

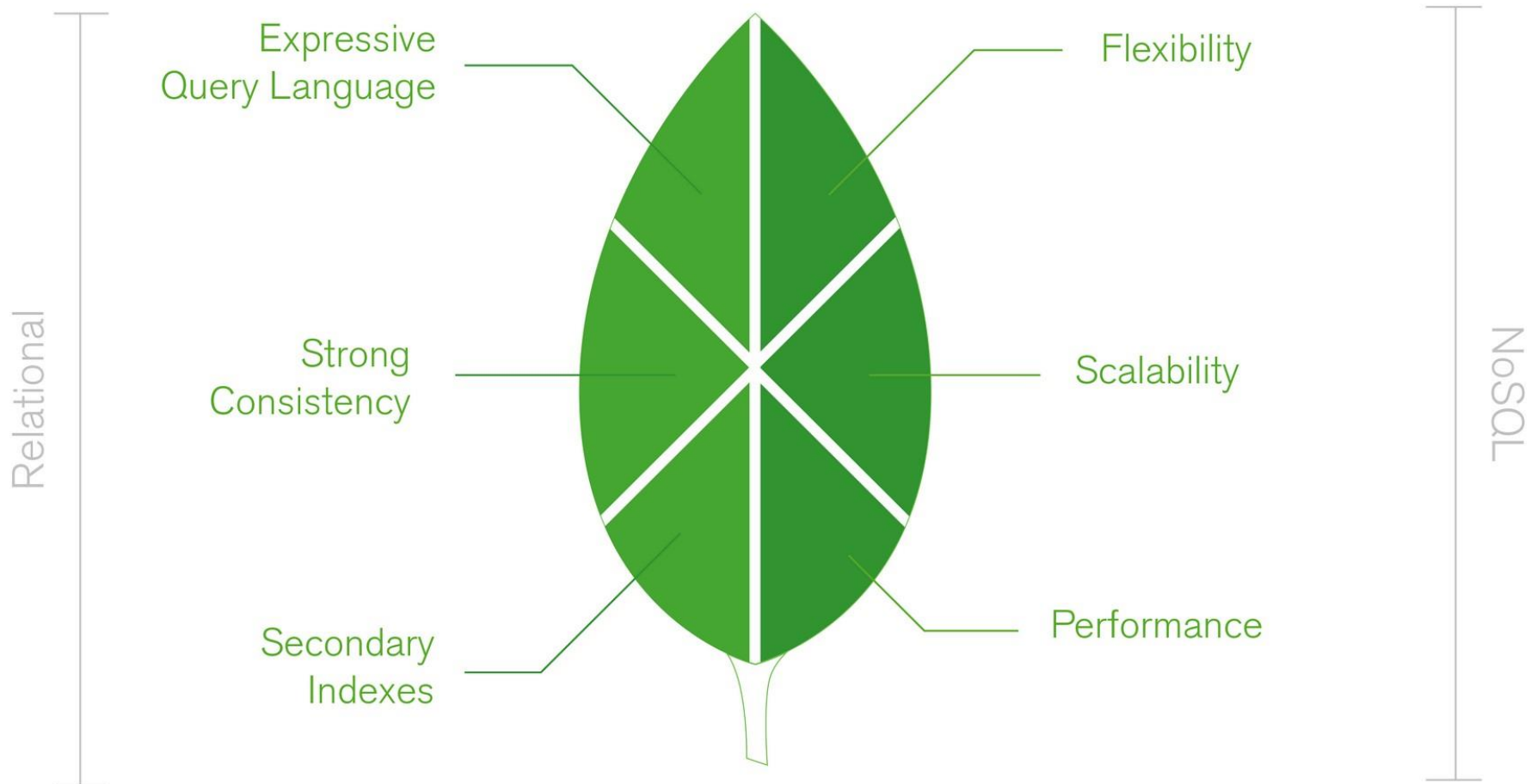
Indexing and Performance Tuning

ANJU MUNOTH

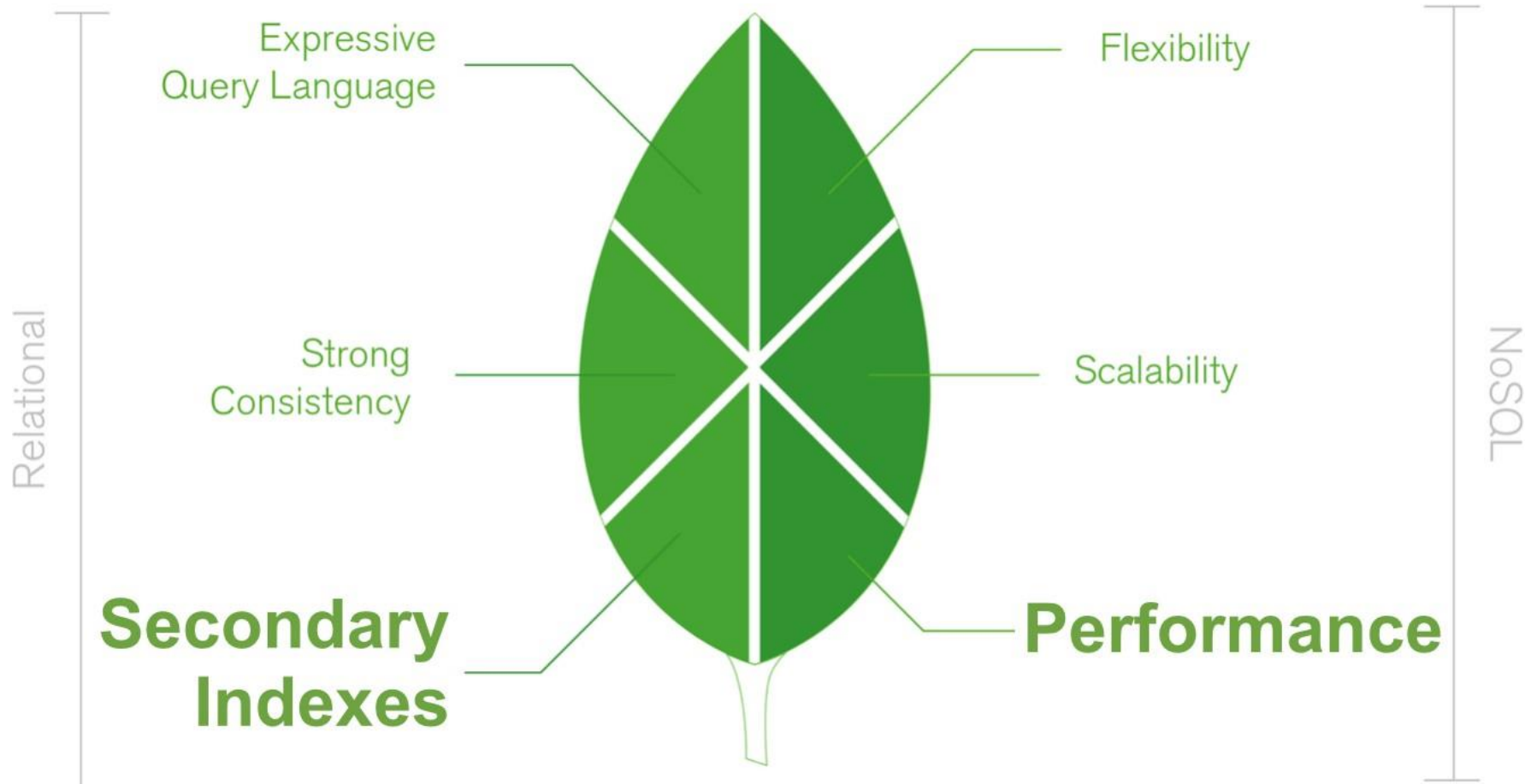
Indexes are data structures that store collection's data set in a form that is easy to traverse. Indexes help perform the following functions:

- Execute queries and find documents that match the query criteria without a collection scan.
- Limit the number of documents a query examines.
- Store field value in the order of the value.
- Support equality matches are range-based queries.

MongoDB's unique architecture



MongoDB's unique architecture



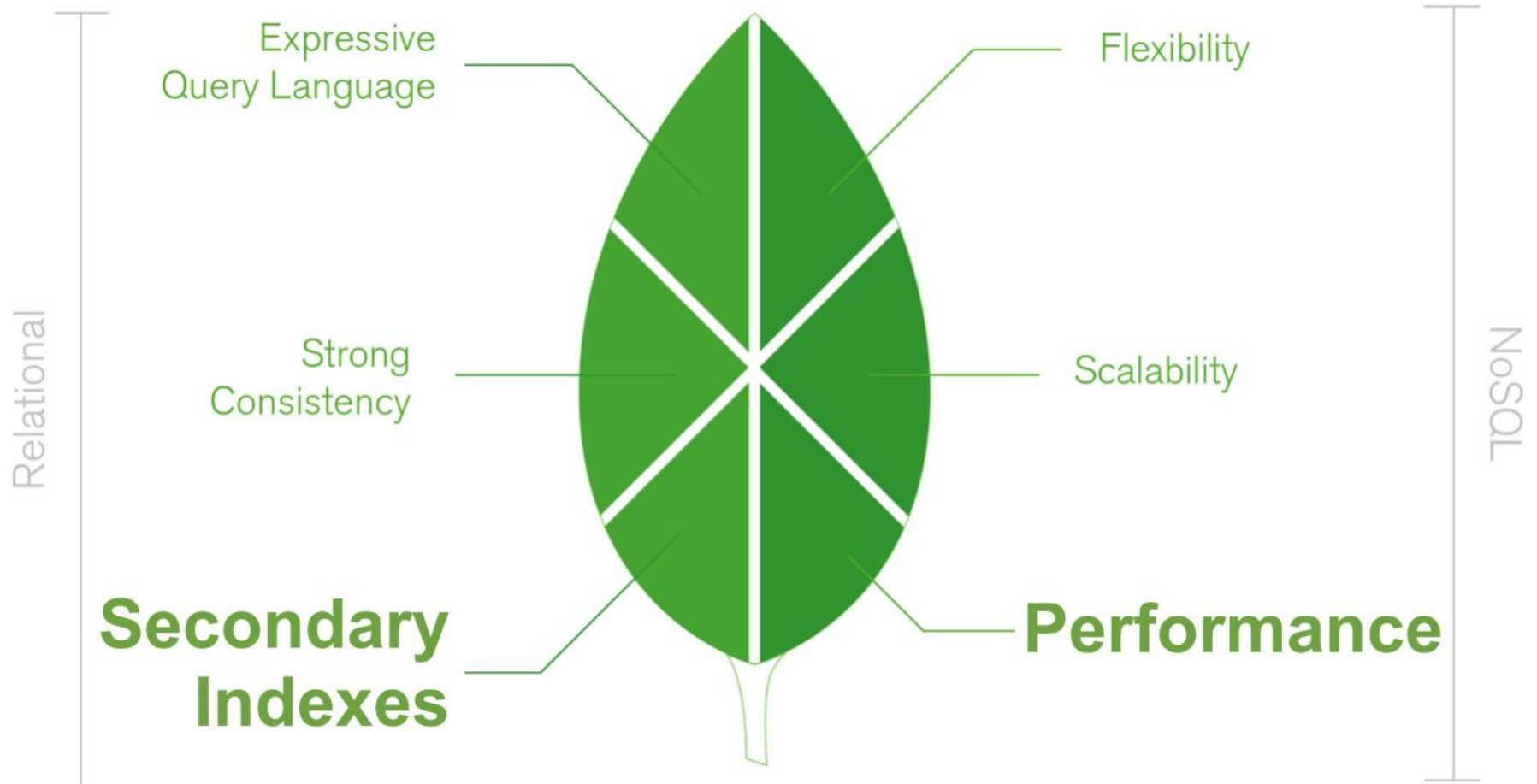
**Secondary
Indexes**



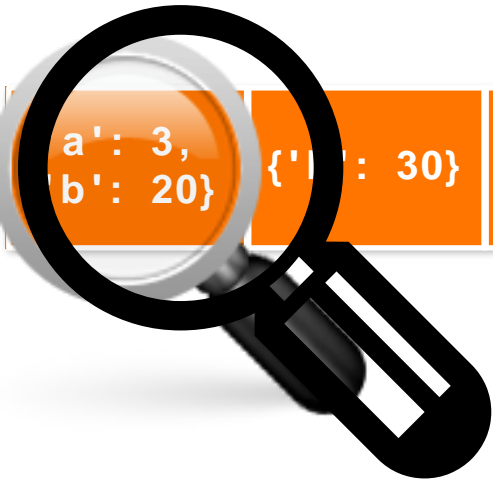
Performance

Indexes are the single biggest tunable performance factor in MongoDB

MongoDB's unique architecture



Full collection scan



{ 'a': 3,
'b': 20 }

{ 'a': 30 }

{ 'b': 40,
'a': 1 }

{ 'a': 9,
'b': 4,
'a': 7 }

{ 'a': 1,
'b': 20 }

{ 'a': 11 }

{ 'a': 17 }

{ 'a': 5,
'b': 80 }

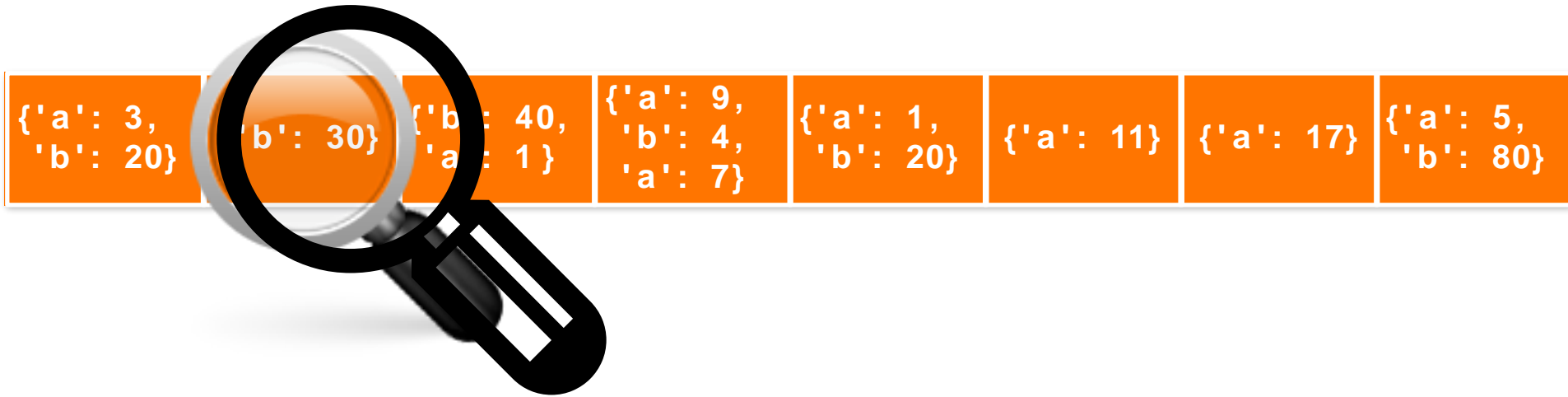
```
db.test.find({a:5})
```

documents scanned:

1

Flexibility

Full collection scan



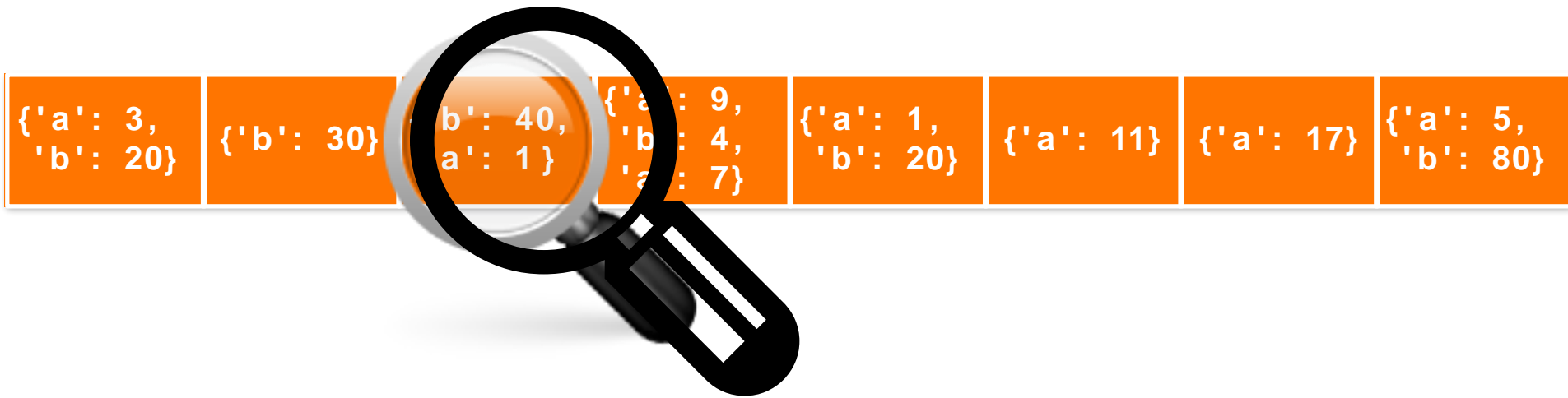
```
db.test.find({a:5})
```

documents scanned:

2

Flexibility

Full collection scan



```
db.test.find({a:5})
```

documents scanned:

3

Flexibility

Full collection scan



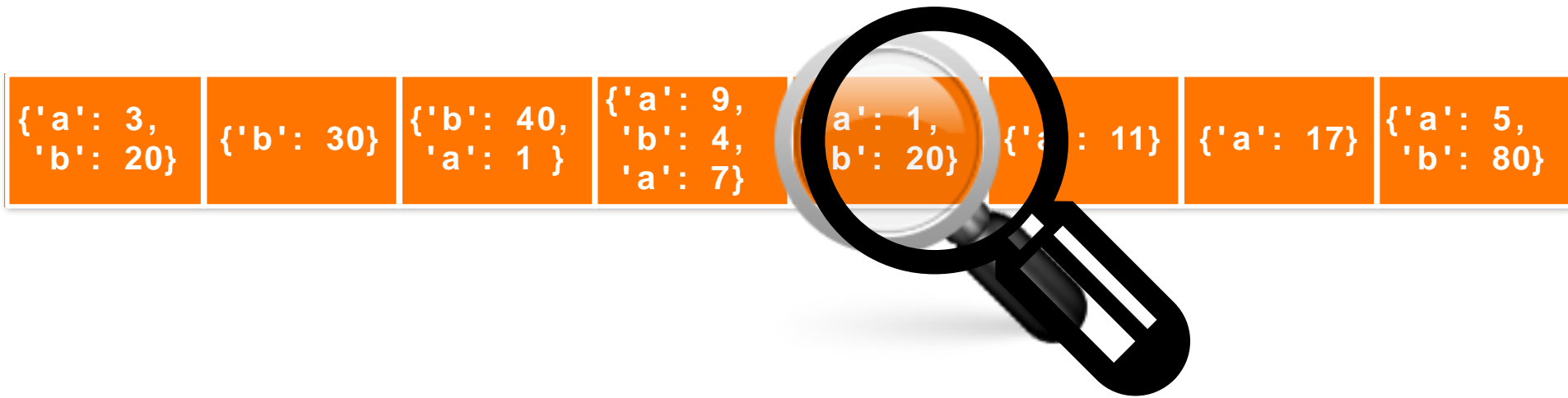
```
db.test.find({a:5})
```

documents scanned:

4

Flexibility

Full collection scan



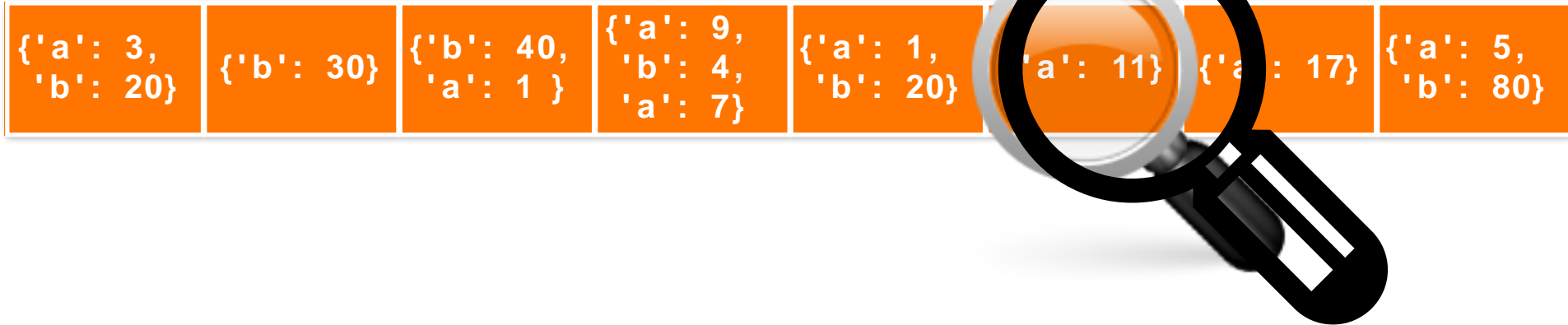
```
db.test.find({a:5})
```

documents scanned:

5

Flexibility

Full collection scan



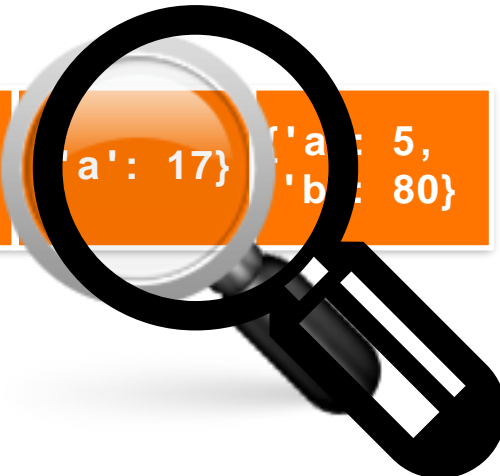
```
db.test.find({a:5})
```

documents scanned:

6

Flexibility

Full collection scan



<code>{ 'a': 3, 'b': 20 }</code>	<code>{ 'b': 30 }</code>	<code>{ 'b': 40, 'a': 1 }</code>	<code>{ 'a': 9, 'b': 4, 'a': 7 }</code>	<code>{ 'a': 1, 'b': 20 }</code>	<code>{ 'a': 11 }</code>	<code>{ 'a': 17 }</code>	<code>{ 'a': 5, 'b': 80 }</code>
----------------------------------	--------------------------	----------------------------------	---	----------------------------------	--------------------------	--------------------------	----------------------------------

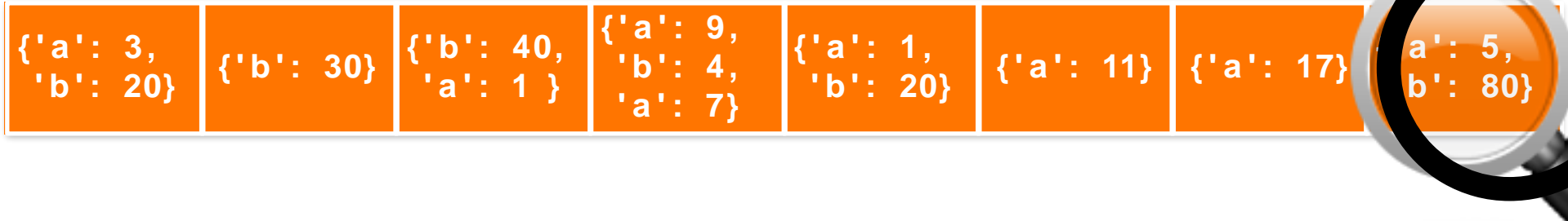
```
db.test.find({a:5})
```

documents scanned:

7

Flexibility

Full collection scan



Time Complexity for searching in a list with n entries: **$O(n)$**

```
db.test.find({a:5})
```

