Air Travel Reservation Modeling Using MongoDB and Cassandra

Contents

Team Members

- 902011, Moleri Andrea, a.moleri@campus.unimib.it
- 865939, Armani Filippo, f.armani1@campus.unimib.it

Introduction

In the report you are about to read, we will explore NoSQL solutions and how they can be used to model real-world application problems. We will focus on MongoDB and Cassandra, two widely-used non-relational databases, and examine their features and differences. By doing so, we will demonstrate how each database can be utilized to model an airline reservation system.

The study will cover several important aspects, such as managing concurrent entries, designing various types of transactions, handling large volume read operations, and dealing with system malfunctions. Key database structures will be highlighted to give a clear understanding of their functionalities.

Additionally, we will analyze the entire data management process, from modeling to query execution. This comprehensive approach will provide a detailed look at how MongoDB and Cassandra perform in real-world scenarios, offering insights into their strengths and limitations in managing complex data systems.

Architecture

MongoDB Architecture

Key Terminology   In the realm of MongoDB, several key terms are essential to comprehend its architecture and functionality.

1. **Document:** the fundamental unit of data within MongoDB is referred to as a "Document". Each document encapsulates a set of key-value pairs representing a single entity. Unlike traditional relational databases, MongoDB's document model allows for flexible and dynamic schemas, enabling developers to store heterogeneous data structures within a collection.

2. **Collection:** a "Collection" in MongoDB is a grouping of documents. Documents within a collection typically share a similar structure, although MongoDB's flexible schema model permits variations in document structure within the same collection. Collections serve as logical containers for organizing related documents and are analogous to tables in relational databases.

3. **Database:** a "Database" in MongoDB serves as a container for collections. It provides a logical separation and management unit for organizing and accessing data. Multiple collections, each containing distinct sets of documents, can reside within a single database. MongoDB's architecture allows for the creation of multiple databases within a MongoDB deployment, facilitating data segregation and management at scale.

MongoDB and the Document Model   We will start by explaining the Document Model of MongoDB, which will be explained in greater detail it the 'Data Representation' chapter. In essence, we aim to comprehend how MongoDB stores data, and data are utilized. MongoDB serves as a General Purpose Document Database, structuring data in the form of Documents, akin to JSON Objects. This is very different from Relational Databases, which organize data into rows and columns within tables. Documents offer a flexible and developer-friendly approach to working with data. Consider the following code snippet as a simple example of a MongoDB Document:

```
{
    "_id": 1,
    "name": {
        "first": "Michael",
        "last": "Jackson"
    },
    "title": "Thriller",
```

```
    "interests": ["singing", "dancing"]
}
```

Documents correspond to Objects in code, rendering them intuitive to manage. This simplifies the planning of how Application Data correlates with data stored in the Database. Furthermore, Documents can be utilized in a highly developer-friendly manner to model data of any shape or structure: Key-Value Pairs, Text, Geospatial Data, Time-Series, Graph Data, and much more can be modeled using documents. The flexibility of documents allows us to employ a format for modeling and querying data for any application. MongoDB provides drivers in all major programming languages, facilitating the connection of a MongoDB Database to our application, regardless of the programming language used. Below is an example of a document used to store product data in a store. We can observe that the document has five fields, including a "colors" field containing an array of strings and an "available" field holding a boolean value.

```
{
    "_id": 1,
    "name": "iPhone 14 Pro Max",
    "colors" : ["space black", "silver", "gold", "deep purple"],
    "price" : 1500,
    "available" : true
}
```

While documents are presented in JSON format, they are stored in the database in a format called BSON, which stands for Binary JSON, an extension of JSON providing additional features that MongoDB can leverage. BSON also adds support for additional data types that would not otherwise be present in standard JSON. Thanks to this choice, the database can support a vast range of data types, including all JSON data types (strings, objects, arrays, booleans, null), as well as dates, numbers, Object IDs, and more. Particularly, ObjectID is a special data type used in MongoDB to create unique identifiers. In the database, each document requires a field "_id" serving as the primary key. If a document does not include this field, MongoDB will automatically add it, generating a specific ObjectID value for the document in question. By default, MongoDB supports a flexible schema model and polymorphic data. This enables us to store documents with different structures in the same collection. Documents can contain various fields, and the fields can hold different data types from one document to another. This is a significant distinction from relational databases, where declaring a table schema is necessary before inserting data. MongoDB's flexible schema allows us to iterate rapidly and evolve as our requirements change. Here's a practical example of this flexible schema. Suppose we have an online tech store with a catalog of items. When we start developing our application, we decide to include an "_id", a "name", and a "price" for each item.

```
{
    "_id": ObjectId("abcdef"),
    "name": "iPhone",
    "price": 1500.00
}
```

In the future, we might decide to add another field, such as a "description" field. With a relational database, we would encounter a complicated chain of dependencies to modify, risking downtime and significant time losses, even for a simple change like this. Instead, to achieve the same with MongoDB, we simply update the classes to include the new fields, and we can start inserting new documents with the new schema. This is facilitated by MongoDB's flexible schema model, which means that documents in the same collection are not required to share a common structure of fields and value types by default.

```
{
```

```
    "_id": ObjectId("abcdef"),
    "name": "iPhone",
    "price": 1500.00,
    "description": "the all new iPhone!"
}
```

Should we desire more control over the structure and contents of the database, we can add optional Schema Validation Rule to impose constraints on the structure of documents in the collection. Nonetheless, the basic syntax of the Document Model remains as indicated in the following code:

```
{
    "key": "value",
    "key": "value",
    "key": "value"
}
```

Nodes in MongoDB   A MongoDB database is physically structured to support high availability, scalability, and resiliency. The physical structure of MongoDB involves various components such as nodes, clusters, and datacenters. Below is an overview of how these components interact and how the reading and writing processes occur. Let's start by saying that a running MongoDB instance is called a "node". There are two main types of nodes:

1. **Primary Nodes:** they are responsible for write operations. Each replica set has only one primary node at a time.
2. **Secondary Nodes:** receive copies of data from the primary node and can serve read operations if configured to do so.

Clusters in MongoDB   MongoDB can also be configured as a cluster, a powerful setup that involves multiple servers or nodes working together. A cluster ensures data availability and reliability by distributing data and tasks across multiple nodes. This architecture provides several key benefits:

1. **Scalability:** clusters can handle increasing amounts of data by adding more nodes, allowing horizontal scaling.
2. **High Availability:** by distributing data across multiple nodes, clusters ensure that the system remains operational even if one or more nodes fail.
3. **Load Balancing:** tasks and queries are distributed across the nodes, balancing the load and improving performance.

Clusters can be easily set up and managed by the programmer, through the command line or by using Atlas user interface. Clusters can be deployed across multiple cloud providers and regions, allowing for global distribution of data and applications.

Sharding in MongoDB   Sharding is a technique used to distribute data across multiple machines. In MongoDB, sharding involves partitioning data into smaller, more manageable pieces called shards. Each shard is a subset of the data and can reside on a separate node within the cluster. This method enables horizontal scaling, making it possible to handle very large datasets and high-throughput operations efficiently. Key concepts in MongoDB sharding include:

1. **Sharded Clusters:** composed of multiple shards, each storing a portion of the data. A sharded cluster includes three main components: shards, query routers, and config servers.
   - **Shards:** store the actual data. Each shard can be a replica set to provide high availability and data redundancy.
   - **Query Routers (mongos):** route client requests to the appropriate shard(s). They handle the distribution of queries and aggregation of results.

- **Config Servers:** maintain metadata and configuration settings for the cluster. They store information about the sharding structure and the location of data.
2. **Shard Keys:** a shard key is a field or a combination of fields that determines how data is distributed across shards. Choosing an appropriate shard key is crucial for balanced data distribution and query performance.

In MongoDB, sharding can be enabled and configured through the console, providing an easy-to-use interface for managing sharded clusters.

Replica Sets in MongoDB   A replica set in MongoDB is a group of MongoDB instances that maintain the same dataset. Replica sets provide redundancy and high availability, ensuring that data is replicated across multiple nodes. Key features and benefits of replica sets include:

1. **Data Redundancy:** data is replicated to multiple nodes, protecting against data loss in case of hardware failure or other issues.
2. **Automatic Failover:** if the primary node fails, an eligible secondary node is automatically elected as the new primary, ensuring continuous availability.
3. **Read Scalability:** read operations can be distributed across multiple nodes, improving read performance and balancing the load.

A typical replica set consists of:

- **Primary Node:** handles all write operations and coordinates replication to secondary nodes.
- **Secondary Nodes:** maintain copies of the data from the primary node. They can be configured to handle read operations, providing load balancing and improved read performance.
- **Arbiter Nodes:** participate in elections but do not store data. They are used to ensure a quorum in elections when there are an even number of data-bearing nodes.

In MongoDB Atlas and in the command line, deploying a replica set is straightforward, and the Atlas platform provides tools for managing and monitoring the health of the replica set. Automatic backups, point-in-time recovery, and monitoring tools are available to ensure the reliability and performance of the replica set.

Data Centers and Information Exchange   In a distributed architecture, nodes can be distributed across multiple data centers to ensure geographic resilience and fault tolerance. This configuration improves database availability in the event of natural disasters or regional outages. Let's now see an overview of how information exchange works in MongoDB:

- **Replica Set:** secondary nodes continuously synchronize data from the primary node over a network connection. They use the oplog (operation log) based replication protocol, a log of operations applied to the primary database.
- **Sharded Clusters:** query routers (mongos) receive requests from the client, consult config servers to determine which shard contains the requested data, and then forward the requests to the appropriate shard.

Reading and Writing Processes   Write operations are sent to the primary node of a replica set. In the case of a sharded cluster, the write request is forwarded to the query router (mongos), which routes it to the appropriate shard. The primary node confirms the write and propagates it to the secondary nodes. On the other hand, read operations can be served by the primary node or secondary nodes, depending on the replica set configuration. In a sharded cluster, the query router (mongos) directs the read request to the appropriate shard that contains the requested data.

In conclusion, MongoDB is designed to manage data distributed across nodes, replica sets, and sharded clusters, ensuring high availability, scalability, and resilience through efficient synchronization and dynamic management of read and write operations.

Cassandra Architecture

Apache Cassandra is a distributed, open-source NoSQL database designed to handle large volumes of data distributed across many servers. Originally created by Facebook to improve the management of enormous amounts of data, Cassandra has become one of the most widely used databases for applications requiring high availability, scalability, and high performance.

Key Components

Node   A node in Cassandra represents any system running the Cassandra software. This system can be a physical host, a cloud instance, or a Docker container. Nodes work together to form a cluster, sharing the responsibility of storing and managing data, responding to queries, and ensuring the overall availability of the system.

Rack   A rack is a group of closely located nodes, either on the same network or within the same cloud availability zone.

Data Center   A data center is a collection of racks connected by a reliable network, corresponding to a physical building or a cloud region. Data centers allow physical and logical separation of nodes to ensure data remains accessible even in case of extensive failures. Cassandra stores replicas of data across different data centers to mitigate potential data loss from significant events like fires or earthquakes. This ensures that if a data center experiences issues, users might experience slower data reads, but data will not be lost.

Cluster   The set of nodes that defines the database, regardless of the number of racks and data centers, is called a cluster. A cluster is the overall collection of all nodes working together to provide a distributed database. Configuring a cluster determines how data is distributed and replicated among nodes.

Ring   To logically organize components, the concept of a ring is often used. The ring consists of nodes within the same data center, each assigned a range of token values. Typically, two adjacent nodes along the ring do not belong to the same rack. The token range assigned to each node ensures that the endpoint of one range is adjacent to the start of the range of the next node. This logical structure is useful as Cassandra often employs hash functions, ensuring a consistent mapping between tokens and nodes.

Internal Protocols   Cassandra uses internal protocols to ensure the system operates efficiently and that data is correctly distributed and replicated throughout the cluster.

Gossip Protocol   The Gossip Protocol allows each node to monitor the state of other nodes in the cluster. A "Gossiper" in each Cassandra instance randomly selects up to three nodes every second to initiate a gossip session. During these sessions, nodes exchange information about their states, allowing all nodes to quickly learn the overall state of the cluster. Using this information, Cassandra determines if a node is active or inactive, aiding in optimal request routing within the cluster.

Snitch Component   The Snitch component informs each node about its proximity to other nodes. This information is crucial for deciding from which nodes to read and to which nodes to write, and for efficiently distributing replicas to maximize availability in case of node, rack, or data center failures. The Snitch can be configured to optimize reads and writes based on latency and network topology, improving the system's overall performance.

Sharding   Sharding is fundamental in Cassandra for distributing and balancing data among the cluster nodes. It involves dividing a large dataset into smaller shards, distributed across different nodes. Each node manages a specific range of data determined by the token range. When data is written to the cluster, unique identifiers, also known as partition keys, are converted into a 64-bit token via a hash function. This token determines the node responsible for storing the data. The entire token space is divided into ranges assigned to nodes. When a node receives a write request, it uses the token to identify the correct range and store the data.

Sharding offers several advantages: it distributes the load across multiple nodes, improving performance and scalability. Each node manages only a portion of the data, reducing the workload for each node. Additionally, it simplifies cluster capacity management as nodes can be added or removed without restructuring the entire database. Cassandra automatically performs load balancing, reassigning token ranges when nodes are added or removed. This process ensures fair load distribution among nodes and balanced data and workload distribution in the cluster.

Node-Level Architecture   The architecture of a Cassandra node is structured according to the following hierarchical layout:

Cassandra Daemon (JVM)   Cassandra is written in Java, so each node runs a JVM that interprets Java code. The JVM manages the execution of the Cassandra software, including all read, write, and data management processes. The JVM also provides an optimized runtime environment for performance, enabling Cassandra to handle high workloads and scale efficiently.

Memtables   Memtables are in-memory data structures used to temporarily store write operations before they are written to disk. Memtables provide a fast storage area for written data, allowing Cassandra to respond quickly to write requests and temporarily store data before transferring it to disk for persistence.

Key Caches   The key cache stores the locations of row keys within the SSTables. This allows Cassandra to quickly locate data on disk without having to read index files for each read operation. The key cache improves read performance by reducing the time required to find the requested data on disk.

Row Caches   Row caches are data structures that store frequently accessed rows to further improve read performance. The row cache allows Cassandra to quickly respond to repetitive read requests for the same data, reducing the need to access the disk and improving the overall performance of the system.

Disk

Commit Logs   Commit logs are files that ensure data durability and consistency. Every write operation is first recorded in the commit log before being applied to memory. These logs are crucial for data recovery in case of a failure. Key information stored in the commit log includes:

- **Timestamp:** the date and time of the operation.
- **Keyspace:** the database to which the write operation belongs.
- **Table:** the affected table.
- **Partition Key:** the partition key.
- **Row Key:** the row key.
- **Column Values:** the written or updated column values.
- **Operation Type:** the type of operation (INSERT, UPDATE, DELETE).
- **TTL (Time to Live):** the data's duration, if applicable.
- **Consistency Level:** the required consistency level.
- **Checksum:** to ensure data integrity.

SSTables SSTables (Sorted Strings Tables) are immutable disk files where memtable data is periodically written for persistence. SSTables are structured to allow fast access to stored data, improving the performance of read and write operations. SSTables are created when data in the memtables reaches a certain size and must be transferred to disk for long-term storage.

Hints Hints are a mechanism used to improve data availability and fault tolerance. When a destination node is unavailable to receive a write, Cassandra records a hint on the source node. This hint contains information similar to that in the commit log, necessary to apply the write once the destination node becomes available. The most important information is the identifier of the node that could not receive the write and the data to be written. Hints are crucial to ensure that data is correctly written as soon as the destination node comes back online.

Consistency

Coordinator When a user connects to the database, they are assigned a coordinator node. This node is responsible for coordinating interactions with other nodes, sending requests, gathering results, and returning the query result to the client. The coordinator also ensures, through the sharding process, that requests are correctly distributed among nodes containing copies of data, avoiding the interrogation of the same set of nodes, and that query results are accurate and complete.

Consistency Levels Since, as previously mentioned, Cassandra redundantly stores data on multiple nodes known as replica nodes, based on a replication factor defined during database configuration, ensuring total consistency among the various copies can take time due to many write operations. However, this might not always be necessary. Cassandra offers a feature that allows users to manage the balance between data consistency and time. Consistency can be managed globally or selectively for individual read and write operations. Users can set data consistency preferences using the CONSISTENCY command or through client drivers. Consistency refers to the number of replicas that must be involved. The parameter can be specified as a numerical value or with keywords like ALL, ANY, and QUORUM $\left(\frac{|replica\_nodes|}{2} + 1 \right)$. These options provide application developers with the flexibility to balance data availability, consistency, and application performance.

Data Writing The fundamental elements in the write-read process are partition keys and clustering keys, as these regulate which node the data is on and how data is grouped and organized within each partition. When the coordinator node receives a write request, it determines which nodes in the cluster are responsible for replicating the data for the specified key in the write request (based on configurations: not necessarily all nodes). The coordinator node determines

the replica node by taking the partition key of the data, applying hash functions, also known as partition functions, from which it obtains a 64-bit token identifying the node where the data is stored (sharding). The coordinator node then forwards the write request to these replica nodes. The replica node then executes the following steps:

1. **Commit Logs:** when data is written to a node, it is initially stored in a commit log, ensuring recovery of the write in case of node failure.
2. **Memtable:** a copy of the data is then stored in the memtable, making it accessible for future reads or updates without accessing the disk.
3. **Row Cache:** if the row cache is active and contains an earlier version of the row, it is invalidated and replaced with the new data.
4. **SSTables:** cassandra monitors the size of the memtable. When it reaches a predefined threshold, the data is written to SSTables (on disk). When several columns of the same row are updated in separate write operations, multiple SSTable files are created.
5. **Hinted Handoff:** if the coordinator detects that a node is unresponsive during a write, it sends hints to the nearest node of the destination node, which temporarily stores the data until the target node is back online.

Data Reading   When the coordinator node receives a read request, it determines the replica nodes responsible for the specified partition key. The read request is then sent to the appropriate replica nodes. The coordinator node can use several consistency levels to determine how many replicas should respond before returning the data. The following steps outline the process:

1. **Row Cache:** the read operation starts by checking the row cache if enabled. The cache is used to store frequently accessed rows and provides the fastest read option.
2. **Memtable:** if the requested data is not in the row cache, the memtable is checked. The memtable contains recent write operations and ensures the latest data is accessible without disk access.
3. **SSTables:** if the data is not in the memtable, Cassandra looks for it in the SSTables. SSTables are immutable and stored on disk, providing persistent storage for data.
4. **Bloom Filter:** to avoid unnecessary disk access, a Bloom filter is used to determine the likelihood that the requested data is in a specific SSTable.
5. **Key Cache:** the key cache is checked to locate the row within the SSTable quickly.
6. **Merge Results:** the data is read from all relevant SSTables and merged to provide the final result. This process ensures that the most recent and consistent data is returned to the client.

Cassandra's architecture ensures high availability, fault tolerance, and scalability through its distributed nature, replication strategies, and efficient data management protocols. It allows users to manage consistency and availability trade-offs, making it suitable for various use cases requiring fast and reliable data access.

Data Representation

MongoDB Data Representation

Data representation is a fundamental aspect of database design, as it serves the purpose of the structuring of data storage and the delineation of relationships among various entities within the data. It serves as a blueprint for organizing information within a database. We refer to the organization of data inside a database as a 'Schema'. When data representation with MongoDB, it is advisable to conceptualize the application itself: its functionalities, the data it will handle, user data access patterns, and the data elements critical to the project's objectives. Addressing these questions aids in understanding the form of the data, its relationships, and the necessary tools

for implementation. A robust data model offers several benefits, facilitating data management, enhancing query efficiency, optimizing resource usage, and reducing operational costs. As a guiding principle within MongoDB, the mantra *"data that is accessed together, should be stored together"* underscores the great importance of structuring data in a manner conducive to operational efficiency. MongoDB employs a flexible document data model, in which collections do not impose any default document structure. As a consequence, documents may exhibit diverse structures, thanks to a concept called polymorphism, as exemplified below:

Document I

```
{
    "name": "Andrea"
    "major": "CS"
    "course": "Architetture Dati"
    "amount": 1000
    "tuition_payment": True
}
```

Document II

```
{
    "name": "Filippo"
    "major": "CS"
    "course": "Qualita del Software"
    "year": 2024
}
```

It is important to clarify that while MongoDB's Document Model is flexible, it is not entirely schema-less but rather schema-flexible. This flexibility extends to employing Schema Validation and accommodating diverse data types within MongoDB. Additionally, MongoDB supports nested or embedded documents, enabling the construction of complex data relationships. Normalization of data is achievable through database references. The complication lies in aligning data representation decisions with application requirements, contrasting with the traditional approach of representing data in relational databases.

Unlike the standard procedure of gathering data requirements, modeling data, and then handing over the data to developers, MongoDB's methodology commences with understanding application requirements, user interactions, and subsequently tailoring data representation accordingly. MongoDB's versatility enables various data storage approaches, including normalization, embedding related data for cohesive access, or employing hybrid methods as dictated by application needs. The final goal of data representation is to optimize storage, querying, and resource utilization, enhancing application performance and reducing database costs as a consequence. Once the foundational data representation framework is established, attention can be directed towards representing data relationships. A well-crafted data model simplifies data management, enhances query efficiency, minimizes resource consumption, and mitigates database operational costs.

Types of Data Relationships When discussing data relationships, it is crucial to delineate between various types: One-To-One, One-To-Many, and Many-To-Many. Additionally, we will dive into the two primary methods for representing these relationships: Embedding and Referencing. As we already said, it is important to structure our data to align with the querying and updating patterns of our application. In that regard, understanding common relationship types in databases is extremely important.

One-To-One Relationship The One-To-One relationship is characterized by one Data Entity in a Set being connected to precisely one Data Entity in another set. In traditional relational databases,

this relationship might be implemented using a JOIN operation. In MongoDB, a One-To-One Relationship can be represented succinctly within a single document, as exemplified below. In the example, a document representing a film encompasses not only the title but also the director's information.

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Her",
    "director": "Spike Jonze",
    "runtime": 126
}
```

One-To-Many Relationship   The One-To-Many relationship is characterized by one Data Entity in a set being associated with multiple Data Entities in another set. For instance, a film may feature several cast members. MongoDB facilitates the representation of this relationship within a single document using features like Nested Arrays, which are advantageous for representing One-To-Many Relationships. The "cast" field in the code shown below exemplifies such a structure.

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Her",
    "director": "Spike Jonze",
    "runtime": 126,
    "cast": [
        {"actor": "Joaquin Phoenix", "character": "Theodore"},
        {"actor": "Scarlett Johansson", "character": "Samantha"},
        {"actor": "Rooney Mara", "character": "Catherine"}
    ]
}
```

Many-To-Many Relationship   The Many-To-Many relationship represents a scenario where any number of Data Entities in one set are connected to any number of Data Entities in another set. As previously mentioned, the primary methods for representing relationships in MongoDB are Embedding and Referencing. Embedding involves incorporating related data within the document, while Referencing entails referring to documents in another collection within the document. The following examples illustrate Embedding and Referencing respectively. In the Embedding example, Actor documents are embedded within Movie documents using Nested Arrays. On the other hand, in the Referencing example, Filming Locations are referenced inside the document via their respective ObjectIDs.

Embedding

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
    "title": "Her",
    "director": "Spike Jonze",
    "runtime": 126,
    "cast": [
        {"actor": "Joaquin Phoenix", "character": "Theodore"},
        {"actor": "Scarlett Johansson", "character": "Samantha"},
        {"actor": "Rooney Mara", "character": "Catherine"}
    ]
}
```

Referencing

```
{
    "_id": ObjectId("573a1390f29313caabcd413b"),
```

```
    "title": "Her",
    "director": "Spike Jonze",
    "runtime": 126,
    "filming_locations": [
        ObjectID("654a1420f29313fggbcd718"),
        ObjectID("654a1420f29313fggbcd719")
    ]
}
```

Representing Data Relationships   In this section, we provide an example of data representation based on a practical scenario illustrated in the code below. When a student enrolls at a university, they fill out a form on a web application that creates their profile, which is then stored in a database. Upon examining the following code, there emerges a need to gather more information about the student, such as the courses taken and their grades. Furthermore, certain aspects of the code are not optimally structured.

```
{
    "student": "Andrea Moleri",
    "student_id": "902011",
    "age": "23",
    "home_phone": "2125550000",
    "cell_phone": "2125550001",
    "email": "andreamoleri@gmail.com",
    "grade_level": "master's degree",
    "street": "Viale della Vittoria 6",
    "city": "Milano",
    "state": "MI",
    "zip": "12345",
    "emergency_contact_name": "Filippo Armani",
    "emergency_contact_number": "212550002",
    "emergency_contact_relation": "Friend"
}
```

An initial observation reveals the presence of three phone numbers at different locations within the code, resulting in not-so-clean code. To address this issue, reorganization is proposed instead of treating them as separate elements indicating a One-To-One Relationship. This reorganization involves transforming it into a One-To-Many Relationship through the use of a Nested Array.

```
{
    "student": "Andrea Moleri",
    "student_id": "902011",
    "age": "23",
    "email": "andreamoleri@gmail.com",
    "grade_level": "master's degree",
    "street": "Viale della Vittoria 6",
    "city": "Milano",
    "state": "MI",
    "zip": "12345",
    "emergency_contact_name": "Filippo Armani",
    "emergency_contact_relation": "Friend",
    "contact_number": [
        {"number": "2125550000", "type": "home"},
        {"number": "2125550001", "type": "cell"},
        {"number": "212550002", "type": "emergency"}
    ]
}
```

In the scenario where additional data regarding the student is available, such as considering the courses taken along with their respective grades, a different data representation approach may

be considered. Here, references to Course ID and Course Name are added within the Student Document.

```
{
    "student": "Andrea Moleri",
    "student_id": "902011",
    "age": "23",
    "contact_number": [
        {"number": "2125550000", "type": "home"},
        {"number": "2125550001", "type": "cell"},
        {"number": "212550002", "type": "emergency"}
    ],
    "email": "andreamoleri@gmail.com",
    "grade_level": "master's degree",
    "gpa": "4.0",
    "street": "Viale della Vittoria 6",
    "city": "Milano",
    "state": "MI",
    "zip": "12345",
    "emergency_contact_name": "Filippo",
    "emergency_contact_relation": "Friend",
    "courses": [
        {"course_id": "2324-1-F1801Q159", "course_name": "Architetture Dati"},
        {"course_id": "2324-1-F1801Q115", "course_name": "Qualita del Software"}
    ]
}
```

Additionally, a dedicated Collection for Courses can be established, wherein the courses inserted within the Student Document are represented in a separate document as demonstrated below. In the provided data representation scenario, the Student Document represents individual student profiles, containing comprehensive information such as student details, contact information, emergency contacts, enrolled courses, and other relevant data. Within the Student Document, a nested array named "courses" is included, which holds references to the courses taken by the student. Each course reference consists of a CourseID and Course Name.

On the other hand, the separate Course Collection stores detailed information about all available courses offered by the university. Each document within the Course Collection represents a distinct course, featuring attributes like Course ID, Course Name, Professor, and Offered Term(s). The interconnection between these two pieces of code lies in the referencing mechanism established within the Student Document. When a student enrolls in a course, instead of duplicating course information within each student profile, a reference to the corresponding course is included within the "courses" array. This reference includes the Course ID, allowing easy retrieval of detailed course information from the Course Collection when and if needed.

```
"courses": [
    {
        "course_id": "2324-1-F1801Q159",
        "course_name": "Architetture Dati",
        "professors": "Andrea Maurino, Marco Cremaschi, Fabio d'Adda",
        "offered": "Spring, Summer, Fall, Winter"
    },

    {
        "course_id": "2324-1-F1801Q115",
        "course_name": "Qualita del Software",
        "professors": "Giovanni Denaro, Luca Guglielmo, Elson Kurian",
        "offered": "Fall, Spring"
    }
]
```

Embedding Data in Documents In the realm of database management, understanding how to model data using embedding is really important. Embedding is frequently employed in scenarios involving One-To-Many or Many-To-Many Relationships within stored data. MongoDB's documentation advocates for embedding to streamline queries and enhance performance. Embedded Documents are also known as Nested Documents, that is documents that encapsulate another document within them.

To better understand this concept, let us consider the following document, which contains two embedded sub-documents for both name and address. The client possesses only one name, embedded as First and Last Name. Regarding addresses, the client has three addresses, constituting a One-To-Many Relationship. Documents structured in this manner facilitate the retrieval of complete address information for a client, aligning with the principle "data that is accessed together should be stored together." Embedding enables the consolidation of various related pieces of information into a single document, potentially simplifying and reducing the number of required queries. One-To-One Relationships and One-To-Many Relationships are the relationships that are most commonly utilizing embedding.

```
{
    "name": {"firstName": "Walt", "lastName": "Disney"},
    "job": "entrepreneur",
    "address": {
        "europeanAddress": {
            "street": "Bd de Parc, Coupvray",
            "city": "Coupvray (FR)",
            "zipcode": "77700"
        },
        "americanAddress": {
            "street": "Epcot Center Dr, Lake Buena Vista",
            "city": "Florida (USA)",
            "zipcode": "32830"
        },
        "asianAddress": {
            "name": "Tokyo DisneySea",
            "street": "1-13 Maihama, Urayasu",
            "city": "Chiba (JP)",
            "zipcode": "279-0031",
            "country": "Japan"
        }
    }
}
```

Incorporating embedding mitigates the necessity of application joins, thereby minimizing queries and enhancing read operation performance. Furthermore, it enables developers to update related data in a single write operation. However, employing Embedded Data Models entails certain risks as swell. Primarily, embedding data within a single document can lead to excessively large documents, potentially causing latency issues. Large documents must be entirely read into memory, which may result in reduced performance for end-users. Additionally, during the embedding process, there's a risk of inadvertently structuring documents in a manner where data is continually added without restraint, leading to Unbounded Documents. These documents pose a risk of exceeding the maximum BSON document threshold of 16MB. Both Large Documents and Unbounded Documents are recognized as Schema Anti-Patterns, and as such, they should be avoided.

Referencing Data in Documents There may be scenarios where it becomes necessary to store related information in separate documents or even in distinct collections. When there is a need to store data across different collections while ensuring clarity regarding their relational nature, References come into play. Working with references is simple, and it is only a matter of saving

the identifier field of one document within another document to establish a link between the two. The utilization of references is often referred to as Linking or Data Normalization. Let's revisit the example previously discussed, wherein we have a university student who has taken various university courses. In the following code snippet, the `course_id` serves as our reference. Referencing enables us to circumvent data duplication, leading to smaller documents. However, this approach may necessitate querying multiple documents, potentially incurring higher read times and costs.

```
{
    "student": "Andrea Moleri",
    "student_id": "902011",
    "age": "23",
    "contact_number": [
        {"number": "2125550000", "type": "home"},
        {"number": "2125550001", "type": "cell"},
        {"number": "212550002", "type": "emergency"}
    ],
    "email": "andreamoleri@gmail.com",
    "grade_level": "master's degree",
    "gpa": "4.0",
    "street": "Viale della Vittoria 6",
    "city": "Milano",
    "state": "MI",
    "zip": "12345",
    "emergency_contact_name": "Filippo",
    "emergency_contact_relation": "Friend",
    "courses": [
        {"course_id": "2324-1-F1801Q159", "course_name": "Architetture Dati"},
        {"course_id": "2324-1-F1801Q115", "course_name": "Qualita del Software"}
    ]
}
```

To summarize the advantages and disadvantages, we employ embedding when we want to use a single query to retrieve data and when performing individual operations for data updates or deletions. However, this approach carries the risk of data duplication and the creation of substantial documents. Regarding referencing, this technique enables us to avoid duplicates, resulting in smaller and more manageable documents. However, this technique introduces the need of data joins from disparate documents. Another realistic example illustrates the utilization of referencing, where `user_id` in the first collection acts as a reference to a document in the `users` collection, thereby establishing a linkage between the two documents through referencing.

Collection I

```
{
    "author": "Mr.MongoDB",
    "title": "Learn The Basics of MongoDB in 90 Minutes",
    "published_date": ISODate("2024-05-18T14:10:30Z"),
    "tags": ["mongodb", "basics", "database", "nosql"],
    "comments": [
        {
            "comment_id": "LM001",
            "user_id": "AL001",
            "comment_date": ISODate("2024-05-19T14:22:00Z"),
            "comment": "Great read!"
        },
        {
            "comment_id": "LM002",
            "user_id": "AL002",
            "comment_date": ISODate("2024-06-01T08:00:00Z"),
            "comment": "So easy to understand - thank you for posting this!"
        }
    ]
```

```
}
```

Collection II

```
...
{
    "id": "AL001",
    "name": "Andrea Moleri"
},
{
    "id": "AL002",
    "name": "Filippo Armani"
},
{
    "id": "AL003",
    "name": "Claudio Finazzi"
},
...
```

Scaling a Data Model   Creating non-scalable Data Models is a common issue that has serious consequences. The principle of "data that is accessed together should be stored together" is not merely a mantra; it is based on the notion that the way we access our data should align with the data model to achieve optimal efficiency in query result times, memory usage, CPU usage, and storage. When designing a Data Model, we aim to avoid unbounded documents, which are documents whose size can grow indefinitely. This can occur with Document Embedding.

Consider the following example in the code snippet below, where we have the structure of a Blog Post and its comments. Currently, all comments on a single blog post are within an array in the Blog Post Document. However, what happens if we have thousands of comments on a single post? There could be issues related to the growth of the comments array, including the fact that the document will occupy increasingly more memory space, potentially leading to write performance issues as, with each comment addition, the entire document is rewritten in the MongoDB Data Storage. Additionally, pagination of comments will be complex. Comments cannot be easily filtered in this manner, so we would need to retrieve them all and potentially filter them in the application. Furthermore, we must not overlook the maximum BSON document size of 16MB, avoiding which can lead to storage problems. The benefits of the model shown are that we can retrieve all documents in a single Read, but this is not a feature we require, so the following code certainly has more drawbacks than advantages.

