TDT4225 – Exercise 1

Fredrik Sundt-Hansen

September 2024

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

To prolong the life of SSDs, where individual blocks will wear down based on how often they are written to, a firmware called "Flash Translation Layer" will try to distribute the writes over the different blocks. Avoiding erase-write cycles repeatedly on the same set of blocks.

Furthermore, FTL runs a garbage collection responsible for cleaning up old blocks that are considered eligible for erasing. Garbage collection consists of an algorithm deciding when to erase a block that consists of a percentage of valid and invalid blocks. However, the valid blocks must not be erased, but copied over to other free blocks before the old one is erased, causing overhead as this adds additional read and write operations and these operations generate write amplification.

2 Why are sequential writes important for performance on SSDs?

Sequential writes are important for performance on SSDs because they make more efficient use of the SSD's internal architecture. Unlike traditional hard drives, SSDs do not have physical moving parts, but they still rely on organizing data in memory cells. When writing data sequentially, the SSD can place data in contiguous blocks, reducing the need for erase operations and minimizing the use of garbage collection. Additionally, as long as write requests are smaller than the size of the clustered bloc, sequential writes perform better than random writes.

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

The effect of alignment of blocks comes down to how the write request's size is to the clustered page size. Write requests that are aligned to clustered pages

can be written to disk with no overhead. However, whenever the write request is larger than the page, it causes overhead. This is because the SSD needs to read the rest of the content in the last page and combine it with the updated data before all the data can be written to a new page.

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

RocksDB is a key-value store that uses an LSM-tree to organize its data. RocksDB consists of SSTables and Membtables. The former can be referred to as the components in the original LSM-tree. Memtables, on the other hand, is the in-memory component C_0, thus, all updated are first inserted into the Membtable. The Membtable handle fast in memory writes, while the SSTables ensure durability and efficient retrieval from disk,

5 RocksDB: What happens during compaction in RocksDB?

RocksDB compactions is essentially the equivalent of the rolling merge process in LSM-trees. Whenever a SSTable is full or has reached a certain threshold, compaction takes those two tables and merges their entries to obtain a new immutable SSTable with the new entries, where the ordering of the entries are defined by the comparator.

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 can suffer from fragmentation, where nodes allocate more space than necessary, leading to inefficient use of storage. Although B+-trees are optimized for low I/O operations, they still require immediate writes to disk for each insert, leading to frequent I/O operations for high insert volumes.

In contrast, LSM-trees are more efficient for large volumes of inserts due to their batched writes. In an LSM-tree, inserts are first accumulated in an in-memory structure, C_0, which amortizes the cost of I/O operations by batching multiple inserts together before writing them to disk. This reduces the number of individual disk writes and minimizes the I/O overhead associated with frequent inserts. Furthermore, LSM-trees delay disk writes until compaction, making them well-suited for high-throughput insert workloads.

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

Fault tolerance can be described as systems that anticipate faults and can cope with them. Where a fault is usually defines as one component of the system deviating from its spec.

7.1 Hardware Faults

Any causes of system failures that are triggered by physical hardware. Examples are hard disk crashes, faulty RAM, power grid blackout, etc. Furthermore, with large store clusters containing tens of thousands of disks, where each hard disk has a mean time to failure (MTTF) of about 10 to 50 years, a hard disk fault every day is expected. Therefore, measures for overcoming these challenges may include redundancy or software-tolerance techniques, or both.

7.2 Software Errors

Software errors can be classified as a fault that is a systematic error within the system. They can be difficult to anticipate, and due to them existing across nodes, they can cause more damage than hardware faults. Examples include software bugs, runaway process that uses up shared resources, a system that is being dependent on slows down, cascading failures where a small fault can trigger faults in other systems, etc. Unlike hardware faults, they lie dormant for a long time until triggered by random unusual events or circumstances.

7.3 Human Errors

Human errors are often the biggest factor in outages and system failures. Therefore, it is many practices to better teach and make us humans more reliable: designing system in a way that minimizes opportunities for error, thoroughly testing, quick and easy recovery from human errors, monitoring systems, management practices and training, etc.

8 Give an overview of tasks/techniques you may take/do to achieve fault tolerance

Many of these techniques were mentioned above.