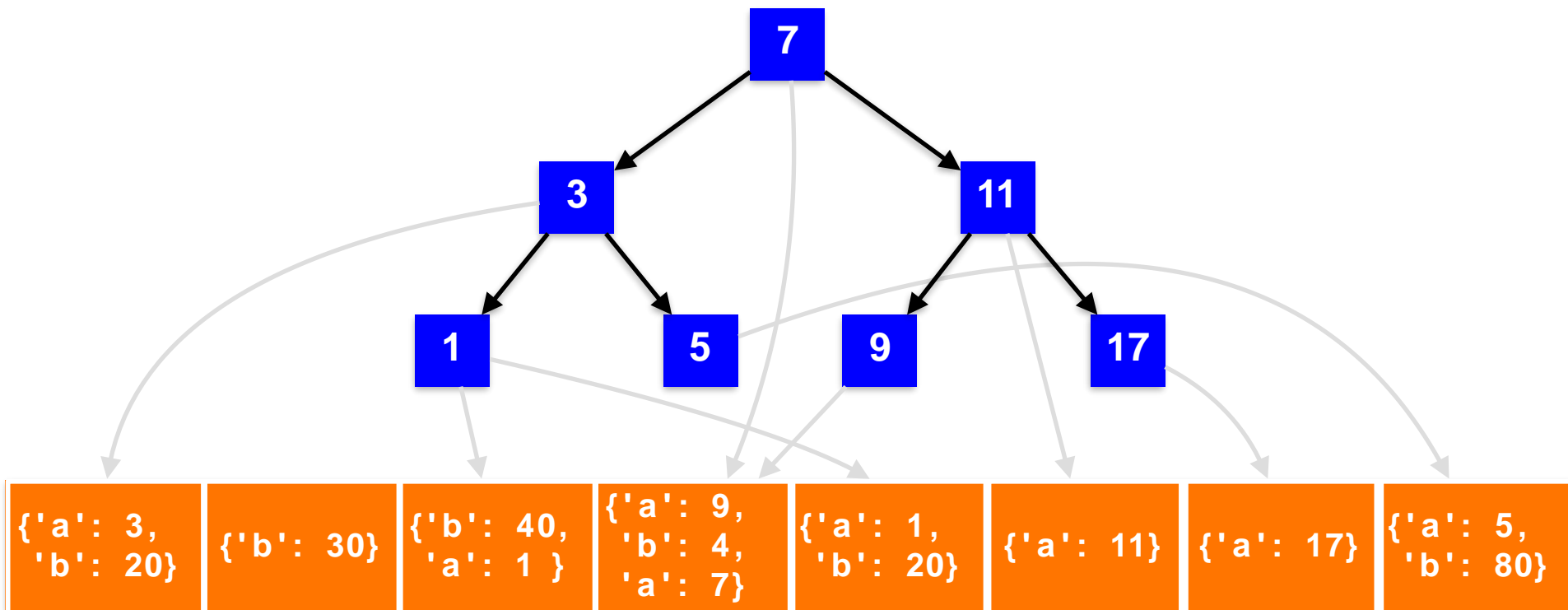
documents scanned:

8

Flexibility

Index scan

Index on 'a'

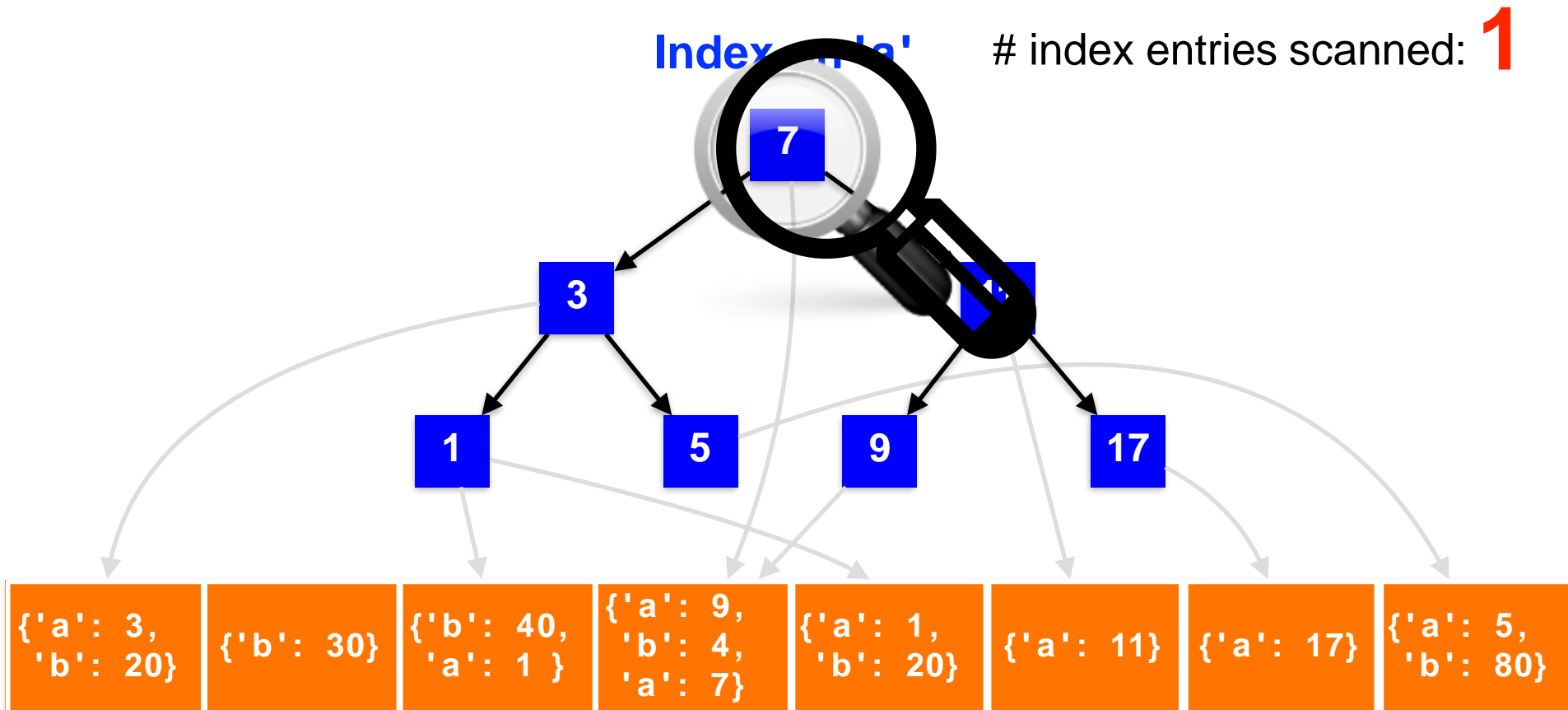


Index scan

```
db.test.find({a:5})
```

Index on 'a'

index entries scanned: **1**

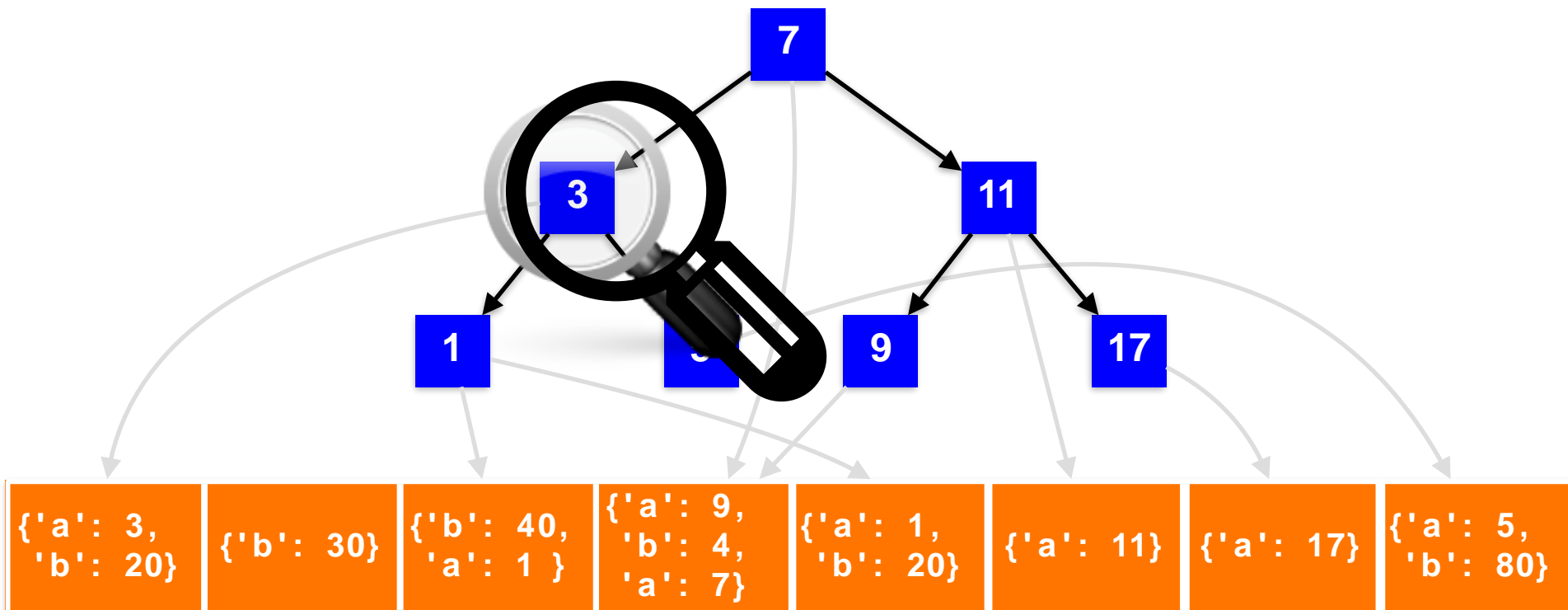


Index scan

```
db.test.find({a:5})
```

Index on 'a'

index entries scanned: **2**

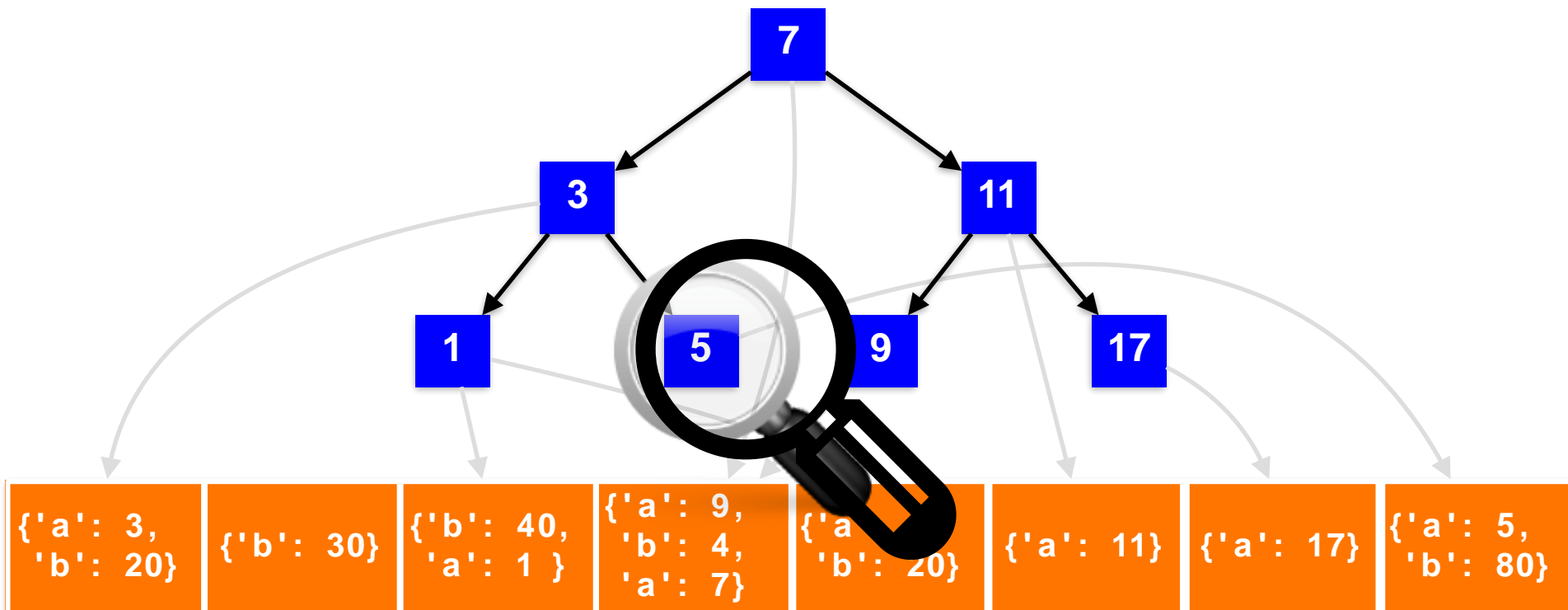


Index scan

```
db.test.find({a:5})
```

Index on 'a'

index entries scanned: **3**



Index scan

```
db.test.find({a:5})
```

Time Complexity for
searching in a binary
search tree with n nodes:

$O(\log_2 n)$

$$\log_2 10.000 = 13$$

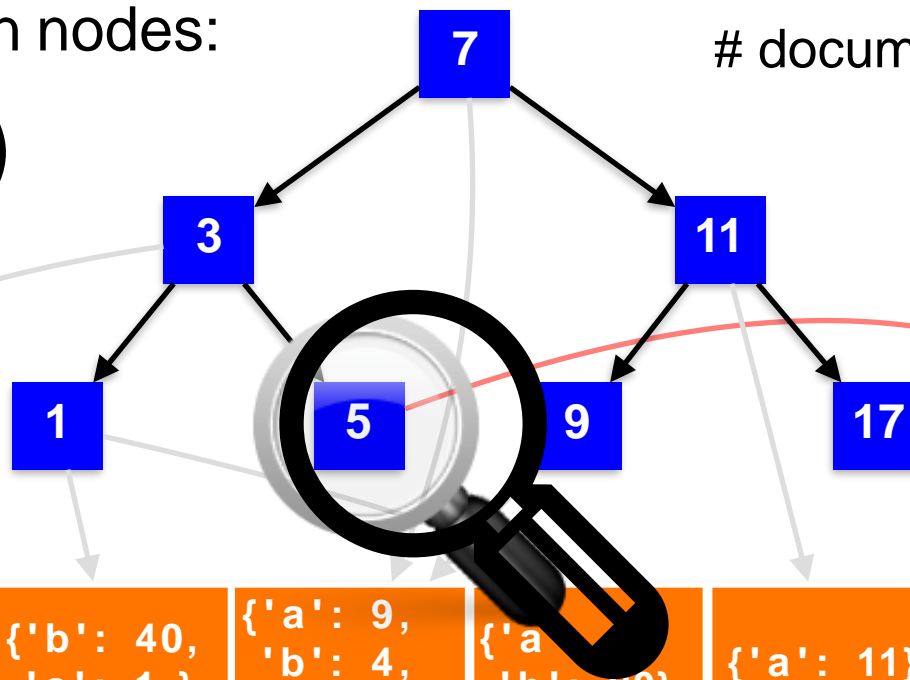
$$\log_2 100.000 = 16$$

$$\log_2 1.000.000 = 19$$

Index on 'a'

index entries scanned: **3**

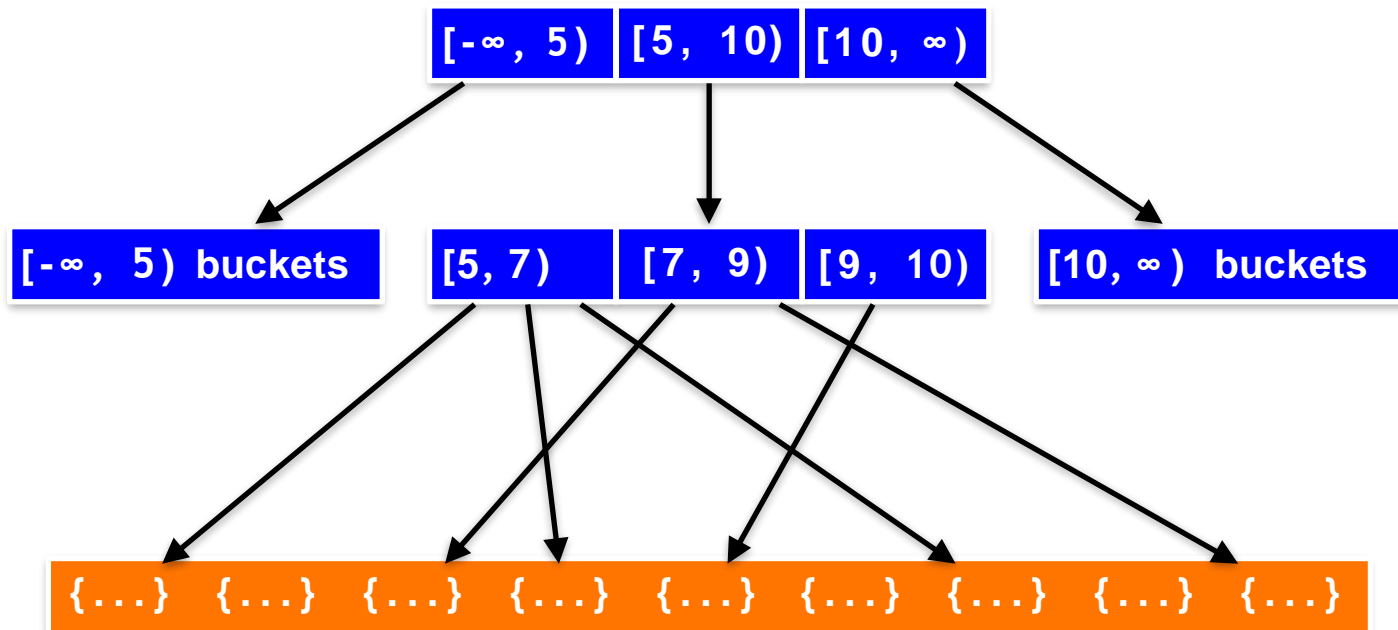
documents scanned: **1**



MongoDB Indexes are B-Trees

Searches, Insertions, and Deletions in logarithmic time.

Index on a



Create indexes

- `db.collection.createIndex({field_name: 1})`
- `db.collection.ensureIndex({field_name: -1})`
- `db.collection.ensureIndex(
 {field_name: -1},
 {background: true}
)`

Manage indexes

- `db.collection.getIndexes()`
- `db.collection.stats({index:"name"})`
- `db.collection.dropIndexes()`
- `db.collection.dropIndex("index_name")`
- `db.collection.reIndex()`

What can be indexed?

- single field `{a: 1}`
- compound keys / multiple fields `{a: 1, b: -1}`
- multikeys indexes / arrays of values `{a: [1,2,3]}`
- subdocuments

```
"address": {  
  "street": "Main"  
  "zipcode": 53511  
  "state": "WI"  
}
```
- embedded fields `{"address.state": 1}`

`_id`

- `_id` index, is the primary key for the collection and every doc must have a unique `_id` field
 - `ObjectId("5038050ef719dd2122000004")`

Options

- Unique, dropDups
- Sparse indexes
- Geospatial indexes (2d)
- TTL collections (expireAfterSeconds)

Types of Index

MongoDB supports the following index types for querying:

- **Default _id:** Each MongoDB collection contains an index on the default _id field.
- **Single Field:** For single-field index and sort operations, MongoDB can traverse the indexes either in the ascending or descending order.
- **Compound Index:** MongoDB supports user-defined indexes, such as compound indexes for multiple fields.
- **Multikey Index:** Used for indexing array data.
- **Geospatial Index:** Uses 2d indexes and 2d sphere indexes.
- **Text Indexes:** Searches data string in a collection.
- **Hashed Indexes:** MongoDB supports hash based sharding and provides hashed indexes.

Single Field Index

- MongoDB supports indexes on any document field in a collection.
- By default, the `_id` field in all collections have indexes.
- Moreover, applications and users add indexes for triggering queries and performing operations.
- MongoDB supports both, single field or multiple field indexes based on the operations the index-type performs.

```
db.items.createIndex( { "item" : 1 } )
```

Single Field Index on Embedded Document

Can index top level fields within a document. Similarly, can create indexes within embedded document fields.

```
{ "_id" : 3, "item" : "Book", "available" : true, "soldQty" :  
    144821, "category" : "NoSQL", "details" :  
    { "ISDN" : "111", "publisher" : "XYZ Company" },  
    "onlineSale" : true  
}
```

Use the query given below to create an index on the ISDN field and an embedded document.

```
db.items.createIndex( {details.ISDN: 1 } )
```

Compound Indexes

- A compound index in Mongo DB contains multiple single field indexes separated by a comma.
- compound indexes- a single index structure holds references to multiple fields within a collection's documents.

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

MongoDB limits the fields of a compound index to a maximum of 32.

Compound Indexes

The following operation creates an ascending index on the item and stock fields:

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

- The order of the fields listed in a compound index is important
- The index will contain references to documents sorted first by the values of the item field and, within each value of the item field, sorted by values of the stock field.
- In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index fields.
- That is, the index supports queries on the item field as well as both item and stock fields:

```
db.products.find( { item: "Banana" } )
```

```
db.products.find( { item: "Banana", stock: { $gt: 5 } } )
```

Compound Indexes

Consider a collection events that contains documents with the fields username and date.

The following index can support both these sort operations:

```
db.events.createIndex( { "username" : 1, "date" : -1 } )
```

Supports the following two sort operations

```
db.events.find().sort( { username: 1, date: -1 } )
```

```
db.events.find().sort( { username: -1, date: 1 } )
```

However, the above index cannot support sorting by ascending username values and then by ascending date values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

Sort Order

The sort operations help retrieve documents based on the sort order in an index. Following are the characteristics of a sort order:

- If sorted documents cannot be obtained from an index, the results will get sorted in the memory.
- Sort operations executed using an index show better performance than those executed without using an index.
- Sort operations performed without an index gets terminated after exhausting 32 MB of memory.
- Indexes store field references in the ascending or descending sort order.
- Sort order is not important for single-field indexes because MongoDB can traverse the index in either direction.
- Sort order is important for compound indexes because it helps determine if the index can support a sort operation.

Multi key Index

When indexing a field containing an array value, MongoDB creates separate index entries for each array component. MongoDB lets you construct multi-key indexes for arrays holding scalar values, such as strings, numbers, and nested documents.

To create a multi-key index, use the method given below.

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

If the indexed field contains an array, MongoDB automatically decides to:

- Create a multi-key index
- Not create a multi-key index

Unique Multikey Index

For unique indexes, the unique constraint applies across separate documents in the collection rather than within a single document.

Because the unique constraint applies to separate documents, for a unique multikey index, a document may have array elements that result in repeating index key values as long as the index key values for that document do not duplicate those of another document.

Compound Multi-Key Indexes

- In compound multi-key indexes, each indexed document can have maximum one indexed field with an array value.
- When more than one field contain an array value, compound multi-key indexes cannot be created.
- A shard key index and a hashed index cannot be a multikey index.

Hashed Indexes

The hashing function does the following:

- Combines all embedded documents.
- Computes hashes for all field values.
- Supports sharding, uses a hashed shard key to shard a collection, and ensures an even data distribution.
- Supports equality queries, however, range queries are not supported.

Cannot create unique or compound index by taking a field whose type is hashed. However, you can create a hashed and non-hashed index for the same field.

To create a hashed index, use the operation given below.

```
db.items.createIndex( { item: "hashed" } )
```

TTL Indexes

Total Time to Live (TTL) indexes can be created by combining `db.collection.createIndex()` and “`expireAfterSeconds`”.

To create a TTL index, use the operation given below.

```
db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
```

TTL indexes have the following limitations:

- Not supported by compound indexes and the `_id` field does not support TTL indexes
- Cannot be created on a capped collection
- Does not allow `createIndex()` to change the value of “`expireAfterSeconds`” of an existing index

To change a non-TTL single-field index to a TTL index, drop the index and recreate the index with the “`expireAfterSeconds`” option.