```
{
    "title": "Learn The Basics of MongoDB in 90 Minutes",
    "url": "https://www.mongodbbasics.com",
    "text": "Let's learn the basics of MongoDB!",
    "comments": [{
        "name": "Andrea Moleri",
        "created_on": "2024-07-21T11:00:00Z",
        "comment": "I learned a lot!"
    }, {
        "name": "Filippo Armani",
        "created_on": "2024-07-22T11:00:00Z",
        "comment": "Looks great"
    }
    ]
}
```

To resolve the issue, we avoid using Embeddings and partition our data into multiple Collections, using References to keep frequently accessed data together, effectively creating two different

collections: one called `blog_post` and another called `comments`, as illustrated below. We can use the `blog_entry_id` field as a reference between the two collections.

Blog Post Collection

```
{
    "_id": 1,
    "title": "Learn The Basics of MongoDB in 90 Minutes",
    "url": "https://www.mongodbbasics.com",
    "text": "Let's learn the basics of MongoDB!"
}
```

Comments Collection

```
{
    "blog_entry_id": 1,
    "name": "Andrea Moleri",
    "created_on": "2024-07-21T11:00:00Z",
    "comment": "I learned a lot!"
},
{
    "blog_entry_id": 1,
    "name": "Filippo Armani",
    "created_on": "2024-07-22T11:00:00Z",
    "comment": "Looks great"
}
```

Cassandra Data Representation

In Apache Cassandra, data is organized using a unique combination of column-based and row-based structures, which significantly differ from the traditional conventions of relational databases. Unlike relational databases where the schema is rigid and predefined, Cassandra's data model is designed for flexibility and scalability. The primary data storage unit in Cassandra is the Column Family, which can be likened to a table in a relational database, albeit with a distinctive twist: each row in a Column Family can contain a variable number of columns, allowing for a highly flexible schema design.

A Column Family in Cassandra is essentially a set of key-value pairs. In this context, the key is utilized for identifying columns and for distributing and partitioning data among the cluster nodes, while the value comprises a set of columns. Conceptually, each key-value pair in a Column Family corresponds to a table row in a relational database, with the key value serving as the identifier for the row.

In this model, rows are collections of columns, where each column is characterized by a name, a value, and a timestamp. The column name serves as an identifier for the specific piece of data, the value holds the actual data, and the timestamp is crucial for conflict resolution during write operations. This design is particularly advantageous because it provides the flexibility needed to handle diverse data models and use cases efficiently. For example, it is possible to have a row where one entry has only a few columns while another has hundreds or even thousands, all within the same Column Family. This adaptability makes Cassandra particularly suited for applications where data structures can vary significantly from one record to another.

Within a Column Family, Cassandra also supports a more advanced data structure known as SuperColumns. Essentially, a SuperColumn is a collection of columns grouped under a single key. Each column within a SuperColumn has its own name, value, and timestamp, much like regular columns. This hierarchical data structure allows for more complex data models and relationships to be represented. For instance, a SuperColumn can be used to model nested data, such as a

KEYSPACE

COLUMN FAMILY

ROW_ID1 →

name	name	...	name
value	value	...	value
timestamp	timestamp	...	timestamp

ROW_ID2 →

name	name	...	name
value	value	...	value
timestamp	timestamp	...	timestamp

...

ROW_IDN →

name	name	...	name
value	value	...	value
timestamp	timestamp	...	timestamp

...

KEYSPACE

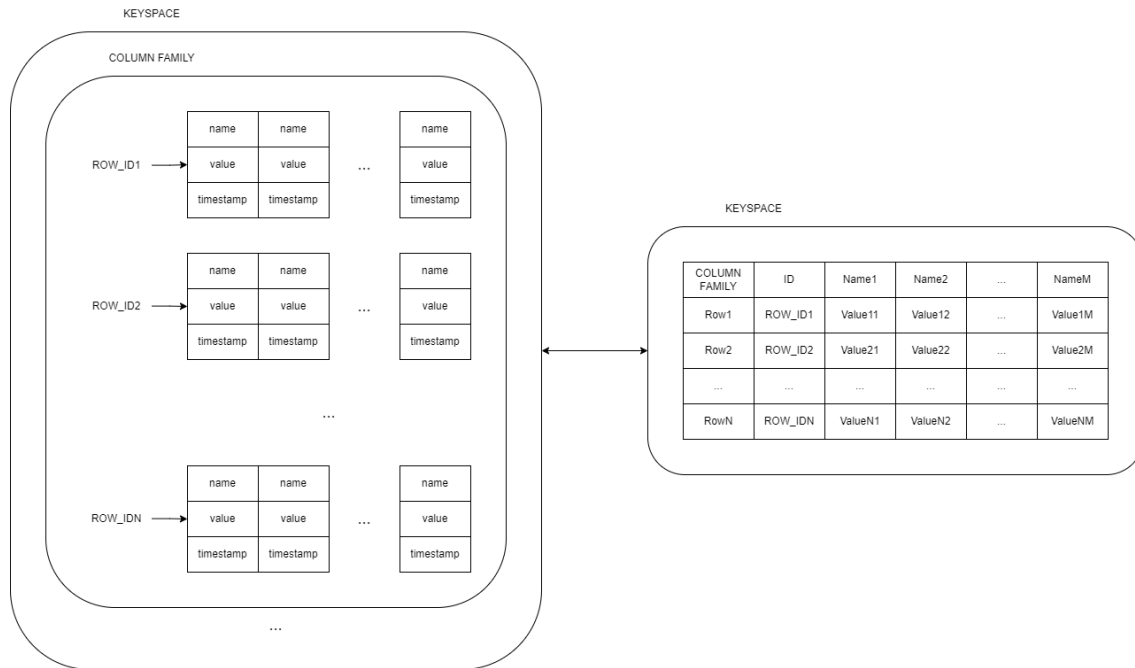| COLUMN FAMILY	ID	Name1	Name2	...	NameM
Row1	ROW_ID1	Value11	Value12	...	Value1M
Row2	ROW_ID2	Value21	Value22	...	Value2M
...	...	...	...	...	...
RowN	ROW_IDN	ValueN1	ValueN2	...	ValueNM

Figure 1: Data Representation in Cassandra

blog post with multiple comments where each comment has attributes like author, timestamp, and content. Despite their utility, SuperColumns have become less favored in newer versions of Cassandra, which now prefer the use of collections such as maps, lists, and sets. These collections offer a more straightforward and efficient approach to data modeling, simplifying the representation of complex relationships like one-to-many and many-to-many associations.

One of Cassandra's key strengths is its support for wide-row tables, where a single row key can have an exceptionally large number of columns. This feature is particularly useful for applications like social media platforms, event logging, or time-series data where it is common to have many related records associated with a single key. For instance, in a social media application, a user's timeline can be stored as a wide row, with each column representing an individual post or activity. This capability allows Cassandra to handle large volumes of related data efficiently under a single key, making it an excellent choice for managing data that grows over time.

Cassandra, unlike relational models and MongoDB, strongly discourages the use of relationships between data/tables, instead favoring data duplication. Instead of using complex joins as in relational databases, data is designed to be read together in the same row or partition, reducing the need for costly joins. This denormalization approach is particularly advantageous in distributed environments, where data retrieval from multiple nodes can be slow. By storing related data together, Cassandra optimizes read and write performance. In fact, from a computational cost point of view, it is much more efficient for Cassandra to perform multiple write operations to keep the data consistent, compared to performing join operations based on the queries.

To facilitate efficient data storage and transmission, Cassandra uses a flexible serialization format for writing data to disk and communicating between nodes. This serialization format is designed to ensure efficient storage and fast network communication, contributing to Cassandra's high performance in managing large volumes of data. The combination of a flexible schema, efficient

serialization, and robust partitioning mechanisms allows Cassandra to adapt to a wide variety of access patterns and workloads while maintaining high performance and scalability.

Database Connection

Connecting to a MongoDB Database

Introducing Atlas, MongoDB's DBaaS   MongoDB is a NoSQL database management system renowned for its flexibility and scalability, including the ability to manage huge amounts of data. A crucial aspect of MongoDB's ecosystem is Atlas, MongoDB's Multi-Cloud Developer Data Platform. Understanding how MongoDB Atlas stores and hosts data through Atlas Clusters is fundamental to using its capabilities effectively.

Atlas serves as a comprehensive developer data platform, with its core offering being a Database as a Service (DBaaS). This implies that utilizing Atlas alleviates the burden of manual MongoDB management, as Atlas takes care of all lifecycle details. Deployments on Atlas benefit from built-in replication, referred to as Replica Sets by MongoDB, ensuring data redundancy and availability even in the event of server failures.

Atlas offers two primary types of Database Deployments: Serverless and Clusters. Serverless instances scale on-demand, charging users only for the resources utilized, making them ideal for applications with highly variable workloads. On the other hand, Clusters consist of multiple MongoDB servers working together. Shared Clusters, suitable for initial exploration, include a free tier, while Dedicated Clusters offer enhanced resources and customization options tailored to specific project requirements, making them optimal for production workloads. Atlas allows deployment across major cloud providers such as AWS, Azure, and Google Cloud.

Global deployment across multiple regions and clouds is facilitated by Atlas, ensuring flexibility to adapt to changing needs. Tier adjustments can be made seamlessly as products scale, without causing any downtime. Operational insights, backups with Point-In-Time Restoration, and features like built-in data tiering with online archiving contribute to Atlas's appeal across various stages of the development cycle. MongoDB serves as the underlying data engine powering Atlas's data management and analysis capabilities. Contrary to a common misconception, MongoDB is not simply the locally deployed version of Atlas.

Creating and Deploying an Atlas Cluster   To effectively utilize MongoDB Atlas, the initial step involves creating an account. Upon account creation, selecting the Database M0 free sandbox tier, which provides 512MB of Storage, Shared RAM, and vCPU, initiates the process. Subsequently, a Dashboard is presented, housing various functionalities. Within this interface, the Organizations feature facilitates the grouping and management of users and projects. Projects, in turn, enable the organization of resources such as Database Clustering, allowing for the creation of distinct projects tailored for Development, Testing, and Production Environments. Notably, by default, databases are generated devoid of users and external IP addresses with access privileges. Consequently, creating an Admin user and an Access Point for the designated IP Address becomes important. Navigating to the Security Quickstart section enables the authentication of connections through username and password protocols, thereby establishing a new administrative user. Given MongoDB's default restriction on IP access to all addresses except those designated by Atlas, configuring access from the local machine necessitates inclusion of the local IP address within the Access List. This can be achieved by selecting "Add My Current IP Address" and, if required, additional addresses can be incorporated via the menu before finalizing with the "Finish and Close" button.

The MongoDB Connection String allows us to connect to the cluster and work with the data. It describes the host we will use and the options for connecting to a MongoDB database. For example,

19

the Connection String can be used to connect from the Mongo Shell, MongoDB Compass, or any other application. MongoDB offers two formats for the Connection String: the Standard Format and the DNS Seed List Format.

- **Standard Format:** this is used to connect to standalone clusters, replica sets, or sharded clusters.
- **DNS Seed List Format:** this format allows us to provide a DNS server list in our connection string. It offers flexibility in deployment and the ability to rotate servers without reconfiguring clients.

Finding Your Connection String Is it possible to find the Connection String on Atlas by navigating to the `"Database"` section and pressing the `"Connect"` button for the cluster you wish to connect to. This will open a menu that provides options for connecting to the database via MongoDB Shell, Application, or Compass. For now, select `"Connect Your Application"`. This will open step-by-step instructions for connecting to the MongoDB instance. You will be given a Connection String to copy and paste, which you will use to connect to MongoDB.

Structure of the Connection String The connection string begins with the required prefix `mongodb+srv` which identifies it as a MongoDB Connection String.

```
mongodb+srv://
    <username>:<password>@cluster0.usqsf.mongodb.net/?retryWrites=true&w=majority
```

- **srv:** automatically sets the TLS Security Options to true and instructs MongoDB to use the DNS Seedlist.
- **username and password:** these are created for the database in the Atlas Dashboard.
- **Host and optional port number:** if the port number is not specified, MongoDB defaults to port 27017.
- **Additional options:** these include Connection Timeout, TLS, SSL, Connection Pooling, and Read & Write Concerns. In this connection string, `retryWrites` is set to true, instructing MongoDB Drivers to automatically retry certain types of operations when they fail.

Connecting to a MongoDB Atlas Cluster with The Shell

Step-by-Step Connection Process To connect to the MongoDB Shell, it is possible to follow these steps:

1. **Login to Atlas:** start by logging into your MongoDB Atlas account. Navigate to the `Databases` section and click on `Connect` for the desired cluster.

2. **Select Connection Method:** choose the option `Connect with the MongoDB Shell`. This will provide step-by-step instructions for connecting via the shell.

3. **Confirm Shell Installation:** click on `I Have the MongoDB Shell Installed`. Then, copy the provided connection string.

4. **Execute in Terminal:** open your terminal, paste the copied connection string, and press Enter. You will be prompted to enter the Admin Password. After doing so, you will be connected to the cluster.

Note: Ensure that the MongoDB Shell is installed before proceeding. On macOS, you can install it using the following command:

```
brew install mongosh
```

Example connection command:

```
mongosh "mongodb+srv://learningmongodb.hikoksa.mongodb.net/" --apiVersion 1
    ↪ --username admin
```

Upon a successful login, the developer will receive a prompt displaying various details, including the MongoShell Log ID, the connected server, and the versions of MongoDB and MongoShell in use. The MongoDB Shell functions as a Node.js REPL (Read-Eval-Print Loop) environment, offering access to JavaScript variables, functions, conditionals, loops, and control flow statements.

Connecting to a MongoDB Atlas Cluster from an Application   It is time to introduce the concept of MongoDB Drivers. MongoDB Drivers connect our application to our database using a Connection String and through the use of a programming language of our choice. In other words, MongoDB drivers provide a way to connect our database with our application. To find a list of languages supported by MongoDB, it is possible to visit the official MongoDB's drivers list website. Numerous languages are supported, and there is also a section called Community Supported Libraries, which contains drivers for languages not officially supported but maintained by the community. On the aforementioned link, we can choose a language and click on it. The MongoDB Documentation contains resources including a Quick Start section to quickly configure the Drivers. We also have a Quick Reference section that contains the syntax of common commands, and sections for Usage Examples and Fundamentals. Since Java is the language we will be using in this report, is it possible to refer to the following documentation page: MongoDB Java Driver Quick Start.

Always remember that if we encounter issues while following one of the connection procedures mentioned, it is often because we have not granted permissions to the IP address of our local machine in Atlas. If necessary, add these permissions by navigating to `Security > Network Access >` `↪ Add IP Address`. If this does not resolve the issue, the correct spelling of usernames and passwords needs to be checked.

Connecting to MongoDB in Java: Spring Boot & Maven   When building a Java application and connecting it with MongoDB, the application requires a series of libraries to interact with the MongoDB deployment. Collectively, these libraries are referred to as "Drivers". MongoDB maintains official Java Drivers for both synchronous and asynchronous applications. For this report, the synchronous version will be used. The MongoDB drivers simplify the connection and interaction between applications and the database, establish secure connections to MongoDB clusters, and execute database operations on behalf of client applications. Additionally, the drivers allow specifying connection options such as security settings, write durability, read isolation, and more. The official drivers adhere to language best practices and enable the use of the full functionality of the MongoDB deployment.

To start with the practical part using Maven, a Java Maven project can be created with Spring Boot. It is possible to use Spring Boot to create a Java Maven project even if there is no intention to use the specific functionalities of Spring. Spring Boot offers many conventions and tools for project configuration, greatly simplifying development and dependency management. Even if the functionalities of Spring, such as dependency injection or the MVC framework, are not needed, Spring Boot can still provide benefits like integration with embedded servers, convention-based automatic configuration, and tools for dependency management. The specific features of Spring can always be utilized or ignored based on the project's needs. This is why this method of project creation is chosen for the report.

Creating a Java Maven Project with Spring Boot   To create a Java Maven project using Spring Boot, one can use a tool called Spring Initializr. This tool provides an intuitive web interface that allows configuring and generating a customized Spring Boot project quickly. First, visit the Spring Initializr website. Here, the project characteristics can be specified. We will choose Java as the language, Maven as the project type, and version 3.2.5 of Spring Boot. The project will be named `quickstart`, belonging to the group `com.mongodb`. The packaging will be executed in `Jar`, and the Java version will be set to 17.

After configuring the project on Spring Initializr, click the generation button to download the project as a zip file. Once downloaded, extract the zip file's contents into a directory on your computer. This project can then be imported into the preferred integrated development environment (IDE), such as IntelliJ IDEA or Eclipse, using the import function to bring in the newly created project. Once imported, a basic structure will be available to begin application development. Many of the fundamental components of a Spring Boot application, such as support for Spring annotations and dependency management through Maven, will already be configured. Spring Boot significantly simplifies the development process, reducing the time needed for initial project setup and allowing the focus to be on developing the features that make the application unique.

Connecting to MongoDB in Java: Pom   This section will outline the steps required to connect to MongoDB using Java and Maven. The process involves updating the `pom.xml` file to include the necessary MongoDB driver dependencies and writing a Java class to establish and manage the connection to a MongoDB Atlas cluster.

Updating the `pom.xml` File   First, open the Java Maven project and locate the `pom.xml` file, that should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                 "http://maven.apache.org/POM/4.0.0
                     ↪ http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>quickstart</artifactId>
    <version>1.0-SNAPSHOT</version>


    <properties>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>18</source>
                    <target>18</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
</project>
```

To add the MongoDB driver to the project's dependencies, the `pom.xml` file should be updated as follows. Ensure the latest version is used, which can be found in the MongoDB documentation. In this step we will also add several other dependencies that will be useful for future functionalities of the project at hand. To summarize, we made the following updates:

1. **Added MongoDB Driver Dependency:** we included the MongoDB driver dependency with version `5.1.0` under `<dependencies>`. This addition enables our project to interact synchronously with MongoDB databases.

2. **Additional Dependencies for Project Functionality:** alongside the MongoDB driver, we also incorporated other dependencies:

   - **OpenCSV (`com.opencsv:opencsv:5.5.2`):** used for handling CSV input and output.
   - **SLF4J API (`org.slf4j:slf4j-api:1.7.32`):** provides a simple facade for various logging frameworks.
   - **Logback Classic (`ch.qos.logback:logback-classic:1.2.6`):** offers robust logging capabilities for the application.
   - **Jansi (`org.fusesource.jansi:jansi:1.18`):** enhances console output with ANSI escape sequences.

These dependencies collectively enhance the functionality and logging capabilities of our project, setting a foundation for future developments and integrations.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                 "http://maven.apache.org/POM/4.0.0
                     ↪ http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>quickstart</artifactId>
    <version>1.0-SNAPSHOT</version>


    <properties>
        <maven.compiler.source>18</maven.compiler.source>
        <maven.compiler.target>18</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.opencsv</groupId>
            <artifactId>opencsv</artifactId>
            <version>5.5.2</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.32</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.6</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.mongodb</groupId>
            <artifactId>mongodb-driver-sync</artifactId>
            <version>5.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.fusesource.jansi</groupId>
            <artifactId>jansi</artifactId>
            <version>1.18</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>18</source>
                    <target>18</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Connecting to a MongoDB Cluster   After adding the MongoDB driver dependency, the next step is to instruct the application to connect to the Atlas cluster using the Java Synchronous Driver from the Maven repository. It is necessary to have a valid connection string or URI to connect to the Atlas cluster. This can be obtained from the Atlas interface by navigating to `"Databases"` ↪ `-> "Connect" -> "Drivers"` and selecting `"Java 4.3 or Later"` as the version. The connection string provided by Atlas should be used in the Java code. The following example demonstrates how to create a new file named `Connection.java` in the `src` folder of the project and utilize the connection string to establish a connection with the Atlas cluster. Create a new file named `Connection.java` in the `src` folder and insert the following code:

```
/**
 * The Connection class provides functionality to connect to a MongoDB instance
     ↪ and list all available databases.
 */
package com.mongodb.quickstart;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.bson.Document;

import java.util.ArrayList;
import java.util.List;

public class Connection {

    /**
     * The main method establishes a connection to the MongoDB instance specified
         ↪ by the URI provided
     * as a system property and lists all available databases.
     *
     * @param args the command-line arguments (not used)
     */
```

```
    public static void main(String[] args) {
        // Retrieve the MongoDB URI from the system properties
        String connectionString = System.getProperty("mongodb.uri");

        // Establish a connection to the MongoDB instance
        try (MongoClient mongoClient = MongoClients.create(connectionString)) {
            // Retrieve the list of databases
            List<Document> databases = mongoClient.listDatabases().into(new
                ↪ ArrayList<>());

            // Print information about each database
            databases.forEach(db -> System.out.println(db.toJson()));
        }
    }
}
```

Compiling and Running the Project To compile the project, execute the following Maven command from the terminal in the project's root directory:

```
mvn --quiet compile
```

To run the application and connect to the Atlas cluster, use the following Maven command, making sure to replace `<username>` with the username, and `<password>` with the password. For demo purposes during the project, both username and password will be set to `admin`. This is just a toy example, and for security reasons the actual database password should be more complex to ensure safety

```
mvn compile exec:java -Dexec.mainClass="com.mongodb.quickstart.Connection"
-Dmongodb.uri="mongodb+srv://admin:admin@learningmongodb.hikoksa.mongodb.net" +
    ↪ "/?retryWrites=true&w=majority&appName=LearningMongoDB"
```

If the command executes successfully, it will return a list of databases contained within the Atlas cluster.

Best Practices for MongoClient Instances For optimal performance and cost efficiency, MongoDB's documentation recommends to have only one `MongoClient` instance per Atlas cluster in the application. Creating multiple `MongoClient` instances can lead to higher-than-normal database costs. The following example demonstrates the use of a Singleton pattern to ensure a single `MongoClient` instance is used throughout the application, thereby preventing the creation of multiple instance:

```
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.bson.Document;

import java.util.ArrayList;
import java.util.List;

public class Connection {
   private static Connection instance;
   private final MongoClient mongoClient;

   /**
    * Constructs a new Connection instance with the specified connection string.
    *
    * @param connectionString the MongoDB connection string
    */
   private Connection(String connectionString) {
      this.mongoClient = MongoClients.create(connectionString);
   }
```

```java
/**
 * Returns the singleton instance of the Connection class, creating it if
 *    ↪ necessary.
 *
 * @param connectionString the MongoDB connection string
 * @return the singleton instance of Connection
 */
public static synchronized Connection getInstance(String connectionString) {
    if (instance == null) {
        instance = new Connection(connectionString);
    }
    return instance;
}

/**
 * Retrieves and prints the list of databases available on the connected
 *    ↪ MongoDB instance.
 */
public void listDatabases() {
    // Retrieve the list of databases
    List<Document> databases = mongoClient.listDatabases().into(new
        ↪ ArrayList<>());

    // Print information about each database
    databases.forEach(db -> System.out.println(db.toJson()));
}

/**
 * Getter method to return the MongoClient instance.
 *
 * @return MongoClient instance
 */
public MongoClient getMongoClient() {
    return this.mongoClient;
}
}
```

Connecting to a Cassandra Database

To make the most of the Cassandra database in this first phase of the relation it was chosen to use Amazon Web Services as a hosting service. In the chapter relating to large volume management, Docker services will be used to better study the behavior of some nodes.

Connection AWS Cassandra requires the use of Transport Layer Security (TLS) to protect client connections. TLS is an encryption protocol that ensures the security and privacy of data exchanged between client and server applications. Here's how to configure the connection:

1. Download the Starfield digital certificate using the following command and save it as `sf-class2-root.crt` in your local directory or home directory. These digital certificates are used to authenticate the identity of a website and to encrypt the communication between the website and its users, ensuring that the data transmitted is secure and protected from interception.

```
curl https://certs.secureserver.net/repository/sf-class2-root.crt -O
```

2. Convert the Starfield digital certificate into a TrustStore file:

```
openssl x509 -outform der -in sf-class2-root.crt -out temp_file.der
```

```
keytool -import -alias cassandra -keystore cassandra_truststore.jks -file
    ↪ temp_file.der
```

During this step, you will need to create a password (e.g., `my_password`) for the keystore and trust the certificate.

3. Attach the TrustStore file in the JVM arguments:

```
-Djavax.net.ssl.trustStore=path_to_file/cassandra_truststore.jks
-Djavax.net.ssl.trustStorePassword=my_password
```

To establish the connection, we use - DataStax Java driver 3.x for Apache Cassandra. - SigV4 (Signature Version 4): an AWS authentication protocol that uses encrypted signatures to authenticate API requests. This protocol allows requests to be signed with access keys, providing a secure alternative to using a username and password. - IAM (Identity and Access Management) credentials: a set of access keys and passwords associated with an IAM user or role on AWS. These credentials are used to authenticate and authorize requests to AWS resources, ensuring that only authorized users can access specified resources.

The SigV4 authentication plugin enables the use of IAM credentials for users and roles when connecting to Amazon Keyspaces, signing API requests with access keys instead of requiring a username and password. To run this code example, complete the following tasks:

4. Create credentials for your IAM user or role: generate an access key and its password from AWS website, and store them as environment variables.

5. Add the DataStax Java driver for Apache Cassandra to your Java project (in the `pom.xml` file for a Maven project):

```
<dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>3.7.2</version>
</dependency>
```

6. Add the authentication plugin to your application, compatible with version 3.x of the DataStax Java driver for Apache Cassandra:

```
<dependency>
    <groupId>software.aws.mcs</groupId>
    <artifactId>aws-sigv4-auth-cassandra-java-driver-plugin_3</artifactId>
    <version>3.0.3</version>
</dependency>
```

7. Identify the endpoint of the keyspace you want to connect to. A keyspace endpoint has the following format:

```
cassandra.<region>.amazonaws.com:portNumber
```

8. Configure your application's main method to create a pool of connections to Amazon Keyspaces. To do this it is necessary to instantiate a Session object to which a Cluster.Builder object is assigned, which follows the following scheme:

```
Session session = Cluster.builder()
                    .addContactPoint("cassandra.<region>.amazonaws.com")
                    .withPort("portNumber")
                    .withAuthProvider(new SigV4AuthProvider("<region>"))
                    .withSSL()
                    .build()
                    .connect();
```

where: - `.addContactPoint("cassandra.<region>.amazonaws.com")`: adds a contact point to the cluster. endPoint is the address of the coordinator node. - `.withPort(portNumber)`: specifies the port on which the node listens and on which the Cassandra server accepts connections. - `.withAuthProvider(new` ↪ `SigV4AuthProvider("<region>"))`: configure the authentication provider using AWS SigV4. - `.withSSL()`: enables SSL for the connection, ensuring that all data transmitted between the client and the Cassandra server is encrypted. - `.build()`: builds the cluster instance with all specified configurations. - `.connect()`: initiates a connection to the Cassandra cluster and returns a session through which database queries can be performed.

This configuration ensures that connections to your Amazon Keyspaces keyspace are secure and authenticated using TLS and IAM credentials.

In conclusion to this paragraph we report the `App.java` class which will contain the script used to work with the data model. For now the script only allows you to connect to the database.

```
package com.example.cassandra;

import com.datastax.driver.core.*;
import software.aws.mcs.auth.SigV4AuthProvider;

public class App
{
    static Session session = null;

    public static void main( String[] args ) {
        String endPoint = "cassandra.eu-north-1.amazonaws.com";
        int portNumber = 9142;

        session = Cluster.builder()
                .addContactPoint(endPoint)
                .withPort(portNumber)
                .withAuthProvider(new SigV4AuthProvider("eu-north-1"))
                .withSSL()
                .build()
                .connect();

        System.out.println("Connected to Cassandra!");
    }
}
```

Connection with Docker   To connect from Java to a Cassandra database using Docker, you can follow these steps:

1. **Docker Compose Configuration:** create a `docker-compose.yml` file to define the Cassandra containers.

2. **Starting the Cassandra Cluster:** run Docker Compose to start the containers:

```
docker-compose up -d
```

3. **Java Dependencies:** add the necessary dependencies to your Java project. If you are using Maven, add the following to your `pom.xml`:

```
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-core</artifactId>
    <version>4.13.0</version>
</dependency>
```

If you are using Gradle:

```
implementation 'com.datastax.oss:java-driver-core:4.13.0'
```

4. **Connecting to Cassandra from Java:** write the Java code to connect to the Cassandra cluster. An example is:

```
package com.example.cassandra;

import com.datastax.oss.driver.api.core.CqlSession;
import java.net.InetSocketAddress;

public class App {

    public static void main(String[] args) {

        try (CqlSession session = CqlSession.builder()
                .addContactPoint(new InetSocketAddress("localhost", 9042))
                .addContactPoint(new InetSocketAddress("localhost", 9043))
                .withLocalDatacenter("datacenter1")
                .build()) {

            System.out.println("Connected to Cassandra!");
        }
    }
}
```

where `InetSocketAddress` is a class from the standard Java library, located in the `java.net` package, representing a combination of an IP address (which can be either IPv4 or IPv6) and a port to uniquely identify an endpoint. It is commonly used in network applications to specify a communication endpoint.

5. **Data Center Configuration:** if necessary, you can verify and modify the Cassandra configuration, adding information such as the maximum response time for a query.

```
DriverConfigLoader loader =
    ↪ DriverConfigLoader.fromString("datastax-java-driver.basic.request.timeout =
    ↪ 10 seconds");

try (CqlSession session = CqlSession.builder()
        .addContactPoint(new InetSocketAddress("localhost", 9042))
        .addContactPoint(new InetSocketAddress("localhost", 9043))
        .withLocalDatacenter("datacenter1")
        .withConfigLoader(loader)
        .build()) {

    System.out.println("Connected to Cassandra!");
}
```

Syntax

MongoDB Syntax and Pipelines

Methods for Inserting Documents   There are two methods available for inserting documents into a collection in MongoDB: `insertOne()` and `insertMany()`. To use the `insertOne()` method, it is appended to the database as follows: `db.<collection>.insertOne()`. For instance, when connected to a sample database, one can use the command `db.grades.insertOne()`.

If the `grades` collection does not yet exist in the database, MongoDB will automatically create the collection. This is an important point to remember to avoid inadvertently creating new collections within the database. Once the command is set, the document intended for insertion is passed as a

parameter within the `insertOne` method. The following example code can be executed in a bash terminal window connected to an Atlas cluster:

```
db.grades.insertOne({
  student_id: 902011,
  products: [
    {
      type: "exam",
      score: 31,
    },
    {
      type: "homework",
      score: 30,
    },
    {
      type: "quiz",
      score: 29,
    },
    {
      type: "homework",
      score: 28,
    },
  ],
  class_id: 550,
})
```

If the operation is successful, it will return `acknowledged: true` along with the `ObjectID` of the newly created document, which is generated automatically.

Inserting Multiple Documents   To insert multiple documents at once, the `insertMany()` method is used with the following syntax:

```
db.<collection>.insertMany([
    <document 1>,
    <document 2>,
    <document 3>
])
```

As the name suggests, this code inserts multiple documents. An array of documents intended for insertion is passed, separated by commas. This method can be executed in the shell similarly to `insertOne()`. Again, an acknowledgment confirming the insertion of multiple documents into the database, along with their respective `ObjectID` values, will be returned. Below is an example of how to use `insertMany()` in a bash terminal window connected to an Atlas cluster:

```
db.grades.insertMany([
  {
    student_id: 902011,
    products: [
      {
        type: "quiz",
        score: 30,
      },
      {
        type: "homework",
        score: 29,
      },
      {
        type: "quiz",
        score: 28,
      },
      {
```

```
        type: "exam",
        score: 31,
      },
    ],
    class_id: 551,
  },
  {
    student_id: 901234,
    products: [
      {
        type: "exam",
        score: 30,
      },
      {
        type: "quiz",
        score: 26,
      },
      {
        type: "quiz",
        score: 24,
      },
      {
        type: "quiz",
        score: 26,
      },
    ],
    class_id: 550,
  },
  {
    student_id: 932234,
    products: [
      {
        type: "exam",
        score: 27,
      },
      {
        type: "homework",
        score: 28,
      },
      {
        type: "quiz",
        score: 23,
      },
      {
        type: "homework",
        score: 21,
      },
    ],
    class_id: 551,
  },
])
```

Finding Documents in a MongoDB Collection Using the `find()` Method   The `find()` method can be used to locate objects within a collection. Additionally, the `$in` operator can be utilized alongside this method. To use the `find()` method, one can simply execute `db.<collection>.find()`. For instance, assuming there is a collection named `zips`, the following command can be executed from a terminal connected to an Atlas cluster: `db.zips.find()`. This command will return some of the documents contained within the collection.

To view more results, the `it` shell directive can be employed. This directive will (it)erate over the extensive list of results. Therefore, by entering `it` and pressing enter, more results from the collection

can be viewed. To retrieve a specific document from the collection, the syntax `{ field: <value> }` can be used. For example, `db.zips.find({ state: "CA" })` will return all documents with `state: CA`. Another example command might be:

```
db.zips.find({ _id: ObjectId("5c8eccc1caa187d17ca6ed16") })
```

Using the `$in` Operator   The `$in` operator allows for the selection of all documents that have a field value matching one of the values specified in an array. A query in the terminal might follow the syntax:

```
db.<collection>.find({
    <field>: {$in:
        [<value>, <value>, ...]
    }
})
```

Here, the keyword `in` is followed by an array of values to be matched. For example, in the following code, the goal is to find every document containing a city value that is either `ORLANDO` or `FLORIDA`. By executing this command, the database will respond with a list of documents that meet the query's criteria:

```
db.zips.find({ city: { $in: ["ORLANDO", "FLORIDA"] } })
```

Finding Documents by Using Comparison Operators   Comparison operators can be utilized to find documents. These include greater than or `$gt`, less than or `$lt`, less than or equal to or `$lte`, and greater than or equal to or `$gte`. To use a comparison operator, the syntax is `<field>: { <operator> : <value> }`.

Consider the following examples, starting with `$gt`, which returns documents where the field contains a value greater than the specified value. For instance, one might search for prices greater than 100 dollars. In the following code, we specify the document field name, followed by the sub-document field name in quotes. In this case, it is the field `items.price`. When this command is executed, all sub-documents with a price greater than $100 are returned.

The same logic applies for elements that are less than, greater than or equal to, or less than or equal to a specified value. In the code provided below, `sales` is the collection, while the sub-document fields are `items.price` and `customer.age`.

```
> db.sales.find({ "items.price": { $gt: 100 } })

> db.sales.find({ "items.price": { $lt: 100 } })

> db.sales.find({ "customer.age": { $gte: 100 } })

> db.sales.find({ "customer.age": { $lte: 100 } })
```

Querying on Array Elements in MongoDB   To understand how to query specific values, also known as elements, within an array in a MongoDB database, consider a common scenario. This involves searching for all documents that have a field containing the specified value. For example, consider a collection named `Accounts` defined as follows:

```
{
    "account_id": 470650,
    "limit": 10000,
    "products": [
        "Commodities",
```

```
        "Currencies",
        "Stocks"
    ]
}
```

Each document in this collection has a field called `products`. A query can be defined to find all documents containing the value `Stocks`. The syntax for this query is as follows:

```
db.accounts.find({ products: "Stocks" })
```

This syntax is familiar to those used for equality matches. Upon executing the query, all documents will be returned that have a `products` field containing either an array or a scalar value that includes `Stocks`. Documents not containing this value will not be returned.

