#CodeMash

Indexing and Query Optimization

Sridhar Nanjundeswaran

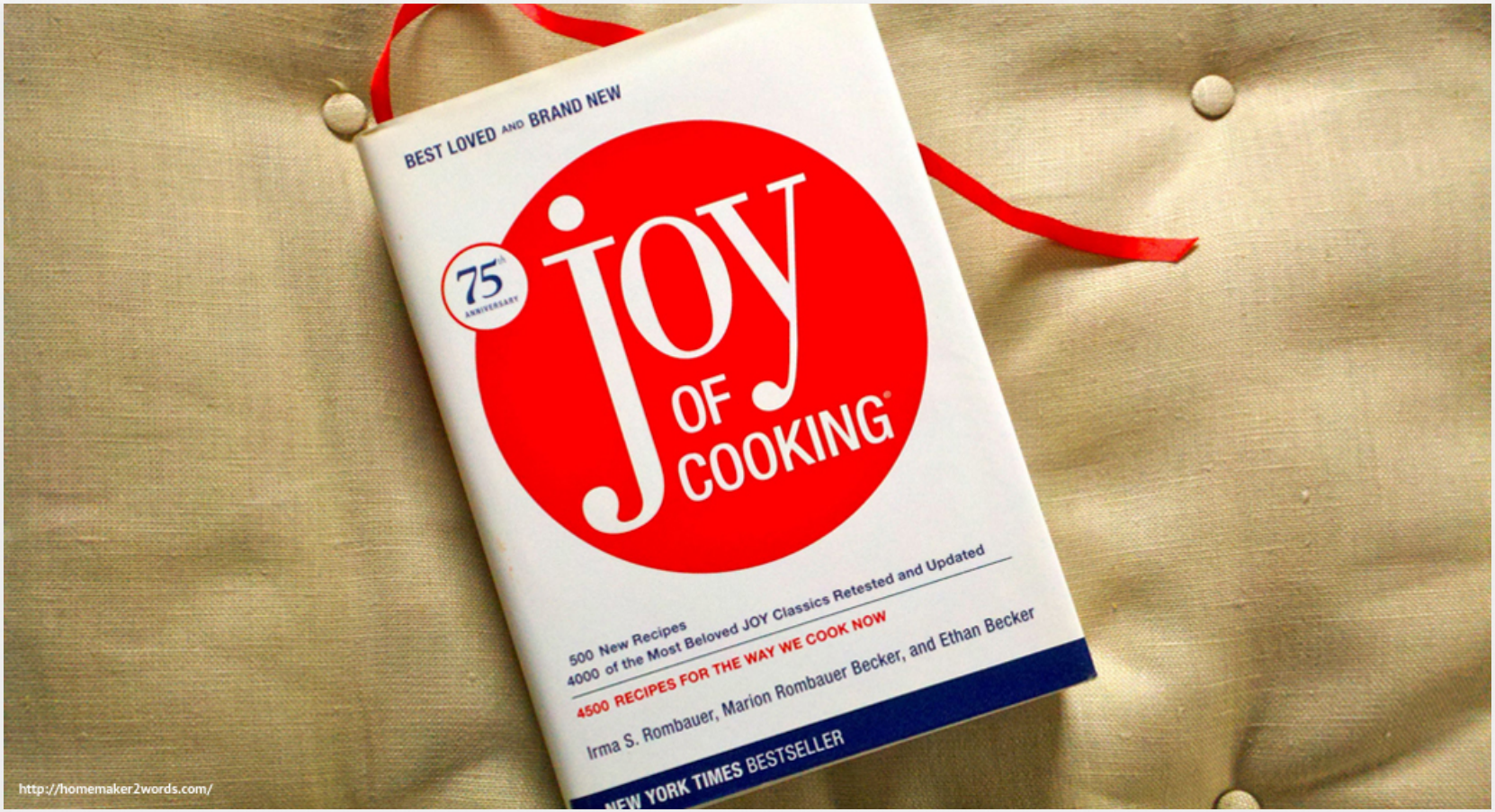
Engineer, 10gen

@snanjund