TTL collections

- Document must have a BSON UTC Date field

```
{  
  a: 12,  
  b: 455,  
  c: 2323,  
  status: ISODate("2013-08-08T12:00:00Z")  
}
```

- `db.collection.ensureIndex(
 {"status": 1},
 {expireAfterSeconds: 3600}
)`

Documents are removed after 'expireAfterSeconds' seconds

Unique Indexes

- Unique indexes can be created by using the `db.collection.createIndex()` method and set the `unique` option to `true`.
- To create a unique index on the `item` field of the `items` collection, execute the operation given below.

```
db.items.createIndex( { "item": 1 }, { unique: true } )
```

- If a unique index has no value, the index stores a null value for the document.
- Because of this `unique` constraint, MongoDB permits only one document without the indexed field.
- For more than one document with a valueless or missing indexed field, the index build process fails.
- To filter null values in a document and avoid error, combine the `unique` constraint with the `sparse` index.

Sparse Indexes

Sparse indexes manage documents with indexed fields and ignores documents which do not contain any index field.

To create a sparse index, use the `db.collection.createIndex()` method and set the `sparse` option to `true`.

```
db.addresses.createIndex( { "xmpp_id": 1 }, { sparse: true } )
```

When a sparse index returns an incomplete index, then MongoDB does not use that index unless it is specified in the `hint` method.

```
{xmpp_id : { $exists: false } }
```

An index combining sparse and unique indexes does not allow duplicate field values for a single field.

Sparse indexes

- `db.collection.createIndex({b: 1}, {sparse: true})`
- `db.collection.createIndex(
 {b: 1},
 {sparse: true, unique: true}
)`

Missing fields are stored as null(s) in the index

1 - `db.collection.insert({a: 12})`

2 - `db.collection.insert({a: 122, b: 12})`

Attention with sort

Geospatial indexes

- `db.collection.ensureIndex({loc:"2dsphere"})`
`{`
 `name:"phplx",`
 `loc:{ type:"Point", coordinates: [-9.145858, 38.731103]}`
`}`
- `db.collection.find({`
 `loc: {$near: {`
 `$geometry: {`
 `$type:"Point",`
 `coordinates: [-9.145858, 38.731103]`
 `}}})`

Text indexes

- Support text search queries on string content.
- Can include any field whose value is a string or an array of string elements.
- A collection can have at most one text index.
- 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
- Can index multiple fields for the text index
- A compound index can include text index keys in combination with ascending/descending index keys.
- Text indexes are always sparse and ignore the sparse option.
- If a document lacks a text index field (or the field is null or an empty array), MongoDB does not add an entry for the document to the text index.
- For inserts, MongoDB inserts the document but does not add to the text index.
- For a compound index that includes a text index key along with keys of other types, only the text index field determines whether the index references a document.

Text indexes

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

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


Specify Weights

- 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.
- 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.

Wildcard Text Indexes

- When creating a text index on multiple fields, you can also use the wildcard specifier (\$**).
- 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" })`
- This index allows for text search on all fields with 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.

Text indexes

- Wildcard text indexes are text indexes on multiple fields.
- As such, you can assign weights to specific fields during index creation to control the ranking of the results.
- Wildcard text indexes, as with all text indexes, can be part of a compound indexes.
- For example, the following creates a compound index on the field a as well as the wildcard specifier:
- `db.collection.createIndex({ a: 1, "$*": "text" })`
- As with all compound text indexes, since the a precedes the text index key, in order to perform a \$text search with this index, the query predicate must include an equality match conditions a.

Text indexes - Restrictions

- One Text Index Per Collection -A collection can have at most one text index.
- Text Search and Hints -Cannot use hint() if the query includes a \$text query expression.
- Text Index and Sort -Sort operations cannot obtain sort order from a text index, even from a compound text index; i.e. sort operations cannot use the ordering in the text index.
- Compound Index -A compound index can include a text index key in combination with ascending/descending index keys.
- Compound indexes have the following restrictions:
- A compound text index cannot include any other special index types, such as multi-key or geospatial index fields.
- If the compound text index includes keys preceding the text index key, to perform a \$text search, the query predicate must include equality match conditions on the preceding keys.
- When creating a compound text index, all text index keys must be listed adjacently in the index specification document

Text indexes —

Storage Requirements and Performance Costs

- Text indexes can be large.
- Contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a text index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large text index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors.
- Text indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, text indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.

Text indexes

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


Text indexes

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

```
db.inventory.createIndex(  
  {  
    dept: 1,  
    description: "text"  
  }  
)
```

Limitations

- collections can not have > 64 indexes
- queries can use more than one index
- indexes have storage requirements, and impacts insert/update speed to some degree
-

Index Creation

- During index creation, operations on a database are blocked and the database becomes unavailable for any read or write operation.
- The read or write operations on the database queue allow the index building process to complete.
- To make MongoDB available even during an index build process, use the command given below.

```
db.items.createIndex( {item:1},{background: true})
```

```
db.items.createIndex({category:1}, {sparse: true, background:true})
```

Finding the indexes in a collection

can use `getIndexes()` method to find all the indexes created on a collection.

Syntax :

```
db.collection_name.getIndexes()
```

To get the indexes of `studentdata` collection,

```
> db.studentdata.getIndexes()
```

Drop indexes in a collection

Can either drop a particular index or all the indexes.

Dropping a specific index:

```
db.collection_name.dropIndex({index_name: 1})
```

To drop the index `student_name` field in the collection `studentdata`.

```
db.studentdata.dropIndex({student_name: 1})
```

Dropping all the indexes:

To drop all the indexes of a collection, use `dropIndexes()` method.

Syntax of `dropIndexes()` method:

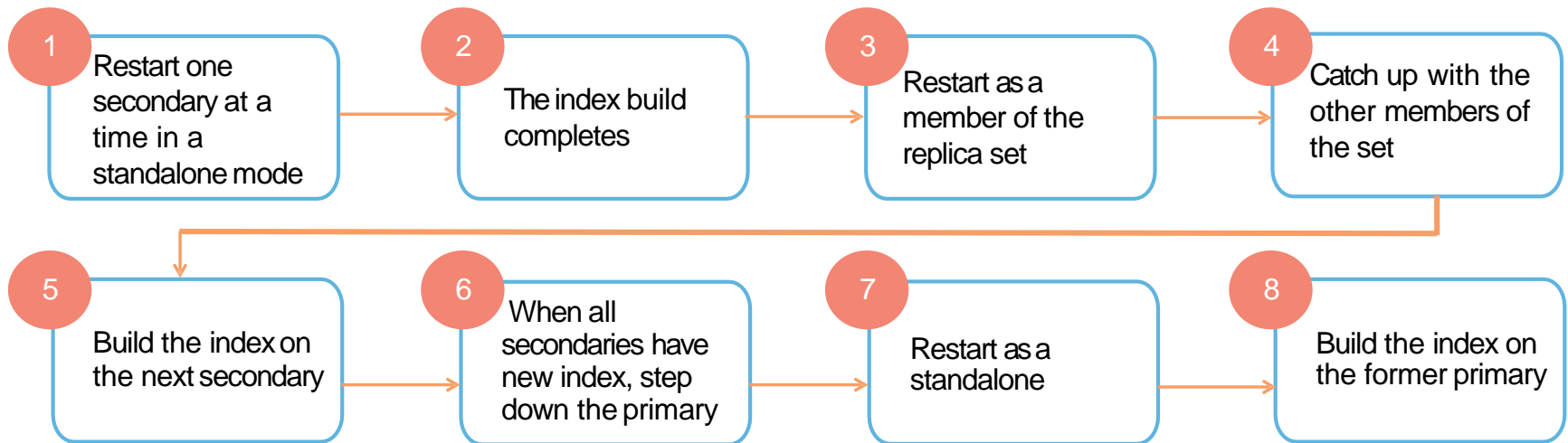
```
db.collection_name.dropIndexes()
```

To drop all the indexes of `studentdata` collection.

```
db.studentdata.dropIndexes()
```


Index Creation on Replica Set

Background index operations on a secondary replica set begin after the index build completes in the primary. To build large indexes on secondaries perform the following steps:



Use the command below to specify a name for an index.

```
db.products.createIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

Rebuild Indexes

To rebuild all indexes of a collection, use the `db.collection.reIndex` method.

This will drop all indexes including `_id` and rebuild all indexes in a single operation.

Can use the following commands when rebuilding indexes:

- **`db.currentOp()`**—Type this command in the mongo shell to view the indexing process status.
- **`db.killOp()`**—Type this command in the mongo shell to abort an ongoing index build process.

Measure Index Use

Query performances indicate index usage. MongoDB provides the following methods to observe index use for your database:

- **The `explain()` method**—used to print information about query execution. Returns a document that explains the process and indexes used to return a query.
- **`db.collection.explain()` or the `cursor.explain()`**—helps measure index usages.

Control Index Use

To force MongoDB to use particular indexes for querying documents, you need to specify the index with the `hint()` method, which can be appended in the `find()` method.

The command given below queries a document whose `item` field value is “Book” and `available` field is “true”.

```
db.items.find({item:"Book",available:"true"}).hint({item:1})
```

To view the execution statistics for a specific index, use the query below.

```
db.items.find({item: "Book", available : true}).hint({item:1}).explain("executionStats")
```

```
db.items.explain("executionStats").find({item: "Book", available : true }).hint( { item:1 } )
```

To prevent MongoDB from using any index, specify the `$natural` operator to the `hint()` method.

```
db.items.find({item: "Book", available : true }).hint({$natural:1}).explain("executionStats")
```

Index Use Reporting

- MongoDB provides different metrics to report index use and operation. These metrics are printed using the following commands.
- `serverStatus`:
 - `scanned`: Displays the documents that MongoDB scans in the index to carry out the operation
 - `scanAndOrder`: A boolean that is true when a query cannot use the order of documents in the index for returning sorted results
- `collStats`:
 - `totalIndexSize`: Returns index size in bytes
 - `indexSizes`: Explains the size of the data allocated for an index
- `dbStats`:
 - `dbStats.indexes`: Contains a count of the total number of indexes across all collections in the database
 - `dbStats.indexSize`: The total size in bytes of all indexes created on this database

Optimizing Queries

Profiling operations

- `db.setProfilingLevel(level, slowms)`
 - `level`
 - 0 = profiler off
 - 1 = record ops longer than `slowms`
 - 2 = record all queries
- `db.system.profile.find()`

the profile collection is a capped collection and fixed in size


```

{
  "ts" : ISODate("2012-12-10T19:31:28.977Z"),
  "op" : "update",
  "ns" : "social.users",
  "query" : {
    "name" : "jane"
  },
  "updateobj" : {
    "$set" : {
      "likes" : [
        "basketball",
        "trekking"
      ]
    }
  },
  "nscanned" : 8,
  "moved" : true,
  "nmoved" : 1,
  "nupdated" : 1,
  "keyUpdates" : 0,
  "numYield" : 0,
  "lockStats" : {
    "timeLockedMicros" : {
      "r" : NumberLong(0),
      "w" : NumberLong(258)
    },
    "timeAcquiringMicros" : {
      "r" : NumberLong(0),
      "w" : NumberLong(7)
    }
  },
  "millis" : 0,

```


Hint a index

- we can tell the database what index to use

- `db.collection.find({b: {$gt: 120}})`
`.hint({a: 1})`

- or tell the database to not use an index

- `db.collection.find({b: {$gt: 120}})`
`.hint({$natural: 1})`

The query optimizer

- for each “type” of query, MongoDB periodically tries all useful indexes
- aborts the rest as soon as one plan win
- the winning plan is temporarily cached for each “type” of query

Mistakes

- trying to use multiple indexes

- `db.collection.ensureIndex({a: 1})`

- `db.collection.ensureIndex({b: 1})`

`// only one of the above indexes is used`

`db.collection.find({a: 3, b: 10})`

• compound indexes

• `db.collection.ensureIndex({a: 1, b: 1, c: 1})`

// can't use the index

`db.collection.find({c: 100})`

// but this can't

`db.collection.find({a: 10, c: 20})`

// and this ???

`db.collection.find({c: 100}).sort({a: 1});`

• low selectivity indexes

- `db.collection.distinct("a")`
`["java", "php", "c++"]`

```
db.collection.ensureIndex({a: 1})
```

```
// low selectivity provide little benefit  
db.collection.find({a: "php"})
```

- `db.collection.ensureIndex({a: 1, created: 1})`

```
// good  
db.collection.find({a: "php"}).sort({created: 1})
```


• regular expressions

- `db.collection.ensureIndex({a: 1})`

// left anchored regex queries can use index
`db.collection.find({a: /^php/})`

// but not generic regex
`db.collection.find({a: /php/})`

// or insensitive
`db.collection.find({a: /Php/i})`

• negation

• `db.collection.ensureIndex({a: 1})`

`// not equal`

`db.collection.find({a: {$ne: 'php'}})`

`// not in`

`db.collection.find({a: {$nin: ['java', 'c++']}})`

`// $not operador`

`db.collection.ensureIndex({a: {$not: 'c#'}})`

TO REMEMBER

choosing the right indexes is
one of the most important
things you can do in
MongoDB

TO REMEMBER

Applications may encounter reduced performance during index builds, including limited read/write access to the collection

TO REMEMBER

If a collection has both a compound index and an index on its prefix (e.g. { a: 1, b: 1 } and { a: 1 }), if neither index has a sparse or unique constraint, then can remove the index on the prefix (e.g. { a: 1 }). MongoDB will use the compound index in all of the situations that it would have used the prefix index.

Index Intersection

- MongoDB can use the intersection of multiple indexes to fulfill queries.
- In general, each index intersection involves two indexes;
- MongoDB can employ multiple/nested index intersections to resolve a query
- To determine if MongoDB used index intersection, run `explain()`; the results of `explain()` will include either an `AND_SORTED` stage or an `AND_HASH` stage.

Index Intersection

To illustrate index intersection, consider a collection orders that has the following indexes:

```
{ qty: 1 }  
{ item: 1 }
```

MongoDB can use the intersection of the two indexes to support the following query:

```
db.orders.find( { item: "abc123", qty: { $gt: 15 } } )
```

Index Prefix Intersection

- With index intersection, MongoDB can use an intersection of either the entire index or the index prefix.
- An index prefix is a subset of a compound index, consisting of one or more keys starting from the beginning of the index.

Consider a collection orders with the following indexes:

```
{ qty: 1 }
```

```
{ status: 1, ord_date: -1 }
```

- To fulfill the following query which specifies a condition on both the qty field and the status field, MongoDB can use the intersection of the two indexes:

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } )
```

Index Intersection & Compound Indexes

- Index intersection does not eliminate the need for creating compound indexes.
- However, because both the **list order** (i.e. the order in which the keys are listed in the index) and **the sort order** (i.e. ascending or descending), matter in compound indexes, a compound index may not support a query condition that does not include the index prefix keys or that specifies a different sort order.
- For example, if a collection orders has the following compound index, with the status field listed before the ord_date field:

```
{ status: 1, ord_date: -1 }
```

Index Intersection & Compound Indexes

The compound index can support the following queries:

```
db.orders.find( { status: { $in: ["A", "P" ] } } )
```

```
db.orders.find(  
  {  
    ord_date: { $gt: new Date("2014-02-01") },  
    status: { $in: [ "P", "A" ] }  
  }  
)
```

But not the following two queries:

```
db.orders.find( { ord_date: { $gt: new Date("2014-02-01") } } )
```

```
db.orders.find( { } ).sort( { ord_date: 1 } )
```

However, if the collection has two separate indexes:

```
{ status: 1 }
```

```
{ ord_date: -1 }
```

The two indexes can, either individually or through index intersection, support all four aforementioned queries.

The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system.

Index Intersection and Sort

Index intersection does not apply when the sort() operation requires an index completely separate from the query predicate.

For example, the orders collection has the following indexes:

```
{ qty: 1 }  
{ status: 1, ord_date: -1 }  
{ status: 1 }  
{ ord_date: -1 }
```

MongoDB cannot use index intersection for the following query with sort:

```
db.orders.find( { qty: { $gt: 10 } } ).sort( { status: 1 } )
```

That is, MongoDB does not use the { qty: 1 } index for the query, and the separate { status: 1 } or the { status: 1, ord_date: -1 } index for the sort.

Index Intersection and Sort

However, MongoDB can use index intersection for the following query with sort since the index { status: 1, ord_date: -1 } can fulfill part of the query predicate.

