MDB200

Indexes and Optimization I

MongoDB Optimization and Performance

Release: 20250221

Topics we cover

How do Indexes work

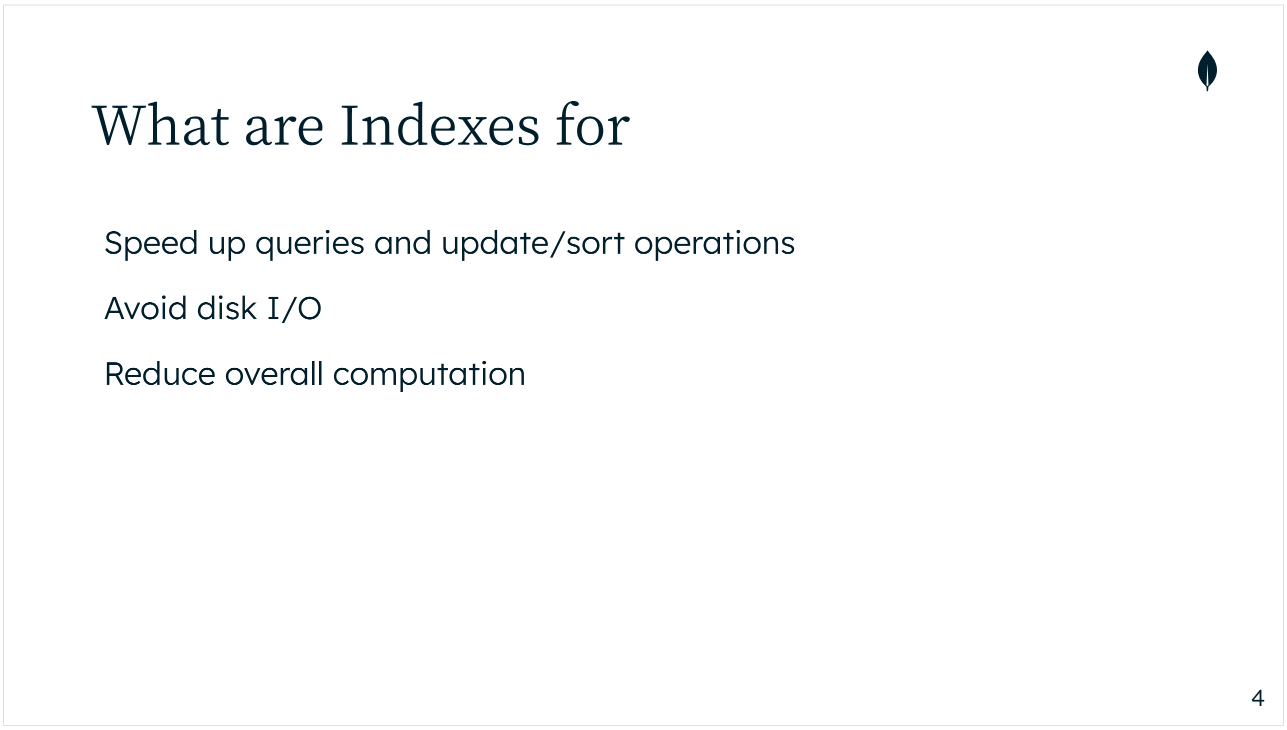
When to use Indexes

How MongoDB uses them

Index types and Options

Performance and limitations

Compression



What are Indexes for

Speed up queries and update/sort operations

Avoid disk I/O

Reduce overall computation

The aim is to reduce the resources required for an operation.
Indexes enable us to dramatically change the resources we use, i.e., memory or disk.
The disk should be avoided where possible as it is slower than the memory.
Indexes are sorted so no extra computation required when querying results in sorted order.

Index misconceptions

Misconception	Reality
MongoDB is so fast that it doesn't need indexes	
Every field is automatically indexed	
NoSQL uses hashes, not indexes	

Index misconceptions

Misconception	Reality
MongoDB is so fast that it doesn't need indexes	Incorrect or missing indexes are the main cause of performance issues
Every field is automatically indexed	Indexes must be manually created. The choice and order of the indexed fields is essential for performance
NoSQL uses hashes, not indexes	MongoDB stores indexes in a B-Tree data structure with index keys in sorted order

How do Indexes work

Data in an index is ordered so it can be searched efficiently.

Values in an index point to document identity.

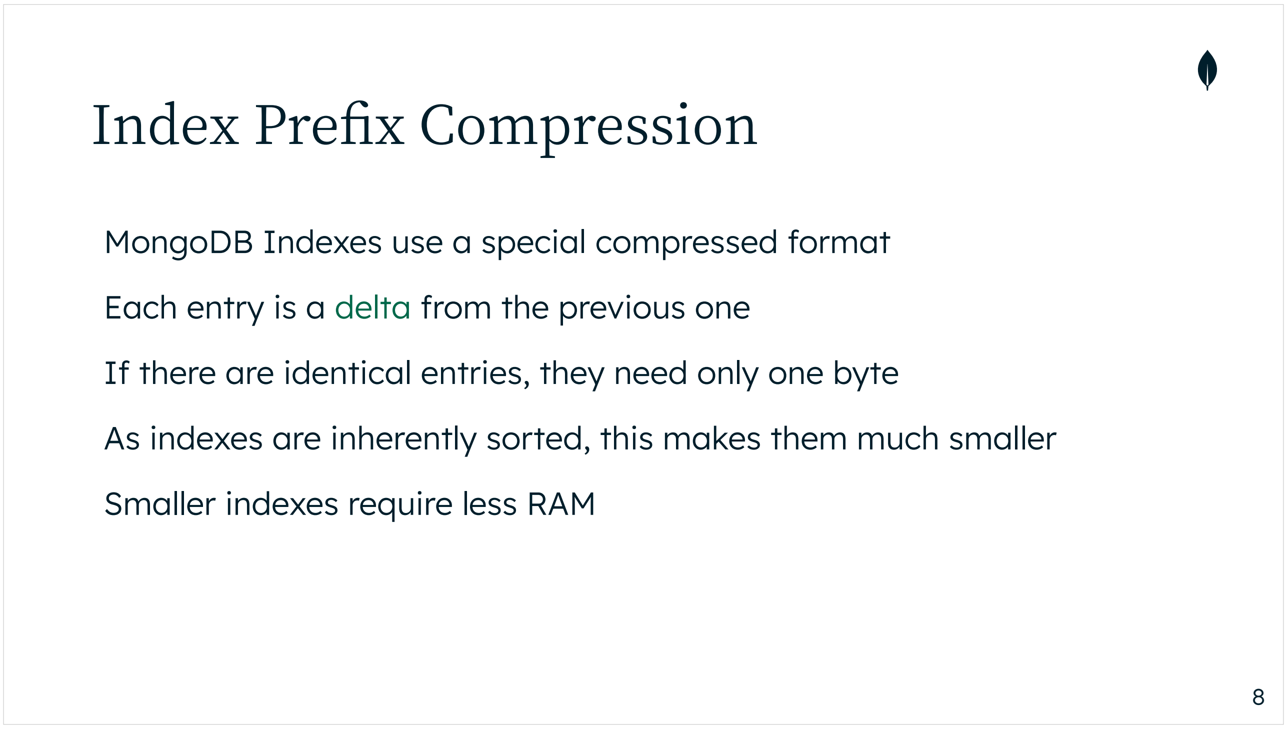
As a result, if the document moves the index doesn't change.

Data in an index is ordered.

- When looking for specific information, you don't need to look in every document.
- Indexes are in a binary (BTree) structure, so O(log N) to lookup.
- Indexes are traversed by value and then point to a document.

Values in a MongoDB index point to document **identity**.

- Identity is a 64-bit value in a hidden field assigned when a document is inserted.
- Documents in collections are in BTrees ordered by **identity**.
- It's like an internal primary key.
- It's not maintained between replicas though
    - You can see them by adding .showRecordId() to a cursor
    - If their physical location changes, the index stays the same



Index Prefix Compression

MongoDB Indexes use a special compressed format

Each entry is a delta from the previous one

If there are identical entries, they need only one byte

As indexes are inherently sorted, this makes them much smaller

Smaller indexes require less RAM

MongoDB uses index prefix compression to reduce the space that indexes consume.
Where an entry shares a prefix with a previous entry in the block, it has a pointer to that entry and length, and then the new data.
So subsequent identical keys take very little space. This helps optimize cache usage.

