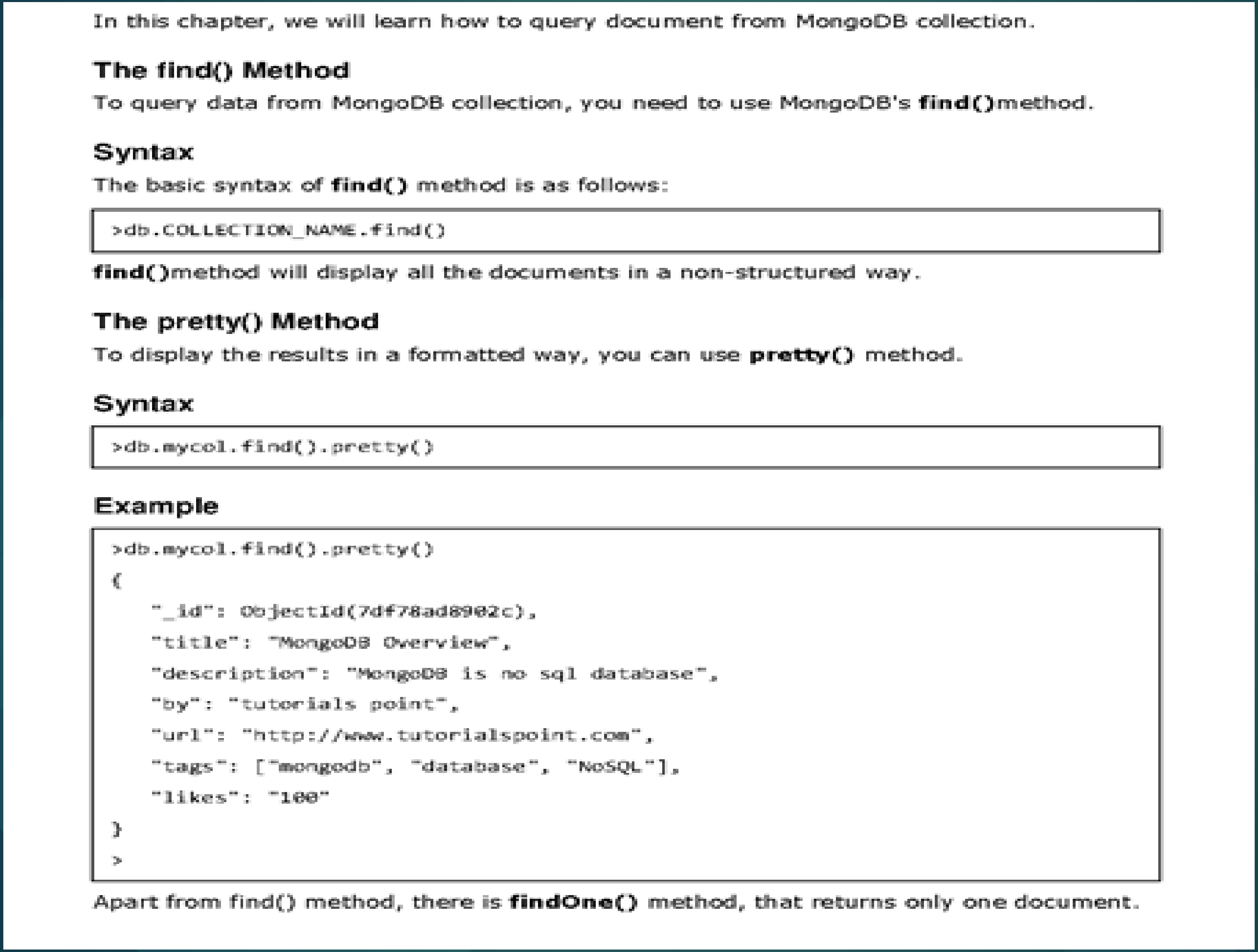
FIND DOCUMENTS IN A COLLECTION

1. FIND
2. FIND ONE
3. COUNT RESULTS
4. FILTER DOCUMENTS
5. BOOLEAN OPERATIONS (AND,OR)
6. PROJECTION
7. LIMIT RESULTS
8. SKIPPING RESULTS
9. SORTING RESULTS
10. INDEXING AND OPTIONS
11. COVERING QUERIES
12. HINTS

Find method



In this chapter, we will learn how to query document from MongoDB collection.

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()**method.

Syntax

The basic syntax of **find()** method is as follows:

```
>db.COLLECTION_NAME.find()
```

find()method will display all the documents in a non-structured way.

The pretty() Method

To display the results in a formatted way, you can use **pretty()** method.

Syntax

```
>db.mycol.find().pretty()
```

Example

```
>db.mycol.find().pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Apart from find() method, there is **findOne()** method, that returns only one document.

FindOne method

db.collection.findOne(*query*, *projection)*

Returns one document that satisfies the specified query criteria.
If multiple documents satisfy the query, this method returns
the first document according to the natural order which reflects
the order of documents on the disk.

Example:

```
db.bios.findOne()
```

Counting results

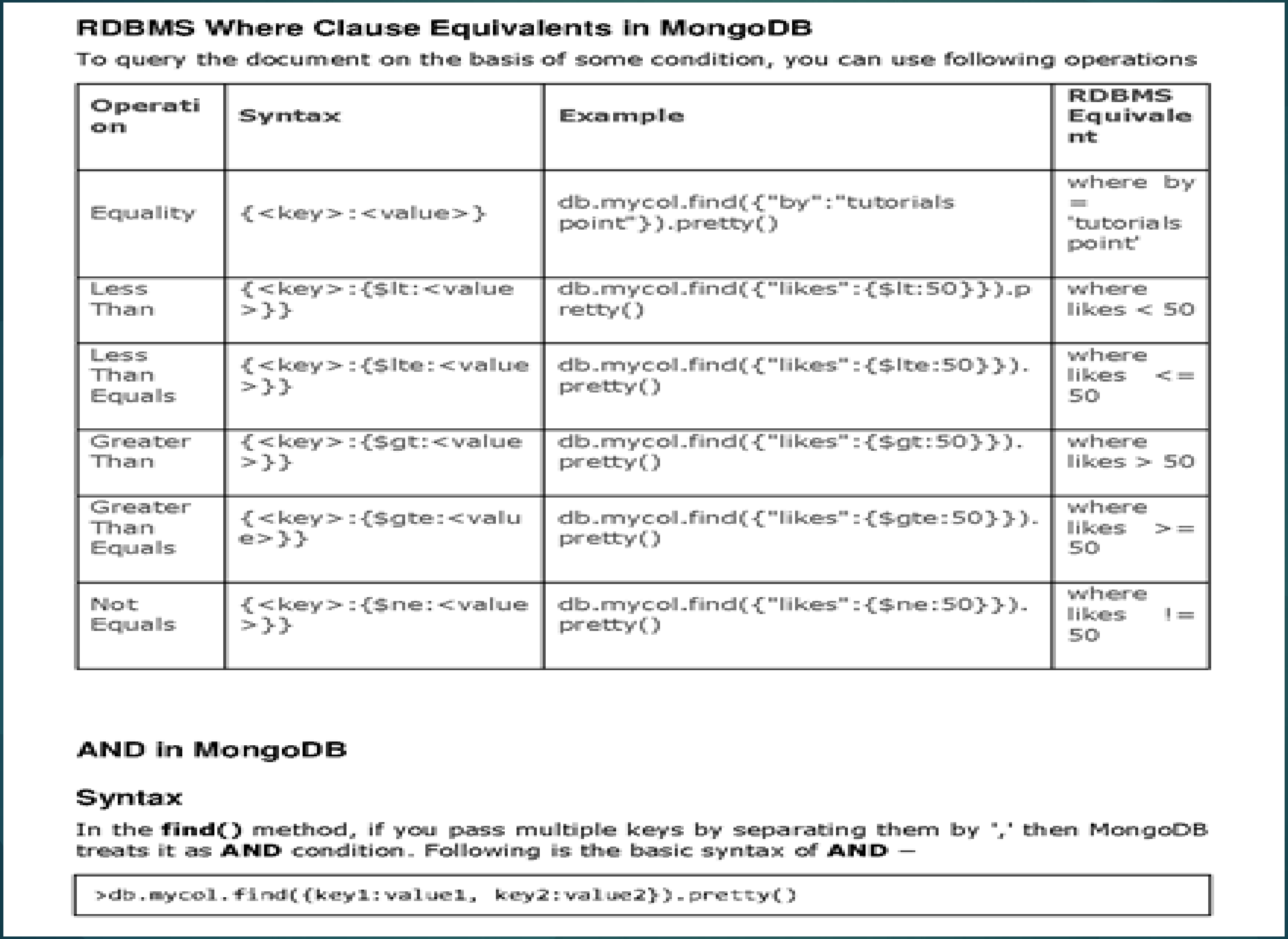
db.collection.count()

Returns the count of documents that would match a find() query. The db.collection.count() method does not perform the find() operation but instead counts and returns the number of results that match a query.

Example:

```
db.orders.count()
```

RDBMS WHERE CLAUSES IN MONGO DB



RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

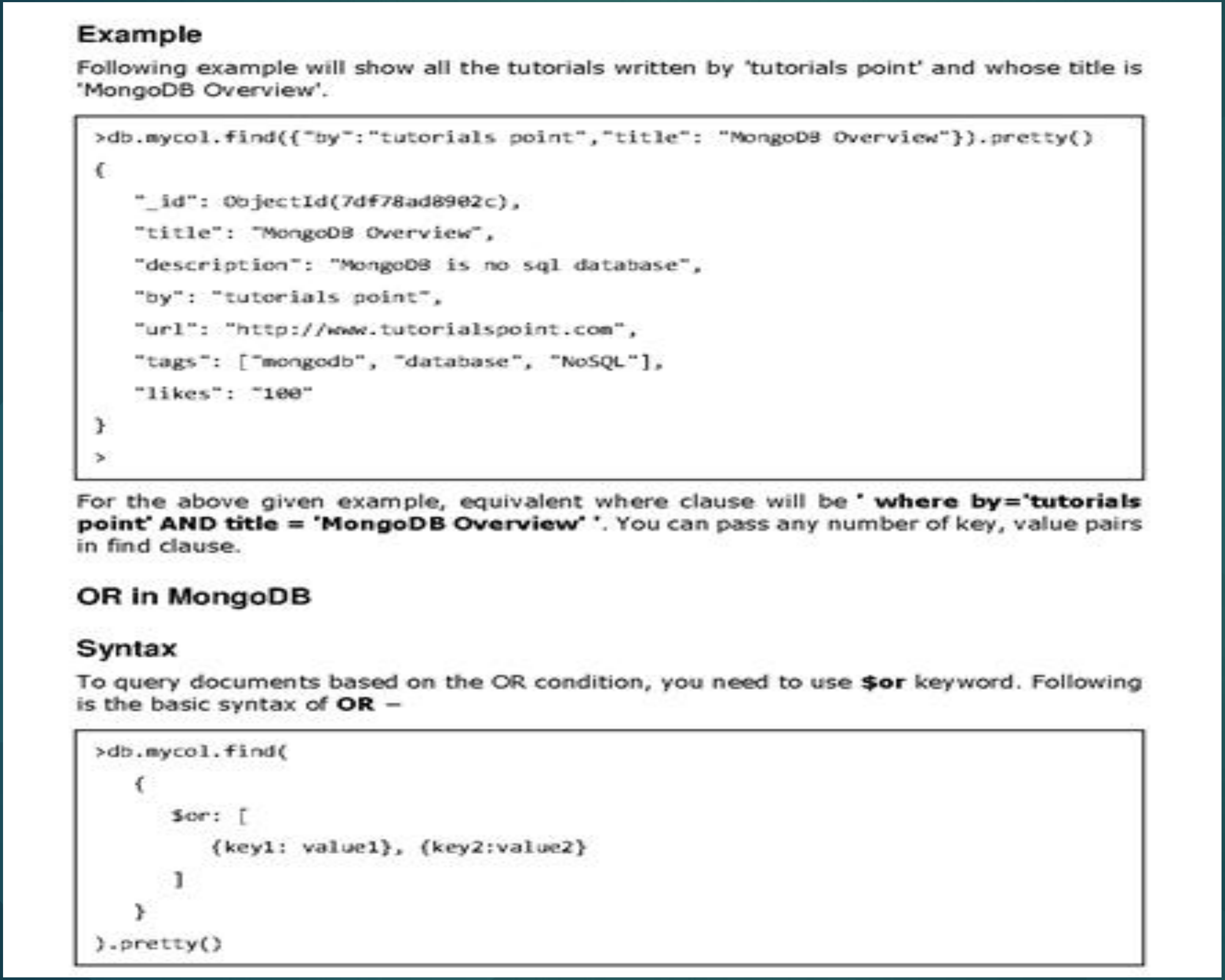
AND in MongoDB

Syntax

In the **find()** method, if you pass multiple keys by separating them by ',' then MongoDB treats it as **AND** condition. Following is the basic syntax of **AND** –

```
>db.mycol.find({key1:value1, key2:value2}).pretty()
```

AND CLAUSE EXAMPLE



Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
>db.mycol.find({"by":"tutorials point","title": "MongoDB Overview"}).pretty()
{
    "_id": ObjectId(7df78ad8902c),
    "title": "MongoDB Overview",
    "description": "MongoDB is no sql database",
    "by": "tutorials point",
    "url": "http://www.tutorialspoint.com",
    "tags": ["mongodb", "database", "NoSQL"],
    "likes": "100"
}
>
```

For the above given example, equivalent where clause will be ' **where by='tutorials point' AND title = 'MongoDB Overview'** '. You can pass any number of key, value pairs in find clause.

