Air Travel Reservation Modeling Using MongoDB and Cassandra

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

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.

MongoDB and the Document Model

After introducing Atlas, it is now time to dive into the Document Model of MongoDB. In essence, we aim to comprehend how MongoDB stores data, how it is utilized, and its behavior in relation to Atlas. 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.

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.

The Document Model

As anticipated, MongoDB stores data in structures known as documents which resemble JSON objects. 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 furniture 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"
}
```

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.

In MongoDB Atlas, clusters can be easily set up and managed through the 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 Atlas, sharding can be enabled and configured through the Atlas 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, deploying a replica set is straightforward, and the 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. By leveraging clusters, sharding, and replica sets, MongoDB Atlas offers a robust and scalable infrastructure that can handle the demands of modern applications, ensuring data availability, reliability, and performance. In summary, MongoDB's architecture is designed to be flexible, scalable and reliable, allowing to manage a wide range of applications and workloads from the simplest to the most complex.

Architettura Cassandra

PLACEHOLDER PER FILIPPO

Data Modeling

MongoDB Data Modeling

Data modeling 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 modeling 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": "Qualità 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 modeling decisions with application requirements, contrasting with the traditional approach of modeling 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 modeling 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 modeling is to optimize storage, querying, and resource utilization, enhancing application performance and reducing database costs as a consequence. Once the foundational data modeling framework is established, attention can be directed towards modeling 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 modeling 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 modeling 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 modeling 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")
        ]
}
```

Modeling Data Relationships

In this section, we provide an example of data modeling 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 modeling 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": "Qualità 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 modeling 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": "Qualità 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 subdocuments 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": "Qualità 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 Modeling

PLACEHOLDER PER FILIPPO

Database Connection

Connecting to a MongoDB Database

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, 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.

Using Maven with Spring Boot

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 s pecific 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
<?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
    <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
   <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>
```

```xml
    </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 MongoDB Atlas 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:

```java
/**
 * The Connection class provides functionality to connect to a MongoDB instance and list all available
 */
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:
```

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);
   }

   /**
    * Returns the singleton instance of the Connection class, creating it if necessary.
    *
```

```java
     * @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

PLACEHOLDER PER FILIPPO

Syntax

MongoDB Syntax

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()).toIns
        .append("result", "No Violations")
        .append("sector", "Theme Parks")
        .append("address", new Document().append("city", "Castelnuovo del Garda").append("zip", 37014)
```

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()).toIns
        .append("result", "No Violations")
        .append("sector", "Theme Parks")
        .append("address", new Document().append("city", "Castelnuovo del Garda").append("zip", 37014)

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", "MDB9911
Document doc2 = new Document().append("account_holder", "Claudia Gatto").append("account_id", "MDB791C

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
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.

```
MongoDatabase database = mongoClient.getDatabase("bank");
MongoCollection<Document> collection = database.getCollection("accounts");
try(MongoCursor<Document> cursor = collection.find(and(gte("balance", 1000), eq("account_type", "check
    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.

```
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"
System.out.println(doc.toJson());
```

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

```
// 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:

```java
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:

```java
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:

```java
// 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) {
```

```java
        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());
    }
}
```

Cassandra Syntax

PLACEHOLDER PER FILIPPO

Transactions and Aggregations

MongoDB Transactions and Aggregations

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 transaction 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
2. using the `WithTransaction()` method on the session object.
3. **Transaction Operations**: Define the operations to be performed within the transaction. This typically includes
4. fetching necessary data, performing updates, and inserting documents.
5. **Transaction Commit**: After executing all operations successfully, commit the transaction to persist the changes.
6. **Handling Timeouts and Resource Closure**: MongoDB automatically cancels any multi-document transaction that
7. exceeds 60 seconds. Additionally, ensure proper closure of resources utilized by the transaction.

Example Code