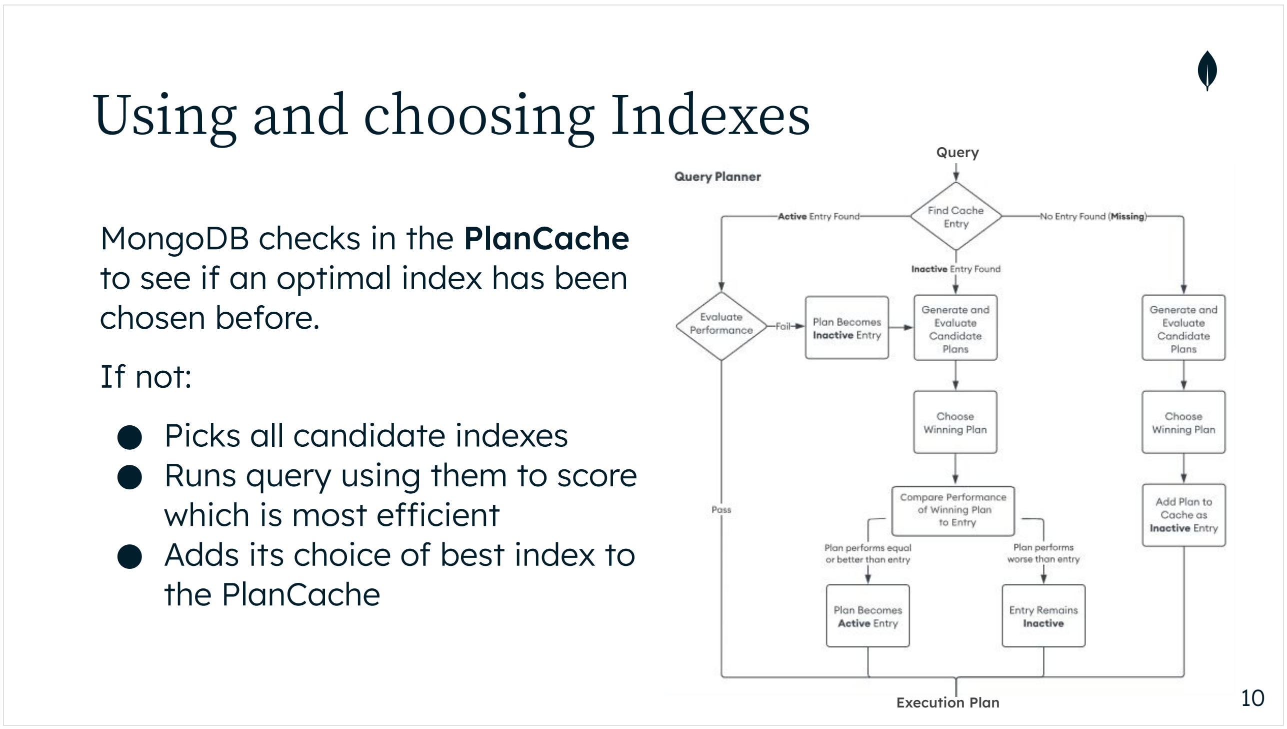
When to use an index

Every query should use an index!

- Scanning records is very inefficient, even if it is not all of them

MongoDB can be configured to disallow queries without indexes (notablescan parameter)

See https://www.mongodb.com/docs/manual/reference/parameters/#mongodb-parameter-param.notablescan

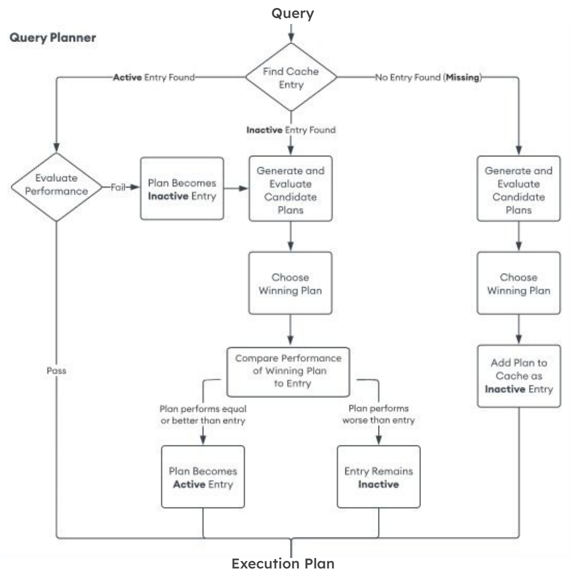


Using and choosing Indexes

MongoDB checks in the **PlanCache** to see if an optimal index has been chosen before.

If not:

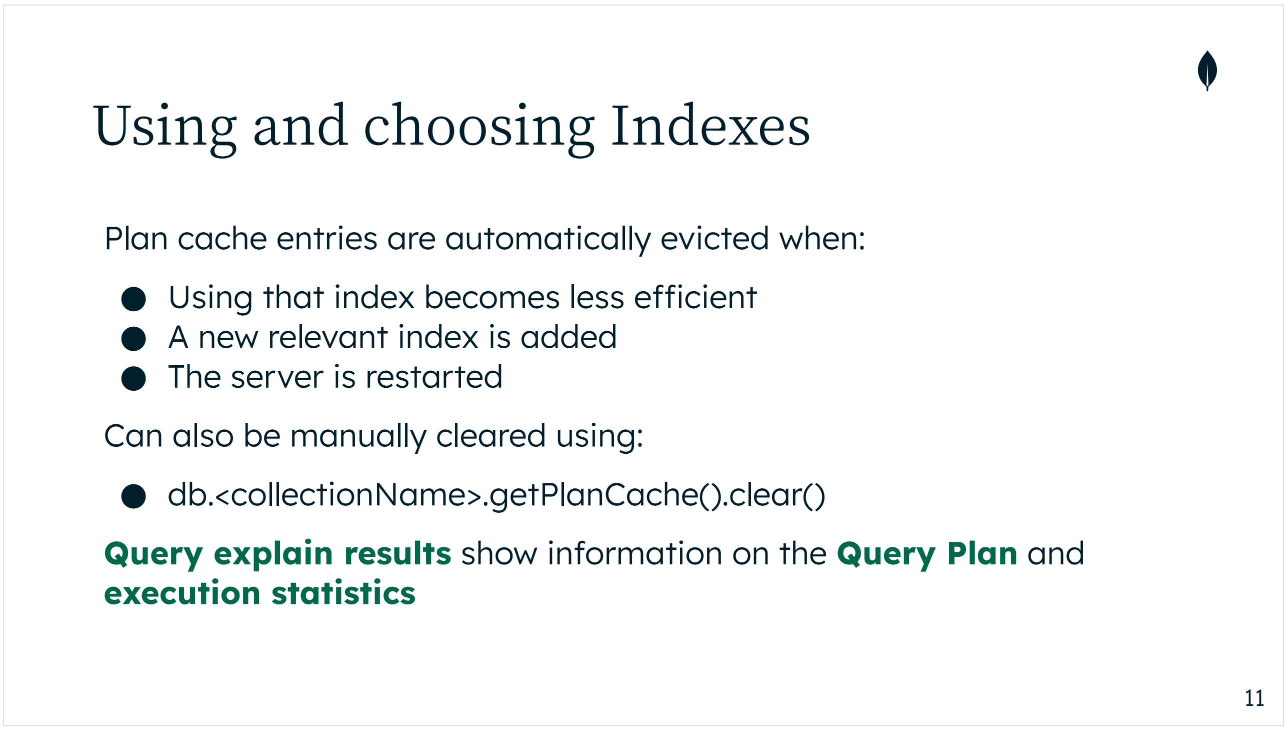
- Picks all candidate indexes
- Runs query using them to score which is most efficient
- Adds its choice of best index to the PlanCache



Query
Query Planner
Find Cache Entry
Active Entry Found
No Entry Found (Missing)
Inactive Entry Found
Evaluate Performance
Fail
Plan Becomes Inactive Entry
Generate and Evaluate Candidate Plans
Generate and Evaluate Candidate Plans
Choose Winning Plan
Choose Winning Plan
Pass
Compare Performance of Winning Plan to Entry
Add Plan to Cache as Inactive Entry
Plan performs equal or better than entry
Plan performs worse than entry
Plan Becomes Active Entry
Entry Remains Inactive
Execution Plan

The plan cache is a repository that contains the access plans for queries that were optimized
MongoDB uses an empirical technique to look for the best index, this does not require any collection statistics and is well suited
to the simple - single collection type queries mongoDB performs, normally with just one index.

If there are no appropriate indexes then it will run a collection scan.
If there is only one it will use that.
If there are more than one it could use it will 'Race' them - the index which returns 100 results first is chosen as the winner.



