THE UNIVERSITY OF SYDNEY

School of Computer Science
Dr. Ying Zhou

COMP5338: Advanced Data Models **Sem. 2/2020**

Week 4: MongoDB Indexing

15.09.2020

Learning Objectives

In this week's lab, we work on the same wikipedia data we used in week 2 and 3. We use the data set to observe MongoDB execute performance. In particular we focus on:

- Indexing and its impact
- Query execution statistics
- Query plan(s) evaluated for various queries

Question 1: Duplicating a collection

In this step we duplicate the `revisions` collection to create a new collection `revisionsWI`.

There are many ways to duplicate a collection. You may use write a JavaScript function to save every document of one collection into a new one. This should be used only for small collection as it transfers all documents in the collection to the client side then send it back to the server to be written into a new collection. The preferred and more efficient way is to use the `$out` stage in aggregation as shown below. The aggregation pipeline run the process entirely on the server side, it does not involve any network transmission.

```
db.revisions.aggregate([
 {$out: "revisionsWI"}
])
```

Question 2: Setting up indexes

Run the following commands to set up index on `revisionsWI` collection.

```
db.revisionsWI.createIndex({user:1})
db.revisionsWI.createIndex({timestamp:1})
db.revisionsWI.createIndex({title:1})
db.revisionsWI.createIndex({parsedcomment:"text"})
```

The collection `revisions` and `revisionsWI` have the same data, but different index structure. Both collections have primary index on the `_id` field. The collection `revisionsWI` has four secondary indexes on four different fields. The indexes on `user`, `timestamp`, `title` fields are regular B-Tree index. The last one is a special `text` index built on `parsedcomment` field.

You can see available indexes of any given collection from the left panel of Robo3T by expending the corresponding collection name. The following shell command will list detailed information about index size as well as data size:

```
db.revisionsWI.stats({scale:1024})
```

The parameter `{scale:1024}` indicates we want to see sizes in `KB` instead of `Byte`.

Run the command on both collections (revisions and revisionsWI) and compare the result. The number of document `count` and average object size `avgObjSize` should be the same for both collections. The `size` output displays the total size in memory of all records in a collection. This should be the same for both collections as well. The `storageSize` shows the total amount of storage allocated to a collection for document storage. The relation between memory size and storage size depends on the actual storage engine used. MongoDB uses `WiredTiger` as default storage engine since version 3.4. A few features of WiredTiger may help to explain the numbers observed. `WiredTiger` supports compression for all collections and indexes, you are likely to see storage size smaller than the memory size. This could be the case of `revisionsWI` collection. `WiredTiger` uses multiversion concurrency control (MVCC) and snapshots are written to disk as checkpoints at intervals of 60 seconds. These are counted in storage size. You may notice that the storage size of `revisions` is much larger than the storage size of `revisionsWI`. This is because week 2 and week 3 lab exercises include many update operations on `revisions` collection. It ended up with more checkpoints written. The newly created `revisionsWI` collection has not been through many updates yet.

The shell command `db.collection.stats({scale:1024})` shows details of index size as well. These include the total index size and the size of individual index. The collection `revisions` contains only one index and the size is relatively small. The `revisionsWI` contains five index, the total index size is about the same as the data size in memory. The text index takes a lot more spaces than the other index. Those information can help you to decide if it is worthwhile to create certain index.

Question 3: Indexing and Query Performance

The `explain` method on `cursor` object can be used to see how those indexes are used in query execution. Run the following four pairs of commands to compare the query execution plan on `revisions` and `revisionsWI`. Each pair should return exactly the same result set, with possible different execution performance. If you want to see the actual result, run each query without the `explain` command first.

a) Querying `_id` field .

```
db.revisions.find({"_id" : ObjectId("5f53204cc89636b3c92e0851")})
    .explain("executionStats");
db.revisionsWI.find({"_id" : ObjectId("5799843ee2cbe65d76ed9133")})
    .explain("executionStats");
```

This pair search for a particular revision with a given `_id` value (**note: replace the ObjectId value with a value that exists in your collection!**). The `_id` field is the primary key with a default index, both queries will use the index.

