1.How to find unused indexes in MongoDB?

Proactively managing MongoDB database indexes across multiple application releases can be challenging for development teams. Because requirements often change with each release, new indexes may be added while older ones become deprecated. Over time, this can make it difficult to track which MongoDB indexes are in use and which are no longer needed.

Indexes have a significant impact on write performance - every time a collection is written, the relevant indexes need to be updated. Insufficient indexes are immediately noticeable and slow down queries - unused indexes are more subtle and need to be proactively deleted to improve write performance. For more information, see Risks of Building Indexes on MongoDB.

Starting from MongoDB version 3.2, the official provides a powerful tool - $indexStats aggregation operator, which allows us to easily obtain index usage statistics, so as to accurately locate those indexes that have not been used for a long time.

Use a script to find all unused indexes

We can use a script to iterate over all collections in the database and collect the usage of each index using $indexStats . You can run the following script in a mongosh environment, which will automatically list all indexes with a usage count of 0:

```
db = db.getSiblingDB("admin")

var dbInfos = db.adminCommand({listDatabases:1, nameOnly: true})

dbNames = []
for(d=0;d<dbInfos.databases.length;d++) {
    dbName = dbInfos.databases[d];
    if (dbName.name == "local" || dbName.name == "config" || dbName.name ==
"admin") {
        continue;
    }
    dbNames.push(dbName.name)
}

collectionInfos = []
indexesInfo = []
indexesInfoNoUse = []

for(d=0;d<dbNames.length;d++){
    // collectionNames = db.getSiblingDB(dbNames[d]).getCollectionNames();
    collectionNames=[];
    db.getSiblingDB(dbNames[d]).getCollectionInfos({ type: "collection"
}).forEach(info => {
      collectionNames.push(info.name);
```

```
    });
    for(c=0;c<collectionNames.length;c++) {
        if (collectionNames[c].startsWith('system.')) {
          continue;
        };

idx=(db.getSiblingDB(dbNames[d]).getCollection(collectionNames[c]).aggregate([{$ind
exStats: {}}, {$project: {indexName:'$name', indexKeys: '$spec.key', indexUsage:
'$accesses.ops' }}]).toArray().map(
          m => (
            {'Namespace':dbNames[d]+'.'+collectionNames[c]
,'IndexName':m.indexName, //'indexKeys':m.indexKeys, in a
              'Usage':m.indexUsage}
          )));

          for(i=0;i<idx.length;i++) {
            if (idx[i].Usage > 0 ){
              indexesInfo.push(idx[i])
            } else {
              indexesInfoNoUse.push(idx[i])
            }
          }
    }
}
//Uncomment this to show also usage for all indexes
//print("\nIndexes with usage")
//console.log(indexesInfo)

print("\nIndexes with no usage")
console.log(indexesInfoNoUse)
```

The returned document will look like this:

```
[
    {
        "Namespace" : "xxx.xxx",
        "IndexName" : "full_mobile_1",
        "Usage" : NumberLong(0)
    },
    {
        "Namespace" : "xxx.xxx",
        "IndexName" : "country_1_mobile_no_1",
        "Usage" : NumberLong(0)
    },
]
```

Delete useless indexes

Once you have identified and verified an unused MongoDB index, you can delete it using the following code:

```
db.users.dropIndex("xxxx")
```

Note: Deletion is irreversible. Before executing the dropIndex() command, please double-check that the collection and index names you are operating on are correct!

2. Risks of Building Indexes on MongoDB

Case Study: E-commerce Platform Downtime Incident

In order to optimize its order query performance, a medium-sized e-commerce platform decided to run a production environment on a database containing tens of millions of records.ordersAdd a new index to the collection. The developer executed the defaultdb.orders.createIndex(...) Order.

Because this triggers a foreground index build, MongoDB will exclusively lock the entire collection at the beginning and end of the index build, making the collection unresponsive to any read or write operations during these two phases.

Therefore, all operations such as writing new orders, querying user orders, and updating backend inventory are blocked at the beginning and end of index building, causing service unavailability for a period of time. Although indexing itself can take several hours to complete, the actual blocking time is concentrated at the beginning and end of the index building phase.

This downtime had a significant impact on business services, impaired user experience, and caused serious business losses.