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } ).sort( { ord_date: -1 } )
```

Index Key Limit

- For MongoDB 2.6 through MongoDB versions with fCV set to "4.0" or earlier, the total size of an index entry, which can include structural overhead depending on the BSON type, must be less than 1024 bytes.
- Starting in version 4.2, MongoDB removes the Index Key Limit for featureCompatibilityVersion (fCV) set to "4.2" or greater.

When the Index Key Limit applies:

- MongoDB will not create an index on a collection if the index entry for an existing document exceeds the index key limit.
- Reindexing operations will error if the index entry for an indexed field exceeds the index key limit. Reindexing operations occur as part of the compact command as well as the `db.collection.reIndex()` method.
- Because these operations drop all the indexes from a collection and then recreate them sequentially, the error from the index key limit prevents these operations from rebuilding any remaining indexes for the collection.
- MongoDB will not insert into an indexed collection any document with an indexed field whose corresponding index entry would exceed the index key limit, and instead, will return an error. Previous versions of MongoDB would insert but not index such documents.
- Updates to the indexed field will error if the updated value causes the index entry to exceed the index key limit.

When the Index Key Limit applies:

- If an existing document contains an indexed field whose index entry exceeds the limit, any update that results in the relocation of that document on disk will error.
- mongorestore and mongoimport will not insert documents that contain an indexed field whose corresponding index entry would exceed the index key limit.
- In MongoDB 2.6, secondary members of replica sets will continue to replicate documents with an indexed field whose corresponding index entry exceeds the index key limit on initial sync but will print warnings in the logs.
- Secondary members also allow index build and rebuild operations on a collection that contains an indexed field whose corresponding index entry exceeds the index key limit but with warnings in the logs.
- With mixed version replica sets where the secondaries are version 2.6 and the primary is version 2.4, secondaries will replicate documents inserted or updated on the 2.4 primary, but will print error messages in the log if the documents contain an indexed field whose corresponding index entry exceeds the index key limit.
- For existing sharded collections, chunk migration will fail if the chunk has a document that contains an indexed field whose index entry exceeds the index key limit.

Index Name Length

- In previous versions of MongoDB or MongoDB versions with fCV set to "4.0" or earlier, fully qualified index names, which include the namespace and the dot separators (i.e. <database name>.<collection name>.\$<index name>), cannot be longer than 127 bytes.
- By default, <index name> is the concatenation of the field names and index type.
- Can explicitly specify the <index name> to the createIndex() method to ensure that the fully qualified index name does not exceed the limit.
- Starting in version 4.2, MongoDB removes the Index Name Length Limit for MongoDB versions with featureCompatibilityVersion (fCV) set to "4.2" or greater.

Restrictions

Number of Indexes per Collection

A single collection can have no more than 64 indexes.

Restrictions

Number of Indexed Fields in a Compound Index

Can be no more than 32 fields in a compound index.

Restrictions

Queries cannot use both text and Geospatial Indexes

Cannot combine the \$text query, which requires a special text index, with a query operator that requires a different type of special index. For example you cannot combine \$text query with the \$near operator

Considerations

- Each index requires at least 8 kB of data space.
- Adding an index has some negative performance impact for write operations.
- For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes.
- Collections with high read-to-write ratio often benefit from additional indexes. Indexes do not affect un-indexed read operations.
- When active, each index consumes disk space and memory. This usage can be significant and should be tracked for capacity planning, especially for concerns over working set size.

Index Builds on Populated Collections

- MongoDB index builds against a populated collection require an exclusive read-write lock against the collection.
- Operations that require a read or write lock on the collection must wait until the mongod releases the lock.
- MongoDB 4.2 uses an optimized build process that only holds the exclusive lock at the beginning and end of the index build.
- The rest of the build process yields to interleaving read and write operations.

Summary of build process

Initialization

- The mongod takes an exclusive lock against the collection being indexed. This blocks all read and write operations to the collection until the mongod releases the lock. Applications cannot access the collection during this time.

Data Ingestion and Processing

- The mongod releases all locks taken by the index build process before taking a series of intent locks against the collection being indexed. Applications can issue read and write operations against the collection during this time.

Cleanup

- The mongod releases all locks taken by the index build process before taking an exclusive lock against the the collection being indexed. This blocks all read and write operations to the collection until the mongod releases the lock. Applications cannot access the collection during this time.

Completion

- The mongod marks the index as ready to use and releases all locks taken by the index build process.

Build Process Steps

Stage 1: Lock

- The mongod obtains an exclusive X lock on the the collection being indexed.
- This blocks all read and write operations on the collection, including the application of any replicated write operations or metadata commands that target the collection.
- The mongod does not yield this lock.

Build Process Steps

Stage 2: Initialisation

- The mongod creates three data structures at this initial state:
- The initial index metadata entry.
- A temporary table (“side writes table”) that stores keys generated from writes to the collection being indexed during the build process.
- A temporary table (“constraint violation table”) for all documents that may cause a duplicate-key constraint violation.

Build Process Steps

Stage 3: Lock

- The mongod downgrades the exclusive X collection lock to an intent exclusive IX lock.
- The mongod periodically yields this lock to interleaving read and write operations.

Build Process Steps

Stage 4: Scan Collection

- For each document in the collection, the mongod generates a key for that document and dumps the key into an external sorter.
- If the mongod encounters a duplicate key error while generating a key during the collection scan, it stores that key in the constraint violation table for later processing.
- If the mongod encounters any other error while generating a key, the build fails with an error.
- Once the mongod completes the collection scan, it dumps the sorted keys into the index.

Build Process Steps

Stage 5: Process Side Writes Table

- The mongod drains the side write table using first-in-first-out priority.
- If the mongod encounters a duplicate key error while processing a key in the side write table, it stores that key in the constraint violation table for later processing.
- If the mongod encounters any other error while processing a key, the build fails with an error.
- For each document written to the collection during the build process, the mongod generates a key for that document and stores it in the side write table for later processing.
- The mongod uses a snapshot system to set a limit to the number of keys to process.

Build Process Steps

Stage 6: Lock

- The mongod upgrades the intent exclusive IX lock on the collection to a shared S lock.
- This blocks all write operations to the collection, including the application of any replicated write operations or metadata commands that target the collection.

Build Process Steps

Stage 7: Finish Processing Temporary Side Writes Table

- mongod continues draining remaining records in the side writes table. The mongod may pause replication during this stage.
- If the mongod encounters a duplicate key error while processing a key in the side write table, it stores that key in the constraint violation table for later processing.
- If the mongod encounters any other error while processing a key, the build fails with an error.

Build Process Steps

Stage 8: Lock

- The mongod upgrades the shared S lock on the collection to an exclusive X lock on the collection.
- This blocks all read and write operations on the collection, including the application of any replicated write operations or metadata commands that target the collection.
- mongod does not yield this lock

Build Process Steps

Stage 9: Drop Side Write Table

- The mongod applies any remaining operations in the side writes table before dropping it.
- If the mongod encounters a duplicate key error while processing a key in the side write table, it stores that key in the constraint violation table for later processing.
- If the mongod encounters any other error while processing a key, the build fails with an error.
- At this point, the index includes all data written to the collection.

Build Process Steps

Stage 10: Process Constraint Violation Table

- The mongod drains the constraint violation table using first-in-first-out priority. The mongod then drops the table.
- If any key in the constraint violation table still produces a duplicate key error, the mongod aborts the build and throws an error.
- The mongod drops the constraint violation table once it is drained or if it encounters a duplicate key violation during processing.

Build Process Steps

Stage 11: Mark the Index as Ready

The mongod updates the index metadata to mark the index as ready for use.

Stage 12: Lock

The mongod releases the X lock on the collection.

Foreground vs Background Builds

- Previous versions of MongoDB supported building indexes either in the foreground or background.
- Foreground index builds were fast and produced more efficient index data structures, but required blocking all read-write access to the parent database of the collection being indexed for the duration of the build.
- Background index builds were slower and had less efficient results, but allowed read-write access to the database and its collections during the build process.

Build indexes in 4.2

- MongoDB 4.2 index builds obtain an exclusive lock on only the collection being indexed during the start and end of the build process to protect metadata changes.
- The rest of the build process uses the yielding behavior of background index builds to maximize read-write access to the collection during the build. 4.2 index builds still produce efficient index data structures despite the more permissive locking behavior.
- MongoDB 4.2 index build performance is at least on par with background index builds.
- For workloads with few or no updates received during the build process, 4.2 index builds can be as fast as a foreground index build on that same data.

Monitor index builds

- Use `db.currentOp()` to monitor the progress of ongoing index builds.

Constraint Violations During Index Build

- For indexes that enforce constraints on the collection, such as unique indexes, the mongod checks all pre-existing and concurrently-written documents for violations of those constraints after the index build completes.
- Documents that violate the index constraints can exist during the index build.
- If any documents violate the index constraints at the end of the build, the mongod terminates the build and throws an error

Constraint Violations During Index Build

- For example, consider a populated collection inventory.
- An administrator wants to create a unique index on the `product_sku` field.
- If any documents in the collection have duplicate values for `product_sku`, the index build can still start successfully.
- If any violations still exist at the end of the build, the mongod terminates the build and throws an error.
- Similarly, an application can successfully write documents to the inventory collection with duplicate values of `product_sku` while the index build is in progress.
- If any violations still exist at the end of the build, the mongod terminates the build and throws an error.

Mitigate Violations

To mitigate the risk of index build failure due to constraint violations:

- Validate that no documents in the collection violate the index constraints.
- Stop all writes to the collection from applications that cannot guarantee violation-free write operations.

Build Failure and Recovery

Interrupted Index Builds on Standalone mongod

- If the mongod shuts down during the index build, the index build job and all progress is lost.
- Restarting the mongod does not restart the index build.
- Must re-issue the `createIndex()` operation to restart the index build.

Build Failure and Recovery

Interrupted Index Builds on a Primary mongod

- If the primary shuts down or steps down during the index build, the index build job and all progress is lost.
- Restarting the mongod does not restart the index build.
- Must re-issue the `createIndex()` operation to restart the index build.

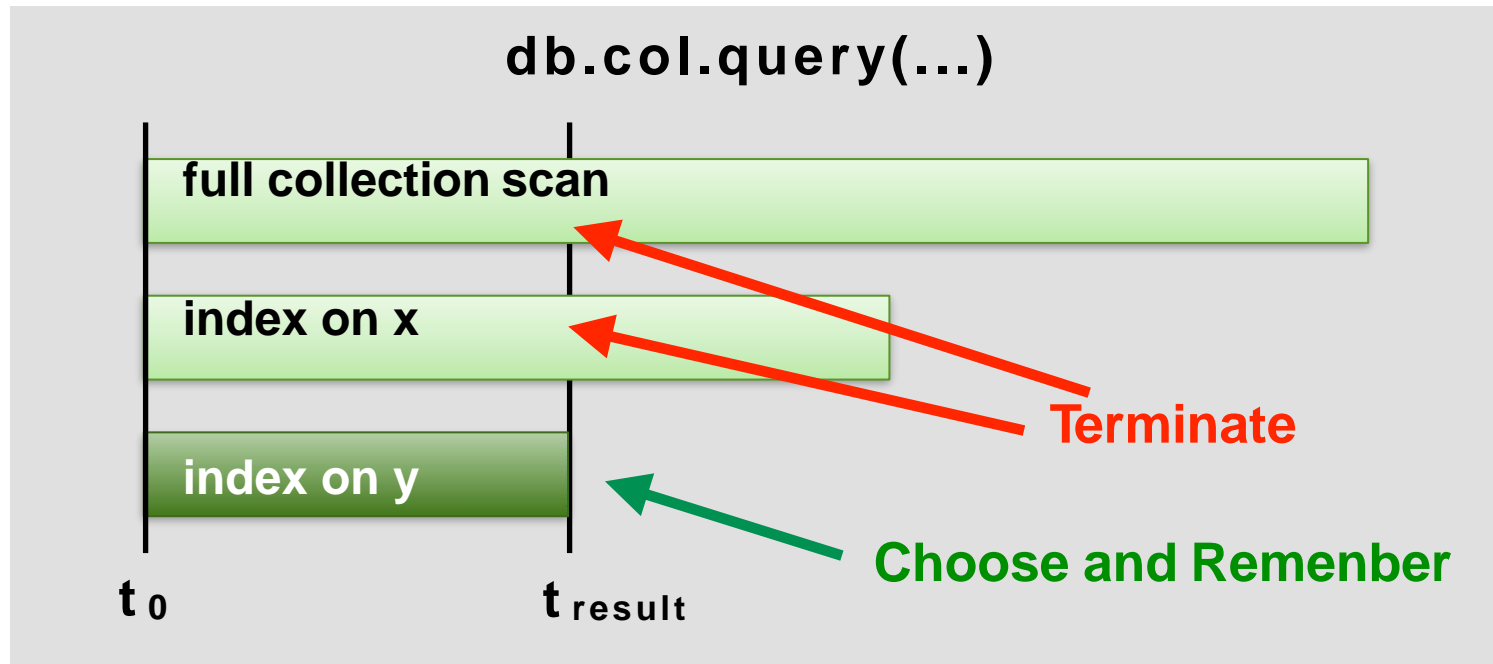
Interrupted Index Builds on a Secondary mongod

- If a secondary shuts down during the index build, the index build job is persisted.
- Restarting the mongod recovers the index build and restarts it from scratch.

Index monitoring

The Query Optimizer

Chooses the most efficient query plan.



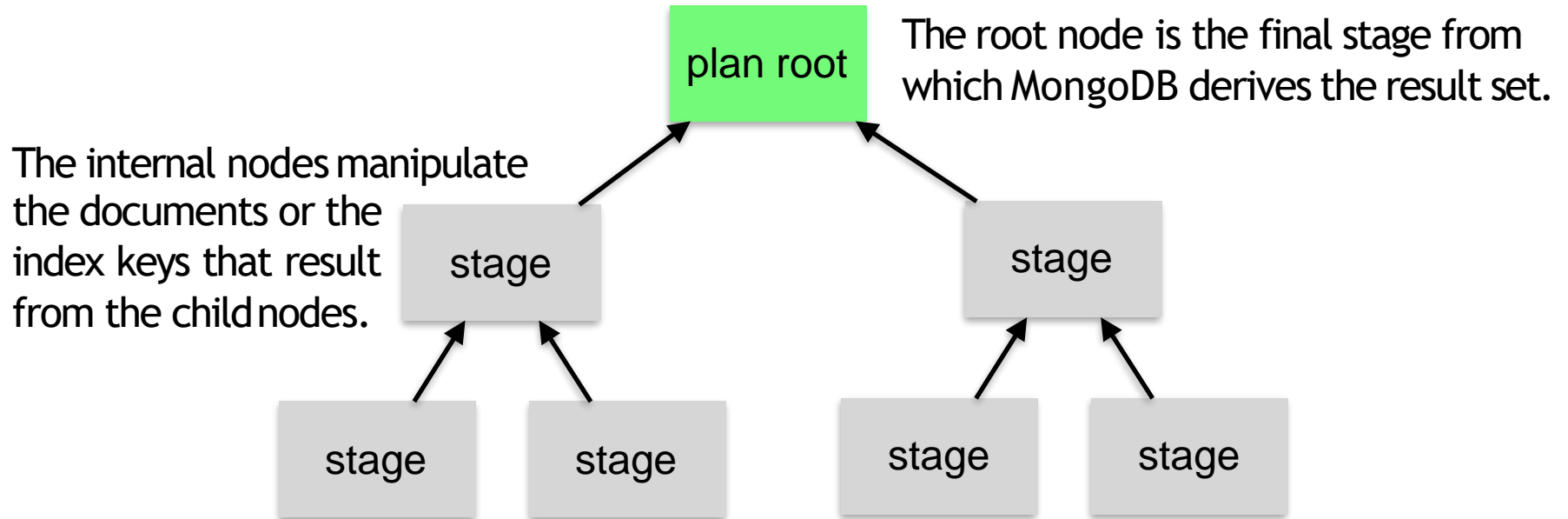
Information on query plans
and their execution statistics:

```
db.col.query.explain()
```

```
db.col.query.explain()
```

The explain results present the query plans as a tree of stages.

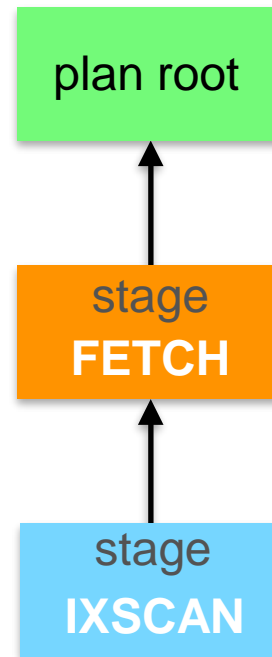
Each stage passes its results (i.e. documents or index keys) to the parent node.



Leaf nodes access the collections or the indices.

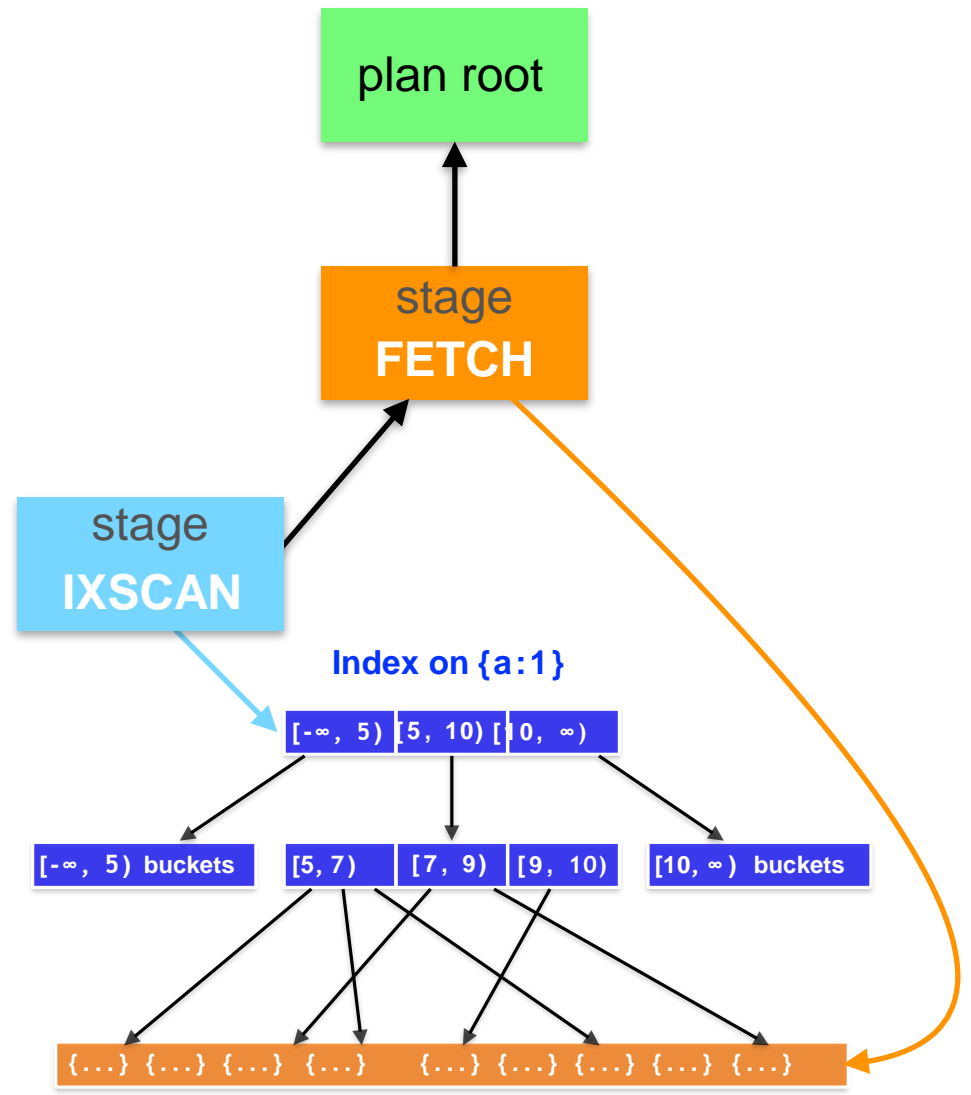
```
db.test.find({a:5}).explain()
```

```
{
  "queryPlanner" : {
    [...],
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "a" : 5
        },
        "indexName" : "a_1",
        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "a" : [
            "[5.0, 5.0]"
          ]
        }
      }
    }
  },
  [...]
}
```




```
db.test.find({a:5}).explain()
```

```
{
  "queryPlanner" : {
    [...],
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "a" : 5
        },
        "indexName" : "a_1",
        "isMultiKey" : false,
        "direction" : "forward",
        "indexBounds" : {
          "a" : [
            "[5.0, 5.0]"
          ]
        }
      }
    }
  },
  [...]
}
```



Explain Levels

queryPlanner

"Which plan will MongoDB choose to run my query?"

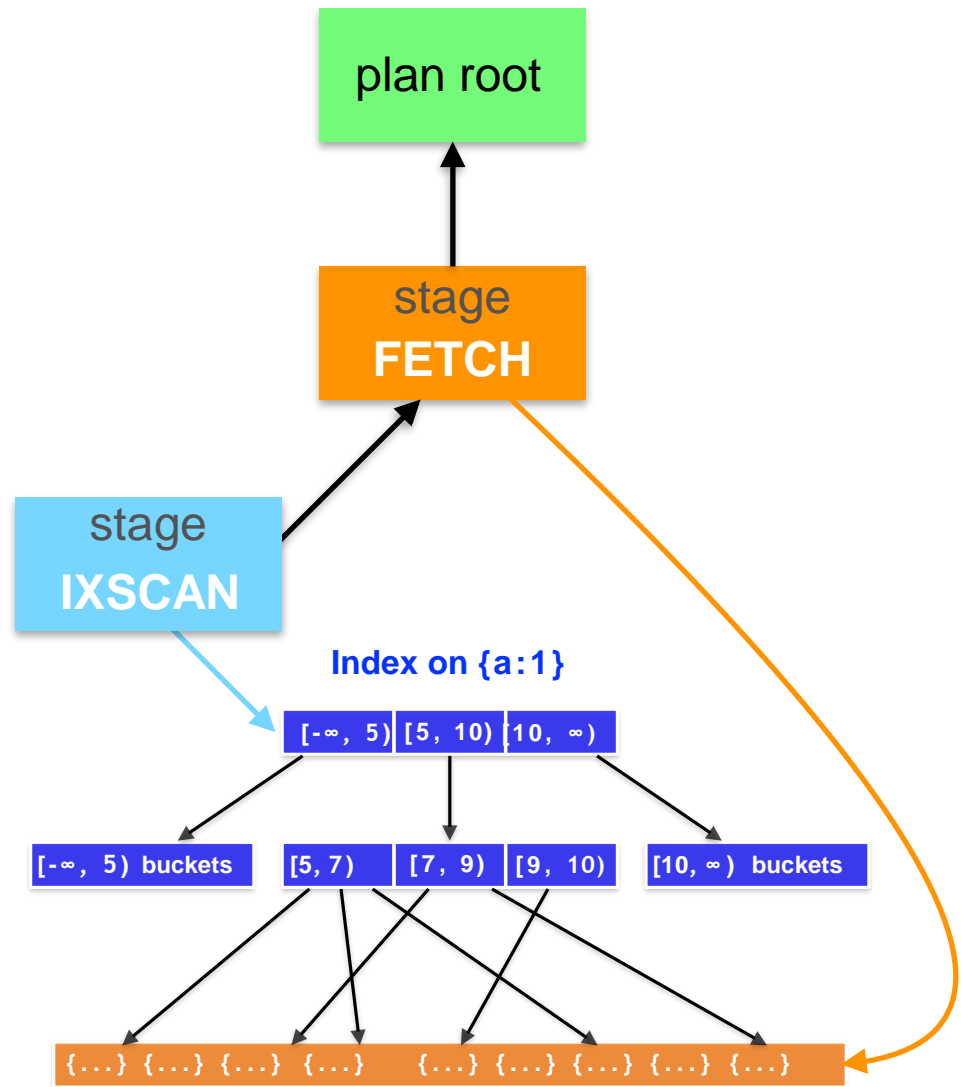
executionStats

"How is my query performing?"

allPlansExecution

"I want as much information as possible to diagnose a slow query."

```
db.test.find({a:5}).explain()
```



```
db.test.find({a:5}).explain("executionStats")
```

```
{
  [...],
  "executionStats" : {

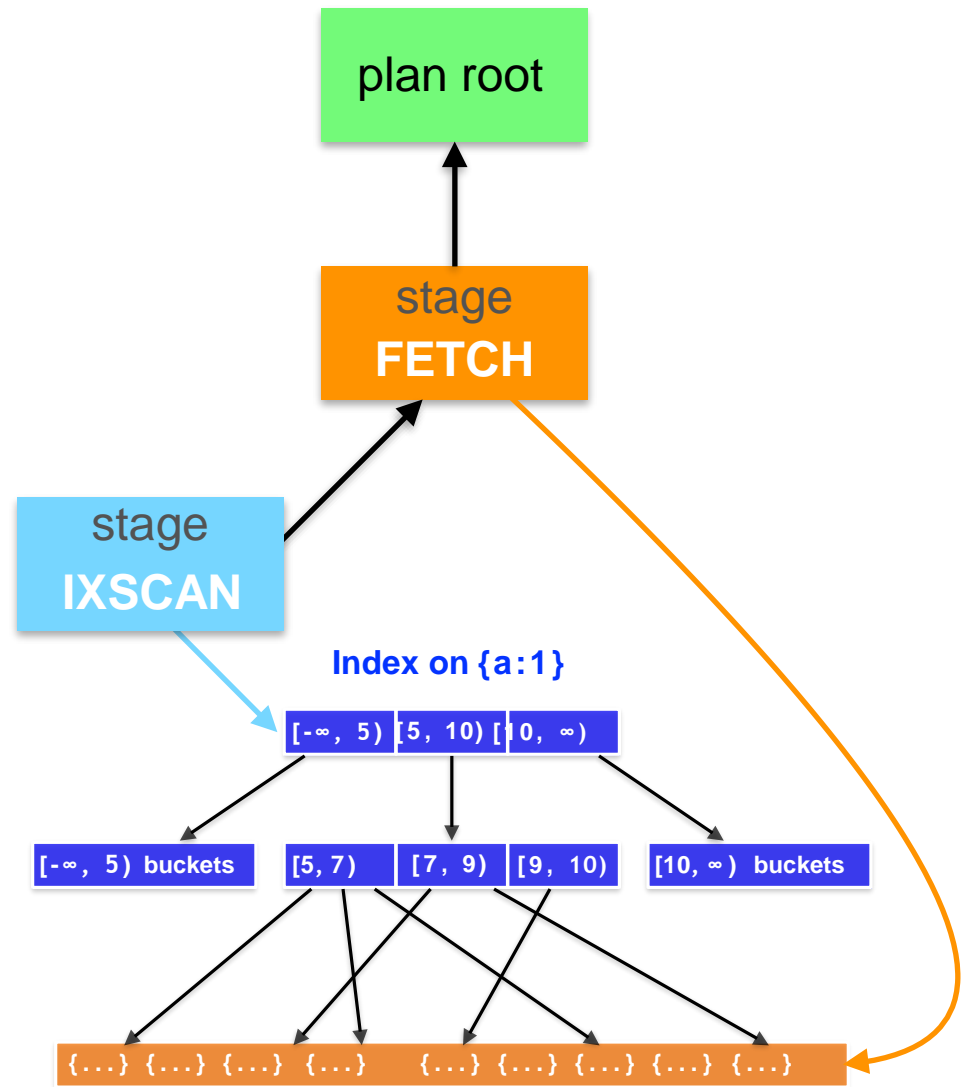
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1,

    "executionStages" : {

      "stage" : "FETCH",
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 0,
      "works" : 2,
      "advanced" : 1,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "docsExamined" : 1,
      "alreadyHasObj" : 0,

      "inputStage" : {

        "stage" : "IXSCAN",
```



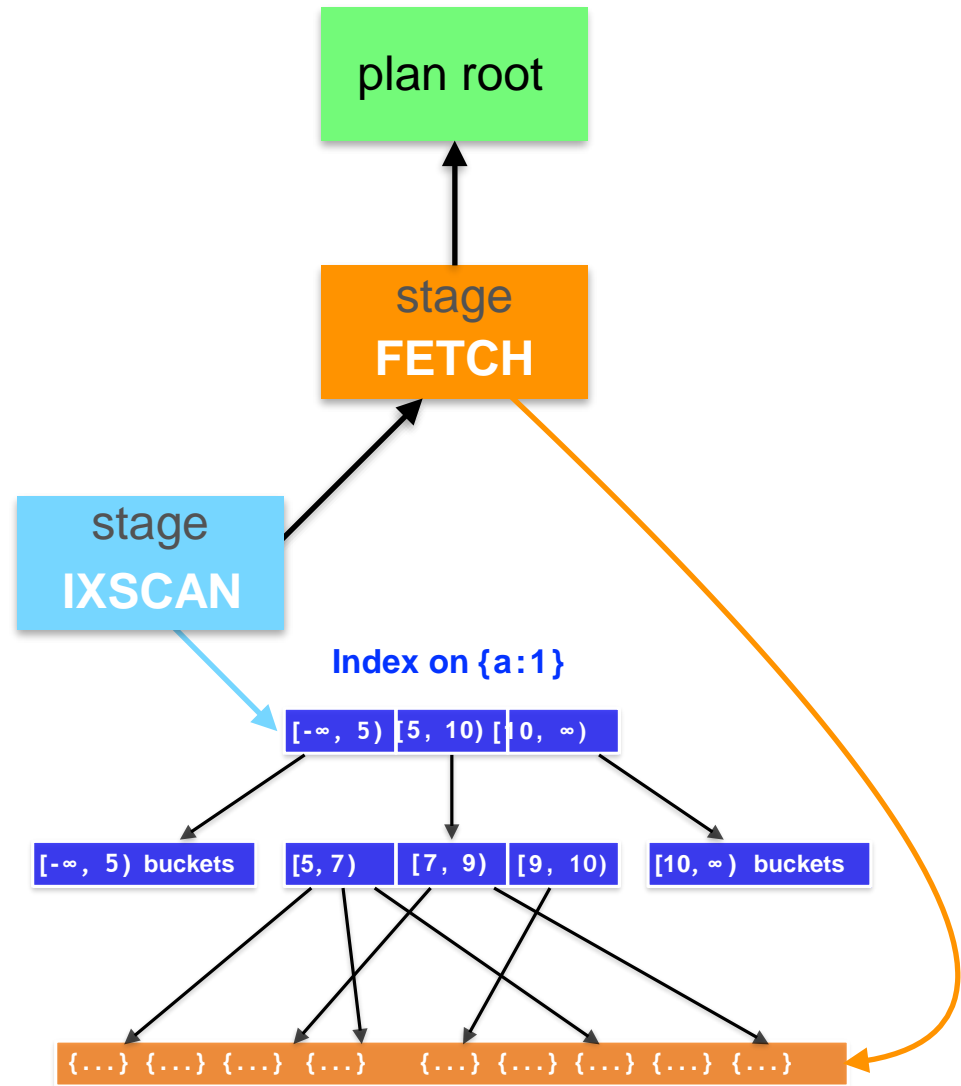
```
db.test.find({a:5}).explain("executionStats")
```

```
"executionStages" : {
```

```
"stage" : "FETCH",  
"nReturned" : 1,  
"executionTime  
MillisEstimate" : 0,  
"works" : 2,  
"advanced" : 1,  
"needTime" : 0,  
"needFetch" : 0,  
"saveState" : 0,  
"restoreState" : 0,  
"isEOF" : 1,  
"invalidates" : 0,  
"docsExamined" : 1,  
"alreadyHasObj" : 0,
```

