



MDB200

Intro to Atlas Search & Vector Search

MongoDB Optimization and Performance



Topics we cover

What is Atlas Search

How to set up Atlas Search Index

What is Atlas Vector Search

How to set up Atlas Vector Search Index

Additional Atlas Vector use cases (RAG)

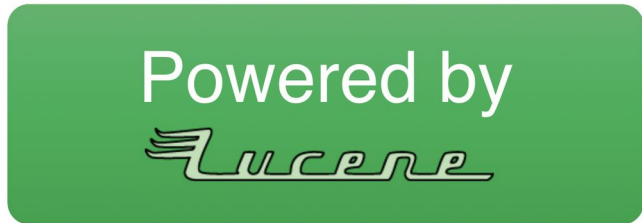
Search nodes

Atlas Search





What is "Atlas" Search?



Atlas Search is an addition to the hosted MongoDB database allowing the creation of indexes to support Search using Apache Lucene, the world's leading text search technology.

Search allows you to make additional searchable indexes in a different format optimised for relevance searching.

Although this is something that can be built and configured yourself with a self-hosted MongoDB, it's complicated to do well. Users often have to install, configure, and pay for Elasticsearch or SOLR to do this.

With Atlas, it's as simple as specifying what data you want to search and the way you intend to search it. The indexes are then built and maintained for you in the background.

And as a developer, you can now perform relevance-ranked searching (and more) on behalf of your users.

Best of all, it's all integrated, managed, and has no additional licensing costs or splitting of support between multiple organisations.



What do we mean by search?

"With a database query - you know exactly what you want.
When you use search you are open to suggestions." - unknown

```
db.movies.find({ title : "Raising Arizona"})
```

Title IS EXACTLY "Raising Arizona"

```
db.movies.find({ title: /Raising Arizona/})
```

Title CONTAINS the phrase "Raising Arizona."

```
db.movies.aggregate([{$search : { text : { query : "Raising Arizona", path: "title" }}}])
```

Title mentions some variations of "Raising," "Arizona," or both

MongoDB is a database. As a database, it can safely perform storage and retrieval of records with guarantees of safety of interaction between multiple users and durability of writes and predictable interaction between them.

It's a primary data store and a system of record.

Atlas is MongoDB, hosted on public cloud, with day to day maintenance done for you by teams inside MongoDB - allowing you to focus on using the database, rather than running it.

So what is Atlas Search?

The "Read" in CRUD, which includes `find()` and `aggregate()` for MongoDB, identifies all documents exactly matching a specific criteria.

This match is boolean. A document matches - or does not - and the notion of matching is very exact.

Search, on the other hand, accepts that some things match better than others and allows us to order by how well they match.

This is most notable in filtering by text fields. Rather than an exact match, we can find where fields contain some of the query terms, or even similar query terms.



Scoring and Relevance

Search: **Javascript Book**

Title	Score
Javascript the Definitive Guide	75
Javascript the Good Parts	60
The Beginners Guide to Node.js	35
Angular for Dummies	25
Introducing MongoDB	10

7

When we use Atlas Search, results are sorted not by a specific value in the data, like a normal database sort, but by those that match our query the best.

To do this, each result is assigned a "score." How exactly that score is assigned is something we have a lot of control over in Atlas Search.

We will be looking at score in more detail later on.

This score is said to denote the "relevance" of a result, so by sorting and getting the highest score first, we can see the results our user will most want to see, the "most relevant."

Accessing Atlas Search





Loading Sample Data in Atlas

Connect to an M0 Atlas cluster from a mongo shell
Add a small number of documents

```
> use searchtraining
> db.example.drop()

//N.B. There are deliberate spelling errors below

> db.example.insertMany([
  { text: "This is a sentence and I can search for it using just one word" },
  { text: "This is a a different sentence, and I can search for it using one or two words" },
  { text: " Whether it is one word or two words could make a difference to the results" }
])

> db.example.find({})
```

9

In order to best see the functionality of Atlas Search, we are going to add a tiny dataset. This makes it easier to understand our results in the context of every record in the collection.

Atlas Search works very quickly on any size of data but by using just a few documents, we can see what it does and does not return and how it scores them relatively.

Run the following in a MongoDB shell to create the sample data:

```
use searchtraining
db.example.drop()
db.example.insertMany([
  { _id:1, text: "This is a sentence and I can search for it using just one word" },
  { _id:2, text: "This is a a different sentence, and I can search for it using one or two words" },
  { _id:3, text: " Whether it is one word or two words could make a difference to the results" }
])
```

You can then see this in the collections tab in Atlas.