Risks and Choices in Index Building Modes

- **Foreground Build**:
    - **risk:** This mode applies an exclusive access lock (write lock) to the entire collection during the critical phases of index building (start and end), blocking all read and write operations. For large, active collections, this may cause a brief service outage or severe performance degradation.
    - Default behavior: In MongoDB versions prior to 4.2, the frontend build will lock the collection throughout the entire process, which has a greater impact; versions 4.2 and later have been optimized, but the collection will still be locked at the beginning and end.
    - **Applicable scenarios:**This is only applicable to collections with very small data volumes and low availability requirements.
- **Background Build**:
    - **risk:**Although it does not block reads and writes, it will consume a lot of system resources during the build process, causing a significant drop in database

performance. What's more dangerous is that if the build process is interrupted (such as a server restart), it will automatically revert to**Foreground mode**, again causing downtime risk.

- **Code example:**

```
db.orders.createIndex({ "status": 1 }, { "background": true })
```

- **Rolling Index Build**:
  - **Best Practices:**This is the safest solution for replica sets.**Node by node** Index building is performed to avoid affecting the performance of the entire cluster.
  - **Workflow:**1. Remove a secondary node from the replica set. 2. Execute on the node **Front-end index building**, since it is now independent, it will not affect the cluster. 3. Add the node that completed the index build back to the replica set. 4. Repeat this process for the remaining slave nodes. 5. Finally, demote the primary node to a slave node and perform the same operation on it.
  - **advantage:**Achieved **Zero downtime**The index construction has minimal impact on the production environment.

In a production environment,**Always prioritize rolling builds** If you must use background builds without replica sets, be sure to do them during off-peak hours and monitor system performance closely.

3. Use MongoDB Atlas Triggers to synchronize data

Business Challenge: The Dilemma of Traditional Data Synchronization

In modern application architectures, developers often use embedded data models to improve read performance. This model redundantly stores related data in a single document, avoiding costly cross-collection joins. For example, in an e-commerce application, an Orders collection might embed some customer or product information.

However, this model also introduces core challenges:**Data consistency** When source data (such as customer information) is updated, all embedded documents that reference this data must be updated synchronously. Manual synchronization not only increases development and maintenance costs, but also easily leads to **Data inconsistency**, affecting the accuracy of business logic.

Solution: Event-driven automated synchronization mechanism

In order to enjoy the high performance advantages brought by the embedded model while solving the problem of data synchronization, we recommend using **MongoDB Atlas Triggers** This solution leverages MongoDB's Change Streams technology, which monitors database

operation events (such as inserts, updates, and deletes) in real time and automatically triggers predefined functions to execute synchronization logic.

The core workflow is as follows:

1. Event monitoring: Configure one or more Atlas Triggers to accurately monitor document changes in the source collection (for example, the Customers collection).
2. **Function execution**：Once the trigger detects a specified event, it automatically calls a **Serverless Function**(Serverless functions).
3. Data synchronization: This function automatically searches for and updates the corresponding embedded data in the target collection (for example, the Orders collection) based on the changed content.

In this way, data synchronization no longer relies on manual or complex logic at the application layer, but instead becomes a **Fully automatic, event-driven** Background tasks.

Sample code: A general data synchronization template

The following code snippet shows a generic Atlas Trigger function that can serve as a template for handling any data synchronization scenario. It listens for changes to the source collection and replicates those changes to the target collection.

```
exports = async function(changeEvent) {
  // A Database Trigger will always call a function with a changeEvent.
  // Documentation on ChangeEvents:
https://docs.mongodb.com/manual/reference/change-events/

  // This sample function will listen for events and replicate them to a collection
in a different Database

  // Access the _id of the changed document:
  const docId = changeEvent.documentKey._id;

  // Get the MongoDB service you want to use (see "Linked Data Sources" tab)
  // Note: In Atlas Triggers, the service name is defaulted to the cluster name.
  const serviceName = "mongodb-atlas";
  const database = "other_db";
  const collection =
context.services.get(serviceName).db(database).collection(changeEvent.ns.coll);

  // Get the "FullDocument" present in the Insert/Replace/Update ChangeEvents
  try {
    // If this is a "delete" event, delete the document in the other collection
    if (changeEvent.operationType === "delete") {
      await collection.deleteOne({"_id": docId});
    }

    // If this is an "insert" event, insert the document into the other collection
    else if (changeEvent.operationType === "insert") {
      await collection.insertOne(changeEvent.fullDocument);
```

```
    }

    // If this is an "update" or "replace" event, then replace the document in the
other collection
    else if (changeEvent.operationType === "update" || changeEvent.operationType
=== "replace") {
      await collection.replaceOne({"_id": docId}, changeEvent.fullDocument);
    }
  } catch(err) {
    console.log("error performing mongodb write: ", err.message);
  }
};
```

The template can be modified according to specific business needs. For example, more complex logic can be added to the function to handle the reference relationship between different collections to ensure the eventual consistency of the data.