Using $elemMatch for Array Elements   To perform a query for one or more values but only return a match if they are elements of an array, the `$elemMatch` operator can be used. The syntax for this is shown below:

```
db.accounts.find({
    products: {
        $elemMatch: { $eq: "Stocks" }
    }
})
```

This query ensures that the `products` field is an array containing `Stocks`. Therefore, the returned documents will have a `products` field that is an array containing an element equal to `Stocks`.

The `$elemMatch` operator can also be used to find documents where a single array element matches multiple query criteria. Each query criterion is placed in `$elemMatch`, separated by a comma, as shown in the following syntax:

```
{ <field>: { $elemMatch:
    {
        <query1>,
        <query2>,
        ...
    }
}}
```

Example with Multiple Criteria   Consider a collection named `sales`, focusing on the `items` field. This field contains an array of sub-documents with information about the items. The following query, executed in the terminal, will find all documents with at least one element in the `sales` collection that is an iPhone priced over $800 and with a quantity of at least 1.

```
db.sales.find({
    items: {
        $elemMatch: { name: "iPhone", price: { $gt: 800 }, quantity: { $gte: 1 } }
    }
})
```

After executing this query, the returned documents will contain iPhones with quantities greater than or equal to 1 and prices greater than 800. *In other words, the* 'elemMatch' operator can be used to find all documents that contain the specified sub-document.

Logical Operators in MongoDB In MongoDB, the logical operators `$and` and `$or` can be used to perform queries. The `$and` operator executes a logical AND on an array of one or more expressions, returning all documents that meet all the criteria specified in the array. The syntax for this operator is as follows:

```
db.<collection>.find({
  $and: [
    {<expression>},
    {<expression>},
    ...
  ]
})
```

This `$and` operator has an implicit syntax often used to simplify a query expression. It is sufficient to add a comma between expressions to specify an implicit AND, for example:

```
db.collection.find({ <expression>, <expression> })
```

Understanding the $and Operator It is important to remember that this comma acts just like the AND operator. When used, if even one of the specified criteria does not pass, the document will not be included in the results. Before proceeding to an example, consider a sample document from a collection called `Routes`. Each element in the collection contains information about a particular flight route:

```
{
  "_id": ObjectId("56e9b39b732b6122f877fa80"),
  "airline": {
    "id": 1234,
    "name": "Fly Emirates",
    "alias": "FE",
    "iata": "AAN"
  },
  "src_airport": "BGY",
  "dst_airport": "MXP",
  "codeshare": "",
  "stops": 1,
  "airplane": "747"
}
```

Consider the following query, where all documents are being searched for those whose airline is "Ryanair" and whose number of stops is greater than or equal to `1`. This query will return the relevant documents:

```
db.routes.find({
  $and: [{ "airline": "Ryanair" }, { "stops": { $gte: 1 } }],
})
```

However, the implicit syntax can simplify the query:

```
db.routes.find({ "airline.name": "Ryanair", stops: { $gte: 1 } })
```

This will return the same documents as before.

The $or Operator Next, consider the `$or` operator, which performs a logical OR on an array of two or more expressions, returning documents that match at least one of the provided expressions.

```
db.<collection>.find({
  $or: [
    {<expression>},
```

```
    {<expression>},
    ...
  ]
})
```

An example query using this operator will return all flights either departing from or arriving at the
BGY airport:

```
db.routes.find({
  $or: [{ dst_airport: "BGY" }, { src_airport: "BGY" }],
})
```

Combining Logical Operators   Logical operators can also be combined. Consider the following
example, where an `$and` operator contains two `$or` operators. This query searches for every flight
that has `MXP` as either the departure or arrival airport, and also all flights operated by Ryanair or
using a Boeing 747 airplane:

```
db.routes.find({
  $and: [
    { $or: [{ dst_airport: "MXP" }, { src_airport: "MXP" }] },
    { $or: [{ "airline.name": "Ryanair" }, { airplane: "747" }] },
  ]
})
```

In such cases, the implicit syntax of `$and` is not used. This is because the first OR expression would
be overwritten by the following OR expression. This occurs because two fields with the same name
cannot be stored in the same JSON object. Thus, as a general rule, when including the same
operator more than once in a query, the explicit `$and` operator must be used.

Replacing a Document in MongoDB   Occasionally, documents are erroneously inserted into
a collection. Fortunately, replacing them is straightforward. To replace a single document, the
`replaceOne()` method is used, for example: `db.collection.replaceOne(filter, replacement, options)`.
This method accepts three arguments: filter, replacement, and options. The latter is optional.
Here is an example: incomplete or temporary documents can be replaced with complete ones while
retaining the same `_id`. Below is an example of a document created before the book was ready for
publication, with both `ISBN` and `thumbnailUrl` set to default values.

```
{
    _id: "62c5671541e2c6bcb528308",
    title: "Harry Potter and the Philosopher's Stone",
    ISBN: "",
    thumbnailUrl: "",
    publicationDate: ISODate ("1997-01-01T00:00:00.000z"),
    authors: ["J.K. Rowley"]
}
```

To replace this document with an updated version, the `replaceOne` method is used on the Book
Collection as follows. The `_id` is provided as the filter criteria because it is guaranteed to be unique.
The entire document is replaced by passing the replacement document as the second parameter.
The program output will return a `matchedCount` (how many documents matched the filter) and a
`modifiedCount` (how many of these documents were modified) to indicate the number of updated
documents. In this case, both values will be `1`.

```
db.books.replaceOne(
  {
    _id: ObjectId("62c5671541e2c6bcb528308")
  },
```

```
  {
    title: "Data Science Fundamentals for Python and MongoDB",
    isbn: "1484235967",
    publishedDate: new Date("2018-05-10"),
    thumbnailUrl: "https://m.media-amazon.com/images/I/71opmUBc2wL._AC_UY218_.jpg",
    authors: ["David Paper"],
    categories: ["Data Science"]
  }
)
```

To confirm the modification, the `db.books.findOne({_id: ObjectId("62c5671541e2c6bcb528308")})` method can be invoked. Running this command will allow confirmation that the document has been updated, as it will display the updated document.

Updating MongoDB Documents by Using `updateOne()`   Next, update operators in the MongoDB Shell are discussed. The `updateOne()` method, used with the update operators `$set` and `$push`, is introduced. This method updates a single document and accepts three arguments: filter, update, and options. When updating documents, the `$set` operator can be used to add new fields and values to a document or replace the value of a field with a specified value, while the `$push` operator appends a value to an array. If the array field is absent, `$push` adds it with the value as its element. Consider managing a database called `audio` that contains a collection named `Podcasts`. Below is an example using the `$set` operator to replace the value of a field with a specified value. After running this, `matchedCount` and `modifiedCount` will again be returned.

```
db.podcasts.updateOne(
  {
    _id: ObjectId("62822febf41a74a98nbbec4e")
  },
  {
    $set: {
      subscribers: 100000
    }
  }
)
```

An example of using the `upsert` option, which allows creating a new document if no documents match the filter criteria, is provided below. Upsert stands for Update or Insert. In the following example, an attempt is made to update a non-existent document, but since it does not exist and `upsert` is set to true, it will be created.

```
db.podcasts.updateOne(
  { title: "Power Pizza" },
  { $set: { topics: ["fun", "talk-show", "friendship"] } },
  { upsert: true }
)
```

The final example demonstrates the `$push` operator, which in the following case adds a new value to the `hosts` array field.

```
db.podcasts.updateOne(
  { _id: ObjectId("62822febf41a74a98nbbec4e") },
  { $push: { hosts: "Sio" } }
)
```

Updating MongoDB Documents by Using `findAndModify()`   The `findAndModify()` method is used to return the document that has just been updated. In other words, it performs in a single operation what would otherwise require two operations with `updateOne()` and `findOne()`. This avoids

two roundtrips to the server and ensures that another user does not modify the document before it is viewed, thus returning the correct version of the document. This powerful method ensures the correct version of the document is returned before another thread can modify it. Below is an example that, in addition to modifying the document, also returns the modified document. The `new: true` option is specified to return the modified document instead of the original.

```
db.podcasts.findAndModify({
  query: { _id: ObjectId("62822febf41a74a98nbbec4e") },
  update: { $inc: { subscribers: 1 } },
  new: true
})
```

Updating MongoDB Documents by Using `updateMany()`   To update multiple documents, the `updateMany()` method can be used, which also accepts a filter, an update document, and an optional options object. The example below updates all books published before 2020 to the status `LEGACY`. If `matchedCount` and `modifiedCount` are the same, the update was successful. This method is not an all-or-nothing operation and will not roll back updates. If this occurs, `updateMany()` must be run again to update the remaining documents. Additionally, `updateMany()` lacks isolation: updates will be visible as soon as they are performed, which may not be appropriate for some business requirements.

```
db.books.updateMany(
  { publishedDate: { $lt: new Date("2020-01-01") } },
  { $set: { status: "LEGACY" } }
)
```

Deleting Documents in MongoDB   To delete documents in MongoDB, the `deleteOne()` and `deleteMany()` methods can be used. Both methods accept a filter document and an options object, as seen previously. Below are examples showing how to delete a single document and multiple documents. Once executed, these methods return an `acknowledged` boolean value and an integer `deletedCount` value to confirm the process was successful.

```
# Delete a Single Document
db.podcasts.deleteOne({ _id: ObjectId("62822febf41a74a98nbbec4e") })

# Delete Multiple Documents
db.podcasts.deleteMany({ category: "true-crime" })
```

Using Cursors in MongoDB   In MongoDB, a Cursor is a pointer to the result set of a query. For instance, the `find()` method returns a cursor that points to the documents matching the query. There are also Cursor Methods that can be chained to queries and used to perform actions on the resulting set, such as sorting or limiting the search results, before returning the data to the client. To begin with, results can be returned in a specified order using the `cursor.sort()` method, which has the following syntax:

```
db.collection.find(<query>).sort(<sort>)
```

Within the parentheses of `sort()`, an object specifying the field(s) to sort by and the order of the sort must be included. Use `1` for ascending order and `-1` for descending order. The following code example illustrates this by returning companies with a `category_code` of `"tech"` in alphabetical order. A projection is also shown to return only the names:

```
# Return data on all tech companies, sorted alphabetically from A to Z.
db.companies.find({ category_code: "tech" }).sort({ name: 1 });
```

```
# Projection to return only names
db.companies.find({ category_code: "tech" }, { name: 1 }).sort({ name: 1 });

# Return data on all tech companies, sorted alphabetically from A to Z. Ensure
    ↪ consistent sort order.
db.companies.find({ category_code: "tech" }).sort({ name: 1, _id: 1 });
```

To ensure that documents are returned in a consistent order, a field containing unique values can be included in the sort. A simple way to achieve this is by including the `_id` field in the sort as demonstrated above. The `sort` method can be applied to virtually any type of field.

Limiting the Number of Results   Limiting the number of returned results can improve application performance by avoiding unnecessary data processing. The `Limit Cursor Method` achieves this by using `cursor.limit()` to specify the maximum number of documents that the cursor will return. The syntax is as follows:

```
db.collection.find(<query>).limit(<number>)
```

Here is an example where the three tech companies with the highest number of employees are returned. A projection can also be added to simplify the returned document:

```
# Return the three tech companies with the highest number of employees. Ensure
    ↪ consistent sort order.
db.companies.find({ category_code: "tech" })
  .sort({ number_of_employees: -1, _id: 1 })
  .limit(3);

# Projection on two fields
db.companies.find({ category_code: "tech" }, { name: 1, number_of_employees: 1 })
  .sort({ number_of_employees: -1, _id: 1 })
  .limit(3);
```

Returning Specific Data From a Query in MongoDB   By default, queries in MongoDB return all fields in the matching document. However, sometimes an application may need to use data only from a subset of these fields. In this case, the amount of data returned by MongoDB can be limited by selecting specific fields to return. This process, known as projection, can be used in most find queries. The syntax is:

```
db.collection.find(<query>, <projection>)
```

To include a field, set its value to `1` in the projection document, as shown in the example below. To exclude a field, set its value to `0`. While the `_id` field is included by default, it can be suppressed by setting its value to `0` in any projection, as illustrated in the third example. Note that inclusion and exclusion cannot be combined in most projections, except for the `_id` field, which can be both included and excluded. Accessing a sub-document is also shown, ensuring that the zip code is excluded:

```
# Return all swimming pools inspections - business name, result, and _id fields
    ↪ only
db.inspections.find(
  { sector: "Swimming Pools" },
  { business_name: 1, result: 1 }  # This is the projection document
)

# Return all inspections with result of "Pass" or "Warning" - exclude date and zip
    ↪ code
db.inspections.find(
  { result: { $in: ["Pass", "Warning"] } },
```

```
  { date: 0, "address.zip": 0 }  # This is the projection document
)

# Return all swimming pools inspections - business name and result fields only
db.inspections.find(
  { sector: "Swimming Pools" },
  { business_name: 1, result: 1, _id: 0 }  # This is the projection document
)
```

Counting Documents in a MongoDB Collection   The `db.collection.countDocuments()` method can be used to count the number of documents matching a query. This method takes two parameters: a query document and an options document. The syntax is:

```
db.collection.countDocuments(<query>, <options>)
```

Here are some code examples:

```
# Count number of documents in trip collection
db.trips.countDocuments({})

# Count number of trips over 100 minutes by paid subscribers
db.trips.countDocuments({ tripduration: { $gt: 100 }, usertype: "Paid-Subscriber"
    ↪ })
```

BSON Format in MongoDB   Binary JSON, or BSON, is the data format that MongoDB uses to organize and store data. BSON is optimized for storage, retrieval, and transmission across the wire. Additionally, it is more secure than plain text JSON and supports more data types. The MongoDB Java Driver provides several classes for representing BSON documents, with the `Document` class being recommended due to its flexible and concise data representation. MongoDB provides a BSON interface for types that can render themselves into a BSON Document, and the `Document` class implements this interface.

Here is an example of a BSON document, which includes the usual `_id` field serving as the Primary Key and a sub-document represented by the `address` field. The `date` field is represented as a String, though it is advisable to use a specific BSON type for dates. Summarizing the aforementioned points, one way to represent BSON documents is by using the `Document` class. The `Document` class offers a flexible representation of a BSON document.

```
{
    "_id": { "$oid": "56d61033a378eccde8a8354f" },
    "business_id": "1507-1975-THME",
    "certificate_number": 51395,
    "business_name": "GARDALAND S.R.L.",
    "date": "15 Jul 1975",
    "result": "No Violations",
    "sector": "Theme Parks",
    "address": {
        "city": "Castelnuovo del Garda",
        "zip": 37014,
        "street": "Via Derna",
        "number": 4
    }
}
```

To instantiate this document in Java, use the following syntax. This example demonstrates instantiating a new document and setting its Primary Key, which in this case is `_id`. Subsequently, the corresponding fields and values are appended, such as `Date` for the date. The document is then ready to be sent to the MongoDB Server.

```
Document inspection = new Document("_id", new ObjectId())
    .append("business_id", "1507-1975-THME")
    .append("certificate_number", 51395)
    .append("business_name", "GARDALAND S.R.L.")
    .append("date", Date.from(LocalDate.of(1975, 7,
        ↪ 15).atStartOfDay(ZoneId.systemDefault()).toInstant()))
    .append("result", "No Violations")
    .append("sector", "Theme Parks")
    .append("address", new Document().append("city", "Castelnuovo del
        ↪ Garda").append("zip", 37014).append("street", "Via
        ↪ Derna").append("number", 4));
```

Inserting a Document in Java Applications   To insert a single document into a collection, use the `getCollection()` method to access the `MongoCollection` object, which represents the specified collection. Then, append the `insertOne()` method to the collection object. Within the parentheses of `insertOne()`, include an object that contains the document data and print the inserted document's ID, as shown in the following example, which also contains a sub-document in the `address` field.

```
MongoDatabase database = mongoClient.getDatabase("sample_training");
MongoCollection<Document> collection = database.getCollection("inspections");

Document inspection = new Document("_id", new ObjectId())
        .append("id", "1507-1975-THME")
        .append("certificate_number", 51395)
        .append("business_name", "GARDALAND S.R.L.")
        .append("date", Date.from(LocalDate.of(1975, 7,
            ↪ 15).atStartOfDay(ZoneId.systemDefault()).toInstant()))
        .append("result", "No Violations")
        .append("sector", "Theme Parks")
        .append("address", new Document().append("city", "Castelnuovo del
            ↪ Garda").append("zip", 37014).append("street", "Via
            ↪ Derna").append("number", 4));

InsertOneResult result = collection.insertOne(inspection);
BsonValue id = result.getInsertedId();
System.out.println(id);
```

Similarly, to insert multiple documents into a collection, append the `insertMany()` method to the collection object. Within the parentheses of `insertMany()`, include an object that contains the document data and print out the IDs of the inserted documents. The following example clarifies this process.

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");

Document doc1 = new Document().append("account_holder", "Andrea
    ↪ Moleri").append("account_id", "MDB99115881").append("balance",
    ↪ 1785).append("account_type", "checking");
Document doc2 = new Document().append("account_holder", "Claudia
    ↪ Gatto").append("account_id", "MDB79101843").append("balance",
    ↪ 1468).append("account_type", "checking");

List<Document> accounts = Arrays.asList(doc1, doc2);
InsertManyResult result = collection.insertMany(accounts);
result.getInsertedIds().forEach((x, y) -> System.out.println(y.asObjectId()));
```

Custom methods can be created to simplify these functions, as demonstrated in the following examples:

```
// Example Methods
```

```java
public void insertOneDocument(Document doc) {
  System.out.println("Inserting one account document");
  InsertOneResult result = collection.insertOne(doc);
  BsonValue id = result.getInsertedId();
  System.out.println("Inserted document ID: " + id);
}

public void insertManyDocuments(List<Document> documents) {
  InsertManyResult result = collection.insertMany(documents);
  System.out.println("\tTotal # of documents: " + result.getInsertedIds().size());
}
```

Querying a MongoDB Collection in Java Applications   The `find()` method can be used to search for specific conditions. For example, in the following code, `find()` is used to locate all checking accounts with a balance of at least 1000. Each document returned by the `find()` method is processed by iterating the `MongoCursor` using a try block and a while loop. The `find()` method accepts a query filter and returns documents that match the filters in the collection.

```java
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
try(MongoCursor<Document> cursor = collection.find(and(gte("balance", 1000),
    ↪ eq("account_type", "checking"))).iterator()) {
    while(cursor.hasNext()) {
        System.out.println(cursor.next().toJson());
    }
}
```

The `find()` and `first()` methods can be concatenated to find and return only the first document that matches the query filter given to the `find()` method. For example, the following code returns a single document from the same query. It is important to remember that all queries on MongoDB should use a Query Filter to optimize the use of database resources. The Java `Filters` builder class helps define more efficient queries by using query predicates.

```java
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
Document doc = collection.find(Filters.and(gte("balance", 1000),
    ↪ Filters.eq("account_type", "checking"))).first();
System.out.println(doc.toJson());
```

Again, useful custom methods can be built to perform the same functions but be invoked more easily:

```java
// Example Methods
public void findOneDocument(Bson query) {
  Document doc = collection.find(query).first();
  System.out.println(doc != null ? doc.toJson() : null);
}

public void findDocuments(Bson query) {
  try (MongoCursor<Document> cursor = collection.find(query).iterator()) {
    while (cursor.hasNext()) {
      System.out.println(cursor.next().toJson());
    }
  }
}
```

Updating Documents in Java Applications

Updating a Single Document   To update a single document, use the `updateOne()` method on a `MongoCollection` object. This method accepts a filter that matches the document to be updated and an update statement that instructs the driver on how to modify the matching document. The `updateOne()` method updates only the first document that matches the filter.

In the following example, one document is updated by increasing the balance of a specific account by 100 and setting the account status to active:

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
Bson query  = Filters.eq("account_id", "MDB12234728");
Bson updates  = Updates.combine(Updates.set("account_status", "active"),
    ↪ Updates.inc("balance", 100));
UpdateResult upResult = collection.updateOne(query, updates);
```

Updating Multiple Documents   To update multiple documents, use the `updateMany()` method on a `MongoCollection` object. This method also accepts a filter to match the documents that need to be updated, along with an update statement. The `updateMany()` method updates all documents that match the filter.

In the following example, the minimum balance of all savings accounts is increased to 100:

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
Bson query  = Filters.eq("account_type", "savings");
Bson updates  = Updates.combine(Updates.set("minimum_balance", 100));
UpdateResult upResult = collection.updateMany(query, updates);
```

Creating Utility Methods   Utility methods can be created and called as shown below:

```
// Example of Methods and Usage #1
public class Crud {
    private final MongoCollection<Document> collection;

    public Crud(MongoClient client) {
        this.collection = client.getDatabase("bank").getCollection("accounts");
    }

    public void updateOneDocument(Bson query, Bson update) {
        UpdateResult updateResult = collection.updateOne(query, update);
        System.out.println("Updated a document:");
        System.out.println("\t" + updateResult.getModifiedCount());
    }
}

Bson query = Filters.eq("account_id", "MDB333829449");
Bson update = Updates.combine(Updates.set("account_status", "active"),
    ↪ Updates.inc("balance", 100));
crud.updateOneDocument(query, update);

// Example of Methods and Usage #2
public class Crud {
    private final MongoCollection<Document> collection;

    public Crud(MongoClient client) {
        this.collection = client.getDatabase("bank").getCollection("accounts");
    }

    public void updateManyDocuments(Document query, Bson update) {
        UpdateResult updateResult = collection.updateMany(query, update);
```

```
        System.out.println("Updated this many documents:");
        System.out.println("\t" + updateResult.getModifiedCount());
    }
}
```

Deleting Documents in Java Applications

Deleting a Single Document To delete a single document from a collection, use the `deleteOne()` method on a `MongoCollection` object. This method accepts a query filter that matches the document to be deleted. If no filter is specified, MongoDB matches the first document in the collection. The `deleteOne()` method deletes only the first document that matches.

In the following example, a single document related to Andrea Moleri's account is deleted. Assume that instances of `MongoClient` and `MongoCollection` have already been instantiated:

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
Bson query = Filters.eq("account_holder", "Andrea Moleri");
DeleteResult delResult = collection.deleteOne(query);
System.out.println("Deleted a document:");
System.out.println("\t" + delResult.getDeletedCount());
```

Deleting Multiple Documents To delete multiple documents in a single operation, use the `deleteMany()` method on a `MongoCollection` object. Specify the documents to be deleted with a query filter. If an empty document is provided, MongoDB matches all documents in the collection and deletes them.

In the following example, all dormant accounts are deleted using a query object, and the total number of deleted documents is printed:

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
Bson query = Filters.eq("account_status", "dormant");
DeleteResult delResult = collection.deleteMany(query);
System.out.println(delResult.getDeletedCount());
```

Creating Utility Methods Utility methods for deletion can also be created and called as shown below:

```
// Example of Methods and Usage #1
public class Crud {
    private final MongoCollection<Document> collection;

    public Crud(MongoClient client) {
        this.collection = client.getDatabase("bank").getCollection("accounts");
    }

    public void deleteOneDocument(Bson query) {
        DeleteResult delResult = collection.deleteOne(query);
        System.out.println("Deleted a document:");
        System.out.println("\t" + delResult.getDeletedCount());
    }
}

// Example of Methods and Usage #2
public class Crud {
    private final MongoCollection<Document> collection;
```

```
    public Crud(MongoClient client) {
        this.collection = client.getDatabase("bank").getCollection("accounts");
    }

    public void deleteManyDocuments(Bson query) {
        DeleteResult delResult = collection.deleteMany(query);
        System.out.println("Deleted this many documents:");
        System.out.println("\t" + delResult.getDeletedCount());
    }
}
```

Creating MongoDB Transactions in Java Applications   In this section, we demonstrate
how to create a multi-document transaction in MongoDB using Java. A multi-document trans-
action ensures the atomicity of reads and/or writes across multiple documents. Specifically, a
transaction is a sequence of operations executed on a database that represents a single unit of
work. Once committed, all write operations within the transaction are persisted. If a transaction is
aborted or fails to complete successfully, all associated write operations are rolled back. Therefore,
all operations within a transaction either succeed or fail together. This property is known as
atomicity. Transactions also ensure the consistency, isolation, and durability of operations. These
qualities—Atomicity, Consistency, Isolation, and Durability—are collectively referred to as ACID
compliance.

Implementation Example   To initiate a transaction in MongoDB using Java, we utilize the
`WithTransaction()` method of a session object. Below are the steps involved in completing a multi-
document transaction, followed by the corresponding code snippet:

1. **Session Initialization and Transaction Start:** Begin by establishing a new session and
   starting a transaction using the `WithTransaction()` method on the session object.

2. **Transaction Operations:** Define the operations to be performed within the transaction.
   This typically includes fetching necessary data, performing updates, and inserting documents.

3. **Transaction Commit:** After executing all operations successfully, commit the transaction
   to persist the changes.

4. **Handling Timeouts and Resource Closure:** MongoDB automatically cancels any multi-
   document transaction that exceeds 60 seconds. Additionally, ensure proper closure of resources
   utilized by the transaction.

```
final MongoClient client = MongoClients.create(connectionString);
final ClientSession clientSession = client.startSession();

TransactionBody txnBody = new TransactionBody<String>() {
   public String execute() {
       MongoCollection<Document> bankingCollection =
           ↪ client.getDatabase("bank").getCollection("accounts");

       Bson fromAccountFilter = eq("account_id", "MDB310054629");
       Bson withdrawalUpdate = Updates.inc("balance", -200);

       Bson toAccountFilter = eq("account_id", "MDB643731035");
       Bson depositUpdate = Updates.inc("balance", 200);

       System.out.println("Withdrawing from Account " +
           ↪ fromAccountFilter.toBsonDocument().toJson() + ": " +
           ↪ withdrawalUpdate.toBsonDocument().toJson());
```

```java
        System.out.println("Depositing to Account " +
            ↪ toAccountFilter.toBsonDocument().toJson() + ": " +
            ↪ depositUpdate.toBsonDocument().toJson());

        bankingCollection.updateOne(clientSession, fromAccountFilter,
            ↪ withdrawalUpdate);
        bankingCollection.updateOne(clientSession, toAccountFilter, depositUpdate);

        return "Transferred funds from Andrea Moleri to Claudia Gatto";
    }
};

try {
    clientSession.withTransaction(txnBody);
} catch (RuntimeException e) {
    System.out.println("Transaction aborted: " + e.getMessage());
} finally {
    clientSession.close();
}
```

This Java code snippet exemplifies the process described. It begins by initializing a MongoDB client and starting a session. Within the `execute()` method of the `TransactionBody`, two updates are performed atomically on specified accounts. If all operations succeed, the transaction commits; otherwise, it rolls back automatically. Finally, the session is closed to release associated resources.

By following these steps and utilizing MongoDB's transaction capabilities in Java, developers can ensure reliable and consistent data operations across multiple documents within a MongoDB database. What follows is another example of a real-world scenario in which we would use the method.

```java
// DemoApp.java
public class DemoApp {
    public static void main(final String[] args) {
        Logger root = (Logger) LoggerFactory.getLogger("org.mongodb.driver");
        // Available levels are: OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL
        root.setLevel(Level.WARN);

        String connectionString = System.getenv("MONGODB_URI");
        try (MongoClient client = MongoClients.create(connectionString)) {
            //Transaction
            Transaction txn = new Transaction(client);
            var senderAccountFilter = "MDB310054629";
            var receiverAccountFilter = "MDB643731035";
            double transferAmount = 200;
            txn.transferMoney(senderAccountFilter, transferAmount,
                ↪ receiverAccountFilter);
        }
    }
}

// Transaction.java
public class Transaction {
    private final MongoClient client;

    public Transaction(MongoClient client) {
        this.client = client;
    }

    public void transferMoney(String accountIdOfSender, double transactionAmount,
        ↪ String accountIdOfReceiver) {
    try (ClientSession session = client.startSession()) {
        UUID transfer = UUID.randomUUID();
```

```
        String transferId = transfer.toString();
        try {
            session.withTransaction(() -> {
                MongoCollection<Document> accountsCollection =
                    ↪ client.getDatabase("bank").getCollection("accounts");
                MongoCollection<Document> transfersCollection =
                    ↪ client.getDatabase("bank").getCollection("transfers");


                Bson senderAccountFilter = eq("account_id", accountIdOfSender);
                Bson debitUpdate = Updates.combine(inc("balance", -1 *
                    ↪ transactionAmount),push("transfers_complete", transferId));

                Bson receiverAccountId = eq("account_id", accountIdOfReceiver);
                Bson credit = Updates.combine(inc("balance", transactionAmount),
                    ↪ push("transfers_complete", transferId));

                transfersCollection.insertOne(session, new Document("_id", new
                    ↪ ObjectId()).append("transfer_id",
                    ↪ transferId).append("to_account",
                    ↪ accountIdOfReceiver).append("from_account",
                    ↪ accountIdOfSender).append("amount",
                    ↪ transactionAmount).append("last_updated", new Date()));
                accountsCollection.updateOne(session, senderAccountFilter,
                    ↪ debitUpdate);
                accountsCollection.updateOne(session, receiverAccountId, credit);
                return null;
            });
        } catch (RuntimeException e) {
            throw e;
        }
    }
}
}
```

Introduction to MongoDB Aggregation In the realm of databases, aggregation involves the analysis and summary of data, where an aggregation stage represents an operation performed on data without permanently altering the source data. MongoDB facilitates the creation of aggregation pipelines, where developers specify aggregation operations sequentially. What distinguishes MongoDB aggregations is the ability to chain these operations into a pipeline, consisting of stages where data can be filtered, sorted, grouped, and transformed. Documents output from one stage become the input for the next. In MongoDB Atlas, developers can access the Aggregation tab to add stages one by one and view results for each stage. Similarly, this can be accomplished using MongoDB CLI or MongoDB Language Drivers. Below is an example of aggregation syntax using the CLI, starting with `db.collection.aggregate` followed by stage names and their contained expressions. Each stage represents a discrete data operation, commonly including `$match` for filtering data, `$group` for grouping documents, and `$sort` for ordering documents based on specified criteria. The use of `$` prefix signifies a field path, referencing the value in that field, useful for operations like concatenation (`$concat: ['$first_name', '$last_name']`).

```
db.collection.aggregate([
    {
        $stage1: {
            { expression1 },
            { expression2 }...
        },
        $stage2: {
```

```
            { expression1 }...
        }
    }
])
```

Using $match and $group Stages in a MongoDB Aggregation Pipeline   The `$match` stage filters documents that match specified conditions, as illustrated in the example below. The `$group` stage groups documents based on a specified group key. These stages are commonly used together in an aggregation pipeline. In the example, the aggregation pipeline identifies documents with a `"state"` field matching `"CA"` and then groups these documents by the `$city` group key to count the total number of zip codes in California. Placing `$match` early in the pipeline optimizes performance by utilizing indexes to reduce the number of documents processed. Conversely, the output of `$group` is a document for each unique value of the group key. Note that `$group` includes `_id` as the group key and an accumulator field, specifying how to aggregate information for each group. For instance, grouping by city and using `count` as an accumulator determines the count of ZIP Codes per city.

```
# Example of Match Stage
{
    $match: {
        "field_name": "value"
    }
}

# Example of Group Stage
{
    $group:
    {
        _id: <expression>, // Group key
        <field>: { <accumulator> : <expression> }
    }
}

# Example Using Both
db.zips.aggregate([
    { $match: { state: "CA" } },
    {
        $group: {
            _id: "$city",
            totalZips: { $count : { } }
        }
    }
])
```

Using $sort and $limit Stages in a MongoDB Aggregation Pipeline   Next, the `$sort` and `$limit` stages in MongoDB aggregation pipelines are discussed. The `$sort` stage arranges all input documents in a specified order, using `1` for ascending and `-1` for descending order. The `$limit` stage restricts output to a specified number of documents. These stages can be combined, such as in the third example where documents are sorted in descending order by population (`pop`), and only the top five documents are returned. `$sort` and `$limit` stages are essential for quickly identifying top or bottom values in a dataset. Order of stages is crucial; arranging `$sort` before `$limit` yields different results compared to the reverse order.

```
# Example of Sort Stage
{
    $sort: {
        "field_name": 1
    }
}
```

```
# Example of Limit Stage
{
    $limit: 5
}

# Example Using Both
db.zips.aggregate([
    { $sort: { pop: -1 } },
    { $limit: 5 }
])
```

Using $project, $count, and $set Stages in a MongoDB Aggregation Pipeline   Moving on
to `$project`, `$set`, and `$count` stages in MongoDB aggregation pipelines. The `$project` stage specifies
output document fields, including (`1` for inclusion, `0` for exclusion), and optionally assigns new values
to fields. This stage is typically the final one to format output. The `$set` stage creates new fields or
modifies existing ones within documents, facilitating changes or additions for subsequent pipeline
stages. The `$count` stage generates a document indicating the count of documents at that stage in
the pipeline. `$set` is useful for field modifications, while `$project` controls output field visibility and
value transformations. `$count` provides a count of documents in the aggregation pipeline stage.

```
# Example of Project Stage
{
    $project: {
        state: 1,
        zip: 1,
        population: "$pop",
        _id: 0
    }
}

# Example of Set Stage
{
    $set: {
        place: {
            $concat: ["$city", ",", "$state"]
        },
        pop: 10000
    }
}

# Example of Count Stage
{
    $count: "total_zips"
}
```

Using the $out Stage in a MongoDB Aggregation Pipeline   The `$out` stage facilitates
the creation of a new collection from the output of an aggregation pipeline. It writes documents
returned by the pipeline into a specified collection. This stage must be the last one in the pipeline.
Note that `$out` creates a new collection if one does not already exist. If the collection exists, `$out`
overwrites it with new data. Therefore, careful consideration of the collection name is advised to
avoid unintentionally overwriting existing data. The `$out` stage expects the database name in the
`db` field and the collection name in the `coll` field. Alternatively, providing just the collection name
directly is also valid. Executing `$out` does not produce command-line output; instead, results of the
aggregation pipeline are written to a new collection, confirmed by `show collections` command in the
terminal.

