Best Practices Guide for MongoDB

More than 20 tips for getting started

MongoDB.

Introduction

The purpose of this guide is to help you get started working with data in MongoDB. A common mistake is not planning how you intend to store data in MongoDB. But the way you store data in MongoDB will affect the way you retrieve data for a range of use cases.

We want you to get the most out of your MongoDB deployment. Whether you're new to document databases or you've worked with MongoDB before, you'll find the tips and sample code here useful for leveraging key features in our application data platform. Once you understand how to utilize the tools available, you can start building highly performant applications while reducing overhead and costs.

We begin with data modeling, because the way you store data in a document database is different from creating tables, rows, and schemas in a relational database. One of the key differences between relational and document databases is that with relational databases, data is stored based on the data itself whereas with document databases, data is stored based on data access patterns. We then move on to working with data, including query patterns, profiling, and optimizing data access patterns.

The last section is on availability and scalability, which are critical for highly performant applications and achieving service level objectives.

Table of Contents

Data Modeling

Tailor documents according to how data will be used and accessed.

Data Modeling

Data that is accessed together should be stored together. This can dramatically improve performance when compared to a relational database because joins are often not necessary. When you can, use the flexibility of MongoDB's document data model to organize data within a document.

In this example, the user alerts are embedded in the users object rather than in a separate table as it would be in a traditional relational database.

```
// db.users
{
  _id: "abc",
  email: "xyz@example.com",
  preferences: {
    alerts:[
      { name: "morning", frequency: "daily", time: { h: 6, m: 0 } }
    ],
    colors: { bg: "#cccccc" }
  }
}
```

Subdocuments – documents within a document – and embedded data models allow you to cleanly separate sections of related fields within a document, while making it possible to update related data in a single atomic write operation.

Learn more about data modeling.

Data Modeling

Avoid creating large, unbounded arrays that might exceed the 16 MB maximum size for a single document. For example, the number of customer reviews a store might receive is unknown. Instead of storing this data in an array, each review should be a separate document with a reference to the store document.

```
// businesses
{
  _id: "100",
  name: "Bake and Go",
  addresses: [
    { street: "40 Elm", state: "NY" },
    { street: "101 Main St", state: "VT" }
  ]
}

 // businessReviews
{
  _id: "61e706e2450fac1271c9b27a",
  storeId: "100",
  rating: 5,
  comment: "Best bagels in town, a must try if you are in the area."
}
{
  _id: "61e706e2450fac1271c9c522",
  storeId: "100",
  rating: 5,
  comment: "Yummy desserts and a nice place to sit and eat."
}
```

Data Modeling

Not all 1:1 and 1:many relationships should be represented in a single document. Referencing between documents should be used when:

- A document is frequently read but contains data that is rarely accessed.
- One part of the document is frequently updated or growing in size, while the rest of the document is relatively static.
- When the total document size could exceed MongoDB's 16 MB document limit.

Take the following example of a manufacturer with an array of models. Since the models are often displayed individually, rather than as an array, querying the data would require transformation before getting displayed. In this case, it makes sense to keep the models in their own collection, referencing the manufacturer in the document.

Car models are often used outside of the parent document. Moving them to a different collection is useful if we are regularly reading or writing to the models collection.

```
// manufacturers
{
  _id: "200",
  name: "Swaab Automotive",
  type: "auto",
  models: [
    { name: "Swaab Model X", year: 2022, sku: "SWA-X-22Z"},
    { name: "Swaab Model Y", year: 2022, sku: "SWA-Y-22Z"},
    // ...
  ]
}

// models
{
  _id: "1201",
  name: "Swaab Model X",
  year: 2022,
  sku: "SWA-X-22Z",
  manufacturer_id: "200"
}
{
  _id: "1202",
  name: "Swaab Model Y",
  year: 2022,
  sku: "SWA-Y-22Z",
  manufacturer_id: "200"
}
```

Data Modeling

Duplication is a powerful technique that allows you to avoid unnecessary joins by storing the same pieces of data in multiple documents, at the cost of some added complexity, to keep them consistent and up to date.

MongoDB offers change streams and triggers to help keep duplicated data in sync across documents.

```
// users
{
  _id: "1",
  name: { first: "Jane", last: "Doe" }
  email: "jdoe@example.com"
}

// posts
{
  _id: "1055",
  title: "My First Post",
  text: "Hello World",
  userId: "1",
  userEmail: "jdoe@example.com"
}
```

Schema Visualization

MongoDB Compass is the free GUI for MongoDB. One of its most useful features is schema visualization, which enables you to explore your schema with histograms that show your documents' fields, data types, and values.

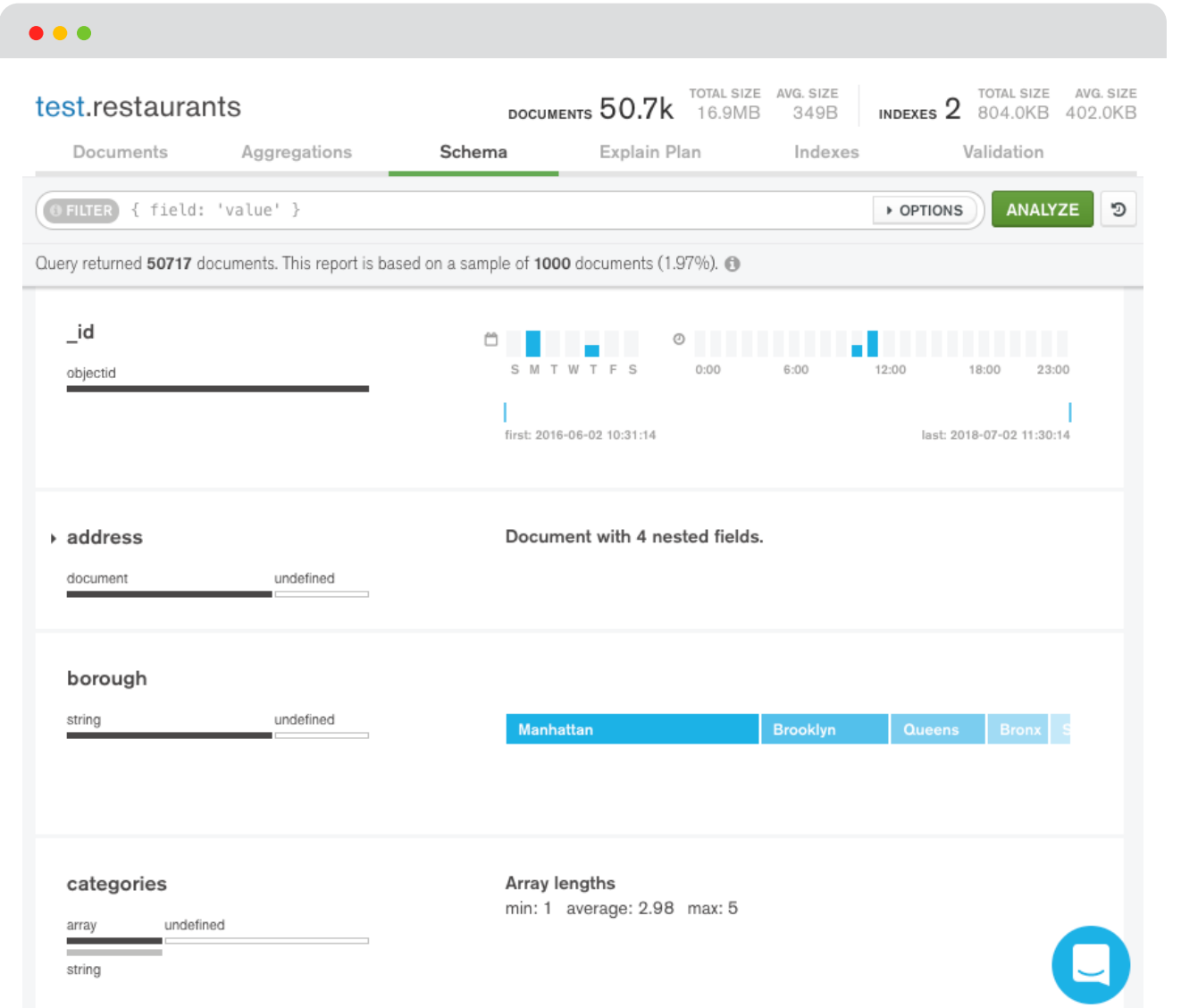


Figure 1: Compass is an interactive tool for querying, optimizing, and analyzing your MongoDB data.

Choosing Instance Sizes

As with most databases, MongoDB performs best when the application's working set (indexes and most frequently accessed data) fits in memory.

When the application's working set fits in RAM, read activity from disk will be low. If your working set exceeds the RAM of your chosen instance size or server, consider moving to a larger instance with more memory, or partition (shard) your database across multiple servers.