4.MongoDB Change Streams: How to achieve uninterrupted real-time data synchronization?

Business Challenge: How to handle unexpected interruptions in data flow?

Modern applications increasingly rely on **Real-time data streaming** To drive business logic, such as real-time analysis, data synchronization between microservices, and cache invalidation.**Change Streams** It is the core technology to achieve this goal, which allows applications to monitor database changes in real time.

However, in actual production environments, unexpected situations such as network failures, server restarts, or application deployments may occur at any time, which may cause the Change Stream connection to be interrupted. If the application cannot be restored from the interruption point, all data changes that occurred during the interruption will be lost, resulting in **Data inconsistency**, seriously affecting the reliability of the business.

Solution: Seamless flow resumption with Resume Token

To solve this pain point, MongoDB Change Streams introduces **Resume Token** mechanism.**Resume Token** Essentially a unique identifier for an event, it marks the position of a specific event in the stream.

When the Change Stream connection is unexpectedly interrupted, the application can use this **Resume Token**, tells MongoDB to continue sending data from the last successfully processed event. This ensures that the application can continue to process data even in the face of transient failures.**Restore data flow seamlessly without any data loss**, thus ensuring the integrity and consistency of data processing.

Sample code: A general recovery model

The following code snippet shows how to exploit resumeAfter option to implement a Change Stream listener with automatic recovery capabilities. This is a powerful pattern that can serve as a template for any application that requires high-reliability data streaming.

```
// A simple example of resuming a change stream after a specific event.

// Connect to your MongoDB collection
const collection = db.collection('inventory');

// Open the initial change stream
const changeStream = collection.watch();

let newChangeStream;

// Listen for the first change event to get a resume token
changeStream.once('change', next => {
  // Store the resume token from the current change stream
  const resumeToken = changeStream.resumeToken;

  // Close the current stream (simulating a disconnection)
  changeStream.close();

  // Reopen a new change stream, using the resumeAfter option
  // This will resume the stream from the last processed event
  newChangeStream = collection.watch([], { resumeAfter: resumeToken });

  // Now, the new stream will process all subsequent changes
  newChangeStream.on('change', next => {
    processChange(next);
  });
});
```

With this pattern, developers do not need to write complex retry and state management logic. The MongoDB driver automatically handles most of the recovery details, greatly simplifying the code and improving the robustness of the application

5.MongoDB Atlas Search: How to UsesearchSequenceTokenEasily build high-performance paging

Business Challenge: Paging performance degrades dramatically with depth

Pagination is a common requirement when building applications that need to process large amounts of data efficiently. However, many developers are accustomed to using traditional

`$skip` and `$limit` This approach works well when the data set is small, but as the amount of data grows and users browse deeper into the result set, its performance bottleneck becomes increasingly apparent.

The main problem is that each time you turn the page, the database must **Scan from the beginning and skip** For all previous documents, this results in:

- **Query time** Integral to page depth **Linear growth**。
- **Resource consumption** It then surges, especially as users visit later pages.
- When the number of concurrent users is large, the overall system performance drops sharply, seriously affecting the user experience.

Solution: Use Atlas Search `searchSequenceToken`

To solve `$skip` To solve the performance bottleneck of MongoDB Atlas Search, a more efficient **cursor-based** paging method is provided. The core is to use `searchSequenceToken` or `searchAfter`。

The core idea of this method is that **Does not depend on page numbers or offsets** Instead, the "position" of the last document on the previous page is passed as a **token** to the next query. This is like giving the database a "signpost" to start retrieving directly from the specified position instead of scanning from the beginning.

Core implementation:

1. First time enquiry: On the initial request,`$search` Add after stage `$project` stage, we can `searchSequenceToken` Export to each document.

```
{
  "pipeline": [
    {
      "$search": {
        "index": "search_jobs_v1",
        "autocomplete": { "query": "dk250", "path": "job_number" }
      }
    },
    { "$limit": 20 },
    {
      "$project": {
        "job_number": 1,
        "metadata": 1,
        "paginationToken": { "$meta": "searchSequenceToken" }
      }
    }
  ]
}
```

2. Follow-up Inquiries: When getting the next page, the last document in the previous page response is `paginationToken` Pass to `$search` stage `searchAfter` Parameters.

```
{
  "pipeline": [
    {
      "$search": {
        "index": "search_jobs_v1",
        "autocomplete": { "query": "dk250", "path": "job_number" },
        "searchAfter": "previousPageToken"
      }
    },
    { "$limit": 20 }
  ]
}
```

Advantages Summary

This method completely eliminates `$skip` The performance issues brought about by this are:

- **Constant query performance**: Whether the user visits page 1 or page 1000, the response time remains consistent.
- **Resource efficient**: The database does not need to scan and discard large amounts of documents, significantly reducing CPU and memory consumption.
- **Robustness**: Handle paging more gracefully even when data is updating in real time.