```
"inputStage" : {
```

```
"stage" : "IXSCAN",  
"nReturned" : 1,  
"executionTime  
MillisEstimate" : 0,  
"works" : 1,  
"advanced" : 1,  
"needTime" : 0,  
"needFetch" : 0,  
"saveState" : 0,  
"restoreState" : 0,  
"isEOF" : 1,
```



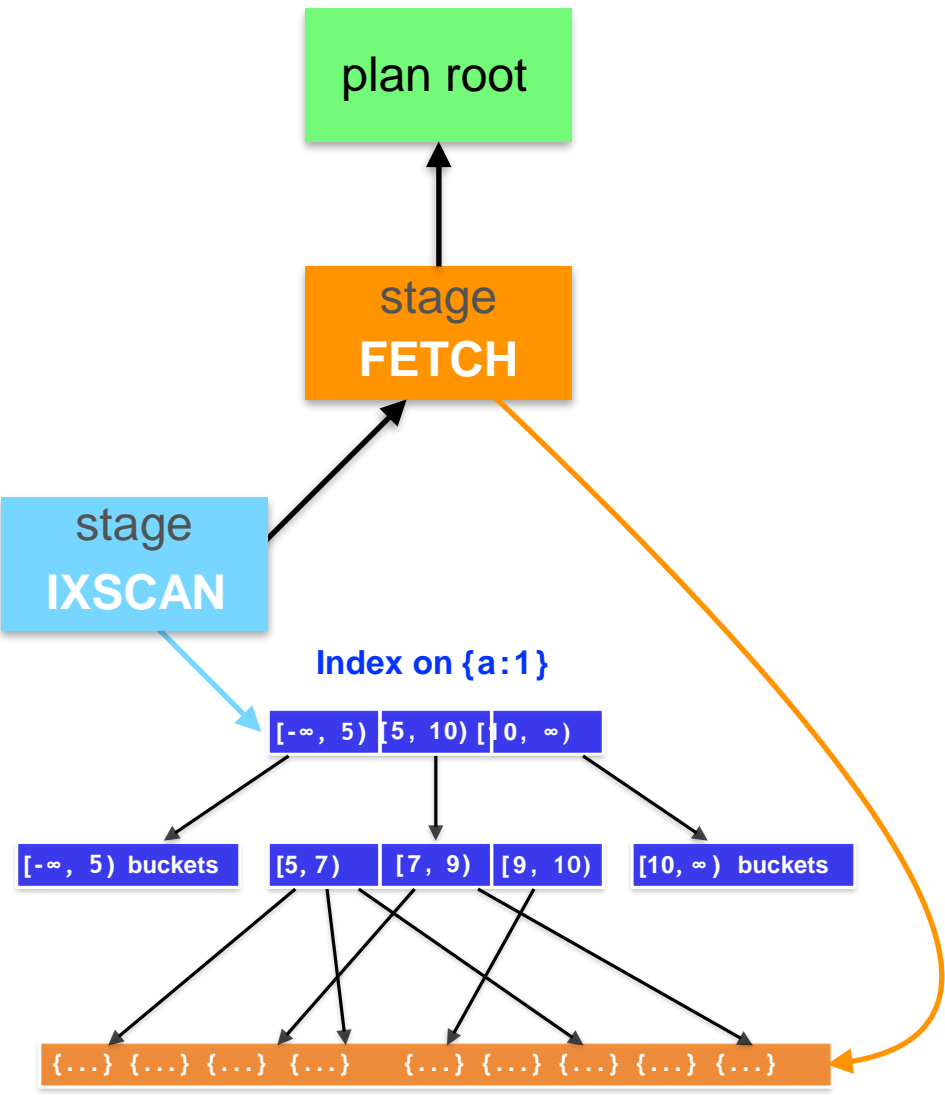
```
db.test.find({a:5}).explain("executionStats")
```

```
"inputStage" : {
  "stage" : "IXSCAN",
  "nReturned" : 1,
  "executionTime
  MillisEstimate" : 0,
  "works" : 1,
  "advanced" : 1,
  "needTime" : 0,
  "needFetch" : 0,
  "saveState" : 0,
  "restoreState" : 0,
  "isEOF" : 1,
  "invalidates" : 0,
  "keyPattern" : {
    "a" : 5
  },
  "indexName" : "a_1",
  "isMultiKey" : false,
  "direction" : "forward",
  "indexBounds" : {
    "a" : [
      "[5.0, 5.0]"
    ]
  },
  "keysExamined" : 1,
  "dupsTested" : 0,
  "dupsDropped" : 0,
  "seenInvalidated" : 0,
  "matchTested" : 0
}
```

```

"stage" : "IXSCAN",
"nReturned" : 1,
"executionTime
MillisEstimate" : 0,
"works" : 1,
"advanced" : 1,
"needTime" : 0,
"needFetch" : 0,
"saveState" : 0,
"restoreState" : 0,
"isEOF" : 1,
"invalidates" : 0,
"keyPattern" : {
  "a" : 5
},
"indexName" : "a_1",
"isMultiKey" : false,
"direction" : "forward",
"indexBounds" : {
  "a" : [
    "[5.0, 5.0]"
  ]
},
"keysExamined" : 1,
"dupsTested" : 0,
"dupsDropped" : 0,
"seenInvalidated" : 0,
"matchTested" : 0

```




```
db.test.find({a:5}).explain("executionStats")
```

```
{
  [...],
  "executionStats" : {

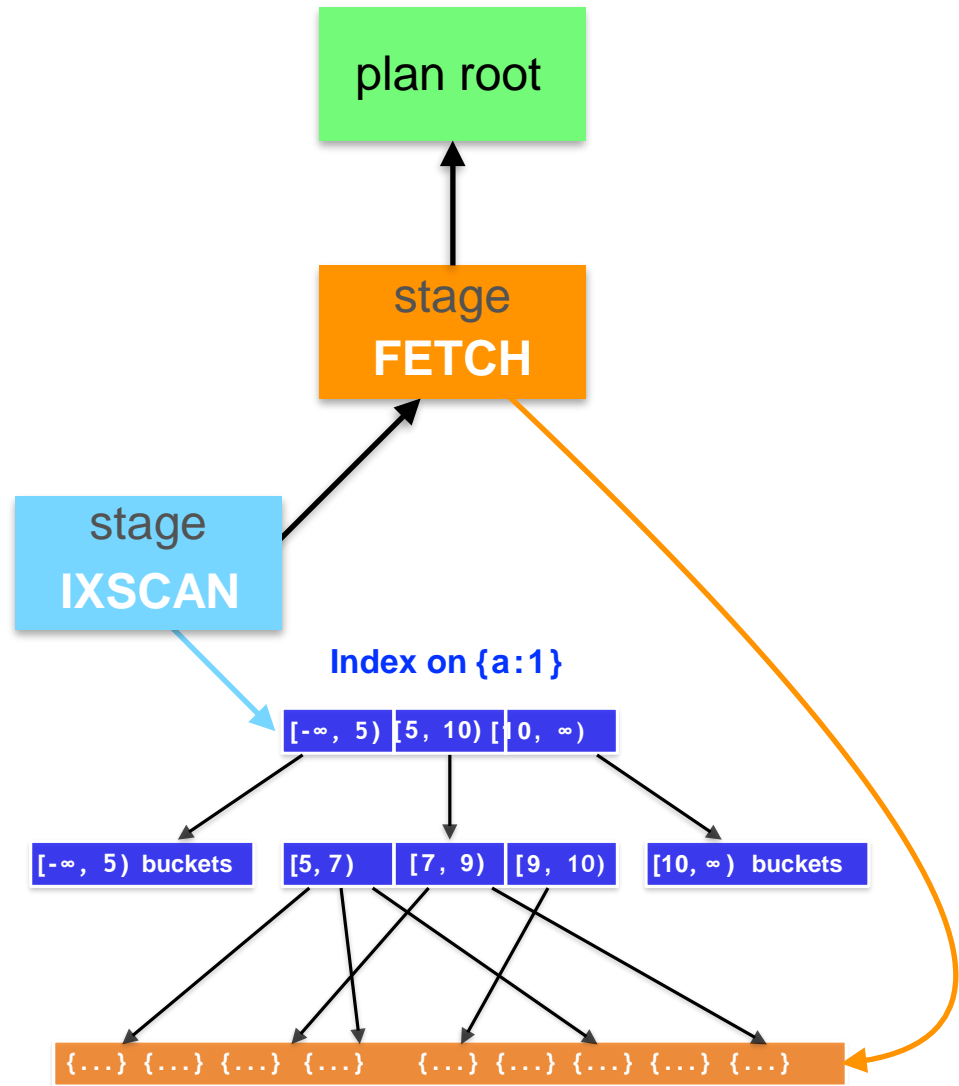
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1,

    "executionStages" : {

      "stage" : "FETCH",
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 0,
      "works" : 2,
      "advanced" : 1,
      "needTime" : 0,
      "needFetch" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 1,
      "invalidates" : 0,
      "docsExamined" : 1,
      "alreadyHasObj" : 0,

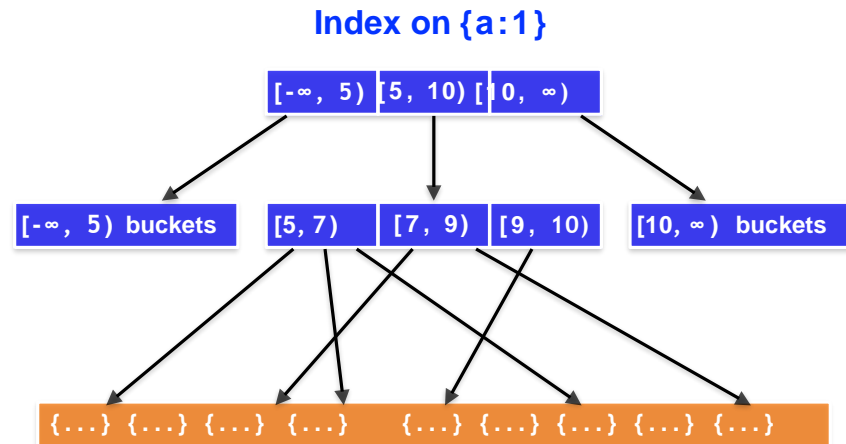
      "inputStage" : {

        "stage" : "IXSCAN",
```



Explain() method key metrics

```
{  
  [...],  
  "executionStats" : {  
  
    "executionSuccess" : true,  
    "nReturned" : 1,  
    "executionTimeMillis" : 0,  
    "totalKeysExamined" : 1,  
    "totalDocsExamined" : 1,  
  
    "executionStages" : {  
  
      "stage" : "FETCH",  
      "nReturned" : 1,  
      "executionTime  
      MillisEstimate" : 0,  
      "works" : 2,  
      "advanced" : 1,  
      "needTime" : 0,  
      "needFetch" : 0,  
      "saveState" : 0,  
      "restoreState" : 0,  
      "isEOF" : 1,  
      "invalidates" : 0,  
      "docsExamined" : 1,  
      "alreadyHasObj" : 0,  
  
      "inputStage" : {  
  
        "stage" : "IXSCAN",
```



Explain() method key metrics

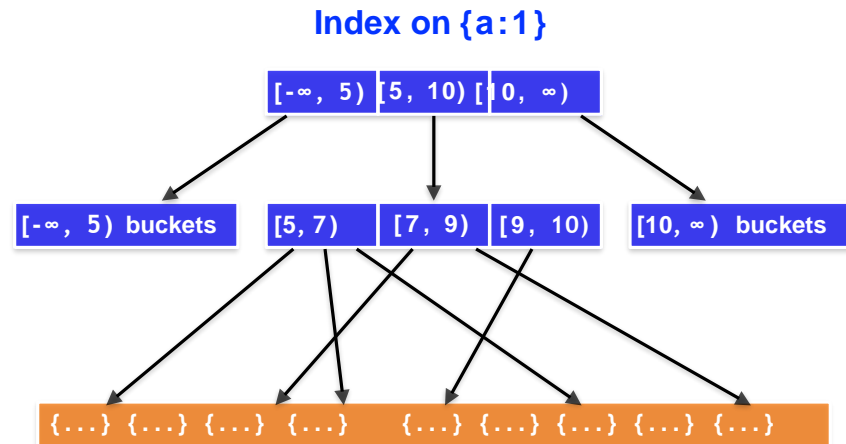
```
{  
  [...],  
  "executionStats" : {  
  
    "executionSuccess" : true,  
    "nReturned" : 1,  
    "executionTimeMillis" : 0  
    "totalKeysExamined" : 1,  
    "totalDocsExamined" : 1,  
  
    "executionStages" : {  
  
      "stage" : "FETCH",  
      "nReturned" : 1,  
      "executionTime  
      MillisEstimate" : 0,  
      "works" : 2,  
      "advanced" : 1,  
      "needTime" : 0,  
      "needFetch" : 0,  
      "saveState" : 0,  
      "restoreState" : 0,  
      "isEOF" : 1,  
      "invalidates" : 0,  
      "docsExamined" : 1,  
      "alreadyHasObj" : 0,  
  
      "inputStage" : {  
  
        "stage" : "IXSCAN",
```

documents returned

How long did the query take

index entries scanned

documents scanned



```

"docsExamined" : 1,
"alreadyHasO" : 0,
"inputStage" : {

```

Explain() method key metrics

```

"stage" : "IXSCAN",
"nReturned" : 1,
"executionTime
MillisEstimate" : 0,
"works" : 1,
"advanced" : 1,
"needTime" : 0,
"needFetch" : 0,
"saveState" : 0,
"restoreState" : 0,
"isEOF" : 1,
"invalidates" : 0,
"keyPattern" : {
  "a" : 5
},
"indexName" : "a_1",
"isMultiKey" : false,
"direction" : "forward",
"indexBounds" : {
  "a" : [
    "[5.0, 5.0]"
  ]
},
"keysExamined" : 1,
"dupsTested" : 0,
"dupsDropped" : 0,
"seenInvalidated" : 0,
"matchTested" : 0
}

```

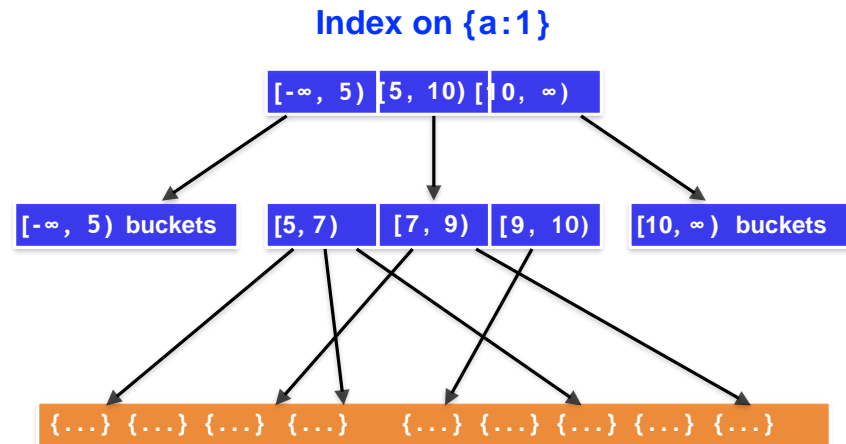
documents returned

How long did the query take

index entries scanned

documents scanned

Index used? Which one?



Performance Tuning with Indexes

Sample Data

db.zips.find()

29353 documents

```
{ "zip": "01001", "city": "AGAWAM", "loc": [ -72.622739, 42.070206 ], "pop": 15338, "state": "MA" }
{ "zip": "01002", "city": "CUSHMAN", "loc": [ -72.51564999999999, 42.377017 ], "pop": 36963, "state": "MA" }
{ "zip": "01005", "city": "BARRE", "loc": [ -72.10835400000001, 42.409698 ], "pop": 4546, "state": "MA" }
{ "zip": "01007", "city": "BELCHERTOWN", "loc": [ -72.41095300000001, 42.275103 ], "pop": 10579, "state": "MA" }
{ "zip": "01008", "city": "BLANDFORD", "loc": [ -72.936114, 42.182949 ], "pop": 1240, "state": "MA" }
{ "zip": "01010", "city": "BRIMFIELD", "loc": [ -72.188455, 42.116543 ], "pop": 3706, "state": "MA" }
{ "zip": "01011", "city": "CHESTER", "loc": [ -72.988761, 42.279421 ], "pop": 1688, "state": "MA" }
{ "zip": "01012", "city": "CHESTERFIELD", "loc": [ -72.833309, 42.38167 ], "pop": 177, "state": "MA" }
{ "zip": "01013", "city": "CHICOPEE", "loc": [ -72.607962, 42.162046 ], "pop": 23396, "state": "MA" }
{ "zip": "01020", "city": "CHICOPEE", "loc": [ -72.576142, 42.176443 ], "pop": 31495, "state": "MA" }
{ "zip": "01022", "city": "WESTOVER AFB", "loc": [ -72.558657, 42.196672 ], "pop": 1764, "state": "MA" }
{ "zip": "01026", "city": "CUMMINGTON", "loc": [ -72.905767, 42.435296 ], "pop": 1484, "state": "MA" }
{ "zip": "01027", "city": "MOUNT TOM", "loc": [ -72.67992099999999, 42.264319 ], "pop": 16864, "state": "MA" }
{ "zip": "01028", "city": "EAST LONGMEADOW", "loc": [ -72.505565, 42.067203 ], "pop": 13367, "state": "MA" }
{ "zip": "01030", "city": "FEEDING HILLS", "loc": [ -72.675077, 42.07182 ], "pop": 11985, "state": "MA" }
{ "zip": "01031", "city": "GILBERTVILLE", "loc": [ -72.19858499999999, 42.332194 ], "pop": 2385, "state": "MA" }
{ "zip": "01032", "city": "GOSHEN", "loc": [ -72.844092, 42.466234 ], "pop": 122, "state": "MA" }
{ "zip": "01033", "city": "GRANBY", "loc": [ -72.52000099999999, 42.255704 ], "pop": 5526, "state": "MA" }
{ "zip": "01034", "city": "TOLLAND", "loc": [ -72.908793, 42.070234 ], "pop": 1652, "state": "MA" }
{ "zip": "01035", "city": "HADLEY", "loc": [ -72.571499, 42.36062 ], "pop": 4231, "state": "MA" }
{ "zip": "01036", "city": "HAMPDEN", "loc": [ -72.43182299999999, 42.064756 ], "pop": 4709, "state": "MA" }
{ "zip": "01038", "city": "HATFIELD", "loc": [ -72.61673500000001, 42.38439 ], "pop": 3184, "state": "MA" }
{ "zip": "01039", "city": "HAYDENVILLE", "loc": [ -72.70317799999999, 42.381799 ], "pop": 1387, "state": "MA" }
{ "zip": "01040", "city": "HOLYOKE", "loc": [ -72.626193, 42.202007 ], "pop": 43704, "state": "MA" }
{ "zip": "01050", "city": "HUNTINGTON", "loc": [ -72.873341, 42.265301 ], "pop": 2084, "state": "MA" }
{ "zip": "01053", "city": "LEEDS", "loc": [ -72.70340299999999, 42.354292 ], "pop": 1350, "state": "MA" }
{ "zip": "01054", "city": "LEVERETT", "loc": [ -72.499334, 42.46823 ], "pop": 1748, "state": "MA" }
{ "zip": "01056", "city": "LUDLOW", "loc": [ -72.471012, 42.172823 ], "pop": 18820, "state": "MA" }
{ "zip": "01057", "city": "MONSON", "loc": [ -72.31963399999999, 42.101017 ], "pop": 8194, "state": "MA" }
{ "zip": "01060", "city": "FLORENCE", "loc": [ -72.654245, 42.324662 ], "pop": 27939, "state": "MA" }
{ "zip": "01068", "city": "OAKHAM", "loc": [ -72.051265, 42.348033 ], "pop": 1503, "state": "MA" }
{ "zip": "01069", "city": "PALMER", "loc": [ -72.328785, 42.176233 ], "pop": 9778, "state": "MA" }
{ "zip": "01070", "city": "PLAINFIELD", "loc": [ -72.918289, 42.514393 ], "pop": 571, "state": "MA" }
{ "zip": "01071", "city": "RUSSELL", "loc": [ -72.840343, 42.147063 ], "pop": 608, "state": "MA" }
{ "zip": "01072", "city": "SHUTESBURY", "loc": [ -72.421342, 42.481968 ], "pop": 1533, "state": "MA" }
{ "zip": "01073", "city": "SOUTHAMPTON", "loc": [ -72.719381, 42.224697 ], "pop": 4478, "state": "MA" }
```



```
{ "zip" : "01256", "city" : "SAVOY", "loc" : [ -73.023281, 42.576964 ], "pop" : 632, "state" : "MA" }
{ "zip" : "01257", "city" : "SHEFFIELD", "loc" : [ -73.361091, 42.100102 ], "pop" : 1839, "state" : "MA" }
{ "zip" : "01258", "city" : "SOUTH EGREMONT", "loc" : [ -73.456575, 42.101153 ], "pop" : 135, "state" : "MA" }
{ "zip" : "01259", "city" : "SOUTHFIELD", "loc" : [ -73.26093299999999, 42.078014 ], "pop" : 622, "state" : "MA" }
{ "zip" : "01262", "city" : "STOCKBRIDGE", "loc" : [ -73.322263000000001, 42.30104 ], "pop" : 2200, "state" : "MA" }
{ "zip" : "01266", "city" : "WEST STOCKBRIDGE", "loc" : [ -73.38251, 42.334752 ], "pop" : 1173, "state" : "MA" }
{ "zip" : "01267", "city" : "WILLIAMSTOWN", "loc" : [ -73.20363999999999, 42.708883 ], "pop" : 8220, "state" : "MA" }
{ "zip" : "01270", "city" : "WINDSOR", "loc" : [ -73.04661, 42.509494 ], "pop" : 770, "state" : "MA" }
{ "zip" : "01301", "city" : "LEYDEN", "loc" : [ -72.601847000000001, 42.601222 ], "pop" : 18968, "state" : "MA" }
{ "zip" : "01330", "city" : "ASHFIELD", "loc" : [ -72.810998, 42.523207 ], "pop" : 1535, "state" : "MA" }
{ "zip" : "01331", "city" : "NEW SALEM", "loc" : [ -72.214644000000001, 42.592065 ], "pop" : 14077, "state" : "MA" }
{ "zip" : "01337", "city" : "LEYDEN", "loc" : [ -72.563439, 42.683784 ], "pop" : 2426, "state" : "MA" }
```

Question

Zip codes in New York City with population more than 100,000 ordered by population in descending order

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}})
.sort({pop:-1})
```

```
{"zip" : "10021", "city" : "NEW YORK", "pop" : 106564, "state" : "NY" }
{"zip" : "10025", "city" : "NEW YORK", "pop" : 100027, "state" : "NY" }
```