```
# Mode 1
```

```
$out: {
    db: "<db>",
    coll: "<newcollection>"
}

# Mode 2
{ $out: "<newcollection>" }

# Example
db.sightings.aggregate([
    {
        $match: {
            date: {
                $gte: ISODate('2024-01-01T00:00:00.0Z'),
                $lt: ISODate('2024-01-01T00:00:00.0Z')
            }
        }
    },
    {
        $out: 'sightings_2024'
    }
])
db.sightings_2022.findOne()
```

Building a MongoDB Aggregation Pipeline in Java Applications When using the MongoDB Aggregation Framework to construct queries, one must conceptualize these queries as composed of discrete stages, where each stage produces an output document that serves as input to the next stage. This aggregation pipeline simplifies debugging and maintenance of individual stages, facilitating query rewriting and optimization. The expression operators used within this framework function akin to functions, offering a broad spectrum including arithmetic, trigonometric, date, and boolean operators. Once assembled, the aggregation pipeline can be validated using tools such as MongoShell, Atlas Aggregation Builder, and Compass before integration into the chosen programming language.

Using MongoDB Aggregation Stages with Java: $match and $group In the following Java examples, the `Aggregates` builder class is employed to configure `$match` and `$group` stages within MongoDB aggregation pipelines. Each example demonstrates how to utilize these stages effectively to manipulate and aggregate data.

```
public static void main(String[] args) {
    String connectionString = System.getProperty("mongodb.uri");
    try (MongoClient mongoClient = MongoClients.create(connectionString)) {
        MongoDatabase db = mongoClient.getDatabase("bank");
        MongoCollection<Document> accounts = db.getCollection("accounts");
        matchStage(accounts);
    }
}

private static void matchStage(MongoCollection<Document> accounts){
    Bson matchStage = Aggregates.match(Filters.eq("account_id", "MDB310054629"));
    System.out.println("Display aggregation results");
    accounts.aggregate(Arrays.asList(matchStage)).forEach(document ->
        ↪ System.out.print(document.toJson()));
}
```

```
public static void main(String[] args) {
    String connectionString = System.getProperty("mongodb.uri");
    try (MongoClient mongoClient = MongoClients.create(connectionString)) {
        MongoDatabase db = mongoClient.getDatabase("bank");
        MongoCollection<Document> accounts = db.getCollection("accounts");
        matchAndGroupStages(accounts);
    }
}

private static void matchAndGroupStages(MongoCollection<Document> accounts){
    Bson matchStage = Aggregates.match(Filters.eq("account_id", "MDB310054629"));
    Bson groupStage = Aggregates.group("$account_type", sum("total_balance",
        ↪ "$balance"), avg("average_balance", "$balance"));
    System.out.println("Display aggregation results");
    accounts.aggregate(Arrays.asList(matchStage, groupStage)).forEach(document ->
        ↪ System.out.print(document.toJson()));
}
```

Using MongoDB Aggregation Stages with Java: $sort and $project   This example
illustrates the use of `$sort` and `$project` stages within MongoDB aggregation pipelines, emphasizing
sorting and projecting fields from queried documents.

```
public static void main(String[] args) {
    String connectionString = System.getProperty("mongodb.uri");
    try (MongoClient mongoClient = MongoClients.create(connectionString)) {
        MongoDatabase db = mongoClient.getDatabase("bank");
        MongoCollection<Document> accounts = db.getCollection("accounts");
        matchSortAndProjectStages(accounts);
    }
}

private static void matchSortAndProjectStages(MongoCollection<Document> accounts){
    Bson matchStage =
            Aggregates.match(Filters.and(Filters.gt("balance", 1500),
                ↪ Filters.eq("account_type", "checking")));
    Bson sortStage = Aggregates.sort(Sorts.orderBy(descending("balance")));
    Bson projectStage = Aggregates.project(
            Projections.fields(
                    Projections.include("account_id", "account_type", "balance"),
                    Projections.computed("euro_balance", new Document("$divide",
                        ↪ Arrays.asList("$balance", 1.20F))),
                    Projections.excludeId()
            )
    );
    System.out.println("Display aggregation results");
    accounts.aggregate(Arrays.asList(matchStage, sortStage,
        ↪ projectStage)).forEach(document -> System.out.print(document.toJson()));
}
```

These examples demonstrate the structured use of MongoDB aggregation stages in Java applications,
showcasing the flexibility and power of the MongoDB Aggregation Framework for data analysis
and manipulation. Each stage—`$match`, `$group`, `$sort`, and `$project`—plays a crucial role in shaping
and refining the results of queries executed against MongoDB databases.

Cassandra Query Language (CQL)

Unlike MongoDB, Cassandra uses a communication language that is very similar to SQL, typical of
relational systems. The most basic way to interact with Apache Cassandra is by using the CQL

shell, cqlsh. With cqlsh, you can create keyspaces and tables, insert data into tables, query tables, and much more. This makes it easier for those familiar with SQL to transition to Cassandra and leverage its capabilities for managing large-scale, distributed datasets.

In the following subparagraphs the main possible operations of the CQL language will be illustrated.

Creating a Keyspace   In Cassandra, a keyspace is the highest level of abstraction for organizing data. It is, basically, a top-level namespace. When creating a keyspace, two parameters must be specified:

- **Replication Factor:** the number of copies of the data maintained within the cluster;

- **Strategy of Replication:** how the data are replicated. The choice of strategy depends on specific performance, fault tolerance, and data compaction needs. The main options are:
  - **SimpleStrategy:** used only for a single datacenter and one rack. It places the first replica on a node determined by the partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering topology.
  - **NetworkTopologyStrategy:** used when you have multiple data centers available. This strategy specifies how many replicas you want in each data center. NetworkTopologyS-trategy places replicas in the same data center by traveling the ring clockwise until it reaches the first node in another rack. NetworkTopologyStrategy tries to place replicas on distinct racks because nodes in the same rack often fail at the same time.

Less important alternatives include the "LeveledCompactionStrategy" for efficient data compaction, and the "DateTieredCompactionStrategy," ideal for long-term storage of time-based data.

```
CREATE  KEYSPACE [IF NOT EXISTS] keyspace_name
   WITH REPLICATION = {
      'class' : 'SimpleStrategy', 'replication_factor' : N }
      | 'class' : 'NetworkTopologyStrategy',
      'dc1_name' : N [, ...]
   }
   [AND DURABLE_WRITES =  true|false] ;
```

An optional parameter is DURABLE_WRITES and if it is set to false it allows the commit log to be ignored when writing to the keyspace by disabling durable writes. The default value is true.

Selecting the Keyspace   After creating a keyspace, it must be selected before performing operations on tables within that keyspace.

```
USE keyspace_name;
```

Creating a Table   As mentioned in the previous chapter, data in Cassandra can be viewed as contents within tables composed of rows and columns. Through the CQL language, developers have a rather abstract understanding of the database architecture; they do not need to have full knowledge of nodes, racks, and datacenters, but only basic information.

Tables in Cassandra are organized around a primary key, which can consist of one or more columns, and clustering keys. It is essential to design primary keys correctly based on data access and performance requirements. The main (not complete) structure of the instruction to create a table involves:

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name (
   column_name column_type [, ...]
   PRIMARY KEY (column_name [, column_name ...])
[WITH CLUSTERING ORDER BY (clustering_column_name order])]
```

The PRIMARY KEY clause identifies the set of columns that form the primary key of the table. This clause must be unique within the table. It can consist of a single column or a combination of columns. If the primary key consists of multiple columns, the first specified column is the partition key, while the subsequent columns are clustering keys that determine the order of rows within each partition.

The WITH CLUSTERING ORDER BY clause specifies the order in which rows with the same partition key are stored and retrieved. You use the name of one of the clustering keys defined in the primary key, followed by ASC (ascending) or DESC (descending), to establish the clustering order of rows.

Data Types   The main data types in Cassandra are:

- **Primitive Data Types:**
  - `tinyint` (8-bit integers), `smallint` (16-bit integers), `int` (32-bit integers), `bigint` (64-bit integers),
  - `varint` (variable-length integers),
  - `float` (7 decimal digits of precision for a number), `double`, (15 decimal digits of precision for a number)
  - `boolean`,
  - `text`,
  - `blob` (intended for storing large data in binary format not directly interpretable by the database)
  - ...
- **Collection Data Types:**
  - `list`: Ordered list of values of the same type (e.g., `list<int>`).
  - `set`: Unordered set of unique values of the same type (e.g., `set<int>`).
  - `map`: Map of key-value pairs with arbitrary types (e.g., `map<int, text>`).
- **Temporal Data Types:**
  - `timestamp`: Timestamp, represented as the number of milliseconds since `1970-01-01`.
  - `date`: Date without timezone, in the format `YYYY-MM-DD`.
  - `time`: Time of day, represented as the number of milliseconds since midnight.
  - `datetime`: Combination of date and time.
- **Special Data Types:**
  - `uuid`: Universally unique identifier (UUID), used to generate unique primary keys.
  - `timeuuid`: Version 1 UUID, incorporating a timestamp, useful for ordering events based on generation time.

It is also possible to define new types using the following syntax:

```
CREATE TYPE type_name (
    type_field1 predefined_type1,
    type_field2 predefined_type2,
    ...
);
```

Inserting Data   To insert data into a table, values must be specified for all mandatory columns, according to the structure defined when the table was created.

```
INSERT INTO table_name (column_name1, column_name2, ...)
VALUES (value1, value2, ...);
```

Updating Data   Data in a table can be updated using the `UPDATE` clause. Columns to be updated and the conditions that must be met for the update to occur must be specified. The current value is referred to using the column name.

```
UPDATE table_name
SET column_name = new_value
WHERE condition;
```

In the `WHERE` clause, the condition can be expressed using mathematical operators (`+`, `-`, `*`, `/`), comparison operators (`<`, `>`, `<=`, `>=`, `=`, `!=`, `<>`), specific functions like `MOD()`, and logical operators:

- `AND`: returns results that satisfy both conditions.

  ```
  SELECT * FROM table_name WHERE condition1 AND condition2;
  ```

- `OR`: returns results that satisfy at least one condition.

  ```
  SELECT * FROM table_name WHERE condition1 OR condition2;
  ```

- `NOT`: returns results that do not meet the specified condition.

  ```
  SELECT * FROM table_name WHERE NOT condition;
  ```

- `IN`: checks if a value is present in a specified list of values.

  ```
  SELECT * FROM table_name WHERE column IN (value1, value2, value3);
  ```

- `CONTAINS`: checks if a set contains a specific value (used primarily with `set` or `map` columns).

  ```
  SELECT * FROM table_name WHERE column CONTAINS value;
  ```

- `CONTAINS KEY`: checks if a map contains a specific key (used with `map` columns).

  ```
  SELECT * FROM table_name WHERE column CONTAINS KEY key;
  ```

- `CONTAINS ENTRY`: checks if a map contains a specific key/value pair (used with `map` columns).

  ```
  SELECT * FROM table_name WHERE column CONTAINS ENTRY (key, value);
  ```

Deleting Data   The `DELETE` clause is used to remove data from a table. All data or a specific part of the data can be deleted based on conditions.

```
DELETE column_to_delete, ...
FROM table_name
WHERE condition;
```

The `WHERE` clause follows the same rules listed in the data update section.

Alternatively, you can change a single variable by using an `UPDATE` statement and setting the value to a default one.

Adding Columns   The structure of a table can be modified by adding new columns using the `ALTER TABLE ADD` command.

```
ALTER TABLE table_name ADD column_name data_type;
```

Since Cassandra follows a dynamic schema and not all columns are required, it is not necessary to set any value of this column for rows previously added to the database.

Dropping a Table   To completely remove a table from the keyspace, the `DROP TABLE` command can be used.

```
DROP TABLE table_name;
```

Secondary Indexes   Secondary indexes are data structures that allow access to table data using columns other than the primary key. In Cassandra, the primary key is fundamental for data partitioning and physical organization on disk. However, there are situations where querying other columns not part of the primary key may be useful. Secondary indexes help in these cases.

When a secondary index is created on a column, Cassandra creates a separate data structure that maps the values of the indexed column to their respective row identifiers. This mapping allows for quickly retrieving all rows containing a specific value in the indexed column.

```
CREATE INDEX index_name ON table_name (column_name);
```

Secondary indexes in Cassandra allow for querying non-key columns without modifying the table structure, offering flexibility in query management. However, they can degrade performance on large tables or with high write throughput, as every modification must also update the index. Additionally, they may not be suitable for large clusters or queries returning many results, requiring querying every node. In conclusion, secondary indexes are useful for extending query capabilities but must be used cautiously to avoid performance and scalability issues.

Selecting Data   SELECT queries in Cassandra are used to retrieve data from tables within a cluster. Since Cassandra is a distributed NoSQL database, its SELECT queries have some differences compared to those in traditional SQL databases.

The basic syntax of a SELECT query in Cassandra is as follows:

```
SELECT column1, column2, ...
FROM table_name
WHERE conditions
[LIMIT number];
```

Where:

- **SELECT Clause:** specifies the columns to be returned.
- **FROM Clause:** specifies the table from which to retrieve the data.
- **WHERE Clause:** defines the conditions for filtering the data. The logic of using this clause has some particularities:
    - **Partition Key:** every query using a WHERE clause must include at least the partition key to ensure optimal performance. This allows Cassandra to quickly locate the node that holds the data using hash functions and hash tables.
    - **Clustering Keys:** in addition to the partition key, clustering keys can also be used to allow Cassandra to retrieve data more quickly.
    - **Secondary Indexes:** if you need to narrow the scope of a query based on a field that is not part of the primary key, you can create secondary indexes to allow WHERE clause queries on those columns. However, using secondary indexes can negatively impact performance, so they should be used with caution.
- **LIMIT Clause:** limits the number of rows returned. Using this clause is useful to prevent retrieving too much data, which could be inefficient and resource-intensive.

It is essential to design the data model with query requirements in mind to ensure high performance and efficient query execution.

Other operations

- **Materialized Views:** they are pre-calculated views of the data present in tables, providing faster access to query results. Automatically updated by the system based on changes to the underlying data, they ensure data consistency between views and base tables. These views optimize query performance, offering flexibility in database design and supporting complex queries.

```
CREATE MATERIALIZED VIEW name_m_view AS
QUERY
PRIMARY KEY (name_key1, ...);
```

- **Configurable Consistency Levels:** determine the number of replicas that must agree on the response before an operation is considered complete. Common consistency levels include ONE (completed when at least one replica responds), QUORUM (completed when a majority of replicas respond), ALL, EACH_QUORUM, etc.
- **Batch Statements:** they in Cassandra allow executing multiple write or modification operations in a single atomic transaction, ensuring that either all operations are executed or none. They can involve one or more tables and can be configured as "unlogged" (write operations are not logged in the commit log, posing higher risks in case of data loss) or "logged" (all write operations are logged in the commit log before being applied to the actual data on disk), with significant differences in performance and data durability. Batches are useful for reducing the number of network calls and improving overall system performance, but it's important to balance data consistency and scalability needs when deciding to use them.

```
BEGIN [UNLOGGED | LOGGED] BATCH
[USING TIMESTAMP [epoch_microseconds]]
  dml_statement [USING TIMESTAMP [epoch_microseconds]];
  [dml_statement; ...]
APPLY BATCH;
```

- **LWT:** they can be used for operations that require strong consistency. You perform an LTW when using the IF command for conditionals and CAS (Compare and Set). These commands are primarily added to INSERT and SELECT operations. LWTs guarantee serializable isolation but can have a performance cost.

Especially the last 2 structures/operations will be explored in depth in the specific chapter.

Optimization

MongoDB Optimization With Indexes

Using MongoDB Indexes in Collections Indexes in MongoDB are specialized data structures that store a subset of the collection's data in a sorted format, facilitating efficient traversal and search operations. They significantly enhance query performance by allowing quick lookup, access, and updates of data. MongoDB utilizes indexes to accelerate queries, reduce disk I/O, and optimize resource utilization. Indexes support various query operations such as equality matches, range-based queries, and sorted results. Without indexes, MongoDB performs a Collection Scan, reading every document in the collection, potentially followed by in-memory sorting if required by the query. When utilizing indexes, MongoDB fetches only the documents identified by the index relevant to the query, avoiding unnecessary document reads. By default, MongoDB creates a single index per collection that includes the `_id` field. Additional indexes can be created to cover specific query patterns. However, indexes also impact write performance as they require updates whenever

documents are inserted or updated. Over-indexing can further degrade performance, necessitating periodic review and removal of redundant indexes.

The most common index types in MongoDB include Single Field Indexes, which index a single field, and Compound Indexes, which involve multiple fields in the index definition. Both types can also function as Multikey Indexes if they index arrays within documents.

Creating a Single Field Index in MongoDB To create a Single Field Index in MongoDB, the `createIndex()` method is utilized. This method specifies the field and optionally its sorting order within the index definition. For instance, the following command creates an ascending index on the `birthdate` field:

```
db.customers.createIndex({
  birthdate: 1
})
```

If searching customers by their email addresses is a common operation, creating an index on the `email` field can improve query performance significantly. Adding `{unique:true}` as an additional parameter ensures that the index enforces uniqueness on email values, preventing duplicates in the collection:

```
db.customers.createIndex({
  email: 1
},
{
  unique:true
})
```

Indexes can also be managed and viewed through MongoDB Atlas. By navigating to the Database > Collections > Indexes section, users can monitor index usage, performance metrics, and create or delete indexes as needed. Additionally, the `explain()` method can be employed to analyze query execution plans and index usage, providing insights into query optimization.

To list all indexes created on a collection, including default and user-defined indexes, the `getIndexes()` method can be used:

```
db.customers.getIndexes()
```

This command displays comprehensive information about each index present in the `customers` collection, aiding in index management and optimization efforts.

Creating a Multikey Index in MongoDB To create a multikey index in MongoDB, the `createIndex()` method is utilized with an object parameter specifying the array field and sort order. For instance, to index the `accounts` array field in ascending order:

```
db.customers.createIndex({
  accounts: 1
})
```

MongoDB imposes a constraint where only one array field can be indexed per index. If multiple fields are indexed, only one of them can be an array.

Query Optimization with `explain()` To verify whether an index is being utilized by a query, the `explain()` method is employed. By executing `explain()` on a query against a collection, MongoDB provides an execution plan detailing various stages such as IXSCAN, COLLSCAN, FETCH, and SORT. These stages indicate how the query is executed, including whether an index is utilized (IXSCAN), or if

a collection scan occurs (`COLLSCAN`), among others. Multikey indexes enhance query efficiency by creating separate index keys for each element in an array field. This optimization allows MongoDB to search for specific index keys rather than scanning entire arrays, resulting in significant performance improvements.

```
db.customers.explain().find({
  accounts: 627788
})
```

Working with Compound Indexes   Compound indexes in MongoDB involve indexing multiple fields within a document. Using the `createIndex()` method, a compound index is defined with an object containing two or more fields and their respective sort orders. Here's an example where the fields `active`, `birthdate`, and `name` are indexed with varying sort orders:

```
db.customers.createIndex({
  active: 1,
  birthdate: -1,
  name: 1
})
```

The sequence of fields in a compound index impacts query optimization. MongoDB recommends organizing fields in the order of Equality, Sort, and Range operations. For instance, queries that match on equality (`active: true`), sort by a field (`birthdate`), and apply range conditions benefit from such indexing. The optimal order of indexed fields ensures efficient query execution by leveraging the index's structure.

```
db.customers.find({
  birthdate: {
    $gte: ISODate("1977-01-01")
  },
  active: true
}).sort({
  birthdate: -1,
  name: 1
})
```

Index Utilization and Projections   Indexes in MongoDB can cover queries entirely when all necessary data is contained within the index itself, without requiring data retrieval from memory. Projections specify which fields to return in query results. By including only indexed fields in projections, MongoDB can efficiently cover queries. For example, projecting `{name: 1, birthdate:` ↪ `1, _id: 0}` ensures that MongoDB returns only the specified fields directly from the index.

```
db.customers.explain().find({
  birthdate: {
    $gte: ISODate("1977-01-01")
  },
  active: true
},
{
  name: 1,
  birthdate: 1,
  _id: 0
}).sort({
  birthdate: -1,
  name: 1
})
```

By following these indexing and querying strategies, MongoDB optimizes query performance and enhances database operations efficiency. Understanding how indexes and query execution plans interact is crucial for maximizing MongoDB's capabilities in handling large datasets and complex queries.

Deleting an Index   In MongoDB, managing indexes is crucial for optimizing query performance and minimizing operational costs associated with write operations. Indexes in MongoDB are automatically created for the `_id` field in every collection and are integral to MongoDB's internal operations; hence, they cannot be deleted.

To view all indexes associated with a collection, the `getIndexes()` method can be utilized. For example, executing `db.customers.getIndexes()` provides a comprehensive list of indexes. Conversely, the `dropIndex()` function facilitates the removal of specific indexes from a collection. This function accepts either an index key object or the name of the index as a string within its parentheses.

The decision to delete indexes should be made with careful consideration of their usage and impact on system performance. While indexes enhance query performance by reducing the number of database accesses required, they also impose overhead on write operations. Consequently, eliminating unused or redundant indexes can mitigate performance degradation in MongoDB collections.

Before deleting an index, it is imperative to ensure that the index is not actively supporting any queries. Deleting the sole index supporting a particular query can severely impair query performance. In production environments, a recommended approach involves initially hiding an index using `db.collection.hideIndex(<index>)` before completely removing it. This precautionary measure allows for the temporary concealment of an index without immediate deletion, thus minimizing disruption to query performance.

```
# Example: Deleting an index by name
db.customers.dropIndex('active_1_birthdate_ -1_name_1')

# Example: Deleting an index by key
db.customers.dropIndex({
  active: 1,
  birthdate: -1,
  name: 1
})
```

Deleting Multiple Indexes   In scenarios where multiple indexes need to be deleted simultaneously, MongoDB offers the `dropIndexes()` method. This command removes all non-default indexes from a collection, preserving the mandatory index on `_id`. Alternatively, `dropIndexes()` can be supplied with an array of index names to selectively delete specific indexes.

The process of index management can also be facilitated through MongoDB Atlas's graphical user interface (GUI), which provides a user-friendly interface for executing operations such as index deletion.

```
# Example: Deleting multiple indexes by names
db.collection.dropIndexes([
  'index1name', 'index2name', 'index3name'
])
```

In conclusion, while indexes play an important role in enhancing query performance in MongoDB, periodic review processes, deletion of redundant indexes, and careful consideration of operational implications—is essential to maintain optimal database performance and efficiency.

Cassandra Optimization

Cassandra, unlike MongoDB, does not require specific operations to optimize certain queries. This is because Cassandra assumes that the data model has already been designed optimally to ensure maximum performance. As explained in previous chapters, Cassandra's architecture is based on the distribution of data across multiple nodes and on a data model that favors rapid writes and scalable reads. In this context, performance optimization mainly depends on the initial design of the data model.

In Cassandra, the focus is on data de-normalization and creating tables in such a way that the main queries can be executed with maximum efficiency. This approach translates into:

1. **Table Design:** tables should be designed with the queries that will be executed in mind, rather than focusing on data normalization. This implies that related data is often stored together in the same table to minimize join operations.

2. **Partition Keys and Clustering:** the choice of partition keys is crucial for evenly distributing data across cluster nodes and ensuring that queries are efficient. Clustering keys are used to order the data within a partition, facilitating quick access to ordered data.

3. **Scalability and Replication:** cassandra is designed to be highly scalable, with write and read capacity distributed across multiple nodes. Data is replicated on different nodes to ensure availability and fault tolerance. The configuration of replicas can affect query performance in terms of latency and throughput.

Therefore, while MongoDB offers various query optimization techniques, such as the use of indexes and the optimization of aggregation pipelines, Cassandra relies on an initial data model design that considers query needs from the outset. This makes careful planning during the database design phase essential to achieve maximum performance in Cassandra. In practice, if a Cassandra data model needs optimization, it was likely poorly designed and needs to be redone.

Data Modeling

Data Ingestion and Processing

After having talked in length about MongoDB and Cassandra technologies, we can finally move on to presenting the case study. In particular, our project concerns the construction of an airline reservation system. To do this, we started by searching for an appropriate dataset, and in this regard we found the OSM World Airports database (https://data.opendatasoft.com/explore/dataset/osm-world-airports@babel/export/) in .csv format. This dataset represents the OpenStreetMap Export of airports in the world, including fields such as: "Geo Point", "Name", "Name (en)", "Name (fr)", "IATA code", "ICAO code", "Wikipedia id", "Wikidata id", "Website", "Phone", "Operator", "Description", "Edit in OSM", "Source", "other_tags", "Country", and "Country Code". Before we could work on it, a data processing phase was carried out using the Python programming language through the use of Jupyter Notebook and the Pandas and NumPy libraries. The data preparation and cleaning work is given by the following code:

```python
import pandas as pd
import numpy as np

# Define the file path of the dataset
file_path = '../Data/osm-world-airports@babel.csv'

# Read the CSV file into a pandas DataFrame
# Skipping lines that cause parsing errors, using Python engine
df = pd.read_csv(file_path, delimiter=';', on_bad_lines='skip', engine='python')
```

```
# Generate a random 'Size' column for each airport in the DataFrame
df['Size'] = np.random.randint(100, 500, size=len(df))

# Drop rows with missing values in critical columns ('Country', 'Country code',
    ↪ 'IATA code', 'ICAO code')
df = df.dropna(subset=['Country', 'Country code', 'IATA code', 'ICAO code'])

# Remove unnecessary columns from the DataFrame
df = df.drop(labels=["other_tags", "Description", "Wikipedia id", "Wikidata id",
    ↪ "Edit in OSM", "Source"], axis=1, inplace=False)

# Replace commas (',') in specific columns with semicolons (';') to avoid CSV
    ↪ parsing issues
df['Geo Point'] = df['Geo Point'].str.replace(',', ';')
df['Name'] = df['Name'].str.replace(',', ';')
df['Phone'] = df['Phone'].str.replace(',', ';')
df['Name (en)'] = df['Name (en)'].str.replace(',', ';')
df['Name (fr)'] = df['Name (fr)'].str.replace(',', ';')
df['Operator'] = df['Operator'].str.replace(',', ';')
df['Website'] = df['Website'].str.replace(',', ';')

# Display or further process the cleaned DataFrame
df

# Save the cleaned DataFrame to Disk
df.to_csv("../Data/Airports.csv")
```

The provided code snippet is used to process a dataset containing airport information using Python's pandas library. Initially, the script imports necessary modules, `pandas` and `numpy`, for data manipulation and generation, respectively. The dataset, located at `'../Data/osm-world-airports@babel.csv'`, is read into a pandas DataFrame, employing specific parsing parameters (`delimiter=';'`, `on_bad_lines='skip'`, `engine='python'`) to handle potential errors.

Following this, a random 'Size' column is introduced to denote the size of each airport, utilizing numpy's random integer generation. Rows with missing values in critical columns ('Country', 'Country code', 'IATA code', 'ICAO code') are subsequently dropped to ensure data integrity in our database "toy" example. Superfluous columns such as 'other_tags', 'Description', 'Wikipedia id', 'Wikidata id', 'Edit in OSM', and 'Source' are removed from the DataFrame to streamline further analysis.

To mitigate potential parsing complications in CSV files, certain columns (`'Geo Point'`, `'Name'`, `'Phone'`, `'Name (en)'`, `'Name (fr)'`, `'Operator'`, `'Website'`) are processed to replace commas with semicolons, enhancing data consistency and facilitating seamless parsing.

The cleaned DataFrame (`df`) represents structured airport data, where:

- **IATA code** refers to the unique three-letter code assigned by the International Air Transport Association to identify airports.
- **ICAO code** denotes the four-letter code designated by the International Civil Aviation Organization for individual airports and airspace globally.
- **Country** signifies the country in which each airport is located.
- **Country code** represents the standard two-letter code assigned to countries according to ISO 3166-1 alpha-2.
- **Name** denotes the official name of the airport.
- **Size** indicates a randomly generated numerical value representing the size of each airport.

The final processed DataFrame (`df`) can be displayed for examination or utilized for subsequent

analytical tasks as per specific project requirements.

Assumptions

At this point, it is good to make some assumptions about what our system must be able to perform. The system will simulate a website where, without needing to log in, people can connect to book an airline flight. The operations that users will carry out on the website will be simulated by the Main method of the Java code, which we will show later. In the form of saving log files, the system will have to simulate and test various situations and operations performed on our database. First of all, flight retrieval will be tested: the user will enter the airport from which he intends to depart, and the system will return which flights will depart from that airport. Once done, the user will be able to choose one of the flights among those available, effectively choosing an air route from the chosen airport to a destination airport.

At this point, a testing seats retrieval will take place: given the selected route, the system will return the number of available seats, and the codes associated with the individual seats. The system will then test a concurrent transaction, in which two people will try to book the same seat at the same time. This will highlight the management of competing writes with the various technologies used, and the fact that only one of the two people will be able to get the desired place, while the other will receive an error code. After that the system will also test a non-competitive transaction, in which a person will try to get a seat without having competition from other users. This operation must be successful. We will also test the budget logic, according to which a user must have enough money to be able to get on a flight, and therefore a test will be done on insufficient funds behavior. Not only this, the logs will also return the balance of people before and after booking, to show that everything is happening correctly.

Creating the Data Model and Transactions in MongoDB

Let's now move on to the creation of the Data Model, which is configured through the use of various Java Classes, which interact with our Database. The class from which it all begins is the `Connection.java` class, whose code is shown below. We already talked in great detail about this class in this document, but it will be shown again in order to aid the reader in better understanding the structure of the overall code without having to refer to previous sections. To be thorough, several more comments were added to better understand the code.

```java
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.bson.Document;

import java.util.ArrayList;
import java.util.List;

/**
 * The Connection class provides a singleton connection to a MongoDB instance.
 * It allows retrieving the list of databases from the MongoDB server and
 * accessing the underlying {@link MongoClient} instance.
 * This class is thread-safe and ensures that only one instance of the connection
 * is created (singleton pattern).
 *
 * @author Andrea Moleri
 * @version 1.0
 * @since 2024-06-23
 */
public class Connection {
    private static Connection instance;
    private final MongoClient mongoClient;
```

```java
    /**
     * Private constructor to prevent direct instantiation.
     * Creates a new Connection instance with the specified connection string.
     *
     * @param connectionString the MongoDB connection string
     */
    private Connection(String connectionString) {
        this.mongoClient = MongoClients.create(connectionString);
    }

    /**
     * Returns the singleton instance of the Connection class, creating it if
         ↪ necessary.
     * This method is synchronized to ensure thread safety.
     *
     * @param connectionString the MongoDB connection string
     * @return the singleton instance of Connection
     */
    public static synchronized Connection getInstance(String connectionString) {
        if (instance == null) {
            instance = new Connection(connectionString);
        }
        return instance;
    }

    /**
     * Retrieves and prints the list of databases available on the connected
         ↪ MongoDB instance.
     * This method fetches the list of databases from the MongoDB server and prints
     * their details in JSON format to the standard output.
     */
    public void listDatabases() {
        // Retrieve the list of databases
        List<Document> databases = mongoClient.listDatabases().into(new
            ↪ ArrayList<>());

        // Print information about each database
        databases.forEach(db -> System.out.println(db.toJson()));
    }

    /**
     * Returns the MongoClient instance used by this Connection.
     * This method provides access to the underlying {@link MongoClient} instance,
     * allowing for further MongoDB operations to be performed.
     *
     * @return the MongoClient instance
     */
    public MongoClient getMongoClient() {
        return this.mongoClient;
    }
}
```

As already explained previously, the class uses the Singleton design pattern to establish a connection with the MongoDB database through the use of a Connection String. The reason why it is implemented with a Singleton is to save resources, following the best practices recommended by the MongoDB documentation. After doing this, we generate a `Modeling.java` class, whose code is as follows:

```java
/**
 * This Java application reads data from a CSV file containing airport information,
 * parses and filters the data, and stores it in a MongoDB database. It also
     ↪ generates
 * flights for each airport based on its size and stores them in the same database.
```

```java
 *
 * The main method initiates the process:
 * - Connects to MongoDB using a provided connection string.
 * - Reads airport data from a specified CSV file.
 * - Parses and filters the data into MongoDB Documents, excluding specific fields.
 * - Stores each airport Document in a MongoDB collection.
 * - Generates flights for each airport based on its size and available seats.
 * - Stores generated flight information in MongoDB, associating them with
 *     ↪ respective airports.
 *
 * The class utilizes external libraries such as MongoDB Java Driver and handles
 *     ↪ various
 * exceptions including I/O errors and data parsing issues.
 *
 * @author Andrea Moleri
 * @version 1.0
 * @since 2024-06-23
 */
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import org.bson.types.ObjectId;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Modeling {

    /**
     * Main method that orchestrates the import of airport data from a CSV file
     *     ↪ into MongoDB.
     * It establishes a connection, processes the CSV data, and generates flights
     *     ↪ for each airport.
     * @param args Command-line arguments (not used)
     */
    public static void main(String[] args) {
        String csvFile = "Data/Airports.csv"; // Path to the CSV file
        String dbName = "Airports"; // MongoDB database name
        String collectionName = "airportCollection"; // MongoDB collection name

        String connectionString =
            ↪ "mongodb+srv://admin:admin@learningmongodb.hikoksa.mongodb.net" +
            ↪ "/?retryWrites=true&w=majority&appName=LearningMongoDB";

        try {
            // Obtain MongoClient and MongoDB/MongoCollection
            MongoClient mongoClient =
                ↪ Connection.getInstance(connectionString).getMongoClient();
            MongoDatabase database = mongoClient.getDatabase(dbName);
            MongoCollection<Document> collection =
                ↪ database.getCollection(collectionName);

            BufferedReader reader = new BufferedReader(new FileReader(csvFile));
            String line;
            boolean isFirstLine = true; // Flag to check if it's the first line
```