Using and choosing Indexes

Plan cache entries are automatically evicted when:

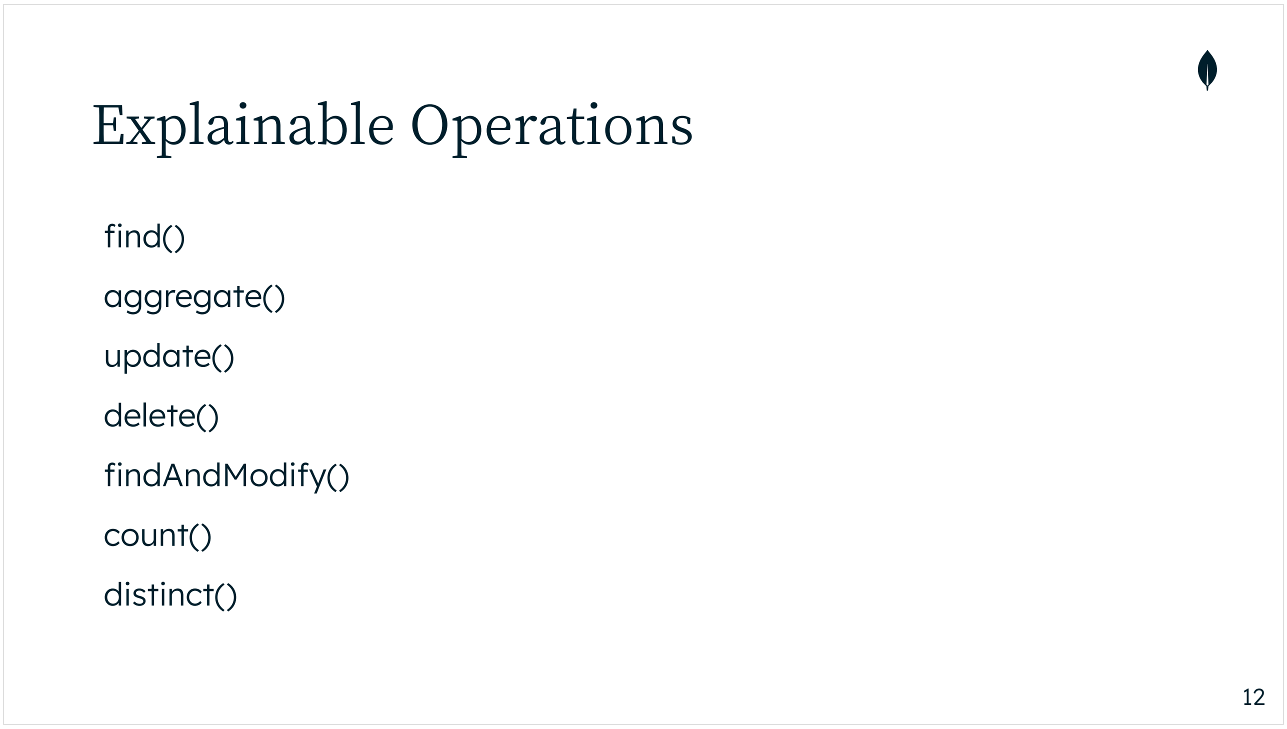
- Using that index becomes less efficient
- A new relevant index is added
- The server is restarted

Can also be manually cleared using:

- db.<collectionName>.getPlanCache().clear()

Query explain results show information on the **Query Plan** and **execution statistics**

MongoDB keeps a cache of the best index for any given "Shape" of query and how well it performed in the "Race" and if future queries do badly it will have a new "Race"
If a new, relevant index is created, or the server is restarted, MongoDB will invalidate the cache entry and try again.
We can run commands to see what's in the plan cache if we have a complex issue to debug but normally there will only be one useful index.



Explainable Operations

find()

aggregate()

update()

delete()

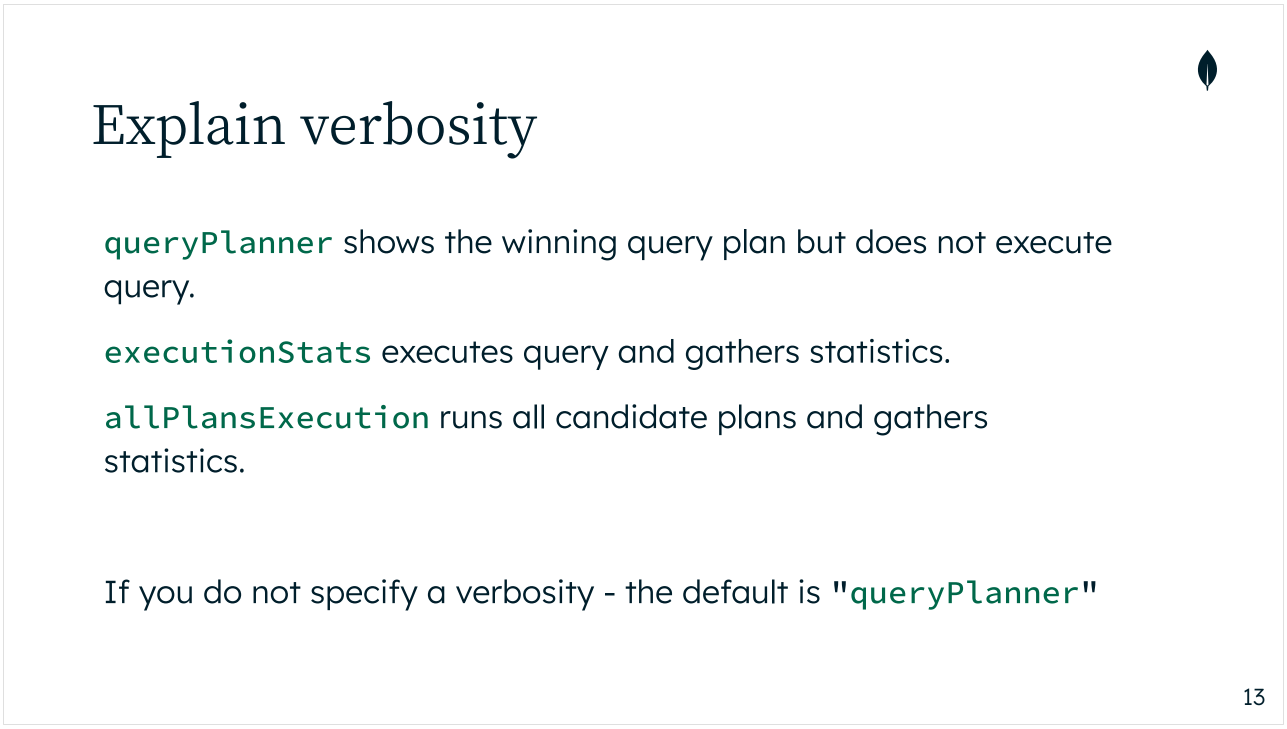
findAndModify()

count()

distinct()

Explain can be run on the operations shown above
Note that updateOne() or updateMany() does not allow explain, so update() with options can be used. Options related to multidoc or single doc updates as well as specific update operators such as upserts or different collation options and even queryFilters.

An explain on an update() does not actually modify the document(s).



Explain verbosity

`queryPlanner` shows the winning query plan but does not execute query.

`executionStats` executes query and gathers statistics.

`allPlansExecution` runs all candidate plans and gathers statistics.

If you do not specify a verbosity - the default is `"queryPlanner"`

By default, we only see what the DB Engine intends to do

“executionStats” gives us more detail, such as how long the query takes to run and how much data it has to examine

“allPlansExecution” runs all candidate plans and gathers statistics for comparison

Interpret Explain Plan Results

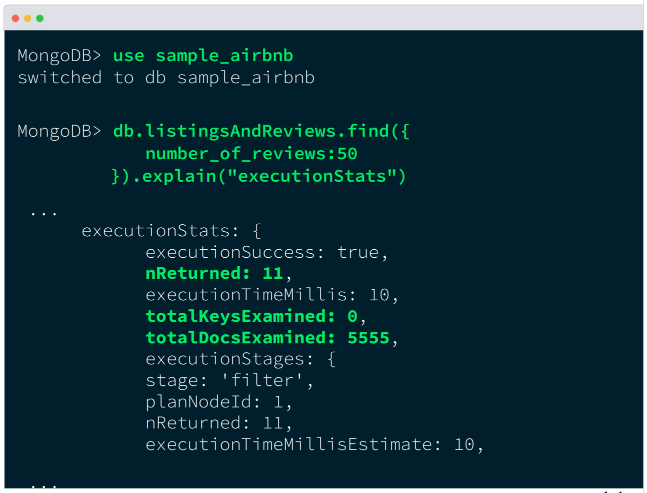
We can see this query looked at all 5,555 documents, returning 11 in 10 milliseconds

We can create an index to improve this.

Key metrics are in green here.

