1 Theory

1. In an SSD, the Flash Translation Layer (FTL) is responsible for wear leveling, meaning distributing writes evenly across the entire disk. The data in the NAND, the type of flash memory used by most SSDs, cannot be updated in-place, only by erase-and-write cycles, which each NAND only has a limited number of. Furthermore, since in an SSD whole blocks have to be deleted when we wish to delete only a single page, we have to rewrite the entire block, except the page we wish to delete, to a new block. Frequent updates like this to a set of blocks will wear the blocks holding these pages out much faster than the other blocks. Therefore, writing to all blocks uniformly over time means no block is more likely to fail than others, and we can prolong the lifespan of the SSD.

2. Sequential writes are important for performance on SSDs as when the disk reaches full capacity, garbage collection has to be performed. This becomes a very complex task when the data is written randomly, due to the FTL having to always maintain the mapping between logical and physical block addresses across the disk. When the data is written sequentially, however, invalidating blocks as eligible for erasure is done block by block since thats how they're written, and the write amplification is greatly reduced.

3. Alignment of blocks to clustered SSD pages improves performance by lowering the write latency in an SSD due to the fact that the write requests can be written to disk without any further write overhead. If the write request is not aligned to the page size, however, the SSD has to read the rest of the content of the previous clustered page, and then subsequently merge this with the updated data before being able to write the entire page back to the disk.

4. The layout of a MemTable in RocksDB follows the structure where updates are first inserted into the memtable until it reaches its maximum size. RocksDB will then subsequently generate a new memtable for new updates, and the old memtable's entries will be flushed to disk as an SSTable.

   The layout of an SSTable in RocksDB follows a block-based table format where key-value pairs are stored in sorted order in blocks, along with metadata and indexes.

5. During compaction in RocksDB, multiple SStables from the same level in the LSM-tree are merged into a single SSTable in the next level, which are together then merged into a new, even larger SStable. If a key has been marked as deleted during this process, it will be discarded and only the most recent version of a key is kept. This newly generated SSTable is then moved to the next level, and if this causes the next level to have too many SSTables, this whole process is repeated in this level.

6. LSM-trees are regarded as more efficient than B+-trees for large volumes of inserts due to the fact that the the B+-tree has to write to disk every time a new entry is inserted into the tree. LSM-trees on the other hand buffer writes in their memtables, and only flushing to disk when the memtable is full.

7. When it comes to fault tolerance, a hardware error could be a disk, power, CPU, or network failure. A software error could be a bug in the system causing errors. It could also be system crashes due to errors caused by the system having been turned on for too long. Human errors are usually due to accidental actions such as deleting or overwriting a file by mistake, or inserting wrong data.

8. To achieve fault tolerance, one could restart computers frequently to avoid crashes due to the system having been running for too long. One could also store the data across multiple computers such that in the case one computer fails, only a small portion of the data is affected, though this is hard to do on stateful systems. Another way to achieve fault tolerance is by replication, i.e. having the same data stored on multiple computers. This way, if one computer fails, the data is still available on the other computers, though this method could be expensive.

9. SQL is a relational database where relationships between different data in the database can be easily defined using joins. In a document database, however, this is not the case. This is

due to the fact that the data is stored in documents, and the documents are self-contained, i.e. not split into multiple tables. This structure also makes it bad for supporting many-to-many relationships, something SQL is good at. In the case where we wish to model a paper that has many sections and words, and additionally many authors, where each author with name and address have many written papers, we would have to store each entity in its own self-contained document. So we would have one document representing each paper with sections and words, in addition to the authors of the paper. We would have another document representing all authors with their name and address. The relationship defining which authors have written which papers would be stored in the paper document, though to extract their name and address we would have to define many-to-many relationshops not contained in any of the documents, presumably using application code or some other method. Using only self-contained documents, we would have to store a lot of redundant, duplicate data where each paper has all authors, with their names and adresses as well, listed in the document. This is however not a good solution due to the fact that in different papers, the same author could have their name spelt differently or have some name changes. Trying to get all papers written by a specific author could prove difficult due to this.

10. If the data comes in the form of self-contained documents where there are few relationshops between them, document databases are a good fit. However, in the case where anything could be related to everything, a graph database would be a better fit. For example, when we want to store data containing people and how they are related to each other, a graph database would be preferable over a document database, as the data is highly relational.

11. One should use textual encodings instead of binary data encodings for sending data when then the data needs to be inspected by humans, since textual encodings are human-readable. Also when the data is supposed to be sent and consumed by systems that use different programming languages, textual encodings are a good choice since they are supported across many different programming languages.

12. (a) Bitmap for the values:
    32: 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
    33: 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1
    43: 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    63: 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
    87: 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
    89: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0

    (b) Runlength encoding for the values:
    32: 7, 1
    33: 8, 4, 3, 1
    43: 0, 3
    63: 5, 2
    87: 3, 2
    89: 12, 3

13. Schema evolution, as well as forward and backward compatibility is supported the following way in the different systems listed below:

    - **MessagePack:** Old readers reading data that contains new fields from newer code will still be able to read the data, but the new fields will be ignored. When it comes to backward compatibility, however, new readers reading data that contains old fields from older code will not be able to read the data. This is due to the fact that the new reader will not know how to handle the old fields, and will therefore not be able to read the data. Schema evolution in MessagePack is therefore not inherently supported.
    - **Apache Thrift and Protocol Buffers:** Like in MessagePack, old readers reading data that contains new fields from newer code will still be able to read the data, and the new fields will be ignored. Unlike MessagePack, however, when it comes to backward compatibility,mnew readers reading data that contains old fields from older code will still be able to read the data. In Apache Thrift, removed fields are marked as void, and can therefore be ignored if old data containing that field is read. In Protocol Buffers, as

long as field numbers pointing to different fields are always unique and reserved, they won't ever be reused accidentally. Therefore, since the new reader knows how to handle the old fields, schema evolution in Apache Thrift and Protocol Buffers is supported.

- **Avro:** Avro also supports schema evolution, and uses two schemas to make this possible: a writer's schema and a reader's schema. When reading data, Avro resolves differences between these schemas, so that in forward compatibility, readers should be able to use the old schema to read all data written with the new schema. The same goes for backward compatibility: readers using a new schema should be able to read all data written with the old schema.