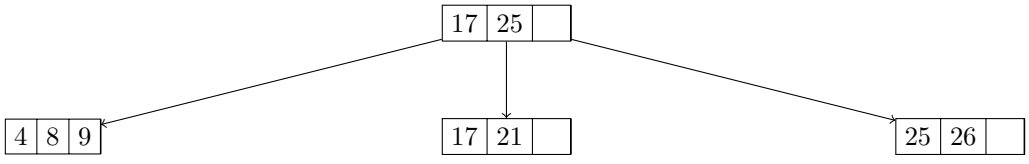
Since the next number to be inserted is 26, we split the rightmost leaf node and insert 25 into a new leaf node.



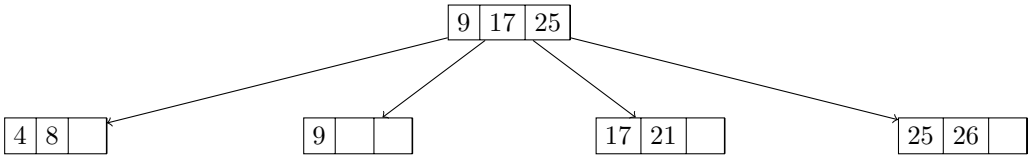
After splitting, we may add the key 26.



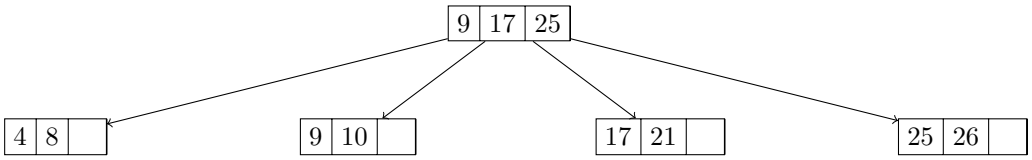
Next we insert 8 between 4 and 9 in the leftmost leaf node.



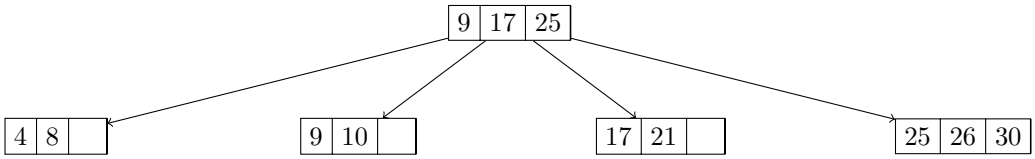
Since the next key to be inserted is 10, which is less than 17, we first have to split the leftmost leaf node.



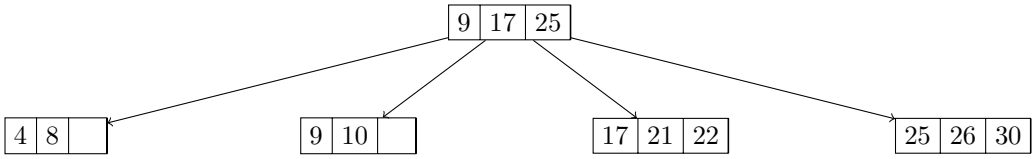
After this is done, we may insert 10 to the newly created node.



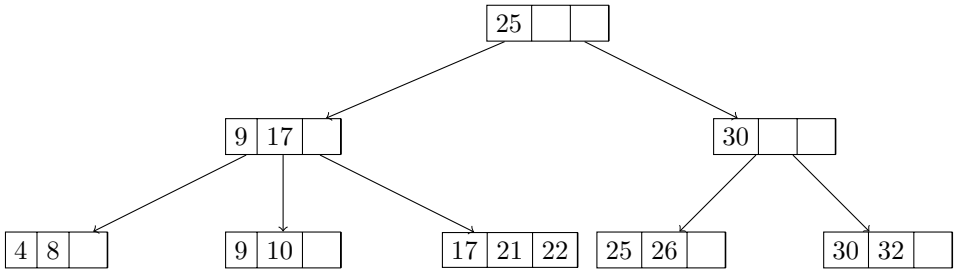
We then proceed by inserting the next key, which is 30.