Execution stages include:

COLLSCAN
IXSCAN
FETCH

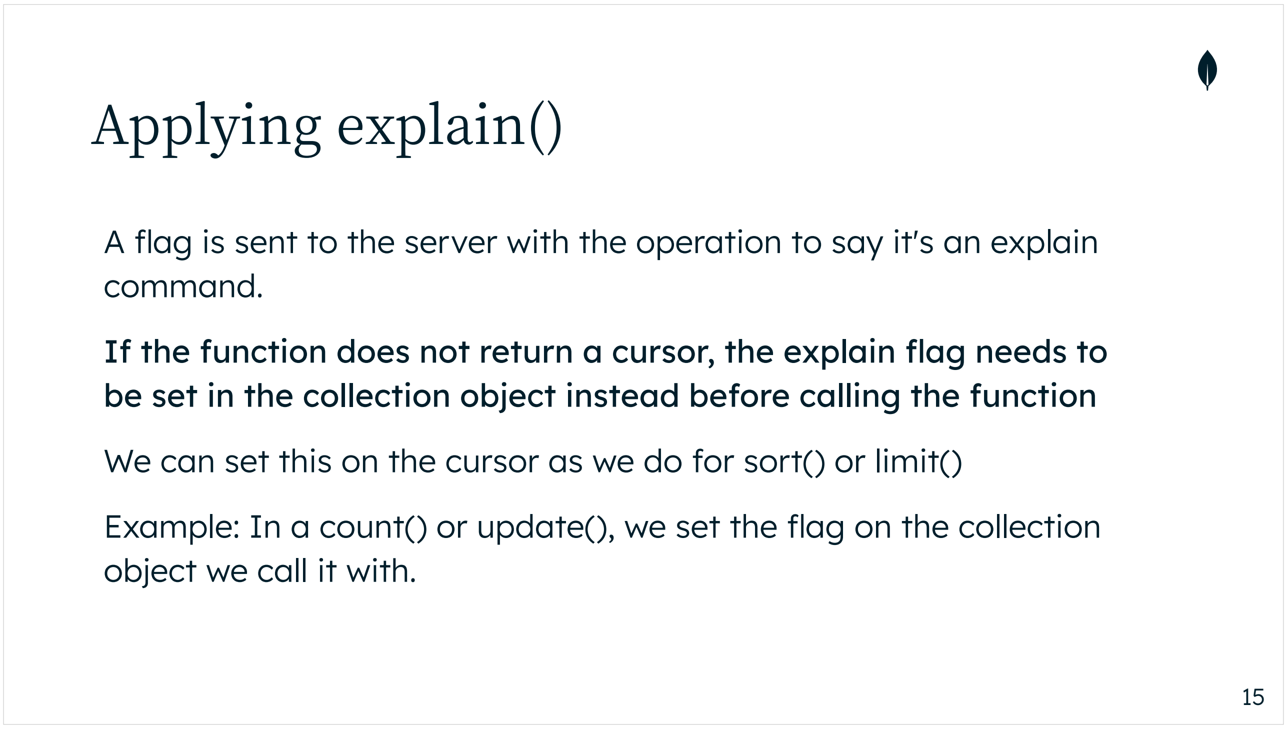


```
MongoDB> use sample_airbnb
switched to db sample_airbnb

MongoDB> db.listingsAndReviews.find({
          number_of_reviews:50
        }).explain("executionStats")

 ...
      executionStats: {
            executionSuccess: true,
            nReturned: 11,
            executionTimeMillis: 10,
            totalKeysExamined: 0,
            totalDocsExamined: 5555,
            executionStages: {
            stage: 'filter',
            planNodeId: 1,
            nReturned: 11,
            executionTimeMillisEstimate: 10,
```

nReturned is how many documents this stage returns - e.g., the index may narrow to 100 documents, but then an unindexed filter drops that to 10 documents
totalKeysExamined - number of index entries
totalDocsExamined - number of documents read

Ideally, all three of the above are the same number.

Stage shows us whether a collection scan or index was used.



Applying explain()

A flag is sent to the server with the operation to say it's an explain command.

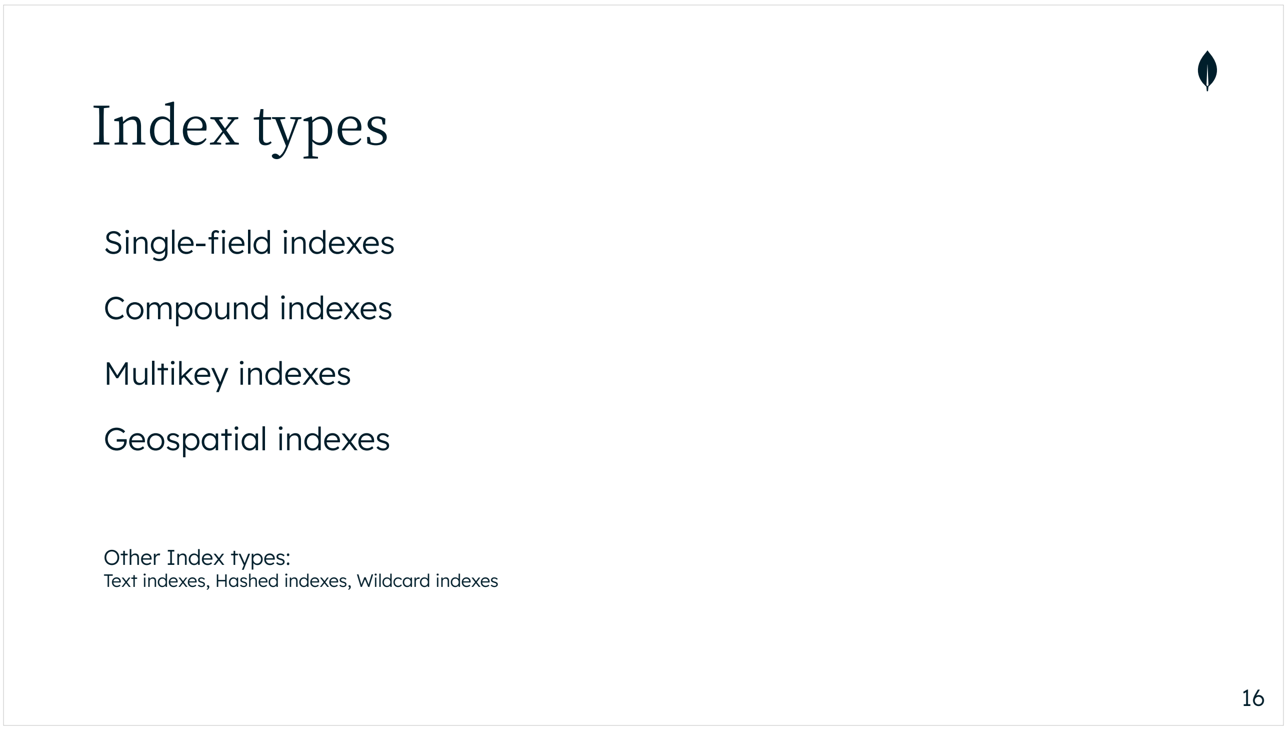
If the function does not return a cursor, the explain flag needs to be set in the collection object instead before calling the function

We can set this on the cursor as we do for sort() or limit()

Example: In a count() or update(), we set the flag on the collection object we call it with.

```
peoplecoll = db.people
explainpeoplecoll = peoplecoll.explain()
explainpeoplecoll.count()
```

A query is sent to the server once we start requesting from the cursor - so we set a flag on the cursor to request the explain plan rather than the results.
If we don't have a cursor, calling explain() on a collection returns a collection with that flag set.



Index types

Single-field indexes

Compound indexes

Multikey indexes

Geospatial indexes

Other Index types:
Text indexes, Hashed indexes, Wildcard indexes

Index types are similar to other DB vendors.
Text indexes, hashed indexes and wildcard indexes are note recommended for general use

- Text - rather use Atlas Search if possible
- Hashed - B-Tree structure is optimized for consecutive values, but hashes are semi-random
- Wildcard - only useful in very specific areas, but can become very large