By moving paging from inefficient `$skip` Upgrade to high performance `searchSequenceToken`, your applications can easily cope with the growing amount of data while providing users with a smooth and stable experience.

6. How to build a high-performance paging system? MongoDB paging practice

Business Problem: Performance Bottlenecks of Traditional Paging Methods

When dealing with massive amounts of data, paging is an indispensable feature. Developers usually use `$skip` and `$limit` However, this traditional approach has serious performance issues:

- **Performance decays with depth**：`$skip` The database needs to scan and skip all previous documents, causing the query time to increase linearly with the increase of page number. When the user visits the later pages, the response speed will slow down dramatically.

- **High resource consumption**: A large number of skip operations consume unnecessary CPU and memory resources, especially in high-concurrency scenarios, which will affect the performance of the entire system.
- **Poor data consistency**: In a collection where data changes frequently, `$skip` This can result in duplicate or missed documents because the data may have changed between queries.

Solution: Cursor-Based Pagination

To solve the above problems, the best practice is to adopt **Cursor-based paging** This method does not rely on page numbers and offsets, but uses a value (i.e., cursor) in the previous query result as the starting point for the next query.

The core idea is: after the first query, record a unique and ordered field of the last document (for example `_id` or timestamp `createdAt`). The next time you query, use this value as the filter condition to query only the documents that follow it.

Core Implementation

The following are two common cursor-based paging implementations.

1. Use `_id` or ordered fields

This is the simplest and most efficient method. Suppose you have an auto-incrementing `_id` Or a timestamp field, which you can use as a cursor.

First query (get the first page)

```
db.posts.find({}).sort({ _id: 1 }).limit(10)
```

Subsequent queries (get the next page)

```
// `lastId` is the _id of the last document from the previous query
db.posts.find({ _id: { $gt: lastId } }).sort({ _id: 1 }).limit(10)
```

2. For complex sorting scenarios

If the query needs to be sorted by multiple fields (for example, `score` Sort by `_id` sort), you need to pass an array containing the field values as a cursor.

First query (get the first page)

```
db.leaderboard.find({}).sort({ score: -1, _id: 1 }).limit(10)
```

Subsequent queries (get the next page)

```
// `lastScore` and `lastId` are from the last document of the previous query
db.leaderboard.find({
  $or: [
{ score: { $lt: lastScore } },
{ score: lastScore, _id: { $gt: lastId } }
  ]
}).sort({ score: -1, _id: 1 }).limit(10)
```

Solution Value

Cursor-based paging offers the following significant advantages over traditional approaches:

- **Stable performance**: Query performance is no longer affected by page depth and always remains efficient.
- **Resource efficient**: No time-consuming skipping operations are required, reducing database load.
- **Data integrity**: Even when the data changes in real time, the correctness of the paging results can be ensured to avoid data duplication or omission.

This approach is the key to building scalable, high-performance applications **Best Practices**, which can easily cope with the paging needs of massive data.

7. How does MongoDB data model design empower HR intelligent chatbots?

Business Problem: Management and Interaction of Complex HR Data

In the human resources field, an intelligent chatbot needs to process a variety of data types, including employee profiles, company organizational structures, recruitment information, and extensive user chat history. Traditional monolithic data design or relational database models are unable to cope with the following challenges:

- **Many-to-many relationships**: For example, an employee can have multiple skills and participate in multiple projects; a position also requires multiple skills.
- **High concurrent write**: Chat history is typical "write-intensive" data that requires efficient storage and fast retrieval.
- **Complex query requirements**: For example, matching internal positions based on user skills, or filtering job postings based on department or position status.

Improper data model design can lead to poor query performance, data redundancy, and high maintenance costs.