Query Plan and Execution Statistics

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

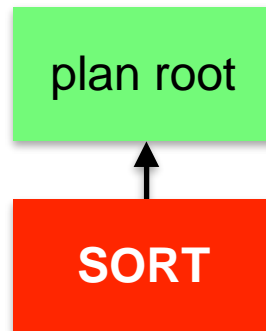
```
{
  "queryPlanner" : {
    "winningPlan" : {
      "stage" : "SORT",
      "sortPattern" : {
        "pop" : -1
      },
      "inputStage" : {
        "stage" : "COLLSCAN",
        "filter" : {
          "$and" : [
            {
              "city" : {
                "$eq" : "NEWYORK"
              }
            },
            {
              "state" : {
                "$eq" : "NY"
              }
            }
          ]
        }
      }
    }
  }
}
```

plan root

Query Plan and Execution Statistics

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

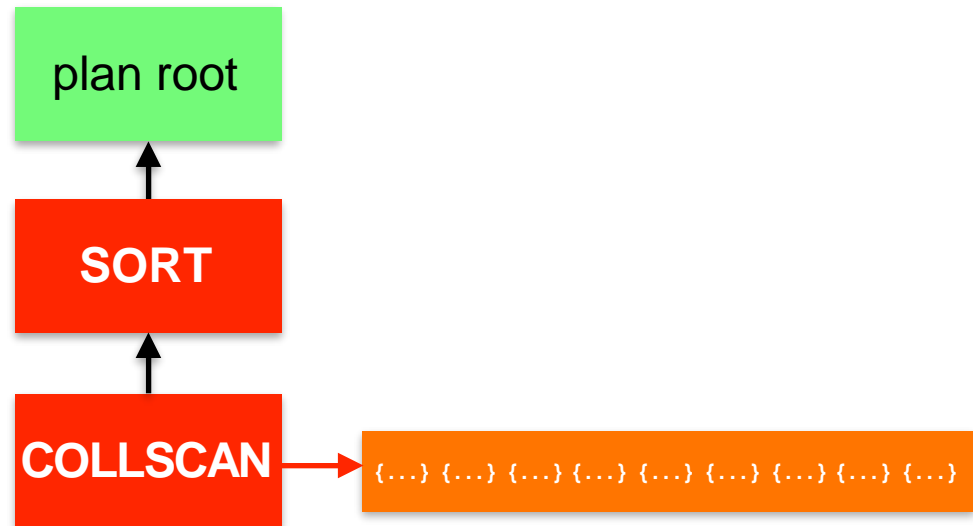
```
{
  "queryPlanner" : {
    "winningPlan" : {
      "stage" : "SORT",
      "sortPattern" : {
        "pop" : -1
      },
    },
    "inputStage" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "$and" : [
          {
            "city" : {
              "$eq" : "NEWYORK"
            }
          },
          {
            "state" : {
              "$eq" : "NY"
            }
          }
        ]
      }
    }
  }
}
```



Collection Scan!

```
db.zips.find({state:'NY',city:'NEW YORK',pop: {'$gt':100000}})
      .sort({pop:-1}).explain("executionStats")
```

```
{
  "queryPlanner" : {
    "winningPlan" : {
      "stage" : "SORT",
      "sortPattern" : {
        "pop" : -1
      },
      "inputStage" : {
        "stage" : "COLLSCAN",
        "filter" : {
          "$and" : [
            {
              "city" : {
                "$eq" : "NEWYORK"
              }
            },
            {
              "state" : {
                "$eq" : "NY"
              }
            }
          ]
        }
      }
    }
  }
}
```

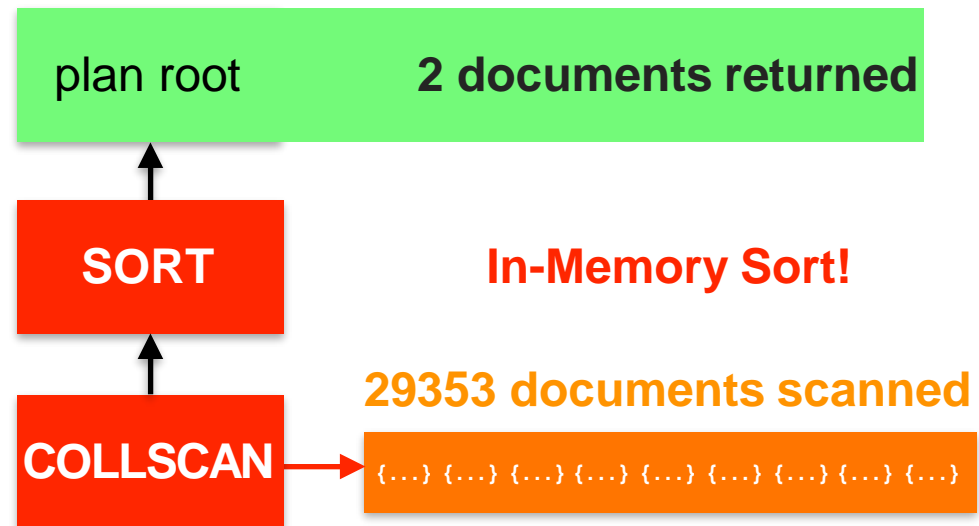


Query Plan and Execution Statistics

Collection Scan!

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1})
```

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 2,
  "executionTimeMillis" : 15,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 29353,
  "executionStages" : {
    "stage" : "SORT",
    "nReturned" : 2,
    "executionTimeMillisEstimate" : 0,
    "works" : 29359,
    "advanced" : 2,
    "needTime" : 29355,
    "needFetch" : 0,
    "saveState" : 229,
    "restoreState" : 229,
    "isEOF" : 1,
    "invalidates" : 0,
    "sortPattern" : {
      "pop" : -1
    }
  }
}
```

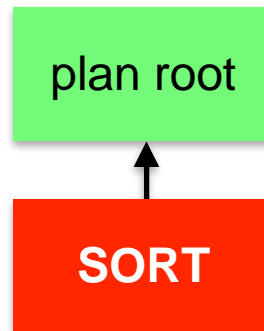


Single Field Index

```
db.zips.createIndex({state:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "SORT",
      "sortPattern": {
        "pop": -1
      },
      "inputStage": {
        "stage": "FETCH",
        "filter": {
          "$and": [
            {
              "city": {
                "$eq": "NEWYORK"
              }
            },
            {
              "pop": {
                "$gt": 100000
              }
            }
          ]
        }
      }
    }
  }
}
```

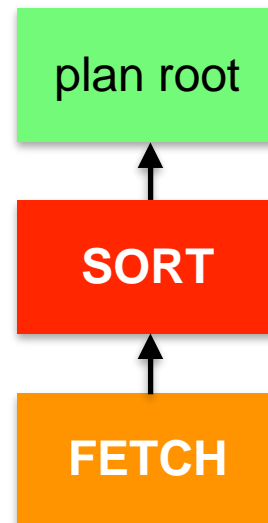


Single Field Index

```
db.zips.createIndex({state:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "SORT",
      "sortPattern": {
        "pop": -1
      },
    },
    "inputStage": {
      "stage": "FETCH",
      "filter": {
        "$and": [
          {
            "city": {
              "$eq": "NEWYORK"
            }
          },
          {
            "pop": {
              "$gt": 100000
            }
          }
        ]
      }
    }
  }
}
```

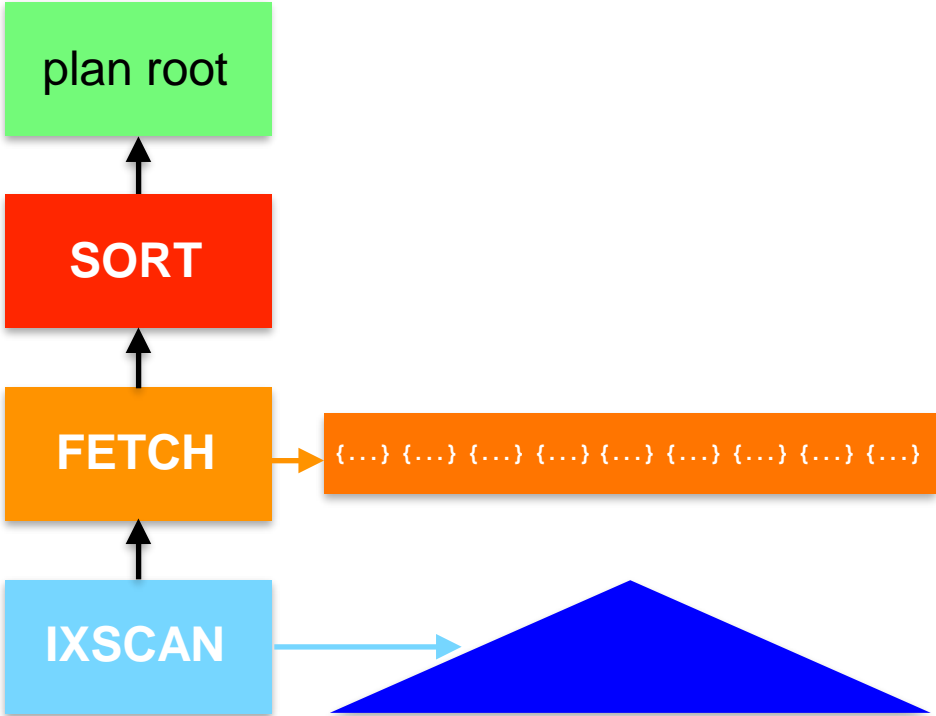


```
db.zips.find({state:'NY',city:'NEW YORK',pop: {'$gt':100000}})
      .sort({pop:-1}).explain("executionStats")
```

```

    }
  }
]
},


```

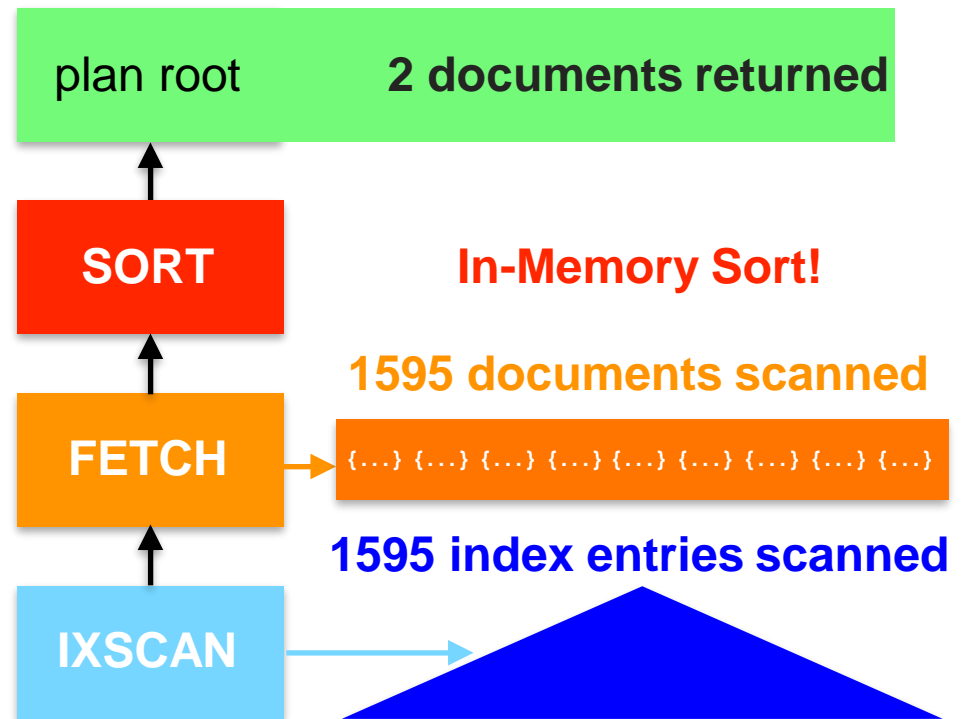


Single Field Index

```
db.zips.createIndex({state:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
"executionStats": {
  "executionSuccess": true,
  "nReturned": 2,
  "executionTimeMillis": 6,
  "totalKeysExamined": 1595,
  "totalDocsExamined": 1595,
  "executionStages": {
    "stage": "SORT",
    "nReturned": 2,
    "executionTimeMillisEstimate": 0,
    "works": 1600,
    "advanced": 2,
    "needTime": 1596,
    "needFetch": 0,
    "saveState": 12,
    "restoreState": 12,
    "isEOF": 1,
    "invalidates": 0,
    "sortPattern": {
      "pop": -1
    }
  }
}
```

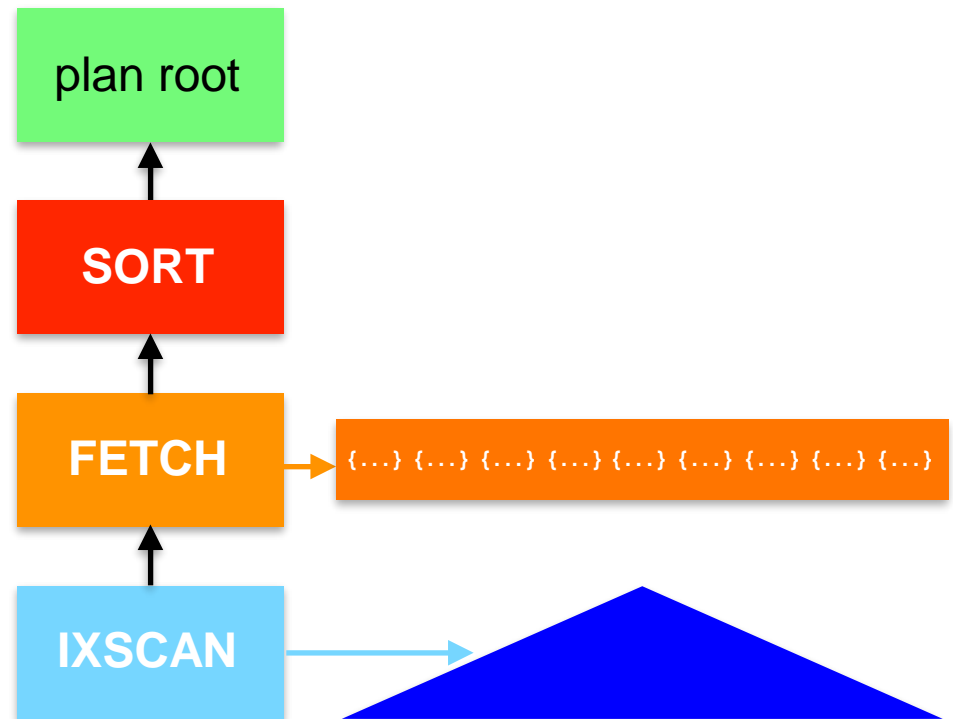


Compound Index on two fields

```
db.zips.createIndex({state:1, city:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "SORT",
      "sortPattern": {
        "pop": -1
      },
    },
    "inputStage": {
      "stage": "FETCH",
      "filter": {
        "pop": {
          "$gt": 100000
        }
      },
    },
    "inputStage": {
      "stage": "IXSCAN",
      "keyPattern": {
        "state": 1,
        "city": 1
      },
    },
  },
}
```

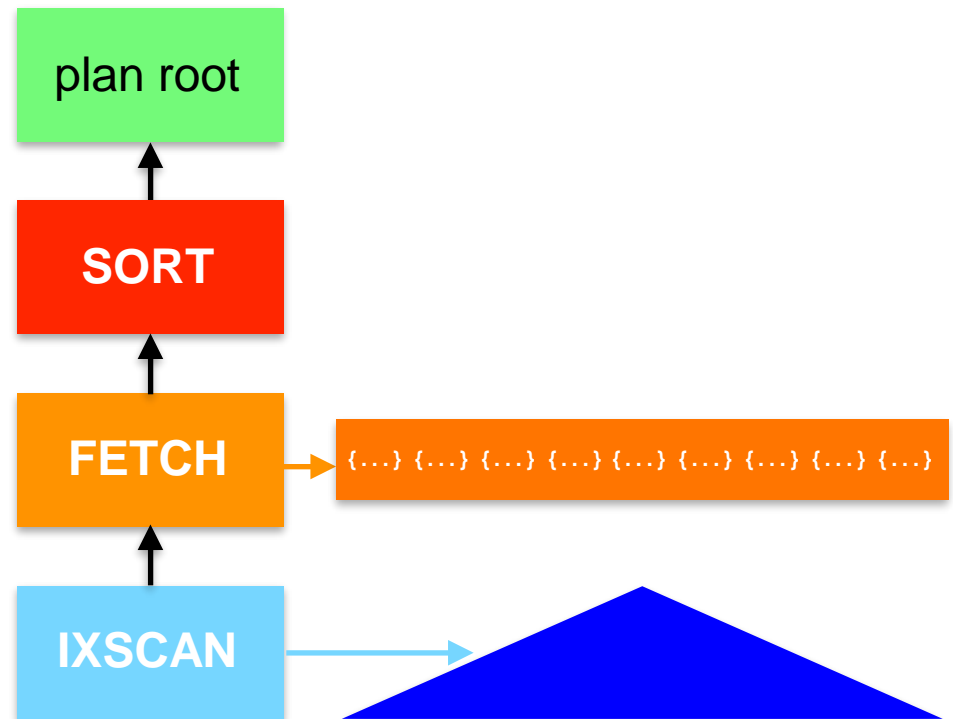


Compound Index on two fields

```
db.zips.createIndex({state:1, city:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})  
  .sort({pop:-1}).explain("executionStats")
```

```
"inputStage": {  
  "stage": "IXSCAN",  
  "keyPattern": {  
    "state": 1,  
    "city": 1  
  },  
  "indexName": "state_1_city_1",  
  "isMultiKey": false,  
  "direction": "forward",  
  "indexBounds": {  
    "state": [  
      "[\"NY\\", \"NY\\"]"  
    ],  
    "city": [  
      "[\"NEW YORK\\", \"NEW YORK  
    ]"  
  }  
}
```

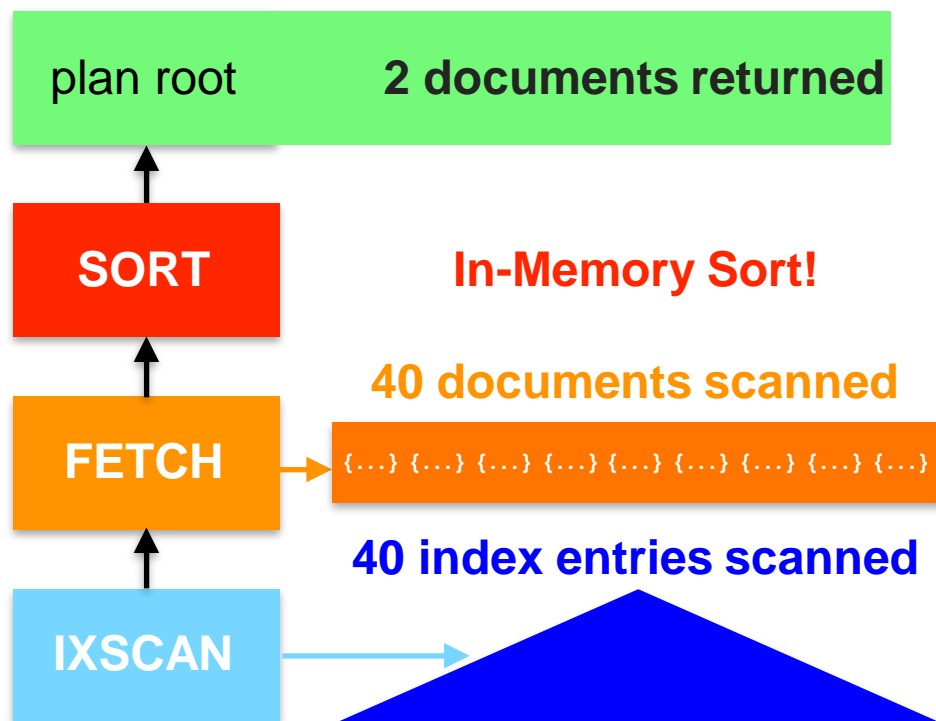


Compound Index on two fields

```
db.zips.createIndex({state:1, city:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
"executionStats": {
  "executionSuccess": true,
  "nReturned": 2,
  "executionTimeMillis": 4,
  "totalKeysExamined": 40,
  "totalDocsExamined": 40,
  "executionStages": {
    "stage": "SORT",
    "nReturned": 2,
    "executionTimeMillisEstimate": 4,
    "works": 45,
    "advanced": 2,
    "needTime": 41,
    "needFetch": 0,
    "saveState": 0,
    "restoreState": 0,
    "isEOF": 1,
    "invalidates": 0,
    "sortPattern": {
      "pop": -1
    }
  }
}
```

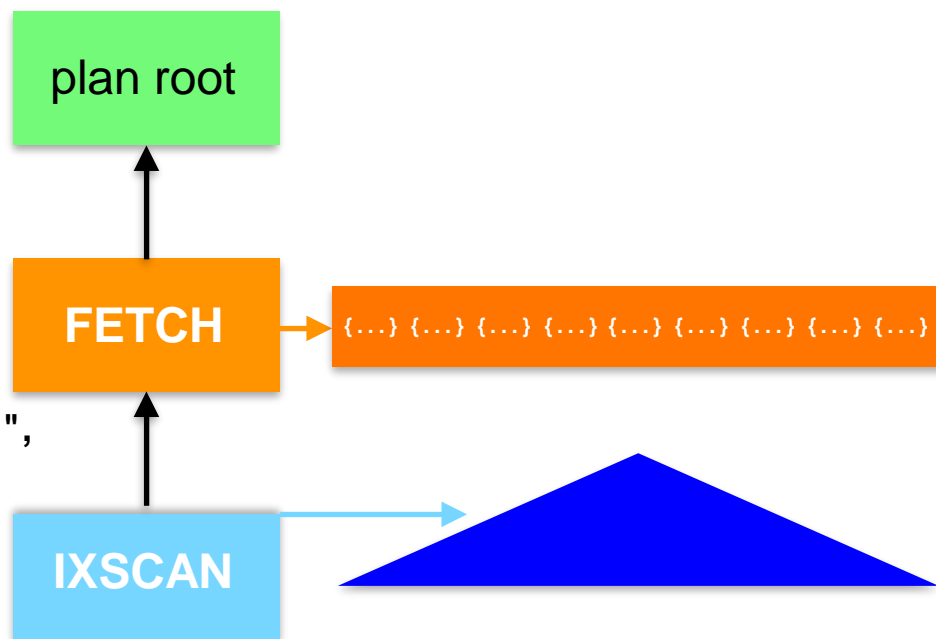


Compound Index on three fields

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop: {'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
{
  "queryPlanner": {
    "winningPlan": {
      "stage": "FETCH",
      "inputStage": {
        "stage": "IXSCAN",
        "keyPattern": {
          "state": 1,
          "city": 1,
          "pop": 1
        },
        "indexName": "state_1_city_1_pop_1",
        "isMultiKey": false,
        "direction": "backward",
        "indexBounds": {
          "state": [
            ["NY", "NY"]
          ],
          "city": [
            ["NEW YORK", "NEW YORK"]
          ]
        }
      }
    }
  }
}
```

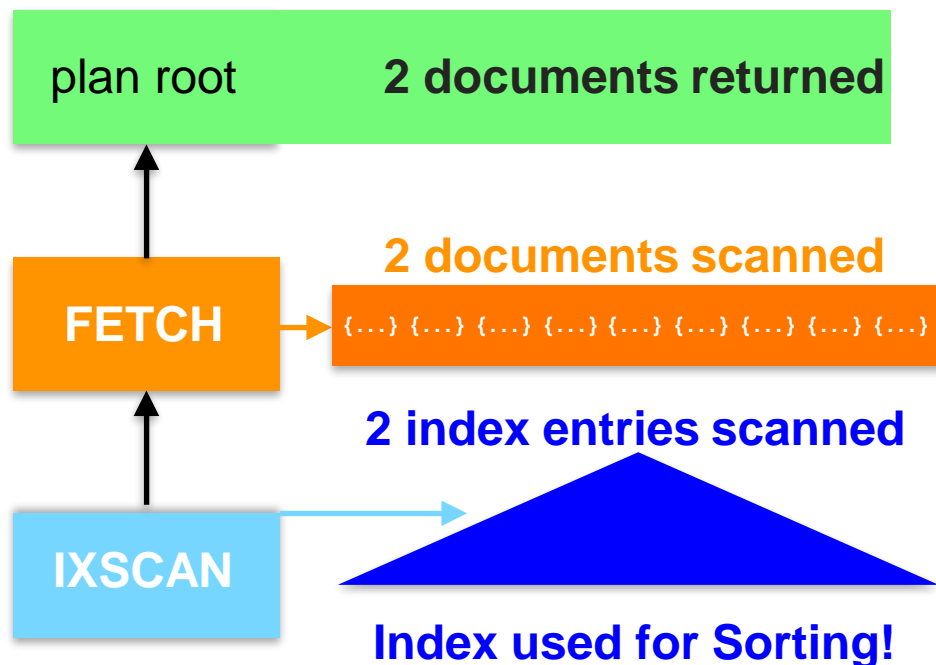


Compound Index on three fields

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop: {'$gt':100000}})
.sort({pop:-1}).explain("executionStats")
```

```
"executionStats": {
  "executionSuccess": true,
  "nReturned": 2,
  "executionTimeMillis": 2,
  "totalKeysExamined": 2,
  "totalDocsExamined": 2,
  "executionStages": {
    "stage": "FETCH",
    "nReturned": 2,
    "executionTimeMillisEstimate": 1,
    "works": 3,
    "advanced": 2,
    "needTime": 0,
    "needFetch": 0,
    "saveState": 0,
    "restoreState": 0,
    "isEOF": 1,
    "invalidates": 0,
    "docsExamined": 2,
    "alreadyHasObj": 0,
```