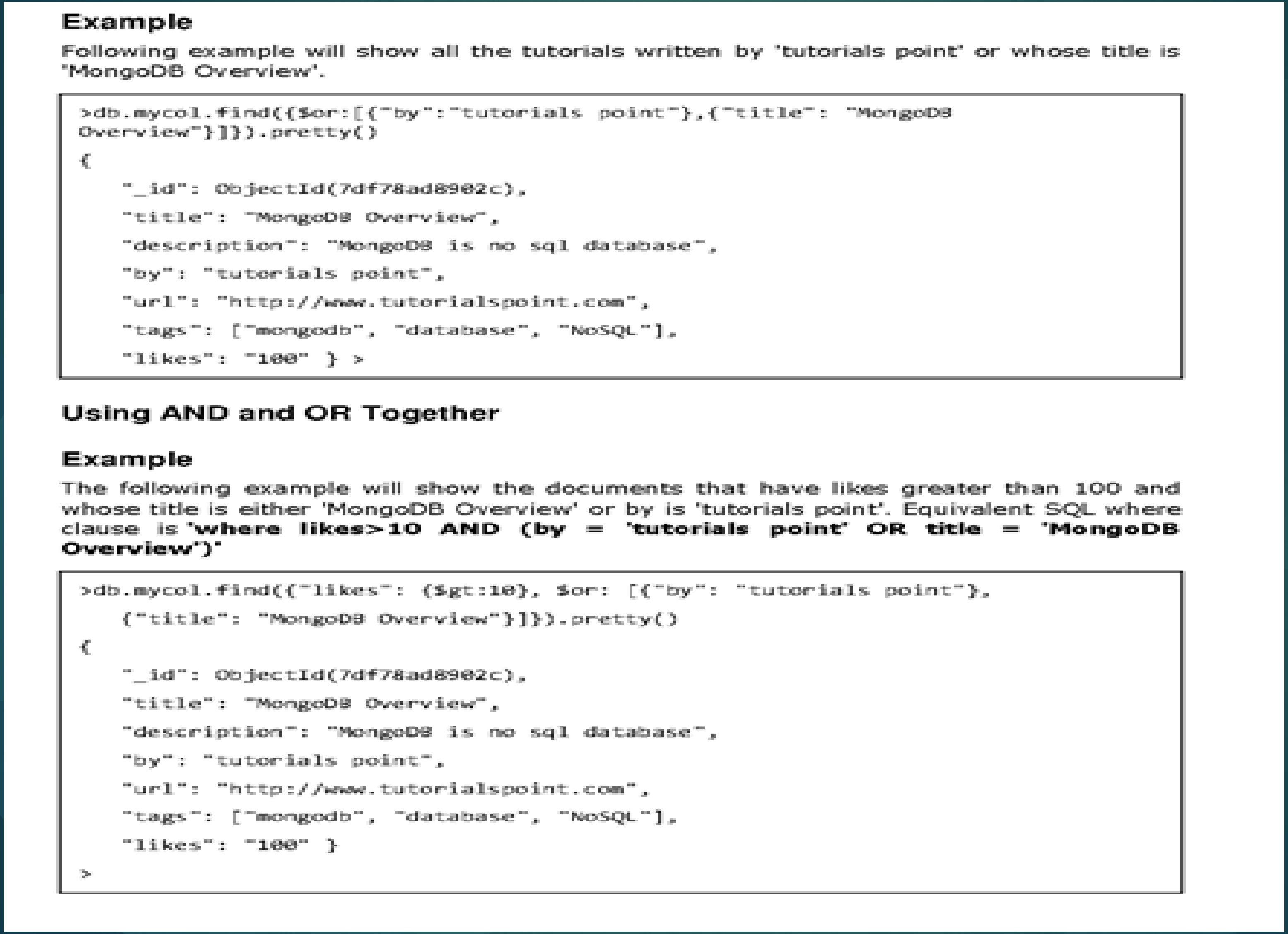
OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** –

```
>db.mycol.find(
    {
       $or: [
          {key1: value1}, {key2:value2}
       ]
    }
).pretty()
```

Or clause example



Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB
Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100" } >
```

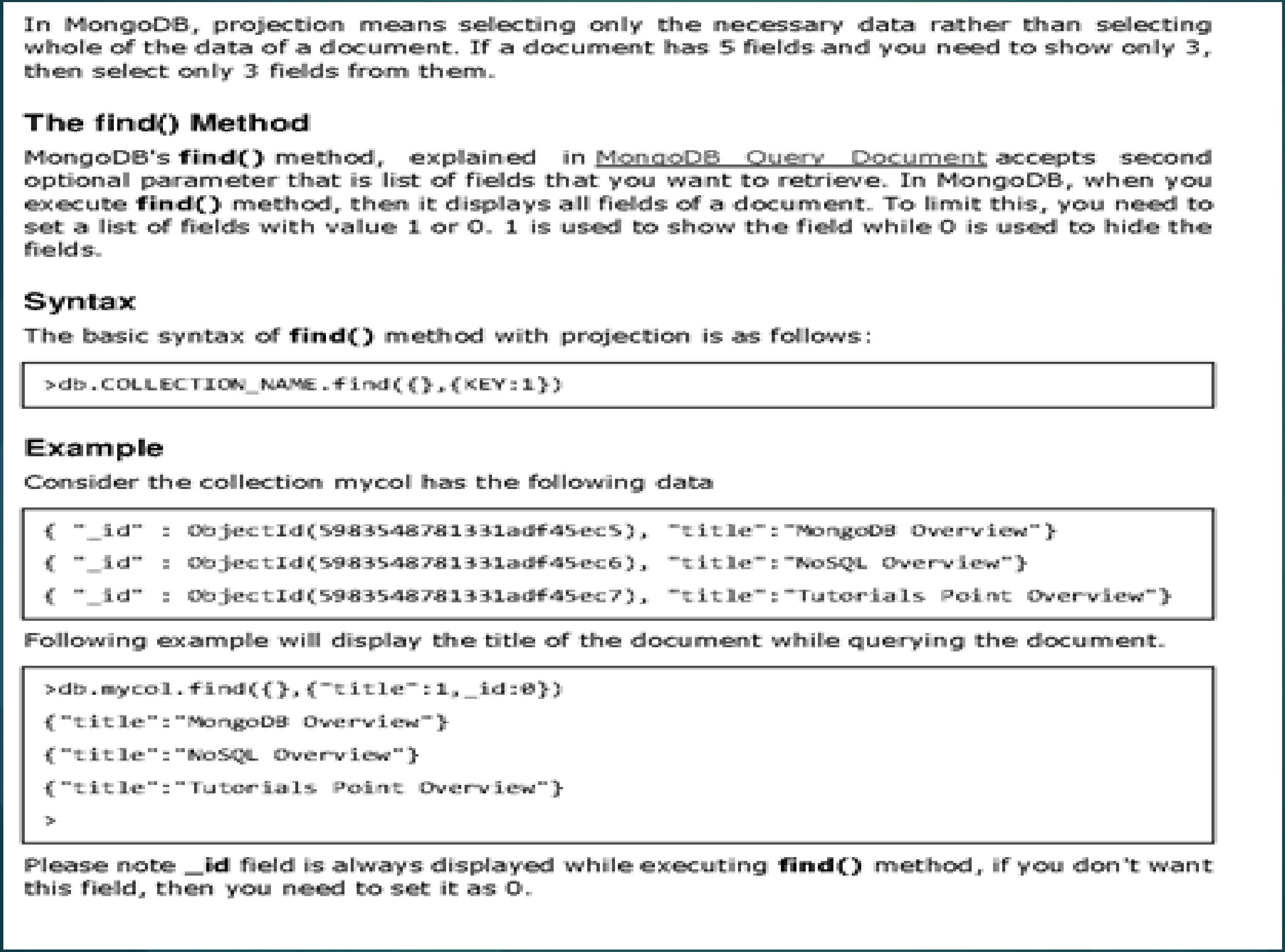
Using AND and OR Together

Example

The following example will show the documents that have likes greater than 100 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100" }
>
```

projection



In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method

MongoDB's **find()** method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute **find()** method, then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax

The basic syntax of **find()** method with projection is as follows:

```
>db.COLLECTION_NAME.find({},{KEY:1})
```