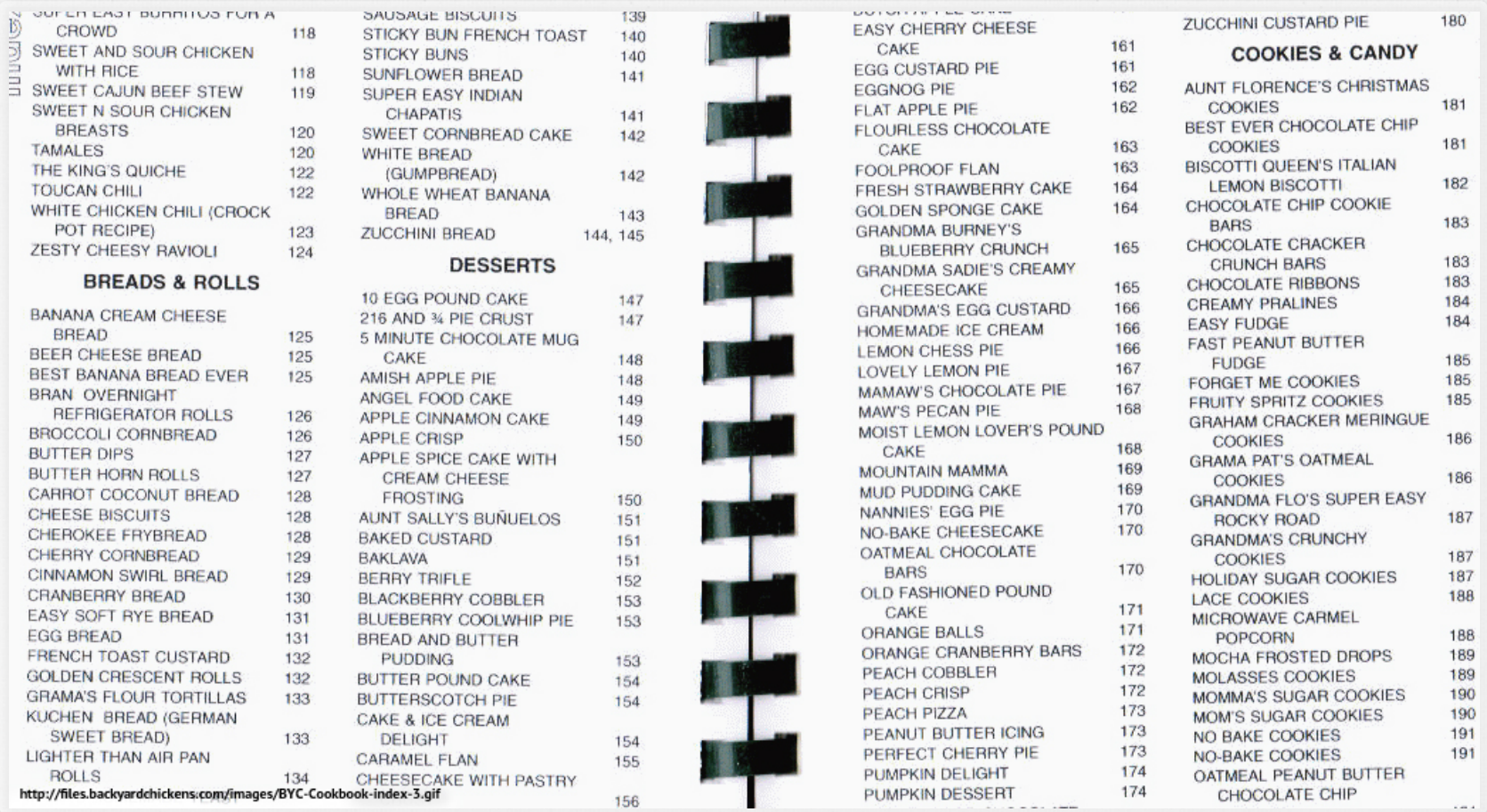
10gen | the MongoDB company

Agenda

- Working with indexes in MongoDB
- Optimize your queries
- Avoiding common mistakes