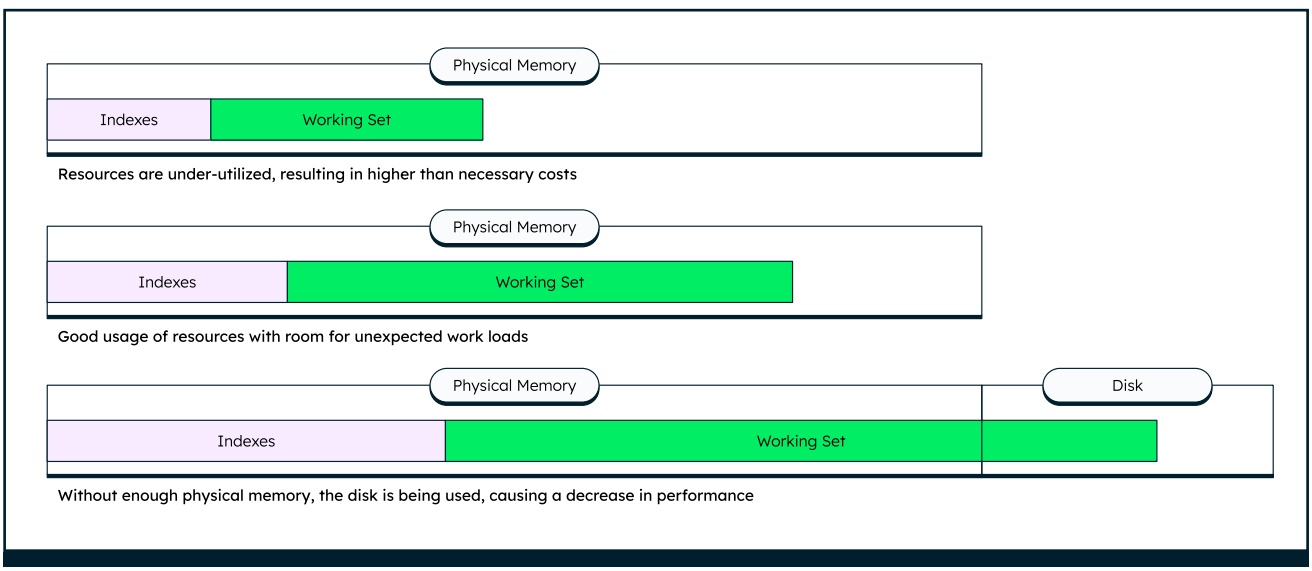


Figure 2: MongoDB works best when indexes and most frequently accessed data fit into memory (RAM).

If price/performance is more of a priority over performance alone, then using fast SSDs is a viable design choice. You should test the optimum balance for your workload and service level agreements (SLAs).

Working With Data

Leverage the flexibility of the document model with a few practical tips.

Query Patterns

Reduce network utilization and database overhead by issuing updates only on fields that have changed rather than retrieving the entire document in your application, updating fields, and then saving the document back to the database.

With fully expressive array updates, you can perform complex array manipulations against matching elements of an array – including elements embedded in nested arrays – all in a single update operation.

```
// books
{
  _id: 1,
  title: "Building CI/CD System Using Tekton",
  publisher: "Packt",
  tags: ["software", "devops"],
  author: "Joel Lord",
  ratings: [
    { by: "reader@example.com", rating: 5 },
    { by: "other_reader@example.com", rating: 4},
    { by: "reviewer@amazon.com", rating: 9}
 ]
}

// Update ratings from user at @example.com domain to 10
db.books.update(
  { _id: 1 },
  {
    $set: {
      "ratings.$[rating].rating": 10
    }
  },
  {
    arrayFilters: [ { "rating.by": { $regex: "(.*)\@example.com" } } ]
  }
);
```

Learn more about the arrayFilters option.

Query Patterns

MongoDB's Aggregation Framework allows you to send an analytics or data processing workload – written using the aggregation pipeline language – to the database to execute on data in place.

An aggregation pipeline is an ordered series of declarative statements called stages, where the output of one stage forms the input of the next stage, and so on.

Rather than manipulating data in your application, you can leverage the power of aggregation pipelines to return exactly what is needed. This reduces the amount of code teams need to write and maintain while limiting the amount of data that has to be pushed back and manipulated in your applications.

```
db.seats.aggregate([
 {
   $match: { "class": "first" }
 },
 {
   $group: { _id: "$flightId", firstClassSeats: { $sum: 1 } }
 }
]);
```

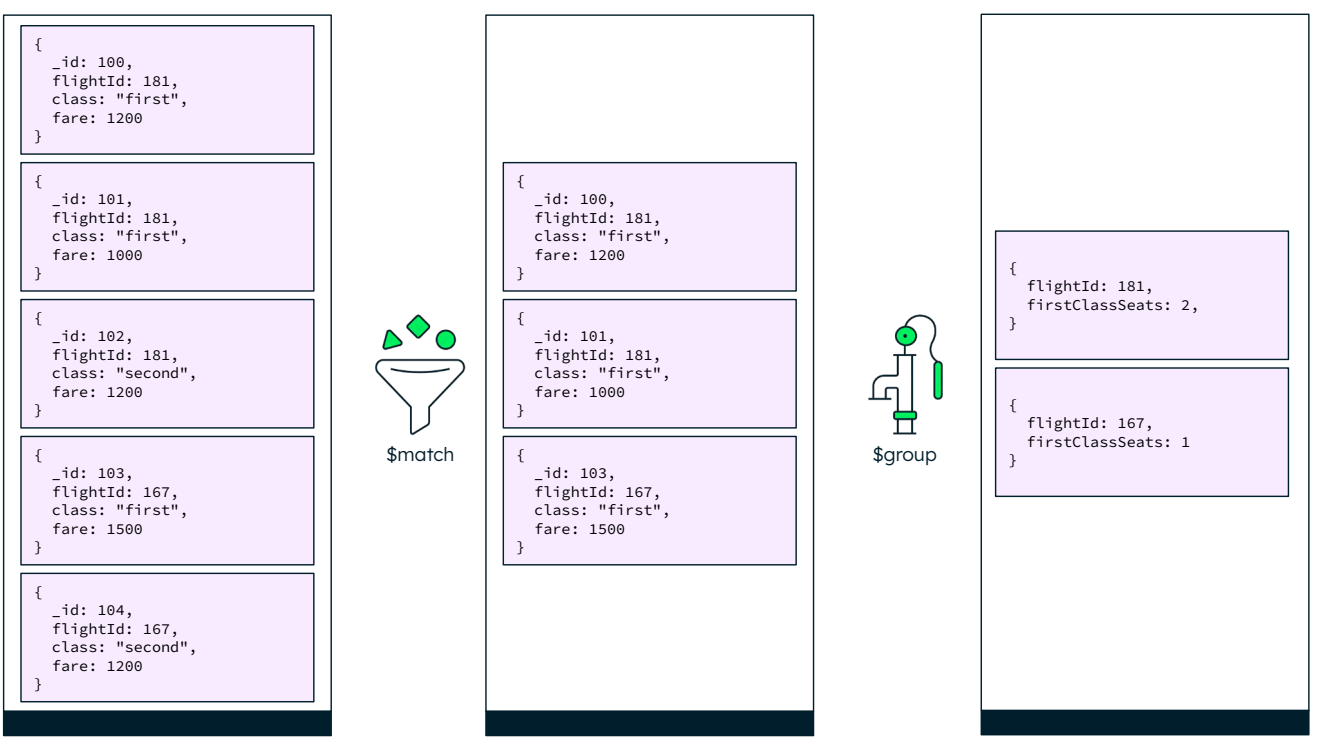


Figure 3: The aggregation pipeline is a set of operations that processes, transforms, and returns results.

Query Patterns

Aggregation pipeline stages exhibit high composability where stages are stateless, self-contained components. This allows you to take a complex problem and break it down into straightforward, individual stages, where each step can be developed and tested in isolation.

Use the following guidelines for crafting pipelines that improve your pace of development:

- Do not start or end a stage on the same line as another stage.
- For every field in a stage and stage in the pipeline, include a trailing comma.
- Include an empty new line between stages.
- For complex stages, add a // comment with an explanation on a new line before the stage.
- Comment out complete stages using /* */

```
// Bad
const pipeline = [
 {"$unset": [
   "_id",
   "address"
 ]}, {"$match": {
   "dateofbirth": {"$gte": ISODate("1970-01-01T00:00:00Z")}
 }}//, {"$sort": {
 //  "dateofbirth": -1
 //}}, {"$limit": 2}
];

// Good

const pipeline = [
  {
    "$unset": [
      "_id",
      "address",
    ]
  },

 // Only match people born on or after January 1st, 1970
  {
    "$match": {
      "dateofbirth": {"$gte": ISODate("1970-01-01T00:00:00Z")},
    }
  },

  /*
  {
    "$sort": {
      "dateofbirth": -1,
    }
  },

  {
    "$limit": 2
  },
  */
];
```

For more information on aggregations, visit docs or check out the Practical MongoDB Aggregations E-Book.