63

```java
String[] headers = null; // Array to hold column headers

List<Document> airports = new ArrayList<>(); // List to hold all
    ↪ airports for flight generation

while ((line = reader.readLine()) != null) {
    String[] fields = line.split(",", -1);

    if (isFirstLine) {
        isFirstLine = false;
        // Save headers and replace spaces with underscores
        headers = new String[fields.length];
        for (int i = 0; i < fields.length; i++) {
            headers[i] = fields[i].trim().replace(" ", "_");
        }
        continue; // Skip header line
    }

    // Create a new Document to store in MongoDB
    Document airportDoc = new Document();

    // Loop through the fields array and add non-null fields to the
        ↪ Document
    for (int i = 0; i < fields.length && i < headers.length; i++) {
        String fieldName = headers[i].trim();
        String fieldValue = fields[i].trim().replaceAll("\"\"", "\"");
        if (!fieldName.isEmpty() && !fieldValue.isEmpty()) {
            // Exclude "Edit_in_OSM" and "other_tags" fields
            if (!fieldName.equals("Edit_in_OSM") &&
                ↪ !fieldName.equals("other_tags")) {
                if (fieldName.equals("Size")) {
                    try {
                        airportDoc.append(fieldName,
                            ↪ Integer.parseInt(fieldValue));
                    } catch (NumberFormatException e) {
                        System.err.println("Error parsing 'Size' field
                            ↪ to integer: " + fieldValue);
                    }
                } else {
                    airportDoc.append(fieldName, fieldValue);
                }
            }
        }
    }

    // Insert Document into MongoDB
    collection.insertOne(airportDoc);

    // Add airportDoc to the list for generating flights later
    airports.add(airportDoc);
}

// After inserting all airports, generate flights for each airport
for (Document airportDoc : airports) {
    generateAndInsertFlights(airportDoc, collection, airports);
}

System.out.println("Data imported successfully into MongoDB");

// Close resources
reader.close();
mongoClient.close(); // Close MongoDB connection
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }


    /**
     * Generates flights for an airport based on its size and inserts them into
        ↪ the database.
     * @param airportDoc Document representing the airport for which flights are
        ↪ generated
     * @param collection MongoDB collection where flights are stored
     * @param airports List of all airports for generating destination airports
     */
    private static void generateAndInsertFlights(Document airportDoc,
        ↪ MongoCollection<Document> collection, List<Document> airports) {
        int airportSize = airportDoc.getInteger("Size", 0); // Airport size
        List<Document> flights = generateFlights(airportSize, airports,
            ↪ airportDoc);

        // Calculate max flights based on airport size
        int maxFlights = airportSize / 100;
        int totalSeats = 0;

        // Calculate total seats from existing flights
        for (Document flight : flights) {
            totalSeats += flight.getInteger("Number_of_Seats", 0);
        }

        // Reduce flights if total seats exceed airport capacity
        if (totalSeats > airportSize) {
            // Remove flights until total seats fit within airport capacity
            while (totalSeats > airportSize && flights.size() > 0) {
                Document lastFlight = flights.remove(flights.size() - 1);
                totalSeats -= lastFlight.getInteger("Number_of_Seats", 0);
            }
        }

        // Update flights array in airportDoc
        airportDoc.put("Flights", flights);

        // Replace the document in MongoDB collection
        collection.replaceOne(new Document("_id", airportDoc.getObjectId("_id")),
            ↪ airportDoc);
    }


    /**
     * Generates a list of flights for an airport based on its size.
     * @param airportSize Size of the airport
     * @param airports List of all airports for selecting destination airports
     * @param currentAirport Document representing the current airport
     * @return List of generated flights
     */
    private static List<Document> generateFlights(int airportSize, List<Document>
        ↪ airports, Document currentAirport) {
        List<Document> flights = new ArrayList<>();
        Random random = new Random();

        // Generate flights
        for (int i = 0; i < 5; i++) { // Generate 5 flights per airport (adjust as
            ↪ needed)
            // Select a random destination airport (ensure it's not the same as
```

```java
                ↪ current airport)
        Document destinationAirport = getRandomDestinationAirport(airports,
            ↪ currentAirport);

        // Generate random future date between 1 and 10 days from today
        LocalDate currentDate = LocalDate.now();
        int daysToAdd = random.nextInt(10) + 1;
        LocalDate futureDate = currentDate.plusDays(daysToAdd);
        String dayString =
            ↪ futureDate.format(DateTimeFormatter.ofPattern("yyyy-MM-dd"));



        // Generate random time
        LocalTime randomTime = LocalTime.of(random.nextInt(24),
            ↪ random.nextInt(60));
        String hourString =
            ↪ randomTime.format(DateTimeFormatter.ofPattern("HH:mm"));

        // Generate random duration between 1 and 14 hours
        int durationHours = random.nextInt(14) + 1;
        String durationString = durationHours + " hours";

        // Array of actual operators
        String[] operators = {"Ryanair", "Lufthansa", "EasyJet", "British
            ↪ Airways", "Air France"};

        // Generate random operator
        String operator = operators[random.nextInt(operators.length)];

        // Generate random price per person between 39 and 499
        int pricePerPerson = random.nextInt(461) + 39; // 39 to 499

        Document flight = new Document();
        flight.append("ID", new ObjectId().toString())
              .append("Number_of_Seats", 100)
              .append("Day", dayString)
              .append("Hour", hourString)
              .append("Operator", operator)
              .append("Duration", durationString)
              .append("Price_per_Person", pricePerPerson);

        // Add destination reference as ObjectId
        flight.append("Destination", destinationAirport.getObjectId("_id"));

        // Generate seats for this flight
        List<Document> seats = generateSeats(100); // Assume 100 seats per
            ↪ flight
        flight.append("Seats", seats);

        flights.add(flight);
    }

    return flights;
}

/**
 * Retrieves a random destination airport document from the list, ensuring
     ↪ it's different from the current airport.
 * @param airports List of all airports
 * @param currentAirport Document representing the current airport
 * @return Random destination airport Document
 * @throws IllegalStateException If the airports list is empty
```

```java
     */
    private static Document getRandomDestinationAirport(List<Document> airports,
        ↪ Document currentAirport) {
        Random random = new Random();
        Document destinationAirport = null;

        // Keep trying until a different airport than currentAirport is found
        while (true) {
            if (airports.isEmpty()) {
                throw new IllegalStateException("Airport list is empty, cannot
                    ↪ select destination.");
            }
            int index = random.nextInt(airports.size());
            destinationAirport = airports.get(index);
            String id_current = currentAirport.getObjectId("_id");
            if (!destinationAirport.getObjectId("_id").equals(id_current)) {
                break;
            }
        }

        return destinationAirport;
    }

    /**
     * Generates a list of seat documents for a flight.
     * @param numberOfSeats Number of seats to generate
     * @return List of generated seat documents
     */
    private static List<Document> generateSeats(int numberOfSeats) {
        List<Document> seats = new ArrayList<>();
        String[] seatLetters = {"A", "B", "C", "D", "E", "F"};
        int seatNumber = 1;

        for (int i = 0; i < numberOfSeats; i++) {
            String seatID = seatNumber + seatLetters[i % 6]; // Cycle through A to
                ↪ F
            Document seat = new Document();
            seat.append("Status", "Vacant")
                .append("ID", seatID)
                .append("Name", "")
                .append("Surname", "")
                .append("Document_Info", "")
                .append("Date_of_Birth", "")
                .append("Balance", 0);

            seats.add(seat);
            if ((i + 1) % 6 == 0) { // Move to the next row of seat numbers (1A,
                ↪ 2A, ...)
                seatNumber++;
            }
        }

        return seats;
    }
}
```

The `Modeling.java` class is designed to import airport data from a CSV file into a MongoDB database and generate associated flight information for each airport. Here's an in-depth explanation of the code:

The `main` method begins by setting up the necessary parameters such as the CSV file path, the database name, and the collection name. It then creates a connection to the MongoDB database

using the `Connection` class discussed earlier. This is crucial as it ensures the MongoDB client is correctly instantiated, following the Singleton pattern to manage resources efficiently. The CSV file, containing airport data, is read line-by-line using a `BufferedReader`. The first line, which consists of the headers, is processed separately to create an array of header names. These headers are cleaned by trimming whitespace and replacing spaces with underscores, ensuring they are suitable for use as field names in MongoDB documents. For each subsequent line in the CSV file, a new `Document` object is created. This document will represent an airport in the MongoDB collection. The fields of each line are split and processed according to the headers array. Specific fields, such as `Size`, are converted to integers where appropriate, while others are added directly to the `Document`.

Each constructed `Document` is inserted into the specified MongoDB collection. Additionally, these documents are stored in a list called `airports` for later use in flight generation. After all airport data has been inserted into MongoDB, the program proceeds to generate flight data for each airport. This is done through the `generateAndInsertFlights` method. This method takes three parameters: the airport document, the MongoDB collection, and the list of all airport documents. It starts by determining the size of the airport, which influences the number of flights to be generated.

The `generateFlights` method is called to create a list of flight documents for the airport. This method generates random details for each flight, including:

- **Destination Airport:** selected randomly from the list of all airports, ensuring it is different from the current airport.
- **Flight Date and Time:** a random date within the next 10 days and a random time.
- **Flight Duration:** a random duration between 1 and 14 hours.
- **Operator:** randomly chosen from a predefined list of airline operators.
- **Price per Person:** a random price between 39 and 499.

Each flight document includes an array of seat documents, generated by the `generateSeats` method. Each seat is initialized with a status of "Vacant" and a unique seat ID. The total number of seats across all flights is calculated and compared to the airport's capacity (determined by its size). If the total seats exceed this capacity, the program reduces the number of flights by removing the last flight until the total seats are within the allowed limit. It should be noted that, for aesthetic and demonstration purposes, each flight has a fixed standard of 100 seats available. Therefore, for example, if an airport has a "size" parameter of value 432, that airport will be able to fly a maximum of 4 flights, with 100 seats each. The remaining 32 seats will be ignored, as it is not sufficient to justify the presence, -and departure-, of an additional plane. The list of flight documents is added to the airport document under the "Flights" field. This updated document then replaces the original document in the MongoDB collection, ensuring the database reflects the newly generated flight data. Finally, the `BufferedReader` and the MongoDB client are closed to free up resources.

Throughout the `main` method and other parts of the code, exception handling is implemented to manage potential I/O errors, data parsing issues, and other exceptions that may occur during execution. During the process, features like MongoDB polymorphism especially help in converting the .csv file to the database. Suffice it to say that in the original .csv there were some null values (on secondary fields, as those on primary fields were removed through the use of .dropna() to avoid problems). MongoDB handles these fields gracefully: for example, if one airport has a phone number and the other doesn't, only the airport document with the phone number will actually have the relevant field added as an attribute. These additional attributes are useful and nice to have, but their absence does not cause any problems or exceptions.

In general, what the Java class just shown does is transform the .csv resulting from the processing carried out using the Jupyter Notebook shown previously into a MongoDB collection called

`airportCollection`. Below is a short-form example of the structure of a single document within the collection, converted to a JSON file for visual simplicity.

```
[{
    "_id": {
        "$oid": "6677f94b7acf35542d8ee8b2"
    },
    "Geo_Point": "51.1075552; 16.8756697",
    "Name": "Port Lotniczy Wroclaw",
    "Name_(en)": "Wroclaw Airport",
    "Name_(fr)": "Aeroport de Wroclaw-Nicolas Copernic",
    "IATA_code": "WRO",
    "ICAO_code": "EPWR",
    "Operator": "Port Lotniczy Wroclaw SA",
    "Country": "Poland",
    "Country_code": "PL",
    "Size": 372,
    "Flights": [
        {
            "ID": "6677f9697acf35542d8eeb54",
            "Number_of_Seats": 100,
            "Day": "2024-06-25",
            "Hour": "04:39",
            "Operator": "Air France",
            "Duration": "9 hours",
            "Price_per_Person": 352,
            "Destination": {
                "$oid": "6677f9567acf35542d8ee985"
            },
            "Seats": [
                {
                    "Status": "Vacant",
                    "ID": "1A",
                    "Name": "",
                    "Surname": "",
                    "Document_Info": "",
                    "Date_of_Birth": "",
                    "Balance": 0
                },
                {
                    "Status": "Vacant",
                    "ID": "1B",
                    "Name": "",
                    "Surname": "",
                    "Document_Info": "",
                    "Date_of_Birth": "",
                    "Balance": 0
                },
                ...
            ]
        }
    ]
}]
```

As can be seen, several best practices among those cited in the report are applied, among which the use of an identification field to represent the flights stands out, in such a way that they refer to another document, rather than containing all the information in the same document. This is an implementation of the **reference relationship** type, which is particularly useful in MongoDB for handling related but distinct pieces of data. In this type of relationship, documents in one collection contain references to documents in another collection. This method is ideal for representing relationships where the related data is extensive and needs to be managed separately. In the given example, each flight document contains a reference to a `Destination` document by using an object

ID (`$oid`). This allows the database to keep flight details separate from destination details, enabling more efficient data management and scalability. What follows are some best practices that can be observed in the document:

1. **Using Identification Fields:** each flight has an `ID` field, which is a reference to another document. This practice is crucial for maintaining normalization within the database and avoiding redundancy. By using IDs to link related documents, the database stays more manageable and less prone to inconsistencies.

2. **Embedding Documents for Nested Data:** the `Seats` array within each `Flight` document is an example of embedding. Embedding is a MongoDB best practice for including related data directly within a document when it is expected that the embedded data will always be accessed along with the parent document. This is useful for performance reasons as it allows retrieving all related data in a single query.

3. **Using Arrays for Related Entities:** the `Flights` field is an array that holds multiple flight objects. This is a practical approach in MongoDB for one-to-many relationships, where a single airport can have multiple flights. This structure allows efficient queries and updates, as all related flights are stored together.

4. **Field Naming Conventions:** consistent and clear field names (such as `Geo_Point`, `Name_(en)`, and `IATA_code`) make the data easier to understand and work with. MongoDB documents should use meaningful names to ensure clarity and facilitate easier maintenance and querying.

5. **Keeping Documents Lightweight:** by referring to other documents (such as `Destination`), the main document remains lightweight. This ensures faster read and write operations and more efficient use of memory. Storing only essential and relevant information in a document helps in maintaining optimal performance.

By implementing these practices, the database is kept as lightweight and efficient as possible while still maintaining a structure that is easy for programmers to work with and understand.

Creating the Data Model and Transactions in Cassandra

In the context of Cassandra, maximizing database efficiency requires a data model design approach focused primarily on anticipated queries. This means the data structure must be conceived to facilitate and optimize query operations, even if this results in an increase in write operations, which are nonetheless executed almost instantaneously thanks to Cassandra's features. Therefore, according to this methodology, it is necessary to create tables that directly reflect the anticipated query types, while ensuring an even distribution of data across the cluster nodes to prevent overloading specific nodes and to guarantee effective, seamless scalability. To successfully implement this approach, it is essential to follow a series of steps:

Define the query flow The first fundamental step is to analyze the application's operational flow. This involves an in-depth study of the operations the user will perform during the application's use that involve the database. Each interaction with the database must be mapped and understood in detail. Beyond understanding specific operations, it is crucial to understand data paths and interactions between various components of the application. This study involves tracing how data flows through the application, from creation to destruction. Interactions between different modules of the application can reveal data access patterns that significantly impact database performance and are useful when it is necessary to ensure consistency between data copies in various tables. Another essential aspect is to clearly specify the application's performance and scalability requirements. This includes defining key metrics such as the acceptable response time for read and

write operations, the transaction volume the system must handle per second, and the requirements for availability and fault tolerance. In the model described at the beginning of the chapter, the following operational flow was hypothesized: 1. The user connects to the application. 2. The user enters the departure airport and, if necessary, the destination airport (if omitted, it is as if "Any destination" is entered) and the temporal information. The goal is to obtain a screen similar to typical flight booking sites, with a list of all available flights based on the search criteria provided by the user. 3. After identifying the desired flight, the user views information about available seats. Ideally, this phase presents a screen with a simplified representation of the plane, where occupied seats are marked in red and available seats in green. 4. Once a seat is selected, the user enters their personal data. If the user's financial availability is greater than or equal to the flight price and there are no competition conflicts, the seat is booked. Additionally, at any time, the user might want to obtain information about a single airport.

Design the necessary queries   After defining the operational flow, the next step is to carefully examine the read and write operations required by the application on the database. Each operation must be analyzed to understand exactly how it will be executed, how often, and what data will be involved. The most frequent operations must then occur as quickly as possible. For example, if an e-commerce application frequently requires access to product details and customer orders, these queries must be optimized to ensure quick response times. It is important to identify the search criteria, aggregations, and filtering operations necessary to meet the application's functional requirements. This helps to design queries that are efficient and fast. Typically, fields that need frequent filtering should be indexed or part of the table's primary key. The main queries identified from the model and the operational flow are, in pseudo-natural language: - "Given the IATA code of an airport, return all information about that airport." - "Given the IATA code of an airport, return a list of all flights departing from that airport." - "Given two IATA codes of airports, respectively the departure and arrival airport, return a list of all flights on that route." - "Given two IATA codes of airports, respectively the departure and arrival airport, and the day, return a list of all flights on that route during that time period." - "Given two IATA codes of airports, respectively the departure and arrival airport, the day and the hour, return a list of all flights on that route during that time period." - "Given the code of a flight, return a list of all seats on that flight." - "Given two codes, respectively the flight code and the seat code, return the status of the seat on that flight." - "Given two codes, respectively the flight code and the seat code, and the user information, insert the user data and change the seat status from 'Vacant' to 'Occupied.'"

Create the table structure   Designing the database tables effectively, considering the identified queries, data relationships, and access frequency, is fundamental. The table structure should optimize the read and write operations needed to satisfy common queries. Frequently accessed data together should be stored in a way that minimizes read operations. Crucial aspects for a good table structure are:

- **Denormalization:** evaluating data denormalization can significantly improve query performance. In many cases, a bit of redundancy can reduce the number of necessary joins, thus speeding up read operations.
- **Primary Keys:** carefully defining the primary keys for each table is crucial, as they ensure not only the uniqueness of data but also its distribution among nodes. Composite keys can be used to support complex queries when necessary. Composite keys organize the data in a way that allows efficient query execution, reducing the number of read operations.
- **Data Type:** choosing the most suitable data types to represent information in the database is essential for ensuring efficiency and speed. This means choosing data types that minimize storage space and maximize read and write operation speed. Native Cassandra data types can be used, as well as collection data types (such as sets, lists, and maps) for efficient modeling

71

of complex data. These data types can greatly improve query performance by enabling normalization and eliminating joins between tables.

In the context described at the beginning of the chapter and based on the operational flow and identified queries, three tables have been identified:

```
CREATE TABLE IF NOT EXISTS airport (
    Name_en text,
    Country text,
    Website text,
    Geo_Point text,
    ICAO_code text,
    Country_code text,
    Phone text,
    Name_fr text,
    Name text,
    IATA_code text,
    Size int,
    Operator text,
    PRIMARY KEY (IATA_code)
);
```

This favors the search for the airport only through its unique IATA code. The main operations, excluding INSERTs, planned for this table are:

- "Given the IATA code of an airport, return all information about that airport."

```
SELECT * FROM airport WHERE IATA_code = 'v';
```

```
CREATE TABLE IF NOT EXISTS flight (
    ID text,
    Number_of_Seats int,
    Day date,
    Hour time,
    Operator text,
    Duration text,
    Price_per_Person float,
    Destination text,
    Departure text,
    PRIMARY KEY ((Departure), Destination, Day, Hour, ID)
);
```

The combination of Departure, Destination, Day, Hour, and ID ensures that each flight is uniquely identified and, most importantly, favors search using only the departure airport code. Adding the ID field allows for distinguishing flights with the same route and time (a very unlikely but possible case). Any variation in the fields that make up the primary key identifies a distinct flight. The main operations, excluding INSERTs, planned for this table are: - "Given the IATA code of an airport, return a list of all flights departing from that airport."

```
SELECT * FROM flight WHERE departure = 'v';
```

- "Given two IATA codes of airports, respectively the departure and arrival airport, return a list of all flights on that route."

```
SELECT * FROM flight WHERE departure = 'v1' AND destination = 'v2';
```

- "Given two IATA codes of airports, respectively the departure and arrival airport, and the day, return a list of all flights on that route during that time period."

```
SELECT * FROM flight WHERE departure = 'v1' AND destination = 'v2' AND day = 'v3';
```

- "Given two IATA codes of airports, respectively the departure and arrival airport, the day and the hour, return a list of all flights on that route during that time period."

```
SELECT * FROM flight WHERE departure = 'v1' AND destination = 'v2' AND day = 'v3'
    ↪ AND hour = 'v4';
```

```
CREATE TABLE IF NOT EXISTS seat (
    Flight text,
    Status text,
    ID text,
    Name text,
    Surname text,
    Document_Info text,
    Date_of_Birth date,
    Balance float,
    PRIMARY KEY ((Flight), ID)
);
```

By including only the flight code in the partition key, it is possible to quickly obtain the entire list of seats for a flight. Including the ID field as a clustering key also ensures cases where a single seat is analyzed. The main operations, excluding INSERTs, planned for this table are:

- "Given the code of a flight, return a list of all seats on that flight."

```
SELECT * FROM seat WHERE flight = 'v';
```

- "Given two codes, respectively the flight code and the seat code, return the status of the seat on that flight."

```
SELECT status FROM seat WHERE flight = 'v1' AND id = 'v2';
```

- "Given two codes, respectively the flight code and the seat code, and the user information, insert the user data and change the seat status from 'Vacant' to 'Occupied.'"

```
UPDATE seat SET status = 'Occupied', balance = 'v1', date_of_birth = 'v2',
    ↪ document_info = 'v3', name = 'v4', surname = 'v5' WHERE flight = 'v6' AND
    ↪ id = 'v7'
```

In conclusion, the Javascript used to communicate with the database has been updated (the UPDATE operation will be covered in the relevant chapter). Specifically, in the main method, after establishing the connection, a query is executed to obtain the list of tables. Subsequently, the

three previously described operations to create the tables are executed. Once the tables have been created, the initial query is executed again to obtain the updated list of tables. By comparing the results of the two queries, it is possible to determine which tables have been created. For each created table, the `INSERT` method is called to populate it. The `INSERT` method reads data from a `.txt` file having the same name as the table to be populated, where each row of the file corresponds to a row in the Cassandra table, and loads them into the database using the `INSERT` operation. Each row is a list of data that follows the order of fields as specified in the columns section of the `INSERT`. This method was designed to maintain consistency with the data created during the MongoDB phase; in fact, the Mongo database was previously exported to `JSON` files and processed to obtain the three distinct files (one for each table). The data is structured into classes: `Seat`, `Flight`, and `Airport`. Each class simply contains the various columns that define the respective table, getter and setter methods, and the toString method.

Then, various query tests are performed. Specifically, the methods: - `getFlightsFromAirport`: this method retrieves a list of flights departing from a specific airport. It uses the airport code as a filter in the Cassandra query. - Creates a `SimpleStatement` query to select all flights from the `flight` table where the `departure` column is equal to the provided airport code. - Calls the `getFlights` method to execute the query and return the results as a list of `Flight` objects. - `getAirportFromIATA`: this method retrieves information about an airport given its IATA code. - Creates a `SimpleStatement` query to select all airport details from the `airport` table where the `IATA_code` column is equal to the provided IATA code. - Executes the query and retrieves the first row of the results. - Extracts the airport details from the row and creates an `Airport` object. - Returns the `Airport` object. - `getFlightsFromDepartureAndDestination`: this method retrieves a list of flights departing from a specific airport and arriving at a specific airport. - Creates a `SimpleStatement` query to select all flights from the `flight` table where the `departure` and `destination` columns match the provided codes. - Calls the `getFlights` method to execute the query and return the results as a list of `Flight` objects. - `getFlights`: this method executes a Cassandra query to retrieve a list of flights and maps the results to `Flight` objects. - Executes the query. - Iterates over the results and for each row, creates a `Flight` object with the flight details. - Adds the `Flight` object to the list of flights. - Returns the list of flights. - `getSeats`: this method retrieves a list of seats for a given flight. - Creates a `SimpleStatement` query to select all seats from the `seat` table where the `flight` column is equal to the provided flight code. - Executes the query. - Iterates over the results and for each row, creates a `Seat` object with the seat details. - Adds the `Seat` object to the list of seats. - Returns the list of seats. - `getAvailableSeats`: this method retrieves a list of available seats for a given flight. - Calls the `getSeats` method to get all seats for the specified flight. - Filters the seats to include only those with the status `Vacant`. - Returns the list of available seats. - `getSeatStatus`: this method retrieves the status of a specific seat for a specific flight. - Creates a `SimpleStatement` query to select the seat status from the `seat` table where the `flight` and `id` columns match the provided values. - Executes the query and retrieves the first row of the results. - If the row exists, returns the seat status. - If the row does not exist, throws an exception. - `getFirstAvailableFlight`: this method retrieves the first available flight from a departure airport to a destination airport. - Calls the `getFlightsFromDepartureAndDestination` method to get all matching flights. - If no flights are found, throws an exception. - Returns the first flight from the list of flights. - `getFirstAvailableSeat`: this method retrieves the first available seat for a given flight. - Calls the `getAvailableSeats` method to get all available seats for the specified flight. - If no seats are available, throws an exception. - Returns the first available seat from the list of seats.

```
package com.example.cassandra;

// Import necessary classes from DataStax driver and AWS MCS for authentication
import com.datastax.driver.core.*;
import software.aws.mcs.auth.SigV4AuthProvider;
```

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class App
{
static Session session = null;

    // Method to get flights from a specific airport
    public static List<Flight> getFlightsFromAirport(String airportCode){
        // Create a query to select all flights from a specific departure airport
        SimpleStatement query = new SimpleStatement("SELECT * FROM flight WHERE
            ↪ departure='" + airportCode + "';");
        return getFlights(query);
    }

    // Helper method to execute a query and return a list of Flight objects
    private static List<Flight> getFlights(SimpleStatement query) {
        ResultSet rs = session.execute(query);
        List<Flight> flightList = new ArrayList<>();

        // Iterate through the result set and create Flight objects
        for (Row row : rs) {
            String departure = row.getString("departure");
            String day = "" + row.getDate("day");
            String destination = row.getString("destination");
            String hour = "" + row.getTime("hour");
            String id = row.getString("id");
            String duration = row.getString("duration");
            int number_of_seats = row.getInt("number_of_seats");
            String operator = row.getString("operator");
            float price_per_person = row.getFloat("price_per_person");

            Flight flight = new Flight(departure, day, destination, hour, id,
                ↪ duration, number_of_seats, operator, price_per_person);
            flightList.add(flight);
        }

        return flightList;
    }

    // Method to get flights based on departure and destination airports
    public static List<Flight> getFlightsFromDepartureAndDestination(String
        ↪ departureCode, String destinationCode) {
        // Create a query to select flights based on departure and destination
        SimpleStatement query = new SimpleStatement("SELECT * FROM flight WHERE
            ↪ departure='" + departureCode + "' AND destination='" +
            ↪ destinationCode + "';");
        return getFlights(query);
    }

    // Method to get airport details based on IATA code
    public static Airport getAirportFromIATA(String airportCode) {
        // Create a query to select an airport based on its IATA code
        SimpleStatement query = new SimpleStatement("SELECT * FROM airport WHERE
            ↪ IATA_code='" + airportCode + "';");
        ResultSet rs = session.execute(query);

        Row row = rs.one();
        Airport airport = null;
        // From result set to Airport
```

```java
        if (row != null) {
            String iata_code = row.getString("iata_code");
            String country = row.getString("country");
            String country_code = row.getString("country_code");
            String geo_point = row.getString("geo_point");
            String icao_code = row.getString("icao_code");
            String name = row.getString("name");
            String name_en = row.getString("name_en");
            String name_fr = row.getString("name_fr");
            String operator = row.getString("operator");
            String phone = row.getString("phone");
            int size = row.getInt("size");
            String website= row.getString("website");

            airport = new Airport(iata_code, country, country_code, geo_point,
                ↪ icao_code, name, name_en, name_fr, operator, phone, size,
                ↪ website);
        }

        return airport;
    }

    // Method to get seats for a specific flight
    public static List<Seat> getSeats(String flightCode) {
        // Create a query to select all seats for a specific flight
        SimpleStatement query = new SimpleStatement("SELECT * FROM seat WHERE
            ↪ flight='" + flightCode + "';");
        ResultSet rs = session.execute(query);

        List<Seat> seatList = new ArrayList<>();

        // Iterate through the result set and create Seat objects
        for (Row row : rs) {
            String flight = row.getString("flight");
            String id = row.getString("id");
            float balance = row.getFloat("balance");
            String date_of_birth = "" + row.getDate("date_of_birth");
            String document_info = row.getString("document_info");
            String name = row.getString("name");
            String status = row.getString("status");
            String surname = row.getString("surname");

            Seat seat = new Seat(flight, id, balance, date_of_birth,
                ↪ document_info, name, status, surname);
            seatList.add(seat);
        }

        return seatList;
    }

    // Method to get available (vacant) seats for a specific flight
    public static List<Seat> getAvailableSeats(String flightCode) {
        List<Seat> seats = getSeats(flightCode);
        List<Seat> availableSeats = new ArrayList<>();

        // Filter seats to only include those that are vacant
        for (Seat seat : seats) {
            if(seat.getStatus().equals("Vacant")) {
                availableSeats.add(seat);
            }
        }

        return availableSeats;
```

```java
}

// Method to get the status of a specific seat on a flight
public static String getSeatStatus(String flightCode, String idSeat) throws
    ↪ Exception {
    // Create a query to select a seat based on flight and seat ID
    SimpleStatement query = new SimpleStatement("SELECT * FROM seat WHERE
        ↪ flight='" + flightCode + "' AND id='"+ idSeat +"';");
    ResultSet rs = session.execute(query);

    Row row = rs.one();

    if (row != null) {
        return row.getString("status");
    }

    throw new Exception("This seat does not exist");
}

// Method to get the first available flight based on departure and destination
    ↪ airports
public static Flight getFirstAvailableFlight(String departureCode, String
    ↪ destinationCode) throws Exception{
    List<Flight> flights =
        ↪ getFlightsFromDepartureAndDestination(departureCode,
        ↪ destinationCode);

    if (flights.isEmpty()) {
        throw new Exception("No flights found");
    }

    return flights.get(0);
}

// Method to get the first available seat on a specific flight
public static Seat getFirstAvailableSeat(String flightCode) throws Exception {
    List<Seat> seats = getAvailableSeats(flightCode);

    if (seats.isEmpty()) {
        throw new Exception("No seats available");
    }

    return seats.get(0);
}

// Method to insert data into tables from text files
public static void insert(String table) {
    List<List<String>> dati = new ArrayList<>();

    String[] fileName = {"airport.txt", "flights.txt", "seats.txt"};

    for (String s: fileName) {
        try (BufferedReader br = new BufferedReader(new FileReader(s))) {
            String line;
            List<String> list = new ArrayList<>();

            // Read data from the text file and add it to the list
            while ((line = br.readLine()) != null) {
                list.add(line.trim());
            }

            dati.add(list);
        } catch (IOException e) {
```

```java
                e.printStackTrace();
            }
        }

        // Insert data into the specified table
        if(table.equals("airport")) {
            System.out.println("Insert Airport");
            for (String l1: dati.get(0)) {
                SimpleStatement query = new SimpleStatement("INSERT INTO airport
                    ↪ (Size, Country, Country_code, Geo_Point, IATA_code,
                    ↪ ICAO_code, Name, Name_en, Name_fr, Operator, Phone,
                    ↪ Website) VALUES " + l1 + ";");
                session.execute(query);
            }
        } else {
            if(table.equals("flight")) {
                System.out.println("Insert Flight");
                for (String l2: dati.get(1)) {
                    SimpleStatement query = new SimpleStatement("INSERT INTO
                        ↪ flight (ID, Departure, Destination, Number_of_Seats,
                        ↪ Day, Hour, Operator, Duration, Price_per_Person) VALUES
                        ↪ " + l2 + ";");
                    session.execute(query);
                }
            } else {
                System.out.println("Insert Seat");
                for (String l3: dati.get(2)) {
                    SimpleStatement query = new SimpleStatement("INSERT INTO seat
                        ↪ (Flight, ID, Status, Name, Surname, Document_Info,
                        ↪ Date_of_Birth, Balance) VALUES " + l3 + ";");
                    session.execute(query);
                }
            }
        }
    }
}

// Main method to initialize the Cassandra session and perform setup tasks
public static void main( String[] args ) {
    String endPoint = "cassandra.eu-north-1.amazonaws.com";
    int portNumber = 9142;

    try {
        // Initialize Cassandra session with AWS MCS authentication
        session = Cluster.builder()
                .addContactPoint(endPoint)
                .withPort(portNumber)
                .withAuthProvider(new SigV4AuthProvider("eu-north-1"))
                .withSSL()
                .build()
                .connect();

        System.out.println("Connected to Cassandra!");
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Create keyspace and use it
    session.execute("CREATE KEYSPACE IF NOT EXISTS my_airport WITH replication
        ↪ = "
            + "{'class':'SimpleStrategy','replication_factor':3};");

    session.execute("USE my_airport;");
```

```
// Get list of existing tables before creating new ones
ResultSet rs = session.execute("SELECT table_name FROM
    ↪ system_schema.tables " +
        "WHERE keyspace_name = 'my_airport' ");

List<String> list_before = new ArrayList<>();
for (Row row : rs) {
    list_before.add(row.getString("table_name"));
}

// Create necessary tables if they don't exist
session.execute("CREATE TABLE IF NOT EXISTS airport (Name_en text, Country
    ↪ text, Website text, Geo_Point text, ICAO_code text, Country_code
    ↪ text, Phone text, Name_fr text, Name text, IATA_code text, Size
    ↪ int, Operator text, PRIMARY KEY (IATA_code));");
session.execute("CREATE TABLE IF NOT EXISTS flight (ID text,
    ↪ Number_of_Seats int, Day date, Hour time, Operator text, Duration
    ↪ text, Price_per_Person int, Destination text, Departure text,
    ↪ PRIMARY KEY ((Departure), Destination, Day, Hour, id));");
session.execute("CREATE TABLE IF NOT EXISTS seat (Flight text, Status
    ↪ text, ID text, Name text, Surname text, Document_Info text,
    ↪ Date_of_Birth date, Balance int, PRIMARY KEY ((Flight), ID));");

// Get list of tables after creation
rs = session.execute("SELECT table_name FROM system_schema.tables " +
        "WHERE keyspace_name = 'my_airport' ");

List<String> list_after = new ArrayList<>();
for (Row row : rs) {
    list_after.add(row.getString("table_name"));
}

// Insert data into newly created tables
if (list_before.size() != list_after.size()) {
    for (String s : list_after) {
        if (!list_before.contains(s)) {
            insert(s);
        }
    }
}

// Test various queries and transactions
// Test Query getFlightsFromAirport
System.out.println("getFlightsFromAirport");
System.out.println(getFlightsFromAirport("BGY"));

// Test Query getAirportFromIATA
System.out.println("getAirportFromIATA");
System.out.println(getAirportFromIATA("BGY"));

// Test Query getFlightsFromDepartureAndDestination
System.out.println("getFlightsFromDepartureAndDestination");
System.out.println(getFlightsFromDepartureAndDestination("BGY", "AAL"));

// Test Query getAvailableSeats
System.out.println("getAvailableSeats");
System.out.println(getAvailableSeats("6677f9a27acf35542d8ef4df"));

// Test Query getSeatStatus
System.out.println("getSeatStatus");
try {
    System.out.println(getSeatStatus("6677f9a27acf35542d8ef4df", "10B"));
} catch (Exception e) {
```

```
            throw new RuntimeException(e);
        }
    }
}
```

Transactions

MongoDB Transactions

Transactions in the Data Model   In addition to the Connection class and the Modeling class, there are various other classes in the project, whose purpose is to model and manage transactions. In this regard, let's start by observing the main class, which serves as the entry point for the application, demonstrating various operations such as MongoDB interactions, logging configurations, and concurrent transaction handling. The Java code of the application is as follows:

```java
// Import necessary classes
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;

import java.io.IOException;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.logging.FileHandler;
import java.util.logging.Formatter;
import java.util.logging.Logger;
import java.util.logging.LogRecord;

/**
 * The Main class serves as the entry point for the application, demonstrating
 *    ↪ various operations
 * such as MongoDB interactions, logging configurations, and concurrent
 *    ↪ transaction handling.
 *
 * @author Andrea Moleri
 * @version 1.0
 * @since 2024-07-02
 */
public class Main {

    private static final Logger logger = Logger.getLogger(Main.class.getName());

    /**
     * The main method configures logging, establishes a connection to a MongoDB
     *    ↪ instance,
     * and performs various database operations such as retrieving flight
     *    ↪ information,
     * booking seats, and handling concurrent transactions.
     *
     * @param args Command line arguments (not used).
     */
    public static void main(String[] args) {
        try {
            // Configure the FileHandler with a custom Formatter
            FileHandler fileHandler = new FileHandler("app.log");
            fileHandler.setFormatter(new SimpleLogFormatter());
            logger.addHandler(fileHandler);
```

```java
// MongoDB connection details
String connectionString =
    ↪ "mongodb+srv://admin:admin@learningmongodb.hikoksa.mongodb.net"
    ↪ + "/?retryWrites=true&w=majority&appName=LearningMongoDB";
String dbName = "Airports";
String collectionName = "airportCollection";

// Initialize MongoDB client and Transactions instance
MongoClient mongoClient = MongoClients.create(connectionString);
Transactions transactions = new Transactions(mongoClient, dbName,
    ↪ collectionName);

// Example usage of Transactions methods
String departureAirportCode = "MXP";
String arrivalAirportCode = "PMV";

Map<String, Map<String, String>> flightsFromBGY =
    ↪ transactions.getFlightsFromAirport(departureAirportCode);

logger.info("|-- TESTING FLIGHT RETRIEVAL");
logger.info("|---|-- Retrieving flights departing from the specified
    ↪ airport (" + departureAirportCode + ")");
for (Map.Entry<String, Map<String, String>> entry :
    ↪ flightsFromBGY.entrySet()) {
    String iataCode = entry.getKey();
    Map<String, String> flightDetails = entry.getValue();
    logger.info("|---|-- Flight to " + iataCode + ": " +
        ↪ flightDetails);
}
logger.info("");

PeopleGenerator generator = null;
String flightID = null;
List<String> availableSeatsDetails = null;
if (flightsFromBGY.containsKey(arrivalAirportCode)) {
    flightID = flightsFromBGY.get(arrivalAirportCode).get("ID");

    logger.info("|-- TESTING SEATS RETRIEVAL");
    logger.info("|---|-- The user chose to depart from " +
        ↪ departureAirportCode + " to " + arrivalAirportCode);
    logger.info("|---|-- Retrieving available seats for the specified
        ↪ flight (" + departureAirportCode + " -> " +
        ↪ arrivalAirportCode + ")");
    availableSeatsDetails =
        ↪ transactions.getAvailableSeats(departureAirportCode,
        ↪ arrivalAirportCode);
    logger.info("|---|-- Number of available seats: " +
        ↪ availableSeatsDetails.size());
    logger.info("|---|-- Available seats are: " +
        ↪ availableSeatsDetails);
    logger.info("");

    if (!availableSeatsDetails.isEmpty()) {
        // Shuffle the list to randomize the seat selection
        Collections.shuffle(availableSeatsDetails);

        // TESTING CONCURRENT TRANSACTIONS
        String seatID = availableSeatsDetails.get(0);
        generator = new PeopleGenerator();
        List<PeopleGenerator.Person> people =
            ↪ generator.generatePeople(2);
        PeopleGenerator.Person person1 = people.get(0);
```

```
        PeopleGenerator.Person person2 = people.get(1);

        logger.info("|-- TESTING CONCURRENT TRANSACTIONS");
        logger.info("|---|-- Testing concurrent booking for the same
            ↪ seat (" + seatID + ") on flight " + flightID + " (" +
            ↪ departureAirportCode + " -> " + arrivalAirportCode +
            ↪ ")");
        logger.info("|---|-- Booking result for person 1: " +
            ↪ transactions.bookFlight(flightID, seatID, person1));
        logger.info("|---|-- Booking result for person 2: " +
            ↪ transactions.bookFlight(flightID, seatID, person2));
        logger.info("|---|-- Person 1 booked the seat first and
            ↪ successfully: " + person1);
        logger.info("|---|-- Person 2 consequently failed to book the
            ↪ same seat: " + person2);
        logBalanceChange(logger, person1);
        logger.info("");

        // TESTING NON-CONCURRENT TRANSACTION
        String newSeatID = availableSeatsDetails.get(1);
        PeopleGenerator.Person newPerson =
            ↪ generator.generatePeople(1).get(0);

        logger.info("|-- TESTING NON-CONCURRENT TRANSACTION");
        logger.info("|---|-- A new user chose to book the " +
            ↪ newSeatID + " seat, without any concurrency from other
            ↪ users");
        logger.info("|---|-- Booking result for new person: " +
            ↪ transactions.bookFlight(flightID, newSeatID,
            ↪ newPerson));
        logger.info("|---|-- New person booked the seat successfully:
            ↪ " + newPerson);
        logBalanceChange(logger, newPerson);
        logger.info("");
    } else {
        logger.info("|---|-- No available seats found for the flight "
            ↪ + flightID + " (" + departureAirportCode + " -> " +
            ↪ arrivalAirportCode + ")");
    }
} else {
    logger.info("|---|-- No flights found departing from " +
        ↪ departureAirportCode);
}

// Testing for the "poor" person attempting to book an available seat
availableSeatsDetails =
    ↪ transactions.getAvailableSeats(departureAirportCode,
    ↪ arrivalAirportCode);
PeopleGenerator.Person poorPerson = generator.generatePoorPerson();
String poorPersonSeatID = availableSeatsDetails.get(1);

logger.info("|-- TESTING INSUFFICIENT FUNDS BEHAVIOUR");
logger.info("|---|-- " + poorPerson.getName() + " " +
    ↪ poorPerson.getSurname() + " with balance " +
    ↪ poorPerson.getBalance() + "$ is attempting to book seat " +
    ↪ poorPersonSeatID);
logger.info("|---|-- Booking result for person with insufficient
    ↪ funds: " + transactions.bookFlight(flightID, poorPersonSeatID,
    ↪ poorPerson));
logger.info("|---|-- The booking failed due to insufficient balance to
    ↪ complete the transaction");
logger.info("");
```

```java
            // Close the MongoDB client
            transactions.close();

            // Close the FileHandler after use
            fileHandler.close();

        } catch (IOException e) {
            logger.severe("|---|-- Error during logger configuration: " +
                ↪ e.getMessage());
        }
    }

    /**
     * A custom formatter to exclude the date, time, class name, and log level
         ↪ from the log messages.
     */
    static class SimpleLogFormatter extends Formatter {
        @Override
        public String format(LogRecord record) {
            return record.getMessage() + System.lineSeparator();
        }
    }

    /**
     * Logs the balance change for a person after booking a flight.
     *
     * @param logger The logger instance to use for logging.
     * @param person The person whose balance change is being logged.
     */
    private static void logBalanceChange(Logger logger, PeopleGenerator.Person
        ↪ person) {
        logger.info(String.format("|---|-- The balance of %s %s before booking the
            ↪ flight was %.2f$",
                person.getName(), person.getSurname(), person.getOldBalance()));
        logger.info(String.format("|---|-- The cost of the flight was %.2f$",
            ↪ person.getDifference()));
        logger.info(String.format("|---|-- This means the new balance of %s %s is
            ↪ %.2f$",
                person.getName(), person.getSurname(), person.getBalance()));
    }
}
```

When executed, this class returns a log file called `app.log`, an example of the sh content of which
is shown below: For visual simplicity, some of the flight parameters available in the testing flight
retrieval section have been removed, but they can be added again by editing the body of the main
class above.

```
|-- TESTING FLIGHT RETRIEVAL
|---|-- Retrieving flights departing from the specified airport (MXP)
|---|-- Flight to CDG: {IATA_code=CDG, Country=France,
    ↪ ID=6677f9907acf35542d8ef21b, Name=Aeroport de Paris-Charles-de-Gaulle}
|---|-- Flight to PMV: {IATA_code=PMV, Country=Venezuela,
    ↪ ID=6677f9907acf35542d8ef21c, Name=Aeropuerto Internacional del Caribe
    ↪ Santiago Marino}

|-- TESTING SEATS RETRIEVAL
|---|-- The user chose to depart from MXP to PMV
|---|-- Retrieving available seats for the specified flight (MXP -> PMV)
|---|-- Number of available seats: 13
|---|-- Available seats are: [1F, 2C, 3A, 4F, 5A, 5B, 5D, 9A, 9C, 10B, 11C, 13B,
    ↪ 13F]

|-- TESTING CONCURRENT TRANSACTIONS
```

```
|---|-- Testing concurrent booking for the same seat (11C) on flight
    ↪ 6677f9907acf35542d8ef21c (MXP -> PMV)
|---|-- Booking result for person 1: true
|---|-- Booking result for person 2: false
|---|-- Person 1 booked the seat first and successfully: Person{name='Salvatore',
    ↪ surname='Montanari', documentInfo='M4J601O', dateOfBirth='1933-08-27',
    ↪ balance=43988.87}
|---|-- Person 2 consequently failed to book the same seat: Person{name='Enrico',
    ↪ surname='Pellegrini', documentInfo='PGUJ8Q0', dateOfBirth='1938-06-12',
    ↪ balance=39196.33}
|---|-- The balance of Salvatore Montanari before booking the flight was 44350.87$
|---|-- The cost of the flight was 362.00$
|---|-- This means the new balance of Salvatore Montanari is 43988.87$

|-- TESTING NON-CONCURRENT TRANSACTION
|---|-- A new user chose to book the 3A seat, without any concurrency from other
    ↪ users
|---|-- Booking result for new person: true
|---|-- New person booked the seat successfully: Person{name='Marcella',
    ↪ surname='D'Agostino', documentInfo='YUW4PE3', dateOfBirth='2005-08-05',
    ↪ balance=92079.26}
|---|-- The balance of Marcella D'Agostino before booking the flight was 92441.26$
|---|-- The cost of the flight was 362.00$
|---|-- This means the new balance of Marcella D'Agostino is 92079.26$

|-- TESTING INSUFFICIENT FUNDS BEHAVIOUR
|---|-- Angela Sanna with balance 10.0$ is attempting to book seat 2C
|---|-- Booking result for person with insufficient funds: false
|---|-- The booking failed to to having insufficient balance to complete the
    ↪ transaction
```

Various test cases are listed within the log file just shown as described by the objectives of our model. In particular, flight retrieval, seats retrieval, concurrent and non-concurrent transactions and testing of insufficient funds behaviour are tested. It should be specified that it would have been possible to add a test case called "TESTING ALREADY BOOKED TRANSACTION", in which the behavior of a user who would have tried to book an already occupied seat would have been tested. However, this test was omitted as it was already logically incorporated by the locking mechanism of the test case called "TESTING CONCURRENT TRANSACTIONS".

At this point it is natural to ask how the actual Transactions work within the database. To do this, let's take a look at how the `Transactions.java` class works below

```java
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.UpdateOptions;
import com.mongodb.client.model.Updates;
import com.mongodb.client.result.UpdateResult;
import org.bson.Document;
import org.bson.types.ObjectId;

import java.io.IOException;
import java.util.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.logging.*;

/**
```

```
    * The Transactions class provides methods for interacting with a MongoDB
       ↪ collection
    * to retrieve flight information, check seat availability, and handle booking
       ↪ operations
    * with thread safety considerations.
    *
    * @version 1.0
    * @since 2024-07-02
    * @author Andrea Moleri
    */
class Transactions {

    private MongoClient mongoClient;
    private MongoDatabase database;
    private MongoCollection<Document> collection;
    private Lock lock = new ReentrantLock();
    private static final Logger logger =
        ↪ Logger.getLogger(Transactions.class.getName());

    /**
     * Constructs a Transactions object with the specified MongoDB client,
        ↪ database name, and collection name.
     *
     * @param mongoClient The MongoDB client instance.
     * @param dbName The name of the database.
     * @param collectionName The name of the collection.
     */
    public Transactions(MongoClient mongoClient, String dbName, String
        ↪ collectionName) {
        this.mongoClient = mongoClient;
        this.database = mongoClient.getDatabase(dbName);
        this.collection = database.getCollection(collectionName);
    }

    /**
     * Retrieves flights departing from the specified airport.
     *
     * @param airportCode The IATA code of the departure airport.
     * @return A map containing flight details keyed by destination airport IATA
        ↪ code.
     */
    public Map<String, Map<String, String>> getFlightsFromAirport(String
        ↪ airportCode) {
        Map<String, Map<String, String>> flightsMap = new HashMap<>();

        FindIterable<Document> iterable = collection.find(Filters.eq("IATA_code",
            ↪ airportCode));

        for (Document airportDoc : iterable) {
            List<Document> flightsFromAirport = airportDoc.getList("Flights",
                ↪ Document.class);
            for (Document flight : flightsFromAirport) {
                ObjectId destinationId = flight.getObjectId("Destination");
                Document destinationAirport = getAirportById(destinationId);
                if (destinationAirport != null) {
                    Map<String, String> flightDetails = new HashMap<>();
                    flightDetails.put("ID", flight.getString("ID"));
                    flightDetails.put("Name",
                        ↪ destinationAirport.getString("Name"));
                    flightDetails.put("IATA_code",
                        ↪ destinationAirport.getString("IATA_code"));
                    flightDetails.put("Country",
                        ↪ destinationAirport.getString("Country"));
```

```java
                flightsMap.put(destinationAirport.getString("IATA_code"),
                    ↪ flightDetails);
            }
        }
    }

    return flightsMap;
}

/**
 * Retrieves the details of an airport by its ID.
 *
 * @param airportId The ObjectId of the airport.
 * @return The document containing airport details, or null if not found.
 */
private Document getAirportById(ObjectId airportId) {
    return collection.find(Filters.eq("_id", airportId)).first();
}

/**
 * Retrieves the list of available seats for a flight from a departure airport
 *    ↪ to an arrival airport.
 *
 * @param departureAirportCode The IATA code of the departure airport.
 * @param arrivalAirportCode The IATA code of the arrival airport.
 * @return A list of seat IDs that are available.
 */
public List<String> getAvailableSeats(String departureAirportCode, String
    ↪ arrivalAirportCode) {
    List<String> availableSeatsList = new ArrayList<>();

    Document departureAirport = collection.find(Filters.eq("IATA_code",
        ↪ departureAirportCode)).first();

    if (departureAirport != null) {
        List<Document> flightsFromDeparture =
            ↪ departureAirport.getList("Flights", Document.class);
        for (Document flight : flightsFromDeparture) {
            ObjectId destinationId = flight.getObjectId("Destination");
            Document destinationAirport = getAirportById(destinationId);
            if (destinationAirport != null &&
                ↪ destinationAirport.getString("IATA_code").equals(arrivalAirportCode))
                ↪ {
                List<Document> seats = flight.getList("Seats", Document.class);
                for (Document seat : seats) {
                    if ("Vacant".equals(seat.getString("Status"))) {
                        availableSeatsList.add(seat.getString("ID"));
                    }
                }
                break;
            }
        }
    }

    return availableSeatsList;
}

/**
 * Attempts to book a flight for a given person, ensuring thread safety and
 *    ↪ atomic updates in MongoDB.
 *
 * @param flightID The ID of the flight to book.
 * @param seatID The ID of the seat to book.
```

```java
 * @param person The person attempting to book the flight.
 * @return True if the booking is successful, false otherwise.
 */
public boolean bookFlight(String flightID, String seatID,
    ↪ PeopleGenerator.Person person) {
    lock.lock();
    try {
        Document flightDocument = collection.find(Filters.eq("Flights.ID",
            ↪ flightID)).first();
        if (flightDocument == null) {
            return false;
        }

        List<Document> flights = flightDocument.getList("Flights",
            ↪ Document.class);
        Document targetFlight = flights.stream()
                .filter(f -> flightID.equals(f.getString("ID")))
                .findFirst()
                .orElse(null);

        if (targetFlight == null) {
            return false;
        }

        Object priceObject = targetFlight.get("Price_per_Person");
        if (!(priceObject instanceof Number)) {
            return false; // Handle case where price is not a valid number
        }
        double seatPrice = ((Number) priceObject).doubleValue(); // Convert
            ↪ seat price to double

        List<Document> seats = targetFlight.getList("Seats", Document.class);
        Document seat = seats.stream()
                .filter(s -> seatID.equals(s.getString("ID")) &&
                    ↪ "Vacant".equals(s.getString("Status")))
                .findFirst()
                .orElse(null);

        if (seat == null) {
            return false;
        }

        if (person.getBalance() < seatPrice) {
            return false;
        }

        // Store old balance and difference in Person object
        person.setOldBalance(person.getBalance());
        person.setDifference(seatPrice);

        // Update seat status and person details
        seat.put("Status", "Booked");
        seat.put("Name", person.getName());
        seat.put("Surname", person.getSurname());
        seat.put("Document_Info", person.getDocumentInfo());
        seat.put("Date_of_Birth", person.getDateOfBirth());
        seat.put("Balance", person.getBalance() - seatPrice);

        // Update MongoDB document atomically
        UpdateResult result = collection.updateOne(
                Filters.and(
                        Filters.eq("Flights.ID", flightID),
                        Filters.eq("Flights.Seats.ID", seatID),
```

```
                            Filters.eq("Flights.Seats.Status", "Vacant")
                    ),
                    Updates.combine(
                            Updates.set("Flights.$[flight].Seats.$[seat].Status",
                                ↪ "Booked"),
                            Updates.set("Flights.$[flight].Seats.$[seat].Name",
                                ↪ person.getName()),
                            Updates.set("Flights.$[flight].Seats.$[seat].Surname",
                                ↪ person.getSurname()),
                            Updates.set("Flights.$[flight].Seats.$[seat].Document_Info",
                                ↪ person.getDocumentInfo()),
                            Updates.set("Flights.$[flight].Seats.$[seat].Date_of_Birth",
                                ↪ person.getDateOfBirth()),
                            Updates.set("Flights.$[flight].Seats.$[seat].Balance",
                                ↪ person.getBalance() - seatPrice)
                    ),
                    new UpdateOptions().arrayFilters(Arrays.asList(
                            Filters.eq("flight.ID", flightID),
                            Filters.eq("seat.ID", seatID)
                    ))
            );

            if (result.getModifiedCount() == 1) {
                // Deduct seat price from person's balance
                person.setBalance(person.getBalance() - seatPrice);
                return true;
            } else {
                return false;
            }
        } finally {
            lock.unlock();
        }
    }

    /**
     * Closes the MongoDB client connection.
     */
    public void close() {
        mongoClient.close();
    }

}
```

The code in the `Transactions.java` class is designed to handle various operations related to flights in a MongoDB collection. It allows for retrieving flight information, checking seat availability, and booking seats in a thread-safe manner. We now want to focus our attention on the management of concurrent read and write operations in MongoDB, and how the database provides tools to manage them. Not only this, but we will also detail how the Java class just shown takes advantage of Locking mechanisms in conjunction with the tools offered by MongoDB to avoid any conflicts

Managing Concurrency   MongoDB is designed to handle high-concurrency environments, supporting concurrent read operations efficiently through its architecture and features like multi-version concurrency control (MVCC). This mechanism allows multiple clients to read data from the database simultaneously without interfering with each other, ensuring that each read operation receives a consistent view of the data as it was at the start of the read operation.

In MVCC, when a read operation begins, MongoDB provides a snapshot of the data at that moment. Any updates made by other clients after the read operation has started do not affect the snapshot seen by the reader. This approach provides several benefits:

1. **Consistency:** readers always see a consistent view of the data, even if other clients are making concurrent writes.
2. **Isolation:** MVCC ensures that readers are isolated from writers. Changes made by a writer are not visible to readers until the write is complete.
3. **Performance:** by allowing multiple clients to read from the same data set concurrently, MongoDB can handle a high number of read requests efficiently.

MongoDB employs snapshot reads for read operations within transactions. A snapshot read captures the state of the database at the start of the transaction, ensuring that all read operations within the transaction see a consistent view of the data. This is particularly useful in scenarios requiring strong consistency guarantees. MongoDB provides various read concern levels to control the consistency and isolation properties of read operations:

- **"local":** Returns the most recent data available, regardless of the acknowledgment of the write operations.
- **"majority":** Returns data that has been acknowledged by the majority of the replica set members, ensuring that the read data is durable.
- **"snapshot":** Provides a snapshot of the data as it was at the start of a transaction, ensuring a consistent view of the data during the transaction.

The `Transactions.java` class leverages MongoDB's ability to handle concurrent operations to ensure efficient and safe access to our dataset. Here is how the class handles concurrent reads and writes:

1. **Retrieving Flights from an Airport:**

- The `getFlightsFromAirport` method retrieves all flights departing from a specified airport. MongoDB handles the concurrent reads internally, ensuring that the data returned is consistent.

2. **Checking Available Seats:**

- The `getAvailableSeats` method checks the availability of seats on a flight. This method involves reading the flight and seat information from the database. MongoDB's concurrent read capabilities ensure that multiple clients can check seat availability simultaneously without interference.

3. **Booking a Flight:**

- The `bookFlight` method handles concurrent write operations. This method is designed to be thread-safe, ensuring that only one thread can book a seat at a time using the `ReentrantLock`.

The `ReentrantLock` in Java provides a way to ensure that only one thread can execute a block of code at a time. In the `bookFlight` method, the lock is used to ensure that the entire process of checking seat availability and updating the seat status is atomic. Here's how it works:

1. **Acquire the Lock:** when a thread enters the `bookFlight` method, it calls `lock.lock()` to acquire the lock. This prevents other threads from entering the critical section of the code until the lock is released.
2. **Check and Update:** inside the locked section, the code performs the following:

- It checks if the flight and the seat are available.
- It verifies that the person has sufficient balance.
- It updates the seat status to "Booked" and deducts the price from the person's balance.
- It performs an atomic update in the MongoDB collection using the `updateOne` method with array filters to ensure only the specified seat is updated.

3. **Release the Lock:** after the update operation, the lock is released using `lock.unlock()`, allowing other threads to attempt booking operations.

By using `ReentrantLock`, the code ensures that even if multiple threads try to book the same seat at the same time, *only one will succeed,* and the others will receive a failure response. This approach prevents race conditions and ensures data consistency within the application.

Other Auxiliary Classes To conclude the overview of the code necessary for the operation of both the Main class and the Transactions class, it is best to shift the focus of our attention to the last two classes that make up the model: PeopleGenerator and its auxiliary class NameSurnamePool. First follows the PeopleGenerator code, whose purpose is to generate a People object. Given the simplicity of its nature (it is effectively a random generator of people based on various parameters), the code will be self-explanatory.

```java
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.ThreadLocalRandom;

/**
 * The PeopleGenerator class is responsible for generating a list of Person
 *     ↪ objects with
 * random attributes, including name, surname, document information, date of
 *     ↪ birth, and balance.
 * It can also generate a "poor" person with a limited balance.
 *
 * @version 1.0
 * @since 2024-07-02
 * @author Andrea Moleri
 */
public class PeopleGenerator {

    private static final Random RANDOM = new Random();
    private static final DecimalFormat DECIMAL_FORMAT = new DecimalFormat("#.##");

    /**
     * Generates a list of Person objects with specified count.
     *
     * @param count The number of Person objects to generate.
     * @return A list of generated Person objects.
     */
    public List<Person> generatePeople(int count) {
        List<Person> people = new ArrayList<>(count);
        Set<String> documentNumbers = new HashSet<>(count);

        for (int i = 0; i < count; i++) {
            String name = NameSurnamePool.NAMES
                    .get(RANDOM.nextInt(NameSurnamePool.NAMES.size()));
            String surname = NameSurnamePool.SURNAMES
                    .get(RANDOM.nextInt(NameSurnamePool.SURNAMES.size()));
            String documentInfo;

            // Generate a unique document number
            do {
                documentInfo = generateDocumentInfo();
            } while (documentNumbers.contains(documentInfo));

            documentNumbers.add(documentInfo);
            String dateOfBirth = generateDateOfBirth();
            double balance = 100 + (100000 - 100) * RANDOM.nextDouble();
```

```java
        // Round the balance to two decimal places
        balance = Double.parseDouble(DECIMAL_FORMAT.format(balance));

        people.add(new Person(name, surname, documentInfo, dateOfBirth,
            ↪ balance));
    }

    return people;
}

/**
 * Generates a 7-character alphanumeric document information string.
 *
 * @return A randomly generated document information string.
 */
private String generateDocumentInfo() {
    String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    StringBuilder sb = new StringBuilder(7);
    for (int i = 0; i < 7; i++) {
        sb.append(chars.charAt(RANDOM.nextInt(chars.length())));
    }
    return sb.toString();
}

/**
 * Generates a random date of birth between 18 and 100 years ago.
 *
 * @return A randomly generated date of birth in "yyyy-MM-dd" format.
 */
private String generateDateOfBirth() {
    long now = System.currentTimeMillis();
    long eighteenYearsAgo = now - (18L * 365 * 24 * 60 * 60 * 1000);
    long hundredYearsAgo = now - (100L * 365 * 24 * 60 * 60 * 1000);
    long randomMillis = ThreadLocalRandom.current().nextLong(hundredYearsAgo,
        ↪ eighteenYearsAgo);
    return new java.text.SimpleDateFormat("yyyy-MM-dd").format(new
        ↪ java.util.Date(randomMillis));
}

/**
 * Generates a Person object with limited balance, representing a "poor"
    ↪ person.
 *
 * @return A generated "poor" Person object.
 */
public Person generatePoorPerson() {
    String name = NameSurnamePool.NAMES
            .get(RANDOM.nextInt(NameSurnamePool.NAMES.size()));
    String surname = NameSurnamePool.SURNAMES
            .get(RANDOM.nextInt(NameSurnamePool.SURNAMES.size()));
    String documentInfo = generateDocumentInfo();
    String dateOfBirth = generateDateOfBirth();
    double balance = 10.0; // Balance limited to 10$

    return new Person(name, surname, documentInfo, dateOfBirth, balance);
}

/**
 * The Person class represents an individual with attributes such as name,
    ↪ surname, document information,
 * date of birth, and balance. It also tracks the previous balance and balance
    ↪ difference for transactions.
 */
```

```java
public static class Person {
    private String name;
    private String surname;
    private String documentInfo;
    private String dateOfBirth;
    private double balance;
    private double oldBalance; // Added to manage the previous balance
    private double difference; // Added to manage balance difference

    /**
     * Constructs a new Person instance.
     *
     * @param name The name of the person.
     * @param surname The surname of the person.
     * @param documentInfo The document information of the person.
     * @param dateOfBirth The date of birth of the person.
     * @param balance The balance of the person.
     */
    public Person(String name, String surname, String documentInfo, String
        ↪ dateOfBirth, double balance) {
      this.name = name;
      this.surname = surname;
      this.documentInfo = documentInfo;
      this.dateOfBirth = dateOfBirth;
      this.balance = balance;
      this.oldBalance = balance; // Initially, the previous balance is equal
          ↪ to the current balance
      this.difference = 0.0; // Initially, there is no balance difference
    }

    // Getters and setters...

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getDocumentInfo() {
        return documentInfo;
    }

    public void setDocumentInfo(String documentInfo) {
        this.documentInfo = documentInfo;
    }

    public String getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(String dateOfBirth) {
        this.dateOfBirth = dateOfBirth;
    }
```

```java
        public double getBalance() {
            return balance;
        }

        public void setBalance(double balance) {
            this.balance = balance;
        }

        public double getOldBalance() {
            return oldBalance;
        }

        public void setOldBalance(double oldBalance) {
            this.oldBalance = oldBalance;
        }

        public double getDifference() {
            return difference;
        }

        public void setDifference(double difference) {
            this.difference = difference;
        }

        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", surname='" + surname + '\'' +
                    ", documentInfo='" + documentInfo + '\'' +
                    ", dateOfBirth='" + dateOfBirth + '\'' +
                    ", balance=" + balance +
                    '}';
        }
    }

    /**
     * The main method to test the generation of Person objects.
     *
     * @param args Command line arguments (not used).
     */
    public static void main(String[] args) {
        PeopleGenerator generator = new PeopleGenerator();
        List<Person> people = generator.generatePeople(10000);

        // Print the first 10 people to verify the output
        for (int i = 0; i < 10; i++) {
            System.out.println(people.get(i));
        }
    }
}
```

Similarly, we present the auxiliary class NameSurnamePool, which is nothing more than a class intended to contain Array of possible names and surnames, which will be used in combination to generate a People object. Again, the code is simple enough that it does not require further explanations other than those already provided by the comments within it.

```java
import java.util.Arrays;
import java.util.List;

/**
```

```java
 * The NameSurnamePool class provides static lists of common Italian names and
 *     ↪ surnames.
 * These lists can be used for generating sample data or for any other purpose
 *     ↪ requiring
 * a set of predefined names and surnames. In the code, this class is used to
 *     ↪ provide names
 * and surnames that are used in the PeopleGenerator.java Class, that was
 *     ↪ explained previously
 *
 * @version 1.0
 * @since 2024-07-02
 * @author Andrea Moleri
 */
public class NameSurnamePool {

    /**
     * A static list of common Italian first names.
     */
    public static final List<String> NAMES = Arrays.asList(
            "Luca", "Marco", "Giulia", "Francesca", "Alessandro", "Chiara",
                ↪ "Matteo", "Valentina", "Simone", "Martina",
            "Andrea", "Sara", "Stefano", "Giorgia", "Davide", "Alice", "Federico",
                ↪ "Elena", "Nicola", "Silvia", "Giulio",
            "Paolo", "Anna", "Antonio", "Laura", "Roberto", "Roberta", "Enrico",
                ↪ "Marta", "Gabriele", "Claudia",
            "Leonardo", "Veronica", "Massimo", "Cristina", "Filippo", "Eleonora",
                ↪ "Giovanni", "Maria", "Pietro", "Federica",
            "Vincenzo", "Daniela", "Riccardo", "Alberto", "Angela", "Emanuele",
                ↪ "Monica", "Salvatore", "Serena", "Beatrice",
            "Tommaso", "Camilla", "Giovanna", "Vittoria", "Gabriella", "Lorenzo",
                ↪ "Michele", "Luigi", "Alessia", "Edoardo",
            "Fabrizio", "Irene", "Fabio", "Patrizia", "Sergio", "Elisabetta",
                ↪ "Raffaele", "Caterina", "Ludovica", "Maurizio",
            "Alberto", "Manuela", "Claudio", "Franco", "Tiziana", "Pasquale",
                ↪ "Debora", "Sabrina", "Giuseppe", "Marcella",
            "Lucrezia", "Dario", "Cristiano", "Eleonora", "Renato", "Cinzia",
                ↪ "Alessia", "Cristian", "Lucia", "Sofia",
            "Matilde", "Ginevra", "Arianna", "Samuele", "Noemi", "Aurora",
                ↪ "Rachele", "Michela", "Melissa", "Rebecca"
    );

    /**
     * A static list of common Italian surnames.
     */
    public static final List<String> SURNAMES = Arrays.asList(
            "Rossi", "Bianchi", "Esposito", "Romano", "Ferrari", "Ricci",
                ↪ "Marino", "Greco", "Bruno", "Gallo",
            "Conti", "De Luca", "Costa", "Giordano", "Manenti", "Rizzo",
                ↪ "Lombardi", "Moretti", "Barbieri", "Fontana",
            "Santoro", "Mariani", "Rinaldi", "Caruso", "Ferraro", "Gatti",
                ↪ "Testa", "Martini", "Pellegrini", "Palumbo",
            "Sorrentino", "Longo", "Leone", "Gentile", "Lombardo", "Moleri",
                ↪ "Colombo", "D'Amico", "Martino", "Bianco",
            "Morelli", "Amato", "Silvestri", "Grasso", "Parisi", "Sanna",
                ↪ "Farina", "Ruggiero", "Monti", "Coppola",
            "De Santis", "Mazza", "Villa", "Marchetti", "Caputo", "Ferretti",
                ↪ "Battaglia", "Fiore", "Messina", "Donati",
            "Vitali", "Fabbri", "Valentini", "Orlando", "Mariani", "Rizzi",
                ↪ "Benedetti", "Neri", "Bellini", "Rossetti",
            "Bertoni", "Piras", "Rosati", "Di Lorenzo", "Martelli", "Marino",
                ↪ "Ferri", "Brunetti", "De Angelis", "Basili",
            "Veneziani", "Pascarella", "Pagano", "Palmieri", "Ferraro", "Armani",
                ↪ "Raimondi", "Montanari", "Carbone", "D'Agostino"
```

```
    );
}
```

This concludes the overview of the code used for the purposes of data ingestion, data processing, data parsing, data modeling and transactions management within the MongoDB database, paying careful attention to the management of concurrent transactions.

Cassandra Transactions

Transaction and concurrency management in a distributed system like Cassandra requires specific structures to ensure consistency and reliability. These include batch operations and Lightweight Transactions (LWT), which facilitate data operations by offering a certain level of atomicity and consistency.

Batch Operations in Cassandra   Batch operations in Cassandra allow executing a group of write operations as a single unit. This means all operations within the batch are sent to the database together and executed in the same context. A batch can include insert, update, or delete instructions. Its syntax can be:

```
BEGIN BATCH
INSERT INTO keyspace_name.table_name (column1, column2) VALUES (value1, value2);
UPDATE keyspace_name.table_name SET column1 = value WHERE column2 = value2;
DELETE FROM keyspace_name.table_name WHERE column1 = value1;
APPLY BATCH;
```

Batch operations are particularly useful for increasing the efficiency of sending operations to the server, as they reduce the number of exchanges between the client and server. When executing a batch, if it involves operations on a single partition, Cassandra can ensure atomicity. This means that all operations in the batch will either be applied together or none of them will be applied. If any operation fails for any reason (e.g., syntax error, data validation error, or network error preventing batch completion), none of the operations in the batch will be applied. Thus, the entire batch is treated as an atomic unit, as if it were a single operation.

When the batch involves operations on multiple partitions, complete atomicity is not guaranteed. In these cases, Cassandra tries to apply all operations, but if one operation fails, the behavior does not ensure the complete reversibility of already applied operations. Therefore, batches in Cassandra provide a form of partial atomicity.

In the context of transaction and concurrency management, batch operations help coordinate multiple writes in a single request. This can reduce concurrency issues as operations are applied in a coherent unit. However, since Cassandra is a distributed system designed for scalability, its consistency model is eventual, meaning that without additional control mechanisms, batch operations cannot guarantee that all nodes will immediately see the same data version.

Lightweight Transactions (LWT) in Cassandra   Lightweight Transactions (LWT) in Cassandra are a mechanism for performing write operations with consensus control that ensures consistency in the presence of concurrency. LWTs use a Paxos-based protocol to achieve consensus among cluster nodes before applying the write. The syntax of an LWT includes the `IF` clause that checks a condition before performing an insert, delete, or update operation. Some examples are:

```
INSERT INTO keyspace_name.table_name (column1, column2) VALUES (value1, value2) IF
    ↪ NOT EXISTS;

UPDATE keyspace_name.table_name SET column1 = value WHERE column2 = value2 IF
    ↪ column3 = value3;
```

```
DELETE FROM keyspace_name.table_name WHERE column1 = value1 IF EXISTS;
```

LWTs are particularly useful for ensuring that write operations respect certain conditions, maintaining data consistency in concurrency scenarios. They are ideal for implementing conditional write operations, such as inserting a record only if it does not already exist (`IF NOT EXISTS`), or updating a record only if another condition is met (`IF column = value`).

When a client sends an LWT to the coordinator node, a Paxos-based protocol is applied in four phases:

- **Preparation Phase:** the coordinator sends a "Prepare" request to all replicas involved in the transaction. The participating nodes locally evaluate whether they can satisfy the condition specified in the `IF` clause before confirming their readiness to proceed. The replicas respond by indicating the highest proposal number they have accepted so far.
- **Proposal Phase:** based on the responses received during the preparation phase, the coordinator generates a new proposal with a unique number and the value to be written. This proposal is then sent to all involved replicas. The replicas compare the new proposal number with any previously accepted proposal numbers. If the new proposal number is higher, the replicas accept the proposal and respond to the coordinator.
- **Commit Phase:** this phase verifies the consensus of the transaction. Once the coordinator has received acceptances from the majority of replicas, it decides to commit the transaction. The coordinator then sends a "Commit" request to all replicas, indicating that the proposal should be applied. The replicas receive this request and mark the proposal as committed. The `IF` clause is re-evaluated before the final commit to ensure that the condition is still met.
- **Application Phase:** the replicas apply the committed value to their local data. Each replica confirms to the coordinator that the value has been correctly applied. Finally, once the coordinator has received confirmations from all replicas, it sends a confirmation to the client that the LWT has been successfully completed.

In the context of transaction and concurrency management, LWTs offer a strong consistency mechanism. Through the use of the protocol, an LWT ensures that only one consistent version of the data is written and made visible to all nodes in the cluster. This is crucial in applications where data integrity must be ensured, such as in a user registration system where each username must be unique.

However, LWTs come with a performance cost. The consensus process requires a series of information exchanges between nodes to reach an agreement, which slows down the operation compared to normal writes.

Application to data model   In the context of a flight booking system, poor concurrency management can cause numerous problems. A particularly serious case can occur when, for example, on a sold-out flight (all seats have been purchased), two people book the same seat due to a system error. To avoid such issues, it has been decided to use Lightweight Transactions (LWT) in the system.

In the implemented model, the crucial operation that requires concurrency control is: "Given a flight code, a seat code, and user information, insert the user data and change the seat status from 'Vacant' to 'Occupied'." In CQL language, this operation translates to:

```
UPDATE seat SET status = 'Occupied', balance = 'v1', date_of_birth = 'v2',
    ↪ document_info = 'v3', name = 'v4', surname = 'v5' WHERE flight = 'v6' AND
    ↪ id = 'v7';
```

Since a seat must be vacant before it can be occupied, we can update the operation using LWT by adding a clause that verifies the seat status is 'Vacant' before performing the operation. The CQL query will therefore be:

```
UPDATE seat SET status = 'Occupied', balance = 'v1', date_of_birth = 'v2',
    ↪ document_info = 'v3', name = 'v4', surname = 'v5' WHERE flight = 'v6' AND
    ↪ id = 'v7' IF status = 'Vacant';
```

In this way, it ensures that two people cannot book the same seat on the same flight.

In the Java code related to the implemented model, all simple checks, such as seat availability and user balance, are handled by the application. This approach improves performance. Specifically, in the updated code below, three methods have been added:

- `transaction`: given the departure and arrival airports, it executes a query to find available flights on that route, selects one (the first), checks if there are available seats on that flight, selects one if available, and calls execute to book the seat with the user data provided as input.
- `parallelTransaction`: this is a test method that essentially performs the same function as transaction but ensures that two people are simultaneously attempting to book the same seat on the same flight.
- `execute`: this method starts by checking the user's available balance. If the balance is sufficient, it proceeds with booking the flight. After sending the UPDATE operation, it checks if it was applied and informs the caller of the outcome. When called by parallelTransaction, it is expected that one booking will succeed while the other will not.