Single-Field Indexes

Optimize finding values for a given field

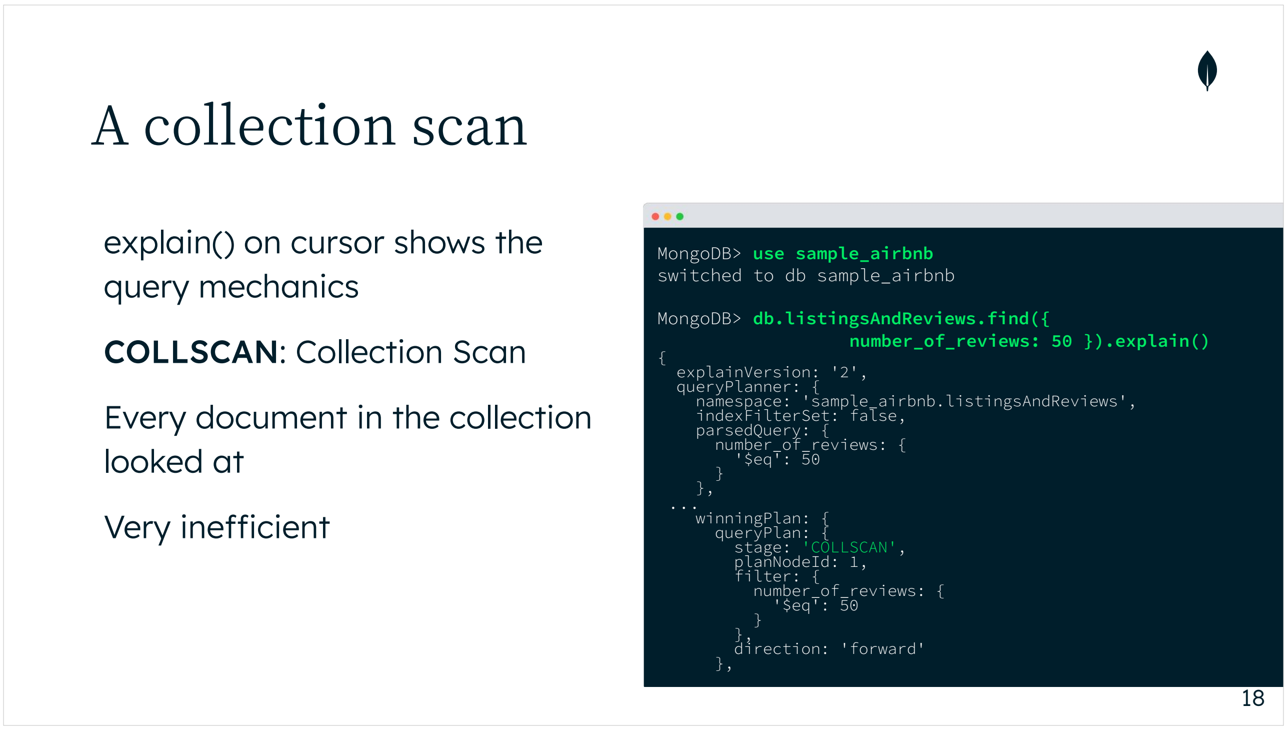
- Specified as field name and direction
- The direction is irrelevant for a single field index
- The field itself can be any data type.

Avoid indexing an Object type:

- Indexes the whole object as a comparable blob
- Can use it for range searches, but better to index individual fields as less error prone

Indexing an Array is covered later

A single-field index is the most basic kind
Builds a tree of all values pointing to the documents they are in
It makes looking up a value O(log n) versus O(n)
Allows range queries and sorting to be performant too
Range searching against objects means if we have an objects like {a:1, b:5} with an index, we can create a range query to find all values with {a:1} or even {a : {$gt:1}} without knowing b. `{ myObj : { $gte: { a:1 }, $lte : { a:1, b: MaxKey() }}}`
When querying objects, both exact matches and ranges rely on the **field order** being the same. We must check our app programming language does not rearrange the field order.



A collection scan

explain() on cursor shows the query mechanics

COLLSCAN: Collection Scan

Every document in the collection looked at

Very inefficient

```
MongoDB> use sample_airbnb
switched to db sample_airbnb

MongoDB> db.listingsAndReviews.find({
                  number_of_reviews: 50 }).explain()
{
  explainVersion: '2',
  queryPlanner: {
    namespace: 'sample_airbnb.listingsAndReviews',
    indexFilterSet: false,
    parsedQuery: {
      number_of_reviews: {
        '$eq': 50
      }
    },
 ...
    winningPlan: {
      queryPlan: {
        stage: 'COLLSCAN',
        planNodeId: 1,
        filter: {
          number_of_reviews: {
            '$eq': 50
          }
        },
        direction: 'forward'
      },
```

Queries tend to always be quick on small datasets, so we will use tools to see the implications rather than merely observing time.

The example query has no index, so the DB Engine must look through every document in the collection.

If the collection is too big and isn't cached in RAM, then a huge amount of Disk IO will be consumed, and it will be very slow.

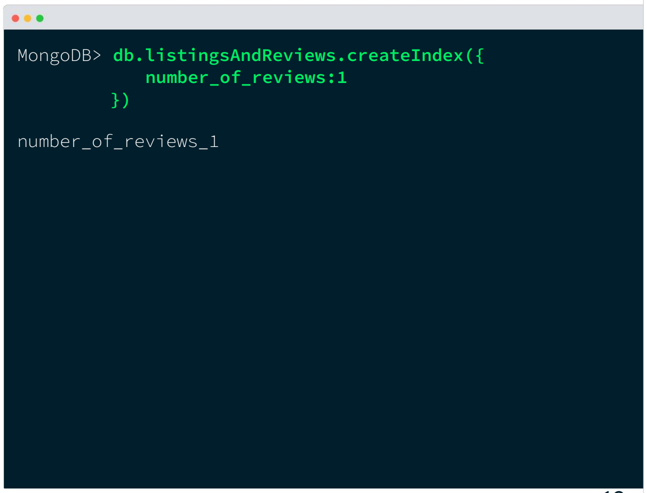
Even if in RAM, it is a lot of data to look through

Creating a simple index

In development, we can create an index using `createIndex()` on the primary

In production, we need to look at the impact this will have.

There are better ways to do it in production.



```
MongoDB> db.listingsAndReviews.createIndex({
             number_of_reviews:1
         })

number_of_reviews_1
```

Creating an index takes an object of fields with an ascending index (1) or descending index (-1) option.

In languages where members of objects aren't ordered, the syntax is a little different.

The return value is the name of the newly created index.

Making indexes in the production system needs to consider the impact on the server, cache, disks, and any locking that may occur.

Rolling index builds are generally preferred in clustered production environments.

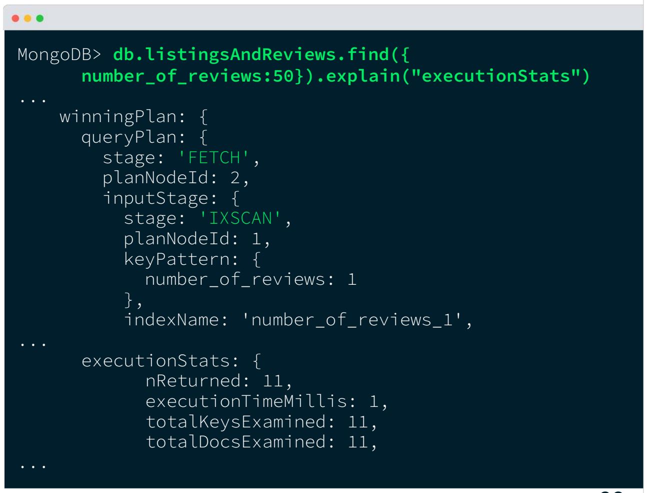
Index Demonstration

Here we see two stages:

IXSCAN (Look through the index) returning a list of 11 documents identities

FETCH (Read known document) getting the document itself

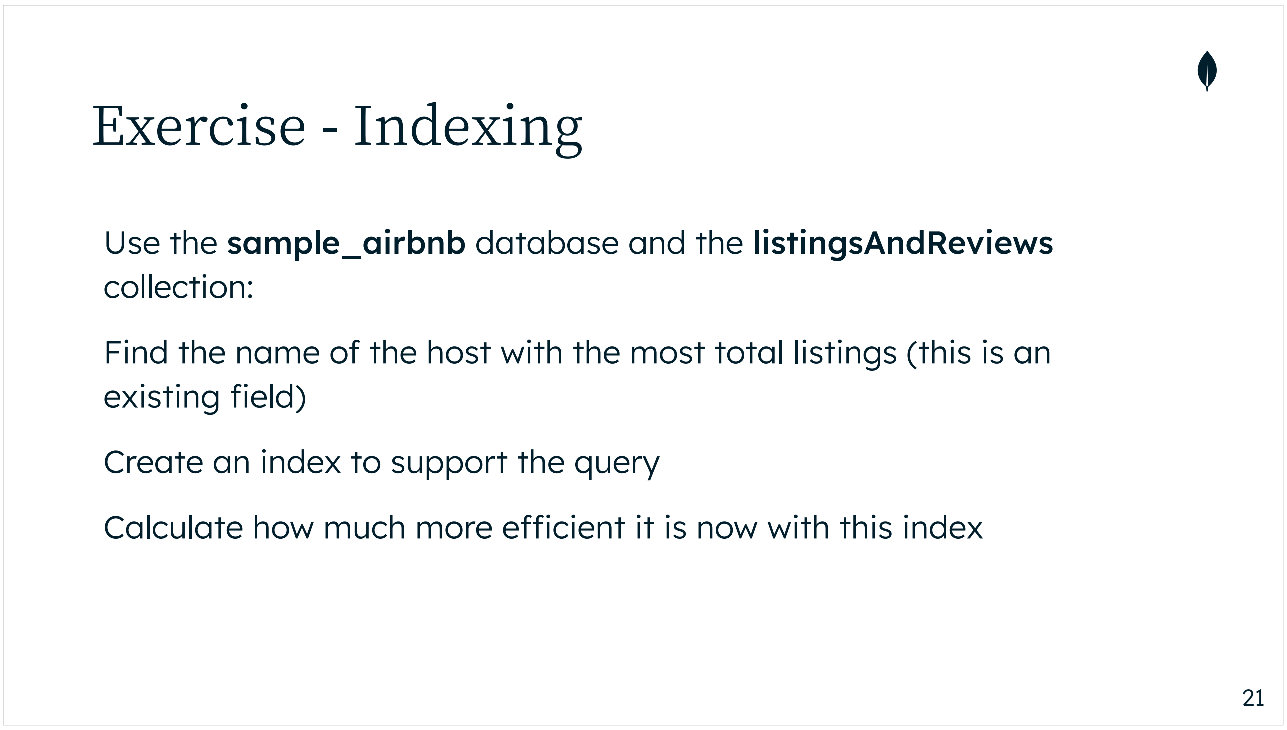
The total time was one millisecond



```
MongoDB> db.listingsAndReviews.find({
      number_of_reviews:50}).explain("executionStats")
...
    winningPlan: {
      queryPlan: {
        stage: 'FETCH',
        planNodeId: 2,
        inputStage: {
          stage: 'IXSCAN',
          planNodeId: 1,
          keyPattern: {
            number_of_reviews: 1
          },
          indexName: 'number_of_reviews_1',
...
      executionStats: {
            nReturned: 11,
            executionTimeMillis: 1,
            totalKeysExamined: 11,
            totalDocsExamined: 11,
...
```

The example shows improved efficiency of using an index - Note how **totalKeysExamined**, **totalDocsExamined**, and **nReturned** are all the same.
Same behavior as an RDBMS

```
 executionStats: {
  executionSuccess: true,
  nReturned: 11,
  executionTimeMillis: 1,
  totalKeysExamined: 11,
  totalDocsExamined: 11,
  executionStages: {
   stage: 'nlj',
   planNodeId: 2,
   nReturned: 11,
   executionTimeMillisEstimate: 0,
   opens: 1,
   closes: 1,
   saveState: 0,
   restoreState: 0,
   isEOF: 1,
   totalDocsExamined: 11,
   totalKeysExamined: 11,
   collectionScans: 0,
   collectionSeeks: 11,
   indexScans: 0,
   indexSeeks: 1,
   indexesUsed: [
 ...
```



Exercise - Indexing

Use the **sample_airbnb** database and the **listingsAndReviews** collection:

Find the name of the host with the most total listings (this is an existing field)

Create an index to support the query

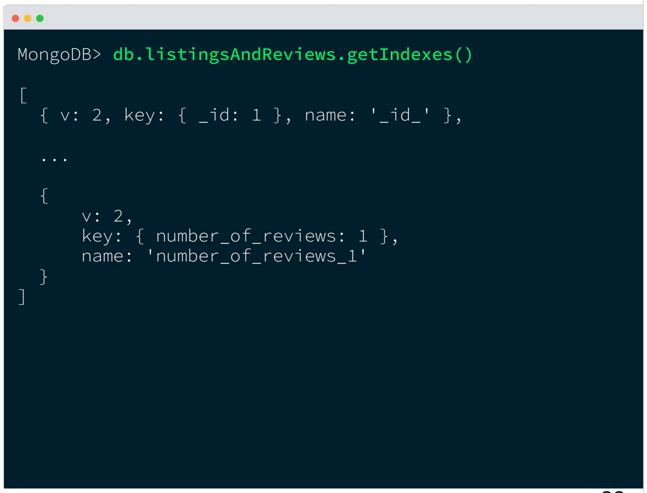
Calculate how much more efficient it is now with this index

Note - this is a tricky question for a number of reasons!

A simpler version of this could also be provided if the above is too hard for the audience:
Find the name of the hosts that have listings greater than 1000 (instead of most total listings)

Listing Indexes

`getIndexes()` on a collection gives us the index definitions for that collection.

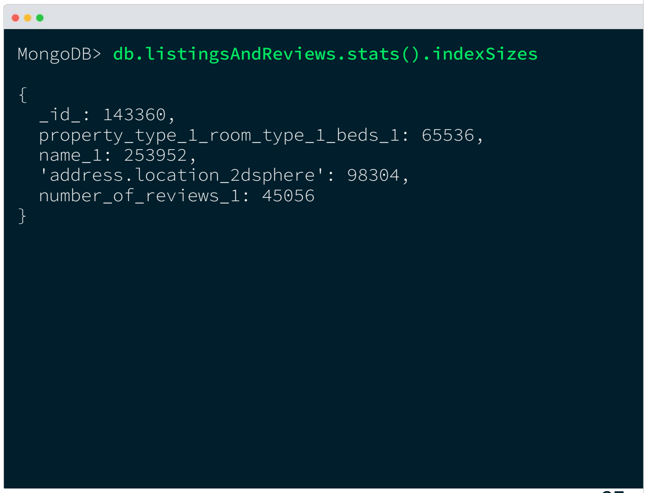


```
MongoDB> db.listingsAndReviews.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },

  ...

  {
      v: 2,
      key: { number_of_reviews: 1 },
      name: 'number_of_reviews_1'
  }
]
```

getIndexes() shows index information (the number of indexes may vary from what you see here on the slide)

Index sizes

We can call `stats()` method and look at the **indexSizes** key to see how large each index is in bytes.



```
MongoDB> db.listingsAndReviews.stats().indexSizes

{
  _id_: 143360,
  property_type_1_room_type_1_beds_1: 65536,
  name_1: 253952,
  'address.location_2dsphere': 98304,
  number_of_reviews_1: 45056
}
```

Among other options, the scaling factor can be passed to see stats in desired unit. `db.listingsAndReviews.stats({ scale: 1024 }).indexSizes` shows the index sizes in KB.

Unique Index

Indexes can enforce a unique constraint

Example,

```
db.a.createIndex({custid: 1}, {unique: true})
```

NULL is a value and so only one record can have a NULL in unique field.

NB: Recall that a missing field in a document has a logical value of 'null'.

Partial Index

Partial indexes index a subset of documents based on values.

Can greatly reduce index size

Example,

```
db.orders.createIndex(
   { customer: 1 },
   { partialFilterExpression: { archived:
false } } )
```

Partial indexes can be used to only index fields that meet a specified filter expression. Useful for reducing index size.
The Index is only used where the query matches the condition for document to be indexed.

Sparse Index

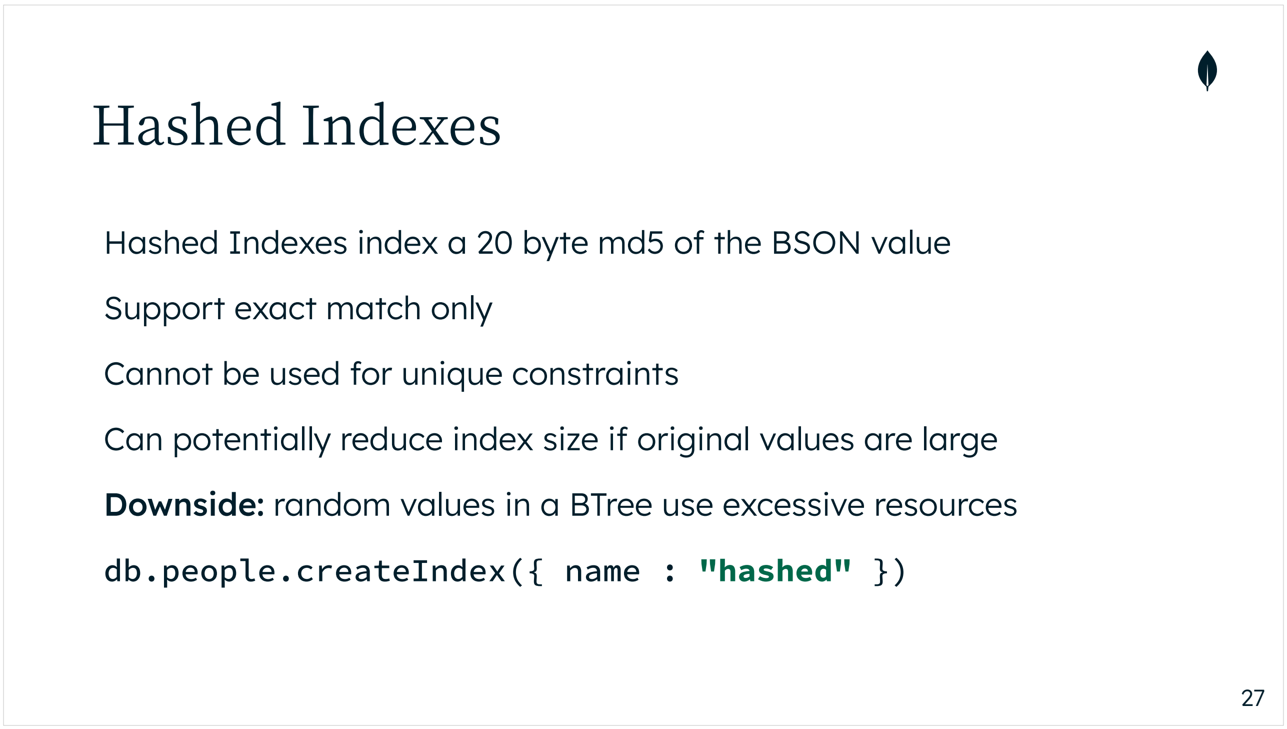
Sparse Indexes don't index missing fields.

