

Module 3

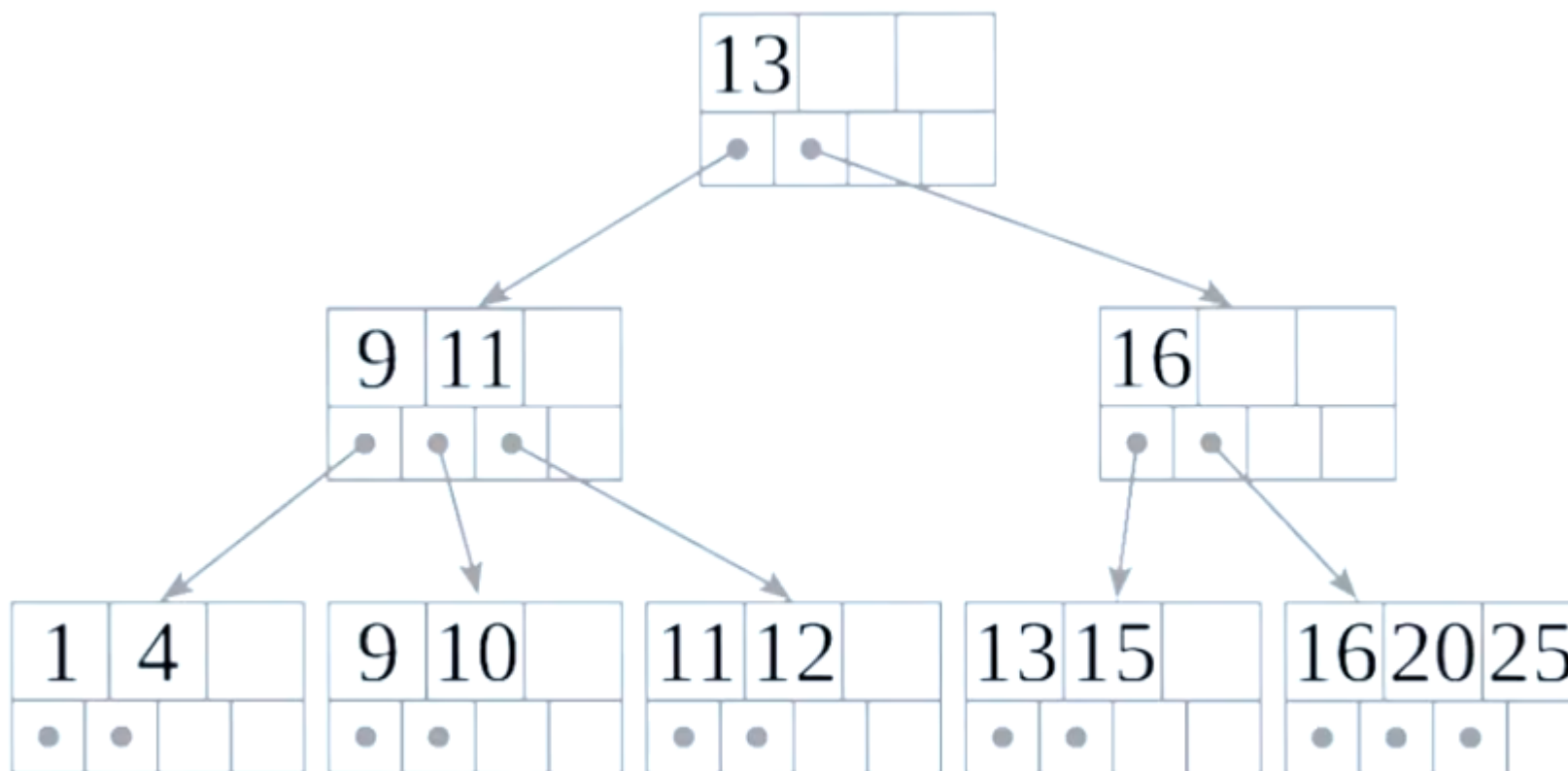
NoSQL Databases

Indexes

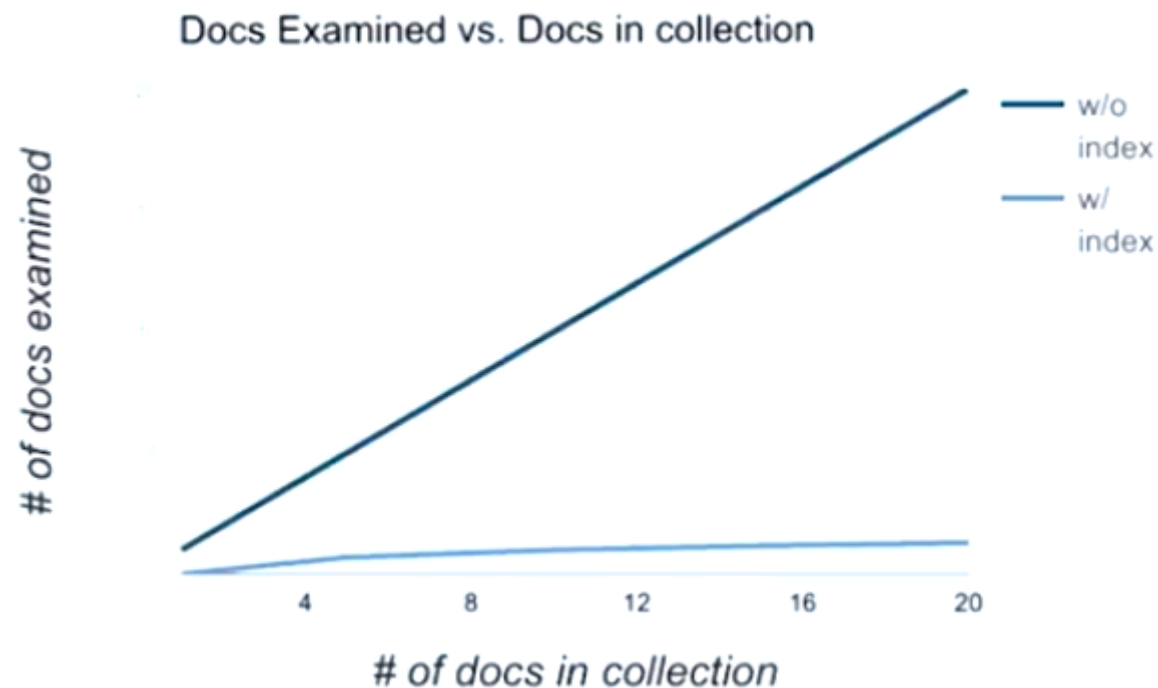
Indexes

- Indexes can support the efficient execution of queries. MongoDB automatically creates an index on the `_id` field upon the creation of a collection.
- Use the `createIndex()` method to create an index on a collection. Indexes can support the efficient execution of queries. MongoDB automatically creates an index on the `_id` field upon the creation of a collection.
- To create an index on a field or fields, pass to the `createIndex()` method an index key specification document that lists the fields to index and the index type for each field:
 - `{ <field1>: <type1>, ... }`
 - For an ascending index type, specify 1 for `<type>`.
 - For a descending index type, specify -1 for `<type>`.
- `createIndex()` only creates an index if the index does not exist.

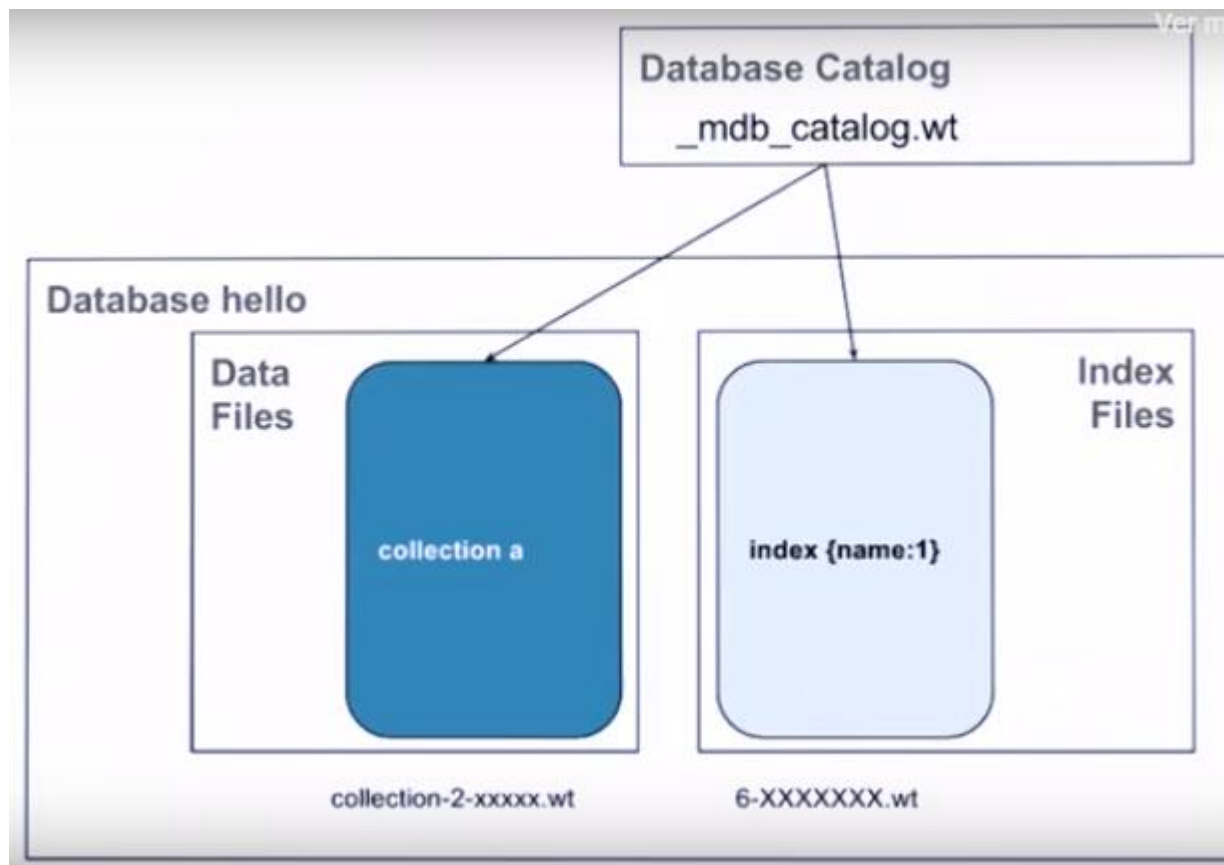
Indexes



Indexes



Indexes



Indexes

```
mongod --dbpath /data/db --fork --logpath /data/db/mongod.log \    --  
directoryperdb --wiredTigerDirectoryForIndexes
```

```
mongod --dbpath /data/db
```

```
/data/db/
```

```
...
```

```
hello/
```

```
collections/
```

