Distributed DBMS

1. Why does query optimization using materialized views resemble Local as View and not Global as View?

In this case, the question is whether the materialized views (MVs) can be used to answer queries written against the original database schema (i.e. without the MVs). MVs are pre-calculated results of queries, so these queries specify the MV. As the MVs act as data sources just like the original relations, this exactly resembles the situation that we find in the case of Local as View: „Relations of source schemata are expressed as views on global schema.“ (slide 32) The only significant difference is that in a typical data integration scenario, there are no sources available that are such simple views on our schema as the tables in the database system are (SELECT * FROM t) .

2. Provide a brief explanation as to why star schemes are typically not suitable for OLTP.

From the lecture you should know that OLAP and OLTP describe workloads that differ in several dimensions (see slides 50 and 51). In particular, OLTP queries are characterized by random-access queries that both read and write and require low latency and high throughput.

A star schema is denormalized and often contains a high degree of redundancy. Due to that reason larger amounts have of data to be written, which in turn can have a negative effect on write performance. Furthermore, denormalization can also lead to inconsistencies in the data.

You could also have mentioned that data in star schemata are often already pre-aggregated and therefore may not contain all the necessary data (although of course you do not have to pre-aggregate).

In addition, it would be valid to argue that a more frequent locking of the central fact table may reduce the potential for parallelizing the queries and thus reduce throughput.

3. When is bitmap compression most effective?

The most important fact you had to mention here is that for bitmap compression to be effective the number of distinct values must be small compared to the number of tuples in the database. Some solutions have only mentioned the number of distinct values (cardinality), ignoring the number of entries as a reference. You should also be aware that bitmap compression alone does not imply run length encoding and therefore the order of values is insignificant.

4. Apply bitmap compression to the string "CABBBBCCBCDBDAA" and give the result.

With the exception of minor mistakes, I think all the solutions were right:

```
  CABBBBCCBCDBDAA
A|010000000000011
B|001111001001000
C|100000110100000
D|000000000010100
```

As with 3, you were not asked for RLE and should therefore not apply it here.

Encoding and Evolution

Remember you were given the following code snippet:

```
public class IntLinkedList {
 int size;
 IntNode first;
 IntNode last;
 …

 private static class IntNode {
  int item;
  IntNode next;
  IntNode prev;
 }
 …
}
```

You were asked to answer the following two questions:

1. Give reasons why the default Java serializer should not be used here.

This question is based on item 75 taken from the book "Joshua Bloch – Effective Java – Second Edition". The example there is a list of strings, but all the arguments given by the author there also apply here. I'll summarize them here.

Space requirement:
The default Java serialization would take up much more space than necessary. Not only would the serialization include the class definitions, but also a lot of information that are not necessary for deserialization, such as the internal entries including their links to the previous and next entry. Precisely because the serialized form contains so much redundant information, it is vulnerable to (potentially malicious) *inconsistencies*.

Time requirement:
Because serialization happens without knowledge of the topology, the serializer must follow all links in the object graph. It would be sufficient to follow each next link. This takes more time, but can also cause *stack overflows* if the list is very large.

Expose internal representation:
The default serialized form would contain references to the private IntNode class, which would thereby become part of the public API. If the implementation changes in the future, you would never completely get away from this LinkedList implementation, even if it is no longer in use.

2. How would a more reasonable serialization look like?

All your submitted solutions proposed alternative serializations with libraries presented in the lecture such as Thrift or Avro. However, this does not fix all of the above-mentioned disadvantages (although it makes the serialization more space efficient). A much simpler serialization would be the length of the list and a correspondingly long sequence of integer primitives.

Storage & Retrieval

The compacted merge of the two segments looks like this:

accident	63
ambition	27
anxiety	78
area	56
argument	79
assistance	50
assumption	87
atmosphere	40
attitude	53

That looked good on every submission I received. There was one question whether the right input segment could be called an SSTable at all since it has duplicate values for certain keys. According to the definition on slide 21 they should indeed not be called SSTable, but for example "segment file".