Query Patterns

Available in MongoDB Compass and within the Atlas UI, the Aggregation Pipeline Builder is a user interface for constructing aggregations with an easy-to-use, drag-and-drop experience.

You can preview results, toggle stages on and off for testing, learn about different operators, and take advantage of auto-complete for aggregation operators, query operators, and document field names.

The Aggregation Pipeline Builder makes it easy to leverage the dozens of stages and hundreds of operators that make up the MongoDB Aggregation Framework.

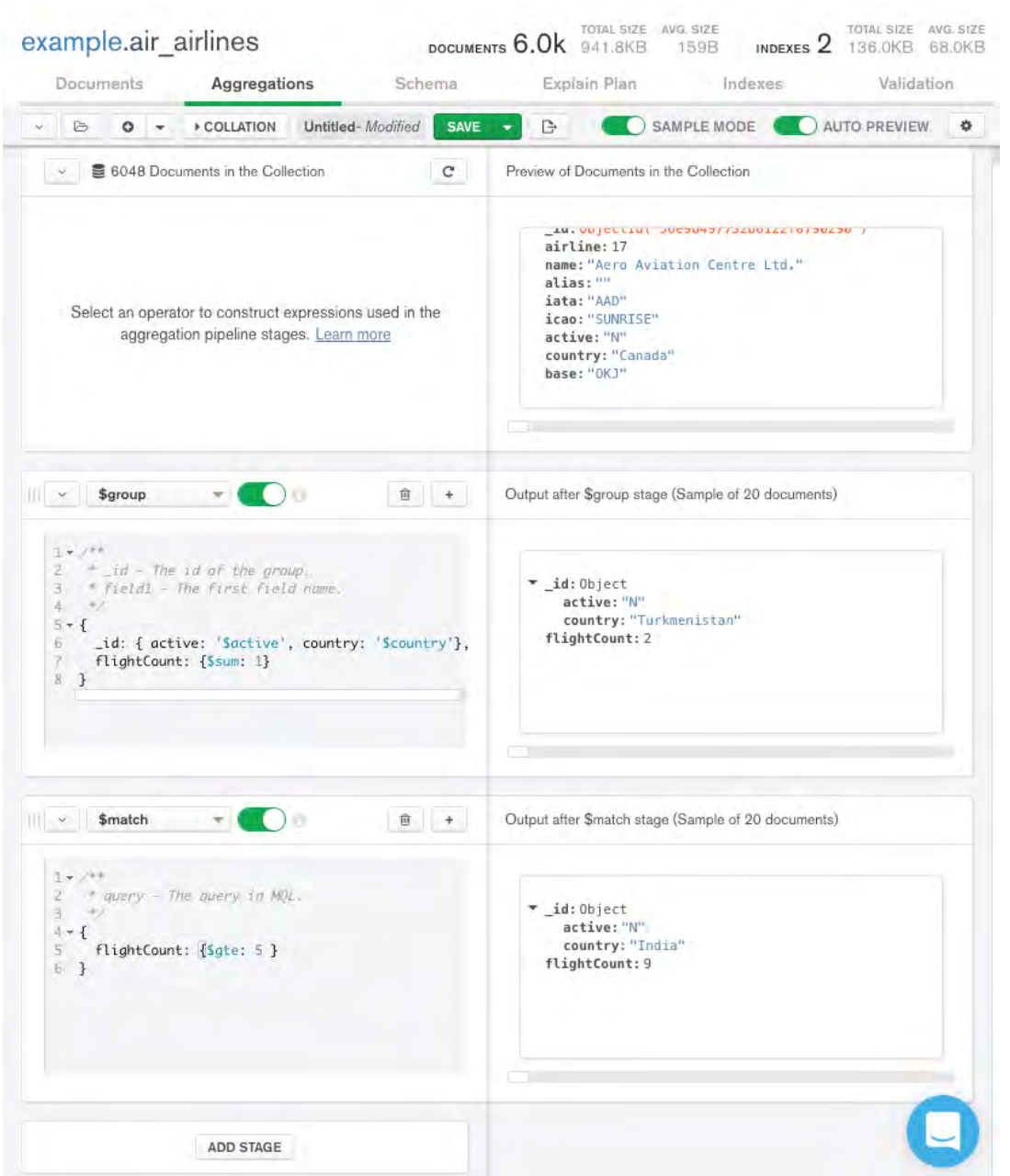


Figure 4: Use the embedded, intuitive builder to construct powerful aggregation pipelines with a few clicks.

Query Profiling

Use the explain() command in mongosh or the “Explain” tab in Compass to test and optimize query performance by examining details such as:

- Which indexes were used
- Whether an in-memory sort was performed
- The number of index entries scanned
- The number of documents returned and the number read
- How long the query took to resolve in milliseconds

The explain plan will show 0 milliseconds if the query was resolved in less than 1 ms, which is typical for well-tuned queries.

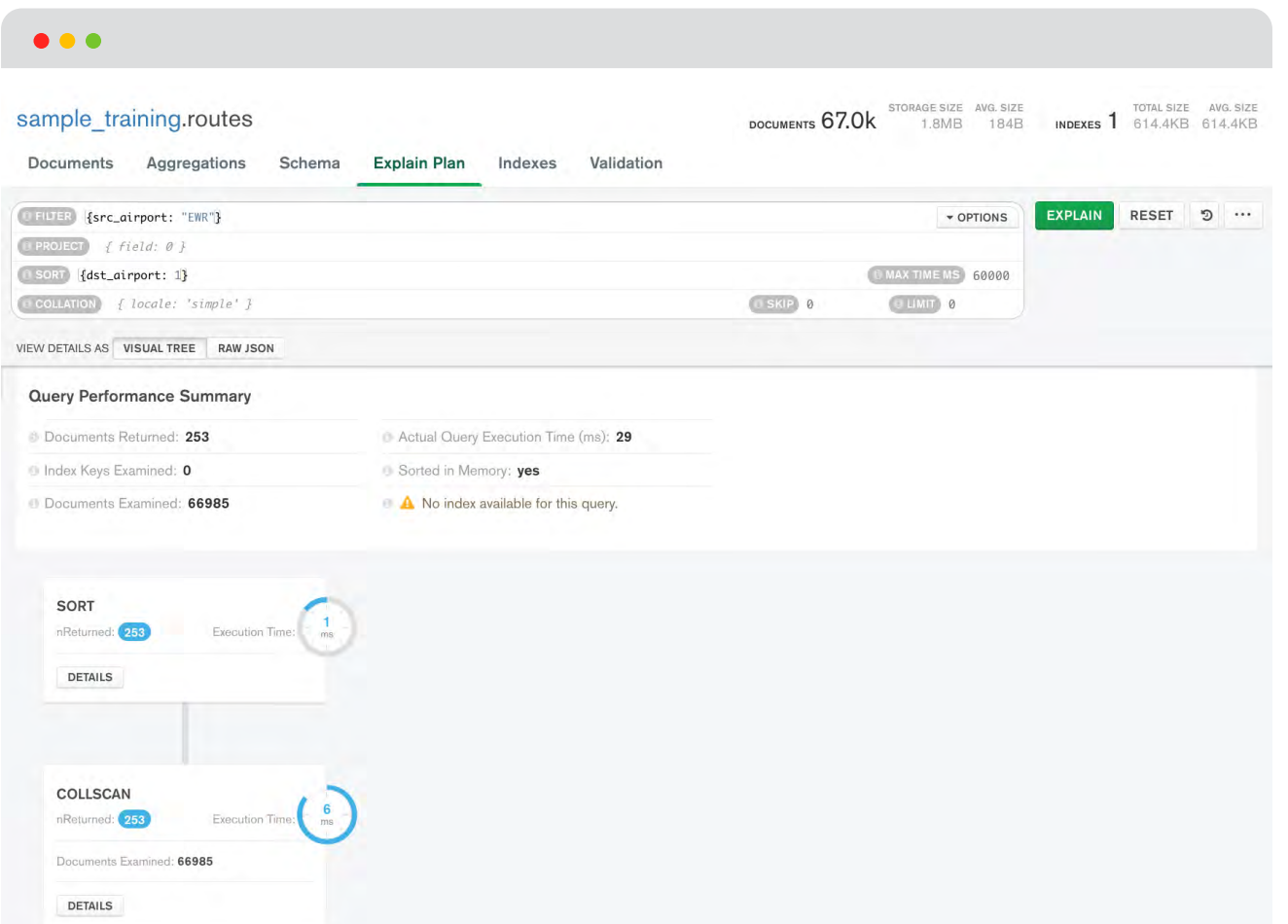


Figure 5: MongoDB Compass visualizes explain output, making it easier to identify and resolve performance issues.

Built-in slow query profiling is also available if you’re deploying MongoDB with Atlas.