View Documents in Atlas

The screenshot shows the MongoDB Atlas interface. The top navigation bar includes tabs for Overview, Real Time, Metrics, Collections (selected), Search, Profiler, Performance Advisor, Online Archive, and Command Line. Below the navigation bar, it says 'DATABASES: 9 COLLECTIONS: 22'. On the left, there's a sidebar with a '+ Create Database' button and a list of namespaces: sample_airbnb, sample_analytics, sample_geospatial, sample_mflix, sample_restaurants, sample_supplies, sample_training, sample_weatherdata, searchtraining (expanded), and example. The main panel displays the 'searchtraining.example' collection. It shows 'COLLECTION SIZE: 315B', 'TOTAL DOCUMENTS: 3', and 'INDEXES TOTAL SIZE: 20KB'. There are tabs for Find (selected), Indexes, Schema Anti-Patterns, Aggregation, and Search Indexes. A 'FILTER' input field contains '{"filter":"example"}' and a 'Find' button. Below, it says 'QUERY RESULTS 1-3 OF 3'. The results show two documents with their _id and text fields.

Document 1	Document 2
<pre>{ "_id": "60dc457d1383026238dc48f3", "text": "This is a sentence and I can search for it using just one word" }</pre>	<pre>{ "_id": "60dc457d1383026238dc48f4", "text": "This is a different sentence, and I can search for it using one or ..." }</pre>

Inside Atlas, use "Browse Collections" to take a look at the three documents you added, complete with deliberate spelling mistakes. They are in a database called searchtraining and a collection called example.



Creating a Default Search Index

Create a Search Index

Configuration Method | Name & Data Source | Review & Refine

Configuration Method

Select how you would like to build and customize your Atlas Search index. You can also create, edit, and manage Atlas Search indexes using the Atlas API.

Visual Editor
Learn about index definitions in a more guided way.

The Visual Editor does not currently support custom analyzers. At this time, Atlas Search indexes cannot be created for time series collections.

JSON Editor
Edit the raw index definition with an unobscured editor.

Cancel

Index Name and Data Source

Atlas Search indexes are specific to a database and collection. Give your index a name, select the right database, and select the database and collection you want to index data from.

Index Name: default

Databases and Collections

- search_training
- search_training
- search_training
- search_training
- search_training
- search_training

Create a Search Index

Configuration Method | Name & Data Source | Review & Refine

Review "default" for searchtraining.example | View JSON

By default, your Atlas Search index will have the following configurations. We recommend starting with this and refining it later if you need to.

Index Analyzer	Parse data to be indexed	lucene.standard
Search Analyzer	Parse data in search queries, usually same as Index Analyzer	lucene.standard
Dynamic Mapping	Automatically index common data types in a collection	On
Field Mappings	Specify data types and input parameters	None

You can edit these defaults at any time to fine-tune relevance and improve search performance.

Refine Your Index

Back | Cancel | Create Search Index

11

Now that we have inserted some documents and we can connect and see them in the GUI and via a shell, we need to tell Atlas we want to create a Search index for this data.

To do this, we click the Search tab and then "Create Search Index." As this is our first time, it takes us through a simple path.

We can use either a GUI editor (The Visual Editor) to configure our index or edit a JSON configuration file. For this first example, we will choose Visual Editor and click Next.

In the next screen, will keep the index name as "default." The index Atlas search uses this index if nothing is explicitly specified.

We need to unfold the **searchtraining** database and select the **example** collection.

We will accept all the default settings for now.