Solution: MongoDB Multi-Collection Flexible Modeling

To address these issues, we leveraged MongoDB's flexible document model and designed a general solution consisting of multiple collections to optimize the performance and scalability of our HR chatbot:

1. **Embedding and referencing**:
    - **Embedding Model**:exist `User` In the collection, the employees**skills、Data that is frequently accessed together, such as projects, is embedded as sub-documents, reducing the time required to read.`join` Operation, achieving single read, multiple access**。
    - **Reference Model**: For cross-collection or many-to-many relationships, such as `User` and `Projects`, using reference (_id) for association. This approach maintains data flexibility and consistency.
2. **Specialized collection design**:
    - **`Message` gather**: In view of the high frequency of writing chat records, we use**Time Series Collection**This special collection type is designed specifically for time-series data and provides excellent write performance and storage efficiency.
    - **`Chat` gather**: Store the session ID, user ID and last update time as metadata of the chat session, and use**TTL (Time-to-Live) index**Automatically delete expired sessions to manage data lifecycle.
3. **Index optimization**:
    - For key fields (such as `email`、`company_id`) create **Unique index** and **Single-field index**, to ensure fast retrieval and uniqueness of data.
    - For complex query scenarios (such as finding positions by department and required skills), create **Composite index**, to support efficient filtering of multiple fields.

Core code example:

The following is the architectural design of some key collections, which reflects the efficient combination of embedded, reference, and time series models in practical applications.

User Collection (Embedding and Reference)

```
{
  "_id": "ObjectId('655a6d5774a810168b941570')",
  "first_name": "Jane",
  "last_name": "Doe",
  "email": "jane.doe@example.com",
  "company_id": "ObjectId('655a6d5774a810168b941571')",
  "skills": [ // Embedded sub-documents for fast reads
    {
      "skill_id": "ObjectId('655a6d5774a810168b941572')",
```

```
      "proficiency": 5,
      "certificates": ["MongoDB Certified Developer"]
    }
  ],
  "projects": [ // Reference to a different collection
    {
      "project_id": "ObjectId('655a6d5774a810168b941573')",
      "role": "Team Lead"
    }
  ]
}
```

Message Collection (Time Series)

```
{
  "from": "ObjectId('655a6d5774a810168b941570')",
  "to": "ObjectId('655a6d5774a810168b941574')",
  "content": "What are the requirements for a senior data scientist role?",
  "timestamp": ISODate("2025-08-19T10:00:00.000Z")
}
```

Query Examples

The following is an example of how this model design can be used to efficiently process complex queries.

Query 1: Query user details based on skills and project experience

```
db.users.find({
  "skills.skill_id": "ObjectId('655a6d5774a810168b941572')"
})
```

Expected Result: Since the skill information has been **Embed** exist `users` In the collection, the query does not need to be `join` operation to directly return the user document containing skill and project references.

Query 2: Find all participants in a specified project

```
db.users.find({
  "projects.project_id": "ObjectId('655a6d5774a810168b941573')"
})
```

Expected Result: Quickly locate all user documents involved in the project through the index.

Query 3: Get the chat history for the last hour

```
db.messages.find({
```

```
  "timestamp": {
    "$gte": ISODate("2025-08-19T09:00:00.000Z"),
    "$lt": ISODate("2025-08-19T10:00:00.000Z")
  }
})
```

Expected Result: With **Time series collection** With the optimization, such time range queries can achieve extremely high performance.

8.How to optimize query performance based on data cardinality?

Business Problem: Indexes are good, but incorrect design can backfire

Indexes are a powerful tool for improving database query speed, but not all indexes can bring performance improvements. **Low base** (e.g., there are only two values for "male" and "female" `gender` fields), its filtering efficiency will be very low and may even be slower than a full table scan.

Even worse, an inappropriate composite index (**Compound Index**) The order of fields will directly affect query performance. MongoDB composite index follows the "leftmost prefix" principle. If the order of the index fields is not in accordance with **Base high to low** If the data is not arranged correctly, the index will not be fully utilized, resulting in poor query performance.

Solution: Utilize field cardinality and automatically recommend indexes

To solve this problem, we propose a **Data cardinality** An intelligent index recommendation solution. This solution combines MongoDB **Performance Advisor** The recommendations and scripts are automatically analyzed to create composite indexes that can best improve query efficiency.

The core idea is:**Place the fields with the highest cardinality at the beginning of the composite index.**High cardinality fields (such as `user_id`、`email`) can narrow the search scope more quickly, thus significantly improving query speed.

This solution is automated through the following steps:

1. **Get existing indexes and query suggestions**: Get index optimization recommendations provided by Performance Advisor through the MongoDB Atlas API.
2. **Calculate field cardinality**: Calculate the cardinality of the fields to be indexed. Cardinality refers to the number of unique values in a field.
3. **Smart Sorting**：All index fields to be created are **Base from high to low** Sort to generate the optimal composite index field order.