Optimizing Data Access Patterns

Native tools in MongoDB for improving query performance and reducing overhead.

Indexing

Indexes support the efficient execution of queries. Without them, the database must scan every document in a collection or table to select those that match the query statement. If an appropriate index exists for a query, the database can use the index to limit the number of documents it must inspect.

Index planning based on your workload is important to ensure that your queries perform.

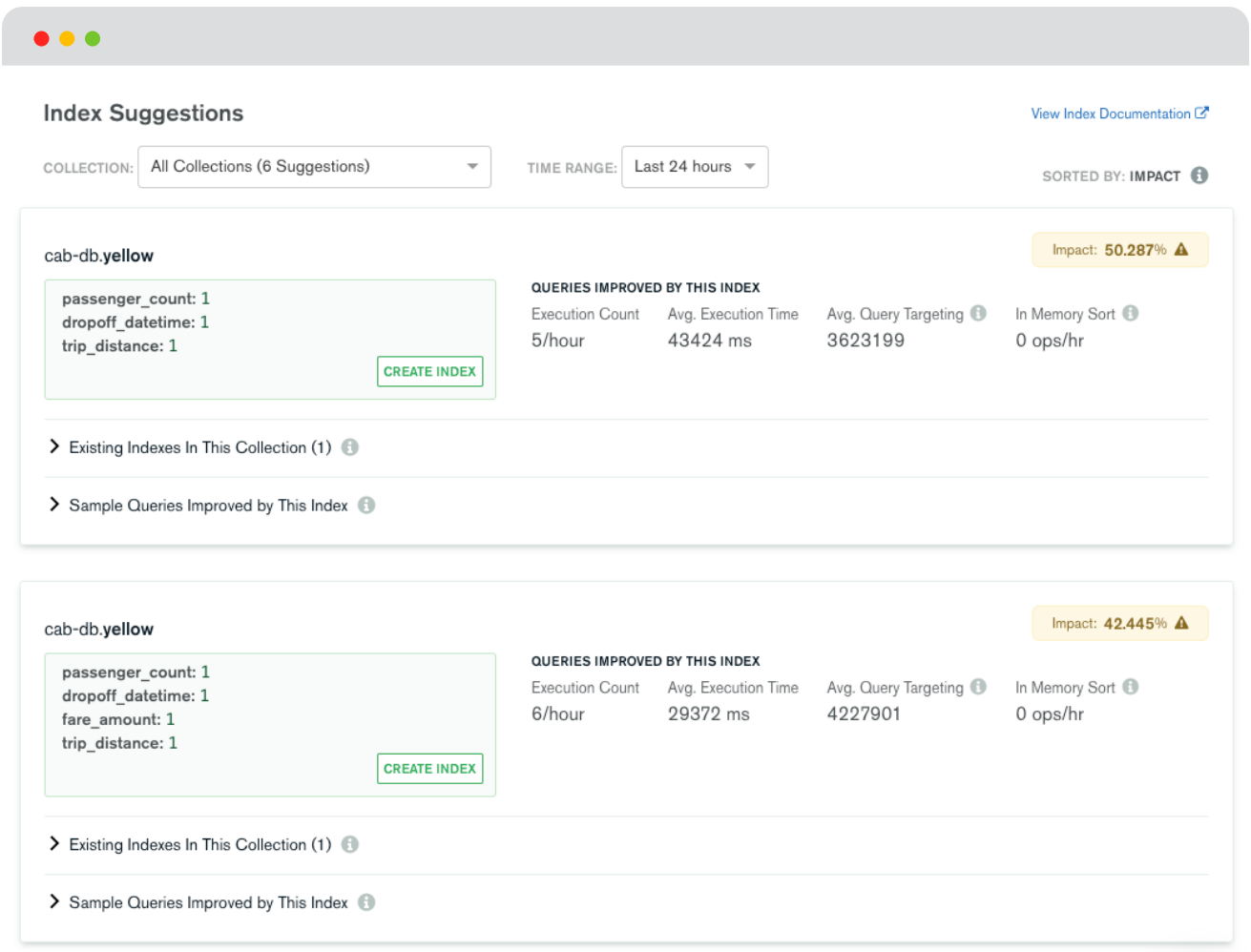


Figure 6: If you're running fully managed databases on Atlas, the built-in Performance Advisor monitors queries that take more than 100 ms to execute and automatically suggests new indexes to improve performance

Indexing

Compound indexes are composed of several different fields. For example, instead of having one index on “Last name” and another on “First name,” it is typically more efficient to create an index that includes both “Last name” and “First name” if you query against both of the names.

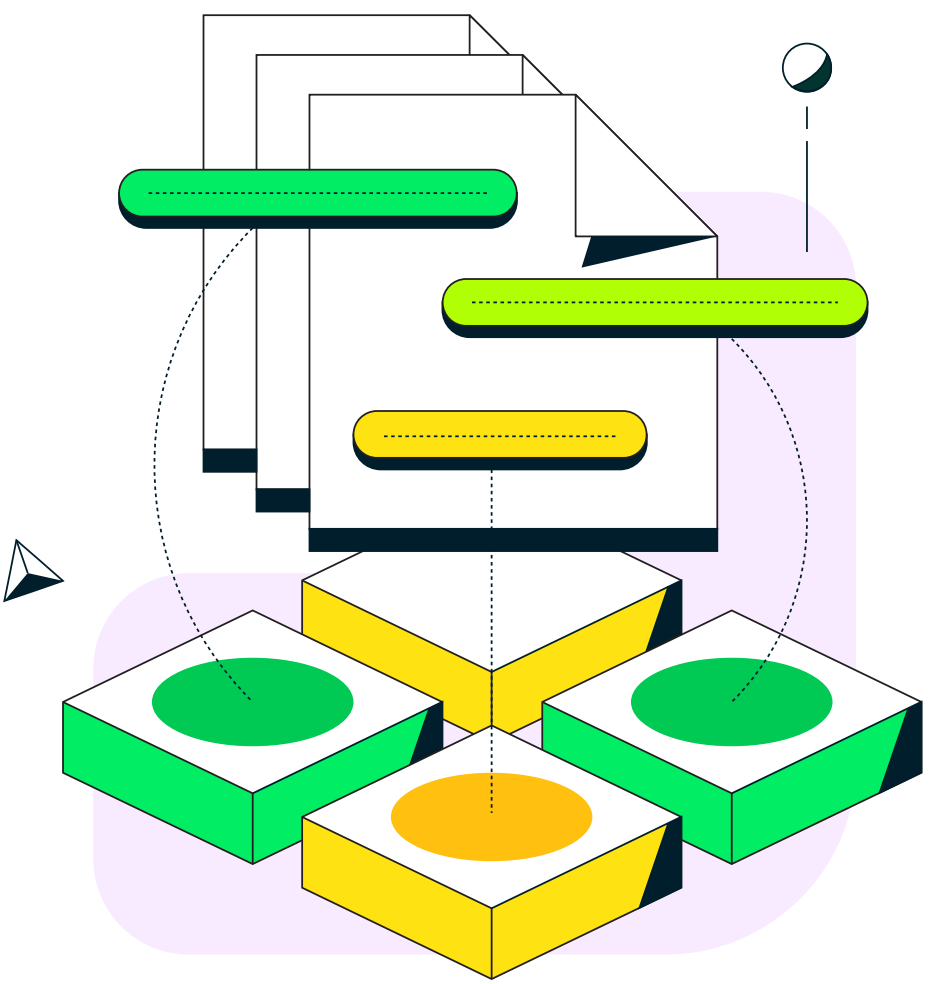
For compound indexes, use the ESR rule to decide the order of fields in the index:

- First, add those fields against which Equality queries are run.
- The next fields to be indexed should reflect the Sort order of the query.
- Finally, the last fields represent the Range of data to be accessed.

Use Covered queries where possible. Covered queries return results from an index directly without having to access the source documents, and are therefore very efficient.

For a query to be covered, all the fields needed for filtering, sorting, and being returned to the client must be present in an index. To determine whether a query is a covered query, use the explain() method. If the explain() output displays totalDocsExamined as 0, this shows the query is covered by an index.

Learn more about using explain().



Indexing

Eliminate unnecessary indexes. Indexes are resource-intensive and consume RAM and disk. As fields are updated, associated indexes must be maintained, incurring additional CPU and disk I/O overhead.

If you're running fully managed databases on MongoDB Atlas, the built-in Performance Advisor suggests dropping unused, redundant, and hidden indexes to improve write performance and increase storage space.