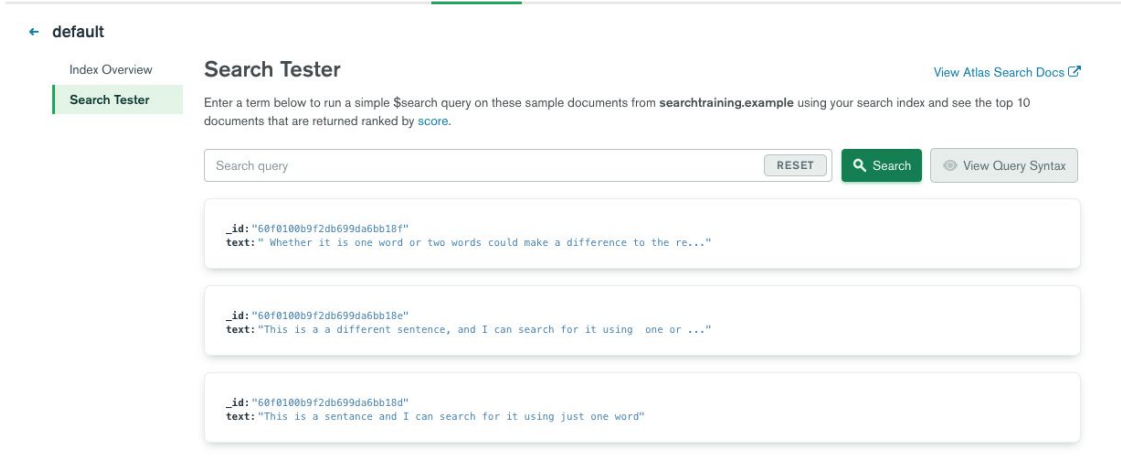
We are using the "Standard" text analyzer, so strings will be split into searchable values based on spaces and punctuations but will not know any specifics of the language.

We are using the "dynamic" mapping modes which means all number, text, and date fields will be indexed, including those in arrays or objects.

Click "Create Search Index" and then close the window that pops up. The index will take a few minutes to create.

We know it's done when it says "3 (100%) indexed of 3 total."

Performing a Simple Text Search



The screenshot shows the 'Search Tester' interface. At the top left, there is a navigation bar with a back arrow and the text 'default'. Below this, there are two tabs: 'Index Overview' and 'Search Tester', with 'Search Tester' being the active tab. To the right of the tabs is a link 'View Atlas Search Docs' with an external link icon. Below the tabs, there is a text input field labeled 'Search query', a 'RESET' button, a 'Search' button with a magnifying glass icon, and a 'View Query Syntax' button with an eye icon. Below the input field, there are three document snippets displayed in a light gray box. Each snippet shows an '_id' and a 'text' field. The first snippet has an '_id' of '60f0100b9f2db699da6bb18f' and a 'text' of 'Whether it is one word or two words could make a difference to the re...'. The second snippet has an '_id' of '60f0100b9f2db699da6bb18e' and a 'text' of 'This is a a different sentence, and I can search for it using one or ...'. The third snippet has an '_id' of '60f0100b9f2db699da6bb18d' and a 'text' of 'This is a sentence and I can search for it using just one word'.

12

Once the index is built, then the "Query" button will be available and take us to the "Search Tester."

"Search Tester" is a very basic search application/GUI we can use to see how our search index would look or behave in an application.

Now we can type in a query, just as we would in any other search engine, and see matching results, scores, and highlighted terms.

Try searching for "**one**" - we can see documents that contain the word one in the context (where the word occurs) and also a score as to how good this was.

Try searching for "**one two**" - note we have a higher score for things with both one and two, but we still find the line with just one.

Try a search for "**sentence**" - note we do not find "sentence." Atlas Search can do this, but this GUI does not enable fuzzy searching. We cover that later.

Click "View Query Syntax" to see what it's doing. Note it's searching the index "default." It's a simple text query and the path (fields to search) is set to the wildcard: "*" which means every field in this index.



Searching from Code

View Query Syntax

To execute the full \$search query, connect to the mongo shell and copy the code below into an aggregation pipeline. You can also export the code to a programming language.

MongoDB API Export to Language

```
1 [{
2   {
3     $search: {
4       index: 'default',
5       text: {
6         query: 'sentence',
7         path: {
8           wildcard: '*'
9         }
10      }
11    }
12  }
13 ]
```

Close

```
[[ { $search : {
  index: "default",
  text : {
    query : "sentence",
    path : {
      wildcard : "*"
    }
  }
} ] ] ] ]
```

13

It's nice that the Atlas GUI gives us a specific search GUI, but that's just a basic, completed app. How can we perform this query in our code?

We can see that using the MongoDB shell, which is itself a REPL allowing us to run code and see the results. Using the connect details in your Atlas cluster, open up a MongoDB shell connected to your cluster. It will be a command like this one but with your unique cluster id and username.
`mongosh "mongodb+srv://cluster1.kshqv.mongodb.net" --username myuser`

Now, in the search tester in the GUI, with the intentionally misspelled word *sentence* typed in the box, click "View Query Syntax" and copy to clipboard.

\$search is an **Aggregation** stage. It must be the first stage in the aggregation. If you are unfamiliar with aggregation, it's a set of transformative steps you can apply to documents before they are returned from the database.