We then insert the next key, which is 22.



The final key is 32. It is apparent that the root node already has 4 pointer values (block identifiers), which means that we cannot simply split the rightmost leaf node anymore. Hence, one more level needs to be added to the tree.



Task 3: Storage and queries with a clustered B+ tree

a)

The leaf level contains all records. Since each block can contain 8 records, and the fill degree is assumed to be $\frac{2}{3}$, the amount of blocks must be $\frac{10000}{8 \cdot \frac{2}{3}} = 1875$.

b)

Assuming that each person has a post of 250 bytes, the tree must have $\lceil \frac{1875}{250 \cdot \frac{2}{3}} \rceil = 12$ blocks at level 1. Since level 1 has 12 blocks, level 2 must have 12 records, since each record in level 2 points to one block in level 1. Each block can store $\frac{2048}{8} = 256$ records, so we clearly only need one block on level 2 of the tree.

c)

An estimated amount of blocks that are accessed by the SQL queries is given below. First of all, the tree has height 3.

1. The number of blocks accessed are 3. The reason is that PersonID is an index. Moreover, since the B+ tree is clustered, one only has to locate the leaf with the given unique ID, where all the selected information is stored. Hence, the number of blocks is 3, since one has to traverse the tree from top to bottom, through one block in each level.

2. The number of blocks accessed are 1877. The reason is that one has to traverse the clustered B+ tree from top to bottom, which implies access to 2 blocks, before traversing the leaf-node level through all the records, which are 1875 blocks.

3. The number of blocks accessed are 1877, for the same reasons as the prior point. This is possible since the B+ tree is ordered in ascending order from left to right in the leaf-node level.

4. The number of blocks accessed are $\lceil 2 + 0.05 \cdot 1875 \rceil = 96$. The reason is that one still has to traverse from top to bottom in the tree, which entails 2 block-accesses. Moreover, since the B+ tree is sorted in ascending order from left to right, one can begin to traverse from the leftmost block in the leaf node and traverse towards the right, until the last block that satisfies the condition is met. The amount of blocks that satisfy the condition is assumed to be 5%.

Task 4: Queries with heap and unclustered B+ tree

An estimated amount of blocks that are accessed by the SQL queries is given below.

1. The number of blocks accessed is 1250. In this case, the index, i.e. the unclustered B+ tree is not helpful, since the query is requesting all information from the Person-table. Hence, the heap file is traversed in its entirety, which implies that 1250 blocks are accessed.

2. The (maximum) number of blocks accessed is 1250. The reason behind this is that the B+ tree is an index on LastName and not PersonID, which means that it is not helpful when the condition is not regarding the index. Hence, the heap file is traversed until we find the requested record.

3. The number of blocks accessed are $2 + 1$ per LastName $+ 1$ per RecordID. The reason is that the B+ tree has to be traversed from top to bottom, which implies 2 blocks accessed. Moreover, assuming that there may be several blocks in the tree that have

this particular last name, one block in the heap file has to be accessed for each, since the index is unclustered. Moreover, if each of these LastName-blocks have several record-pointers to the heap file, each of these blocks in the heap file also need to be accessed. For example, if we assume that there are 3 people with surname 'Søkerud' and that all the records in the B+ tree with 'Søkerud' as LastName are located in the same block in the tree, the number of blocks accessed is $3 + 3 = 6$; 3 for blocks for traversing down the B+tree and accessing the relevant block, and then we need to access a block in the heap-file for each relevant record.

4. The number of blocks accessed are $2 + 300 = 302$. The reason is that the B+ tree has to be traversed from top to bottom, which implies 2 blocks accessed. Moreover, the entire bottom level has to be traversed, which yields 300 block accesses.