Hardware faults: redundancy and software-tolerance techniques
Software Errors: carefully thinking of assumption and interactions in the system, thorough testing, monitoring and analyzing
Human errors: Thorough testing, monitoring systems, management practices

and training, designing system in a way that minimizes opportunities for error, quick and easy recovery from human errors

9 Compare SQL and the document model

9.1 SQL (Relational Databases)

Advantages:

- **Structured Data and Relationships**: SQL databases follow a strict schema and are highly structured, making them suitable for well-defined relationships and structured data.
- **ACID Compliance**: SQL databases provide strong guarantees for transactions (Atomicity, Consistency, Isolation, Durability), making them ideal for systems where data integrity is crucial.
- **Complex Queries**: SQL supports powerful queries, including complex joins, aggregations, and filtering, making it easy to retrieve related data from multiple tables.
- **Normalization**: SQL ensures minimal redundancy through normalization, which maintains data consistency and reduces storage costs.

Disadvantages:

- **Scalability**: SQL databases are harder to scale horizontally compared to NoSQL databases, as scaling often requires partitioning (sharding) or replication.
- **Rigid Schema**: SQL requires a predefined schema, making it less flexible when frequent schema changes are required. Altering schemas can be complicated and expensive.
- **Performance Overhead**: SQL databases often need joins between multiple tables, which can cause performance degradation as data grows, especially with many-to-many relationships.

9.2 Document Model (NoSQL Databases)

Advantages:

- **Schema Flexibility**: Document databases (e.g., MongoDB) store data in flexible, JSON-like documents, making it easier to adapt to schema changes without downtime.
- **Horizontal Scalability**: Document stores scale out horizontally with ease, making them well-suited for large-scale applications with massive datasets.

- **Embedded Relationships**: Data can be stored as embedded documents, reducing the need for joins and improving query performance when related data is stored together.
- **High Availability**: NoSQL databases often prioritize availability and partition tolerance, ensuring high availability in distributed environments.

Disadvantages:

- **Complex Queries**: While document databases offer flexibility, querying complex relationships (e.g., many-to-many) is more challenging compared to SQL and can result in performance issues.
- **Consistency**: Document databases prioritize availability over consistency, leading to potential issues with stale or inconsistent data.
- **Data Redundancy**: Without normalization, document models can introduce data duplication, making updates cumbersome and increasing the risk of errors.

subsectionExample: Many-to-Many Relationships in the Document Model

In this example, a paper has many sections and words, and it has many authors. Each author has multiple attributes (name, address) and can write multiple papers.

9.2.1 SQL Approach

In SQL, this is handled using normalized tables and many-to-many relationships with join tables:

```
-- Tables for normalized SQL model
CREATE TABLE Paper (
    paper_id INT PRIMARY KEY,
    title VARCHAR(255)
);

CREATE TABLE Author (
    author_id INT PRIMARY KEY,
    name VARCHAR(255),
    address VARCHAR(255)
);

CREATE TABLE PaperAuthor (
    paper_id INT,
    author_id INT,
    FOREIGN KEY (paper_id) REFERENCES Paper(paper_id),
    FOREIGN KEY (author_id) REFERENCES Author(author_id),
    PRIMARY KEY (paper_id, author_id)
```

```
);

CREATE TABLE Section (
    section_id INT PRIMARY KEY,
    paper_id INT,
    title VARCHAR(255),
    FOREIGN KEY (paper_id) REFERENCES Paper(paper_id)
);

CREATE TABLE Word (
    word_id INT PRIMARY KEY,
    section_id INT,
    word TEXT,
    FOREIGN KEY (section_id) REFERENCES Section(section_id)
);
```

9.2.2 Document Model Approach

In a document database like MongoDB, you would use an embedded document structure:

```
{
    "paper_id": 1,
    "title": "Introduction to AI",
    "sections": [
        {
            "section_id": 1,
            "title": "Section 1",
            "words": ["Artificial", "Intelligence", "is", "important"]
        },
        {
            "section_id": 2,
            "title": "Section 2",
            "words": ["Machine", "Learning", "is", "part", "of", "AI"]
        }
    ],
    "authors": [
        {
            "author_id": 1,
            "name": "John Doe",
            "address": "123 AI Street"
        },
        {
            "author_id": 2,
            "name": "Jane Smith",
            "address": "456 ML Avenue"
```

```
        }
    ]
}
```

Challenges in the Document Model:

- **Duplication**: If an author writes multiple papers, their name and address are duplicated across several paper documents, leading to potential inconsistency if the author's details change.
- **Query Complexity**: To find all papers written by an author, you need to query across multiple paper documents, making the query more complex than in SQL.
- **Update Complexity**: Changes in paper structure (e.g., sections or words) require rewriting large portions of the document, which can be inefficient.

9.3 Conclusion

SQL is ideal for structured data with complex relationships, providing strong consistency and powerful querying capabilities. The document model (NoSQL) offers flexibility and scalability but may introduce data duplication and complexity in querying many-to-many relationships.