4. **Automatically generate index commands**: Based on the sorting results, generate a `createIndex()` Order.

Core code logic:

The following is the key logic of the automation script, which shows how to get the field **Cardinality** And generate index creation commands based on this.

```
// Connect to MongoDB
const { MongoClient } = require("mongodb");
const uri = "mongodb://localhost:27017";
const client = new MongoClient(uri);

// Function to get cardinality of a field
async function getCardinality(databaseName, collectionName, indexKey) {
  const database = client.db(databaseName);
  const collection = database.collection(collectionName);
  // Get the first field of the index key
  const field = Object.keys(indexKey)[0];
  // Use distinct() to count unique values for a field
  const cardinality = await collection.distinct(field).then(values =>
values.length);
  return cardinality;
}

// Function to generate index creation commands
async function generateIndexCommands(collectionName, indexInfo) {
  // Sort index fields by cardinality in descending order
  indexInfo.sort((a, b) => b.cardinality - a.cardinality);

  // Generate index creation commands
  const commands = indexInfo.map(index => {
    return `db.${collectionName}.createIndex(${JSON.stringify(index.key)}, { name:
"${index.name}" })`;
  });

  return commands;
}

// Main function to orchestrate the process
async function main() {
  try {
    await client.connect();
    console.log("Connected to MongoDB");

    const databaseName = "yourDatabase";
    const collectionName = "yourCollection";

    // Assume these are index recommendations from Performance Advisor or other
sources
    const recommendedIndexes = [
      { key: { email: 1 }, name: "email_1" },
```

```
      { key: { department: 1 }, name: "department_1" },
      { key: { userId: 1, email: 1 }, name: "userId_1_email_1" },
    ];

    const indexInfoWithCardinality = [];
    for (const index of recommendedIndexes) {
      const cardinality = await getCardinality(databaseName, collectionName,
index.key);
      indexInfoWithCardinality.push({
        key: index.key,
        name: index.name,
        cardinality: cardinality
      });
    }

    const commands = await generateIndexCommands(collectionName,
indexInfoWithCardinality);

    console.log("\nGenerated Index Commands:");
    commands.forEach(cmd => console.log(cmd));

  } finally {
    await client.close();
    console.log("\nConnection closed");
  }
}

main().catch(console.error);
```

Expected Result:

Assume that your data`userId`The base number is the highest.`email`Secondly,`department`minimum, then the script will sort the index according to cardinality and generate the corresponding commands.

```
Connected to MongoDB

Generated Index Commands:
db.yourCollection.createIndex({"userId":1,"email":1}, { name: "userId_1_email_1" })
db.yourCollection.createIndex({"email":1}, { name: "email_1" })
db.yourCollection.createIndex({"department":1}, { name: "department_1" })

Connection closed
```

Solution Value

This approach automatically generates the most optimized index for query optimization, eliminating the need to manually analyze complex query patterns and data distribution. This not only improves query performance but also significantly reduces the complexity of database management, ensuring long-term, efficient system operation.

9. MongoDB Atlas: How to save costs and optimize performance by automatically scaling IOPS?

Business Problem: Unable to Balance Performance and Cost

Many business systems face a common challenge:**Tidal nature of workloads** For example, office applications require high throughput during peak hours on weekdays, but the load drops sharply at night or on weekends; e-commerce platforms experience a surge in traffic during promotional periods, but are relatively stable during normal times.

In this scenario, if the database cluster **IOPS (Input/Output Operations Per Second)** While maintaining peak performance guarantees peak performance, it wastes significant resources during off-peak hours, leading to high costs. Conversely, keeping IOPS at a lower level to save money can lead to performance bottlenecks during peak periods, impacting user experience.

also,MongoDB Atlas's autoscaling feature currently supportsCPU utilization, memory utilization, and storage capacity usageAutomatic scaling of storage capacity does not include automatic scaling of IOPS.This means that for customers who need to periodically adjust IOPS based on business load, the built-in features of Atlas cannot meet their fine-grained management needs.

Solution: Automated IOPS Management Based on Business Cycles

To solve this problem, we can use **MongoDB Atlas API** Build an automated script to dynamically adjust the cluster's IOPS based on preset business cycles (such as weekdays, weekends, and peak hours).