Example

Consider the collection mycol has the following data

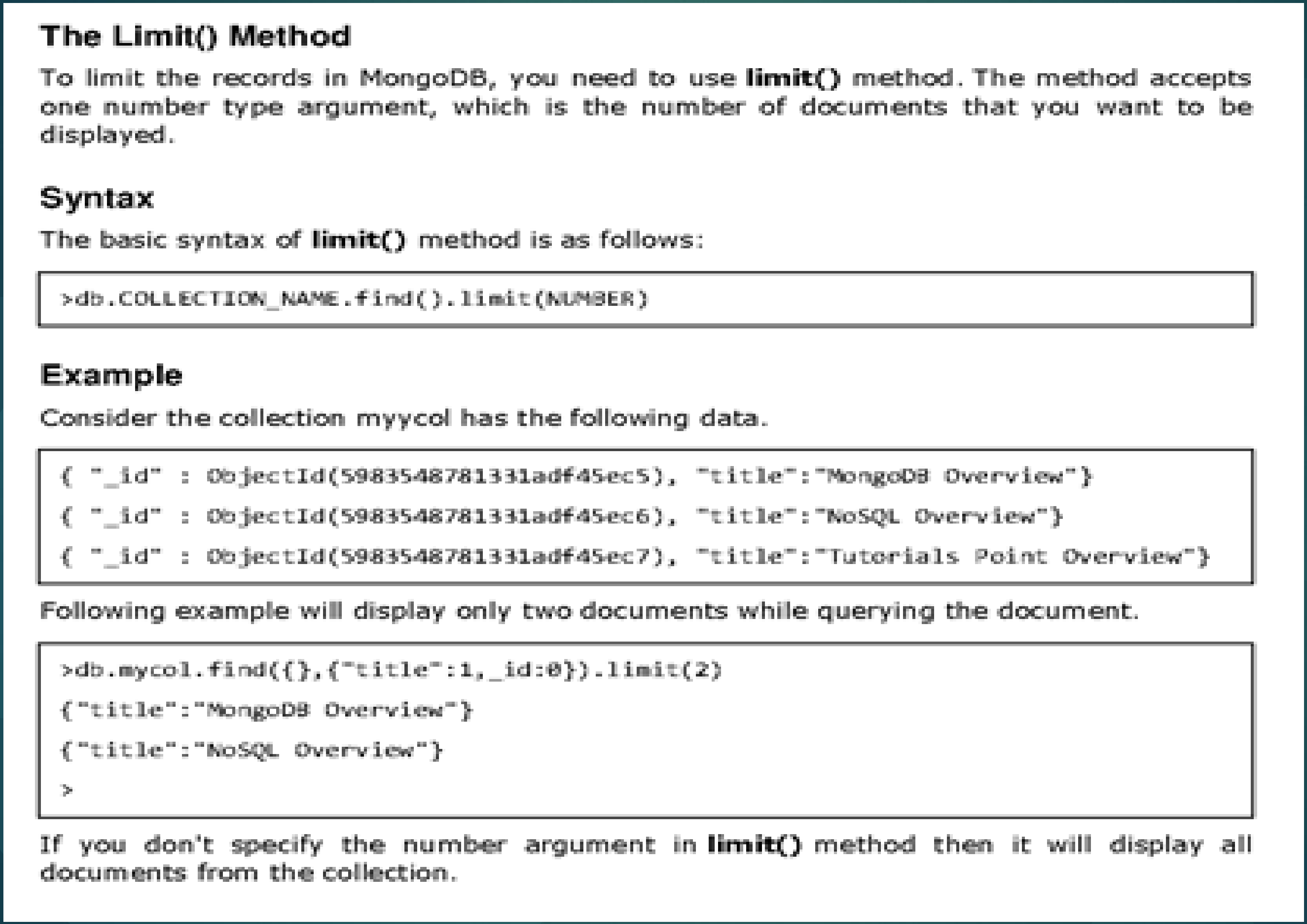
```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display the title of the document while querying the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

Please note **_id** field is always displayed while executing **find()** method, if you don't want this field, then you need to set it as 0.

LIMIT NUMBER OF DOCUMENTS RETURNED – LIKE RDBMS TOP clause



The Limit() Method

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of **limit()** method is as follows:

```
>db.COLLECTION_NAME.find().limit(NUMBER)
```

Example

Consider the collection myycol has the following data.

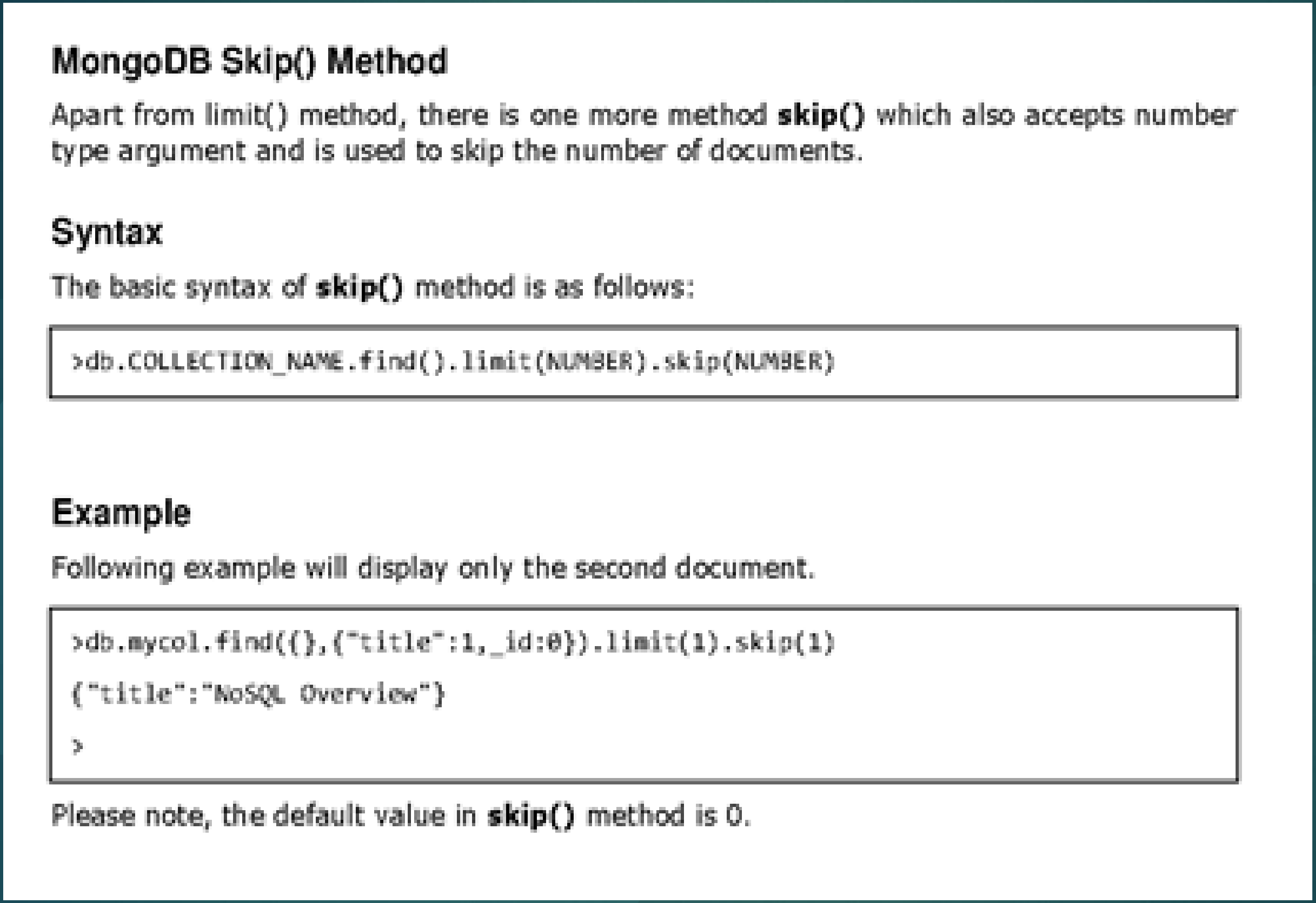
```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

If you don't specify the number argument in **limit()** method then it will display all documents from the collection.

