TDT4150 2019 Exam solutions

Question 1 (20%)

1.a) 1%
Index a: primary Index
Index b: secondary index

1.b) 1%
$b_f = 4096/512 = 8$

1.c) 1%
100K records/ 8(records/block)= 12500 blocks

1.d) 2%
1 IO for the leaf node and 1 for the data block= 2 blocks. Acceptable if someone has taken into account the tree-depth (x+1). Assuming that the data block is the leaf node of the index is considered partially correct.

1.e) 3%
Since it is a secondary index each record may be in a different block. So we will access 10K blocks for the data, 1 for the leaf node of the index and some for the pointers of the disc blocks. Any answer that takes into account the 10K blocks for the data (i.e. is based on one block access per record) is correct.

1.f) 3%
linear search = 6250 IOs if the id exists, 12500 if the id does not exist. Assuming the id exists is also acceptable.
primary index search = 2 IOs. One for the leaf node of the tree and one for the record. Correct if one takes into account the non-leaf nodes of the index. Partially correct if one assumes that the leaf node of the index is the data block.

1.g1) 3%
linear search = 12500 IOs on average.
index search = 5 disck blocks for the data and 1 disk block for the index leaf node = 6. Correct if one takes into account the non-leaf nodes of the index.

1.g2) 3%
linear search = 12500 IOs
index search = estimated records with quantity = 1 => 100K/10=10K. 10K blocks plus one for the leaf node and a few more for the pointers and one for the leaf node => approx 10K. Correct if one takes into account the non-leaf nodes of the index.

1.h)3%
linear search = 12500 IOs
index search = 50K IOs for the data one for the leaf node of the tree and some for the pointers => appox 50K IOs.

Question 2 (10%)

2.a) 7%

Blocks for A: 12500
Blocks for B: 1250

Nested loop join: 12500 * 1250 /(buffersize-2) +1250. (Inclusion of buffer optional).
15626250 IOs for buffersize =3

HashJoin: 3(12500 +1250) = 41250 IOs

2.b) 3%
B is read once and stored in the memory. A is accessed once and each tuple is checked against B

12500 +1250 = 13750 IOs

Question 3 (10%)

3.a) 5%
It partitions the space according to the data distribution. Finer partitioning where the data distribution is dense. Many children per node which results in using big parts of each page (i.e. less space wasted in half empty disc pages). There is no replication of indexed points or MBRs. The R-tree is always balanced. It offers a relatively shallow index structure for a large amount of data.

3.b) 5%
Two MBRs are generated. One containing the 6 top-left ones and one (relatively large) containing the rest. Since m=5 each node should have at least 5 points. Hence, the new point should be in the same MBR as the lower right points.

Question 4) 5%
A leaf node is entries of the form (oid, mbr, di) where oid is the id of the object, mbr is the MBR bounding the object and di document id for the respective object. A leaf node contains also a pointer to an inverted index for the documents of the objects of the node.

A non leaf node contains entries of the form (cp, mbr,cp.di) where cp is the id of a child node mbr is the MBR of the child node and cp.di is the document id of the pseudo document aggregating the documents contained in the child node. It also contains a pointer to an inverted index that indexes all the pseudo-documents of the node.

Question 5 15%

1 Trondheim 1000	a Bergen 1000	UB=2000 LB = Infinity Top-2={}
2 Drammen 3000	b Drammen 2000	UB=3000 LB = Infinity Top-2= {2b(5K), -}
3 Trondheim 4000	c Bergen 4000	UB= 5000 LB= Infinity Top-2 ={2b(5K), -}
4 Drammen 4000	d Trondheim 4000	UB= 5000 LB= 5000 Top-2= {2b(5K), d1(5K)}

At this point any other combination cannot have a better score, so the algorithm can stop and report the result. Any explanation that reports the result and states that the algorithm will stop here is acceptable. Any explanation that will state a later point of report and the stopping criterion includes both tables is partially correct. Any answer that the stopping criterion includes only one table is wrong.

Question 6 15%

X	Y	Z	
d:5	b:5	b:5	Current scores: d:5, b:10 Top2: d,b Best possible scores for items not in top-2: 15
c:3	d:4	d:4	Current scores: c:3 d:13, b:10 Top-2: d,b Best possible scores for items not in top-2: c: 11
a:3	c:4	a:2	Current scores: a: 5, b:10, c:7 d:13 Top-2: d,b Best possible scores for items not in top-2: c:9, a: 9

The top-2 is d,b and we stop. We do not need to find the exact score of b. Similarly to Question 5.

Question 7 (10%)

7.a) 3%
a) lazy primary
b) eager anywhere
c) eager primary
d) lazy anywhere

7.b) 7%
Sample answer:

Comments can be quite many and consistency and durability are not critical. In addition, a user would like to write and store a comment fast and not wait long for the message to be stored. A lazy-anywhere protocol would give the necessary scalability and low-latency that is desired for such service. Eager primary or eager anywhere are more appropriate for storing the transactions as consistency and durability is essential for not losing transaction data.

Question 8 (5%)

1. We need to use the 2-sided algorithm three times, because the first query only has one endpoint on the range and both the endpoints of the second query range land in separate cracks/pieces of the intermediate column.
2. <b,a|x,s,u,t,p,ø,j> and <a|b|j,s,p,t|u,ø,x>.
3. First query: <(x,b)>. Second query: <(b,a), (x,j), (u,p)>.

Question 9 (5%)

9.a) 2.5%

On a typical database system, records are stored row-wise, i.e the attributes of a record (or tuple) are placed contiguously in storage. In column oriented databases the values for each single column (or attribute) are stored contiguously. Row-oriented databases are write-optimized while column oriented databases are read optimized.

9.b) 2.5%

Two of the following:

Self-order, few distinct values: A column is represented by a sequence of triples, (v, f, n) such that v is a value stored in the column, f is the position in the column where v first appears, and n is the number of times v appears in the column. For example, if a group of 4's appears in positions 12-18, this is captured by the entry, (4, 12, 7). For columns that are self-ordered, this requires one triple for each distinct value in the column. To support search queries over values in such columns, Type 1-encoded columns have clustered B-tree indexes over their value fields. Since there are no online updates to RS, we can dense pack the index leaving no empty space. Further, with large disk blocks (e.g., 64-128K), the height of this index can be kept small (e.g., 2 or less).

Type 2: Foreign-order, few distinct values: A column encoded using Type 2 encoding is represented by a sequence of tuples, (v, b) such that v is a value stored in the column and b is a bitmap indicating the positions in which the value is stored. For example, given a column of integers 0,0,1,1,2,1,0,2,1, we can Type 2-encode this as three pairs: (0, 110000100), (1, 001101001), and (2,000010010).

Type 3: Self-order, many distinct values: The idea for this scheme is to represent every value in the column as a delta from the previous value in the column. Thus, for example, a column consisting of values 1,4,7,7,8,12 would be represented by the sequence: 1,3,3,0,1,4, such that the first entry in the sequence is the first value in the column, and every subsequent entry is a delta from the previous value.

Question 10: 5%

Key-Value stores: These systems have a simple data model based on fast access by the key to the value associated with the key; the value can be a record or an object or a document or even have a more complex data structure.

Document stores: These systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object). Documents are accessible via their document id, but can also be accessed rapidly using other indexes.

Graph databases: Data is represented as graphs. Entities are represented as nodes and relations between entities as edges between the nodes. Information can be retrieved trough graph traversal. Graph databases are suitable when then relations between the entities are important.

Column based databases: Column-based or wide column NOSQL systems: These systems partition a table by column into column families where each column family is stored in its own files. They also allow versioning of data values.