A01 - Assignment 1

- **Deidier Simone** - *student number: 133020*

Theory questions

1. The **Flash Translation Layer (FTL)** is a fundamental component in **Solid State Drives (SSDs)** that manages the translation of logical addresses (those used by the operating system) into physical addresses (those actually present on the flash memory chips). The **FTL** maintains a *mapping table* that associates logical addresses with physical addresses. When the operating system requests data to be read or written, the **FTL** translates the logical address provided by the operating system into the corresponding physical address. Flash memory has a limited number of write/read cycles. The **FTL** implements *wear-leveling algorithms* to evenly distribute writes across all memory blocks, preventing some blocks from wearing out faster than others. When data is erased, it is not immediately removed from the flash memory. The **FTL** periodically performs *garbage collection operations* to reclaim free space by consolidating valid data and deleting unused blocks. In summary, the **FTL** is responsible for *optimising the performance, durability, and reliability* of **SSDs** by efficiently managing the underlying flash memory.

2. **SSDs** are designed to handle *sequential writes* better than *random writes*. Sequential writes allow large blocks of data to be written continuously, reducing the number of management operations required. Sequential writes reduce data fragmentation. Less fragmentation means that data is stored in contiguous blocks, facilitating successive read and write operations. With sequential writes, the **FTL** can handle *garbage collection* better, since the data is written in an orderly manner. This reduces the need to move data during clean-up operations, improving overall performance. Sequential writes also help with *wear leveling* by evenly distributing writes across all memory blocks. This helps prolong **SSD** life. Data written sequentially is easier to read efficiently. Sequential read operations can be performed faster than random reads.

3. When data blocks are correctly aligned to **SSD** pages, write operations can be performed more efficiently. Each write corresponds exactly to one or more pages, reducing the number of operations required. Correctly aligned data reduces fragmentation, as data blocks are stored contiguously. This facilitates subsequent read and write operations. With correct alignment, *garbage collection operations* are more efficient, as valid data can be consolidated more easily and unused blocks can be deleted without having to move a lot of data. Correct alignment helps with *wear leveling* by evenly distributing writes across all memory blocks. This helps prolong **SSD** life. The overall performance of the **SSD** improves with proper alignment, as read and write operations are faster and less prone to overhead.

4. **RocksDB** is a storage engine based on the **LSM (Log-Structured Merge) tree**, which uses two main data structures to manage data: **MemTable** and **SSTable**. The **MemTable** is a data structure in memory that stores recent writes. It is generally implemented as a skip list or a red-black tree to allow efficient insertions, updates and searches. When an application writes data, it is first inserted into the **MemTable**. Read operations search the **MemTable** first to find the most recent data. When the **MemTable** reaches a certain size, it is converted to an **SSTable** and written to disk. To ensure durability, each write operation is also recorded in a **Write-Ahead Log (WAL)** before being inserted into the **MemTable**. In the event of a crash, the **WAL** can be used to rebuild the **MemTable**. **SSTables** are immutable files stored on disk that contain a set of ordered keys and values. Each **SSTable** consists of several components: **Data Block**, **Index Block**, **Filter Block** and **Meta Block**. When the **MemTable** is downloaded to disk, it is converted into an **SSTable**. **SSTables** are immutable, which means that once written, they are never changed. Read operations first search the **MemTable** and then the **SSTables**, starting with the most recent **SSTable**.

5. **Compacting** in **RocksDB** is a crucial process for maintaining database performance and efficiency. **RocksDB** selects one or more **SSTables** to be compacted based on criteria such as size and level; the selected ones are locked to prevent changes during compaction. The selected **SSTables** are read into memory and the keys and values are sorted and merged, removing obsolete versions and deleted data. The sorted data are written into new **SSTables**; these new **SSTables** replace the old ones, which are then deleted. The **SSTable mapping table** is updated to reflect the new **SSTables**, the old **SSTables** are removed from the disk.

6. **LSM-trees** optimise *sequential writes*. Writes are first accumulated in a structure in memory (**MemTable**) and then written to disk in large, ordered blocks (**SSTable**). This reduces the number of random I/O operations, while **B+-trees** require random writes to maintain data order. Each insertion can cause nodes to split and changes to propagate upwards, resulting in many random I/O operations. **LSM-trees** use *compaction* to merge and sort data periodically. This process eliminates obsolete data and reduces fragmentation, improving the efficiency of successive reads and writes, **B+-trees** do not have a similar compaction mechanism. The structure can become fragmented over time, reducing the efficiency of read and write operations. *Sequential writes* of **LSM-trees** reduce disk wear, as data blocks are written contiguously. This is especially important for **SSDs**, which have a limited number of write cycles, while random writes of **B+-trees** can cause uneven disk wear, reducing the overall life of the device. **LSM-trees** are designed to optimise *write performance*, whereas the *write performance* of **B+-trees** can degrade with increasing data volume, due to frequent node splitting operations and the need to maintain data order. **LSM-trees** utilise memory efficiently by accumulating writes

in a memory structure (**MemTable**) before writing them to disk. This reduces the number of I/O operations and improves overall performance, **B+-trees** require more memory to maintain the tree structure and indexes, especially when the data volume increases.