```java
package com.example.cassandra;

// Import necessary classes from DataStax driver and AWS MCS for authentication
import com.datastax.driver.core.*;
import software.aws.mcs.auth.SigV4AuthProvider;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class App
{
static Session session = null;

    // Method to get flights from a specific airport
    public static List<Flight> getFlightsFromAirport(String airportCode){
        // Create a query to select all flights from a specific departure airport
        SimpleStatement query = new SimpleStatement("SELECT * FROM flight WHERE
            ↪ departure='" + airportCode + "';");
        return getFlights(query);
    }

    // Helper method to execute a query and return a list of Flight objects
    private static List<Flight> getFlights(SimpleStatement query) {
        ResultSet rs = session.execute(query);
        List<Flight> flightList = new ArrayList<>();

        // Iterate through the result set and create Flight objects
        for (Row row : rs) {
            String departure = row.getString("departure");
            String day = "" + row.getDate("day");
            String destination = row.getString("destination");
```

```java
        String hour = "" + row.getTime("hour");
        String id = row.getString("id");
        String duration = row.getString("duration");
        int number_of_seats = row.getInt("number_of_seats");
        String operator = row.getString("operator");
        float price_per_person = row.getFloat("price_per_person");

        Flight flight = new Flight(departure, day, destination, hour, id,
            ↪ duration, number_of_seats, operator, price_per_person);
        flightList.add(flight);
    }

    return flightList;
}


// Method to get flights based on departure and destination airports
public static List<Flight> getFlightsFromDepartureAndDestination(String
    ↪ departureCode, String destinationCode) {
    // Create a query to select flights based on departure and destination
    SimpleStatement query = new SimpleStatement("SELECT * FROM flight WHERE
        ↪ departure='" + departureCode + "' AND destination='" +
        ↪ destinationCode + "';");
    return getFlights(query);
}


// Method to get airport details based on IATA code
public static Airport getAirportFromIATA(String airportCode) {
    // Create a query to select an airport based on its IATA code
    SimpleStatement query = new SimpleStatement("SELECT * FROM airport WHERE
        ↪ IATA_code='" + airportCode + "';");
    ResultSet rs = session.execute(query);

    Row row = rs.one();
    Airport airport = null;
    // From result set to Airport
    if (row != null) {
        String iata_code = row.getString("iata_code");
        String country = row.getString("country");
        String country_code = row.getString("country_code");
        String geo_point = row.getString("geo_point");
        String icao_code = row.getString("icao_code");
        String name = row.getString("name");
        String name_en = row.getString("name_en");
        String name_fr = row.getString("name_fr");
        String operator = row.getString("operator");
        String phone = row.getString("phone");
        int size = row.getInt("size");
        String website= row.getString("website");

        airport = new Airport(iata_code, country, country_code, geo_point,
            ↪ icao_code, name, name_en, name_fr, operator, phone, size,
            ↪ website);
    }

    return airport;
}


// Method to get seats for a specific flight
public static List<Seat> getSeats(String flightCode) {
    // Create a query to select all seats for a specific flight
    SimpleStatement query = new SimpleStatement("SELECT * FROM seat WHERE
        ↪ flight='" + flightCode + "';");
    ResultSet rs = session.execute(query);
```

```java
        List<Seat> seatList = new ArrayList<>();

        // Iterate through the result set and create Seat objects
        for (Row row : rs) {
            String flight = row.getString("flight");
            String id = row.getString("id");
            float balance = row.getFloat("balance");
            String date_of_birth = "" + row.getDate("date_of_birth");
            String document_info = row.getString("document_info");
            String name = row.getString("name");
            String status = row.getString("status");
            String surname = row.getString("surname");

            Seat seat = new Seat(flight, id, balance, date_of_birth,
                ↪ document_info, name, status, surname);
            seatList.add(seat);
        }

        return seatList;
}


// Method to get available (vacant) seats for a specific flight
public static List<Seat> getAvailableSeats(String flightCode) {
        List<Seat> seats = getSeats(flightCode);
        List<Seat> availableSeats = new ArrayList<>();

        // Filter seats to only include those that are vacant
        for (Seat seat : seats) {
            if(seat.getStatus().equals("Vacant")) {
                availableSeats.add(seat);
            }
        }

        return availableSeats;
}

// Method to get the status of a specific seat on a flight
public static String getSeatStatus(String flightCode, String idSeat) throws
    ↪ Exception {
        // Create a query to select a seat based on flight and seat ID
        SimpleStatement query = new SimpleStatement("SELECT * FROM seat WHERE
            ↪ flight='" + flightCode + "' AND id='"+ idSeat +"';");
        ResultSet rs = session.execute(query);

        Row row = rs.one();

        if (row != null) {
            return row.getString("status");
        }

        throw new Exception("This seat does not exist");
}

// Method to get the first available flight based on departure and destination
    ↪ airports
public static Flight getFirstAvailableFlight(String departureCode, String
    ↪ destinationCode) throws Exception{
        List<Flight> flights =
            ↪ getFlightsFromDepartureAndDestination(departureCode,
            ↪ destinationCode);

        if (flights.isEmpty()) {
```

```java
            throw new Exception("No flights found");
        }

        return flights.get(0);
    }

    // Method to get the first available seat on a specific flight
    public static Seat getFirstAvailableSeat(String flightCode) throws Exception {
        List<Seat> seats = getAvailableSeats(flightCode);

        if (seats.isEmpty()) {
            throw new Exception("No seats available");
        }

        return seats.get(0);
    }

    // Method to perform a transaction to book a flight and a seat for a user
    public static void transaction(User user, String departureCode, String
        ↪ destinationCode) {
        Flight flight;
        Seat seat;
        try {
            flight = getFirstAvailableFlight(departureCode, destinationCode);
            seat = getFirstAvailableSeat(flight.getId());
            System.out.println(execute(user, flight, seat));
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    // Method to perform parallel transactions for two users
    public static void parrallelTransaction(User user1, User user2, String
        ↪ departureCode, String destinationCode) {
        Flight flight;
        Seat seat;
        try {
            flight = getFirstAvailableFlight(departureCode, destinationCode);
            seat = getFirstAvailableSeat(flight.getId());

            // Create threads to execute transactions for both users in parallel
            Thread thread1 = new Thread(() -> { System.out.println(execute(user1,
                ↪ flight, seat)); });
            Thread thread2 = new Thread(() -> { System.out.println(execute(user2,
                ↪ flight, seat)); });

            thread1.start();
            thread2.start();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

    }

    // Method to execute the booking of a seat for a user
    public static String execute(User user, Flight flight, Seat seat) {
        if (flight.getPrice_per_person() > user.getBalance()) {
            return "Insufficient balance";
        }

        user.setBalance(user.getBalance() - flight.getPrice_per_person());

        String updateQuery = "UPDATE seat SET status = 'Occupied', balance = ?,
```

```java
                ↪ date_of_birth = ?, document_info = ?, name = ?, surname = ? WHERE
                ↪ flight = ? AND id = ? IF status = 'Vacant'";
        PreparedStatement updateStmt = session.prepare(updateQuery);
        updateStmt.setConsistencyLevel(ConsistencyLevel.LOCAL_QUORUM);

        LocalDate birthDate = LocalDate.fromYearMonthDay(
                Integer.parseInt(user.getDate_of_birth().substring(0, 4)),
                Integer.parseInt(user.getDate_of_birth().substring(5, 7)),
                Integer.parseInt(user.getDate_of_birth().substring(8, 10))
        );

        ResultSet resultSet = session.execute(updateStmt.bind(user.getBalance(),
            ↪ birthDate, user.getDocument_info(), user.getName(),
            ↪ user.getSurname(), seat.getFlight(), seat.getId()));

        if (!resultSet.wasApplied()) {
            return user.getName() + " " + user.getSurname() + ": seat reservation
                ↪ " + seat.getId() + " failed. It might already be 'Occupied'.";
        }

        return user.getName() + " " + user.getSurname() + ": seat " + seat.getId()
            ↪ + " successfully booked up.";
}

// Method to insert data into tables from text files
public static void insert(String table) {
    List<List<String>> dati = new ArrayList<>();

    String[] fileName = {"airport.txt", "flights.txt", "seats.txt"};

    for (String s: fileName) {
        try (BufferedReader br = new BufferedReader(new FileReader(s))) {
            String line;
            List<String> list = new ArrayList<>();

            // Read data from the text file and add it to the list
            while ((line = br.readLine()) != null) {
                list.add(line.trim());
            }

            dati.add(list);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Insert data into the specified table
    if(table.equals("airport")) {
        System.out.println("Insert Airport");
        for (String l1: dati.get(0)) {
            SimpleStatement query = new SimpleStatement("INSERT INTO airport
                ↪ (Size, Country, Country_code, Geo_Point, IATA_code,
                ↪ ICAO_code, Name, Name_en, Name_fr, Operator, Phone,
                ↪ Website) VALUES " + l1 + ";");
            session.execute(query);
        }
    } else {
        if(table.equals("flight")) {
            System.out.println("Insert Flight");
            for (String l2: dati.get(1)) {
                SimpleStatement query = new SimpleStatement("INSERT INTO
                    ↪ flight (ID, Departure, Destination, Number_of_Seats,
                    ↪ Day, Hour, Operator, Duration, Price_per_Person) VALUES
```

```
                              ↪ " + l2 + ";");
                    session.execute(query);
                }
            } else {
                System.out.println("Insert Seat");
                for (String l3: dati.get(2)) {
                    SimpleStatement query = new SimpleStatement("INSERT INTO seat
                        ↪ (Flight, ID, Status, Name, Surname, Document_Info,
                        ↪ Date_of_Birth, Balance) VALUES " + l3 + ";");
                    session.execute(query);
                }
            }
        }
    }
}

// Main method to initialize the Cassandra session and perform setup tasks
public static void main( String[] args ) {
    String endPoint = "cassandra.eu-north-1.amazonaws.com";
    int portNumber = 9142;

    try {
        // Initialize Cassandra session with AWS MCS authentication
        session = Cluster.builder()
                .addContactPoint(endPoint)
                .withPort(portNumber)
                .withAuthProvider(new SigV4AuthProvider("eu-north-1"))
                .withSSL()
                .build()
                .connect();

        System.out.println("Connected to Cassandra!");
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Create keyspace and use it
    session.execute("CREATE KEYSPACE IF NOT EXISTS my_airport WITH replication
        ↪ = "
            + "{'class':'SimpleStrategy','replication_factor':3};");

    session.execute("USE my_airport;");

    // Get list of existing tables before creating new ones
    ResultSet rs = session.execute("SELECT table_name FROM
        ↪ system_schema.tables " +
            "WHERE keyspace_name = 'my_airport' ");

    List<String> list_before = new ArrayList<>();
    for (Row row : rs) {
        list_before.add(row.getString("table_name"));
    }

    // Create necessary tables if they don't exist
    session.execute("CREATE TABLE IF NOT EXISTS airport (Name_en text, Country
        ↪ text, Website text, Geo_Point text, ICAO_code text, Country_code
        ↪ text, Phone text, Name_fr text, Name text, IATA_code text, Size
        ↪ int, Operator text, PRIMARY KEY (IATA_code));");
    session.execute("CREATE TABLE IF NOT EXISTS flight (ID text,
        ↪ Number_of_Seats int, Day date, Hour time, Operator text, Duration
        ↪ text, Price_per_Person float, Destination text, Departure text,
        ↪ PRIMARY KEY ((Departure), Destination, Day, Hour, id));");
    session.execute("CREATE TABLE IF NOT EXISTS seat (Flight text, Status
        ↪ text, ID text, Name text, Surname text, Document_Info text,
```

```
        ↪ Date_of_Birth date , Balance float , PRIMARY KEY (( Flight ), ID ));" );

// Get list of tables after creation
rs = session.execute ("SELECT table_name FROM system_schema.tables " +
        "WHERE keyspace_name = 'my_airport' " );

List<String> list_after = new ArrayList <>();
for (Row row : rs) {
    list_after.add(row.getString("table_name"));
}

// Insert data into newly created tables
if (list_before.size() != list_after.size()) {
    for (String s : list_after) {
        if (!list_before.contains(s)) {
            insert(s);
        }
    }
}

// Test various queries and transactions
// Test Query getFlightsFromAirport
System.out.println("getFlightsFromAirport");
System.out.println(getFlightsFromAirport("BGY"));

// Test Query getAirportFromIATA
System.out.println("getAirportFromIATA");
System.out.println(getAirportFromIATA("BGY"));

// Test Query getFlightsFromDepartureAndDestination
System.out.println("getFlightsFromDepartureAndDestination");
System.out.println(getFlightsFromDepartureAndDestination("BGY", "AAL"));

// Test Query getAvailableSeats
System.out.println("getAvailableSeats");
System.out.println(getAvailableSeats("6677f9a27acf35542d8ef4df"));

// Test Query getSeatStatus
System.out.println("getSeatStatus");
try {
    System.out.println(getSeatStatus("6677f9a27acf35542d8ef4df", "10B"));
} catch (Exception e) {
    throw new RuntimeException(e);
}

User user1 = new User(200, "2001-11-14", "CA11111XT", "Mario", "Verdi");

// Test Simple Transaction
System.out.println("transaction");
transaction(user1, "BGY", "AAL");

// Test Failed Transaction
System.out.println("transaction with no balance");
user1.setBalance(0);
transaction(user1, "BGY", "AAL");

// Test Parallel Transaction
user1.setBalance(300);
User user2 = new User(300, "2000-12-01", "CAAAAAAAA", "Luca", "Rossi");
System.out.println("parrallelTransaction");
parrallelTransaction(user1, user2, "BGY", "AAL");
    }
}
```

An example of query execution with concurrent LWTs can be:

```
Trace ID: c1512810 -3a2f -11ef -9bb1 -0f67a87d0426
/2 Preparing c1587b12 -3a2f -11ef -da45 -6843 ca902dd1  47694ms
/2 Promising ballot c1587b12 -3a2f -11ef -da45 -6843 ca902dd1  65716ms
/2 Appending to commitlog 66332ms
/2 Adding to paxos memtable 66706ms
/2 Reading existing values for CAS precondition 69460ms
/2 Executing single -partition query on seat 71186ms
/2 Acquiring sstable references 71351ms
/2 Merging memtable contents 71474ms
/2 Read 1 live rows and 0 tombstone cells 72472ms
/2 CAS precondition is met; proposing client -requested updates for
    ↪ c1587b12 -3a2f -11ef -da45 -6843 ca902dd1  85767ms
/2 Accepting proposal
    ↪ Accepted (139394090253450002: c1587b12 -3a2f -11ef -da45 -6843 ca902dd1 ,
    ↪ 1720116225345000: key=6677 f9a27acf35542d8ef4df
Row: id=11E | balance=74, date_of_birth =2001-11-14, document_info =CA11111XT ,
    ↪ name=Mario , status=Occupied , surname=Verdi) 88749ms
/2 Appending to commitlog 91469ms
/2 Adding to paxos memtable 91883ms
/2 Committing proposal
    ↪ Committed (139394090253450002: c1587b12 -3a2f -11ef -da45 -6843 ca902dd1 ,
    ↪ 1720116225345000: key=6677 f9a27acf35542d8ef4df
Row: id=11E | balance=74, date_of_birth =2001-11-14, document_info =CA11111XT ,
    ↪ name=Mario , status=Occupied , surname=Verdi) 99883ms
/2 Appending to commitlog 100387ms
/2 Adding to seat memtable 101132ms
/2 Appending to commitlog 108416ms
/2 Adding to paxos memtable 108859ms
/2 CAS applied successfully 110613ms


Trace ID: c151281a -3a2f -11ef -9bb1 -0f67a87d0426
/2 Preparing c156f472 -3a2f -11ef -c9d4 -a4b8589ef39a  41340ms
/2 Promising ballot c156f472 -3a2f -11ef -c9d4 -a4b8589ef39a  54533ms
/2 Appending to commitlog 60670ms
/2 Adding to paxos memtable 61286ms
/2 Reading existing values for CAS precondition 68610ms
/2 Executing single -partition query on seat 70832ms
/2 Acquiring sstable references 71145ms
/2 Merging memtable contents 71372ms
/2 Read 1 live rows and 0 tombstone cells 72515ms
/2 CAS precondition is met; proposing client -requested updates for
    ↪ c156f472 -3a2f -11ef -c9d4 -a4b8589ef39a  87881ms
/2 Rejecting proposal for
    ↪ Accepted (139394090253350002: c156f472 -3a2f -11ef -c9d4 -a4b8589ef39a ,
    ↪ 1720116225335000: key=6677 f9a27acf35542d8ef4df
Row: id=11E | balance=174, date_of_birth =2000-12-01, document_info =CAAAAAAAA ,
    ↪ name=Luca , status=Occupied , surname=Rossi) because inProgress is now
    ↪ c1587b12 -3a2f -11ef -da45 -6843 ca902dd1  94375ms
/2 Paxos proposal not accepted (pre -empted by a higher ballot) 95414ms
/2 Preparing c1621802 -3a2f -11ef -110e -7f625f30ee43  110763ms
/2 Promising ballot c1621802 -3a2f -11ef -110e -7f625f30ee43  112032ms
/2 Appending to commitlog 112670ms
/2 Adding to paxos memtable 112945ms
/2 Reading existing values for CAS precondition 114621ms
/2 Executing single -partition query on seat 115328ms
/2 Acquiring sstable references 115805ms
/2 Merging memtable contents 116994ms
/2 Read 1 live rows and 0 tombstone cells 118135ms
/2 CAS precondition does not match current values [my_airport.seat]
    ↪ key=6677 f9a27acf35542d8ef4df
    ↪ partition_deletion =deletedAt = -9223372036854775808 , localDeletion =2147483647
    ↪ columns=[[] | [balance date_of_birth document_info name status surname ]]
```

```
Row[info=[ts=1720115220302524] ]: id=11E | [balance=74 ts=1720116225345000],
    ↪ [date_of_birth=2001-11-14 ts=1720116225345000], [document_info=CA11111XT
    ↪ ts=1720116225345000], [name=Mario ts=1720116225345000], [status=Occupied
    ↪ ts=1720116225345000], [surname=Verdi ts=1720116225345000] 128739ms
/2 CAS precondition is met; proposing client-requested updates for
    ↪ c1621802-3a2f-11ef-110e-7f625f30ee43 129063ms
/2 Accepting proposal
    ↪ Accepted(139394090254080002:c1621802-3a2f-11ef-110e-7f625f30ee43,
    ↪ 1442880000000000:key=6677f9a27acf35542d8ef4df) 130634ms
/2 Appending to commitlog 131213ms
/2 Adding to paxos memtable 131646ms
/2 CAS did not apply 138740ms
```

Large Volume Data Management

In this chapter, we will delve into how it is possible to manage and read large volumes of data in MongoDB/Cassandra, using the `"Airports"` database. We will explore how to efficiently distribute the database across multiple nodes to optimize performance and ensure fault tolerance. Additionally, we will discuss the mechanisms MongoDB/Cassandra employs to handle failures within the cluster and strategies to mitigate potential issues. Finally, we will provide detailed instructions on using Docker for deployment and management of the MongoDB cluster.

Docker

Docker is a platform and toolset designed to simplify the creation, deployment, and management of applications using containerization technology. Containers allow developers to package an application and its dependencies into a standardized unit for software development, ensuring consistency across different computing environments. At its core, Docker utilizes the concept of containers, which are lightweight and portable environments that include everything needed to run an application: code, runtime, system tools, libraries, and settings. Unlike traditional virtual machines, containers share the host operating system kernel and are isolated processes, making them efficient and fast. Docker provides several key components to facilitate container management:

1. **Docker Engine:** the core of Docker, responsible for creating and managing containers on a single host. It includes a server and a command-line interface (CLI) tool (`docker`) for interacting with containers.

2. **Images:** docker images are read-only templates used to create containers. An image includes the application code, runtime, libraries, and dependencies needed for the application to run.

3. **Containers:** running instances of Docker images. Containers can be started, stopped, moved, and deleted using Docker's CLI commands. They encapsulate the application and its environment, ensuring consistency across different environments, from development to production.

4. **Dockerfile:** a text document that contains instructions for building a Docker image. It defines the environment inside the container, including base image, dependencies, environment variables, and commands to run when the container starts.

5. **Docker Compose:** a tool for defining and running multi-container Docker applications. It uses a YAML file (`docker-compose.yml`) to configure the application's services and their dependencies. Docker Compose simplifies the process of defining, managing, and scaling multi-container applications.

How Docker Works   Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, libraries, etc. This ensures that the application runs consistently across different computing environments. Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. Containers are built from images, which are essentially snapshots of a container's filesystem. Images are stored in repositories, such as Docker Hub, and can be pulled and pushed between different Docker installations.

Benefits of Using Docker

- **Portability:** docker containers can run on any machine that has Docker installed, regardless of the underlying operating system. This eliminates the "it works on my machine" problem and streamlines deployment across different environments.

- **Isolation:** containers isolate applications from each other and from the underlying infrastructure, ensuring that changes or updates to one application do not affect others running on the same host.

- **Efficiency:** containers share the host operating system kernel, making them lightweight and efficient compared to traditional virtual machines. Multiple containers can run on the same hardware without significant overhead.

- **Scalability:** docker's architecture allows applications to be quickly deployed and scaled horizontally by adding or removing containers, making it ideal for microservices architectures and distributed applications.

Large Volume Data Management in MongoDB

We can't talk about reading large volumes of data in MongoDB without explaining what sharding is first. Even though we went over sharding in detail in the Architecture section of this document, we will explain it again here, in a more focused way for better reading comprehension. Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. This is essential for managing large volumes of data by spreading the data across several servers, ensuring that no single server is overwhelmed by requests or storage demands.

As an arbitrary number, we will choose to distribute the `"airportCollection"` across four nodes. In order to do this, we will need to define a sharding key. The sharding key determines how the data is partitioned and distributed. In our case, the ideal sharding key might be the `"Country_code"` attribute since it ensures a relatively even distribution of data based on the airports' countries. This approach helps in balancing the load and optimizing query performance. To be able to carry out the operations we need, the use of Docker is necessary. Therefore, we show below what it is, how it works, and how to configure it to handle the needs of our application.

Using Docker with MongoDB   In the context of MongoDB, Docker simplifies the deployment and management of MongoDB clusters. By defining MongoDB instances as containers and using Docker Compose to orchestrate them, developers can easily set up replica sets and sharded clusters. This approach ensures consistent database configurations across development, testing, and production environments, enhancing reliability and scalability. In the next sections, we will explore how to leverage Docker for deploying and managing a MongoDB cluster to handle large volumes of data efficiently, focusing on distributing the "airportCollection" across multiple nodes and optimizing performance with sharding based on the `"Country_code"` attribute.

To better explain what this all means, in the context of sharding in MongoDB, shards do not directly contain collection data such as `"airportCollection"`. Instead, the collection data is distributed among the various shards based on the specified sharding key. So, what we will do is shard the `"airportCollection"` collection using the `"Country_code"` field as the sharding key. Therefore, MongoDB will distribute the "airportCollection" documents among the various shards based on the value of the `"Country_code"` field. This means that each shard will contain documents corresponding to specific `"Country_code"` values. For example, suppose we have four shards and that "airportCollection" contains airports from various countries, such as the USA, UK, France, and Italy. MongoDB will distribute documents so that all documents with `"Country_code"` = `"USA"` can end up on one shard, those with `"Country_code"` = `"UK"` on another shard, and so on. So, each shard will contain airports from different countries. In other words, each Shard will contain different data from the other, in such a way as to correctly distribute the entire database.

A question that may arise spontaneously is wondering how the `"Country_codes` will be distributed, given that we have much more than four countries, but only four nodes. It is good to know that it is completely normal and standard for MongoDB to automatically manage data distribution when using sharding without manually specifying which country codes go on which shards (in other words, in the explanations that follow, there will be no real phase of assigning the individual `"Country_codes"` to the individual nodes). MongoDB takes care of this by automatically using chunking and ensuring good data balancing between shard nodes.

Docker Compose Configuration   First of all, we need to ensure Docker is installed on your system. Assuming that Docker is installed, we need to place the following `docker-compose.yml` file in the root directory of our project. This location ensures that the Docker commands and configuration are easily accessible and maintainable.

Reference Note: the reason why the ports of the following YAML document, and consequently of the documentation that follows the code, have not been implemented according to an ascending order is because, in the working machine on which this report was written, certain ports they were already occupied by other running Docker containers, and therefore different values were used to ensure that each container was associated with a unique port and in a conflict-free manner. Nobody forbids the re-elaboration of the following YAML document by choosing entirely different ports. This must be done provided that the ports that the developer wants to use are free. The code is as follows

```
services:
  configsvr:
    image: mongo:4.4.0-bionic
    container_name: configsvr
    command: ["mongod", "--configsvr", "--replSet", "rsConfig", "--port",
      ↪ "27019"]
    ports:
      - 27019:27019
    volumes:
      - configsvr:/data/db

  shard1:
    image: mongo:4.4.0-bionic
    container_name: shard1
    command: ["mongod", "--shardsvr", "--replSet", "rsShard1", "--port", "27018"]
    ports:
      - 27018:27018
    volumes:
      - shard1:/data/db

  shard2:
```

```
        image: mongo:4.4.0-bionic
        container_name: shard2
        command: ["mongod", "--shardsvr", "--replSet", "rsShard2", "--port", "27018"]
        ports:
            - 27020:27018
        volumes:
            - shard2:/data/db

    shard3:
        image: mongo:4.4.0-bionic
        container_name: shard3
        command: ["mongod", "--shardsvr", "--replSet", "rsShard3", "--port", "27018"]
        ports:
            - 27021:27018
        volumes:
            - shard3:/data/db

    shard4:
        image: mongo:4.4.0-bionic
        container_name: shard4
        command: ["mongod", "--shardsvr", "--replSet", "rsShard4", "--port", "27018"]
        ports:
            - 27022:27018
        volumes:
            - shard4:/data/db

    mongos:
        image: mongo:4.4.0-bionic
        container_name: mongos
        command: ["mongos", "--configdb", "rsConfig/configsvr:27019", "--port",
            ↪ "27017"]
        ports:
            - 27017:27017
        depends_on:
            - configsvr
            - shard1
            - shard2
            - shard3
            - shard4

volumes:
    configsvr:
    shard1:
    shard2:
    shard3:
    shard4:
```

A Docker-compose file is a YAML configuration file used to define and manage multi-container applications with Docker. It simplifies the management of Dockerized applications by allowing easy definition of multiple containers, their configurations, networks, and volumes required for them to interact with each other. Regarding the specific `docker-compose.yml` file provided:

This file sets up an environment for a distributed MongoDB database system in a clustered mode. It lists several services (`services`), each representing a specific Docker container:

- **Configuration Server (`configsvr`):** a MongoDB configuration server (`mongod`) running in configuration server mode (`--configsvr`). It is part of a replica set named `rsConfig` and listens on port 27019. It has a dedicated volume (`configsvr`) for persistent storage.

- **Shard Nodes (`shard1`, `shard2`, `shard3`, `shard4`):** four MongoDB shard nodes (`mongod`), each configured as a shard server (`--shardsvr`) and part of their respective replica sets (`rsShard1`,

`rsShard2`, `rsShard3`, `rsShard4`). They all listen on port 27018 internally, but are exposed on different host ports (27018, 27020, 27021, 27022). Each shard has a dedicated volume for data storage.

- **Routing Server (`mongos`):** a MongoDB routing server (`mongos`) that routes queries between clients and the shard nodes. It connects to the configuration server (`configsvr`) on port 27019 and listens on port 27017. It depends on the configuration server and all shard nodes to be up and running.

Additionally, the file defines several volumes (`volumes`) to ensure data persistence for each service, maintaining data integrity even after container destruction or restart. To recap, this Docker-compose file orchestrates a distributed MongoDB environment, managing complex network and volume configurations across multiple containers required for a scalable and reliable database system.

Deploying the MongoDB Cluster

1. **Start the MongoDB Cluster:** Create a `docker-compose.yml` file with your desired configuration, then start the cluster using the following command.

2. This will start all containers defined in the `docker-compose.yml` file in detached mode (`-d`).

```
docker-compose up -d
```

3. **Initialize the Configuration Server:** Connect to the configuration server container and initiate the replica set.

```
docker exec -it configsvr bash
mongo --port 27019 --eval 'rs.initiate({_id: "rsConfig", configsvr: true,
    ↪ members: [{ _id : 0, host : "configsvr:27019" }]})'
exit
```

4. **Initialize Shard Servers:** Connect to each shard container and initiate the replica sets. Repeat the following steps for each shard (shard1, shard2, shard3, shard4):

```
docker exec -it shard1 bash
mongo --port 27018 --eval 'rs.initiate({_id: "rsShard1", members: [{ _id: 0,
    ↪ host: "mongod11:27018" }, { _id: 1, host: "mongod12:27018" }, { _id:
    ↪ 2, host: "mongod13:27018" }]})'
exit

docker exec -it shard2 bash
mongo --port 27018 --eval 'rs.initiate({_id: "rsShard2", members: [{ _id: 0,
    ↪ host: "mongod21:27018" }, { _id: 1, host: "mongod22:27018" }, { _id:
    ↪ 2, host: "mongod23:27018" }]})'
exit

docker exec -it shard3 bash
mongo --port 27018 --eval 'rs.initiate({_id: "rsShard3", members: [{ _id: 0,
    ↪ host: "mongod31:27018" }, { _id: 1, host: "mongod32:27018" }, { _id:
    ↪ 2, host: "mongod33:27018" }]})'
exit

docker exec -it shard4 bash
mongo --port 27018 --eval 'rs.initiate({_id: "rsShard4", members: [{ _id: 0,
    ↪ host: "mongod41:27018" }, { _id: 1, host: "mongod42:27018" }, { _id:
    ↪ 2, host: "mongod43:27018" }]})'
exit
```

5. **Configure the MongoS Router:** Connect to the `mongos` container and add the shards to the cluster.