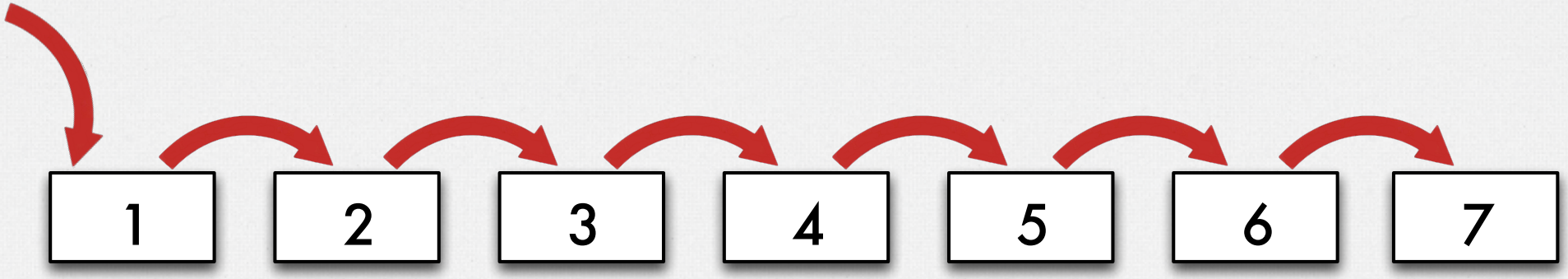
What are indexes?



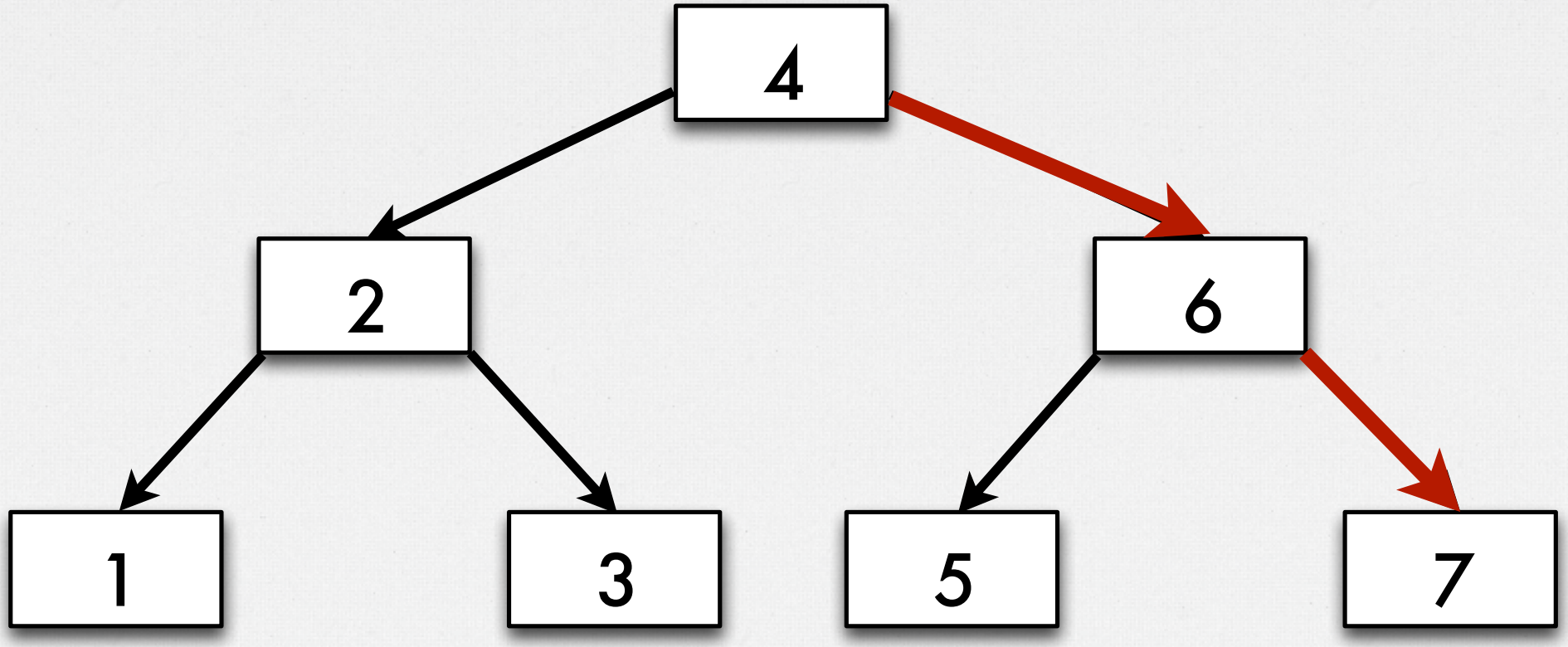
http://files.backyardchickens.com/images/BYC-Cookbook-index-3.gif

Consult the index!