We could add more stages like more matching, sorting, limiting the number of results, or rewriting the shape of the output by adding more pipeline stages.

We specify an **index**. Otherwise, it uses 'default.' We are specifying a **text** operation (find values in text field). There are other types of search. We will talk more about this later.

In the the mongodb shell type:

```
use searchtraining
query = <PASTE VALUE FROM GUI or SLIDE HERE>
db.example.aggregate(query)
```

We can now see how easy it is to run a search from the shell or from any other driver.

We will be using the Mongo shell to demonstrate functionality in this class as we need to pass various options, and the Search Test UI has those hard coded for what it does.



Viewing the Result Relevance

We can see the score by using
\$set to add an additional output
field

```
> let query = [ { $search: { index: 'default',  
...   text: { query: 'sentance',  
....   path: { 'wildcard': '*' }  
....   } } },  
... { $set: { score : { $meta: "searchScore" }}}  
... ]  
  
Atlas [primary] searchtraining> db.example.aggregate(quer  
y)  
[  
  {  
    _id: ObjectId("..."),  
    text: 'This is a sentence and I can search for it  
using just one word',  
    score: 0.46227604150772095  
  }  
]
```

14

The number in the search tester GUI, the score, shows how relevant our results are. In this case, relevance is judged on how often the value appears and also how many different fields it appears in.

We can also see on the list which fields it matched in. If a term is shown appearing in the same field multiple times, those will be different instances of the field as part of an array of objects.

To see this scoring information, we have to project a special metadata field called searchScore. We can do this with an additional pipeline stage where we add the score metadata to the output. We use \$set to add a field to the aggregation output.

```
use searchtraining  
let query = [ { $search: { index: 'default',  
    text: { query: 'sentance',  
      path: { 'wildcard': '*' }  
    } } },  
  { $set: { score : { $meta: "searchScore" }}}  
]  
db.example.aggregate(query)
```

Now we can see a score for this result. If we had multiple matching results, they would be ordered by this score.



Exercise

You searched for the word "sentence" and output it with the score in the shell

Try to search for "one" and then "one two" and output those with the score in the shell

Atlas Vector Search

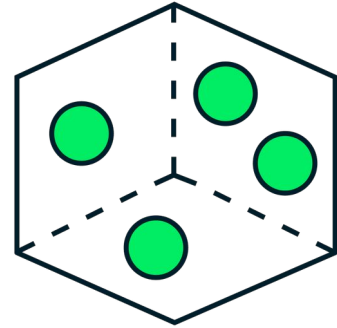




What is Atlas Vector Search?

Atlas Vector search allows:

- Semantic search
- Question and Answer systems
- Image search
- Synonym generation
- Feature extraction
- Recommendations & Relevance Scoring
- Improve AI systems
- Retrieval Augmented Generation (RAG)
- etc.



17

In Atlas you can create and store vector embeddings with machine learning models like OpenAI for retrieval augmented generation (RAG), semantic search, recommendation engines, dynamic personalization, and other use cases.

This new functionality allows you to Build semantic search* and AI-powered applications.

Semantic search uses context clues to determine the meaning of a word across a dataset of millions of examples. Semantic search also identifies what other words can be used in similar contexts.

For example: finding a “sweater” with the query “warm clothing” or “how can I keep my body warm in the winter?”.

We can use vector search to build and enhance Q&A systems and generate Large language model (LLM) memory increasing the context for AI applications, providing up to date information, and reducing AI hallucinations.

Atlas Vector Search allows searching through data based on semantic meaning captured in vectors, whereas **Atlas Search** allows for keyword search i.e. based on the actual text and any defined synonym mappings.



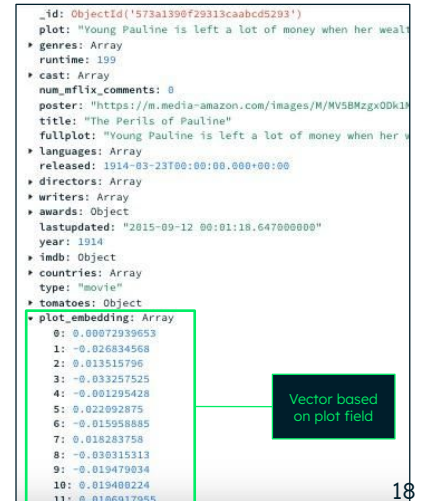
What is Atlas Vector Search?

