Assignment 1

Q1: How does the Flash Translation Level (FTL) work in SSDs?

The Flash Translation Level is an additional software layer located in the SSD controller, between the file system and the NAND flash memory, which allows operating systems to read and write to NAND flash memory devices in the same way as disk drives. This layer provides the translation from virtual to physical addresses, also known as Logical Block Addressing (LBA), and includes mechanisms for wear leveling and garbage collection, which it calls when required.

Because of the relatively long erase times on NAND flash memory, as ERASE operations are done one block at a time, the Flash Translation Level writes the data to another physical page and marks the data contained in the previous physical page as invalid. This makes the long erase time transparent, because instead of erasing a block to be able to rewrite it, it writes the data block directly to another location/page.

> One last note about wear leveling. Solid State Drives has one limitation in addition to all its advantages; limited number of writes possible at each address, also called wearing. For expanding the life expectancy of these drives, we have to try to evenly spread the data, also called as **wear leveling**. The FTL is responsible for this.

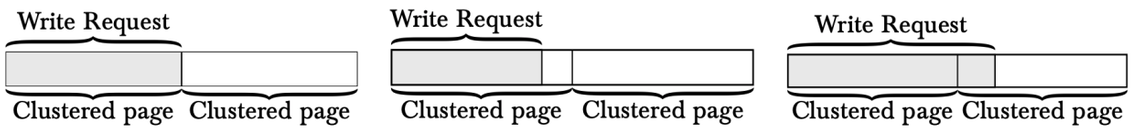
Q2: Why are sequential writes important for performance on SSDs?

Sequential writes is significantly faster than random writes, both for SSDs and HDDs. This is because of the task of finding an address (seek) is only done once, instead of after each write. As each random write gets smaller, you pay more and more of a penalty for the disk seeks. The size of the gap of performance is therefore based on the size and type of storage media. SSDs has for example better random write performance than HDDs because of the lack of moving write-head. The reason why sequential writes is important for the performance of SSDs, is because of the advantage of being less reliant on garbage collection. This is because of the blocks being invalid right next to each other, and once we find a writeable memory location, we can write all the data sequential and reaching a write amplification of close to 1.

Q3: Discuss the effect of alignment of blocks to SSD pages.

Using blocks and pages, the size of the write requests is important to consider as it has an impact on performance. Ideally, a write request should be a multiple of the clustered page size, as these can be written to disk with no further write overhead. Write requests which is not aligned generate overhead as the SSD controller needs to read the rest of the content in the last written page, and combine it with the new data and write to a new

location. This can also lead to fragmentation, and we want to have data relative to each other, stored closely together - giving the principle of spatial locality.



(a) W = P **(b)** W < P **(c)** W > P

A figure illustrating the importance of alignment talked about above. Figure fetched from Dybvik, Chapter 2: Survey of Storage, Indexing, and Database Systems.

Q4: Describe the layout of MemTable and SSTable of RocksDB.

Sorted String Tables (SSTables) is a persistent file format for taking in-memory data stored in MemTables, order it for faster access, and store it on disk in a persistent, ordered, immutable set of files. MemTable is an in-memory datastructure holding data before flushed as an SSTable, acting like a write-back cache. After reaching a certain size, it is flushed as a SSTable to the database.

Let's take an example server which is doing all the IO operations of reads and writes to the db. The server has a MemTable and a disk (SSD or HDD). We push key-value-pairs to the MemTable, and when it reaches a certain size - we flush it to the disk as an SSTable. Remember, SSTable is a sorted MemTable stored on disk. We sort the values as we append them to the MemTable, fulfilling the characteristics of SSTable right away. SSTable is immutable, once written it is set, and for each flush we create a new SSTable. If we write a new value to the same key, it's not a problem as during flush it will be stored in a new SSTable, and we have two SSTables containing the same key but different values. Each SSTable have a timestamp for checking which of the values is the most recent. For reading we first check the MemTable, then proceed to check each of the SSTable on disk if not found in MemTable.

Q5: What happens during compaction in RocksDB?

The worst case read time from disk in the example of question 4 is $nlog(n)$, where n is the number of SSTables; as number of SSTables in database increase, the worst case read time also increases. Also as we keep flushing SSTables to disk, which is immutable, the same key may be present in multiple SSTables. The solution is to implement a compactor which usually runs in the background and merges SSTables by removing redundant & deleted keys, creating more compact/merged SSTables. The compactor does not edit the SSTables directly, but creates a larger single SSTable, leaving out the redundant and deleted keys. The compaction process is usually ran every thirty minutes or so

Q6: Give some reasons for why LSM-trees are regarded as more efficient than B+-trees for large volumes of inserts.

LSM-trees is a data structure with performance characteristics that make it very attractive to store data with high insert and update rates. This is reached because of the

structure using MemTables and SSTables. LSM trees primarily use three (four) concepts to optimize reads and writes:

- SSTables
- MemTable
- Compaction
- (Bloom filters)

> *A little note about Bloom filters as I wanted to read up on the concept:*
> As the number of SSTables increases (before compaction), the read latency per key increases. It is not desirable to fully depend on the compaction, running only every 30min or so. Therefore, we want to implement some kind of filtration. This is where Bloom filters comes into place. The bloom filter is a probabilistic data structure which gives the opportunity to check if a key may be present or not, with $O(1)$ complexity. The function does not say that it is 100% certain to be included, but can rather say with certainty if it is definitely not present. This improves the read performance for keys to some extent.