The core idea of this solution is to automatically increase IOPS to a high level to meet performance requirements during business peak periods; and automatically reduce IOPS during non-peak periods, thereby achieving **The best balance between performance and cost**。

Core implementation:

This automation logic can be implemented using a simple Atlas Serverless Function. Its core steps include:

1. **Get the current configuration**: Get the current configuration information of the cluster through the Atlas API to ensure that other parameters are not accidentally reset during updates.
2. **Defining business logic**: Write logic in your code to calculate the target IOPS value based on the current date and time (such as the day of the week and the hour).
3. **Dynamically update IOPS**: If the current IOPS does not match the target value,`PATCH` Request to send update command to Atlas API to automatically adjust the IOPS of the cluster

Core code logic

The following is the key code snippet of this solution, showing how to automatically calculate the target IOPS based on time and call the API to update it.

```
exports = async function() {
  const axios = require('axios');
  const moment = require('moment-timezone');

  const groupId = context.values.get("GROUP_ID");
  const clusterName = context.values.get("CLUSTER_NAME");
  const apiKey = context.values.get("API_KEY");
  const apiSecret = context.values.get("API_SECRET");

  const baseUrl =
`https://cloud.mongodb.com/api/atlas/v1.0/groups/${groupId}/clusters/${clusterName}
`;

  // Step 1: Retrieve the current cluster configuration
  const currentConfig = await axios.get(baseUrl, {
    auth: { username: apiKey, password: apiSecret }
  }).then(response => response.data);

  // Step 2: Determine the target IOPS based on the current time and business logic
  const now = moment().tz("GMT");
  const day = now.day();
  const hour = now.hour();

  let targetIops;
  if (day >= 2 && day <= 6) { // Tuesday to Saturday
    if ((hour >= 7 && hour < 17) || (hour >= 18 && hour < 22)) {
      targetIops = 3000;
    } else {
      targetIops = 2000;
    }
  } else { // Sunday and Monday
    targetIops = 2000;
  }

  // Step 3: Check if IOPS needs to be updated and build the new config payload
  if (currentConfig.providerSettings.diskIOPS !== targetIops) {
const newConfig = {
      providerSettings: {
        providerName: currentConfig.providerSettings.providerName,
        instanceSizeName: currentConfig.providerSettings.instanceSizeName,
        diskIOPS: targetIops,
        regionName: currentConfig.providerSettings.regionName,
        autoScaling: {
          diskGBEnabled: true
        }
      },
      replicationFactor: currentConfig.replicationFactor,
      clusterType: currentConfig.clusterType
```

```
    };

    // Step 4: Update the cluster configuration using a PATCH request
    await axios.patch(baseUrl, newConfig, {
      auth: { username: apiKey, password: apiSecret }
    });
    console.log(`Cluster IOPS updated to ${targetIops}`);
  } else {
    console.log(`IOPS is already at the target value of ${targetIops}. No update
needed.`);
  }
};
```

Through this automation, your database cluster can act like an intelligent "steward" and adapt to the fluctuations of business load without sacrificing performance.**Automatic cost optimization**, so that every penny is spent wisely.

10. How to ensure data quality? Best practices for MongoDB schema validation

Business Problem: Data Inconsistency in Flexible Mode

MongoDB is known for its**Flexible document model**It is known for allowing developers to quickly iterate and adapt to changing data structures. However, this flexibility also brings a major challenge:**Data inconsistency**In large-scale team collaboration or long-term projects, due to the lack of mandatory structural constraints, documents in the collection may have problems such as missing fields, incorrect data types, or mismatched formats.

For example, a user collection might have `email` The field was accidentally stored as a number, or `address` Occasionally, fields are strings instead of embedded documents. These inconsistent data not only affect query accuracy but are also likely to cause unexpected errors in the application at runtime.

Solution: Use MongoDB Schema Validation

To ensure data quality while maintaining flexibility, **MongoDB Schema Validation** Schema Validation is the best solution. It allows developers to define **JSON Schema** Rules enforce data structure and type constraints at the database level.

It works by setting validation rules on a collection. When a new insert or update operation violates these rules, MongoDB blocks the operation based on the configuration, thus preventing the writing of unqualified data at the source.

Core implementation:

In addition to validation on writes, MongoDB schema validation provides a powerful feature:**Find existing documents that do not match the pattern** This is especially important when cleaning up historical data or migrating legacy systems.