Projection on Index-only Data

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}})  
      .sort({pop:-1})
```

```
{"zip" : "10021", "city" : "NEW YORK", "pop" : 106564, "state" : "NY" }  
{"zip" : "10025", "city" : "NEW YORK", "pop" : 100027, "state" : "NY" }
```

Projection on Index-only Data

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}})  
      .sort({pop:-1})
```

Projection on Index-only Data

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop: {'$gt':100000}},  
             {state:1, city:1, pop:1})  
             .sort({pop:-1})
```

```
{"state" : "NY", "city" : "NEW YORK", "pop" : 106564 }
```

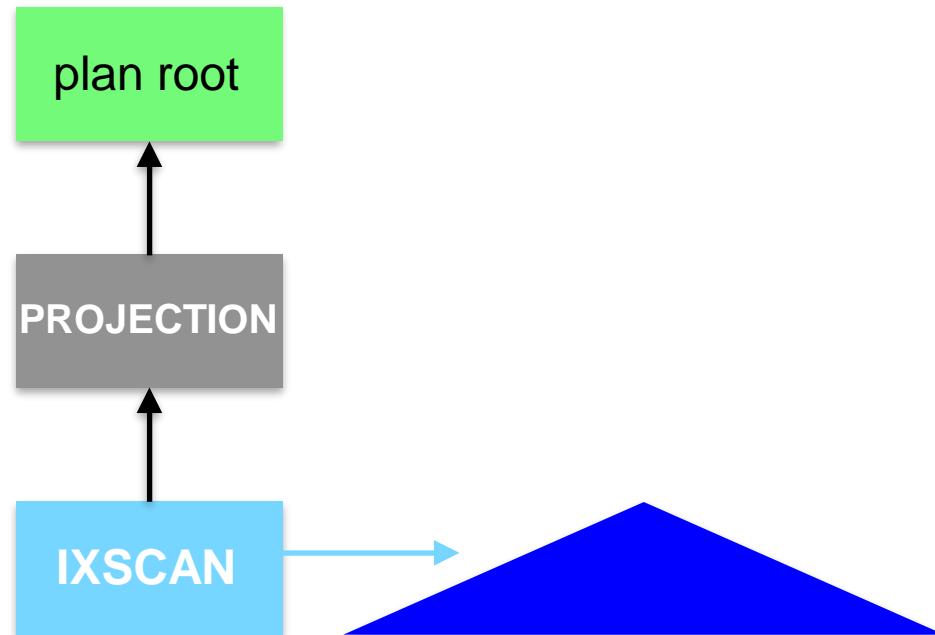
```
{"state" : "NY", "city" : "NEW YORK", "pop" : 100027 }
```

Projection on Index-only Data

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}},  
            {state:1, city:1, pop:1})  
            .sort({pop:-1}).explain("executionStats")
```

```
{  
  "queryPlanner": {  
    "winningPlan": {  
      "stage": "PROJECTION",  
      "transformBy": {  
        "state": 1,  
        "city": 1,  
        "pop": 1,  
        "zip": 0  
      },  
    },  
    "inputStage": {  
      "stage": "IXSCAN",  
      "keyPattern": {  
        "state": 1,  
        "city": 1,  
        "pop": 1  
      },  
      "indexName": "state_1_city_1_pop_1",  
    },  
  },  
}
```

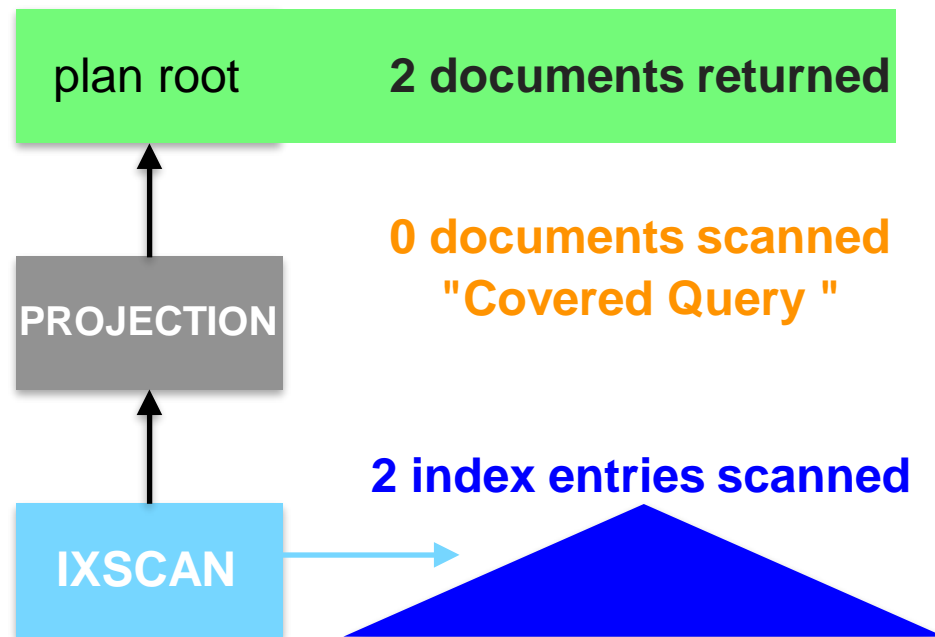


Projection on Index-only Data

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}},  
            {state:1, city:1, pop:1})  
            .sort({pop:-1}).explain("executionStats")
```

```
"executionStats": {  
  "executionSuccess": true,  
  "nReturned": 2,  
  "executionTimeMillis": 0,  
  "totalKeysExamined": 2,  
  "totalDocsExamined": 0,  
  "executionStages": {  
    "stage": "PROJECTION",  
    "nReturned": 2,  
    "executionTimeMillisEstimate": 0,  
    "works": 3,  
    "advanced": 2,  
    "needTime": 0,  
    "needFetch": 0,  
    "saveState": 0,  
    "restoreState": 0,  
    "isEOF": 1,  
    "invalidates": 0,
```



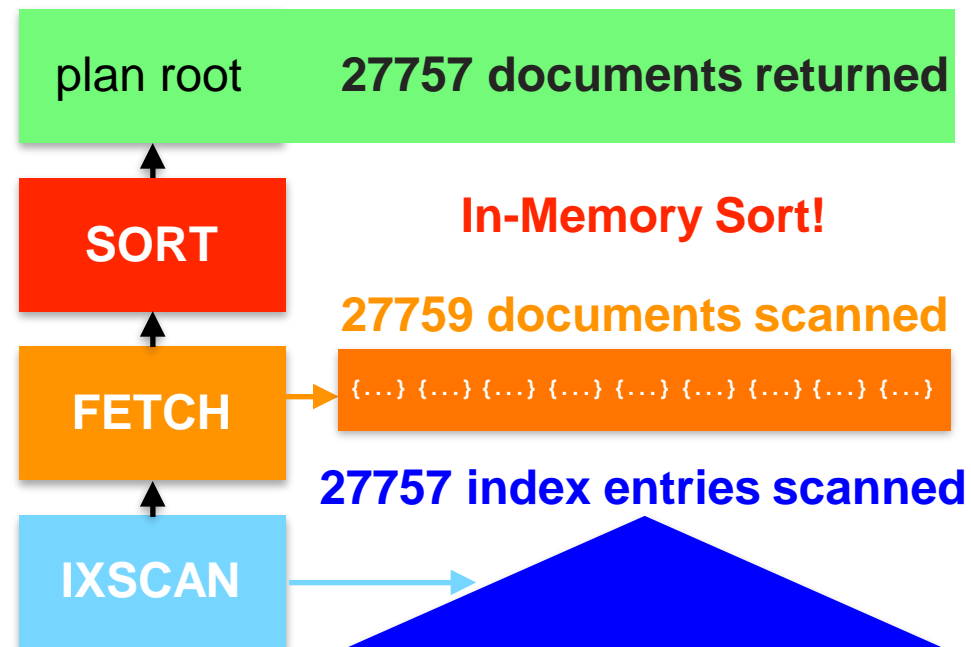
Negation

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:{$ne:'NY'},city:'NEW YORK',pop:{$gt:100000}})
.sort({pop:-1})
```

```
db.zips.find({state:{$ne:'NY'},
             city:{$ne:'NEW YORK'},
             pop:{$not:{$gt:100000}}})
.sort({pop:-1}).explain("executionStats")
```

```
{
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 27757,
    "executionTimeMillis" : 156,
    "totalKeysExamined" : 27759,
    "totalDocsExamined" : 27757,
    "executionStages" : {
      "stage" : "SORT",
      "nReturned" : 27757,
      "executionTimeMillisEstimate" : 110,
      "works" : 55519,
      "advanced" : 27757,
      "needTime" : 27760,
      "needFetch" : 0,
      "saveState" : 433,
```



Negation

```
db.zips.createIndex({state:1,city:1,pop:1})
```

```
db.zips.find({state:{$ne:'NY'},city:'NEW YORK',pop:{$gt:100000}})
.sort({pop:-1})
```

```
db.zips.find({state:{$ne:'NY'},
              city:{$ne:'NEW YORK'},
              pop:{$not:{$gt:100000}}})
.sort({pop:-1}).explain("executionStats")
```

```
"indexName" : "state_1_city_1_pop_1",
```

```
"isMultiKey" : false,
```

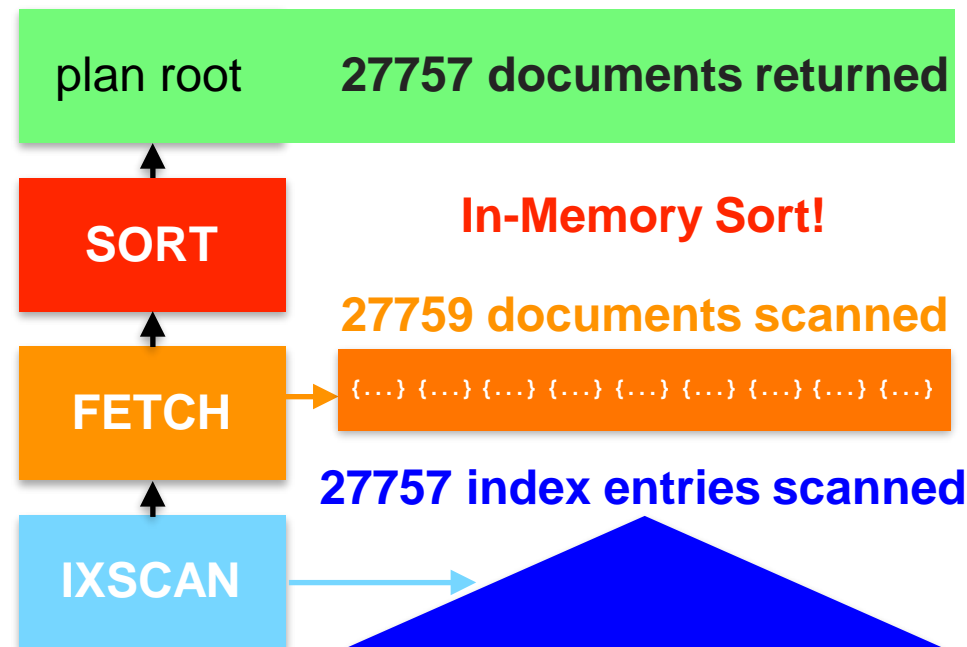
```
"direction" : "forward",
```

```
"indexBounds" : {
```

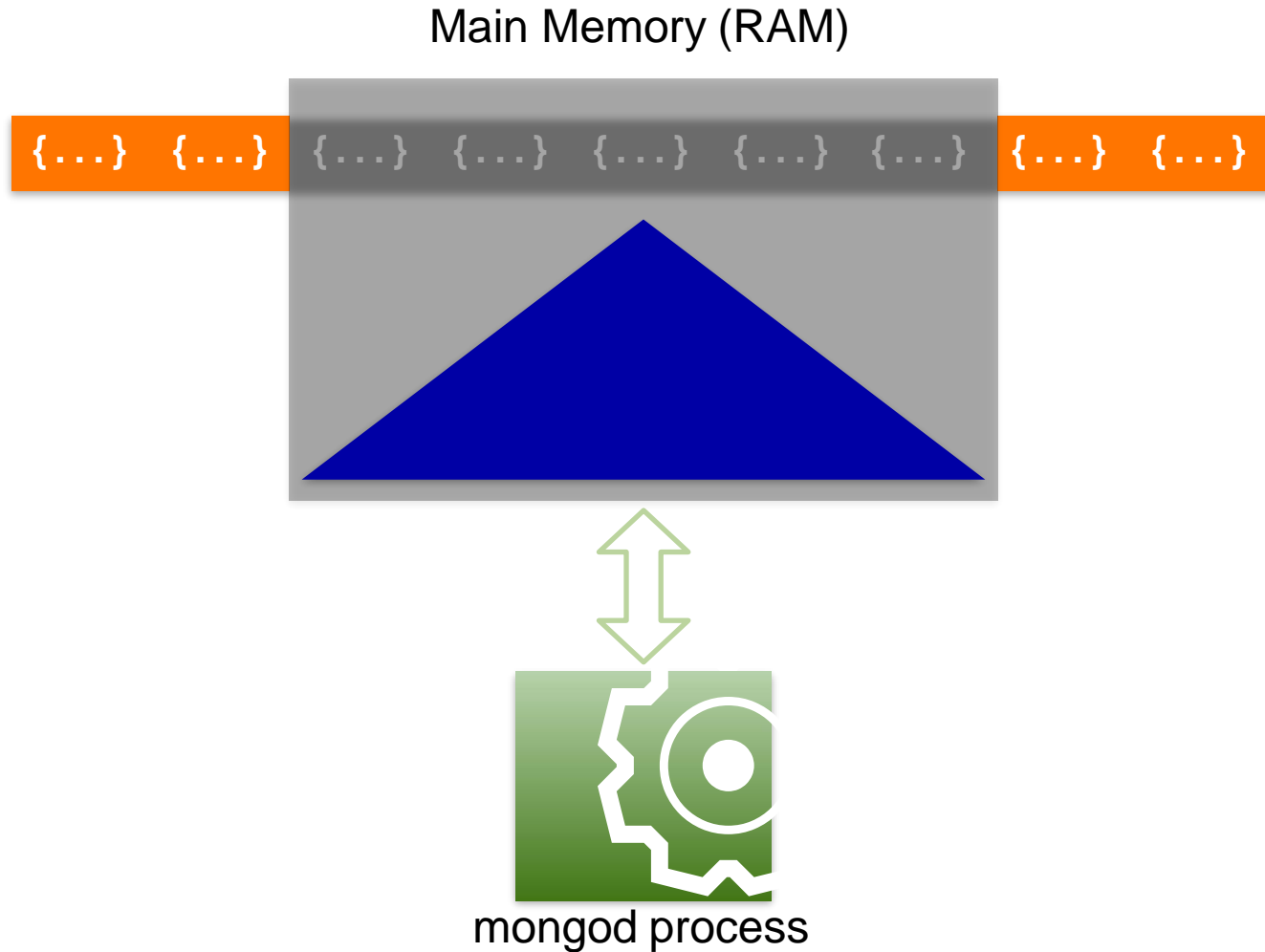
```
  "state" : [
    "[MinKey, \"NY\"]",
    "[\"NY\", MaxKey]"
  ],
```

```
  "city" : [
    "[MinKey, \"NEW YORK\"]",
    "[\"NEW YORK\", MaxKey]"
  ],
```

```
  "pop" : [
    "[MinKey, 100000.0]",
    "[inf.0, MaxKey]"
  ]
}
```

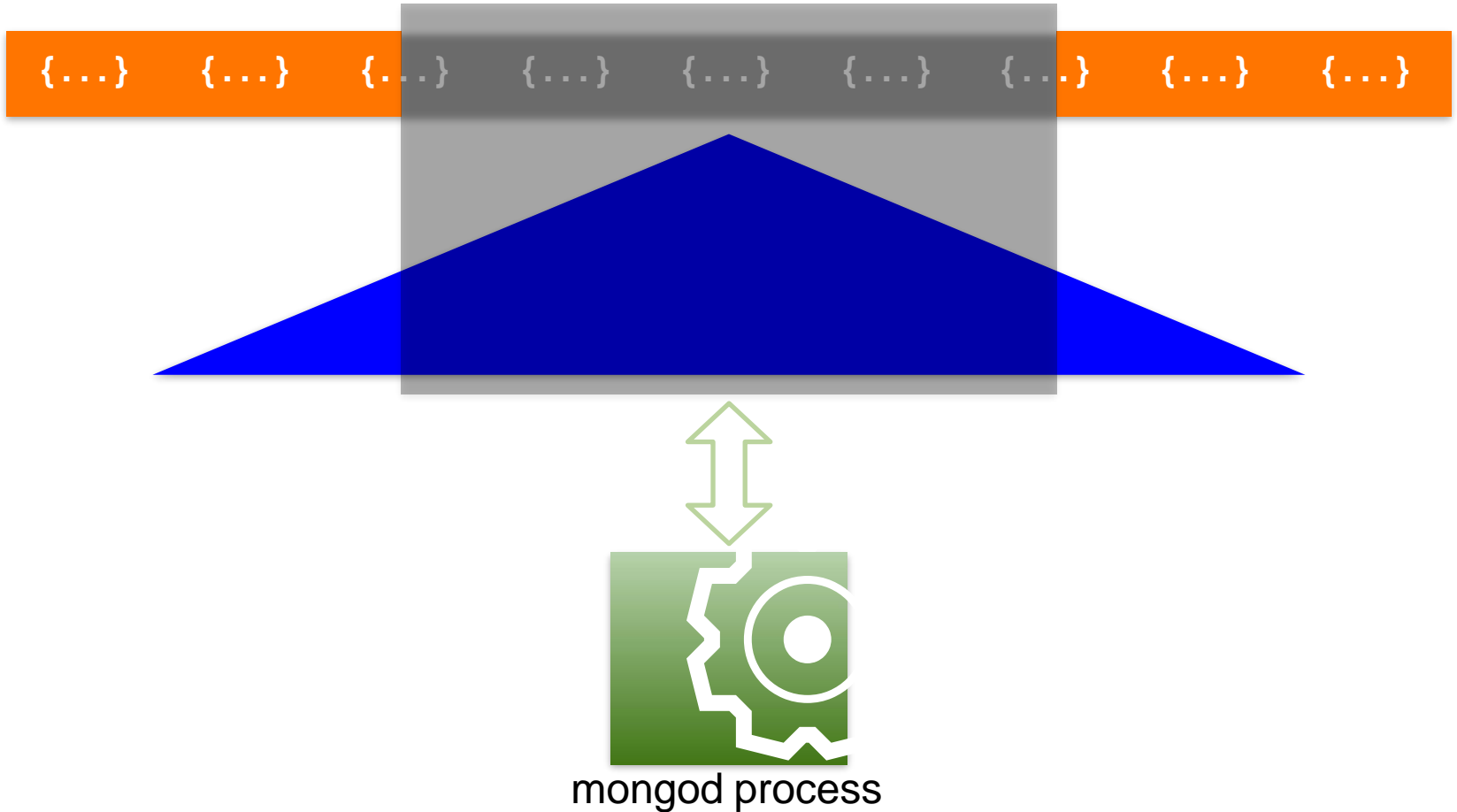


Memory Contention

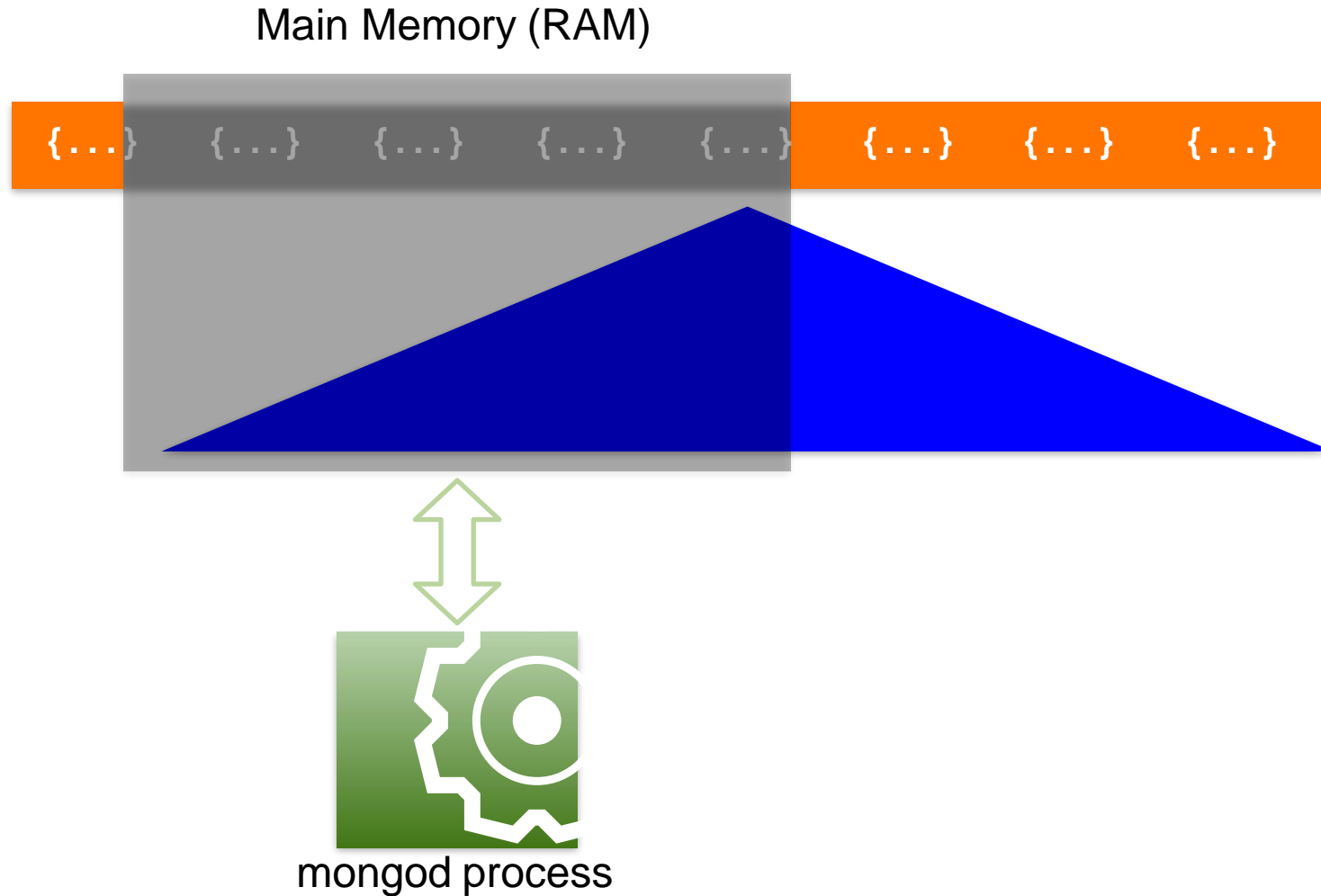


Memory Contention

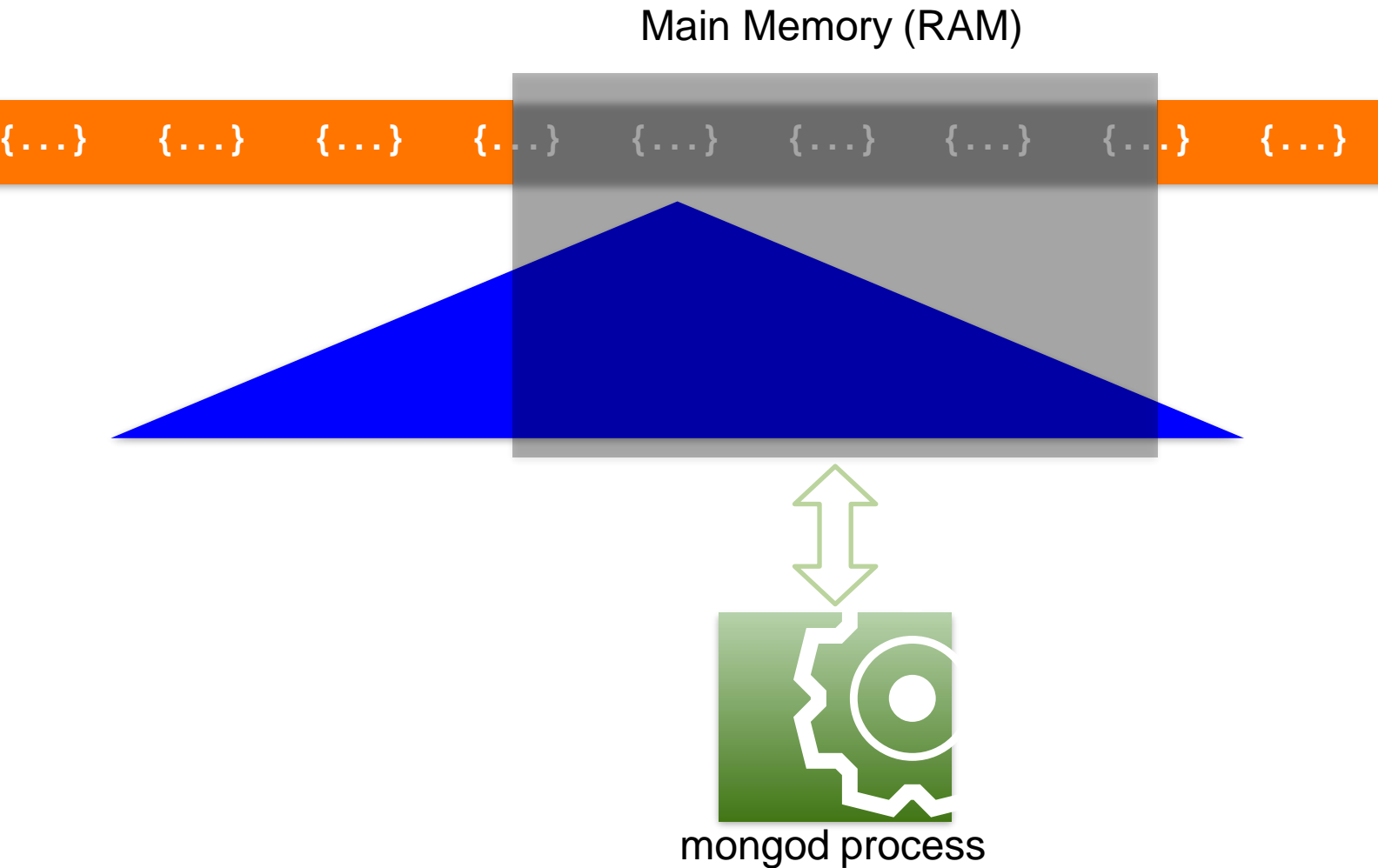
Main Memory (RAM)