Example,

```
db.scores.createIndex({score: 1}, {sparse: true})
```

Sparse Indexes are superseded by Partial Indexes

To create a sparse index, we recommend creating a Partial index with `{field: {$exists: true}}` as your partialFilterExpression.



Hashed Indexes

Hashed Indexes index a 20 byte md5 of the BSON value

Support exact match only

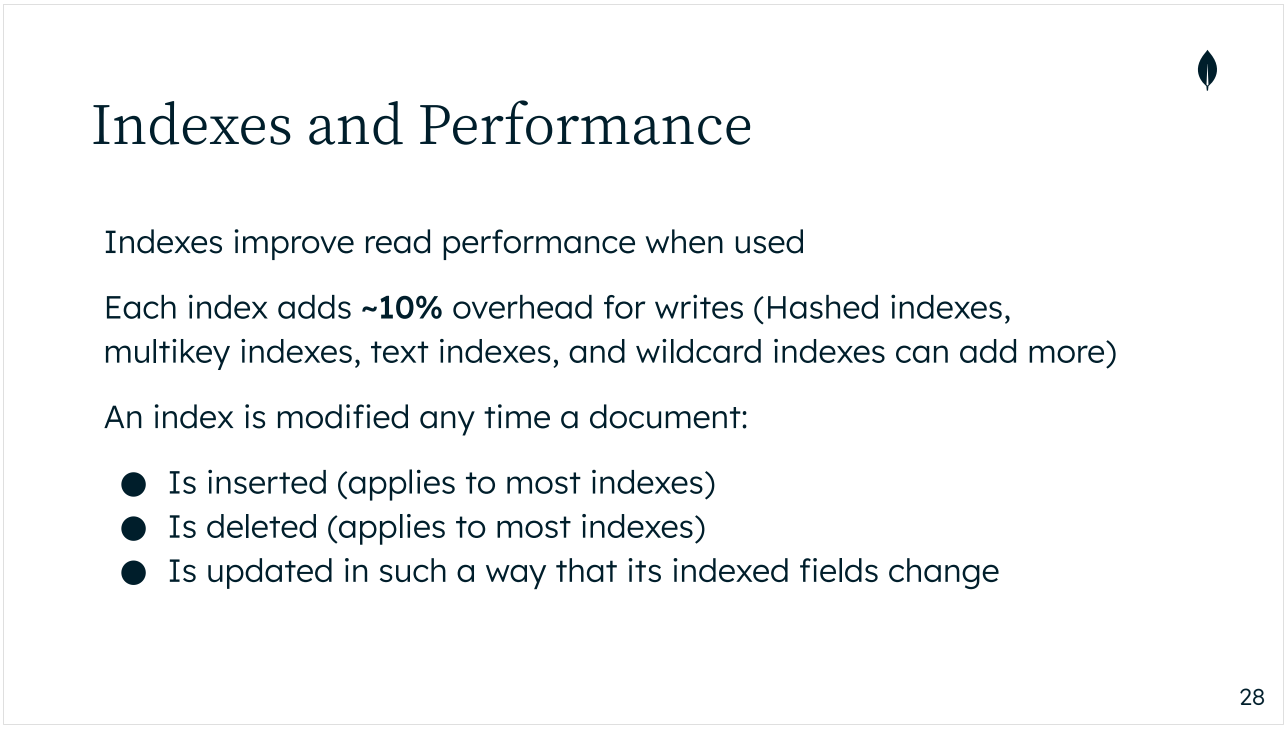
Cannot be used for unique constraints

Can potentially reduce index size if original values are large

Downside: random values in a BTree use excessive resources

```
db.people.createIndex({ name : "hashed" })
```

Hashed indexing creates performance challenges for range matches etc., so it should be used with caution.
Random values (Hashes or traditional GUIDS) in a BTree maximize the requirement for RAM and Disk/IO and so should be avoided. This is covered later in the course.



Indexes and Performance

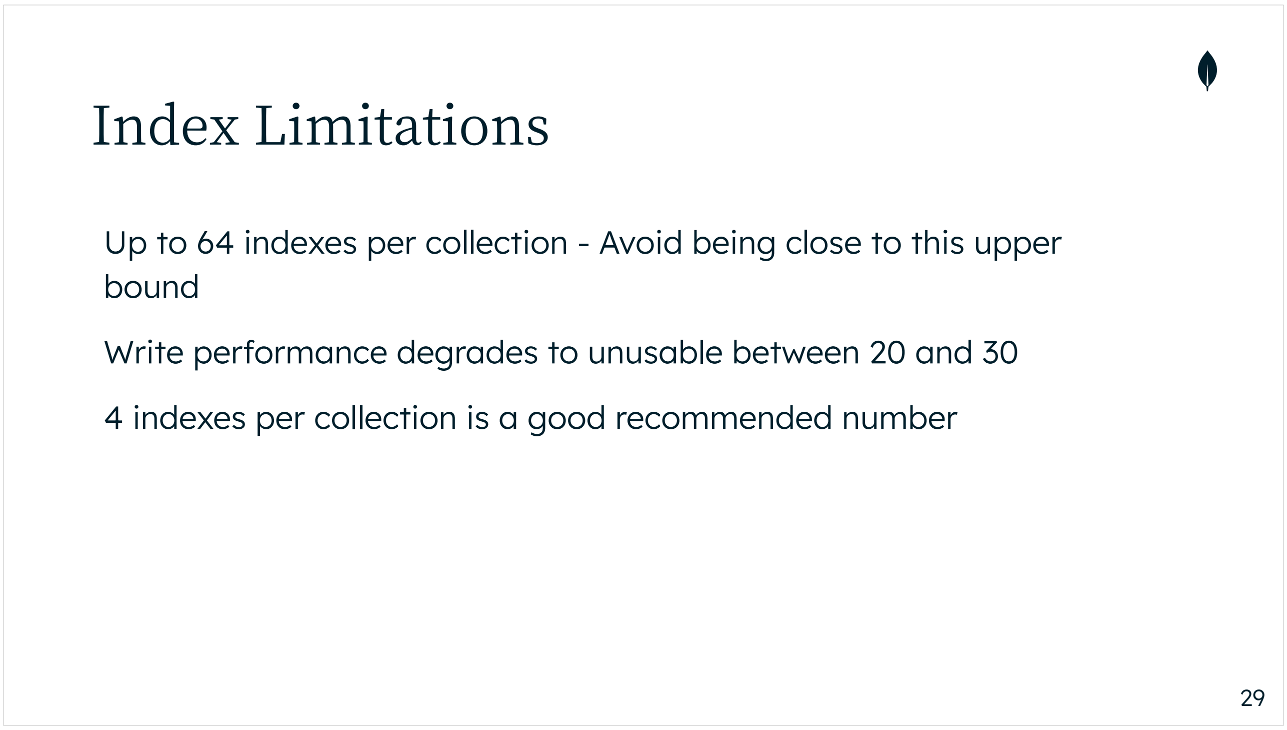
Indexes improve read performance when used

Each index adds **~10%** overhead for writes (Hashed indexes, multikey indexes, text indexes, and wildcard indexes can add more)

An index is modified any time a document:

- Is inserted (applies to most indexes)
- Is deleted (applies to most indexes)
- Is updated in such a way that its indexed fields change

Indexes must be applied with careful consideration as they do create overhead when writing data
Unused indexes should be identified and removed
At times the partial and sparse indexes remain unaffected while insert or delete operations.