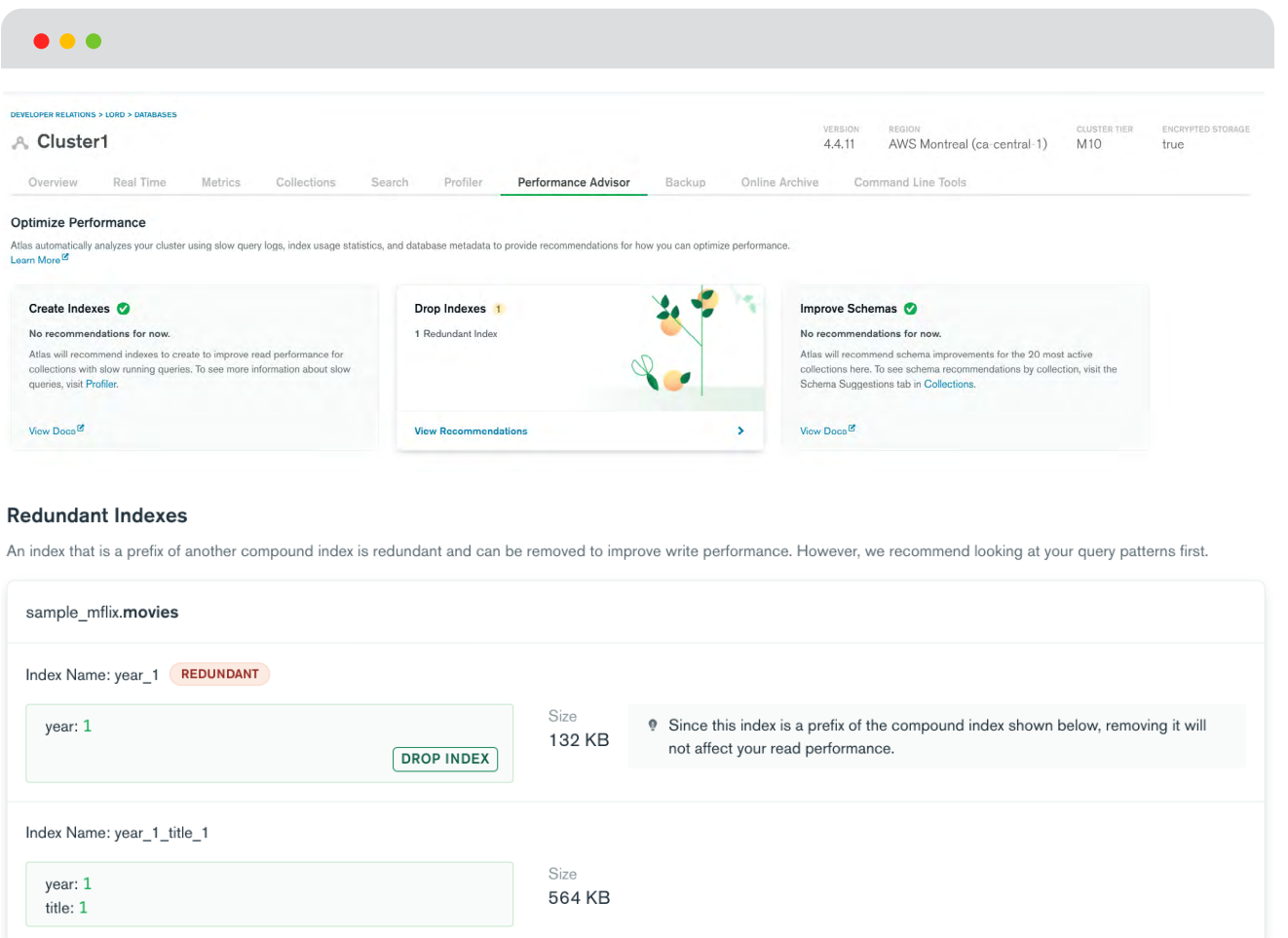


Figure 7: Remove indexes that are unused, either because the field is not used to query the database or because the index is redundant.

Reduce the size and performance overhead of indexes by only including documents that will be accessed through the index. For example, create a partial index on the orderID field that only includes order documents with an orderStatus of "In progress," or only indexes the emailAddress field for documents where it exists.

Workload Type: Analytics

If your application performs complex or long-running operations, such as BI and data visualizations, you may want to isolate analytics queries from the rest of your operational workload. By isolating different workloads, you can ensure different query types never contend for system resources.

If you are running MongoDB on your own infrastructure, you can configure replica set tags to achieve read isolation.

With MongoDB Atlas, you can achieve workload isolation with dedicated analytics nodes. Visualization tools like Atlas Charts can be configured to read from analytics nodes only.

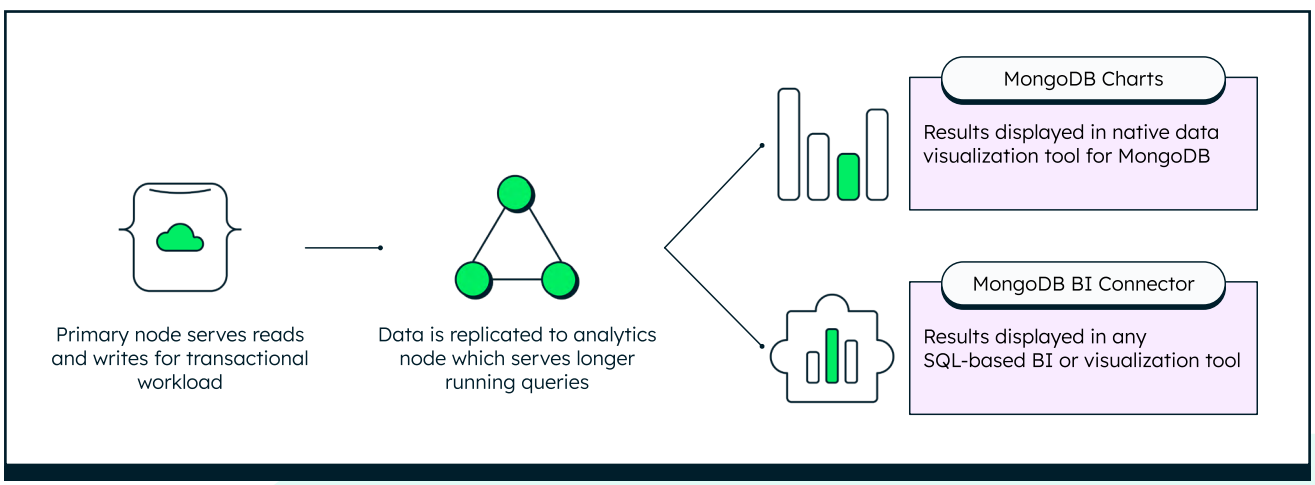


Figure 8: Serve multiple workloads simultaneously without data duplication or performance impact using workload isolation.

Workload Type: Search

If your application requires rich full-text search functionality and you are running MongoDB on Atlas, consider using Atlas Search. The service is built on fully managed Apache Lucene but exposed to users through the MongoDB Aggregation Framework.

Atlas Search is built for the MongoDB document data model and provides higher performance and greater flexibility to filter, rank, and sort through your database to quickly surface the most relevant results to your users.

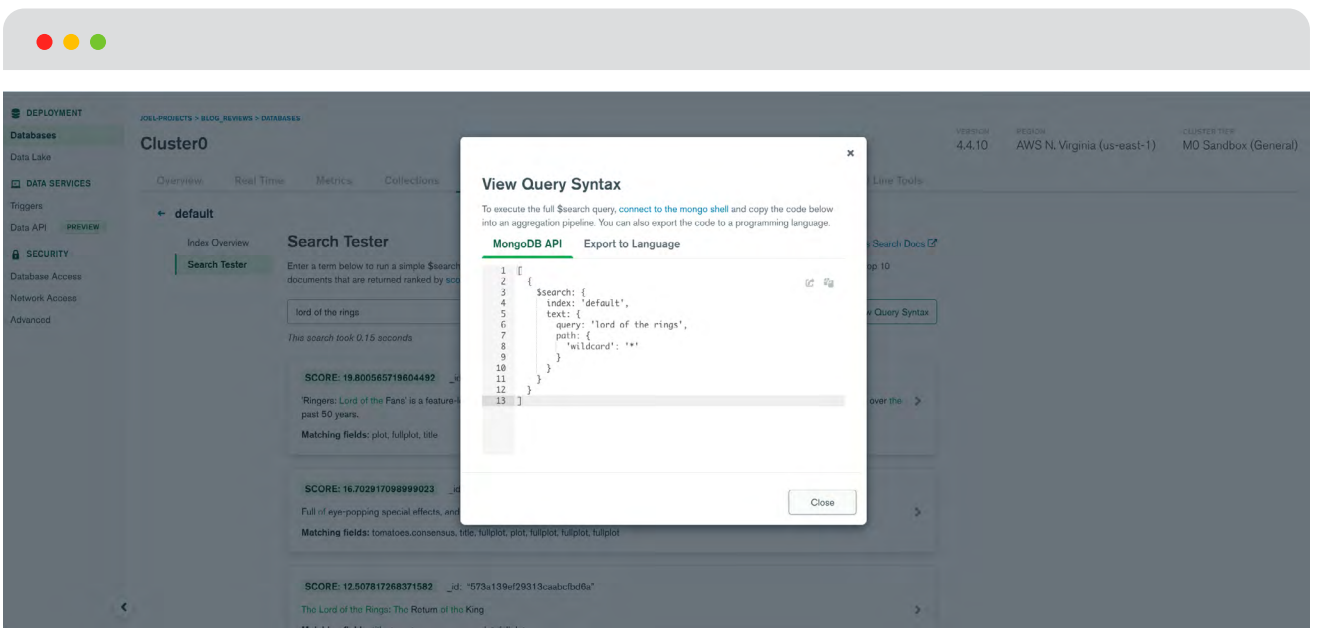


Figure 9: Atlas Search queries are expressed through the MongoDB Query API and backed by the leading search engine library, Apache Lucene.