Skipping documents



MongoDB Skip() Method

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows:

```
>db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

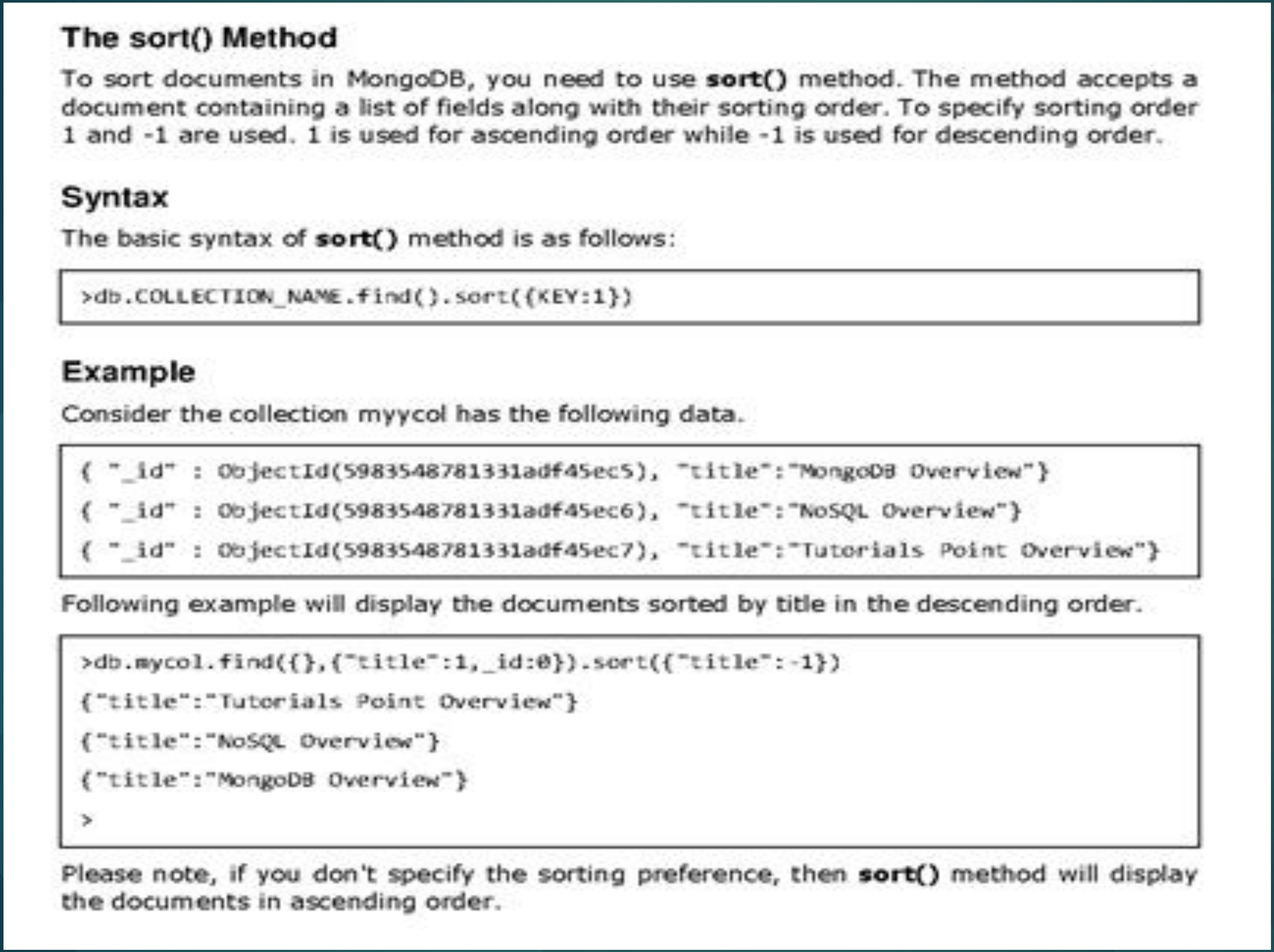
Example

Following example will display only the second document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
{"title":"NoSQL Overview"}
>
```

Please note, the default value in **skip()** method is 0.

SORTING DOCUMENTS – LIKE RDBMS order by clause



The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

Syntax

The basic syntax of **sort()** method is as follows:

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

Example

Consider the collection myycol has the following data.

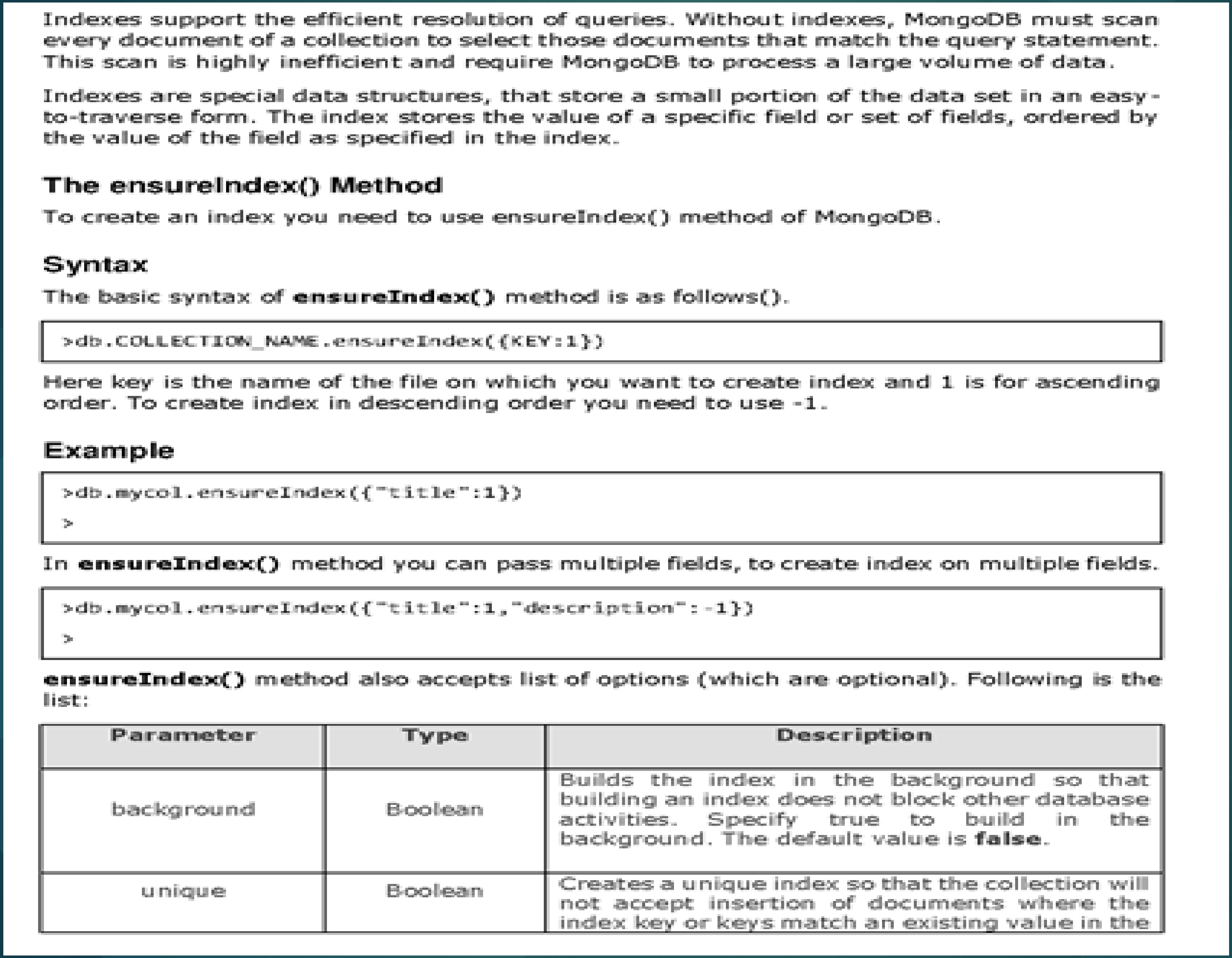
```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Please note, if you don't specify the sorting preference, then **sort()** method will display the documents in ascending order.

Indexes in MONGO DB



Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The ensureIndex() Method

To create an index you need to use ensureIndex() method of MongoDB.

Syntax

The basic syntax of **ensureIndex()** method is as follows().

