Nikolai L. Børseth

Oppg. 1:
FTL is used to prolong the lifetime and reduce the latency of the SSD. FTL implements wear leveling which distributes writes over the entire disk in order to avoid erase-write cycles that are run on the same set of blocks repeatedly. Wear leveling is typically divided into three categories:

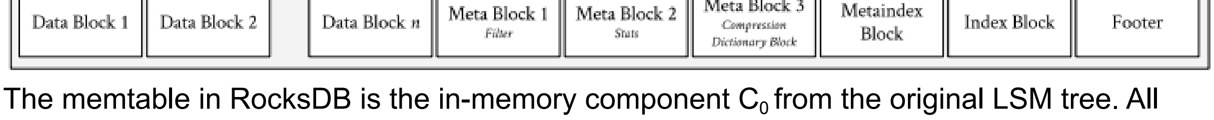
- No wear leveling
- Dynamic wear leveling: Each time a block is about to be rewritten, the original block is marked as invalid and the updated block will be written to a new location. Works well for dynamic blocks, but not for static blocks.
- Static wear leveling: Same as dynamic, but also moves static blocks periodically.

These categories come with the typical caveat for these kinds of situations, the lifetime increases the further down you go (best for static wear leveling), but so does the performance cost and complexity.
Updates are done out.of-place by maintaining a mapping between logical and physical pages. A garbage collector is responsible for cleaning up old blocks eligible for erasure. The final decision of when to erase a block is made by the garbage collection algorithm, it might for example decide to erase a block when a certain percentage of its pages are invalid. The valid pages will then be copied to a new block before erasure.

Oppg. 2:
The most important reason is the effect it has on wear leveling, garbage collection and write amplification. If using random writes, the FTL will have to work intensively to ensure wear leveling, perform garbage collection and maintain up to date mapping between logical block addresses (LBA) and the physical block addresses (PBA). The internal fragmentation the random writes might cause will cause the SSD performance to fall, and might reduce the lifetime of the SSD.
There is an exception in the case where the writes are several blocks in size. In this case random writes will work just as well as sequential writes because the writes will fill up block with equal effectivity, removing the problem of internal fragmentation.

Oppg. 3:
Write requests that are aligned with the clustered page size cause no further overhead. For requests that are larger than the clustered page size, randomized write will perform just as well as sequential writes, as we avoid the internal fragmentation problem. When writes are smaller than the size of a clustered page, randomized write performs worse as the SSD controller needs to read the rest of the content in the last clustered page and combine it with the updated data before it can be written to a new block.

Oppg. 4:



The memtable in RocksDB is the in-memory component C_0 from the original LSM tree. All updates are inserted into the memtable until it reaches its maximum size, at which point a new memtable is created where further updates will be inserted. The old memtable is then frozen and all its entries are written to the SSTable. The default index structure of the memtable is a sorted skip list. The default SSTable format is the block based table as seen above, with key-value pairs in sorted order along with metadata and indexes. When

RocksDB performs a lookup, it will check the memtable for the desired value. If it is unsuccessful it will try the SSTable. Write ahead logs are kept in case of failure and are deleted when the table is written to disk.

Oppg. 5:
RocksDB uses LSM trees and therefore the rolling merge process, which is equivalent to compaction. Compaction takes two or more SSTables as input and merges their entries into a new immutable SSTable. Keys marked as delete are excluded from the output, and if a key exists in multiple of the inputs, only the latest one is kept. There are three different compaction styles:
Leveled: The original style implemented in LevelDB. The SSTables are stored in sorted runs, and a sorted run contains multiple SSTables that does not overlap in key range and whose entries form a total ordering. Each sorted run is stored at seperate levels (L_0-L_n), with the newest being on top (L_0). Memtables are flushed to disk as SSTables in L_0, with any merge operations skipped. This may cause overlap in key range (only in L_0). Because of the possible overlap any read operations must check all SSTables in L_0, making searching L_0 more expensive than other layers.
Universal: Universal compaction differs from leveled in that the SSTables in the sorted runs can contain duplicate keys, but not overlapping time ranges. These trees might grow deeper faster, but may contain many empty layers.
FIFO compaction: All files are stored at level 0, but with only a certain lifespan. When the lifespan of the file runs out, it is deleted. The size of this tree is constant, as the oldest SSTable is deleted as soon as the database size reaches its configured maximum size. The behavior of FIFO compaction is similar to a time-to-live cache.

Oppg. 6:
B-trees usually have high write amplification, since no matter how much data is updated, the entire page must be fetched to memory. That means that if 100 bytes are changed, and the page size is 4096, the write amplification will be 4096 / 100 ~= 400. From this we can see that small and medium writes will be costly, while the performance for larger writes may outperform LSM trees.
LSM trees usually have much lower write amplification which depends on the size ratio between adjacent levels, the key range of the SSTables that is about to undergo compaction, and type of compression for each level. LSM are better when low write amplification is critical, but this comes at the cost of higher read amplification and space amplification.

