Partial Index

A **partial index** is an index that contains only a subset of values based on a filter rule. So, in the case of the unevenly distributed boolean field, we can create an index on it specifying that we want to consider only the **false** values. This way we avoid recording the remaining 98% of useless **true** entries. The index will be smaller, we’ll save disk and memory space, and the most frequent writes – when entering the **true** values – won’t initiate the index management activity. As a result, we won’t have lots of penalties during writes but we’ll have a useful index when searching the **false** values.

Let’s say that, when you have an uneven distribution, the most relevant searches are the ones for the minority of the values. This is in general the scenario for real applications.

Let’s see now how to create a Partial Index.

First, let’s create a collection with one million random documents. Each document contains a boolean field generated by the javascript function *randomBool()*. The function generates a **false** value in 5% of the documents, in order to have an uneven distribution. Then, test the number of false values in the collection.

> function randomBool() { var bool = true; var random\_boolean = Math.random() >= 0.95; if(random\_boolean) { bool = false }; return bool; }

> for (var i = 1; i <= 1000000; i++) { db.test.insert( { \_id: i, name: "name"+i, flag: randomBool() } ) }

WriteResult({ "nInserted" : 1 })

> db.test.find().count()

1000000

> db.test.find( { flag: false } ).count()

49949

Create the index on the flag field and look at the index size using db.test.stats().

> db.test.createIndex( { flag: 1 } )

{ "createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1 }

> db.test.stats().indexSizes

{ "\_id\_" : 13103104, "flag\_1" : 4575232 }

The index we created is 4575232 bytes.

Test some simple queries to extract the documents based on the flag value and take a look at the index usage and the execution times. (For this purpose, we use an explainable object)

/ create the explainable object

> var exp = db.test.explain( "executionStats" )

// explain the complete collection scan

> exp.find( {  } )

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.test",

"indexFilterSet" : false,

"parsedQuery" : {

},

"winningPlan" : {

"stage" : "COLLSCAN",

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 1000000,

"executionTimeMillis" : 250,

"totalKeysExamined" : 0,

"totalDocsExamined" : 1000000,

"executionStages" : {

"stage" : "COLLSCAN",

"nReturned" : 1000000,

"executionTimeMillisEstimate" : 200,

"works" : 1000002,

"advanced" : 1000000,

"needTime" : 1,

"needYield" : 0,

"saveState" : 7812,

"restoreState" : 7812,

"isEOF" : 1,

"invalidates" : 0,

"direction" : "forward",

"docsExamined" : 1000000

}

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

// find the documents flag=true

> exp.find( { flag: true } )

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.test",

"indexFilterSet" : false,

"parsedQuery" : {

"flag" : {

"$eq" : true

}

},

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

"stage" : "IXSCAN",

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[true, true]"

]

}

}

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 950051,

"executionTimeMillis" : 1028,

"totalKeysExamined" : 950051,

"totalDocsExamined" : 950051,

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 950051,

"executionTimeMillisEstimate" : 990,

"works" : 950052,

"advanced" : 950051,

"needTime" : 0,

"needYield" : 0,

"saveState" : 7422,

"restoreState" : 7422,

"isEOF" : 1,

"invalidates" : 0,

"docsExamined" : 950051,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 950051,

"executionTimeMillisEstimate" : 350,

"works" : 950052,

"advanced" : 950051,

"needTime" : 0,

"needYield" : 0,

"saveState" : 7422,

"restoreState" : 7422,

"isEOF" : 1,

"invalidates" : 0,

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[true, true]"

]

},

"keysExamined" : 950051,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0,

"seenInvalidated" : 0

}

}

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

// find the documents with flag=false

> exp.find( { flag: false } )

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.test",

"indexFilterSet" : false,

"parsedQuery" : {

"flag" : {

"$eq" : false

}

},

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

"stage" : "IXSCAN",

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[false, false]"

]

}

}

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 49949,

"executionTimeMillis" : 83,

"totalKeysExamined" : 49949,

"totalDocsExamined" : 49949,

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 49949,

"executionTimeMillisEstimate" : 70,

"works" : 49950,

"advanced" : 49949,

"needTime" : 0,

"needYield" : 0,

"saveState" : 390,

"restoreState" : 390,

"isEOF" : 1,

"invalidates" : 0,

"docsExamined" : 49949,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 49949,

"executionTimeMillisEstimate" : 10,

"works" : 49950,

"advanced" : 49949,

"needTime" : 0,

"needYield" : 0,

"saveState" : 390,

"restoreState" : 390,

"isEOF" : 1,

"invalidates" : 0,

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : false,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[false, false]"

]

},

"keysExamined" : 49949,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0,

"seenInvalidated" : 0

}

}

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

As expected, MongoDB does a COLLSCAN when looking for ***db.test.find( {} )***. The important thing here is that it takes 250 milliseconds for the entire collection scan.