The following code snippet shows how to use `$jsonSchema` Operator and `$nor`(Non) logic to efficiently find all documents in a collection that do not match a pre-set pattern.

```
// Define JSON Schema
const schema = {
  bsonType: "object",
  required: ["name", "address"],
  properties: {
    name: {
      bsonType: "string",
      description: "Must be a string"
    },
    address: {
      bsonType: "object",
      required: ["city", "zipcode"],
      properties: {
        city: { bsonType: "string" },
        zipcode: { bsonType: "string" }
      }
    }
  }
};

// Use $jsonSchema with $nor to find invalid documents
const invalidDocs = await collection.find({
$nor: [{ $jsonSchema: schema }]
}).toArray();

console.log("Documents that do not conform to the schema:", invalidDocs);
```

Solution Value

By adopting MongoDB schema authentication, you can:

- **Improve data quality**: Enforce data constraints at the database level to avoid the generation of "dirty data".
- **Simplify application logic**: Move the responsibility for data validation from the application layer to the database layer, reducing duplicate code and potential errors.
- **Ensure system robustness**: Ensure that all written data meets expectations and reduce runtime errors caused by data format issues.

This approach provides developers with a powerful tool that allows them to enjoy the flexibility of MongoDB while also having the data consistency guarantees of a relational database.

11.MongoDB database health check: How to accurately diagnose your data model?

Business Problem: Database Visibility Challenges

In application development, the database internal **Data distribution** and **Resource consumption**Visibility is often lacking. Developers often focus solely on implementing functionality and neglect the internal state of the database. As the business grows, this lack of transparency can lead to serious performance and cost issues:

- **Unexpected storage space growth**: Unreasonable document design or abuse of large fields may cause storage space to expand rapidly.
- **Inefficient indexing**: Invalid indexes take up valuable storage resources but cannot effectively improve query performance.
- **Low resource utilization**: It is difficult to perform targeted optimization because it is impossible to identify which collections consume the most resources.

Lack of visibility into database internal statistics makes it difficult to make sound performance optimization and cost control decisions.

Solution: Use database statistics scripts for proactive diagnosis

To address the lack of database visibility, you can write a script to **Automatically obtain detailed statistics for databases and collections** This script can traverse all collections and provide a series of key indicators to help developers and database administrators (DBAs) perform accurate diagnosis and optimization.

Core indicators include:

- `dataSize`: The uncompressed size of all documents in the collection.
- `storageSize`: The actual compressed storage space occupied by the collection on disk.
- `totalIndexSize`: The total storage space occupied by all indexes.
- `count`: Number of documents.
- `averageDocumentSize`: Average document size.

By analyzing these metrics, you can quickly identify the problem: Is it because of excessive data volume, too many indexes, or is it because individual documents are too large, leading to inefficient storage?

Core code logic

The following is the key logic of the diagnostic script, which shows how to iterate over each collection in the database and collect statistics.

```
var collections = db.getCollectionNames();

var totalDataSize = 0;
var totalStorageSize = 0;
var totalIndexSize = 0;
var totalDocumentCount = 0;

collections.forEach(function(collectionName) {
    var stats = db.getCollection(collectionName).stats();
totalDataSize += stats.size;
    totalStorageSize += stats.storageSize;
    totalIndexSize += stats.totalIndexSize;
    totalDocumentCount += stats.count;
});

print("Total Data Size: " + (totalDataSize / (1024 * 1024)).toFixed(2) + " MB");
print("Total Storage Size: " + (totalStorageSize / (1024 * 1024)).toFixed(2) + "
MB");
print("Total Index Size: " + (totalIndexSize / (1024 * 1024)).toFixed(2) + " MB");
print("Average Document Size: " + (totalDataSize / totalDocumentCount).toFixed(2) +
" bytes");

print("Data/Storage Size Ratio: " + ((totalDataSize / totalStorageSize) *
100).toFixed(2) + "%");
print("Index/Storage Size Ratio: " + ((totalIndexSize / totalStorageSize) *
100).toFixed(2) + "%");
```

The returned document will look like this:

```
Total Data Size: xxx MB
Total Storage Size: xxx MB
Total Index Size: xxx MB
Average Document Size: xxx bytes
Data/Storage Size Ratio: xxx%
Index/Storage Size Ratio: xxx%
```

Solution Value

By running this script regularly, you can get a clear "physical report" of your database, which is crucial for the following scenarios:

- **Data model optimization**: Determine whether the document design is reasonable based on the average document size.
- **Index maintenance**: Identify indexes that take up too much space and evaluate their necessity.
- **Capacity Planning**: Forecast future storage and performance requirements based on data growth trends.

This enables you to move from reacting to performance issues to **Proactively manage database health**, thereby ensuring the long-term stability and efficient operation of the system.