Search unstructured data based on **semantic meaning** captured in *vectors*

Vectors are numeric representations of the data in an array of doubles that is embedded with the document

Use with \$vectorSearch aggregation stage

```
db.embedded_movies.aggregate([{"$vectorSearch": {"index": "vector_index", "path": "plot_embedding", "queryVector": [0.024552748, -0.015502235, ..., -0.033900633], "numCandidates": 150, "limit": 10} }])
```



18

Vector embeddings are integrated with application data in the same document and seamlessly indexed for semantic queries, enabling you to build easier and faster.

It uses the Hierarchical Navigable Small Worlds algorithm to perform the semantic search. You can use Atlas Vector Search support for approximate Nearest Neighbor (aNN) queries to search for results similar to a queryVector

Note: In the sample data set, under the sample_mflix DB, there is already a movies collection with the embedded array (vector) for the plot field.

The plot_embedding field contains embeddings created using OpenAI's text-embedding-ada-002 embedding model.

See more info here: <https://platform.openai.com/docs/guides/embeddings/what-are-embedding-s>



Creating the Index

In Atlas,

1. Go to Atlas Search
2. Create an Index
3. Select Atlas Vector Search - JSON Editor

Create a Vector Search Index



Configuration Method

Select how you would like to build and customize your search index. You can also create, edit, and manage search indexes using the [Atlas API](#).

NOTE

At this time, search indexes cannot be created for time series collections.

[View Atlas Vector Search Docs](#)

Atlas Search

Visual Editor

Learn about index definitions in a more guided experience.

JSON Editor

Edit the raw index definition with an embedded JSON editor.

Atlas Vector Search

JSON Editor

Create a vector search index definition with an embedded JSON editor.

Cancel

Next

You need MongoDB version 6.0.11, 7.0.2, or higher and the Project Data Access Admin or higher role to create and manage Atlas Vector Search indexes.



Creating the Index

```
{
  "fields": [
    {
      "type": "vector",
      "path": "plot_embedding",
      "numDimensions": 1536,
      "similarity": "euclidean"
    }
  ]
}
```

Database and Collection

- > sample_airbnb
- > sample_analytics
- > sample_geospatial
- > sample_guides
- ▼ sample_mflix
 - ☐ comments
 - ☒ embedded_movies

Index Name

```
1 {
2   "type": "vectorSearch",
3   "fields": [
4     {
5       "type": "vector",
6       "path": "plot_embedding",
7       "numDimensions": 1536,
8       "similarity": "euclidean"
9     }
10  ]
11 }
12
```

The "similarity" field can be set to "euclidean", "cosine", or "dotProduct"

Limitation:

- You can't index fields inside arrays of documents or fields inside arrays of objects
- Atlas Vector Search supports embeddings up to 2048 dimensions

The index definition in the example for the sample_mflix.embedded_movies collection dynamically indexes all the dynamically indexable fields in the collection and statically indexes plot_embedding field as the Vector type.

The plot_embedding field contains embeddings created using OpenAI's text-embedding-ada-002 embeddings model.

The index definition specifies 1536 vector dimensions and measures similarity using euclidean.

Note: You can also optionally index fields for pre-filtering the data against which you want to run your Atlas Vector Search query.

Using vector search

```
db.embedded_movies.aggregate([
  { "$vectorSearch": {
    "exact": false
    "index": "vector_index",
    "path": "plot_embedding",
    "queryVector": [0.024552748,...],
    "numCandidates": 150,
    "limit": 10
  }
},
  { "$project": {
    "_id": 0,
    "plot": 1,
    "title": 1,
    "score": { $meta: "vectorSearchScore" }
  }
}
])
```

```
{
  plot: 'A reporter, learning of time know by averting upcoming disasters.',
  title: 'Thrill Seekers',
  score: 0.7892673227455139
},
{
  plot: 'At the age of 21, Tim discovers his decision to make his world a better place might not be the best thing he could have done.',
  title: 'About Time',
  score: 0.7843684683876838
},
{
  plot: 'Hoping to alter the events of the future, where he finds humankind divided into two groups, he travels back in time to prevent the war.',
  title: 'The Time Machine',
  score: 0.7881866657839185
},
{
  plot: 'After using his mother's new invention, a young boy finds himself in a children's crusade where he must save the world from a mad scientist.',
  title: 'Crusade in Jeans',
  score: 0.7789178742834912
},
{
  plot: 'An officer for a security agency who has a tie to his past.',
  title: 'Timecop',
  score: 0.7771612485776978
},
{
  plot: 'A time-travel experiment in which a man is sent back in time to prevent a disaster.',
  title: 'A.P.E.X.',
  score: 0.7738885744894849
},
{
  plot: 'Agent J travels in time to prevent a disaster and changing history.',
  title: 'Men in Black 3',
  score: 0.7712388886877881
},
{
  plot: 'Bound by a shared destiny, a man on a mission to unearth the secrets of the past.',
  title: 'Tomorrowland',
  score: 0.7669923981567922
},
{
  plot: 'With the help of his uncle, a young boy discovers the secrets of his family's past.',
  title: 'Love Story 2050',
  score: 0.7649372816885815
},
{
  plot: 'A dimension-traveling wizard discovers his magic. With his home world on the brink of destruction, he must save the world.',
  title: 'The Portal',
  score: 0.7648786178959473
}
}
```