For a write request in a LSM-tree, it is first stored in WAL (Write-ahead log) which stores the operation on disk for backup if anything were to go wrong. Then it is written to the MemTable, which is flushed as SSTables onto disk when it reaches a certain size. A bloom filter is also created when a new SSTable is created. This approach is more efficient for larger writes than the m-way tree structure of B-trees, as data is just appended and flushed when the MemTable is full. It also has minimized storage overhead, which is desirable for high write throughput.

It can also be mentioned that the disadvantages of LSM-trees are slow reads (the usage of bloom filters helps to some extent), and that the compaction/merging process sometimes interfere with the performance of ongoing reads and writes.

Q7: Regarding fault tolerance, give a description of what hardware, software and human errors may be?

Regarding hardware faults, this may come by hard disks crash, faulty RAM, power outage etc. A hard disk has an estimated mean time to failure (MTTF) of 10-50years. This may be one of the biggest advantages of clusters, storing media on multiple devices, as hard drives may die.

For software errors on the other hand, this may correlate multiple hard drive faults, as computers usually run the same software. An approach to avoid this is to use N-version programming, running different software on each computer. Software faults may also be dormant for long times, and suddenly occur on a large amount of machines. Systematic kill and restart may be a solution for fixing this.

Lastly, we have human errors. Configuration errors by humans is estimated to be the biggest cause of outages. We humans stand for 10-25% of hardware faults. How may we fix this? Well, we could create well-designed APIs, train with real data using sandboxing, test thoroughly, monitor performance and errors - and give some tasks of

monitoring over to autonomic computing. Systems may also be trained for fault tolerance performance.

Q8: Compare SQL and the document model. Give advantages and disadvantages of each approach. 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?

There is different criteria for the optimal way of storing data in different projects, and in many cases you will have to weigh the advantages and disadvantages of the relational model up against the document model. The first advantage of the relational model is the predefined schema, which combined with transactions is making sure that the model follows the principles of ACID - Atomicity, Consistency, Isolation and Durability. In addition, you have the support for complex joins. This is where the advantages of a row-based relational model stops. The document based model on the other side, consists of a dynamic schema and follows the principles of CAP - Consistency, Availability and Partition tolerance, giving some advantages where the relational model don't succeeds. The document model provides faster inserts, as transactions and atomicity is not a priority. The database is always available, and is better suited for hierarchical data storage. Nevertheless, this approach does not support joins, and the lack of a strict format can in some use cases be the reason why the document model is not the best option to go for.

One of the problems with building up the database using documents, is the bad support relations, especially many-to-many relationships. However, if I were to use a document model for modeling the case above, I would have had one document for each paper that is written. For each paper, have a key with a list of authors, with a list of their current papers and references to the other documents if created. This is a workable solution, but will include a lot of redundancy as the same information will be stored in multiple documents. A better solution is to use a graph model, which will be discussed in the next question.

Q9: When should you use a graph model instead of a document model? Explain why. Give an example of a typical problem that would benefit from using a graph model.

As seen in the last question, a case with many-to-many relationships can often be problematic for the document model. Take LinkedIn as an example. This social platform has a lot of documents, such as pages for educational institutions, companies, public figures and other workers. For all the individuals using LinkedIn, there is a web of connections between them and others based on education, work experience and voluntary work to name some. When you have connections and many-to-many relationships everywhere, as in the LinkedIn example, a graph model can be beneficial.

Q10: You have the following values for a column: 43 43 43 87 87 63 63 32 33 33 33 33 89 89 89 33

a)

column_encoding

values	43	43	43	87	87	63	63	32	33	33	33	33	89	89	89	33
Bitmap																
value == 32	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
value == 33	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	1
value == 43	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
value == 63	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
value == 87	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
value == 89	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

b)

Based on the bitmap above, we can create a run-length encoding. This is done by summarizing how many 0s there is, followed by how many 1s in a row - and continue until the last 1 is included.

$value == 32 : 7, 1$	(7 zeros, 1 one, rest zeros)
$value == 33 : 8, 4, 3, 1$	(8 zeros, 4 ones, 3 zeros, 1 one)
$value == 43 : 0, 3$	(0 zeros, 3 ones, rest zeros)
$value == 63 : 5, 2$	(5 zeros, 2 ones, rest zeros)
$value == 87 : 3, 2$	(3 zeros, 2 ones, rest zeros)
$value == 89 : 12, 3$	(12 zeros, 3 ones, rest zeros)

Q11: In case we need to do schema evolution, e.g., we add a new attribute to a Person structure: Labour union, which is a String. How is this supported by the different systems? How is forward and backward compatibility supported?

We have different binary formats / techniques for sending data across the network:

- ***MessagePack***
- ***Apache Thrift***
- ***Protocol Buffers***
- ***Avro***

Object: Add an attribute to a Person structure: Labour_union (string)
How is this supported and how is forward and backward compatibility supported?

As we want to add a new attribute, we have to take a closer look on the different structures. MessagePack, stating to be *like JSON, but fast and small*, does not have a schema and the format is therefore defined indirectly in the JSON-documents. Therefore, forward and backward compatibility is not a problem as new documents can have the attributes it wants, without giving errors in the database. The encoding/decoding won't be affected by this. How it is handled by the application using it, is however up to the developer either having to make it optional, or either giving it a default value or change all the former documents if compulsory.

When it comes to the three different techniques - Apache Thrift, Protocol Buffers and Avro - they all uses schemas. Therefore, they are dependent on all the versions of schemas to be stored for backward and forward compatibility. Thrift and Protocol Buffers have fields which are identified with tag numbers from schema, and can change or add new attributes that must be optional or have a default value. Avro, on the other hand, is interestingly different as it resolves differences automatically. Schema version number are included with each record, and large files have the schema stored with them.