Linked List

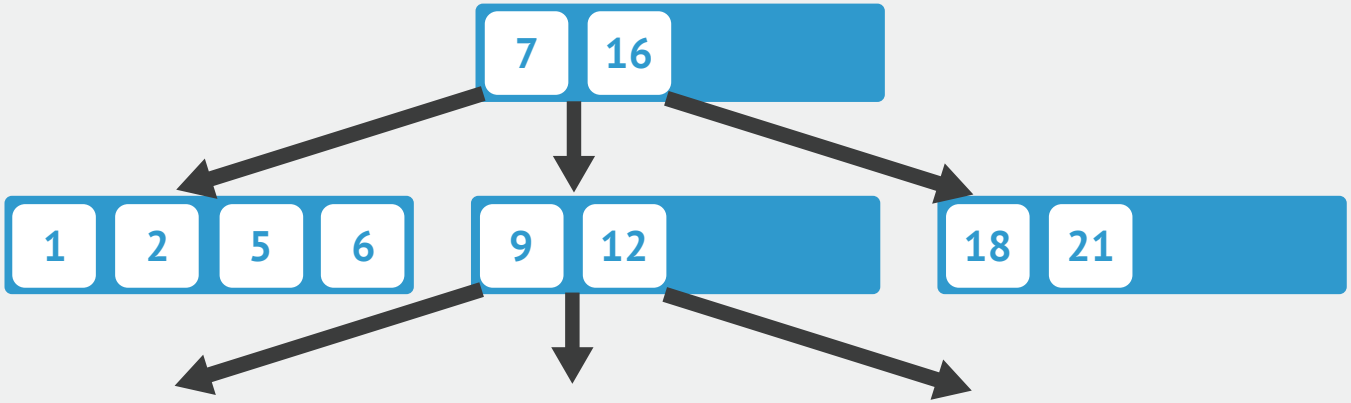


Finding 7 in Linked List



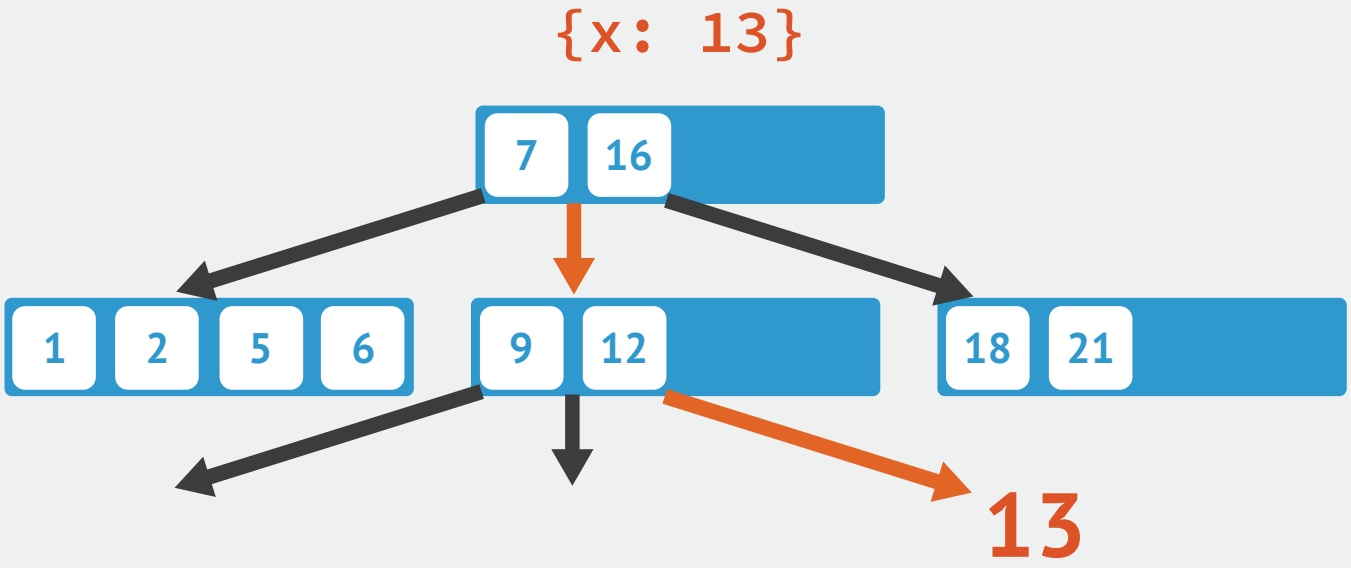
Finding 7 in Tree

B-Trees



Indexes in MongoDB are B-trees

B-Trees



Queries, inserts and deletes: O(log(n)) time

Indexes are the single biggest tunable performance factor in MongoDB

Working with Indexes in MongoDB

10gen | the MongoDB company

How do I create indexes?

```
// Create an index if one does not exist
// If an index exists throw an error
db.recipes.createIndex({ main_ingredient: 1 })


// Create an index if it does not exists
// Does not raise errors if index exists
db.recipes.ensureIndex({ main_ingredient: 1 })
```