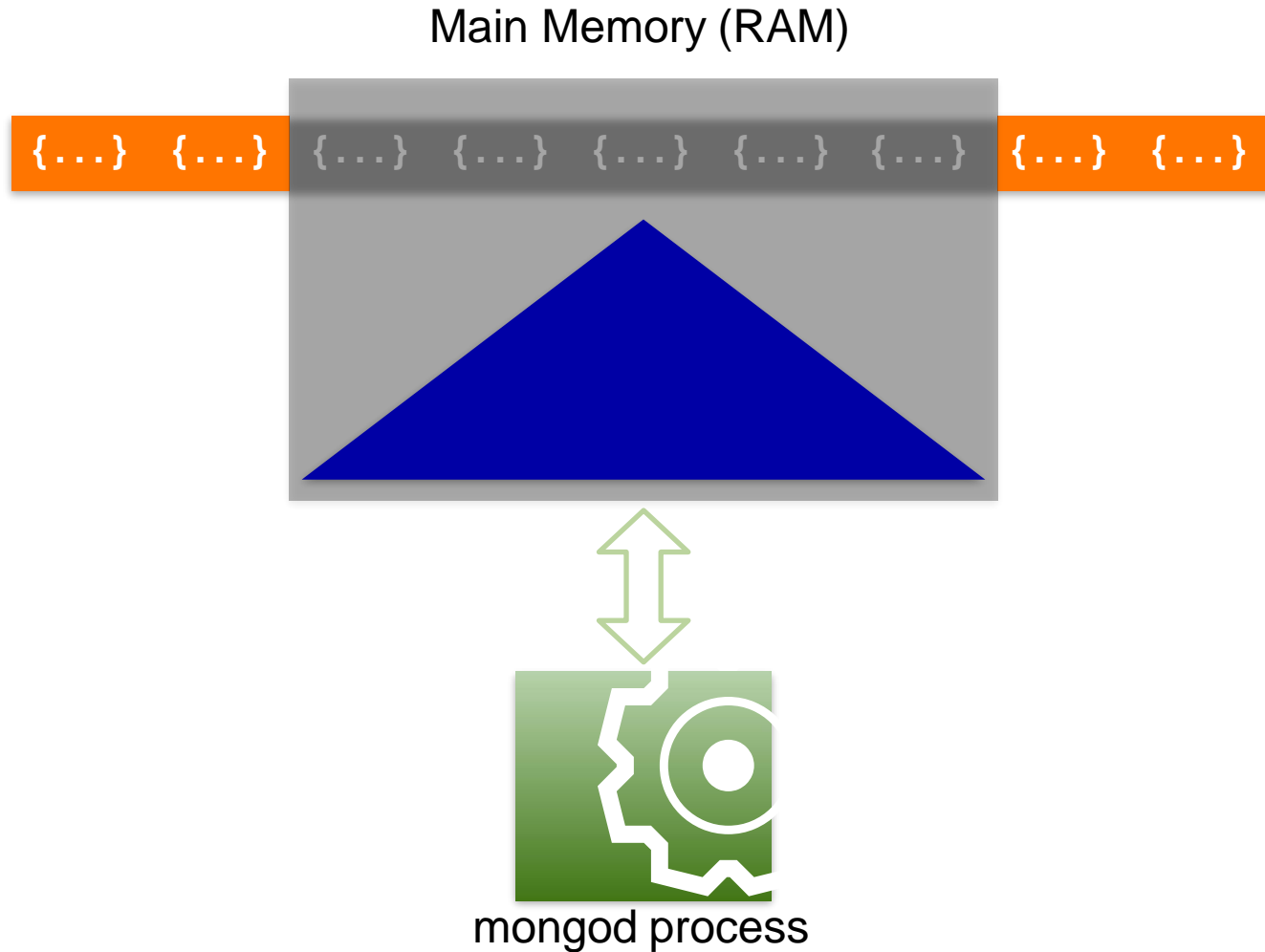
Memory Contention



Memory Contention

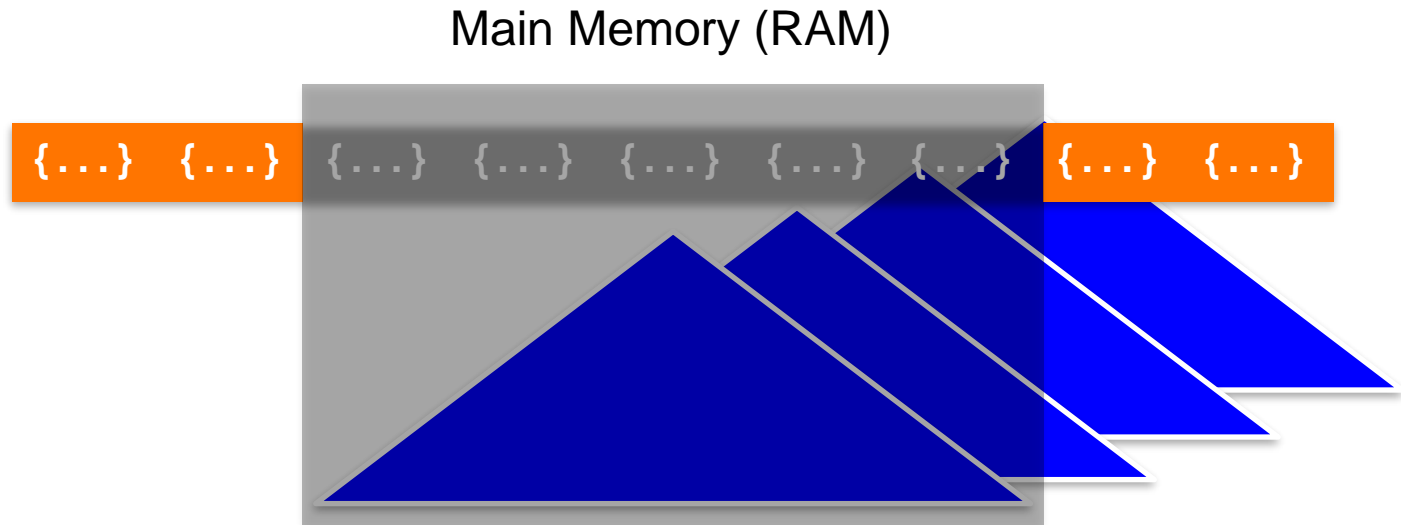


Memory Contention



Reduce Memory Contention

Remove unneeded indexes



```
db.zips.find({state:'NY',city:'NEW YORK',pop:{$gt:100000}})
.sort({pop:-1})
```

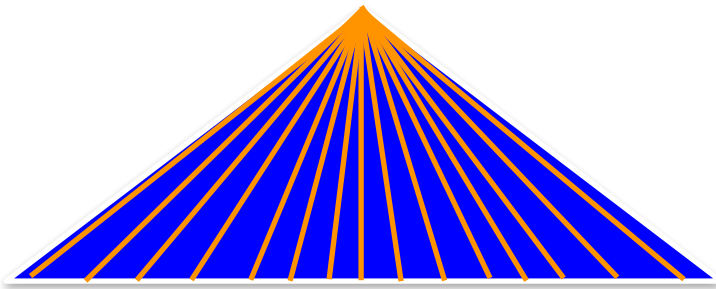
```
db.zips.createIndex({state:1, city:1, pop:1})
```

```
db.zips.dropIndex({state:1, city:1})
```

```
db.zips.dropIndex({state:1})
```

Reduce Memory Contention

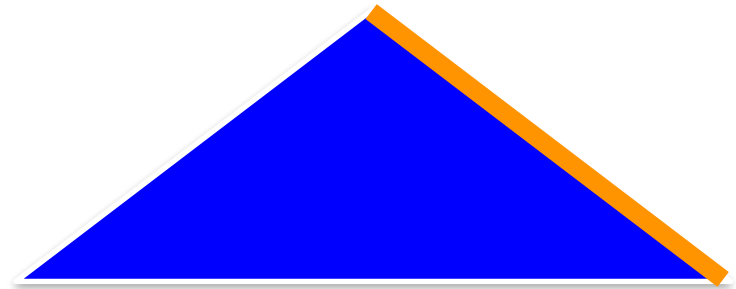
Right-Balanced Index Access



Random Index Access

The entire index is “hot”

The entire index must fit into RAM



Right-Balanced Index Access

Spatial Data Locality

Amount of Index Data in RAM
is significantly reduced

Reduce Memory Contention

Index compression

```
mongod --dbpath DBPATH--storageEngine wiredTiger  
--wiredTigerIndexPrefixCompression
```

Prefix Compression for Indexes on disk and in RAM

Reduce Data Transfer Latency

Indexes on a separate storage device

```
mongod --dbpath DBPATH--storageEngine wiredTiger  
--wiredTigerDirectoryForIndexes
```

One file per Collection under DBPATH/collection

One file per Index under DBPATH/collection

Indexes can be placed on a dedicated storage device (e.g. SSD) for higher performance



**Secondary
Indexes**

Performance

Summary

Indexes are the single biggest tunable performance factor in MongoDB

Create indexes that support your queries

Analyse your query plans with `explain()`

Create highly selective indexes

Remove unneeded indexes

Use covered queries for maximum read performance

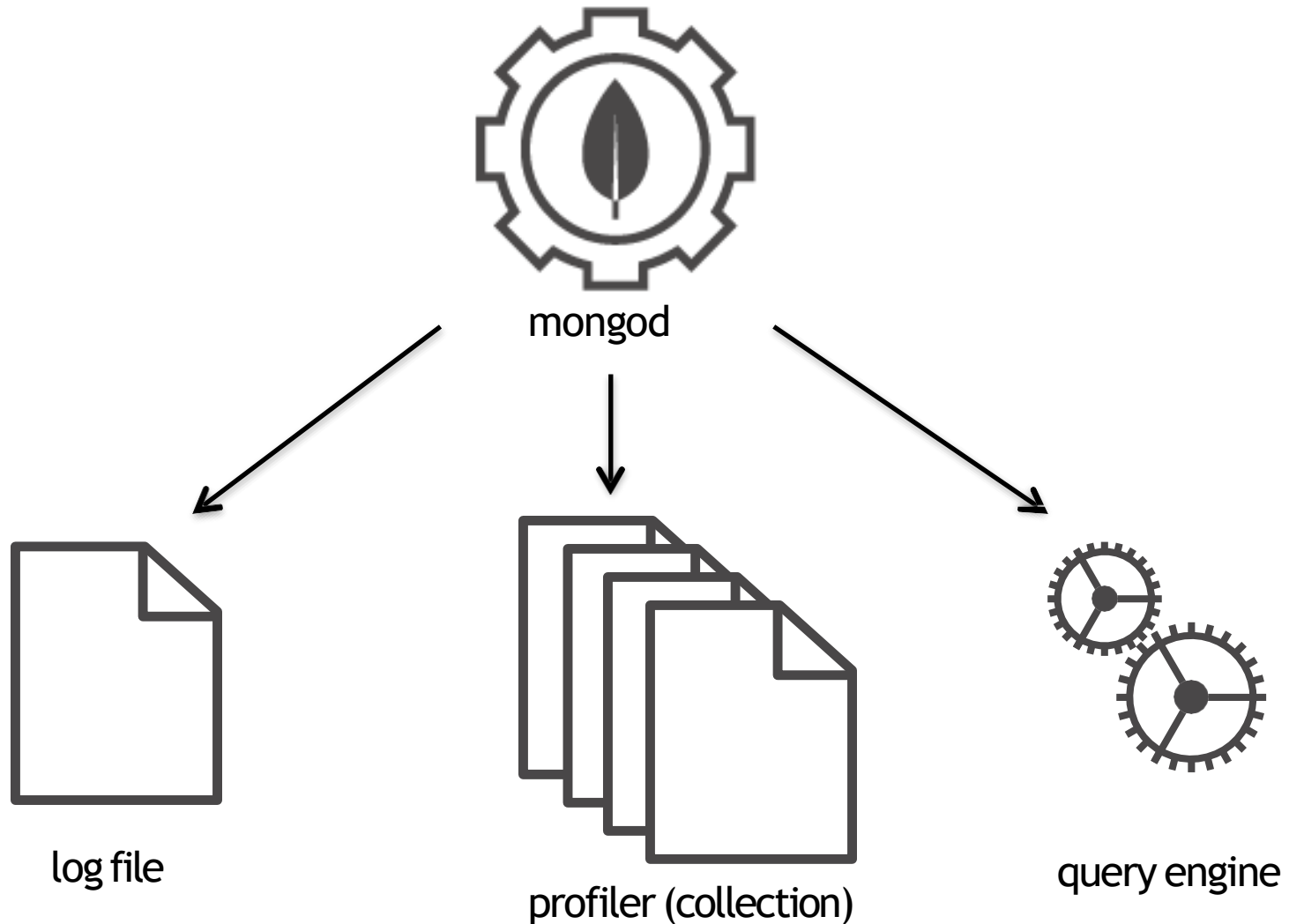
Negation Queries can't benefit from indexes

Use wiredTiger feature to place indexes on high performance volumes

MongoDB

Performance Measurement Tools

Log files, Profiler, Query Optimizer



mongod Log Files

date and time

thread

operation

namespace

number of
yields

lock
times

duration

n...
counters

```
Sun Jun 29 06:35:37.646 [conn2]
query test.docs query:
{ parent.company: "22794",
  parent.employeeId: "83881" }
ntoreturn:1 ntoskip:0 nscanned:
806381 keyUpdates:0 numYields: 5
locks(micros) r:2145254
nreturned:0 reslen:20 1156ms
```

Parsing Log Files

```
5533 Wed Feb 28 22:02:18 [conn3858] end connection 18.3.1.16:58852 (84 connections now open)
5534 Wed Feb 28 22:02:18 [initandlisten] connection accepted from 18.3.1.16:58853 (84 connections now open)
5535 Wed Feb 28 22:02:18 [conn3717] query a.fs.chunks query: { query: { files_id: 627283867766556888, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:589112 $returned:1 $res:81
5536 Wed Feb 28 22:02:18 [conn3858] query a.fs.chunks query: { query: { files_id: 5582289612292227672, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:5889 $returned:1 $res:5888
5537 Wed Feb 28 22:02:22 [conn3859] end connection 18.3.1.13:48841 (84 connections now open)
5538 Wed Feb 28 22:02:22 [initandlisten] connection accepted from 18.3.1.13:48899 (84 connections now open)
5539 Wed Feb 28 22:02:23 [initandlisten] connection accepted from 18.3.1.12:37516 (86 connections now open)
5540 Wed Feb 28 22:02:23 [conn3886] end connection 18.3.1.12:37516 (85 connections now open)
5541 Wed Feb 28 22:02:23 [initandlisten] connection accepted from 18.3.1.12:37524 (86 connections now open)
5542 Wed Feb 28 22:02:38 [conn3867] end connection 18.3.1.12:37524 (85 connections now open)
5543 Wed Feb 28 22:02:38 [conn3851] query a.fs.chunks query: { query: { files_id: 58443849666666666666, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:583214 $returned:1 $res:26
5544 Wed Feb 28 22:02:38 [conn3851] query a.fs.chunks query: { query: { files_id: 7879453347278774272, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:78961 $returned:1 $res:3285
5545 Wed Feb 28 22:02:38 [conn3851] query a.fs.chunks query: { query: { files_id: 4798633818482711328, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:133834 $returned:1 $res:87
5546 Wed Feb 28 22:02:39 [conn3851] query a.fs.chunks query: { query: { files_id: 6662729362531899362, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:286382 $returned:1 $res:262
5547 Wed Feb 28 22:02:39 [conn3851] query a.fs.chunks query: { query: { files_id: 7888533395853289184, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:463779 $returned:1 $res:1487
5548 Wed Feb 28 22:02:44 [initandlisten] connection accepted from 18.3.1.12:37544 (85 connections now open)
5549 Wed Feb 28 22:02:44 [conn3886] end connection 18.3.1.12:37544 (85 connections now open)
5550 Wed Feb 28 22:02:48 [conn3864] end connection 18.3.1.16:58857 (84 connections now open)
5551 Wed Feb 28 22:02:48 [initandlisten] connection accepted from 18.3.1.16:58953 (85 connections now open)
5552 Wed Feb 28 22:02:48 [initandlisten] connection accepted from 18.3.1.13:48899 (84 connections now open)
5553 Wed Feb 28 22:02:52 [conn3885] end connection 18.3.1.13:48899 (84 connections now open)
5554 Wed Feb 28 22:02:52 [initandlisten] connection accepted from 18.3.1.13:48946 (85 connections now open)
5555 Wed Feb 28 22:02:52 [conn3726] query a.fs.chunks query: { query: { files_id: 82582368676666666666, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:578391 $returned:1 $res:18
5556 Wed Feb 28 22:02:58 [initandlisten] connection accepted from 18.3.1.12:37568 (85 connections now open)
5557 Wed Feb 28 22:02:58 [conn3871] end connection 18.3.1.12:37568 (85 connections now open)
5558 Wed Feb 28 22:02:57 [rsync] info DPM:findAll(): extent 63:6a100000 was empty, skipping ahead. no local replica. skip
5559 Wed Feb 28 22:03:00 [initandlisten] connection accepted from 18.3.1.12:37576 (86 connections now open)
5560 Wed Feb 28 22:03:00 [conn3872] end connection 18.3.1.12:37576 (85 connections now open)
5561 Wed Feb 28 22:03:04 [initandlisten] connection accepted from 18.3.1.12:37587 (86 connections now open)
5562 Wed Feb 28 22:03:04 [conn3873] end connection 18.3.1.12:37587 (85 connections now open)
5563 Wed Feb 28 22:03:09 [initandlisten] connection accepted from 18.3.1.12:37597 (86 connections now open)
5564 Wed Feb 28 22:03:09 [conn3874] end connection 18.3.1.12:37597 (85 connections now open)
5565 Wed Feb 28 22:03:18 [conn3880] end connection 18.3.1.16:58953 (84 connections now open)
5566 Wed Feb 28 22:03:18 [initandlisten] connection accepted from 18.3.1.16:58996 (85 connections now open)
5567 Wed Feb 28 22:03:18 [conn3847] query a.fs.chunks query: { query: { files_id: 8718236481386831872, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:128572 $returned:1 $res:81
5568 Wed Feb 28 22:03:22 [conn3878] end connection 18.3.1.13:48846 (84 connections now open)
5569 Wed Feb 28 22:03:22 [initandlisten] connection accepted from 18.3.1.13:48866 (85 connections now open)
5570 Wed Feb 28 22:03:22 [conn3734] query a.fs.chunks query: { query: { files_id: 75843349873278544, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:578888 $returned:1 $res:834
5571 Wed Feb 28 22:03:23 [conn3847] query a.fs.chunks query: { query: { files_id: 828434989436733184, n: 0 }, $readPreference: { mode: "nearest" } } $store:2 $skip:0 $scan:1 $keyUpdates:0 $acks($icrs): r:1296 $returned:1 $res:4462
5572 Wed Feb 28 22:03:38 [initandlisten] connection accepted from 18.3.1.12:37626 (86 connections now open)
5573 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5574 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5575 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5576 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5577 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5578 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5579 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5580 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5581 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5582 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5583 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5584 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5585 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5586 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5587 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5588 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5589 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5590 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5591 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5592 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5593 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5594 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5595 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5596 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5597 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5598 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5599 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
5600 Wed Feb 28 22:03:38 [conn3877] end connection 18.3.1.12:37626 (85 connections now open)
```

Database Profiler

- Collect actual samples from a running MongoDB instance
- Tunable for level and slowness
- Can be controlled dynamically

Using Database profiler

- Enable to see slow queries
 - (or all queries)
 - Default 100ms

```
// Enable database profiler on the console, 0=off 1=slow 2=all
> db.setProfilingLevel(1, 50)
{ "was" : 0, "slowms" : 50, "ok" : 1 }

// View profile with
> show profile

// See the raw data
>db.system.profile.find().pretty()
```

Profiler

- 1MB capped collection named system.profile per database, per replica set
- One document per operation
- Examples:
 - > db.setProfilingLevel(1) // log all operations greater than 100ms
 - > db.setProfilingLevel(1, 20) // log all operations greater than 20ms
 - > db.setProfilingLevel(2) // log all operations regardless of duration
 - > db.setProfilingLevel(0) // turn off profiling
 - > db.getProfilingStatus() // display current profiling level

```
{
  "slowms": 100,
  "was": 2
}
```
- In a sharded cluster, you will need to connect to each shard's primary mongod, not mongos