TDT4225 Exercise 1

Hauk Aleksander Olaussen

September 14, 2021

1. **SSD: How does the Flash Translation Level (FTL) work in SSDs?**

The Flash Translation Level implements wear leveling to prolong the SSDs life and reduce latency. It accomplishes this by distributing the writes over the entire disk, and not only on the same blocks every time. This avoids erase-write cycles on these blocks, making them last longer than they otherwise would. We seperate between *dynamic* and *static* wear leveling. *Dynamic* wear leveling maps logical addresses to physical ones. When a block is about to be rewritten, the original block is marked as invalid, and the updated one is written to a new location. This works fine for dynamic blocks, but static blocks are still not affected by this. *Static* wear leveling introduces the same as *dynamic* wear leveling, but it also moves the static blocks around the disk periodically to reduce wear. FTL also runs a garbage collector which cleans the drive of block eligible for erase.

2. **SSD: Why are sequential writes important for performance on SSDs?**

Sequential writes are important because of the way the garbage collector works. When the disk is empty, there is no need for the garbage collector to run. Hence, random writes work practically just as well as sequential writes when the disk is empty. When the disk starts filling up however, the garbage collector needs to start doing its thing. Because of the way the sequential writes fills up blocks in a sequential order and random writes may cause internal fragmentation, the garbage collector will have to run a more complex job to clean up the blocks. This causes the performance of the disk to fall, and may reduce the lifetime of the disk.

There is a case, however, when the size of the writes are a multiple of the block size. In this case, random writes will work just as well as sequential writes because they are able to fill the blocks equally effective - removing internal fragmentation.

3. **SSD: Discuss the effect of alignment of blocks to SSD pages.**

Since an entire block would need to be rewritten for each write request, we might experience some overhead when when this request is written in a block already containing some data. This is however not a large problem when using sequential writes, as the block would fill up, and the garbage collector would not need to do additional work. With random writes however, this is not something we want, as it would not fill the block it writes to, leading to internal fragmentation (as we discussed in 2.). If the write requests are equal to or larger than the clustered block size however, blocks would fill up for every request, leading to this not being a problem. Write requests not a multiple of the block size would lead to more read-write operations, hence reducing performance.

Random writes aligned to clustered blocks will cause fragmentation between the blocks.

This is however not a problem for SSDs where the issue is the garbage collection, and not the head seeks as it would for HDDs. Therefore, random writes will perform just as sequential writes for SSDs when the writes are larger then the block size, but when the writes are smaller, sequential writes will outperform random writes.

4. **RocksDB: Describe the layout of MemTable and SSTable of RocksDB.**

MemTable is the C0 in-memory component from the LSM tree. It is a key-value-based store for read and write operations. When writing, the data will be written into the MemTable. When this becomes full, a new MemTable is generated and it becomes impossible to write to the old one. The old MemTable wil then be written into the SSTable before being deleted. Read with RocksDB will first try to find what we are looking for in the MemTable. If search proves unsuccessful, it will try the SSTable in the database. Write-ahead logs are used in case of failure, and will be deleted when the table is written to the disk.

5. **RocksDB: What happens during compaction in RocksDB?**

As RocksDB uses LSM-trees, is uses the rolling merge process - *compaction*. Compaction happens when two or more SSTables are merged into one immutable SSTable. If a key exists in multiple tables, the latest one is used for the new table. There are three styles of compaction: *Leveled* compaction stores SSTables in sorted runs. The runs are separated into levels, with the oldest data being in the last level. Keys can overlap in the first level, meaning that searching this level will cause the need to search all other levels as well - making the cost of searching L0 significantly higher than i.e. L4, as only L0 may have overlapping keys. Because of this, we do not want more SSTables at L0 than necessary. *Universal* compaction differs from *leveled* compaction in that they may overlap each others key-range, but does not overlap each others time-range. The resulting SSTables still have the last inputs in the last level of the sorted run. There are often many more levels (depth) in this type of compaction. *FIFO* compaction is the easiest one of the three, and it functions like a time-to-live cache. All the data is stored in the first level, and will be deleted after a set amount of time. The oldest table will be deleted when the when the database reaches its maximum size.

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

B+-trees usually has more write-amplification than LSM-trees. This will cause less performance when it comes to write requests. LSM-trees also usually has larger write requests. Combined with the sequential writes this will make performance better than B+-trees - and lead to better usage of memory because of the way larger write units are better for compaction. On the other hand, reads may be faster on B+-trees.

7. **Regarding fault tolerance, give a description of what hardware, software and human errors may be?**

Hardware errors are (as one would imagine) errors which occur on the hardware side of things. This can be anything from a faulty disk, fire in a data center, to power outage causing lack of power to the hardware. One way to mitigate the consequence of such errors are to introduce redundant hardware which will replace the faulty hardware if something unwanted occurs. *Software* errors are errors which occur on the sofware side of the system. These errors are often very hard to predict, and are therefore often dormant until they occur. This can be a bug where timezones are not taken into consideration in the program, causing times to be recorded wrong in different parts of the world. It can be that a dependency software crashes, causing errors in the main software. Ways to minimize the impact of such errors are to maintain and develop the software as the technology ages and letting the software crash to better prepare for future crashes. *Human* errors exist everywhere, as humans are the ones designing and developing the systems in use. Human errors might occur when the programmer introduces software bugs into a program - but does not write thoroughly enought tests to catch those bugs. Human errors occur on all levels of development.

8. **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?**

9. **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.**

10. **Column compression: 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) Create a bitmap for the values.

b) Create a runlength encoding for the values

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

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