7. **Fault tolerance** is crucial to ensure the *reliability and availability* of large-scale distributed data systems.

    - **Hardware failures**: Hard disk drives (**HDDs**) and solid-state drives (**SSDs**) can fail due to wear and tear, manufacturing defects or physical damage. A server can fail due to power problems, overheating, or failure of internal components such as **CPU**, **RAM**, or motherboards. Network connectivity problems may disrupt communication between nodes in the distributed system.
    - **Software Errors**: Programming errors can cause unexpected behaviour, crashes or data corruption. Incorrect configurations can lead to system malfunctions or security vulnerabilities. Software updates may introduce new bugs or incompatibilities.
    - **Human errors**: Operators may make mistakes while managing the system, such as deleting data by mistake or misconfiguring components. Users may expose the system to security risks, such as using weak passwords or sharing credentials. Inadequate planning can lead to system overloads, lack of resources or unplanned downtime.

8. To achieve **fault tolerance** in large-scale distributed data systems, it is necessary to implement a number of techniques and practices covering various aspects of the system. The main techniques are:

    - Copying data across multiple nodes or data centres to ensure that a copy is always available in the event of a failure.
    - Splitting data into smaller parts (*shards*) and distributing them across several nodes to balance the load and improve scalability.
    - Distributing data requests evenly across nodes to avoid overloads and improve performance.
    - Automatically switching to a backup node in the event of a failure of a primary node.
    - Create back-up copies of data and metadata so that they can be restored in the event of loss or corruption.
    - Continuously monitor system status and record significant events to detect and diagnose problems.
    - Manage and distribute system configurations in a centralised and controlled manner.
    - Perform regular testing to verify the system's ability to withstand failures.

9. The advantages of the **SQL (relational) model** are: **data integrity**, relationships between tables are well defined and maintained through primary and foreign keys, **ACID transactions**, it guarantees *Atomicity, Consis-*

tency, Isolation and Durability, making the system robust and reliable, **powerful queries**, in fact **SQL** offers a powerful and standardised query language for querying and manipulating data and **normalisation**, which reduces data redundancy and improves data integrity through normalisation. The main disadvantages are **scalability**, in fact it can be difficult to scale relational databases horizontally, **schema rigidity**, schema changes can be complex and time-consuming, and **performance**, in fact join operations can be costly in terms of performance, especially with large volumes of data. The main advantages of the **document model (NoSQL)** are **schema flexibility**, i.e. schemas can be easily modified without downtime, **horizontal scalability**, i.e. the model is designed to easily scale across multiple nodes, **performance**, i.e. reads and writes can be very fast, especially for denormalised data, and the fact that it is suitable for **hierarchical data**, ideal for representing hierarchical and nested data. The main disadvantages are the lack of **ACID transactions**, in fact not all document databases support full ACID transactions, **data redundancy**, i.e. the fact that denormalisation can lead to redundant data and inconsistency, and **limited queries**, in fact query capabilities can be less powerful than in SQL. In the example of a problem with many-to-many relations, the scenario is as follows: a document represents a paper, each paper has many sections and words, each paper has many authors and each author has a name and address and has written many papers. The main problems are **data redundancy**, i.e. if an author writes many papers, the author's information must be repeated in every paper of the paper, and **inconsistency**, i.e. if an author's address changes, it must be updated in every paper of the paper in which it appears. With a **relational model** we have several advantages, including the elimination of **redundancy**, i.e. author information is stored only once in the Authors table, and **consistency**, i.e. changes to author information need only be made once in the Authors table.

10. The ideal situations for the **graph model** are in the case of **complex and interconnected relationships**, when the data have many complex and interconnected relationships, a graph model is more suitable, in fact graphs naturally represent the relationships between entities via nodes and arcs, in the case of **deep relationship queries**, i.e. when queries require traversing many relationships to find answers, graphs are more efficient, in fact traversing operations are optimised in graph databases. In case of **network analysis**, i.e. to analyse complex networks such as transport networks, communication networks, or biological networks, graphs offer powerful tools for analysis, and finally **hierarchies and adjacent structures**, in fact when data represent hierarchies or adjacent structures, graphs can model these relationships in a more natural and flexible way. One scenario can be modelling a social network where users can have friends, follow other users, join groups, and share common interests, with a graph model the nodes represent users, groups, and interests, while the arcs represent the

relationships between the nodes, such as friendships, group membership, and common interests. The main advantages of the **graph model** are **efficient traversing**, i.e. finding all the friends of a friend (second-level query) is much more efficient in a graph, **dynamic relationships** (adding new relationships, such as following a user, is simple and does not require complex structural changes), and **community analysis**, i.e. identifying groups of users with common interests or analysing the centrality of users is more intuitive and efficient. Comparing it with a **document model**, where each user could be a document with nested fields for friends, groups, and interests, we can identify the following problems: **redundancy**, in fact information on friends must be repeated in each user document, **complex queries**, (queries that traverse many relationships, such as finding friends of friends, become complex and inefficient) and **difficult updates**, in fact modifying relationships requires multiple updates and can lead to inconsistency.