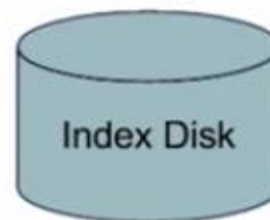
```
...
```



Data Disk

```
index/
```

```
...
```



Index Disk

Indexes

```
mongod --dbpath /data/db --fork --logpath /data/db/mongod.log \
directoryperdb --wiredTigerDirectoryForIndexes
```

```
mongod --dbpath /data/db
```

```
/data/db/
```

```
...
```

```
hello/
```

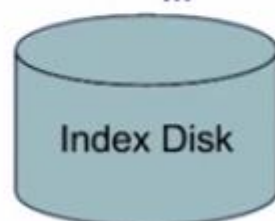
```
collections/
```

```
...
```



```
index/
```

```
...
```



```
storage:
  dbPath: "second_mongo"
  directoryPerDB: true
  wiredTiger:
    engineConfig:
      directoryForIndexes: true
systemLog:
  path: "second_mongo/log.mongod.log"
  destination: "file"
net:
  bindIp: "127.0.0.1"
  port: 4545
security:
  authorization: enabled
processManagement:
  fork : true
```


Indexes

```
mongod --dbpath /data/db --fork --logpath /data/db/mongod.log \      --  
directoryperdb --wiredTigerDirectoryForIndexes
```

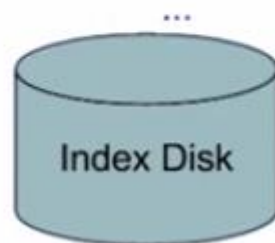
```
mongod --dbpath /data/db
```

```
/data/db/
```

```
...  
hello/
```

```
collections/
```

```
index/
```



```
mongod -f seg2_mongod.conf  
mongo --port 4545 admin  
db.createUser({user: "master-admin",pwd: "master-pass",roles:  
[{"role: "root", db: "admin"}]})
```

```
# Connect as root  
mongo -u master-admin -p master-pass --  
authenticationDatabase "admin"  
use admin  
db.shutdownServer()
```

Indexes

- Create a Single-Field Index

- Create an ascending index on the "cuisine" field of the restaurants collection.

```
db.restaurants.createIndex( { "cuisine": 1 } )
```

- The method returns a document with the status of the operation.

```
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 1,  
  "numIndexesAfter" : 2,  
  "ok" : 1  
}
```

- Upon successful index creation, the "numIndexesAfter" value is one greater than the "numIndexesBefore" value.
- EXERCISE: Use `.explain("executionStats")` to see how it affects to different FIND queries

Indexes

- EXERCISE: Use `.explain("executionStats")` to see how it affects to different FIND queries
 - Without using the indexed field
 - Using the indexed field
 - Using range queries

Indexes

- Create a compound index.
 - MongoDB supports compound indexes which are indexes on multiple fields. The order of the fields determine how the index stores its keys. For example, the following operation creates a compound index on the "cuisine" field and the "address.zipcode" field. The index orders its entries first by ascending "cuisine" values, and then, within each "cuisine", by descending "address.zipcode" values.

```
db.restaurants.createIndex( { "cuisine": 1, "address.zipcode": -1 } )
```

- The method returns a document with the status of the operation.

```
{  
  "createdCollectionAutomatically" : false,  
  "numIndexesBefore" : 2,  
  "numIndexesAfter" : 3,  
  "ok" : 1  
}
```

- Upon successful index creation, the "numIndexesAfter" value is one greater than the "numIndexesBefore" value.
- EXERCISE: Use `.explain("executionStats")` to see how it affects to different FIND queries

Indexes

- EXERCISE: Use `.explain("executionStats")` to see how it affects to different FIND queries
 - Without using indexed field
 - Using indexed field in a different order
 - Using indexed fields in the same order
 - Ordering the results in the same index order or not
 - Index prefixes!

Clustered Indexes

- Stored sorted by the `_id`
- Faster queries on clustered collections without needing a secondary index, such as queries with range scans and equality comparisons on the clustered index key.
- Clustered collections have a lower storage size, which improves performance for queries and bulk inserts.

Clustered Indexes

- Clustered collections can eliminate the need for a secondary TTL (Time To Live) index.
- A clustered index is also a TTL index if you specify the `expireAfterSeconds` field.
- To be used as a TTL index, the `_id` field must be a supported date type. See TTL Indexes.
- If you use a clustered index as a TTL index, it improves document delete performance and reduces the clustered collection storage size.

Clustered Indexes

- Clustered collections have additional performance improvements for inserts, updates, deletes, and queries.
- All collections have an `_id` index.
- A non-clustered collection stores the `_id` index separately from the documents. This requires two writes for inserts, updates, and deletes, and two reads for queries.
- A clustered collection stores the index and the documents together in `_id` value order. This requires one write for inserts, updates, and deletes, and one read for queries.
- **Let's see it in MongoDB Compass ...**

Clustered Indexes

- Load data
- FIND: `{ $and: [{ "_id": { $gte: 30075445 } }, { "_id": { $lte: 40361606 } }] }`

Indexes

- Multikey indexes
 - Used when indexing arrays
 - One key is created for each array value in each document
 - Only one array can be included in an index!

Indexes

- Multikey indexes
 - Used when indexing arrays
 - One key is created for each array value in each document
 - Only one array can be included in an index!
- **Exercise**
 - Score
 - Analyze the distribution
 - Create index
 - Query
 - Compare

Indexes

- Partial indexes
 - Used when your queries only use certain range of values
 - Reduces the size of the index

// insert a restaurant document

```
db.restaurants.insert({ "name" : "Han Dynasty", "cuisine" : "Sichuan", "stars" : 4.4,  
"address" : { "street" : "90 3rd Ave", "city" : "New York", "state" : "NY", "zipcode" :  
"10003" } });
```

// and run a find query on city and cuisine

```
db.restaurants.find({'address.city': 'New York', 'cuisine': 'Sichuan'})
```

// create an explainable object

```
var exp = db.restaurants.explain()
```

// and rerun the query

```
exp.find({'address.city': 'New York', 'cuisine': 'Sichuan'})
```

Indexes

- Partial indexes
 - Used when your queries only use certain range of values
 - Reduces the size of the index

// create a partial index

```
db.restaurants.createIndex( { "address.city": 1, cuisine: 1 }, {  
  partialFilterExpression: { 'stars': { $gte: 3.5 } } })
```

// rerun the query (doesn't use the partial index)

```
db.restaurants.find({'address.city': 'New York', 'cuisine': 'Sichuan'})
```

// adding the stars predicate allows us to use the partial index

```
exp.find({'address.city': 'New York', cuisine: 'Sichuan', stars: { $gt: 4.0 }})
```

Indexes

- Create Text Index

- To create a text index, use the `db.collection.createIndex()` method. To index a field that contains a string or an array of string elements, include the field and specify the string literal "text" in the index document, as in the following example:

```
db.reviews.createIndex( { comments: "text" } )
```

- A collection can have at most one text index.
- However, you can specify multiple fields for the text index.

Indexes

- Wildcard Text Indexes

- To allow for text search on all fields with string content, use the wildcard specifier (\$**) to index all fields in the collection that contain string content. Such an index can be useful with highly unstructured data if it is unclear which fields to include in the text index or for ad-hoc querying.
- With a wildcard text index, MongoDB indexes every field that contains string data for each document in the collection. The following example creates a text index using the wildcard specifier:

```
db.collection.createIndex( { "$**": "text" } )
```

Indexes

- Specify the Default Language for a text Index
 - The default language associated with the indexed data determines the rules to parse word roots (i.e. stemming) and ignore stop words. The default language for the indexed data is english.
 - To specify a different language, use the `default_language` option when creating the text index. See Text Search Languages for the languages available for `default_language`.
 - The following example creates for the quotes collection a text index on the content field and sets the `default_language` to spanish:

```
db.quotes.createIndex(  
  { content : "text" },  
  { default_language: "spanish" }  
)
```


Indexes

- Specify the Index Language within the Document
 - If a collection contains documents or embedded documents that are in different languages, include a field named language in the documents or embedded documents and specify as its value the language for that document or embedded document.
 - MongoDB will use the specified language for that document or embedded document when building the text index:
 - The specified language in the document overrides the default language for the text index.
 - The specified language in an embedded document override the language specified in an enclosing document or the default language for the index.

Indexes

- For example, a collection quotes contains multi-language documents that include the language field in the document and/or the embedded document as needed:

Indexes

```
{  
  _id: 1,  
  language: "portuguese",  
  original: "A sorte protege os audazes.",  
  translation:  
    [{  
      language: "english",  
      quote: "Fortune favors the bold."  
    }, {  
      language: "spanish",  
      quote: "La suerte protege a los audaces."  
    }]  
}
```

Indexes

```
_id: 2,  
  language: "spanish",  
  original: "Nada hay más surrealista que la realidad.",  
  translation:  
    [{  
      language: "english",  
      quote: "There is nothing more surreal than reality."  
    }, {  
      language: "french",  
      quote: "Il n'y a rien de plus surréaliste que la réalité."  
    }]  
}
```

Indexes

```
_id: 3,  
  original: "is this a dagger which I see before me.",  
  translation:  
  {  
    language: "spanish",  
    quote: "Es este un puñal que veo delante de mí."  
  }  
}
```

Indexes

- If you create a text index on the quote field with the default language of English.

```
db.quotes.createIndex( { original: "text", "translation.quote": "text" } )
```
- Then, for the documents and embedded documents that contain the language field, the text index uses that language to parse word stems and other linguistic characteristics.
- For embedded documents that do not contain the language field,
 - If the enclosing document contains the language field, then the index uses the document's language for the embedded document.
 - Otherwise, the index uses the default language for the embedded documents.
- For documents that do not contain the language field, the index uses the default language, which is English.