For the second part of the question (in which order the elements would be accessed) there were several valid answers. You can either run the compaction of the right file separately or run the compaction and the merge in one step. For the latter method, we could run two iterators backwards over both files simultaneously. In each step we would then consider the iterator with the larger key or in case of identical keys the iterator on the new file. If this iterator’s key does not correspond to the last key written, we would include the current key-value pair in the output file and advance the iterator. We would repeat process that until both files were read entirely. This would result in the following order of access:

Iterator A	Iterator B	Output

attitude	53	assistance	50	attitude	53
atmosphere	40	assistance	50	atmosphere	40
assumption	87	assistance	50	assumption	87
argument	59	assistance	50	assistance	50
argument	59	argument	79	argument	79
argument	59	argument	85		
argument	59	area	56	area	56
area	71	anxiety	78		
ambition	62	anxiety	78	anxiety	78
ambition	62	ambition	27	ambition	27
ambition	62	ambition	14		
		accident	63	accident	63

Replication

The first part of the task was to find out which quorum configurations are possible. Our configuration consists of n=3 nodes. We know that we need w + r > n to guarantee that each query will contain the newest version of a value. Furthermore, we know that a write query succeeded with only two available nodes, so w <= 2. This leaves us with three possible configurations: r=2,w=2; r=3,w=1 and r=3,w=2.

The number of nodes that can be unavailable for a successful read query is given as n – r. This means that for r=3 all the nodes must be available, so no combinations of unavailable nodes are possible in this case. For r=2 one node can fail, which means there are three different node combinations {n1}, {n2} and {n3} that can be unavailable in case of a successful read query.

Partitioning

Among the possible disadvantages of consistent hashing are the following:

- When a node exits, all its data must be copied to exactly one other node. On the one hand, this node must bear the complete write load of this copy process and on the other hand, this node is responsible for now (expectedly) double the amount of data.

- It is assumed that all machines involved are of about the same power and should therefore be responsible for the same amount of data.

- The hash values could be non-uniformly distributed and therefore

inadvertently cause an uneven load among the nodes.

All of these problems can be mitigated by virtual nodes. Each physical node is hashed to several virtual positions on the ring. This solves the first problem because it is likely that the following nodes of the virtual nodes of a physical node are different and therefore the load is distributed among them. The second problem can be solved by making the number of virtual nodes per physical machine dependent on their power. The third problem is diminished by the larger number of nodes on the ring. As the ranges for which individual virtual nodes are responsible shrink, the probability of a disproportionately large number of values in one of the ranges decreases.

Distributed Systems

Here we were given heartbeat intervals (in s):
14, 34, 15, 11, 17, 10, 35, 29, 28, 21
The accrual failure detector would first estimate the underlying distribution:
Samle mean x=(14+34+15+11+17+10+35+29+28+21)/10=21,4
Sample standard deviation s = sqrt(((14-21,4)^2+...(21-21.4)^2)/9) = 9.42
The probability for P_later can now for example be looked up in a tableau: https://en.wikipedia.org/wiki/Standard_normal_table
Our value for Z would be (31-21,4)/9,42~=1,02 which would give us a probability of about 15,4% that the heartbeat will still arrive.
The value for φ is -log_10(0,154)~=0,81

For the parameters μ=15.0 and σ^2=100.0, the value for Z would be (31-15)/10=1.6 and the probability P_later about 5,5%.
So our estimation for the probability is off by a factor of 2.8.

Consistency and Consensus

1) First you were asked to give the Lamport timestamp according to the provided rules. Here is the solution:

e1,1: 1
e1,2: 3

e2,1: 1
e2,2: 2
e2,3: 3

e3,1: 1
e3,2: 2

2)
The following event has a larger Lamport timestamp than e2,2 but cannot have been influenced by e2,2: e1,2
The following event has a smaller Lamport timestamp than e2,2 but cannot have influenced e2,2: e3,1

3)
Vector clocks provide us with a mechanism to exactly determine which events might have influenced a certain event:
e1,1: [1,0,0]
e1,2: [2,0,2]

e2,1: [0,1,0]
e2,2: [1,2,0]
e2,3: [1,3,0]

e3,1: [0,0,1]
e3,2: [0,0,2]

Now there is only one event with a larger vector clock than VC(e2,2): e2,3, which can have been influenced by e2,2.
Also, all events with a smaller VC than VC(e2,2) can indeed have influenced e2,2: e1,1 and e2,1.

Stream Processing

Here the task was to describe an algorithm that selects a random set of fixed size from a stream where each element has equal probability to be in the sample. The usual solution in literature for this problem is reservoir sampling (https://en.wikipedia.org/wiki/Reservoir_sampling). It is an algorithm that every computer scientist should have heard about and should know its basic concept (the algorithm is also often asked for in job interviews).
For this task, variations of reservoir sampling would also have been accepted. For example, algorithms that assign a random number (from a correspondingly large space) to each element and always retain the k elements with the k largest/smallest random numbers. The only requirement for the solution is to use max O(k) memory. This would be the case here, because only the k elements and the k numbers have to remain in memory.