In both the other cases – ***find ({flag:true})*** and find({flag:false}) – MongoDB uses the index. But let’s have a look at the execution times:

* for *db.test.find({flag:true})* is 1028 milliseconds. The execution time is more than the COLLSCAN. The index in this case is not useful. COLLSCAN should be preferable.
* for *db.test.find({flag:false})* is 83 milliseconds. This is good. The index in this case is very useful.

Now, create the partial index on the *flag* field. To do it we must use the **PartialFilterExpression** option on the [createIndex](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \t "_blank) command.

// drop the existing index

> db.test.dropIndex( { flag: 1} )

{ "nIndexesWas" : 2, "ok" : 1 }

// create the partial index only on the false values

> db.test.createIndex( { flag : 1 }, { partialFilterExpression : { flag: false } } )

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

// get the index size

> db.test.stats().indexSizes

{ "\_id\_" : 13103104, "flag\_1" : 278528 }

// create the explainalbe object

> var exp = db.test.explain( "executionStats" )

// test the query for flag=false

> exp.find({ flag: false })

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.test",

"indexFilterSet" : false,

"parsedQuery" : {

"flag" : {

"$eq" : false

}

},

"winningPlan" : {

"stage" : "FETCH",

"inputStage" : {

"stage" : "IXSCAN",

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : true,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[false, false]"

]

}

}

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 49949,

"executionTimeMillis" : 80,

"totalKeysExamined" : 49949,

"totalDocsExamined" : 49949,

"executionStages" : {

"stage" : "FETCH",

"nReturned" : 49949,

"executionTimeMillisEstimate" : 80,

"works" : 49950,

"advanced" : 49949,

"needTime" : 0,

"needYield" : 0,

"saveState" : 390,

"restoreState" : 390,

"isEOF" : 1,

"invalidates" : 0,

"docsExamined" : 49949,

"alreadyHasObj" : 0,

"inputStage" : {

"stage" : "IXSCAN",

"nReturned" : 49949,

"executionTimeMillisEstimate" : 40,

"works" : 49950,

"advanced" : 49949,

"needTime" : 0,

"needYield" : 0,

"saveState" : 390,

"restoreState" : 390,

"isEOF" : 1,

"invalidates" : 0,

"keyPattern" : {

"flag" : 1

},

"indexName" : "flag\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"flag" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : true,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"flag" : [

"[false, false]"

]

},

"keysExamined" : 49949,

"seeks" : 1,

"dupsTested" : 0,

"dupsDropped" : 0,

"seenInvalidated" : 0

}

}

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

// test the query for flag=true

> exp.find({ flag: true })

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.test",

"indexFilterSet" : false,

"parsedQuery" : {

"flag" : {

"$eq" : true

}

},

"winningPlan" : {

"stage" : "COLLSCAN",

"filter" : {

"flag" : {

"$eq" : true

}

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"executionStats" : {

"executionSuccess" : true,

"nReturned" : 950051,

"executionTimeMillis" : 377,

"totalKeysExamined" : 0,

"totalDocsExamined" : 1000000,

"executionStages" : {

"stage" : "COLLSCAN",

"filter" : {

"flag" : {

"$eq" : true

}

},

"nReturned" : 950051,

"executionTimeMillisEstimate" : 210,

"works" : 1000002,

"advanced" : 950051,

"needTime" : 49950,

"needYield" : 0,

"saveState" : 7812,

"restoreState" : 7812,

"isEOF" : 1,

"invalidates" : 0,

"direction" : "forward",

"docsExamined" : 1000000

}

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

We can notice the following:

* *db.test.find({flag:false})* uses the index and the execution time is more or less the same as before
* *db.test.find({flag:true})* doesn’t use the index. MongoDB does the COLLSCAN and the execution is better than before
* note the index size is only 278528 bytes. now A great saving in comparison to the complete index on flag. There won’t be overhead during the writes in the great majority of the documents.

Partial option on other index types

You can use the partialFilterExpression option even in compound indexes or other index types. Let’s see an example of a compound index.

Insert some documents in the students collection

db.students.insert( [

{ \_id:1, name: "John", class: "Math", grade: 10 },

{ \_id: 2, name: "Peter", class: "English", grade: 6 },

{ \_id: 3, name: "Maria" , class: "Geography", grade: 8 },

{ \_id: 4, name: "Alex" , class: "Geography", grade: 5},

{ \_id: 5, name: "George" , class: "Math", grade: 7 },

{ \_id: 6, name: "Tony" , class: "English", grade: 9 },

{ \_id: 7, name: "Sam" , class: "Math", grade: 6 },

{ \_id: 8, name: "Tom" , class: "English", grade: 5 }

])

Create a partial compound index on name and class fields for the grade greater or equal to 8.