Indexes

- Use any Field to Specify the Language for a Document
- To use a field with a name other than language, include the language_override option when creating the index.
- For example, give the following command to use idioma as the field name instead of language:

```
db.quotes.createIndex( { quote : "text" },  
                        { language_override: "idioma" } )
```

- The documents of the quotes collection may specify a language with the idioma field:

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }  
{ _id: 2, idioma: "spanish", quote: "Nada hay más surrealista que la realidad." }  
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }
```

Indexes

- The default name for the index consists of each indexed field name concatenated with `_text`. For example, the following command creates a text index on the fields `content`, `users.comments`, and `users.profiles`:

```
db.collection.createIndex(  
  {  
    content: "text",  
    "users.comments": "text",  
    "users.profiles": "text"  
  }  
)
```

- The default name for the index is:
 - `"content_text_users.comments_text_users.profiles_text"`

Indexes

- To avoid creating an index with a name that exceeds the index name length limit, you can pass the name option to the `db.collection.createIndex()` method:

```
db.collection.createIndex(  
  {  
    content: "text",  
    "users.comments": "text",  
    "users.profiles": "text"  
  },  
  {  
    name: "MyTextIndex"  
  }  
)
```

Indexes

- Then, to remove this text index, pass the name "MyTextIndex" to the `db.collection.dropIndex()` method, as in the following:

```
db.collection.dropIndex("MyTextIndex")
```

- To get the names of the indexes, use the `db.collection.getIndexes()` method.

Index Weight

- Text search assigns a score to each document that contains the search term in the indexed fields. The score determines the relevance of a document to a given search query.
- For a text index, the weight of an indexed field denotes the significance of the field relative to the other indexed fields in terms of the text search score.
- For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results. Using this sum, MongoDB then calculates the score for the document. See `$meta` operator for details on returning and sorting by text scores.
- The default weight is 1 for the indexed fields. To adjust the weights for the indexed fields, include the `weights` option in the `db.collection.createIndex()` method.

Index Weight - Example

- A collection blog has the following documents:

```
{  
  _id: 1,  
  content: "This morning I had a cup of coffee.",  
  about: "beverage",  
  keywords: [ "coffee" ]  
}
```

```
{  
  _id: 2,  
  content: "Who doesn't like cake?",  
  about: "food",  
  keywords: [ "cake", "food", "dessert" ]  
}
```

Index Weight - Example

- To create a text index with different field weights for the content field and the keywords field, include the weights option to the createIndex() method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```
db.blog.createIndex(  
  {  
    content: "text",  
    keywords: "text",  
    about: "text"  
  },  
  {  
    weights: {  
      content: 10,  
      keywords: 5  
    },  
    name: "TextIndex"  
  }  
)
```

Index Weight - Example

- The text index has the following fields and weights:
 - content has a weight of 10,
 - keywords has a weight of 5, and
 - about has the default weight of 1.
- These weights denote the relative significance of the indexed fields to each other. For instance, a term match in the content field has:
 - 2 times (i.e. 10:5) the impact as a term match in the keywords field and
 - 10 times (i.e. 10:1) the impact as a term match in the about field.

Index – Search Depth

- A collection inventory contains the following documents:
 - { _id: 1, dept: "tech", description: "lime green computer" }
 - { _id: 2, dept: "tech", description: "wireless red mouse" }
 - { _id: 3, dept: "kitchen", description: "green placemat" }
 - { _id: 4, dept: "kitchen", description: "red peeler" }
 - { _id: 5, dept: "food", description: "green apple" }
 - { _id: 6, dept: "food", description: "red potato" }
- Consider the common use case that performs text searches by individual departments, such as:

```
db.inventory.find( { dept: "kitchen", $text: { $search: "green" } } )
```
- To limit the text search to scan only those documents within a specific dept, create a compound index that first specifies an ascending/descending index key on the field dept and then a text index key on the field description:

```
db.inventory.createIndex(  
  {  
    dept: 1,  
    description: "text"  
  }  
)
```
- Then, the text search within a particular department will limit the scan of indexed documents. For example, the following query scans only those documents with dept equal to kitchen:

```
db.inventory.find( { dept: "kitchen", $text: { $search: "green" } } )
```

Text Search

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

A string of terms that MongoDB parses and uses to query the text index. MongoDB performs a logical OR search of the terms unless specified as a phrase. To match on a phrase, as opposed to individual terms, enclose the phrase in escaped double quotes (\"), as in:
\"ssl certificate\"

If the \$search string includes a phrase and individual terms, text search will only match the documents that include the phrase. More specifically, the search performs a logical AND of the phrase with the individual terms in the search string. For example, passed a \$search string:
\"ssl certificate\" authority key

The \$text operator searches for the phrase "ssl certificate" and ("authority" or "key" or "ssl" or "certificate").

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{  
  $text:  
  {  
    $search: <string>,  
    $language: <string>,  
    $caseSensitive: <boolean>,  
    $diacriticSensitive: <boolean>  
  }  
}
```

Prefixing a word with a hyphen-minus (-) negates a word:

- The negated word excludes documents that contain the negated word from the result set.
- When passed a search string that only contains negated words, text search will not match any documents.
- A hyphenated word, such as pre-market, is not a negation. The \$text operator treats the hyphen-minus (-) as a delimiter.

The \$text operator adds all negations to the query with the logical AND operator.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{  
  $text:  
  {  
    $search: <string>,  
    $language: <string>,  
    $caseSensitive: <boolean>,  
    $diacriticSensitive: <boolean>  
  }  
}
```

Stop Words

The \$text operator ignores language-specific stop words, such as the and and in English.

Stemmed Words

For case insensitive and diacritic insensitive text searches, the \$text operator matches on the complete stemmed word. So if a document field contains the word blueberry, a search on the term blue will not match. However, blueberry or blueberries will match.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Case Sensitive Search and Stemmed Words

For case sensitive search (i.e. \$caseSensitive: true), if the suffix stem contains uppercase letters, the \$text operator matches on the exact word.

Diacritic Sensitive Search and Stemmed Words

For diacritic sensitive search (i.e. \$diacriticSensitive: true), if the suffix stem contains the diacritic mark or marks, the \$text operator matches on the exact word.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Text Score

The \$text operator assigns a score to each document that contains the search term in the indexed fields. The score represents the relevance of a document to a given text search query. The score can be part of a sort() method specification as well as part of the projection expression. The { \$meta: "textScore" } expression provides information on the processing of the \$text operation. See \$meta projection operator for details on accessing the score for projection or sort.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Optional.

The language that determines the list of stop words for the search and the rules for the stemmer and tokenizer. If not specified, the search uses the default language of the index. For supported languages, see [Text Search Languages](#).

If you specify a language value of "none", then the text search uses simple tokenization with no list of stop words and no stemming.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Optional. A boolean flag to enable or disable case sensitive search. Defaults to false; i.e. the search defers to the case insensitivity of the text index.

Text Search

- \$text performs a text search on the content of the fields indexed with a text index. A \$text expression has the following syntax:

```
{
  $text:
  {
    $search: <string>,
    $language: <string>,
    $caseSensitive: <boolean>,
    $diacriticSensitive: <boolean>
  }
}
```

Optional. A boolean flag to enable or disable diacritic sensitive search against version 3 text indexes. Defaults to false; i.e. the search defers to the diacritic insensitivity of the text index.

Text searches against earlier versions of the text index are inherently diacritic sensitive and cannot be diacritic insensitive. As such, the \$diacriticSensitive option has no effect with earlier versions of the text index.

Text Search - Example

```
db.articles.createIndex( { subject: "text" } )
```

```
db.articles.insert(
```

```
[
```

```
  { _id: 1, subject: "coffee", author: "xyz", views: 50 },
```

```
  { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
```

```
  { _id: 3, subject: "Baking a cake", author: "abc", views: 90 },
```

```
  { _id: 4, subject: "baking", author: "xyz", views: 100 },
```

```
  { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
```

```
  { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
```

```
  { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
```

```
  { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }
```

```
]
```

```
)
```

Text Search - Example

Search for a Single Word

The following query specifies a \$search string of coffee:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

This query returns the documents that contain the term coffee in the indexed subject field, or more precisely, the stemmed version of the word:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }  
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }  
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

Text Search - Example

Match Any of the Search Terms

If the search string is a space-delimited string, `$text` operator performs a logical OR search on each term and returns documents that contains any of the terms. The following query specifies a `$search` string of three terms delimited by space, "bake coffee cake":

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

This query returns documents that contain either bake or coffee or cake in the indexed subject field, or more precisely, the stemmed version of these words:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }  
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }  
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }  
{ "_id" : 3, "subject" : "Baking a cake", "author" : "abc", "views" : 90 }  
{ "_id" : 4, "subject" : "baking", "author" : "xyz", "views" : 100 }
```

Text Search - Example

Search for a Phrase

To match the exact phrase as a single term, escape the quotes. The following query searches for the phrase coffee shop:

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

This query returns documents that contain the phrase coffee shop:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

Text Search - Example

Exclude Documents That Contain a Term

A negated term is a term that is prefixed by a minus sign -. If you negate a term, the \$text operator will exclude the documents that contain those terms from the results. The following example searches for documents that contain the words coffee but do not contain the term shop, or more precisely the stemmed version of the words:

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

The query returns the following documents:

```
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }  
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

Text Search - Example

Search a Different Language

Use the optional `$language` field in the `$text` expression to specify a language that determines the list of stop words and the rules for the stemmer and tokenizer for the search string. If you specify a language value of "none", then the text search uses simple tokenization with no list of stop words and no stemming. The following query specifies es, i.e. Spanish, as the language that determines the tokenization, stemming, and stop words. The `$text` expression can also accept the language by name, spanish:

```
db.articles.find(  
  { $text: { $search: "leche", $language: "es" } }  
)
```

The query returns the following documents:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }  
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz", "views" : 10 }
```

Text Search - Example

Case and Diacritic Insensitive Search

The `$text` operator defers to the case and diacritic insensitivity of the text index. The version 3 text index is diacritic insensitive and expands its case insensitivity to include the Cyrillic alphabet as well as characters with diacritics. For details, see [text Index Case Insensitivity](#) and [text Index Diacritic Insensitivity](#). The following query performs a case and diacritic insensitive text search for the terms `сырники` or `CAFÉS`:

```
db.articles.find( { $text: { $search: "сырники CAFÉS" } } )
```

Using the version 3 text index, the query matches the following documents.