Oppg. 7:
Hardware: Can be anything from a faulty disk, power outages, water damage, fire in the data center and other physical aspects. A possible mitigation is to introduce redundant hardware in the case of faulty hardware or accidents.
Software: Appears in the software of the system. Often hard to detect before they occur. This can be conflicting dependencies, software crashes, misconfigured / unrealized dynamic settings (timezone, language, etc). We can minimize software errors by performing regular testing (both unit and user) after changes to the software, update the software as technology advances, have regular timed reboots (the longer a system runs, the more likely a bug will occur) and develop / prepare responses for what to do if it crashes.
User: Humans are the ones designing and developing the system. Errors might occur when a programmer makes a mistake when developing the software (fail to account for div by 0,

functions that don't do what they are supposed to, does not take in account for outliers, etc.), or when someone spills their coffee all over the servers. The chances will be lower if the workers are experienced, motivated, rested and in good health.

Oppg. 8:
See 7, mitigations are mentioned.

Oppg. 9:
SQL is a query language used for relational models. Here each data-class has its own table, where each row has a unique identifier and columns containing the classes attributes. Some of these attributes can be references to keys for rows in other tables. This makes it possible to relate different data-classes for ease of querying.

The document model basically takes the data-class and stores it as a JSON object with the attributes being key value pairs. In SQL, different tables are joined together to create more complex objects, while in the document model all attributes of such an object are stored in a single document.

A relational model has more “moving parts”, which means it can be improved through vertical scaling (increasing processing power). While the document model is horizontally scalable, meaning that rather than pure processing power, more servers are needed.

Therefore SQL is preferable when the data is regular, predictable and reasonably separable, so that it can be spread across multiple tables. The document model can on the other hand be useful if the data is not necessarily as structured. If there are frequent changes to the model, separating the data or updating the tables, can be tricky and expensive.

<u>Give an example which shows the problem with many-to-many relationships in the document model, e.g., how would you model that a paper has many sections and words, and additionally it has many authors, and that each author with name and address has written many papers?:</u>
Using the document models means that the authors and papers have to be stored as a key-value pair. Considering that each paper is probably larger (in data), we should repeat authors instead of papers. This means that each paper would have a list of authors who wrote it, and an author might appear in several papers (as duplicates). Using SQL, papers and authors could have their own tables, and use an associative table to link between authors and papers, achieving many-to-many. A combination might have been for the best, with authors and information about the papers being stored with SQL, while the papers table points to a document actually containing the sections and words (large mostly unrelated data that can not be reused).

If each paper could only have one author however, the document model would be a good choice, as we could simply list the authors, each with a list of their own papers.

Nikolai L. Børseth

Oppg. 10:
The graph model is made for many-to-many relationships. If we had data where such relationships were common, e.g. Y shops at X and X had the customer Y showing relations between stores and customers, a graph model would be better. With a typical problem being social relations (X shops and multiple stores, and Y has multiple customers)

Oppg. 11:
a)

	43	43	43	87	87	63	63	32	33	33	33	33	89	89	89	33
32	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	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
63	0	0	0	0	0	1	1	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
89	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

b)
- $32{:}\ 7,\ 1\ (7\ zeros,\ 1\ ones,\ rest\ zero)$
- $33{:}\ 8,\ 4,\ 3,\ 1\ (8\ zeros,\ 4\ ones, 3\ zero,\ 1\ one)$
- $43{:}\ 0,\ 3\ (0\ zeros,\ 3\ ones,\ rest\ zero)$
- $63{:}\ 5,\ 2\ (5\ zeros,\ 2\ ones,\ rest\ zero)$
- $87{:}\ 3,\ 2\ (3\ zeros,\ 2\ ones,\ rest\ zero)$
- $89{:}\ 12,\ 3\ (12\ zeros,\ 3\ ones,\ rest\ zero)$

Oppg. 12:
- MessagePack: Efficient JSON encoding. No need for real data structures and changes can be made without issues with forward / backwards compatibility.
- Apache Thrift: Schema dependent binary encoding. Maintaining backwards compatibility requires every new field to be optional or have a default value. Only optional fields may be deleted. As long as a field has an unique tag, new code can read old data.
- Protocol Buffers: Like Apache Thrift, but with a different array implementation.
- Avro: Since there is no tag in Avro, unlike Apache Thrift and Protocol Buffers, the possibility of adding and removing fields depends on if they were provided with a default value. Binary data can only be decoded if the sender and receiver uses the same schema. The reader with a new version of the schema might have aliases for names from earlier versions, and be able to read a field by looking up the alias, but a reader with an older schema may not do the same. This means that name change is backwards compatible, not forwards.