```
>db.COLLECTION_NAME.ensureIndex({KEY:1})
```

Here key is the name of the file on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.ensureIndex({"title":1})
>
```

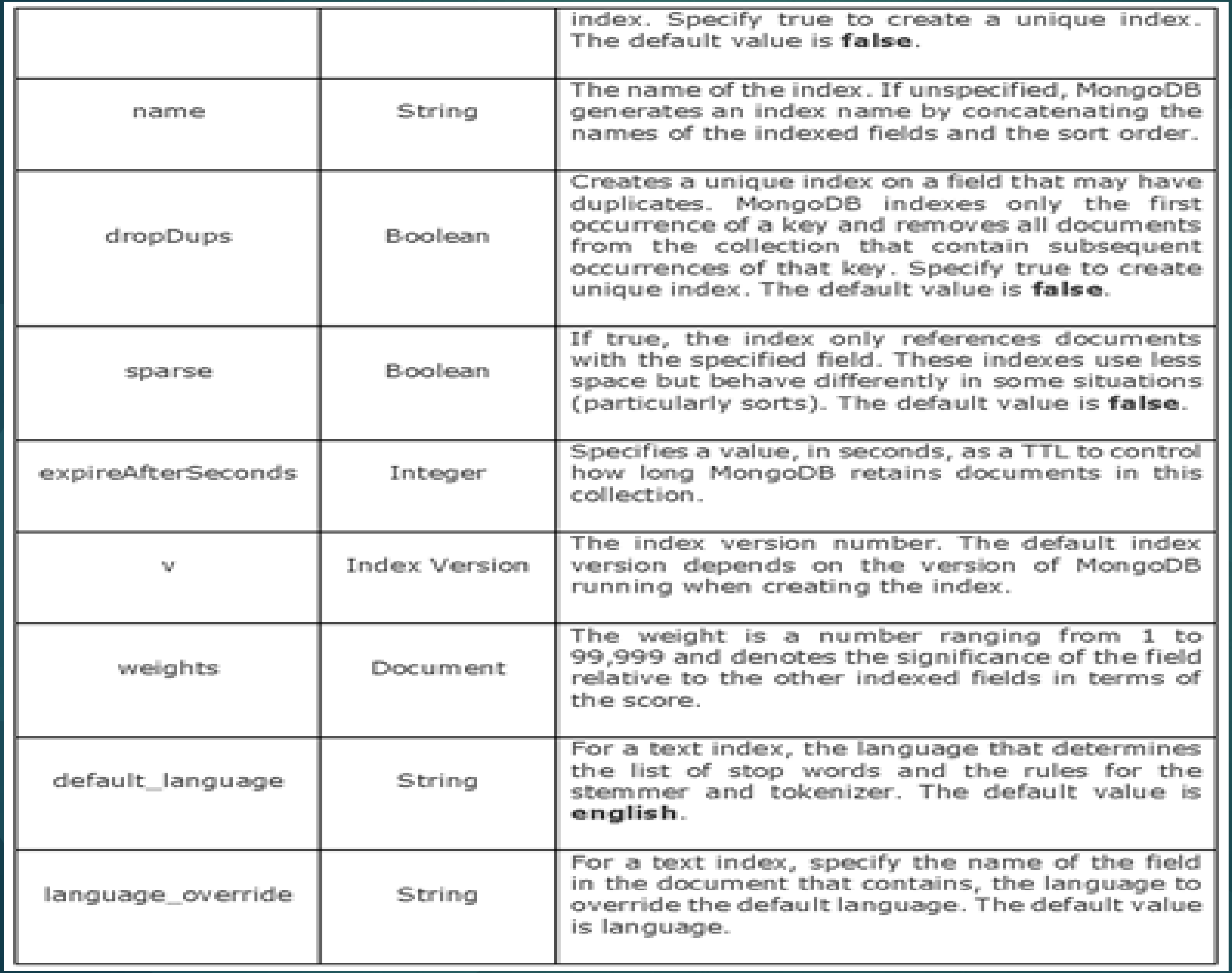
In **ensureIndex()** method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.ensureIndex({"title":1,"description":-1})
>
```

ensureIndex() method also accepts list of options (which are optional). Following is the list:

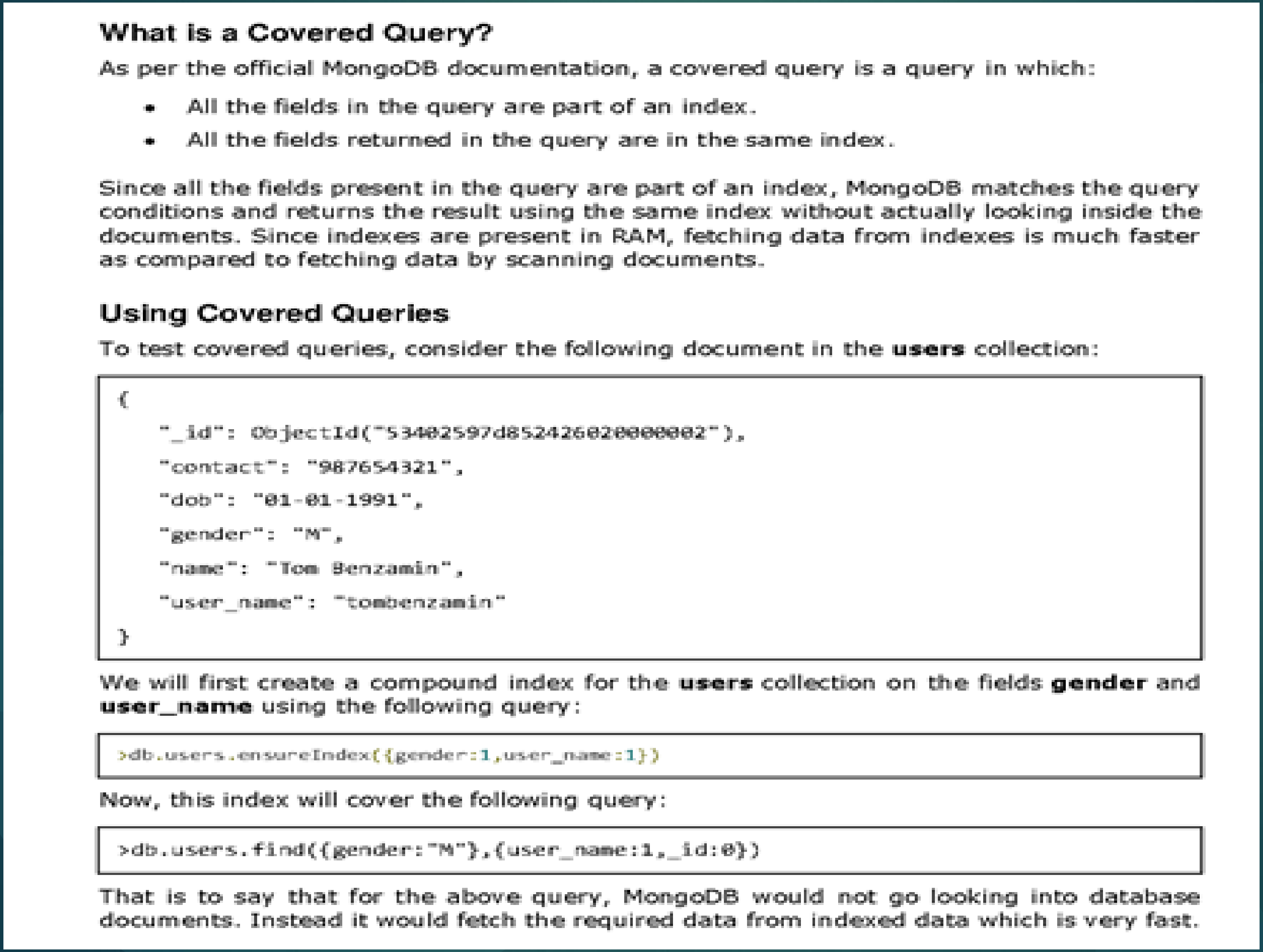
Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is **false**.
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the

Mongo DB Index options – continued



		index. Specify true to create a unique index. The default value is **false**.
name	String	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
dropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key. Specify true to create unique index. The default value is **false**.
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is **false**.
expireAfterSeconds	Integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.
v	Index Version	The index version number. The default index version depends on the version of MongoDB running when creating the index.
weights	Document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	String	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is **english**.
language_override	String	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

MONGO DB COVERED QUERY



What is a Covered Query?

As per the official MongoDB documentation, a covered query is a query in which:

- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.

Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries

To test covered queries, consider the following document in the **users** collection:

```
{
   "_id": ObjectId("53402597d852426020000002"),
   "contact": "987654321",
   "dob": "01-01-1991",
   "gender": "M",
   "name": "Tom Benzamin",
   "user_name": "tombenzamin"
}
```

We will first create a compound index for the **users** collection on the fields **gender** and **user_name** using the following query:

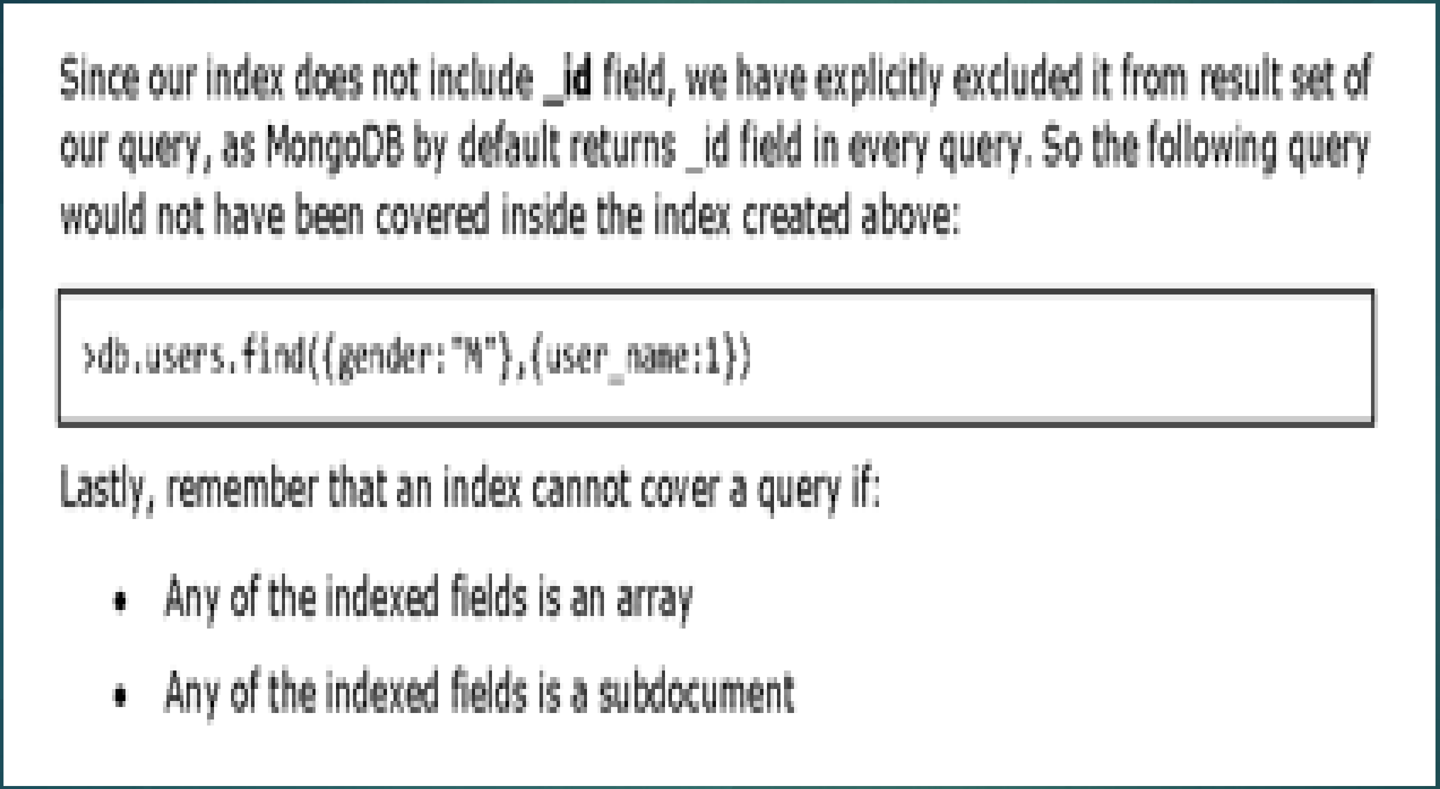
```
>db.users.ensureIndex({gender:1,user_name:1})
```

Now, this index will cover the following query:

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
```

That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.