```
{ "_id" : 6, "subject" : "Сырники", "author" : "jkl", "views" : 80 }  
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }  
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz", "views" : 10 }
```

With the previous versions of the text index, the query would not match any document.

Text Search - Example

Perform Case Sensitive Search

To enable case sensitive search, specify `$caseSensitive: true`. Specifying `$caseSensitive: true` may impact performance. The following query performs a case sensitive search for the term `Coffee`:

```
db.articles.find( { $text: { $search: "Coffee", $caseSensitive: true } } )
```

The search matches just the document:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

Text Search - Example

Case Sensitive Search for a Phrase

The following query performs a case sensitive search for the phrase Café Con Leche:

```
db.articles.find( {  
  $text: { $search: "\"Café Con Leche\"", $caseSensitive: true }  
})
```

The search matches just the document:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
```

Text Search - Example

Case Sensitivity with Negated Term

A negated term is a term that is prefixed by a minus sign -. If you negate a term, the \$text operator will exclude the documents that contain those terms from the results. You can also specify case sensitivity for negated terms. The following example performs a case sensitive search for documents that contain the word Coffee but do not contain the lower-case term shop, or more precisely the stemmed version of the words:

```
db.articles.find( { $text: { $search: "Coffee -shop", $caseSensitive: true } } )
```

The query matches the following document:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg" }
```

Text Search - Example

Return the Text Search Score

The following query searches for the term cake and returns the score assigned to each matching document:

```
db.articles.find(  
  { $text: { $search: "cake" } },  
  { score: { $meta: "textScore" } }  
)
```

The returned document includes an additional field score that contains the document's score associated with the text search.

Text Search - Example

Sort by Text Search Score

To sort by the text score, include the same `$meta` expression in both the projection document and the sort expression. [1] The following query searches for the term `coffee` and sorts the results by the descending score:

```
db.articles.find(  
  { $text: { $search: "coffee" } },  
  { score: { $meta: "textScore" } }  
) .sort( { score: { $meta: "textScore" } } )
```

The query returns the matching documents sorted by descending score.

Text Search - Example

Return Top 2 Matching Documents

Use the `limit()` method in conjunction with a `sort()` to return the top n matching documents. The following query searches for the term `coffee` and sorts the results by the descending score, limiting the results to the top two matching documents:

```
db.articles.find(  
  { $text: { $search: "coffee" } },  
  { score: { $meta: "textScore" } }  
) .sort( { score: { $meta: "textScore" } } ).limit(2)
```

Text Search - Example

Text Search with Additional Query and Sort Expressions

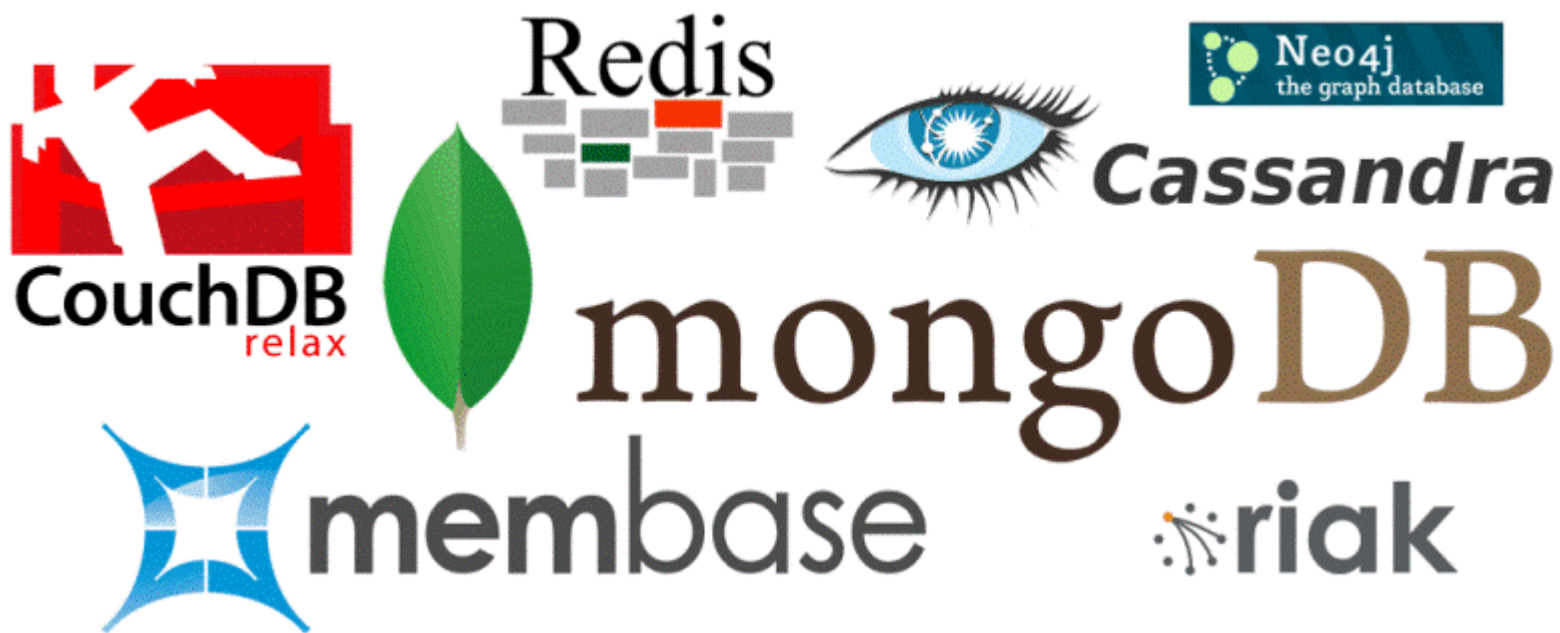
The following query searches for documents where the author equals "xyz" and the indexed field subject contains the terms coffee or bake. The operation also specifies a sort order of ascending `_id`, then descending text search score:

```
db.articles.find(  
  { author: "xyz", $text: { $search: "coffee bake" } },  
  { score: { $meta: "textScore" } }  
) .sort( { date: 1, score: { $meta: "textScore" } } )
```

Table of contents

- Spatial Indexes
- Spatial Queries

Introduction



Spatial Indexes

2dsphere Indexes

- A 2dsphere index supports queries that calculate geometries on an earth-like sphere. 2dsphere index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity.
- The 2dsphere index supports data stored as GeoJSON objects and as legacy coordinate pairs. For legacy coordinate pairs, the index converts the data to GeoJSON Point.
- MongoDB 2.6 introduces a version 2 of 2dsphere indexes. Version 2 is the default version of 2dsphere indexes created in MongoDB 2.6 and later series. To override the default version 2 and create a version 1 index, include the option { "2dsphereIndexVersion": 1 } when creating the index.

2dsphere Indexes

- 2dsphere (Version 2) indexes are sparse by default and ignores the `sparse: true` option. If a document lacks a 2dsphere index field (or the field is null or an empty array), MongoDB does not add an entry for the document to the index. For inserts, MongoDB inserts the document but does not add to the 2dsphere index.
- For a compound index that includes a 2dsphere index key along with keys of other types, only the 2dsphere index field determines whether the index references a document.
- Earlier versions of MongoDB only support 2dsphere (Version 1) indexes. 2dsphere (Version 1) indexes are not sparse by default and will reject documents with null location fields.

GeoJSON Support:

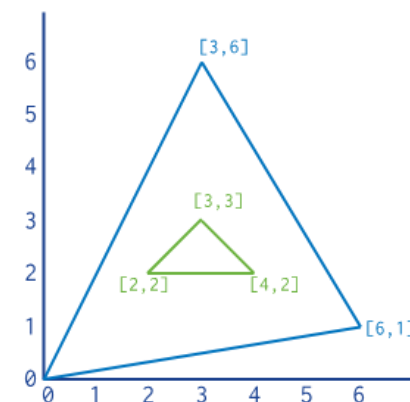
- Point: { **type**: "**Point**", **coordinates**: [40, 5] }
- LineString: { **type**: "**LineString**", **coordinates**: [[40, 5], [41, 6]] }
- Polygon:

- Polygons with a Single Ring

```
{
  type: "Polygon",
  coordinates: [ [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ] ]
}
```

- Polygons with Multiple Rings

```
{
  type : "Polygon",
  coordinates : [
    [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ],
    [ [ 2, 2 ], [ 3, 3 ], [ 4, 2 ], [ 2, 2 ] ]
  ]
}
```



GeoJSON Support:

- MultiPoint:

```
{  
  type: "MultiPoint",  
  coordinates: [  
    [ -73.9580, 40.8003 ],  
    [ -73.9498, 40.7968 ],  
    [ -73.9737, 40.7648 ],  
    [ -73.9814, 40.7681 ]  
  ]  
}
```

GeoJSON Support:

- MultiLineString:

```
{  
  type: "MultiLineString",  
  coordinates: [  
    [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],  
    [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],  
    [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],  
    [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]  
  ]  
}
```

GeoJSON Support:

- MultiPolygon:

```
{  
  type: "MultiPolygon",  
  coordinates: [  
    [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -  
73.9814, 40.7681 ], [ -73.958, 40.8003 ] ] ],  
    [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -  
73.958, 40.8003 ] ] ]  
  ]  
}
```


GeoJSON Support:

- GeometryCollection:

```
{
  type: "GeometryCollection",
  geometries: [
    {
      type: "MultiPoint",
      coordinates: [
        [ -73.9580, 40.8003 ],
        [ -73.9498, 40.7968 ],
        [ -73.9737, 40.7648 ],
        [ -73.9814, 40.7681 ]
      ]
    },
    {
      type: "MultiLineString",
      coordinates: [
        [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],
        [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],
        [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],
        [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]
      ]
    }
  ]
}
```

2dsphere Indexes

- 2dsphere (Version 2) includes support for additional GeoJSON object: MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection.
- geoNear and \$geoNear Restrictions
 - The geoNear command and the \$geoNear pipeline stage require that a collection have at most only one 2dsphere index and/or only one 2d index whereas geospatial query operators (e.g. \$near and \$geoWithin) permit collections to have multiple geospatial indexes.
 - The geospatial index restriction for the geoNear command and the \$geoNear pipeline stage exists because neither the geoNear command nor the \$geoNear pipeline stage syntax includes the location field. As such, index selection among multiple 2d indexes or 2dsphere indexes is ambiguous.
 - No such restriction applies for geospatial query operators since these operators take a location field, eliminating the ambiguity.

2dsphere Indexes

- 2dsphere Indexed Field Restrictions
 - Fields with 2dsphere indexes must hold geometry data in the form of coordinate pairs or GeoJSON data. If you attempt to insert a document with non-geometry data in a 2dsphere indexed field, or build a 2dsphere index on a collection where the indexed field has non-geometry data, the operation will fail.

2dsphere Indexes

- Create a 2dsphere Index

- To create a 2dsphere index, use the `db.collection.createIndex()` method, specifying the location field as the key and the string literal "2dsphere" as the index type:

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

- Unlike a compound 2d index which can reference one location field and one other field, a compound 2dsphere index can reference multiple location and non-location fields.

2dsphere Indexes

- For the following examples, consider a collection places with documents that store location data as GeoJSON Point in a field named loc:

```
db.places.insert(  
  {  
    loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },  
    name: "Central Park",  
    category : "Parks"  
  }  
)
```

```
db.places.insert(  
  {  
    loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },  
    name: "La Guardia Airport",  
    category : "Airport"  
  }  
)
```

2dsphere Indexes

- Create a 2dsphere Index

- The following operation creates a 2dsphere index on the location field loc:

```
db.places.createIndex( { loc : "2dsphere" } )
```

- Create a Compound Index with 2dsphere Index Key

- A compound index can include a 2dsphere index key in combination with non-geospatial index keys. For example, the following operation creates a compound index where the first key loc is a 2dsphere index key, and the remaining keys category and names are non-geospatial index keys, specifically descending (-1) and ascending (1) keys respectively.

```
db.places.createIndex( { loc : "2dsphere" , category : -1, name: 1 } )
```

- Unlike the 2d index, a compound 2dsphere index does not require the location field to be the first field indexed. For example:

```
db.places.createIndex( { category : 1 , loc : "2dsphere" } )
```

2dsphere Indexes

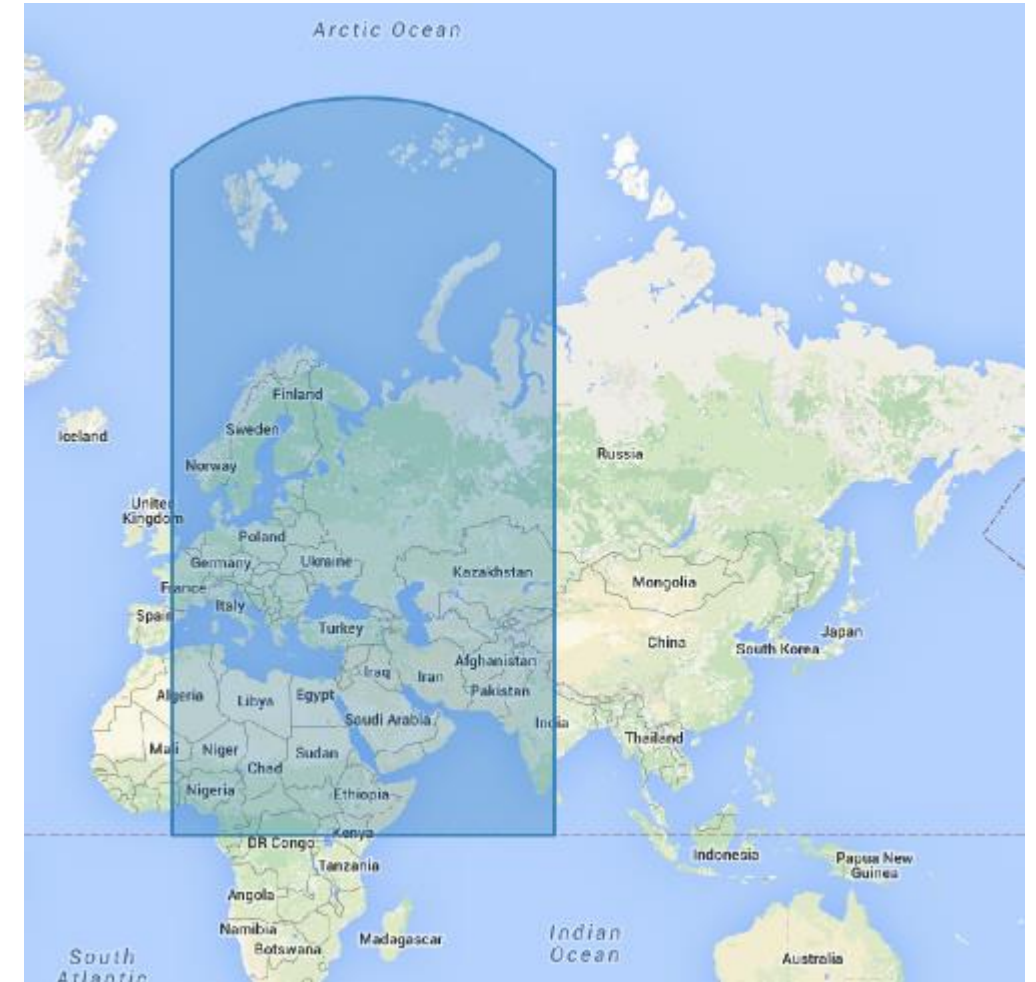
- Geospatial queries can use either flat or spherical geometries, depending on both the query and the type of index in use. 2dsphere indexes support only spherical geometries, while 2d indexes support both flat and spherical geometries.
- However, queries using spherical geometries will be more performant and accurate with a 2dsphere index, so you should always use 2dsphere indexes on geographical geospatial fields.

2dsphere Indexes

- The following table shows what kind of geometry each geospatial operator will use:

Query Type	Geometry Type	Notes
<code>\$near</code> (GeoJSON point, 2dsphere index)	Spherical	
<code>\$near</code> (legacy coordinates, 2d index)	Flat	
<code>\$nearSphere</code> (GeoJSON point, 2dsphere index)	Spherical	
<code>\$nearSphere</code> (legacy coordinates, 2d index)	Spherical	Use GeoJSON points instead.
<code>\$geoWithin:{ \$geometry: ... }</code>	Spherical	
<code>\$geoWithin:{ \$box: ... }</code>	Flat	
<code>\$geoWithin:{ \$polygon: ... }</code>	Flat	
<code>\$geoWithin:{ \$center: ... }</code>	Flat	
<code>\$geoWithin:{ \$centerSphere: ... }</code>	Spherical	
<code>\$geoIntersects</code>	Spherical	

- Spherical geometry will appear distorted when visualized on a map due to the nature of projecting a three dimensional sphere, such as the earth, onto a flat plane.
- For example, take the specification of the spherical square defined by the longitude latitude points $(0,0)$, $(80,0)$, $(80,80)$, and $(0,80)$. The following figure depicts the area covered by this region:



2dsphere Indexes - Search

- Download the example datasets from <https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/neighborhoods.json> and <https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/restaurants.json>. These contain the collections restaurants and neighborhoods respectively.

- After downloading the datasets, import them into the database:

```
mongoimport <path to restaurants.json> -c restaurants  
mongoimport <path to neighborhoods.json> -c neighborhoods
```

Exercises

- Suppose you are designing a mobile application to help users find restaurants in New York City. The application must:
- Determine **the user's current neighborhood** using \$geoIntersects,
- Show the number of **restaurants in that neighborhood** using \$geoWithin, and
- Find restaurants **within a specified distance of the user** using \$nearSphere.