Learn more about Atlas Search.

Availability and Scaling

Achieve service level objectives with replica sets and sharding.

Resiliency and Scale

With MongoDB, continuous availability of data is achieved through replication and automated failover. A replica set is a group of nodes that maintain the same data set and are the basis for all production deployments. Within a replica set, there is always one primary node, which by default serves all reads and writes. All other members are called secondaries.

Replica sets use elections to determine which set member will become the primary. Production deployments should have a minimum of three members in a replica set to ensure that a majority can be achieved during election and failover.

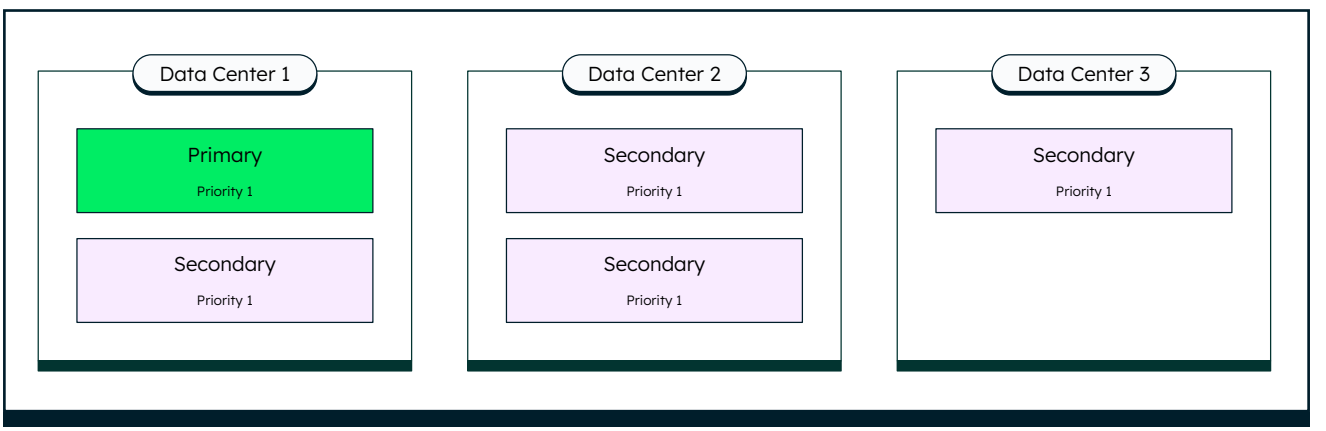


Figure 10: With horizontal scaling, secondary servers are available to respond if the primary goes down.

If possible, distribute replica set members across multiple data centers. If you're using MongoDB Atlas, the service will automatically distribute replica set members across the availability zones within your selected cloud region, ensuring no single point of failure.

Learn more about replication in MongoDB.

Resiliency and Scale

While the primary within a replica set serves reads by default, MongoDB allows you to configure secondary reads for several use cases:

- Scaling read operations
- Reducing read latency by placing a secondary in a data center close to end users
- To isolate analytics workloads that would otherwise compete for resources with ongoing operational workloads

With secondary reads, data is eventually consistent by default.

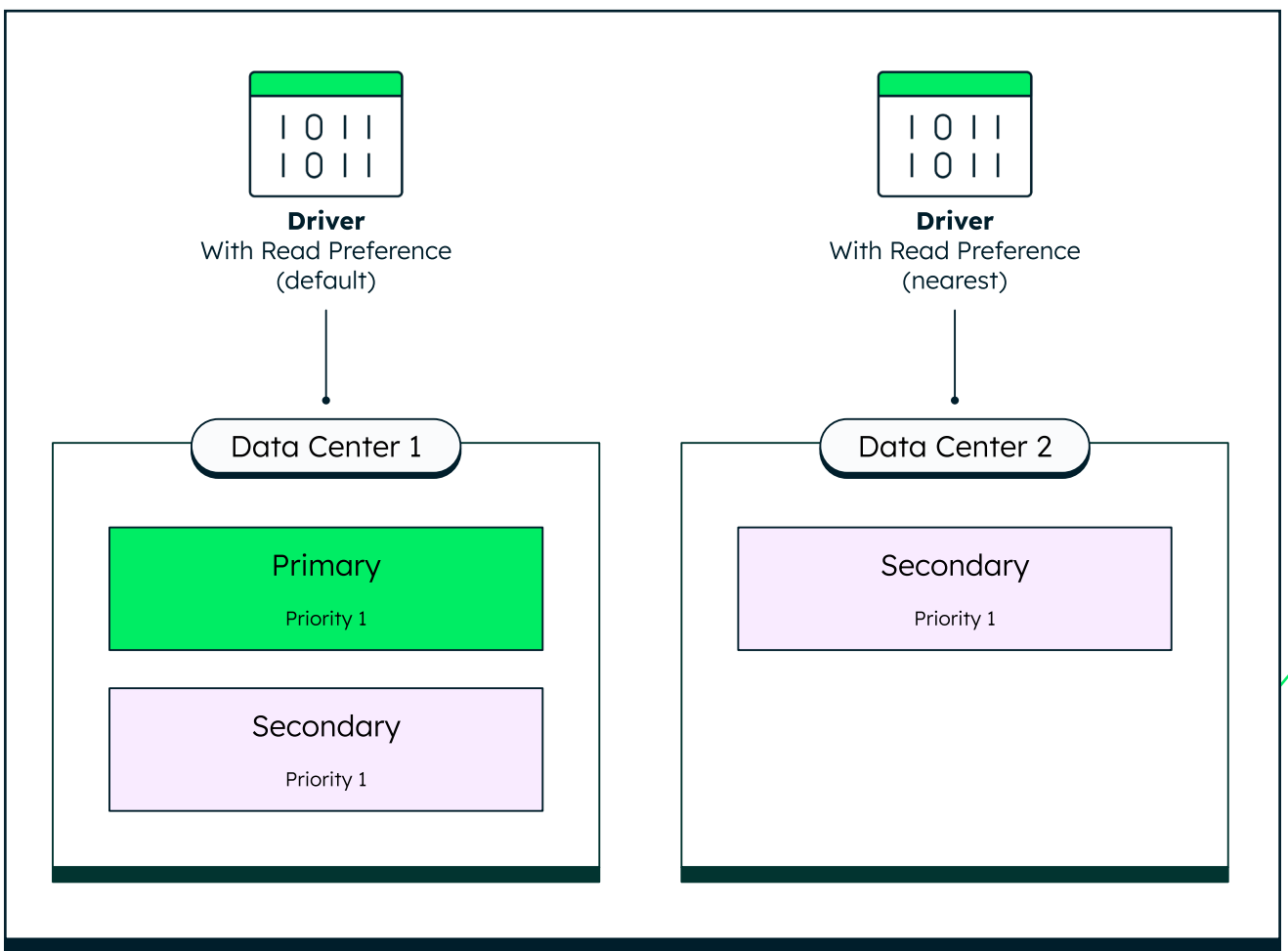


Figure 11: By default, an application directs its read operations to the primary member in a replica set, but you can specify to send read operations to secondaries.

In the example on the right, the driver reads from a member whose network latency falls within the acceptable latency window.

Learn more about read preference.

Resiliency and Scale

As with all databases, vertically scaling MongoDB is always an option. MongoDB also supports horizontal scaling through a process called sharding. Sharding allows you to automatically partition your data across multiple nodes and regions to address growing data sizes, write-intensive workloads, and data residency requirements.

To respond to changing workload demands, documents can be migrated between shards, and nodes can be added or removed from the cluster at any time. With the latest version of MongoDB, you can easily change your sharding strategy (see Live Resharding) and how your data is distributed, with no downtime.