10 When to Use a Graph Model Instead of a Document Model

A graph model should be used when the relationships between data points are complex and central to the problem being solved. While document models (e.g., NoSQL databases like MongoDB) excel at handling hierarchical or loosely-structured data, they are less efficient when working with data that involves many interconnected relationships. In contrast, graph models (e.g., Neo4j) represent data as nodes (entities) and edges (relationships) between those nodes, making them ideal for traversing and querying interconnected data.

10.1 When to Use a Graph Model

A graph model is beneficial in the following scenarios:

- **Complex Relationships**: When the relationships between data points are as important as the data itself. In scenarios where many-to-many relationships are common and frequently queried, a graph model allows efficient traversal of these connections.
- **Deep Traversal Queries**: When you need to frequently explore or traverse the connections between entities, such as finding the shortest path or navigating through several degrees of relationships.

- **Evolving Schema**: If the schema is likely to change over time, with more relationships or connections between entities, a graph model provides flexibility without the need for expensive schema migrations.

10.2 Example Problem: Social Networks

A typical example where a graph model would be advantageous is a social network. In a social network, users are represented as nodes, and their relationships (e.g., friendships, followers, likes) are represented as edges. A common query would be finding friends-of-friends or discovering communities within the network. These types of queries involve traversing several connections between entities and can become highly inefficient in a document model due to the need for multiple joins or complex query logic.

10.2.1 Graph Representation in a Social Network

In a graph model, the data might look like this:

```
(Node) User1 -- [FRIENDS_WITH] --> User2
(Node) User1 -- [FOLLOWS] --> User3
(Node) User2 -- [FRIENDS_WITH] --> User3
```

The graph model allows for efficient querying of relationships like:

- Finding all friends of User1
- Identifying the shortest path between two users
- Recommending new friends based on mutual connections

11 What are the situations that decide that you should use textual encodings instead of binary encodings for sending data?

There are several situations where textual encodings are preferable over binary encodings for transmitting data:

11.1 1. Human Readability

Situation: When the data being transmitted or stored needs to be easily readable and editable by humans.
Explanation: Textual encodings (such as JSON, XML, or plain text) are human-readable and allow for easier debugging, troubleshooting, and manual inspection. This is beneficial in use cases where system administrators or developers may need to inspect or modify the data directly.
Example: Configuration files (e.g., in JSON or XML format) or logs sent over the network, which may need to be manually read and edited.

11.2 2. Interoperability Across Platforms

Situation: When the data needs to be exchanged between systems with different architectures or software environments.
Explanation: Textual encodings are platform-independent and can be easily transferred between systems using different hardware architectures or programming languages. This avoids compatibility issues that binary encodings might introduce, such as byte-order (endianness) or word size mismatches.
Example: Web APIs that transfer data using formats like JSON or XML to allow interaction between clients and servers using different technologies.

11.3 3. Wide Adoption and Standardization

Situation: When leveraging widely accepted data exchange standards.
Explanation: Textual encodings such as JSON, XML, and CSV have become standard formats for data interchange over networks, especially in web services and APIs. These standards ensure broad compatibility with various libraries, frameworks, and tools.
Example: RESTful web services commonly use JSON or XML to transmit data because these formats are universally supported and expected by most systems.

11.4 4. Ease of Debugging

Situation: When developers need to frequently debug data transmission issues.
Explanation: Textual encodings provide an easier way to inspect the data in transit, making debugging more straightforward compared to binary encodings, where additional tools might be needed to decipher the contents.
Example: Debugging HTTP requests where payloads are transmitted in JSON or plain text makes it easier to track down issues or verify the transmitted data.

11.5 5. Less Sensitivity to Size and Performance

Situation: When the size of the data being transmitted is not a critical concern.
Explanation: Textual encodings generally require more space than binary encodings because they need to represent complex structures (e.g., numbers or images) using characters. However, when the size of the data or bandwidth is not an issue, textual encoding provides greater flexibility and convenience.
Example: Transmitting small or moderate-sized datasets between web services, where the overhead of text-based encodings is acceptable.

12 Column compression

Given the values for the column:

43 43 43 87 87 63 63 32 33 33 33 33 89 89 89 33

a) Bitmap Encoding

For the unique values {43, 87, 63, 32, 33, 89}, the bitmap encoding is as follows:

Value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
43	1	1	1	0	0	0	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
63	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
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
89	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

b) Run-Length Encoding

The run-length encoding for the values is:

$$(43, 3), \quad (87, 2), \quad (63, 2), \quad (32, 1), \quad (33, 4), \quad (89, 3), \quad (33, 1)$$

13 Schema Evolution and Compatibility in Different Binary Formats

We have various binary formats and techniques for sending data across the network:

- MessagePack
- Apache Thrift
- Protocol Buffers
- Avro

Suppose we need to add a new attribute, `Labour union`, to a `Person` structure, which is a string. Let's explore how schema evolution is supported and how forward and backward compatibility is handled in each system.