```
{  
  $nearSphere: {  
    $geometry: {  
      type : "Point",  
      coordinates : [ <longitude>, <latitude> ]  
    },  
    $minDistance: <distance in meters>,  
    $maxDistance: <distance in meters>  
  }  
}
```

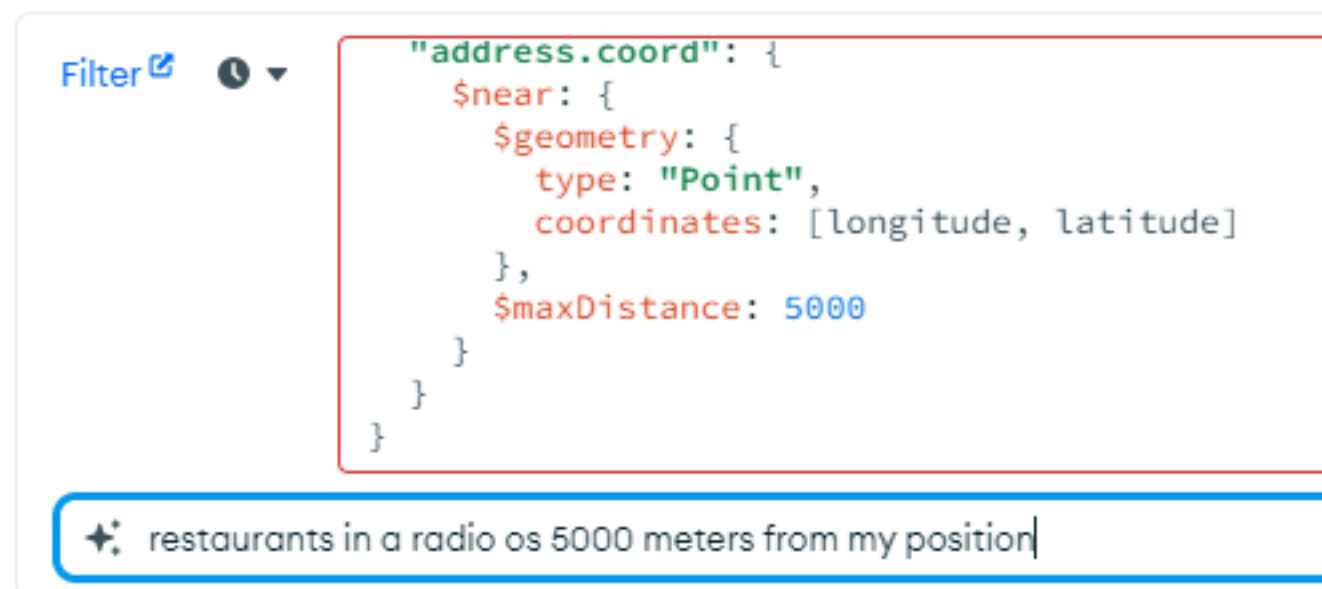
Mongo Compass

Find restaurants **within a specified distance of the user** using \$nearSphere.

```
{ address.coord:
  { $nearSphere:
    { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ]
  },
  $maxDistance: 5000
}
}
```

operator sorts documents by distance

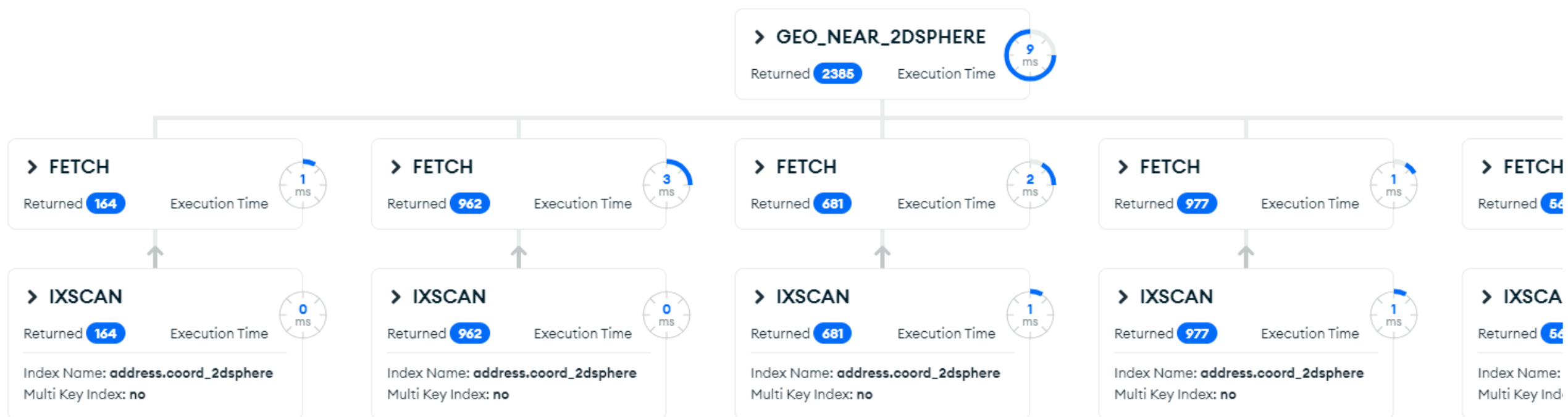
The \$nearSphere operator is the spherical equivalent of the \$near operator.





Find restaurants **within a specified distance of the user** using \$nearSphere.

Mongo Compass



Find restaurants **within a specified distance of the user** using \$nearSphere.

Mongo Compass

```
{ address.coord:  
  { $geoWithin:  
    { $centerSphere: [ [ -73.93414657, 40.82302903 ], 5 / 6378.1 ] } } }
```

To use [\\$centerSphere](#), specify an array that contains:

- The grid coordinates of the circle's center point, and
- The circle's radius measured in radians. To calculate radians, see [Convert Distance to Radians for Spherical Operators](#).

2dsphere Indexes - Search

- The geoNear command requires a geospatial index, and almost always improves performance of \$geoWithin and \$geoIntersects queries.
- Because this data is geographical, create a 2dsphere index on each collection using the mongo shell:

```
db.restaurants.createIndex({ location: "2dsphere" })
```

```
db.neighborhoods.createIndex({ geometry: "2dsphere" })
```


2dsphere Indexes - Search

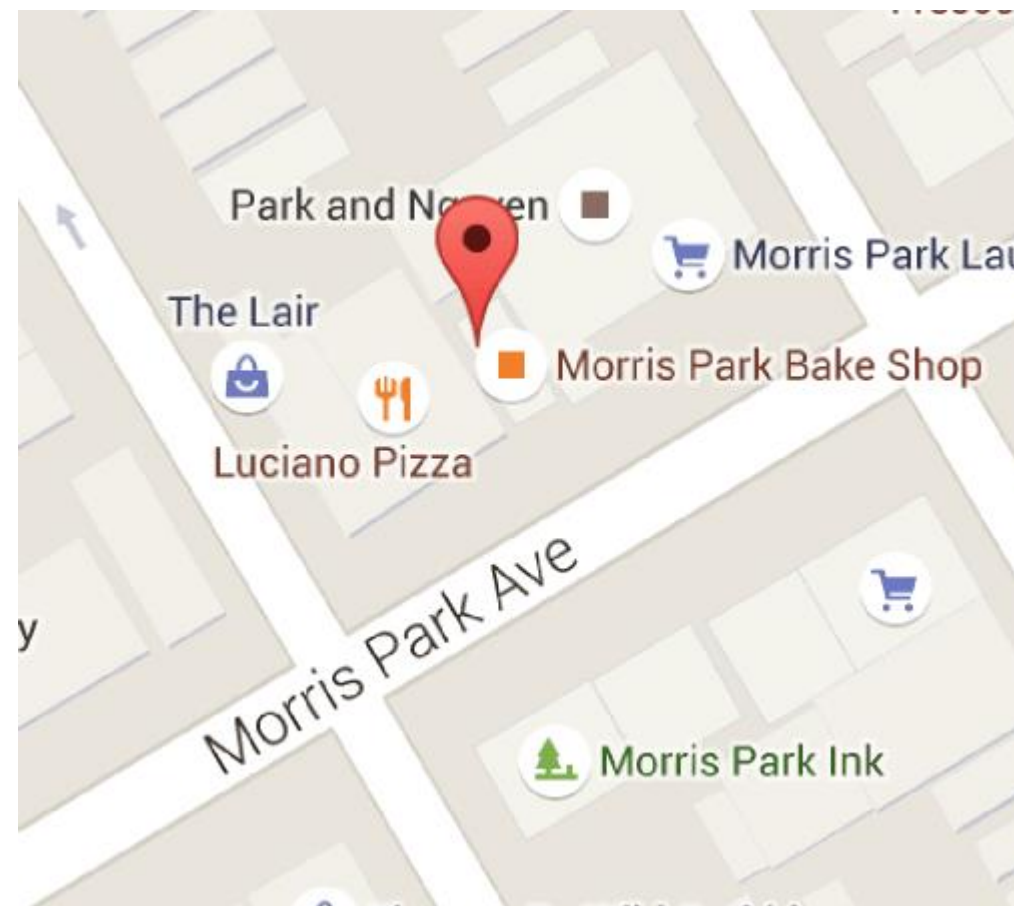
- Inspect an entry in the newly-created restaurants collection from within the mongo shell:

```
db.restaurants.findOne()
```

- This query returns a document like the following:

```
{  
  location: {  
    type: "Point",  
    coordinates: [-73.856077, 40.848447]  
  },  
  name: "Morris Park Bake Shop"  
}
```

- This restaurant document corresponds to the location shown in the following figure:

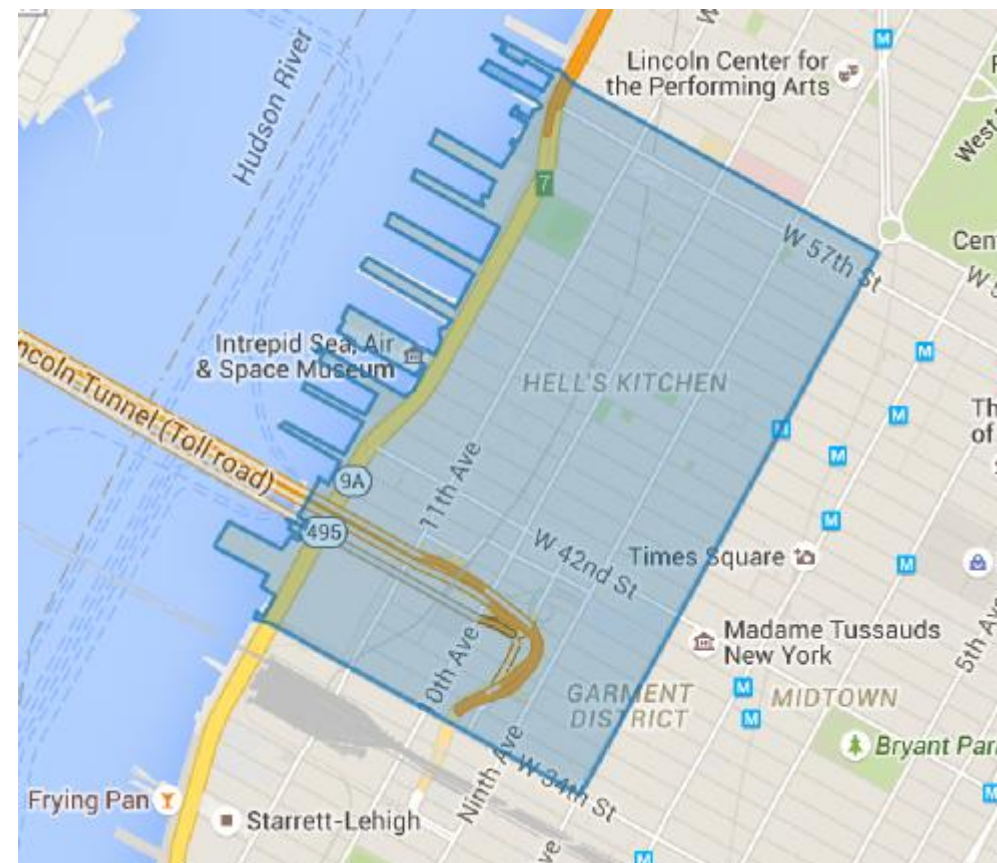


2dsphere Indexes - Search

- Now inspect an entry in the neighborhoods collection:
`db.neighborhoods.findOne()`

- This query will return a document like the following:

```
{  
  geometry: {  
    type: "Polygon",  
    coordinates: [[  
      [ -73.99, 40.75 ],  
      ...  
      [ -73.98, 40.76 ],  
      [ -73.99, 40.75 ]  
    ]]  
  },  
  name: "Hell's Kitchen"  
}
```



- This geometry corresponds to the region depicted in the following figure:

2dsphere Indexes - Search

- Assuming the user's mobile device can give a reasonably accurate location for the user, it is simple to find the user's current neighborhood with `$geoIntersects`. Suppose the user is located at -73.93414657 longitude and 40.82302903 latitude. To find the current neighborhood, you will specify a point using the special `$geometry` field in GeoJSON format:

```
db.neighborhoods.findOne({ geometry: { $geoIntersects: { $geometry: { type: "Point", coordinates: [ -73.93414657, 40.82302903 ] } } } })
```

- This query will return the following result:

```
{
  "_id" : ObjectId("55cb9c666c522cafdb053a68"),
  "geometry" : {
    "type" : "Polygon",
    "coordinates" : [
      [
        [
          -73.93383000695911,
          40.81949109558767
        ],
        ...
      ]
    ]
  },
  "name" : "Central Harlem North-Polo Grounds"
}
```



Mongo Compass

Suppose you are designing a mobile application to help users find restaurants in New York City. The application must:

Determine **the user's current neighborhood** using `$geoIntersects`,

Show the number of **restaurants in that neighborhood** using `$geoWithin`, and

Find restaurants **within a specified distance of the user** using `$nearSphere`.



Mongo Compass

Determine **the user's current neighborhood** using
\$geoIntersects,

<https://raw.githubusercontent.com/mongodb/docs-assets/geospatial/neighborhoods.json>

AGAPA.vecindarios

Documents

Aggregations

Schema

Indexes

Validation

Filter



Type a query: { field: 'value' } or [Generate query](#)

Explain

Re

ADD DATA

EXPORT DATA

0 - 0 of 0



This collection has no data

It only takes a few seconds to import data from a JSON or CSV file.

Import Data

Determine **the user's current neighborhood** using
\$geoIntersects,

Mongo Compass

 ADD DATA ▾  EXPORT DATA ▾

1 – 20 of 195 

```
_id: ObjectId('55cb9c666c522cafdb053a1a')
▼ geometry: Object
  ► coordinates: Array (1)
    type: "Polygon"
  name: "Bedford"
```

```
_id: ObjectId('55cb9c666c522cafdb053a1b')
► geometry: Object
  name: "Midwood"
```

Determine **the user's current neighborhood** using
\$geoIntersects,

```
{ geometry:
  { $geoIntersects:
    { $geometry:
      { type: "Point",
        coordinates: [ -73.93414657, 40.82302903 ]
      }
    }
  }
}
```

```
_id: ObjectId('55cb9c666c522cafdb053a68')
▶ geometry: Object
  name: "Central Harlem North-Polo Grounds"
```

Filter   {"geometry": {"\$geoIntersects": {"\$geometry": {"type": "Point", "coordinates": [-73.93414657, 40.82302903]}}}}

✦ Vecindario en el que estaría si estuviera en las coordenadas -73.93414657, 40.82302903 |


Mongo Compass



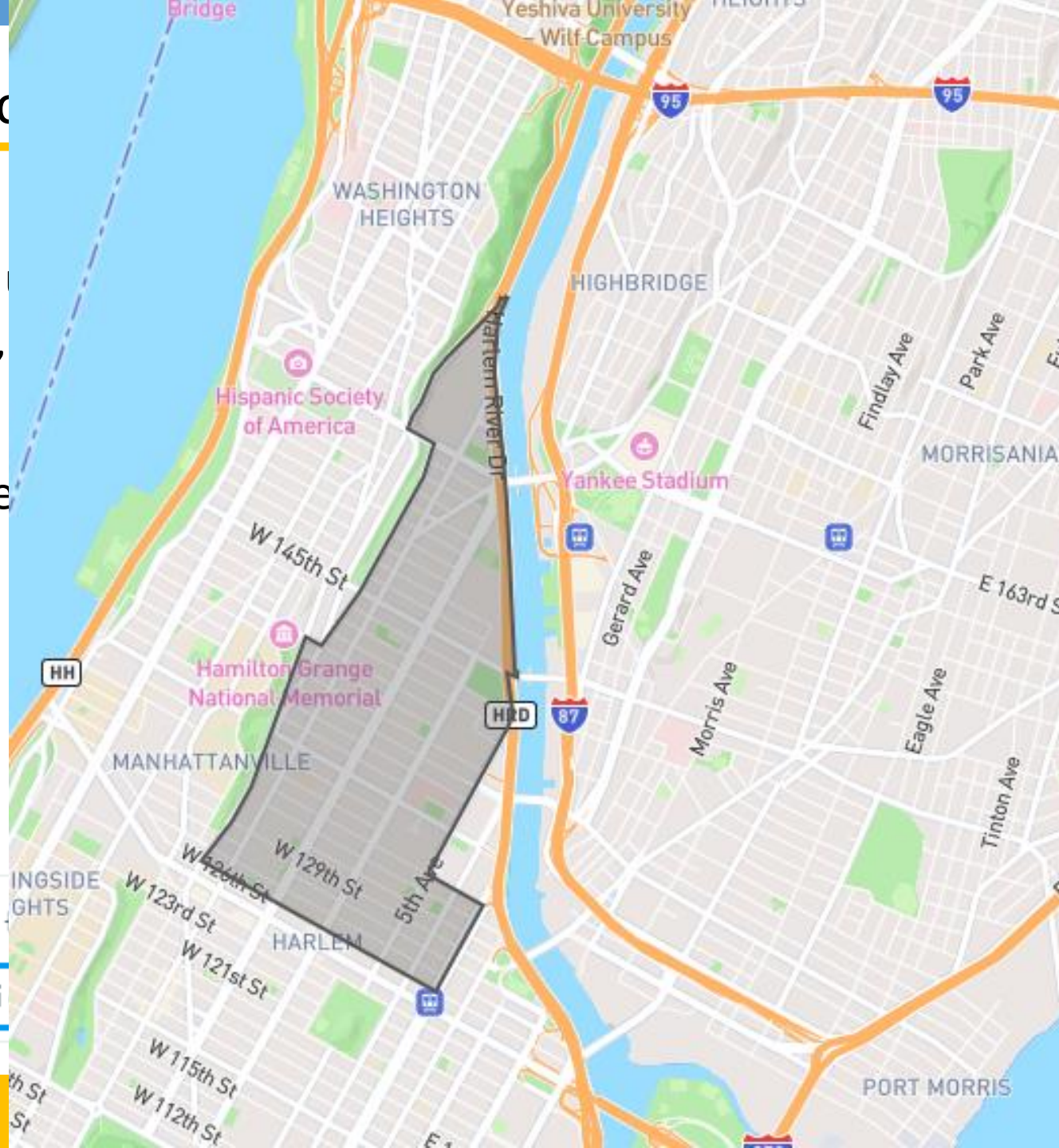
Determine the
\$geoIntersects,