* 1 means ascending, -1 descending

What can be indexed?

```
// Multiple fields (compound indexes)
db.recipes.ensureIndex({
    main_ingredient: 1,
    calories: -1
})

// Arrays of values (multikey indexes)
{
    name: 'Chicken Noodle Soup’,
    ingredients : ['chicken', 'noodles']
}

db.recipes.ensureIndex({ ingredients: 1 })
```

What can be indexed?

```
// Subdocuments
{
    name : 'Apple Pie',
    contributor: {
        name: 'Joe American',
        id: 'joea123'
    }
}

// Index field in a sub document
db.recipes.ensureIndex({ 'contributor.id': 1 })
```

How do I manage indexes?

```
// List a collection's indexes
db.recipes.getIndexes()
db.recipes.getIndexKeys()

// Drop a specific index
db.recipes.dropIndex({ ingredients: 1 })

// Drop all indexes and recreate them
db.recipes.reIndex()

// Default (unique) index on _id
```

Background Index Builds

```
// Index creation is a blocking operation that can take a long time
// Background creation yields to other operations
db.recipes.ensureIndex(
   { ingredients: 1 },
   { background: true }
)
```

Options

- Uniqueness constraints (unique, dropDups)
- Sparse Indexes
- Geospatial (2d) Indexes
- TTL Collections (expireAfterSeconds)

Unique Indexes/Constraints

```
// Only one recipe can have a given value for name
db.recipes.ensureIndex( { name: 1 }, { unique: true } )


// Force index on collection with duplicate recipe names – drop the duplicates
db.recipes.ensureIndex(
   { name: 1 },
   { unique: true, dropDups: true }
)
```

* dropDups is probably never what you want

Sparse Indexes

```
// Only documents with field calories will be indexed
db.recipes.ensureIndex(
   { calories: -1 },
   { sparse: true }
)
// If present field calories should be unique
db.recipes.ensureIndex(
   {calories: 1 },
   { unique: true, sparse: true }
)
```

Geospatial Indexes

```
// Add latitude, longitude coordinates
{
    name: '10gen Palo Alto',
    loc: [ 37.449157, -122.158574 ]
}
// Index the coordinates
db.locations.ensureIndex( { loc : '2d' } )

// Query for locations 'near' a particular coordinate
db.locations.find({
    loc: { $near: [ 37.4, -122.3 ] }
})
```

TTL Collections

```
// Documents must have a BSON UTC Date field
{ 'status' : ISODate('2012-10-12T05:24:07.211Z'), ... }

// Documents are removed after 'expireAfterSeconds' seconds
db.recipes.ensureIndex(
   { submitted_date: 1 },
   { expireAfterSeconds: 3600 }
)
```

Limitations

- Collections can not have > 64 indexes.
- Index keys can not be > 1024 bytes (1K).
- The name of an index, including the namespace, must be < 128 characters.
- Queries can only use 1 index*
- Indexes have storage requirements, and impact the performance of writes.
- In memory sort (no-index) limited to 32mb of return data.

Optimize Your Queries

Profiling Slow Ops

db.setProfilingLevel(*n* , *slowms=100ms*)

n=0 profiler off

n=1 record operations longer than *slowms*

n=2 record all queries

db.system.profile.find()