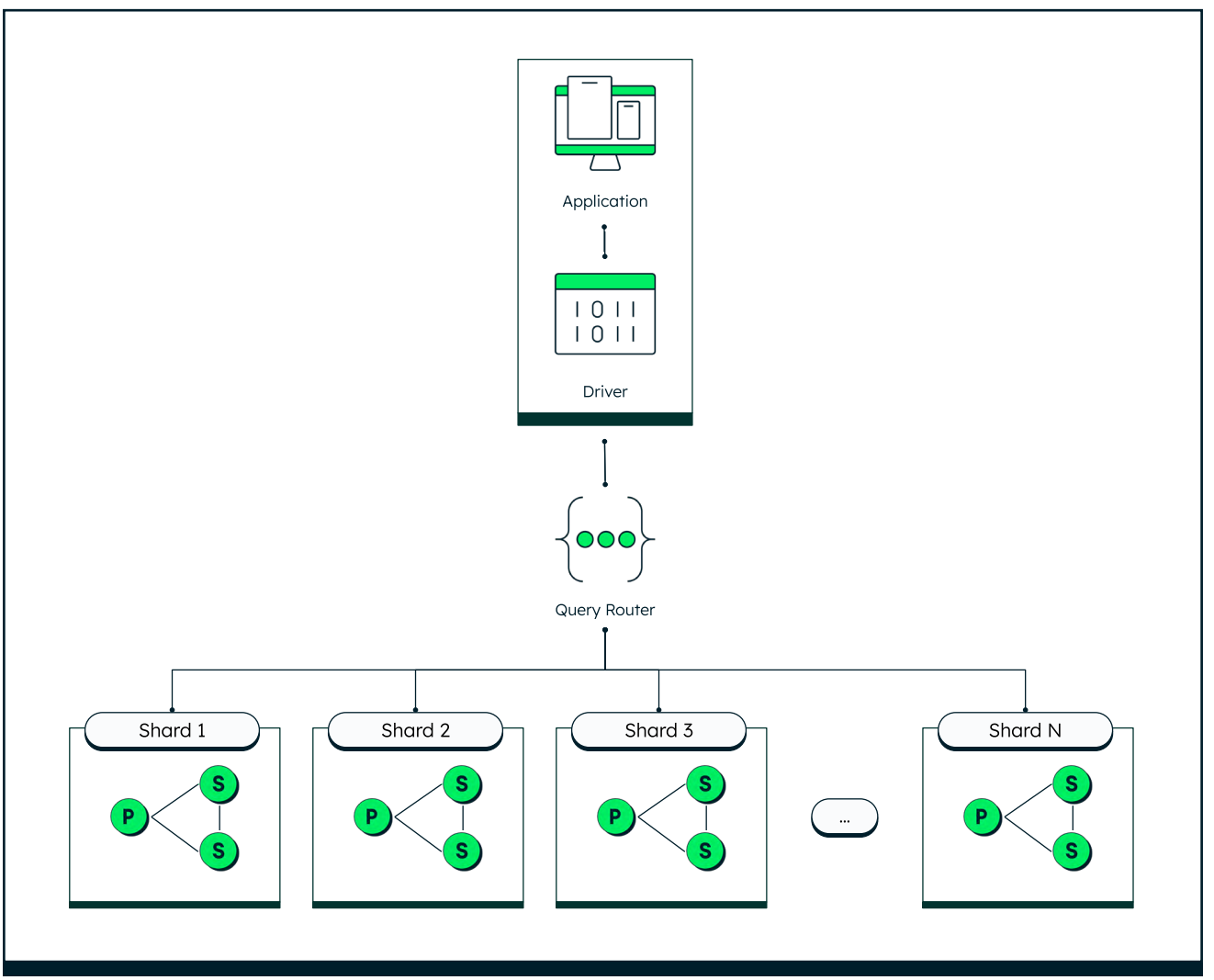


Figure 12: Through sharding, you can automatically scale your MongoDB database out across multiple nodes and regions to handle write-intensive workloads and growing data sizes, and for data residency.

The image above shows a sharded MongoDB cluster with N shards. Each shard is a replica set, ensuring the continuous availability of each partition of the data set.

Learn more sharding best practices.

Resiliency and Scale

MongoDB supports multiple sharding strategies to allow you to more closely match your data access patterns or data placement requirements.

- Ranged Sharding – Documents are partitioned across shards according to the shard key value (e.g., A-M are on one shard, N-Z are on another).
- Hashed Sharding – Documents are distributed according to an MD5 hash of the shard key value, which guarantees a uniform distribution of writes across shards when necessary for a given use case.
- Zoned Sharding – You can define specific rules governing data placement in a sharded cluster.

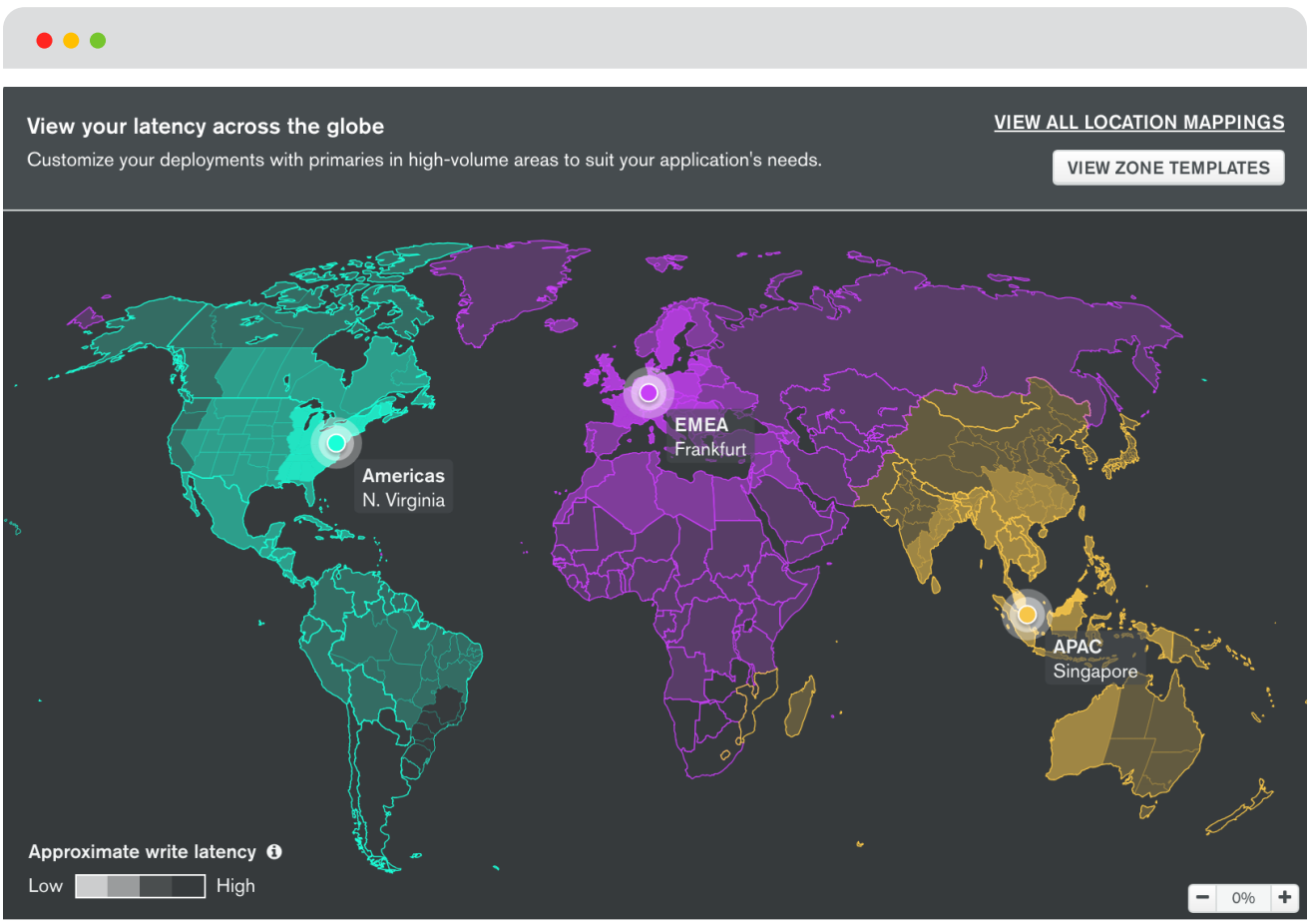


Figure 13: Atlas Global Clusters use a highly curated implementation of sharded cluster zones to support location-aware read and write operations for globally distributed application instances and clients.

Global Clusters in MongoDB Atlas allows you to quickly implement zoned sharding using a visual UI or Admin API. You can easily create distributed databases to support geo-distributed apps, with policies enforcing data residence within specific regions.

Learn more sharding best practices.

Resiliency and Scale

With sharding, ensure a uniform distribution of shard keys. When shard keys are not uniformly distributed for reads and writes, operations may be limited by the capacity of a single shard.

In sharded deployments, queries that cannot be routed by shard key are broadcast to all shards for evaluation. It's recommended that you include the shard key in your queries to prevent these scatter-gather queries. The exception to this rule is for aggregations, which may benefit from parallelizing the query across multiple shards.