5. The number of blocks accessed are $3 + 1 = 4$. Assuming that there is room for the last name "Persen" in a block in the index, and no re-ordering of any records in the index has to take place, it takes 3 block accesses to find the correct block-placement. Moreover, one more block is accessed in the heap file, where the rest of the information about the person is stored.

Task 5: Nested loop join

Since Student has the lowest amount of blocks, this is used as the outer-loop-factor, as it will give the smallest amount of blocks read in total. Since we have 34 blocks available in buffer, we may use 32 blocks to read in the Student-blocks, since we need to use one block to read in ExamReg-blocks iteratively and one block for the output. The amount of read-ins of the students are $800/32 = 25$, in which 12800 blocks must be checked per iteration. This means that, the total number of blocks that are read in throughout the join is $25 \cdot (32 + 12800) = 320800$. In comparison, choosing ExamReg as the outer-loop-factor would give 332800 read-ins for the entire join operation, which is 12000 more blocks that need to be read.

Task 6: Transactions

a)

- Ensures atomic access to data and consistent database states.
- Supports concurrency.

b)

- A - Atomic. The atomic property means that each transaction is regarded as one unit, and it should not be split into smaller parts. This entails that either all queries in the transaction are completed without fault, or none of them will be committed.
- C - Consistent. The transaction ensures that the database state stays consistent, i.e. primary keys are NOT NULL, no reference anomalies, checks pass, etc.

- I - Isolation. Isolation has to do with concurrency. All transactions are isolated from one another, such that they are independent of each other.
- D - Durable. The queries in a transaction are permanent in the data base, once they have been committed.

c)

- H_1: Unrecoverable, since c2 is committed before c1.
- H_2: ACA, since c1 is committed before X is read. Moreover, this is not strict, e.g., since transaction 2 writes to Y before transaction 3 i s committed, which wrote to Y first.
- H_3: Recoverable, since transaction 1, which reads Y, is committed after transaction 3, which wrote to Y. Moreover, this is not ACA, since Y is read in transaction 1 before transaction 3 is committed.

d)

Two operations in a schedule are in conflict if all the following hold

- The operations belong to different transactions.
- Both operations operate on the same data object.
- At least one of the operations is a write operation.

e)

The presedence graph is shown in figure 1. Hence, the schedule is conflict serializable, since there are no cycles in the graph.

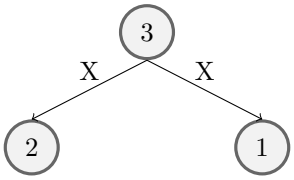


Figure 1: Precedence graph.

f)

A deadlock means that two or more transactions are mutually waiting for each other's lock.

g)

H_4 is rewritten with rigorous two-phase locking in figure 1.

T_1	T_2	T_3
		$wl_3(X)$
		$w_3(X)$
trylock(X)		
	trylock(X)	
		$rl_3(Y)$
		$r_3(Y)$
		$c_3; ul_3(X, Y);$
$rl_1(X)$	$rl_2(X)$	
$r_1(X)$	$r_2(X)$	
$c_1; ul_1(X);$	trylock(X)	
	$wl_2(X)$	
	$w_2(X)$	
	$c_2; ul_2(X)$	

Table 1: H_4 rewritten with rigorous two-phase locking.

Task 7: Crash recovery with Aries

a)

T_2 is a winner because it is committed before the crash. On the other hand, T_1 and T_3 are not committed before then crash and are hence losers.

Since T_1 and T_3 are losers, they are given the UNDO operation. Since T_2 is a winner, it is given the REDO operation.

b)

The resulting transaction table and dirty page table are given below in table 2 and 3, respectively.

Transaction_ID	Last_LSN	Stauts
T_1	151	Active
T_2	150	Committed
T_3	152	Active

Table 2: Transaction table.

Page_ID	Rec_LSN
A	148
B	149

Table 3: Dirty Page table.