21

See a basic example in the documentation:

<https://www.mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage/>

Note: This can take a few minutes as the vector is long

Execute this aggregation command in mongosh using the Instruqt lab.

We have stored a sample embedding into a variable called `vector_embedding_demo`

To set the variable you need to run the `query_embedding.js` file - see steps below.

In the lab terminal:

1. Connect to the atlas cluster
2. Once connected, run:
use sample_mflix

```
load("query_embedding.js")
```

```
vector_embedding_demo
```

```
const pipeline = [ { $vectorSearch: { index: "vector_index", path:
"plot_embedding", queryVector: vector_embedding_demo, numCandidates:
100, limit: 10 } }, { $project: { title: 1, plot: 1, score: { $meta:
"vectorSearchScore" } } }];
```

```
db.embedded_movies.aggregate(pipeline)
```

Change the limit parameter to see the different results returned.

If the "exact" parameter is set to 'true', it enables ENN (Exact Nearest Neighbour). If "exact" is set to false or omitted, search will be ANN (Approximate Nearest Neighbor), and the numCandidates field must be present.

Exercise: Find similar movies with Vector search

Titanic was a great movie, let's get some recommendations on related movies based on this movie plot using vector search

1. Use the `sample_mflix.embedded_movies` collection
2. Find the movie using the title filter:
`{title: "Titanic"}`
3. Use vector search, with the Titanic's `plot_embedding` as the `queryVector` to find movies with similar plots.

```

_id: ObjectId('573a139af29313caabcefb1d')
plot: "The story of the 1912 sinking of the large
• genres: Array (3)
runtime: 173
• cast: Array (4)
poster: "https://m.media-amazon.com/images/M/MV5
title: "Titanic"
fullplot: "The plot focuses on the romances of t
• languages: Array (1)
released: 1996-11-17T00:00:00.000+00:00
rated: "PG-13"
• awards: Object
lastupdated: "2015-08-30 00:47:02.163000000"
year: 1996
• imdb: Object
• countries: Array (2)
type: "series"
• tomatoes: Object
num_mflix_comments: 0
• plot_embedding: Array (1536)