Index Limitations

Up to 64 indexes per collection - Avoid being close to this upper bound

Write performance degrades to unusable between 20 and 30

4 indexes per collection is a good recommended number

The hard limit is 64 indexes per collection, but you should not have anywhere near this number

Use Indexes with Care

Every query should use an index

Every index should be used by a query

Indexes require server memory, be mindful about the choice of key

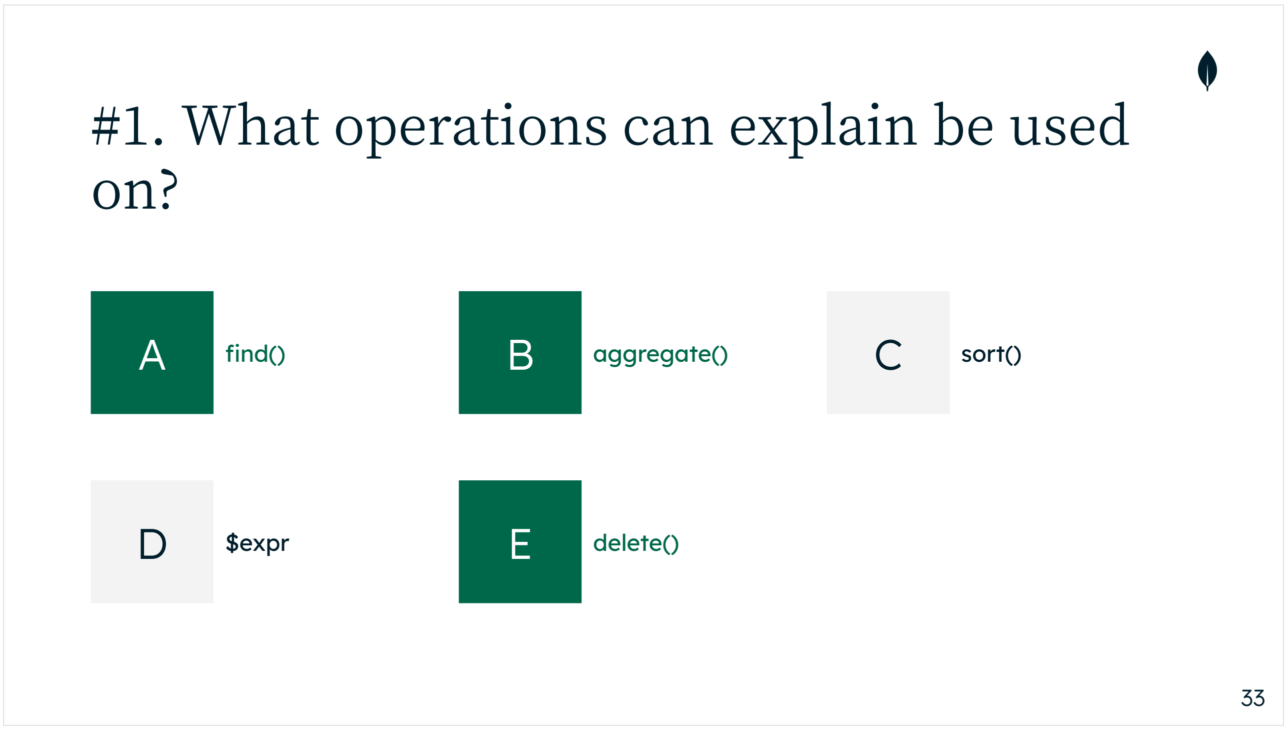
Depending on the size and available resources, indexes will either be used from disk or cached. You should aim to fit indexes in the cache. Otherwise, performance will be seriously impacted.

Quiz Time!

#1. Which 3 operations can explain be used on?

A	find()	B	aggregate()	C	sort()
D	$expr	E	delete()		

Answer in the next slide.



#1. What operations can explain be used on?

[✓ A] find()

[✓ B] aggregate()

C sort()

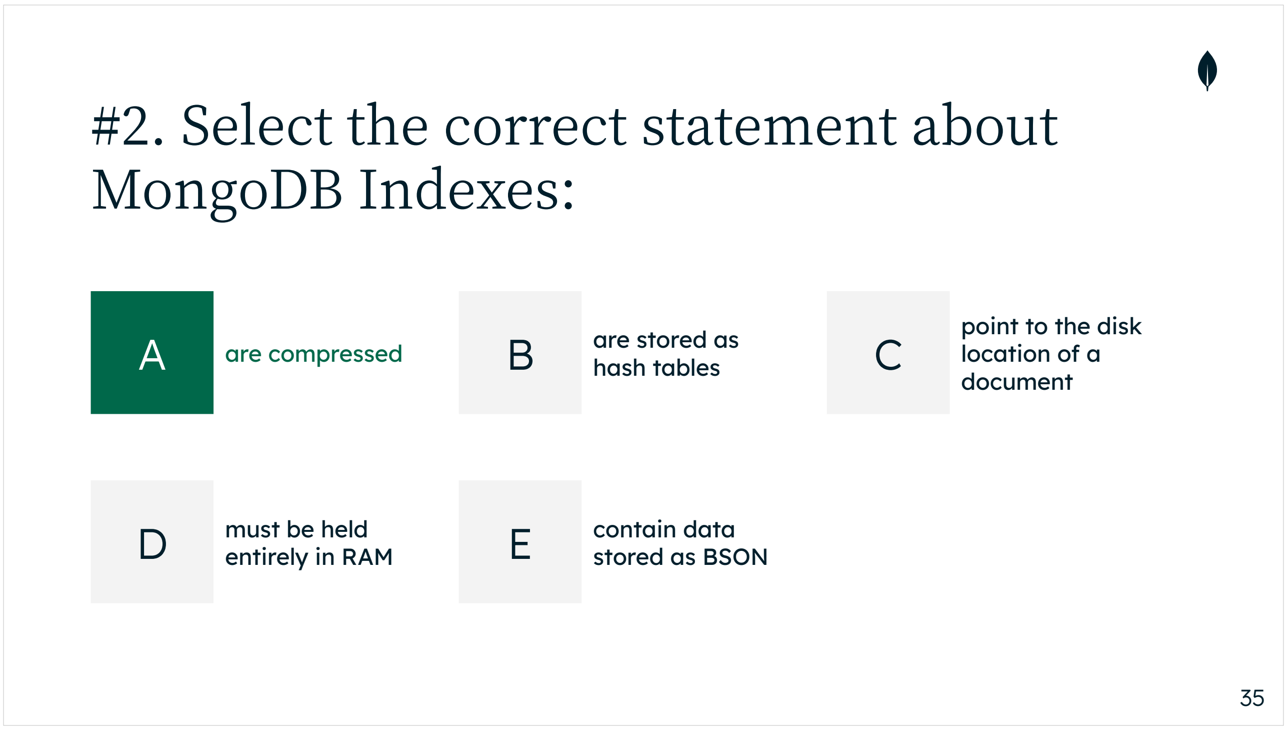
D $expr

[✓ E] delete()

Explain() can be applied to finds, aggregations, deletes, updates and count.
$expr is a specific update operator not the operation itself
sort() is a modifier for find, not a distinct operation. However, you can (and should!) explain find operations where you sort the results.

#2. Select the correct statement about MongoDB Indexes:

A	are compressed
B	are stored as hash tables
C	point to the disk location of a document
D	must be held entirely in RAM
E	contain data stored as BSON



#2. Select the correct statement about MongoDB Indexes:

[✓ A] are compressed

B are stored as hash tables

C point to the disk location of a document

D must be held entirely in RAM

E contain data stored as BSON

Indexes are always compressed in MongoDB regardless of if its in the wiredTiger cache or on disk or on the OS cache. Indexes aren't stored in a hash table structure but a structure more akin to a b-tree, it points to the recordID of a document not the location of it on disk. Indexes can be paged into cache as necessary and they contain data in the index prefix so only A is correct here.

Recap

Indexes improve efficiency and speed of reads

Every query should use an index

MongoDB can explain() how an operation is being indexed

Exercise Answers

Answer - Exercise: Indexing

Find the name of the host with the most total listings

```
> db.listingsAndReviews.find({},{"host.host_total_listings_count":1,
"host.host_name":1}).sort({"host.host_total_listings_count":-1}).limit(1)
{ "_id" : "12902610", "host" : { "host_name" : "Sonder", "host_total_listings_count" : 1198 } }
```

Create an index to support the query and show how much more efficient it is:

```
"executionTimeMillis" : 11,
"totalDocsExamined" : 5555,
"works" : 5559,

> db.listingsAndReviews.createIndex({"host.host_total_listings_count": 1})
"executionTimeMillis" : 1,
"totalKeysExamined" : 1,
"totalDocsExamined" : 1,
"works" : 2,
```

Note: In the first query where we apply a sort and limit, MongoDB doesn't guarantee consistent return of the same document especially when there are simultaneous writes happening. It is advisable to include the _id field in sort if consistency is required.