Implementation of DBMS
Exercise Sheet 7, Solutions
Klingemann, WS 2024 / 2025

1) Suppose that blocks can hold either three records, ten key-pointer pairs, or fifty pointers.
a) We use the indirect bucket scheme. If each search-key value appears in 10 records, how many blocks do we need to hold 3000 records and its secondary index structure? How many blocks would we need if we did not use buckets but a key-pointer pair for each record?
Solution:
For the data file we need $\lceil$(3000 records) / (3 records / block)$\rceil$ = 1000 blocks.
When we use the indirect bucket scheme, we need to store

- 300 key-pointer pairs as we have 300 distinct key values. This requires $\lceil$(300 key-pointer pairs) / (10 key-pointer pairs / block)$\rceil$ = 30 blocks
- in the buckets one pointer for each record so that we need $\lceil$(3000 pointers) / (50 pointers / block)$\rceil$ = 60 blocks

As a result, we need 90 blocks for the index and 1090 blocks including the data file.
When we use instead a key-pointer pair for each record we need
$\lceil$(3000 key-pointer pairs) / (10 key-pointer pairs / block)$\rceil$ = 300 blocks for the index and 1300 blocks including the data file.

b) We want to retrieve all records for a particular search key. In doing so, we sequentially scan the key-pointer pairs in the index from the beginning until we find the search key value we are looking for. We assume that records with the same search key never share a block. We also assume that buckets never extend across different blocks and that key-pointer pairs with the same key are all in the same block. How many blocks do you have to inspect on average, for the two scenarios in a)?
Solution:
When we use the indirect bucket scheme, we start with a sequential search of the blocks containing key-pointer-pairs. It is equally likely that we find the key in any of these blocks. Thus, the number of blocks to inspect is uniformly distributed. The best case is that we can stop after the first block. The worst case is that we need to inspect all 30 blocks. Thus, the average number of inspected blocks is (1 + 30) / 2 = 15.5 blocks. The pointer, that accompanies the key, points to the corresponding bucket so that we have to inspect exactly one bucket-block. The bucket contains pointers to the ten records in the data file. Each record is in a different block so that we have to inspect 10 data blocks. In total we need to inspect
15.5 + 1 + 10 = 26.5 blocks.
In the alternative approach we also perform a sequential search of the blocks containing key-pointer-pairs. With a similar reasoning as above, we inspect on average (1 + 300) / 2 = 150.5 blocks. However, this time the pointers point directly to the records of the data file. Therefore, the remaining blocks to be inspected are the 10 data blocks containing the records. In total we need to inspect 150.5 + 10 = 160.5 blocks.

2) We would like to sort the tuples of a relation R(A, B, C). Initially, the tuples are stored in secondary storage in a random order. At the end, they have to be in secondary storage sorted according to the values of attribute A (search key).
The following information is known about the relation.

- The relation R has 100000 tuples.
- The size of a block is 4096 bytes. Blocks have a header of 60 bytes.
- The sizes of attributes are 32 bytes for A, 200 bytes for B and 140 bytes for C. Records have a header of 28 bytes.
- Each block holding R tuples is full of as many R tuples as possible. We use unspanned storage for the records and the key-pointer pairs in c).
- A record pointer is 8 bytes.

Answer the following questions based on the information above.
a) If we use the 2PMMS sorting algorithm with two passes, what is the minimum amount of main memory (in number of blocks) required?
b) What is the cost of the two pass sorting algorithm in terms of number of I/Os?
c) Consider the following variant of the sorting algorithm. Instead of sorting the entire tuples, we just sort the pairs <key, recordPointer> for each tuple. As in the conventional two pass sorting algorithm, we sort chunks of <key, recordPointer> in main memory and write the chunks to disk. In the merge phase, <key, recordPointer> entries from different chunks are merged. The record pointers are used to recover the rest of the tuple (from the original copy of R) and write the sorted relation to the disk. What is the cost in terms of number of I/Os?
Solution:
A record consists of the record header and the three attributes A, B and C. Therefore, the size of a record is (28 + 32 + 200 + 140) bytes = 400 bytes.
We have $\lfloor$((4096 – 60) bytes available for records/block) / (400 bytes/record)$\rfloor$ = 10 records/block and therefore, we need $\lceil$100000 records / (10 records/block)$\rceil$ = 10000 blocks to store the complete relation.
a) We know that with n main memory blocks we can sort a file with up to n(n-1) blocks. Therefore, it has to hold that n(n-1) ≥ 10000. The minimum value with this property is 101 blocks.
b) We need 4 I/O's per block so that we get 4 * 10000 = 40000 I/O's.
c) The size of a key-pointer-pair (kpp) is (32 + 8) bytes = 40 bytes.
We have $\lfloor$((4096 – 60) bytes available for kpp/block) / (40 bytes/key-pointer-pair)$\rfloor$ = 100 key-pointer-pairs/block and therefore, we need $\lceil$100000 key-pointer-pairs / (100 key-pointer-pairs/block)$\rceil$ = 1000 blocks to store all key-pointer-pairs. The variant of the algorithm then needs the following I/O's:

1. Read the relation and construct the key-pointer-pairs: 10000 I/O's
2. Write the sorted sublists of key-pointer-pairs to disk: 1000 I/O's
3. Read the sorted sublists of key-pointer-pairs from disk: 1000 I/O's
4. Retrieve the tuples pointed to by the record pointers: 100000 I/O's
5. Write the sorted relation to disk: 10000 I/O's

In total we get 122000 I/O's.