```

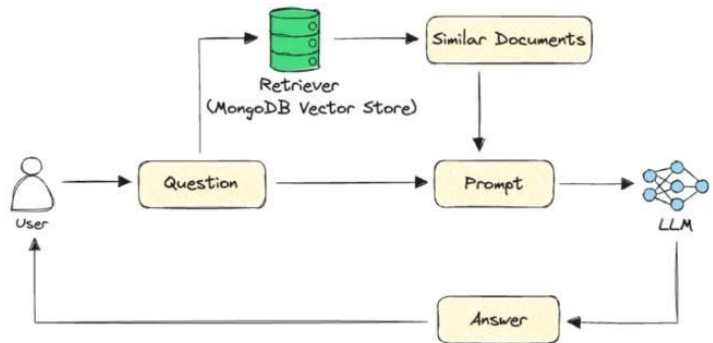
RAG with Vector Search



RAG: Retrieval Augmented
Generation

Prompt the LLM with
relevant information

Reduces hallucinations



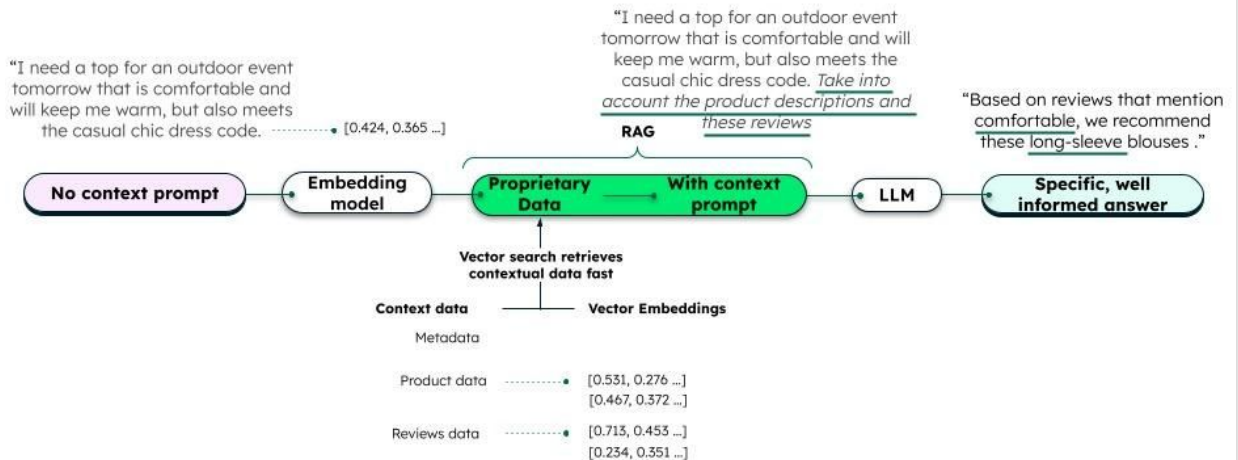
23

Large Language Models (LLMs) have revolutionized the field of natural language processing, but they do come with certain limitations: hallucinations, stale data, no access to users' local data, etc.

RAG uses vector search to retrieve relevant documents based on the input query. It then provides these retrieved documents as context to the LLM to help generate a more informed and accurate response.

Instead of generating responses purely from patterns learned during training, RAG uses those relevant retrieved documents to help generate a more informed and accurate response.

RAG Basic Architecture



24

A basic retrieval-augmented generation architecture consists of these main components:

- A pre-trained LLM: The LLM is responsible for generating text, images, audio, and video.
- Vector search (or semantic search): The retrieval system is responsible for finding relevant information from a knowledge base that is external to the LLM. There are a variety of general purpose databases or single-purpose vector databases to choose from that can store vector embeddings and run approximate nearest neighbor search queries against them. Vector search is core to being able to precisely augment proprietary knowledge provided to a general purpose LLM.
- Vector embeddings: Sometimes referred to simply as "vectors" or "embeddings", vector embeddings are essentially numerical representations capturing the semantic, or underlying meaning of a piece of data. Generally speaking, they're an array of floats, where each float represents a single dimension of the numerical representations.
- Orchestration: The fusion mechanism is responsible for combining the output of the LLM with the information from the retrieval system to generate the final output.

Atlas Search Nodes



Dedicated Search nodes



Search nodes allow:

- Resource Optimization and Flexibility
- Better Performance
- Higher Availability

Search Nodes for workload isolation NEW

Provide nodes dedicated to search indexes to support your mission-critical workloads. [Learn more](#)

Continue to Cluster Tier to select an appropriate tier for your search workload.

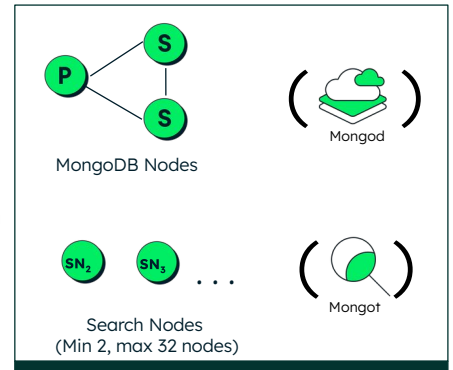
Search Nodes per region:

Region: AWS / Ireland (eu-west-1)

Considerations for creating a cluster with Search Nodes

- You can have a minimum of 2 Search Nodes and up to 32.

☐ I understand and agree to the considerations for creating a cluster with Search Nodes



26

Atlas Vector Search also takes advantage of our new Search Nodes dedicated architecture, enabling better optimization for the right level of resourcing for specific workload needs. Search Nodes provide dedicated infrastructure for Atlas Search and Vector Search workloads, allowing you to optimize compute resources and fully scale search needs independent of the database.