Ranged sharding can be great for applications that regularly issue range-based queries, but it requires a good understanding of your data and access patterns; don't be afraid to use hash-based sharding when appropriate.

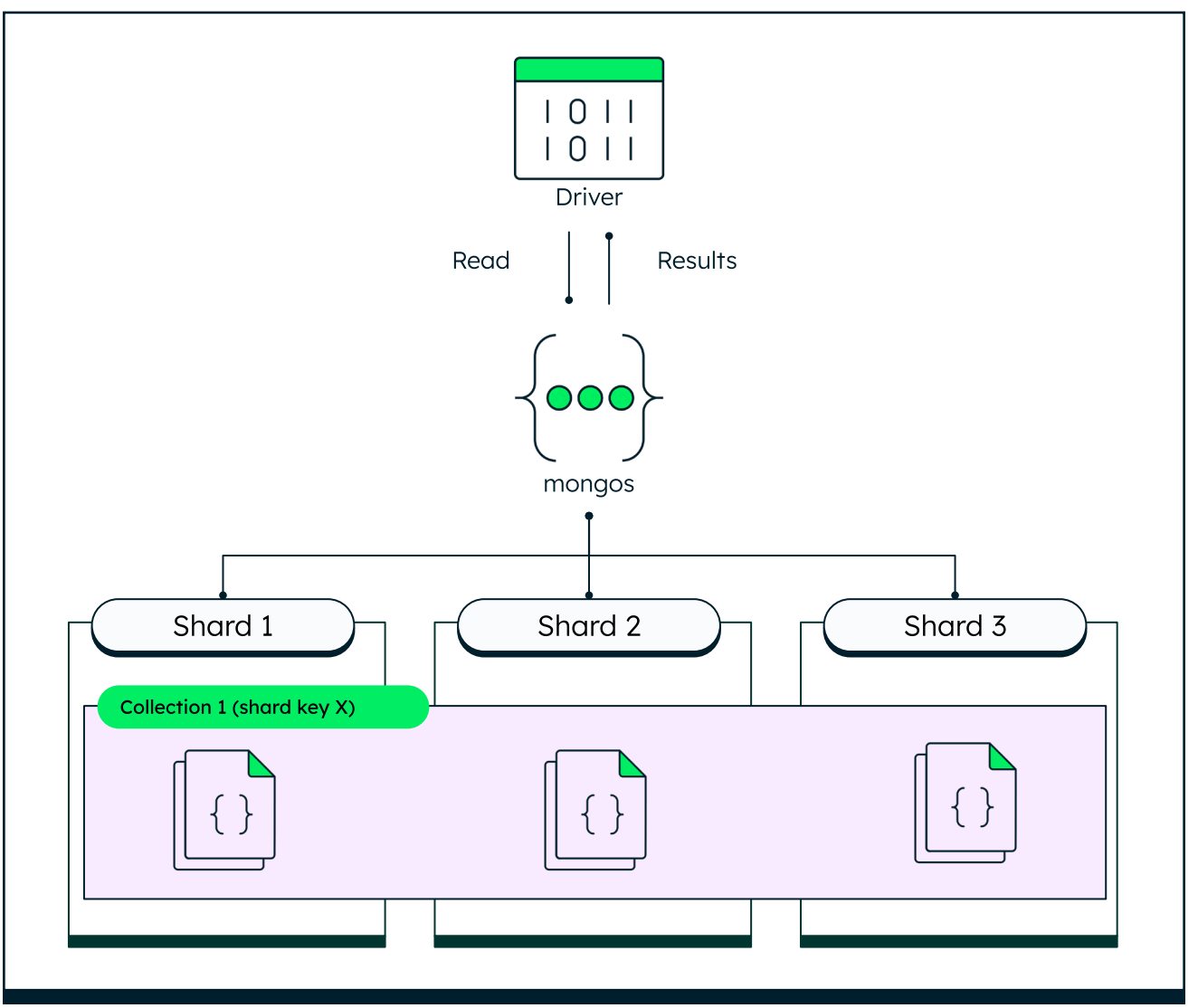


Figure 13: Scatter-gather queries (shown above) are typically unfeasible for routine operations.

Learn more sharding best practices.

Resiliency and Scale

Automate capacity management and auto-scale storage in response to workload changes using Atlas Cluster Auto-Scaling:

- Storage auto-scaling is enabled by default. Additional disk space is provisioned when usage reaches 90%.
- Cluster tier auto-scaling is opt-in – define the minimum and maximum sizes and Atlas will auto-scale your cluster based on key utilization metrics against thresholds.
- All auto-scaling events are applied on a rolling basis to ensure no downtime to connected applications.

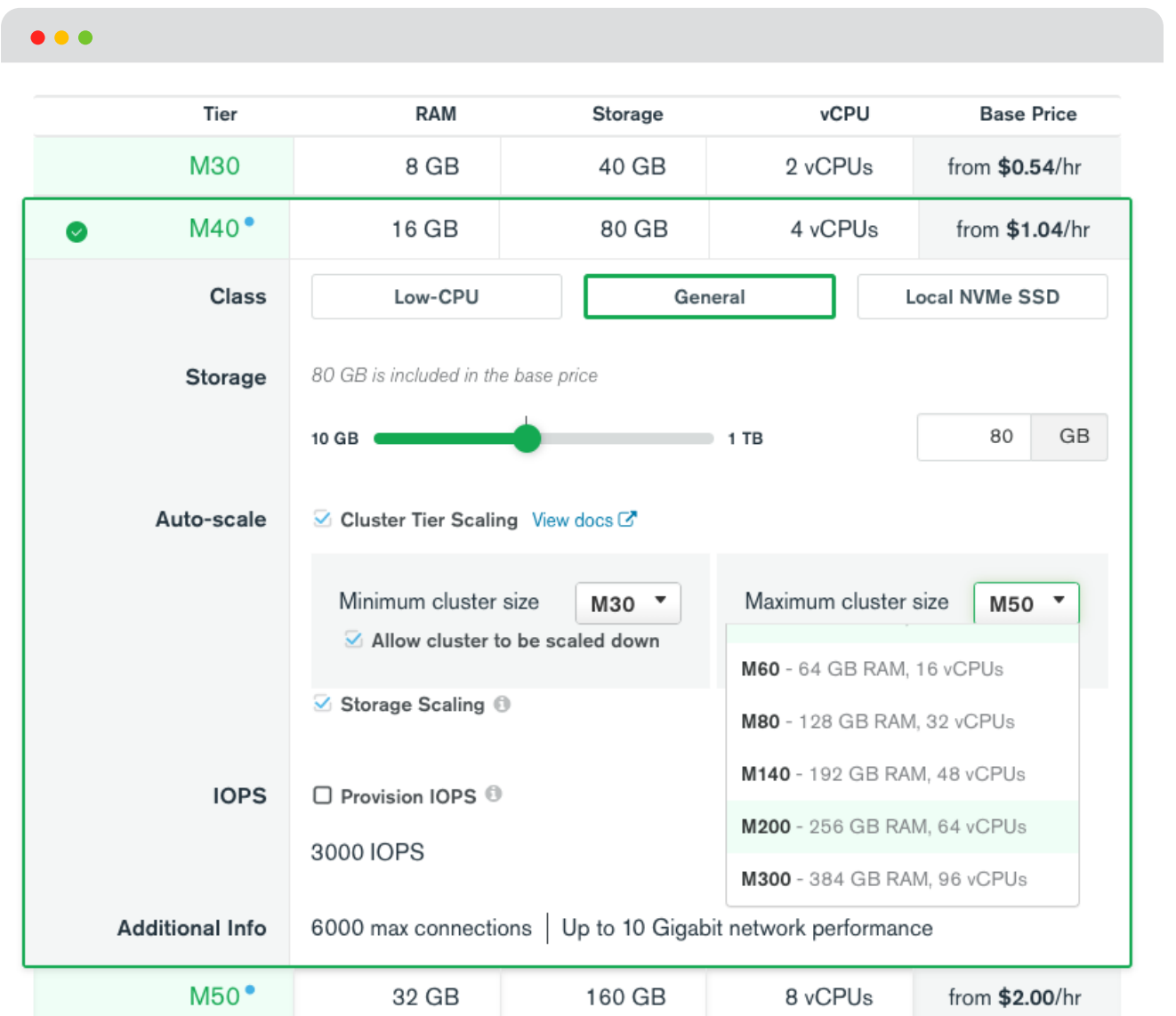


Figure 14: You can configure the cluster tier ranges that Atlas will use to automatically scale your cluster tier, storage capacity, or both in response to cluster usage.

Learn more about automated capacity management.

More Resources

For more on getting started in MongoDB:

Review our Use Case Guidance White Paper to identify applicable use cases for MongoDB.

Spin up a fully managed MongoDB cluster on the Atlas free tier, or download MongoDB for local development.

Review the MongoDB manuals and tutorials in our documentation.