11. The choice between **textual and binary encodings** for sending data depends on various factors, including readability, compatibility, efficiency and system complexity. Some situations to use **textual encodings** are, for example, **human readability**, when it is important that the data be readable and understandable by humans, **debugging and maintenance**, i.e. during system development and maintenance, it is useful to be able to easily read and interpret the data, **compatibility and interoperability**, i.e. when data must be exchanged between different systems that may not share the same binary format, **standardisation** (when there are well-defined and widely adopted standards for representing data in textual format), **data size**, i.e. when the data is not too voluminous and the overhead of textual encoding is acceptable, and finally **simplicity of implementation**, e.g. using JSON to serialise objects in programming languages such as Java. Examples of **textual encodings** are in fact **JSON**, **XML** and **CSV**.

12. To create a **bitmap** for the values, we must first identify all unique values in the column and then create a bitmap for each value, in this case the unique values are: 32, 33, 43, 63, 87, 89. Bitmap for each value:

    - 32: 0000000010000000
    - 33: 0000000000111101
    - 43: 1110000000000000
    - 63: 0000110000000000
    - 87: 0000001100000000
    - 89: 0000000000000110

Run-length encoding represents data as value pairs and counts consecutive occurrences: - (43, 3) - (87, 2) - (63, 2) - (32, 1) - (33, 5) - (89, 3) - (33, 1)

13. When it comes to **schema evolution**, it is important to consider how each system handles the addition, removal or modification of attributes.

MessagePack does not have native support for schemas, in fact it is a binary serialisation format that focuses on compactness and speed. For **forward compatibility** the receiver can ignore unknown fields if the deserialiser is designed to do so, while for **backward compatibility** the deserialiser must be able to handle the lack of optional fields. **Apache Thrift** uses interface definition files (IDLs) to define schemas, IDLs can be updated to add new fields. For **forward compatibility**, added fields must have a unique identifier, older clients will ignore unknown fields, for **backward compatibility** fields can be added as optional, newer clients can handle the lack of these fields. **Protocol Buffers** uses .proto files to define patterns, which can be updated to add new fields. For **forward compatibility** the fields added must have a unique field number, older clients ignore unknown fields, while for **backward compatibility** the fields can be added as optional and newer clients can handle the lack of these fields. **Avro** uses JSON schemas to define data that can be updated to add new fields. For **forward compatibility** added fields must have a default value, older readers ignore unknown fields, while for **backward compatibility** fields can be added with default values, newer readers can handle the lack of these fields.

14. The ideal situations for **multi-leader replication** are those where **distributed writes** occur, i.e. when writes need to be performed in multiple data centres or different geographic regions, in fact multi-leader replication allows writes to be performed locally in each data centre, reducing latency and improving performance, in case of **high availability**, i.e. when high availability and fault tolerance is required, in fact, multi-leader replication provides redundancy, as each leader can accept writes (if one leader fails, the others can continue to operate) and in the case of **collaboration and conflicts**, i.e. when multiple users or applications need to collaborate and write conflicts may occur, in fact, multi-leader replication can handle write conflicts through application-side conflict resolution or through automatic resolution algorithms. Instead, the ideal situations for **leader-based replication** are in case of **strong consistency**, i.e. when strong consistency is required and reads must immediately reflect writes, in fact leader-based replication ensures that all writes go through a single leader, maintaining the order of operations and guaranteeing consistency, in case of **simplicity of implementation**, i.e. when simplicity of implementation and management is a priority, In fact, leader-based replication is easier to implement and manage than multi-leader replication, as it does not require the management of write conflicts, and finally in the case of **predominantly read workloads**, i.e. when the workload is mainly read and the writes are relatively few, precisely because leader-based replication can scale the reads by distributing them among the followers, while the leader manages the writes. The advantages of **log shipping** instead of replicating SQL statements are **consistency and order** (log shipping replicates changes at the transaction log level, ensuring that operations are

applied in the same order as they were executed on the leader), **efficiency**, in fact log shipping is generally more efficient in terms of space and time than replicating SQL statements, the support for **complex operations** as log shipping can handle complex operations such as multi-statement transactions and rollbacks, which can be difficult to replicate correctly with SQL statements, and the **reduction of the load on the leader**, in fact log shipping reduces the load on the leader as there is no need to re-execute SQL statements on followers.