From Atlas version 6.0 (for clusters M10 +), Atlas supports deploying search nodes for Atlas Search and Vector Search separately:

- Resource Optimization and Flexibility: Confidently handle and scale demanding Search workloads
- Better performance : 40% - 60% decrease in query time for many complex queries
- Higher Availability: Workload isolation avoids resource contention between database and search

<https://www.mongodb.com/docs/atlas/cluster-config/multi-cloud-distribution/#std-label-configure-search-nodes>

Quiz Time!





#1. In which of the following scenarios would you use Atlas Search vs MongoDB query?

A

In a database of song lyrics, find a song you heard on the radio

B

In a database of bank statements, find the name of a restaurant you went to

C

In a database of hotels, find one with a pool and archery as activities

D

In a database of customers, find the highest spenders and what they buy

E

Display pop-up adverts on a website

Answer in the next slide.



#1. In which of the following scenarios would you use Atlas Search vs MongoDB query?

A

In a database of song lyrics, find a song you heard on the radio

B

In a database of bank statements, find the name of a restaurant you went to

C

In a database of hotels, find one with a pool and archery as activities

D

In a database of customers, find the highest spenders and what they buy

E

Display pop-up adverts on a website

If you need to run a query where you know exactly what you're looking for with an additional sort in mind, then using a query is the way to go as option D suggests. All other options can be handed by Atlas Search, including finding names of restaurant (implies a like) and even curating ads based on previous searches.



#2. Vector Search enables which 3 features?

A

Semantic searches

B

Structured queries

C

Recommendations & Scoring

D

Image searches

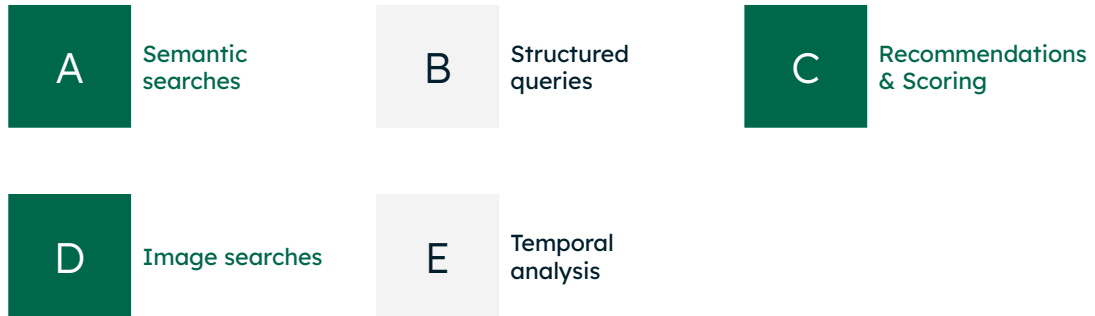
E

Temporal analysis

Answer in the next slide.



#2. Vector Search enables which 3 features?



Vector search doesn't support temporal analysis or structured queries but can be used for A, C, D

Recap

Atlas Search is an Atlas feature that creates indexes to support Search using Apache Lucene.

Search results are scored to rate how well they match the search.

Atlas search includes many search operators that can be combined. It also includes autocomplete, synonyms, and highlight capabilities.

Atlas Vector Search allows searching through data based on semantic meaning captured in vectors



Appendix



Exercise Vector Search - Solution

Use the plot_embedding from Titanic's movie to find similar movies

```
use sample_mflix;
my_movie = db.embedded_movies.findOne({ title: "Titanic" });

pipeline = [
  { "$vectorSearch": {
    "exact": false,
    "index": "vector_index",
    "path": "plot_embedding",
    "queryVector": my_movie.plot_embedding,
    "numCandidates": 150,
    "limit": 10
  } },
  { "$project": {
    "_id": 0,
    "plot": 1,
    "title": 1,
    "score": { $meta: "vectorSearchScore" }
  } }
]
print("Similar movies to: " + my_movie.title);
movies = db.embedded_movies.aggregate(pipeline); // you can add .skip(1) to remove Titanic
```

34

```
// Now try searching for movies based on directors!
// Below should be same movie as {directors: "George Lucas", title: /^Star Wars: Episode IV/ }
my_movie = db.embedded_movies.findOne({ directors: "George Lucas" });

//pipeline is a variable so can be updated in-place.
pipeline[0]["$vectorSearch"].queryVector = my_movie.plot_embedding;

print("Similar movies to: " + my_movie.title);
db.embedded_movies.aggregate(pipeline).skip(1);
```