```
docker exec -it mongos bash
mongo --port 27017 --eval
    ↪ 'sh.addShard("rsShard1/mongod11:27018,mongod12:27018,mongod13:27018")'
mongo --port 27017 --eval
    ↪ 'sh.addShard("rsShard2/mongod21:27018,mongod22:27018,mongod23:27018")'
mongo --port 27017 --eval
    ↪ 'sh.addShard("rsShard3/mongod31:27018,mongod32:27018,mongod33:27018")'
mongo --port 27017 --eval
    ↪ 'sh.addShard("rsShard4/mongod41:27018,mongod42:27018,mongod43:27018")'
```

6. **Enable Sharding on the Database and Collection:** Enable sharding on your desired database (`myDatabase`) and collection (`myCollection`), specifying the sharding key.

```
docker exec -it mongos bash
mongo --port 27017 --eval 'sh.enableSharding("myDatabase")'
mongo --port 27017 --eval 'sh.shardCollection("myDatabase.myCollection", {
    ↪ "Country_code": 1 })'
```

As we just saw, Docker simplifies the deployment and management of MongoDB clusters. Docker containers can encapsulate MongoDB processes, ensuring consistency across different environments. Using Docker allows for easy scaling and management of services, which is crucial for maintaining a robust sharded cluster. Let's quickly recap the benefits of using Docker in a scenario like the one we just mentioned:

- **Isolation:** Each MongoDB instance runs in its own container, ensuring isolation.
- **Portability:** Docker images can be easily moved across different environments without compatibility issues.
- **Scalability:** Adding more shard nodes or configuration servers is straightforward with Docker.
- **Consistency:** Docker ensures the same environment setup across different deployments.

Handling Node Failure   After having analyzed and implemented the functioning of Docker in favor of the sharding process, it is good to spend a few moments to talk about the error management on large volumes that MongoDB performs by design. MongoDB's sharding architecture is designed to be fault-tolerant. Each shard is a replica set, which means data is replicated across multiple servers within the shard. This replication ensures data availability even if a node fails. If a node in a shard fails, the replica set protocol elects a new primary from the remaining members. The system continues to operate with minimal disruption. The following mitigation strategies are suggested to lower the risk of failure when using sharding in a MongoDB database:

1. **Replica Sets:** ensure each shard is a replica set with at least three members.
2. **Regular Backups:** perform regular backups to recover data in case of catastrophic failures.
3. **Monitoring:** use monitoring tools like MongoDB Atlas, which provides monitoring and alerts for cluster health and performance.

Let's now analyze what would happen in the event of failure of one of the nodes. Suppose one of the nodes in `shard1` fails:

1. **Detection:** MongoDB detects the failure and the replica set initiates an election.
2. **Election:** a new primary is elected from the remaining nodes.
3. **Recovery:** the failed node is either repaired or replaced and then re-added to the replica set.

After these considerations, we can deduce that Sharding is a powerful feature in MongoDB that allows for horizontal scaling and efficient management of large datasets. By distributing the "airportCollection" across four nodes using a sharding key, we ensure optimal performance and fault tolerance. Docker further simplifies deployment and management of these clusters. Understanding and implementing these strategies is crucial for handling large volumes of data

effectively. By leveraging the connection string provided, users can easily connect to and manage their MongoDB cluster hosted on MongoDB Atlas. The combination of sharding, fault tolerance, and containerization with Docker provides a robust solution for managing large volumes of data in MongoDB.

Reference Note: the whole procedure we have just shown can be performed in a much simpler way using MongoDB native commands. In particular, the entire operation just detailed can be summarized through the use of the following very simple commands:

```
mongosh "mongodb+srv://
        admin:admin@learningmongodb.hikoksa.mongodb.net/" +
            ↪ "Airports?retryWrites=true&w=majority"
use Airports
sh.enableSharding("Airports")
sh.shardCollection("Airports.airportCollection", { "Country_code": 1 })
```

The reason why the report was based on the docker counterpart is that the MongoDB-specific commands just mentioned are not available for the M0 free cluster tier. Therefore, to make the opportunity to use sharding accessible to anyone, the Docker version was shown instead. In addition to this, the Docker version brings with it numerous advantages, which we have already had the opportunity to mention in the previous subchapters.

For the purpose of testing the functionality of the newly built Docker nodes, the MongoDBSharded-Connection class was written. The class code is described as follows. Reference note: the class relies on other auxiliary classes called Airport, Flight and Seat, which are used to convert the data contained on the Atlas server into a format ready to be assimilated by the main class, and consequently by the structure we just created on Docker. For simplicity and brevity, auxiliary classes will not be listed below. However, the MongoDBShardedConnection code will be present below, and commented in a clear and exhaustive manner.

```java
package org.example;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;

import org.bson.Document;
import org.json.JSONArray;
import org.json.JSONObject;
import org.json.JSONTokener;

import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;

/**
* The {@code MongoDBShardedConnection} class provides methods for reading airport
    ↪ data from a JSON file and
* inserting it into a MongoDB collection. This class demonstrates the process of
    ↪ parsing complex JSON data,
* constructing MongoDB documents, and managing connections to a MongoDB sharded
    ↪ cluster.
* <p>
* The primary functionalities include:
* </p>
* Reading and parsing a JSON file containing airport data.
* Constructing MongoDB documents from the parsed JSON data.
* Connecting to a MongoDB database and inserting the documents into a specified
    ↪ collection.
```

```
*
* <p>The JSON structure includes nested arrays representing flights and seats,
    ↪ which are also parsed and included
* in the corresponding MongoDB documents.
*
*/
public class MongoDBShardedConnection {

    /**
     * Reads airport data from a JSON file and constructs a list of MongoDB
         ↪ documents.
     * The JSON file is expected to contain an array of airport objects, each of
         ↪ which may
     * contain nested arrays for flights and seats.
     *
     * @return a list of {@link Document} objects representing the airport data.
     */
    public static List<Document> getData() {
        List<Document> airportDocuments = new ArrayList<>();

        try (FileReader reader = new FileReader("Airports Modeling Export.json")) {
            // Create a JSONTokener from the FileReader
            JSONTokener tokener = new JSONTokener(reader);

            // Create a JSONArray from the tokener
            JSONArray jsonArray = new JSONArray(tokener);

            // Iterate through the JSON array elements
            for (int i = 0; i < jsonArray.length(); i++) {
                // Get the current JSON object
                JSONObject jsonObject = jsonArray.getJSONObject(i);

                // Extract desired fields for the airport document
                String geoPoint = jsonObject.optString("Geo_Point", "");
                String name = jsonObject.optString("Name", "");
                String nameEn = jsonObject.optString("Name_(en)", "");
                String nameFr = jsonObject.optString("Name_(fr)", "");
                String iataCode = jsonObject.optString("IATA_code", "");
                String icaoCode = jsonObject.optString("ICAO_code", "");
                String operator = jsonObject.optString("Operator", "");
                String country = jsonObject.optString("Country", "");
                String countryCode = jsonObject.optString("Country_code", "");
                int size = jsonObject.optInt("Size", 0);

                // Extract the array of flights for the airport
                JSONArray flightsArray = jsonObject.getJSONArray("Flights");
                List<Flight> flights = new ArrayList<>();

                // Iterate through the flights array elements
                for (int j = 0; j < flightsArray.length(); j++) {
                    JSONObject flightObject = flightsArray.getJSONObject(j);

                    // Extract desired fields for the flight
                    String flightId = flightObject.optString("ID", "");
                    int numberOfSeats = flightObject.optInt("Number_of_Seats", 0);
                    String day = flightObject.optString("Day", "");
                    String hour = flightObject.optString("Hour", "");
                    String flightOperator = flightObject.optString("Operator", "");
                    String duration = flightObject.optString("Duration", "");
                    int pricePerPerson = flightObject.optInt("Price_per_Person",
                        ↪ 0);

                    // Extract the array of seats for the flight
```

```java
                JSONArray seatsArray = flightObject.getJSONArray("Seats");
                List<Seat> seats = new ArrayList<>();

                // Iterate through the seats array elements
                for (int k = 0; k < seatsArray.length(); k++) {
                    JSONObject seatObject = seatsArray.getJSONObject(k);

                    // Extract desired fields for the seat
                    String status = seatObject.optString("Status", "");
                    String seatId = seatObject.optString("ID", "");
                    String nameOnTicket = seatObject.optString("Name", "");
                    String surnameOnTicket = seatObject.optString("Surname",
                        ↪ "");
                    String documentInfo =
                        ↪ seatObject.optString("Document_Info", "");
                    String dateOfBirth = seatObject.optString("Date_of_Birth",
                        ↪ "");
                    int balance = seatObject.optInt("Balance", 0);

                    // Create a Seat object and add it to the list of seats
                    Seat seat = new Seat(status, seatId, nameOnTicket,
                        ↪ surnameOnTicket, documentInfo, dateOfBirth,
                        ↪ balance);
                    seats.add(seat);
                }

                // Create a Flight object and add it to the list of flights
                Flight flight = new Flight(flightId, numberOfSeats, day, hour,
                    ↪ flightOperator, duration, pricePerPerson, seats);
                flights.add(flight);
            }

            // Create an Airport object and add it to the list of documents
            Airport document = new Airport(geoPoint, name, nameEn, nameFr,
                ↪ iataCode, icaoCode, operator, country, countryCode, size,
                ↪ flights);
            airportDocuments.add(document.toDocument());
        }

    } catch (Exception e) {
        e.printStackTrace();
    }

    return airportDocuments;
}

/**
 * Main method to establish a connection to the MongoDB sharded cluster,
     ↪ retrieve airport data,
 * and insert it into a specified MongoDB collection.
 *
 * @param args the command-line arguments (not used).
 */
public static void main(String[] args) {
    // Connection to the mongos router
    String connectionString = "mongodb://localhost:27017";
    try (MongoClient mongoClient = MongoClients.create(connectionString)) {
        MongoDatabase database = mongoClient.getDatabase("myDatabase");
        System.out.println("Connected to database: " + database.getName());

        MongoCollection<Document> collection =
            ↪ database.getCollection("myCollection");
        List<Document> airports = getData();
```

```
                collection.insertMany(airports);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example of role variation between nodes   Consider `shard1` having one primary node (PRI-
MARY) and two secondary nodes (SECONDARY). The `rs.status` command displays:

```
{
  "set" : "rsShard1",
  "members" : [
    {
      "_id" : 0,
      "name" : "mongod11:27018",  // this is the primary node
      "stateStr" : "PRIMARY",
      // ...
    },
    {
      "_id" : 1,
      "name" : "mongod12:27018",  // this is a secondary node
      "stateStr" : "SECONDARY",
      // ...
    },
    {
      "_id" : 2,
      "name" : "mongod13:27018",  // this is a secondary node
      "stateStr" : "SECONDARY",
      // ...
    }
  ]
}
```

If the primary node (`mongod11`) were to malfunction or become unreachable for any reason, MongoDB
automatically elects a new node as primary. This election process is managed internally by the
replica set.

After the primary node malfunctions, running `rs.status()` again might show output similar to this:

```
{
  "set" : "rsShard1",
  "members" : [
    {
      "_id" : 0,
      "name" : "mongod11:27018",  // this node is no longer primary, could be
          ↪ offline or in recovery state
      "stateStr" : "DOWN",
      // ...
    },
    {
      "_id" : 1,
      "name" : "mongod12:27018",  // the previously secondary node is now elected
          ↪ as primary
      "stateStr" : "PRIMARY",
      // ...
    },
    {
      "_id" : 2,
      "name" : "mongod13:27018",  // this is still a secondary node
      "stateStr" : "SECONDARY",
```

```
        // ...
    }
  ]
}
```

In the above example, `mongod12:27018` has been elected as the new primary node (PRIMARY) of the `rsShard1` replica set, since the originally primary node (`mongod11`) is now in a DOWN state.

Large Volume Data Management in Cassandra

Managing large volumes of data in Cassandra involves a combination of distribution, replication, load balancing, partitioning, and sharding techniques.

Peer-to-Peer Architecture   Cassandra uses a masterless peer-to-peer architecture where every node in the cluster is equally important. This avoids bottlenecks and single points of failure, enhancing scalability and reliability.

Data Distribution, Partitioning, and Sharding   In Cassandra, data is partitioned and distributed among the nodes in the cluster using a partitioning hash. The partition key is processed by a hash function that determines the storage node, ensuring uniform data distribution and quick access. This process, known as sharding, divides the data into smaller segments called shards, improving system scalability and load balancing since each node manages only a portion of the total data, reducing the load on each node and allowing the system to grow linearly by adding new nodes.

Data Replication   Cassandra offers strong fault tolerance through data replication. Each piece of data is replicated across multiple nodes based on the configured replication factor. Data replicas not only provide fault tolerance but also improve system availability and resilience. When a node fails, replicas on other nodes ensure data accessibility. Cassandra also allows configuring replicas to be distributed across different data centers, further enhancing resilience against large-scale failures like power outages or natural disasters in a single data center.

Eventual Consistency   Cassandra uses an eventual consistency model, allowing greater flexibility in managing distributed data. Writes are quickly distributed among nodes, and synchronization happens later, enabling high availability and low latency. Users can configure the required consistency level for read and write operations based on the specific needs of the application.

Data Compaction   Cassandra uses a technique called "compaction" to manage storage and optimize performance. During compaction, data is organized and consolidated into larger files, eliminating obsolete data and reducing disk fragmentation. This process improves read efficiency and reduces disk space usage.

Load Balancing   Cassandra automatically balances the load among the nodes in the cluster. When new nodes are added, existing data is redistributed to take advantage of the new capacity. This allows horizontal scaling of the cluster without significant service disruptions.

Performance Optimization   Cassandra is optimized for handling high read and write throughput. It uses an append-only write model, where data is always added to the end of existing files, reducing write time. Reads are managed using in-memory indexes, allowing quick access to the requested data.

Monitoring and Optimization To effectively manage large data volumes, it is essential to continuously monitor cluster performance. Cassandra provides various tools and metrics to monitor cluster status, resource usage, and query performance. Additionally, configurations can be optimized based on specific workloads to improve efficiency.

Failure Management in Cassandra In Cassandra, managing failures is crucial to ensure continuous availability and data consistency within the cluster.

Gossip Protocol The gossip protocol is a key mechanism for managing failures in distributed systems. This protocol relies on a simple and robust communication method inspired by how information spreads in a human community: through gossip. The protocol operates in four phases: 1. Periodic Information Exchange: Cluster nodes periodically exchange information about their status and that of other nodes. This exchange occurs at regular intervals and can involve a random number of nodes. 2. Information Propagation: Each node, after receiving updated information from other nodes, transmits it to additional nodes in the cluster. This continuous propagation process allows information to spread quickly throughout the system. 3. Failure Detection: When a node stops responding to gossip messages, other nodes can detect its absence. After a certain number of failed communication attempts, the node is considered unavailable or "failed." 4. Global State Update: Once a failure is detected, the information is propagated to other nodes. This allows all cluster nodes to have an updated and consistent view of the state of other nodes, facilitating failure management and reassignment of the failed node's tasks.

The gossip protocol is highly scalable. Even in large clusters with thousands of nodes, the time needed to disseminate information remains contained. Due to its decentralized nature, the gossip protocol is inherently fault-tolerant. There is no single point of failure since every node contributes to information dissemination. Gossip uses a reduced number of messages for information dissemination compared to centralized protocols, reducing network load. The protocol is relatively simple to implement, making it a popular choice for many distributed systems.

Consistency Levels Consistency levels are critical for managing read and write operations. When performing a write operation, it is possible to specify a consistency level that defines how many nodes must confirm the write for it to be considered valid. For example, selecting the "QUORUM" consistency level requires that the majority of nodes holding the replicas confirm the operation.

Hinted Handoff In case a node becomes unavailable, Cassandra uses a technique called hinted handoff to ensure continuity of operations. This technique is crucial for maintaining data reliability and availability in a distributed cluster.

When a node is temporarily out of service, the data that should be written to that node is stored as "hints" on another node in the cluster. This alternative node records the information that specific data was supposed to be written to the inactive node. The system continues to operate without interrupting read and write operations, as other available nodes handle the data.

The hinted handoff not only allows uninterrupted data writes but also ensures no information is lost during the original node's downtime. Once the inactive node comes back online and becomes available again, the node storing the hints transfers the missing data to the original node. This process ensures all nodes can update and synchronize correctly, restoring data integrity in the cluster.

Read Repair   During read operations, Cassandra verifies that all data replicas are consistent with each other. This process is known as read repair. When a client requests data, Cassandra reads it from all configured replicas (the number depends on the chosen consistency level). If the system detects discrepancies between replicas, it automatically performs a repair operation to correct them. This repair can be synchronous or asynchronous:

- Synchronous: When a read requires synchronous read repair, Cassandra not only returns the correct data to the client but also updates all involved replicas before completing the read operation.
- Asynchronous: In the case of asynchronous read repair, Cassandra returns the correct data to the client and schedules the update of replicas for a later time.

This mechanism ensures replicas remain synchronized and that read data is always correct and up-to-date, improving data consistency within the cluster.

Manual Repair   In addition to automatic read repair, Cassandra provides tools for manual data repair, essential for maintaining cluster integrity and consistency over time. One such tool is the `nodetool repair` command. This command is used to manually synchronize replicas and correct any discrepancies that may have occurred due to unavailable nodes or failed writes.

Regular execution of `nodetool repair` is crucial, especially in environments where nodes may encounter failures or periodic maintenance. This command scans the cluster and compares data across various replicas, ensuring each replica contains the most up-to-date information and correcting any divergences.

The importance of `nodetool repair` lies in its ability to prevent the accumulation of inconsistencies that could compromise data integrity over the long term. In practice, diligent administration of Cassandra includes regular execution of the `nodetool repair` command as part of periodic cluster maintenance operations. This ensures that even in the presence of temporary failures or maintenance interventions, the system remains robust, consistent, and reliable.

Example of Gossip and Hinted Handoff in Cassandra   Consider the scenario with the following `docker-compose.yml`:

```yaml
services:
  cassandra-node1:
    image: cassandra:latest
    container_name: cassandra-node1
    environment:
      - CASSANDRA_DC=datacenter1
      - CASSANDRA_RACK=rack1
    ports:
      - "9042:9042"
    networks:
      - cassandra-net

  cassandra-node2:
    image: cassandra:latest
    container_name: cassandra-node2
    environment:
      - CASSANDRA_DC=datacenter1
      - CASSANDRA_RACK=rack1
      - CASSANDRA_SEEDS=cassandra-node1
    networks:
      - cassandra-net

  cassandra-node3:
    image: cassandra:latest
```

```
    container_name: cassandra-node3
    environment:
      - CASSANDRA_DC=datacenter1
      - CASSANDRA_RACK=rack1
      - CASSANDRA_SEEDS=cassandra-node1
    ports:
      - "9043:9042"
    networks:
      - cassandra-net

networks:
  cassandra-net:
    driver: bridge
```

Run the command:

```
docker-compose up -d
```

to start the containers.

Execute the commands:

```
docker exec -it cassandra-node1 bash
nodetool describecluster
```

to obtain the current status of the nodes:

```
root@a619f2004d70:/# nodetool describecluster
Cluster Information:
        Name: Test Cluster
        Snitch: org.apache.cassandra.locator.SimpleSnitch
        DynamicEndPointSnitch: enabled
        Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
        Schema versions:
                54e17321-3f2e-37ca-9b08-d91ba7bdd369: [172.22.0.2, 172.22.0.3,
                    ↪ 172.22.0.4]

Stats for all nodes:
        Live: 3
        Joining: 0
        Moving: 0
        Leaving: 0
        Unreachable: 0

Data Centers:
        datacenter1 #Nodes: 3 #Down: 0

Database versions:
        4.1.5: [172.22.0.2:7000, 172.22.0.3:7000, 172.22.0.4:7000]

Keyspaces:
        system_traces -> Replication class: SimpleStrategy {replication_factor=2}
        system_distributed -> Replication class: SimpleStrategy
            ↪ {replication_factor=3}
        system_auth -> Replication class: SimpleStrategy {replication_factor=1}
        system_schema -> Replication class: LocalStrategy {}
        system -> Replication class: LocalStrategy {}
```

Next, populate the database by running the following code, analogous to the AWS version explained
in previous chapters:

```
package com.example.cassandra;
```

```java
import com.datastax.oss.driver.api.core.DefaultConsistencyLevel;
import com.datastax.oss.driver.api.core.config.DriverConfigLoader;
import com.datastax.oss.driver.api.core.CqlSession;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.List;

import com.datastax.oss.driver.api.core.cql.ResultSet;
import com.datastax.oss.driver.api.core.cql.SimpleStatement;
import com.datastax.oss.driver.api.core.cql.Row;

public class App {

    public static void insert(CqlSession session, String table) {
        List<List<String>> data = new ArrayList<>();

        String[] fileNames = {"airport.txt", "flights.txt", "seats.txt"}; // Your
            ↪ txt file names

        for (String fileName : fileNames) {
            try (BufferedReader br = new BufferedReader(new FileReader(fileName)))
                ↪ {
                String line;
                List<String> list = new ArrayList<>();

                while ((line = br.readLine()) != null) {
                    list.add(line.trim());
                }

                data.add(list);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (table.equals("airport")) {
            System.out.println("Insert Airport");
            for (String row : data.get(0)) {
                try {
                    SimpleStatement query = SimpleStatement.newInstance(
                            "INSERT INTO airport (Size, Country, Country_code,
                                ↪ Geo_Point, IATA_code, ICAO_code, Name, Name_en,
                                ↪ Name_fr, Operator, Phone, Website) VALUES " +
                                ↪ row + ";");
                    query =
                        ↪ query.setConsistencyLevel(DefaultConsistencyLevel.LOCAL_QUORUM);
                    session.execute(query);
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        } else if (table.equals("flight")) {
            System.out.println("Insert Flight");
            for (String row : data.get(1)) {
                try {
                    SimpleStatement query = SimpleStatement.newInstance(
                            "INSERT INTO flight (ID, Departure, Destination,
                                ↪ Number_of_Seats, Day, Hour, Operator, Duration,
                                ↪ Price_per_Person) VALUES " + row + ";");
```

```java
                query =
                    ↪ query.setConsistencyLevel(DefaultConsistencyLevel.LOCAL_QUORUM);
                session.execute(query);
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    } else {
        System.out.println("Insert Seat");
        for (String row : data.get(2)) {
            try {
                SimpleStatement query = SimpleStatement.newInstance(
                        "INSERT INTO seat (Flight, ID, Status, Name, Surname,
                            ↪ Document_Info, Date_of_Birth, Balance) VALUES "
                            ↪ + row + ";");
                query =
                    ↪ query.setConsistencyLevel(DefaultConsistencyLevel.LOCAL_QUORUM);
                session.execute(query);
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

public static void main(String[] args) {

    DriverConfigLoader loader =
        ↪ DriverConfigLoader.fromString("datastax-java-driver.basic.request.timeout
        ↪ = 10 seconds");

    try (CqlSession session = CqlSession.builder()
            .addContactPoint(new InetSocketAddress("localhost", 9042))
            .addContactPoint(new InetSocketAddress("localhost", 9043))
            .withLocalDatacenter("datacenter1")
            .withConfigLoader(loader)
            .build()) {

        System.out.println("Connected to Cassandra!");

        session.execute("CREATE KEYSPACE IF NOT EXISTS my_airport WITH
            ↪ replication = "
                + "{'class':'SimpleStrategy','replication_factor':1};");
        session.execute("USE my_airport;");

        ResultSet rs = session.execute("SELECT table_name FROM
            ↪ system_schema.tables " +
                "WHERE keyspace_name = 'my_airport' ");

        List<String> tablesBefore = new ArrayList<>();
        for (Row row : rs) {
            tablesBefore.add(row.getString("table_name"));
        }

        session.execute("CREATE TABLE IF NOT EXISTS airport (Name_en text,
            ↪ Country text, Website text, Geo_Point text, ICAO_code text,
            ↪ Country_code text, Phone text, Name_fr text, Name text,
            ↪ IATA_code text, Size int, Operator text, PRIMARY KEY
            ↪ (IATA_code));");
        session.execute("CREATE TABLE IF NOT EXISTS flight (ID text,
            ↪ Number_of_Seats int, Day date, Hour time, Operator text,
            ↪ Duration text, Price_per_Person int, Destination text,
            ↪ Departure text, PRIMARY KEY ((Departure), Destination, Day,
```

```
                   ↪ Hour, id));");
             session.execute("CREATE TABLE IF NOT EXISTS seat (Flight text, Status
                   ↪ text, ID text, Name text, Surname text, Document_Info text,
                   ↪ Date_of_Birth date, Balance int, PRIMARY KEY ((Flight), ID));");

             rs = session.execute("SELECT table_name FROM system_schema.tables " +
                   "WHERE keyspace_name = 'my_airport' ");

             List<String> tablesAfter = new ArrayList<>();
             for (Row row : rs) {
                 tablesAfter.add(row.getString("table_name"));
             }

             if (tablesBefore.size() != tablesAfter.size()) {
                 for (String table : tablesAfter) {
                     if (!tablesBefore.contains(table)) {
                         insert(session, table);
                     }
                 }
             }

         }
     }
}
```

Next, suppose you suspend the `cassandra-node2` node. Execute the commands:

```
docker exec -it cassandra-node1 bash
nodetool describecluster
```

to get the current status of the nodes:

```
root@a619f2004d70:/# nodetool describecluster
Cluster Information:
        Name: Test Cluster
        Snitch: org.apache.cassandra.locator.SimpleSnitch
        DynamicEndPointSnitch: enabled
        Partitioner: org.apache.cassandra.dht.Murmur3Partitioner
        Schema versions:
                ee264747-9443-3a0b-b39b-39e6e74d67f8: [172.22.0.2, 172.22.0.4]

                UNREACHABLE: [172.22.0.3]

Stats for all nodes:
        Live: 2
        Joining: 0
        Moving: 0
        Leaving: 0
        Unreachable: 1

Data Centers:
        datacenter1 #Nodes: 3 #Down: 1

Database versions:
        4.1.5: [172.22.0.2:7000, 172.22.0.3:7000, 172.22.0.4:7000]

Keyspaces:
        system_traces -> Replication class: SimpleStrategy {replication_factor=2}
        system_distributed -> Replication class: SimpleStrategy
            ↪ {replication_factor=3}
        system_auth -> Replication class: SimpleStrategy {replication_factor=1}
        system_schema -> Replication class: LocalStrategy {}
        system -> Replication class: LocalStrategy {}
```

Then suppose, always keeping the node suspended, to start a seat reservation via command line from `cassandra-node1` (without considering concurrency or data management problems):

```
UPDATE seat
    SET
        status = 'Occupied',
        balance = 100,
        date_of_birth = '2000-01-01',
        document_info = 'AAAAA',
        name = 'Mario',
        surname = 'Rossi'
    WHERE
        flight = '6677f96c7acf35542d8eebc9'
    AND     id = '10A';
```

Finally, restart the `cassandra-node2` node. After a couple of minutes the following statistics and the log file were analyzed, obtaining in particular:

Pool Name	Active	Pending	Completed	Blocked	All time blocked
RequestResponseStage	0	0	200	0	0
MutationStage	0	0	172726	0	0
ReadStage	0	0	58	0	0
CompactionExecutor	0	0	1102	0	0
MemtableReclaimMemory	0	0	41	0	0
PendingRangeCalculator	0	0	6	0	0
GossipStage	0	0	3965	0	0
SecondaryIndexManagement	0	0	1	0	0
HintsDispatcher	0	0	1	0	0
MigrationStage	0	0	22	0	0
MemtablePostFlush	0	0	98	0	0
PerDiskMemtableFlushWriter_0	0	0	38	0	0
ValidationExecutor	0	0	0	0	0
Sampler	0	0	0	0	0
ViewBuildExecutor	0	0	0	0	0
MemtableFlushWriter	0	0	41	0	0
TracingStage	0	0	1002	0	0
CacheCleanupExecutor	0	0	0	0	0
Native-Transport-Requests	0	0	283	0	0

```
(base) PS C:\Users\Filippo> docker logs cassandra-node1
OpenJDK 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in
    ↪ version 9.0 and will likely be removed in a future release.
...
INFO  [GossipStage:1] 2024-07-07 14:09:56,174 Gossiper.java:1419 - Node
    ↪ /172.22.0.3:7000 has restarted, now UP
INFO  [GossipStage:1] 2024-07-07 14:09:56,203 StorageService.java:3075 - Node
    ↪ /172.22.0.3:7000 state jump to NORMAL
INFO  [GossipStage:1] 2024-07-07 14:09:56,413 TokenMetadata.java:539 - Updating
    ↪ topology for /172.22.0.3:7000
INFO  [GossipStage:1] 2024-07-07 14:09:56,419 TokenMetadata.java:539 - Updating
    ↪ topology for /172.22.0.3:7000
INFO  [GossipStage:1] 2024-07-07 14:09:56,426 Gossiper.java:1366 - InetAddress
    ↪ /172.22.0.3:7000 is now UP
INFO  [Messaging-EventLoop-3-5] 2024-07-07 14:10:03,529
    ↪ InboundConnectionInitiator.java:529 -
    ↪ /172.22.0.3:7000(/172.22.0.3:53846)->/172.22.0.2:7000-SMALL_MESSAGES-816dfb2b
    ↪ messaging connection established, version = 12, framing = CRC, encryption =
    ↪ unencrypted
INFO  [Messaging-EventLoop-3-6] 2024-07-07 14:10:03,540
    ↪ InboundConnectionInitiator.java:529 -
    ↪ /172.22.0.3:7000(/172.22.0.3:53856)->/172.22.0.2:7000-LARGE_MESSAGES-9533f979
    ↪ messaging connection established, version = 12, framing = CRC, encryption =
    ↪ unencrypted
```

```
INFO  [Messaging-EventLoop-3-2] 2024-07-07 14:10:03,558
   ↪ OutboundConnection.java:1153 -
   ↪ /172.22.0.2:7000(/172.22.0.2:37782)->/172.22.0.3:7000-SMALL_MESSAGES-d46d55a5
   ↪ successfully connected, version = 12, framing = CRC, encryption =
   ↪ unencrypted
INFO  [Messaging-EventLoop-3-3] 2024-07-07 14:10:03,577
   ↪ OutboundConnection.java:1153 -
   ↪ /172.22.0.2:7000(/172.22.0.2:37796)->/172.22.0.3:7000-LARGE_MESSAGES-2edb7d80
   ↪ successfully connected, version = 12, framing = CRC, encryption =
   ↪ unencrypted
INFO  [HintsDispatcher:1] 2024-07-07 14:10:12,849 HintsStore.java:213 - Deleted
   ↪ hint file 04a24d9c-ae90-4ae5-ac76-a8ab431d9ad7-1720361192009-2.hints
INFO  [HintsDispatcher:1] 2024-07-07 14:10:12,852 HintsDispatchExecutor.java:301 -
   ↪ Finished hinted handoff of file
   ↪ 04a24d9c-ae90-4ae5-ac76-a8ab431d9ad7-1720361192009-2.hints to endpoint
   ↪ /172.22.0.3:7000: 04a24d9c-ae90-4ae5-ac76-a8ab431d9ad7
```

where the information

```
HintsDispatcher 0 0 1 0 0
```

and the last lines of the log file highlight the use of the gossip protocol, to identify the moment when the node came back online, and hinted handoff to transmit the data to it.

Conclusions

MongoDB and Apache Cassandra are two of the most popular NoSQL databases, each with unique characteristics and applications that make them suitable for different needs. Despite their similarities in supporting unstructured data and scalability, the differences in their architectures and functionalities distinguish them clearly. MongoDB utilizes a document-based data model, where documents are stored in BSON format, similar to JSON. These documents are collected in collections and do not require a fixed schema, making it particularly flexible for handling dynamic and unstructured data. In contrast, Cassandra uses a wide-column data model, which is a hybrid between a tabular storage and a key-value store. Each row can have a different set of columns, organized into column families, with Sorted String Tables (SSTables) serving as the basic storage unit containing key-value pairs for a specific column family.

In terms of query language, MongoDB employs the MongoDB Query Language (MQL), which supports CRUD operations with commands similar to those in Node.js. Cassandra, on the other hand, uses the Cassandra Query Language (CQL), which is similar to SQL but adapted to its data model. This difference makes CQL more intuitive for those familiar with SQL, while MQL offers greater flexibility for document-specific operations. MongoDB offers various types of indexes, including single field, compound, multikey, geospatial, and text search indexes. Cassandra supports secondary indexes and SSTable-Attached Secondary Indexes (SASI) for complex queries, although these can impact system performance.

Concurrency control and transaction management represent another area of difference between MongoDB and Cassandra. MongoDB manages concurrency through Multi-Version Concurrency Control (MVCC) and document-level locking, allowing for ACID transactions at the single-document level. Starting from version 4.0, MongoDB also offers multi-document transactions, albeit with some performance limitations. Cassandra handles concurrency with tunable consistency and row-level atomicity. It does not natively support full ACID transactions but allows for batches of operations with row-level isolation, providing eventual consistency with configurable consistency levels. From a scalability and availability perspective, both systems are designed to support horizontal scalability through data partitioning and replication across multiple nodes. MongoDB employs a primary node with replicas for read and write operations. Its sharding architecture allows data distribution across

multiple nodes with load balancing and elastic scalability. This approach offers greater control over partitioning through sharding keys, optimizing data distribution and queries.

Cassandra is designed for high availability with a peer-to-peer topology and multiple master nodes. Each node in the cluster is equal, eliminating a single point of failure. This architecture allows for linear scalability by simply adding new nodes to the cluster without the need to reconfigure the entire infrastructure. MongoDB uses a primary-secondary replication model, where a primary node handles all write operations and secondary replicas are used for reading and failover. If the primary node fails, one of the secondaries is elected as the new primary, ensuring operational continuity. Cassandra employs a masterless replication model, where any node can handle read and write operations. Data is automatically replicated across multiple nodes to ensure fault tolerance. This model ensures that even in the event of one or more node failures, the system continues to operate without interruptions.

Both databases are open source in their initial versions, with MongoDB developed by MongoDB, Inc. and Cassandra initially developed by Facebook and later released by Apache. They support a wide range of programming languages, with MongoDB offering support for 12 languages, while Cassandra supports a slightly smaller but still diverse number. Security is a critical aspect for both databases. In their base versions, the security of MongoDB and Cassandra is primarily managed by the user, with SSL and TLS for client connections and user authentication. MongoDB Atlas, the paid version of MongoDB, offers extended security features such as X.509, client-side and server-side encryption, and integration with Kerberos and LDAP. In summary, MongoDB is ideal for handling unstructured and dynamic data due to its flexibility and scalability, whereas Cassandra is excellent for applications requiring high availability and high performance for read and write operations. Both databases provide powerful solutions for various needs, making them indispensable tools in the modern data management landscape.