13.1 MessagePack

Schema Definition: MessagePack is schema-less, meaning there is no predefined structure for the data. Serialization and deserialization occur as key-value pairs in a binary format.

Schema Evolution:

- **Backward Compatibility**: Adding new fields (like `Labour union`) does not break existing systems. Older systems will ignore any extra fields they don't recognize.

- **Forward Compatibility**: Missing fields are treated as `null` or ignored, allowing new versions to deserialize older data without issues.

Example: Adding the `Labour union` field is straightforward—older systems ignore it, and newer systems can process it when available.

13.2 Apache Thrift

Schema Definition: Thrift uses IDL files to define the schema. Each field in the schema has a unique ID.

Schema Evolution:

- **Backward Compatibility**: New fields are added as optional with unique field IDs. Older systems will ignore fields they don't recognize.
- **Forward Compatibility**: Newer systems can read older data, and unrecognized fields are skipped during deserialization.

Example:

```
struct Person {
    1: string name,
    2: i32 age,
    3: optional string labour_union // new field
}
```

13.3 Protocol Buffers (Protobuf)

Schema Definition: Protobuf uses `.proto` files for defining message structures, where each field has a unique tag.

Schema Evolution:

- **Backward Compatibility**: New fields are optional with unique tags. Older systems skip unrecognized fields during deserialization.
- **Forward Compatibility**: Older data can be read by newer systems, with missing fields treated as optional.

Example:

```
message Person {
    string name = 1;
    int32 age = 2;
    string labour_union = 3; // new field
}
```

13.4 Avro

Schema Definition: Avro stores schema information with the data itself, allowing for dynamic interpretation.

Schema Evolution:

- **Backward Compatibility**: New fields can be added with default values. Older data without these fields will use the default.
- **Forward Compatibility**: Older systems will skip fields they do not recognize and continue functioning normally.

Example:

```
{
  "type": "record",
  "name": "Person",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "age", "type": "int"},
    {"name": "labour_union", "type": ["null", "string"], "default": null}
  ]
}
```

14 Replication Strategies

14.1 a) Multi-Leader Replication vs. Leader-Based Replication

Multi-Leader Replication should be used in cases where:

- There are **multiple data centers** that need to handle **writes** locally. Multi-leader replication allows each data center to act as a leader for its local writes, improving write availability and latency.
- Applications require **conflict resolution** strategies and can tolerate potential **write conflicts** since multiple leaders are allowed to accept writes simultaneously. This is typical in applications where users in different regions need to update the same dataset concurrently.

Multi-leader replication works well in environments where the system prioritizes write availability and low-latency operations, and the application can handle conflict resolution. However, this method introduces complexity due to potential conflicts, which must be resolved by the system or application.

Leader-Based Replication, on the other hand, is better suited when:

- You need **strict consistency**, and you want to ensure that **all writes** are processed in a single order.

- There is no need for **high write throughput** across geographically distributed data centers. With a single leader, the system guarantees consistency without dealing with conflicting updates.
- You want a simpler replication model where a single leader handles all writes, and replicas simply replicate changes.

Leader-based replication is ideal for systems that prioritize consistency and need a simpler model without the complexities of handling conflicts.

14.2 b) Log Shipping vs. SQL Statement Replication

Log Shipping should be used as a replication means instead of replicating SQL statements for several reasons:

- **Consistency**: Log shipping replicates the **exact changes** made to the data at the byte level, ensuring that replicas remain consistent with the leader. This avoids issues like nondeterministic SQL functions (e.g., `NOW()` or `RAND()`) producing different results on replicas.
- **Efficiency**: Log shipping can be more **space-efficient** as it only replicates the minimal data needed to make a change, whereas SQL statement replication can lead to the replication of entire SQL queries, which may involve complex parsing and execution on the replicas.
- **Performance**: Log shipping avoids the need to re-execute SQL statements on the replicas, which can improve replication performance, especially for complex or resource-intensive queries.

In contrast, **SQL Statement Replication** can lead to inconsistencies if queries contain nondeterministic elements or if they rely on database state (e.g., auto-incremented IDs) that may differ across replicas. Therefore, log shipping provides more reliable and consistent replication.