Questions for session 2

3.1.3 Representing data elements

1. They usually say that to keep your database as performant as possible you have to keep your tables as small as possible. So that leads to my conclusion to use VARCHAR(4) over the CHAR(4). Then I come to the question: why should we even bother using CHAR if VARCHAR is better? **(Lee)**

2. I'm wondering why should i have to use a BIT type? Why not store it in a VARCHAR type or CHAR type? Are there some performance improvements when using the BIT type or is there another reason? **(Lee)**

3. In paragraph 3.1.3, on page 89, there is a small discussion about sequences of bits. It is stated here that bits are packed eight to a byte. The remaining bits of the last byte is then filled with zeros. How do you know that these zeros are not part of the sequence of bits? E.g., how can you distinguish 00110000 and 0011 from each other? **(Niels)**

3.2.1 Building fixes-length records

4. Why do some machines allow more efficient reading and writing of data that begins at a byte of main memory whose address is a multiple of 4 or 8? **(Richard)**

5. In section 3.2.1, page 91, it is explained that the fields should all start with byte that is a multitude of 4. When this is done in the example on page 92 the record's last byte also ends with a multitude of 4 (in the example it's 304). My question is how does the system know that it doesn't belong to the interpretation of the field. **(Laurence)**

3.2.2 Record headers

6. Explain the need for record schemas in the case of fixed-format records. Why can't we immediately determine the record type just from its location in the file system? **(Patrick)**

3.2.3 Packing fixed-length records into blocks

7. They're talking about block size. In example 3.8 they say that for each block there is one block header and the rest of the space is

filled with records from the same type and if there is some space left so be it. That leads to the conclusion that if you know that you're records are fairly small (like with for instance online transactions) you should better choose a smaller block size than the default 4 KB on Windows NTFS (I think). But from I know it's possible to change the block size in some databases (at least it's possible in the Oracle database). Would it be better to change block sizes on file system level or on the database level? **(Lee)**

3.3.1 Client-Server Systems

8. The addresses in the database address space are represented with Physical and Logical Addresses. What are the differences between them? **(Ivaylo)**

3.3.2 Logical and structured addresses

9. In section 3.3.2 they speak about an offset table that creates a certain level of indirection within the block. Next to that they speak about the many advantages of this offset table, are there any disadvantages you can think of? **(Rudy)**

10. In what way can structured addresses help to find a record, and how must we structure our header to use this? **(Patrick)**

3.3.3 Pointer swizzling

11. The third paragraph proposes an "interesting option". So if i understand it correctly than he just tries to follow a memory pointer and, if this fails, some function is triggered in the database that leads us to the database pointer... it looks a little bit like the try catch exception handling in java, but in java they say that you should avoid to use the try and catch for normal programma logic, so only for real exceptions like a network connection exception and stuff because of the fact that raising exceptons in java is really costly. So don't we have more or less the same situation here? Where they propose to make use of the "failure catch" system of the database for our addresses logic? Or am i seeing it wrong? **(Lee)**

12. Which way of swizzling is the most efficient? **(Richard)**

13. What is pointer swizzling and how is used? What is the difference between automatic and on-demand swizzling? **(Ivaylo)**

14. In paragraph 3.3.3 swizzling of pointers is discussed. Not only the pointers from the block that is loaded from the secondary storage

into the main memory should be altered, but also the pointers to that specific block. Can you explain how and when these pointers to the specific block are updated from their database addresses to their memory addresses? Is this done with translation tables? On page 101, the book talks about avoiding costs related with translation tables, but then again their use is mentioned on page 102, and seems unavoidable to me. **(Niels)**

15. Imagine we use pointer swizzling. How can we ensure that if we unswizzle the pointer that it still points to the same address? **(Patrick)**

16. How can you make an unswizzling operation efficient? i.e., how can we make sure that each unswizzling operation does not require a search of the entire address translation table? **(Patrick)**

3.4.2 Records with repeating fields

17. In paragraph 3.4.2, on page 110, there is a small discussion about Null values. It is stated there that when the address in the example becomes Null, a Null pointer is stored in the "to address" part of the record. What happens if the name in the example becomes Null? There is no pointer to the name field... **(Niels)**

18. How can we efficiently save space when we have NULL values in our database? **(Patrick)**

19. In which situations is having variable length records less efficient? How can we fix this? **(Patrick)**

3.4.3 Variable-format records

20. According to paragraph 3.4.3, one of the reasons to use tagged fields is to allow records with a flexible schema. This is illustrated by example 3.12. This example is about a movie star table in which every movie star can be the director of multiple movies. However, in relational databases this would normally be translated into 2 different tables and a one-to-many relation. In other words: you would not use a flexible schema. Is it true that the concept of tagged fields only makes sense in the context of non relational databases? **(Gerwert)**

21. Explain why tagged fields are very useful when we have records with many fields that may or may not appear at all. **(Patrick)**

3.4.5 BLOBS

22. They always told me not to store files like images or movies in a database, for performance reasons, so why did they create the possibility to store large files in databases? Wouldn't it be better to store the file on the filesystem and store the path to the file in the database? **(Lee)**

3.5.1 Insertion

23. What is the best thing to do, if in case of an insertion there is not enough room in the block to fit the new record? In section 3.5.1 they speak about two major approaches ("Find space in a nearby block" and "Create an overflow block"). Which method is best practice or commonly used, concerning their advantages and disadvantages? **(Bram)**

24. Which of the discussed methods – find space in a nearby block, create an overflow buffer – of handling the case of a block not being big enough to fit a new record is used in what case? Is there some way to determine the best approach for every case on-the-fly or is that impossible or unnecessary? **(Jeroen)**

25. When inserting a new record, how can we keep the need for sliding records to a minimum, while still preserving as much disk space as possible? **(Patrick)**

26. What happens if we have records to insert in a certain block, but it does not fit in? Do we move records to another block, or do we just create a new block? What method is most efficient? **(Patrick)**

3.5.2 Deletion

27. Why do we need to use tombstones for pointers to deleted records? Can't we just replace these pointers with NULL values? Explain why. **(Patrick)**

4.1 Indexes on sequential files

28. Explain the differences between a dense index and a sparse index. In what situations do we prefer a sparse index over a dense index? **(Patrick)**

4.1.5 Indexes with duplicate search keys

29. I think example 4.8 contains a fault, if not can you explain the technique a bit further? **(Richard)**

4.1.6. Managing indexes during data modifications

30. How should a sparse index handle the insertion/deletion/sliding of a record? **(Patrick)**

31. In paragraph 4.1.6, on page 135, there is a small table with actions on a sequential file and their effect on the index file. When an empty sequential block is created, an entry in the sparse index is inserted for the block that was created. This index does not point to a key as with the rest of the indexes in the index table, because it cannot. Where does this pointer point to? **(Niels)**