```
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("accour

        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() +
        System.out.println("Depositing to Account " + toAccountFilter.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.

```
// 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 accountIdOfRe
    try (ClientSession session = client.startSession()) {
        UUID transfer = UUID.randomUUID();
        String transferId = transfer.toString();
        try {
            session.withTransaction(() -> {
                MongoCollection<Document> accountsCollection = client.getDatabase("bank").getCollectic
                MongoCollection<Document> transfersCollection = client.getDatabase("bank").getCollecti


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

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

                transfersCollection.insertOne(session, new Document("_id", new ObjectId()).append("tra
                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.

Example 1: Using $match

```
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
}
```

Example 2: Using $match and $group

```
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
    System.out.println("Display aggregation results");
    accounts.aggregate(Arrays.asList(matchStage, groupStage)).forEach(document -> System.out.print(doc
}
```

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", "chec
    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("$divide", Arrays.asList("$balar
                    Projections.excludeId()
            )
    );
    System.out.println("Display aggregation results");
    accounts.aggregate(Arrays.asList(matchStage, sortStage, projectStage)).forEach(document -> System.
}
```

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 Transactions and Aggregations

PLACEHOLDER PER FILIPPO

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 a pivotal role in enhancing query performance in MongoDB, judicious management—including periodic review, deletion of redundant indexes, and careful consideration of operational implications—is essential to maintain optimal database performance and efficiency.

Cassandra Optimization

PLACEHOLDER PER FILIPPO

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:

```
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 coc
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", "Sourc

# 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 retreival 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 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:

```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.
```

```java
 * 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;

   /**
    * 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.
 *
 * 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/?retr

        try {
            // Obtain MongoClient and MongoDB/MongoCollection
            MongoClient mongoClient = Connection.getInstance(connectionString).getMongoClient();
            MongoDatabase database = mongoClient.getDatabase(dbName);
```

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

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

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: " + fie
                    }
                } 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();
```

```java
            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> collec
        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 c
        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 current airport)
            Document destinationAirport = getRandomDestinationAirport(airports, currentAirport);

            // Generate random future date between 1 and 10 days from today
```

```java
        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
 * @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
 */
private static Document getRandomDestinationAirport(List<Document> airports, Document currentAirpc
    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.");
```

```java
                }
                int index = random.nextInt(airports.size());
                destinationAirport = airports.get(index);
                if (!destinationAirport.getObjectId("_id").equals(currentAirport.getObjectId("_id"))) {
                    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.

```json
[{
   "_id": {
      "$oid": "6677f94b7acf35542d8ee8b2"
   },
   "Geo_Point": "51.1075552; 16.8756697",
   "Name": "Port Lotniczy Wrocław",
   "Name_(en)": "Wroclaw Airport",
   "Name_(fr)": "Aéroport de Wrocław-Nicolas Copernic",
   "IATA_code": "WRO",
   "ICAO_code": "EPWR",
   "Operator": "Port Lotniczy Wrocław 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.

Modello dati Cassandra

PLACEHOLDER PER FILIPPO

Transazioni

Spiegazione su cosa è una transazione e perchè è importante per il nostro caso di studio (semplicemnte non posso avere due persone sullo stesso posto)

Transazioni in Mongo

spieghi come hai modellato le transazioni e come lato backend vengono gestite (quindi come le gestisce il db)

Transazioni in Cassandra

PLACEHOLDER PER FILIPPO

Gestione su larghi volumi

Devo ancora capire bene cazzo dobbiamo fare qua ma poi anch'esso sarà diviso in Mongo e Cassandra

Conclusioni

Tiriamo un po' le somme: quali sono le principalissime differenze e quando è meglio usare uno al posto di un altro

Riassumiamo tipo questi 3 siti: https://aws.amazon.com/it/compare/the-difference-between-cassandra-and-mongodb/#:~:text=Riepilogo%20delle%20differenze%3A%20Cassandra%20e%20MongoDB,-Apache%20Cassandra&text=Documenti%20JSON%20serializzati.&text=Cassandra%20supporta%20indici%20secondari%20e,offre%20diverse%20opzioni%20di%20indicizzazione.

https://www.mongodb.com/resources/compare/cassandra-vs-mongodb https://www.ionos.it/digitalguide/server/know-how/mongodb-e-cassandra/

questa parte è giusto per rispondere alla domanda "e quindi quando usate uno al posto dell'altro?"