> db.students.createIndex( { name: 1, class: 1  }, { partialFilterExpression: { grade: { $gte: 8} } } )

{

"createdCollectionAutomatically" : false,

"numIndexesBefore" : 1,

"numIndexesAfter" : 2,

"ok" : 1

}

Notice that the grade field doesn’t necessarily need to be part of the index.

## Query coverage

Using the students collection, we want now to show when a partial index can be used.

The important thing to remember is that a partial index is “partial”. It means that it doesn’t contain all the entries.

In order for MongoDB to use it the conditions in the query must include an expression on the filter field and the selected documents must be a subset of the index.

Let’s see some examples.

The following query can use the index because we are selecting a subset of the partial index.

> db.students.find({name:"Tony", grade:{$gt:8}})

{ "\_id" : 6, "name" : "Tony", "class" : "English", "grade" : 9 }

// let's look at the explain

> db.students.find({name:"Tony", grade:{$gt:8}}).explain()

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.students",

"indexFilterSet" : false,

"parsedQuery" : {

"$and" : [

{

"name" : {

"$eq" : "Tony"

}

},

{

"grade" : {

"$gt" : 8

}

}

]

},

"winningPlan" : {

"stage" : "FETCH",

"filter" : {

"grade" : {

"$gt" : 8

}

},

"inputStage" : {

"stage" : "IXSCAN",

"keyPattern" : {

"name" : 1,

"class" : 1

},

"indexName" : "name\_1\_class\_1",

"isMultiKey" : false,

"multiKeyPaths" : {

"name" : [ ],

"class" : [ ]

},

"isUnique" : false,

"isSparse" : false,

"isPartial" : true,

"indexVersion" : 2,

"direction" : "forward",

"indexBounds" : {

"name" : [

"[\"Tony\", \"Tony\"]"

],

"class" : [

"[MinKey, MaxKey]"

]

}

}

},

"rejectedPlans" : [ ]

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

The following query cannot use the index because the condition on grade > 5 is not selecting a subset of the partial index. So the COLLSCAN is needed.

> db.students.find({name:"Tony", grade:{$gt:5}}).explain()

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.students",

"indexFilterSet" : false,

"parsedQuery" : {

"$and" : [

{

"name" : {

"$eq" : "Tony"

}

},

{

"grade" : {

"$gt" : 5

}

}

]

},

"winningPlan" : {

"stage" : "COLLSCAN",

"filter" : {

"$and" : [

{

"name" : {

"$eq" : "Tony"

}

},

{

"grade" : {

"$gt" : 5

}

}

]

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

Even the following query cannot use the index. As we said the grade field is not part of the index. The simple condition on grade is not sufficient.

> db.students.find({grade:{$gt:8}}).explain()

{

"queryPlanner" : {

"plannerVersion" : 1,

"namespace" : "test.students",

"indexFilterSet" : false,

"parsedQuery" : {

"grade" : {

"$gt" : 8

}

},

"winningPlan" : {

"stage" : "COLLSCAN",

"filter" : {

"grade" : {

"$gt" : 8

}

},

"direction" : "forward"

},

"rejectedPlans" : [ ]

},

"serverInfo" : {

"host" : "ip-172-30-2-181",

"port" : 27017,

"version" : "4.0.4",

"gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"

},

"ok" : 1

}

Sparse Index

A **sparse index** is an index that contains entries only for the documents that have the indexed field.

Since MongoDB is a schemaless database, not all the documents in a collection will necessarily contain the same fields. So we have two options when creating an index:

* create a regular “non-sparse” index
  + the index contains as many entries as the documents
  + the index contains entries as null for all the documents without the indexed field
* create a sparse index
  + the index contains as many entries as the documents with the indexed field

We call it “sparse” because it doesn’t contain all the documents of the collection.

The main advantage of the sparse option is to reduce the index size.

Here’s how to create a sparse index:

db.people.createIndex( { city: 1 }, { sparse: true } )

Sparse indexes are a subset of partial indexes. In fact you can emulate a sparse index using the following definition of a partial.

db.people.createIndex(

{city: 1},

{ partialFilterExpression: {city: {$exists: true} } }

)

Partial indexing is a great feature in MongoDB. You should consider using it to achieve the following advantages:

* have smaller indexes
* save disk and memory space
* improve writes performance

You are strongly encouraged to consider partial indexes if you have one or more of these use cases:

* you run queries on a boolean field with an uneven distribution, and you look mostly for the less frequent value
* you have a low cardinality field and the majority of the queries look for a subset of the values
* the majority of the queries look for a limited subset of the values in a field
* you don’t have enough memory to store very large indexes – for example, you have a lot of page evictions from the WiredTiger cache