Covered query - continued



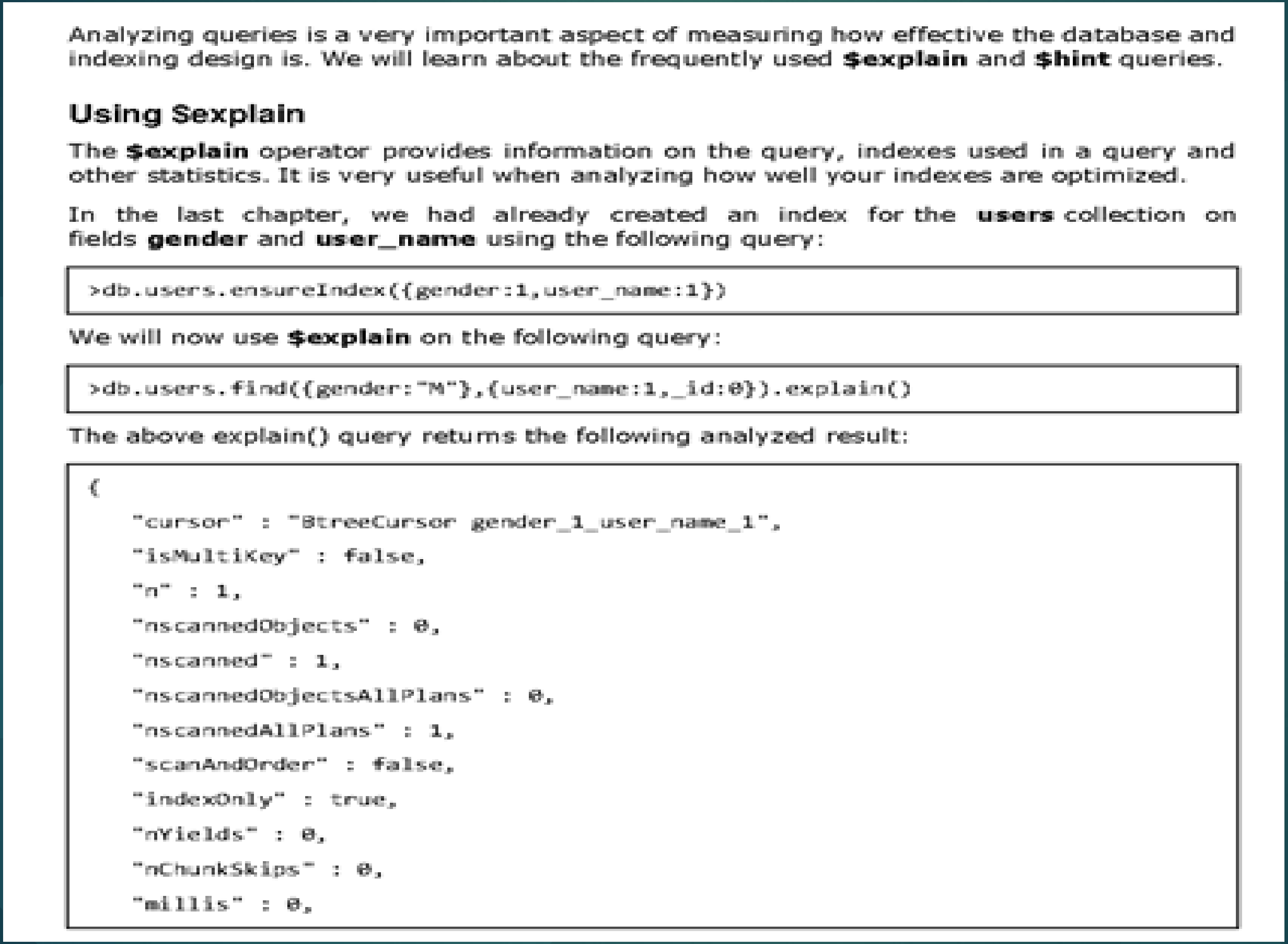
Since our index does not include **_id** field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above:

```
>db.users.find({gender:"M"},{user_name:1})
```

Lastly, remember that an index cannot cover a query if:

- Any of the indexed fields is an array
- Any of the indexed fields is a subdocument

ANALYZING A MONGO DB QUERY



Analyzing queries is a very important aspect of measuring how effective the database and indexing design is. We will learn about the frequently used **$explain** and **$hint** queries.

Using Sexplain

The **$explain** operator provides information on the query, indexes used in a query and other statistics. It is very useful when analyzing how well your indexes are optimized.

In the last chapter, we had already created an index for the **users** collection on fields **gender** and **user_name** using the following query:

```
>db.users.ensureIndex({gender:1,user_name:1})
```

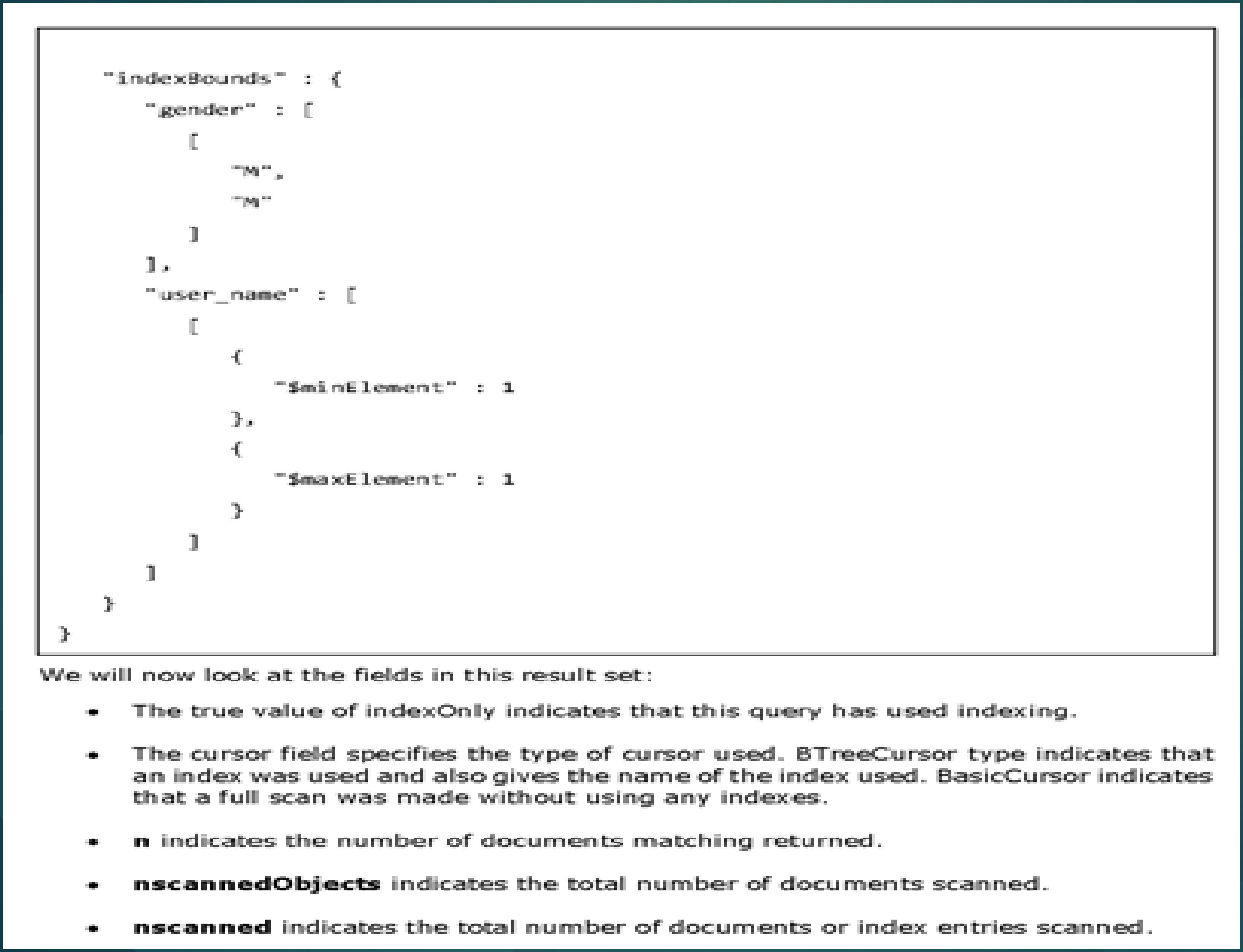
We will now use **$explain** on the following query:

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).explain()
```

The above explain() query returns the following analyzed result:

```
{
    "cursor" : "BtreeCursor gender_1_user_name_1",
    "isMultiKey" : false,
    "n" : 1,
    "nscannedObjects" : 0,
    "nscanned" : 1,
    "nscannedObjectsAllPlans" : 0,
    "nscannedAllPlans" : 1,
    "scanAndOrder" : false,
    "indexOnly" : true,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
```

ANALYZING QUERIES - CONTINUED



```
    "indexBounds" : {
        "gender" : [
            [
                "M",
                "M"
            ]
        ],
        "user_name" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ]
    }
}
```

We will now look at the fields in this result set:

- The true value of indexOnly indicates that this query has used indexing.
- The cursor field specifies the type of cursor used. BTreeCursor type indicates that an index was used and also gives the name of the index used. BasicCursor indicates that a full scan was made without using any indexes.
- **n** indicates the number of documents matching returned.
- **nscannedObjects** indicates the total number of documents scanned.
- **nscanned** indicates the total number of documents or index entries scanned.