```
{ geometry:  
  { $geo
```

1

Filter   {"geometry":

★ Vecindario en el que estaría si



Mongo Compass

21

55cb9c666c522cafdb053a68')

Harlem North-Polo Grounds"

3.93414657, 40.82302903]]}}}}

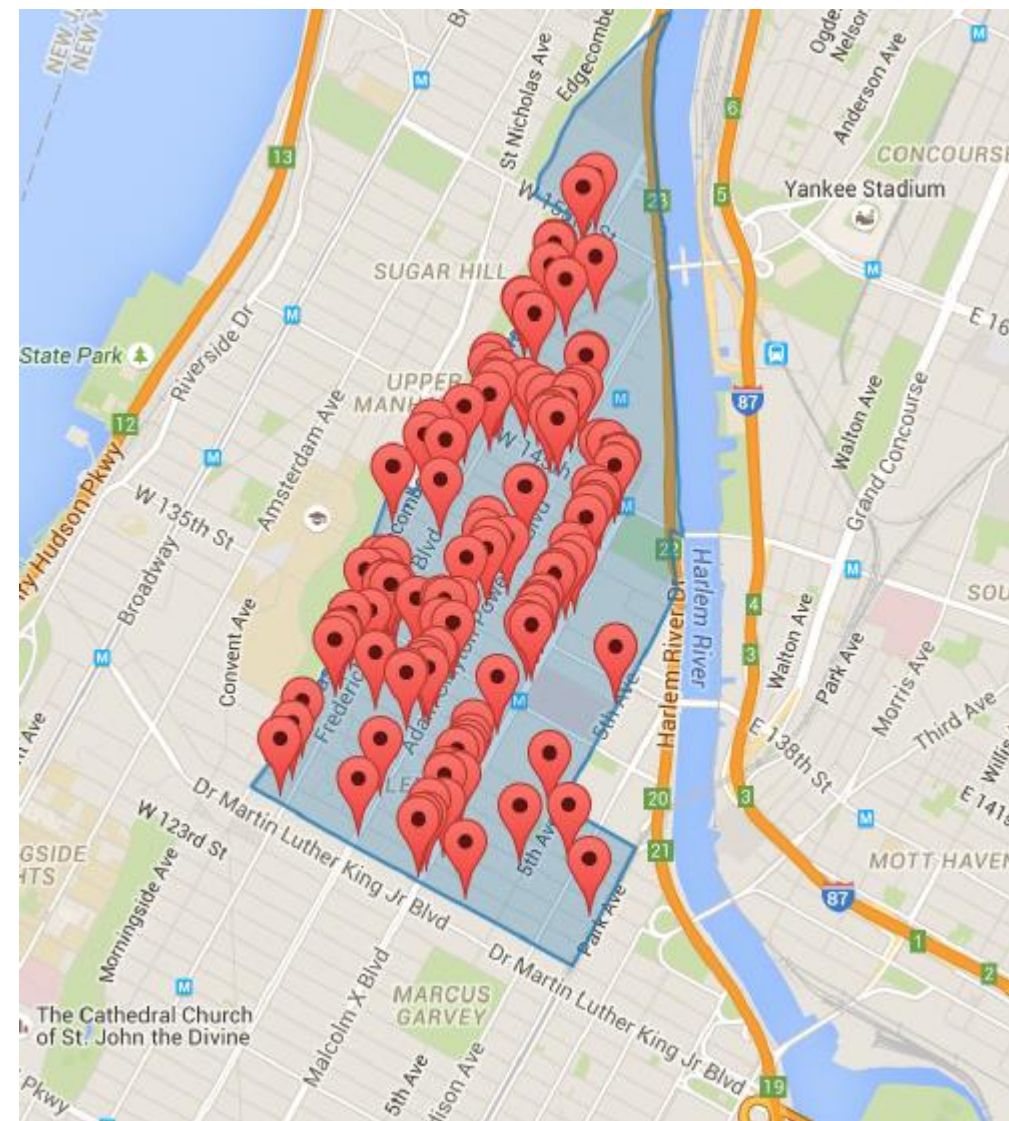
2dsphere Indexes - Search

- You can also query to find all restaurants contained in a given neighborhood. Run the following in the mongo shell to find the neighborhood containing the user, and then count the restaurants within that neighborhood:

```
var neighborhood = db.neighborhoods.findOne( {  
  geometry: { $geoIntersects: { $geometry: { type: "Point",  
    coordinates: [ -73.93414657, 40.82302903 ] } } } }
```

```
db.restaurants.find( { location: { $geoWithin: {  
  $geometry: neighborhood.geometry } } } ).count()
```

- This query will tell you that there are 127 restaurants in the requested neighborhood, visualized in the following figure:



2dsphere Indexes - Search

- To find restaurants within a specified distance of a point, you can use either `$geoWithin` with `$centerSphere` to return results in unsorted order, or `nearSphere` with `$maxDistance` if you need results sorted by distance.
 - Unsorted with `$geoWithin`
 - To find restaurants within a circular region, use `$geoWithin` with `$centerSphere`. `$centerSphere` is a MongoDB-specific syntax to denote a circular region by specifying the center and the radius in radians.
 - `$geoWithin` does not return the documents in any specific order, so it may show the user the furthest documents first.
 - The following will find all restaurants within five miles of the user:

```
db.restaurants.find({ location:
  { $geoWithin:
    { $centerSphere: [ [ -73.93414657, 40.82302903 ], 5 / 3963.2 ] } } })
```
- `$centerSphere`'s second argument accepts the radius in radians, so you must divide it by the radius of the earth in miles. See [Calculate Distance Using Spherical Geometry](#) for more information on converting between distance units.

2dsphere Indexes - Search

- Sorted with \$nearSphere
- You may also use \$nearSphere and specify a \$maxDistance term in meters. This will return all restaurants within five miles of the user in sorted order from nearest to farthest:

```
var METERS_PER_MILE = 1609.34
```

```
db.restaurants.find({ location: { $nearSphere: { $geometry: { type: "Point", coordinates: [ -73.93414657,  
40.82302903 ] }, $maxDistance: 5 * METERS_PER_MILE } } })
```

2dsphere Indexes - Search

- The \$geoWithin operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

```
db.<collection>.find( { <location field> :  
    { $geoWithin :  
      { $geometry :  
        { type : "Polygon" ,  
          coordinates : [ <coordinates> ]  
        }  
      }  
    }  
  }  
})
```

- The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :  
    { $geoWithin :  
      { $geometry :  
        { type : "Polygon" ,  
          coordinates : [ [  
              [ 0, 0 ],  
              [ 3, 6 ],  
              [ 6, 1 ],  
              [ 0, 0 ]  
            ]  
          }  
        }  
      }  
    }  
  }  
})
```

2dsphere Indexes - Search

- The \$geoIntersects operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty. This includes documents that have a shared edge. The \$geoIntersects operator uses the following syntax:

```
db.<collection>.find( { <location field> :  
  { $geoIntersects :  
    { $geometry :  
      { type : "<GeoJSON object type>" ,  
        coordinates : [ <coordinates> ]  
      }  
    }  
  }  
})
```

- The following example uses \$geoIntersects to select all indexed points and shapes that intersect with the polygon defined by the coordinates array.

```
db.places.find( { loc :  
  { $geoIntersects :  
    { $geometry :  
      { type : "Polygon" ,  
        coordinates: [ [  
          [ 0 , 0 ],  
          [ 3 , 6 ],  
          [ 6 , 1 ],  
          [ 0 , 0 ]  
        ]  
      }  
    }  
  }  
})
```

2dsphere Indexes - Search

- Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a 2dsphere index. To query for proximity to a GeoJSON point, use either the \$near operator or geoNear command. Distance is in meters. The \$near uses the following syntax:

```
db.<collection>.find( { <location field> :  
    { $near :  
        { $geometry :  
            { type : "Point" ,  
              coordinates : [ <longitude> , <latitude> ] } ,  
          $maxDistance : <distance in meters>  
        }  
    }  
})
```
- The geoNear command uses the following syntax:

```
db.runCommand( { geoNear : <collection> ,  
    near : { type : "Point" ,  
            coordinates : [ <longitude> , <latitude> ] } ,  
    spherical : true } )
```
- The geoNear command offers more options and returns more information than does the \$near operator. To run the command, see geoNear.

2dsphere Indexes - Search

- To select all grid coordinates in a “spherical cap” on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:
 - The grid coordinates of the circle’s center point
 - The circle’s radius measured in radians. To calculate radians, see [Calculate Distance Using Spherical Geometry](#).
- Use the following syntax:

```
db.<collection>.find( { <location field> :  
    { $geoWithin :  
      { $centerSphere :  
        [ [ <x>, <y> ], <radius> ] }  
    } })
```
- The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude 88 W and latitude 30 N. The example converts the distance, 10 miles, to radians by dividing by the approximate equatorial radius of the earth, 3963.2 miles:

```
db.places.find( { loc :  
    { $geoWithin :  
      { $centerSphere :  
        [ [ -88 , 30 ] , 10 / 3963.2 ]  
      }  
    } })
```

2dsphere Indexes - Search

- <https://docs.mongodb.com/v3.2/reference/geojson/>
- <https://docs.mongodb.com/v3.2/reference/operator/query-geospatial/>

Other issues

- <https://docs.mongodb.com/v3.2/core/write-operations-atomicity/>
- <https://docs.mongodb.com/v3.2/core/read-isolation-consistency-recency/>
- <https://docs.mongodb.com/v3.2/core/distributed-queries/>
- <https://docs.mongodb.com/v3.2/core/distributed-write-operations/>
- <https://docs.mongodb.com/v3.2/tutorial/perform-two-phase-commits/>
- <https://docs.mongodb.com/v3.2/tutorial/perform-findAndModify-quorum-reads/>
- <https://docs.mongodb.com/v3.2/core/query-plans/>
- <https://docs.mongodb.com/v3.2/core/query-optimization/>
- <https://docs.mongodb.com/v3.2/tutorial/evaluate-operation-performance/>
- <https://docs.mongodb.com/v3.2/tutorial/optimize-query-performance-with-indexes-and-projections/>
- <https://docs.mongodb.com/v3.2/core/write-performance/>
- <https://docs.mongodb.com/v3.2/reference/explain-results/>
- <https://docs.mongodb.com/v3.2/tutorial/analyze-query-plan/>
- <https://docs.mongodb.com/v3.2/core/tailable-cursors/>
- <https://docs.mongodb.com/v3.2/data-modeling/>
- <https://docs.mongodb.com/v3.2/core/index-hashed/>
- <https://docs.mongodb.com/v3.2/core/index-properties/>
- <https://docs.mongodb.com/v3.2/tutorial/manage-indexes/>
- <https://docs.mongodb.com/v3.2/tutorial/measure-index-use/>
- <https://docs.mongodb.com/v3.2/applications/indexes/>
- <https://docs.mongodb.com/v3.2/tutorial/create-indexes-to-support-queries/>
- <https://docs.mongodb.com/v3.2/tutorial/sort-results-with-indexes/>
- <https://docs.mongodb.com/v3.2/tutorial/ensure-indexes-fit-ram/>
- <https://docs.mongodb.com/v3.2/tutorial/create-queries-that-ensure-selectivity/>
- <https://docs.mongodb.com/v3.2/reference/indexes/>

SQL – MongoDB Comparison

- <https://docs.mongodb.org/manual/reference/sql-comparison/>