* The profile collection is a capped collection

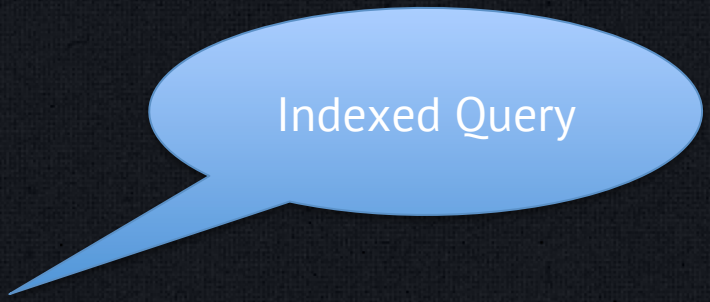
The Explain Plan (Pre Index)

```
db.recipes.find( { calories:
   { $lt : 40 } }
).explain( )
{
  "cursor" : "BasicCursor" ,
  "n" : 42,
  "nscannedObjects” : 12345
  "nscanned" : 12345,
   ...
  "millis" : 356,
   ...
}
```

* Doesn’t use cached plans, re-evals and resets cache

The Explain Plan (Post Index)

```
db.recipes.find( { calories:
   { $lt : 40 } }
).explain( )
{
  "cursor" : "BtreeCursor calories_-1" ,
  "n" : 42,
  "nscannedObjects": 42
  "nscanned" : 42,
   ...
  "millis" : 0,
   ...
}
```



* Doesn't use cached plans, re-evals and resets cache

The Query Optimizer

- For each "type" of query, MongoDB periodically tries *all* useful indexes
- Aborts the rest as soon as one plan wins
- The winning plan is cached for each “type” of query
  - Up to 1000 writes
  - Change in indexes

Manually Select Index to Use

```
// Tell the database what index to use
db.recipes.find({
    calories: { $lt: 1000 } }
).hint({ _id: 1 })

// Tell the database to NOT use an index
db.recipes.find(
    { calories: { $lt: 1000 } }
).hint({ $natural: 1 })
```

Use Indexes to Sort Query Results

```
// Given the following index
db.collection.ensureIndex({ a:1, b:1 , c:1, d:1 })

// The following query and sort operations can use the index
db.collection.find( ).sort({ a:1 })
db.collection.find( ).sort({ a:1, b:1 })

db.collection.find({ a:4 }).sort({ a:1, b:1 })
db.collection.find({ b:5 }).sort({ a:1, b:1 })
```

Indexes that won't work for sorting query results

```
// Given the following index
db.collection.ensureIndex({ a:1, b:1, c:1, d:1 })

// These can not sort using the index
db.collection.find( ).sort({ b: 1 })
db.collection.find({ b: 5 }).sort({ b: 1 })
```

Covered Index Queries

```
// MongoDB can return data from just the index
db.recipes.ensureIndex({ main_ingredient: 1, name: 1 })

// Return only the ingredients field
db.recipes.find(
   { main_ingredient: 'chicken' },
   { _id: 0, name: 1 }
)

// indexOnly will be true in the explain plan
db.recipes.find(
   { main_ingredient: 'chicken' },
   { _id: 0, name: 1 }
).explain()
{
   "indexOnly": true,
}
```

Absent or suboptimal indexes are the most common avoidable MongoDB performance problem.

Avoiding Common Mistakes

Trying to Use Multiple Indexes

```
// MongoDB can only use one index for a query
db.collection.ensureIndex({ a: 1 })
db.collection.ensureIndex({ b: 1 })

// Only one of the above indexes is used
db.collection.find({ a: 3, b: 4 })
```

Compound Key Mistakes

```
// Compound key indexes are very effective
db.collection.ensureIndex({ a: 1, b: 1, c: 1 })

// But only if the query is a prefix of the index

// This query can't use the index
db.collection.find({ c: 2 })

// ...but this query can
db.collection.find({ a: 3, b: 5 })
```

Low Selectivity Indexes

```
db.collection.distinct('status')
[ 'new', 'processed' ]

db.collection.ensureIndex({ status: 1 })

// Low selectivity indexes provide little benefit
db.collection.find({ status: 'new' })

// Better
db.collection.ensureIndex({ status: 1, created_at: -1 })
db.collection.find(
    { status: 'new' }
).sort({ created_at: -1 })
```

Regular Expressions

```
db.users.ensureIndex({ username: 1 })

// Left anchored regex queries can use the index
db.users.find({ username: /^joe smith/ })

// But not generic regexes
db.users.find({username: /smith/ })

// Or case insensitive queries
db.users.find({ username: /Joe/i })
```

Choosing the right indexes is one of the most important things you can do as a MongoDB developer so take the time to get your indexes right!

#CodeMash

Thank you

Sridhar Nanjundeswaran

Engineer, 10gen

@snanjund

10gen | the MongoDB company