The `explain` method provides detailed information on the query plan. By default, it shows the `queryPlanner` information. The `executionStats` argument allows to output query execution information such as the time used to answer a query and the number of documents scanned. If an index is used, it also shows the bounds of index. Figure 1 shows the high level output of explain method in Robo3T. All three elements: `QueryPlanner`, `executionStats` and`sererinfo` can be expanded to see details. Detailed description of `explain` method can be found in MongoDB document `http://docs.mongodb.org/manual/reference/method/cursor.explain/`. The page also contains links to each individual element.

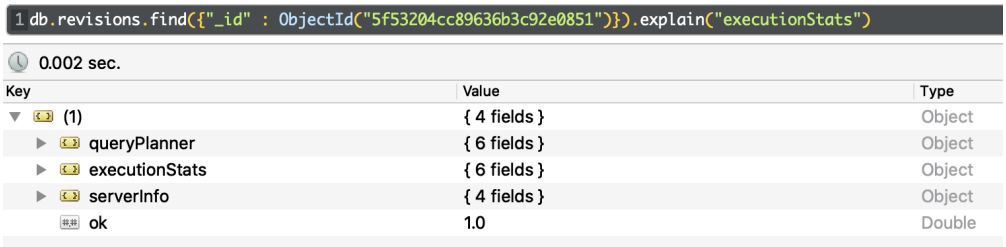

Figure 1: Explain High Level Output

The primary based id search is the only possible plan, there is no other plan, so the `rejectedPlans` field under `queryPlanner` is an empty array (see figure 2).

The `executionStats` shows that the total number of document returned is 1 (`nReturned`); the `keysExamined` is 1 and the `docsExamined` is 1 as well (see figure 3). This means the results is located directly by inspecting the index. The overall query execution plan is the same for both collections.

Figure 2: Search by _id Query Planner

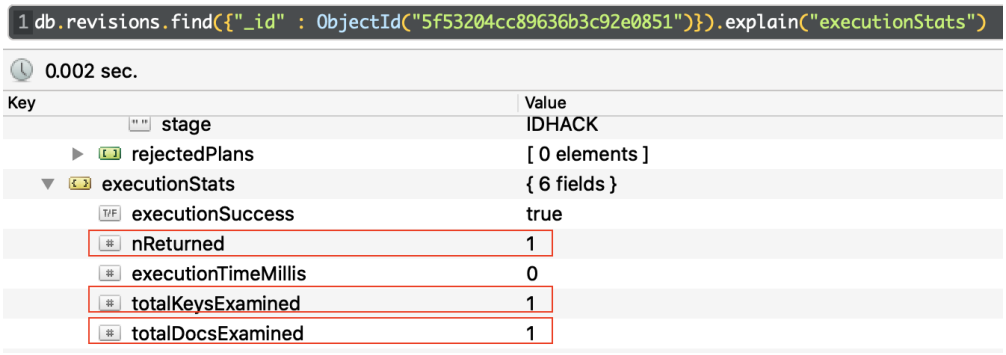

Figure 3: Search By _id Execution Status

b) Querying a single indexed field.

```
db.revisions.find(
    {user:"Muboshgu"},
    { title: 1}
).explain("executionStats")

db.revisionsWI.find(
    {user:"Muboshgu"},
    { title: 1}
).explain("executionStats");
```

The second pair searches for the revisions made by user “Muboshgu”. The `revisionsWI` collection has an index on field `user`. This index is used in the `winningPlan`. Since the query condition contains only the `user` field, which has a index on it, no alternative plan needs to be evaluated for this query (see figure 4). The `executionStats` shows that the `nReturned` is 5, and this is the result of examining 5 keys and 5 documents.

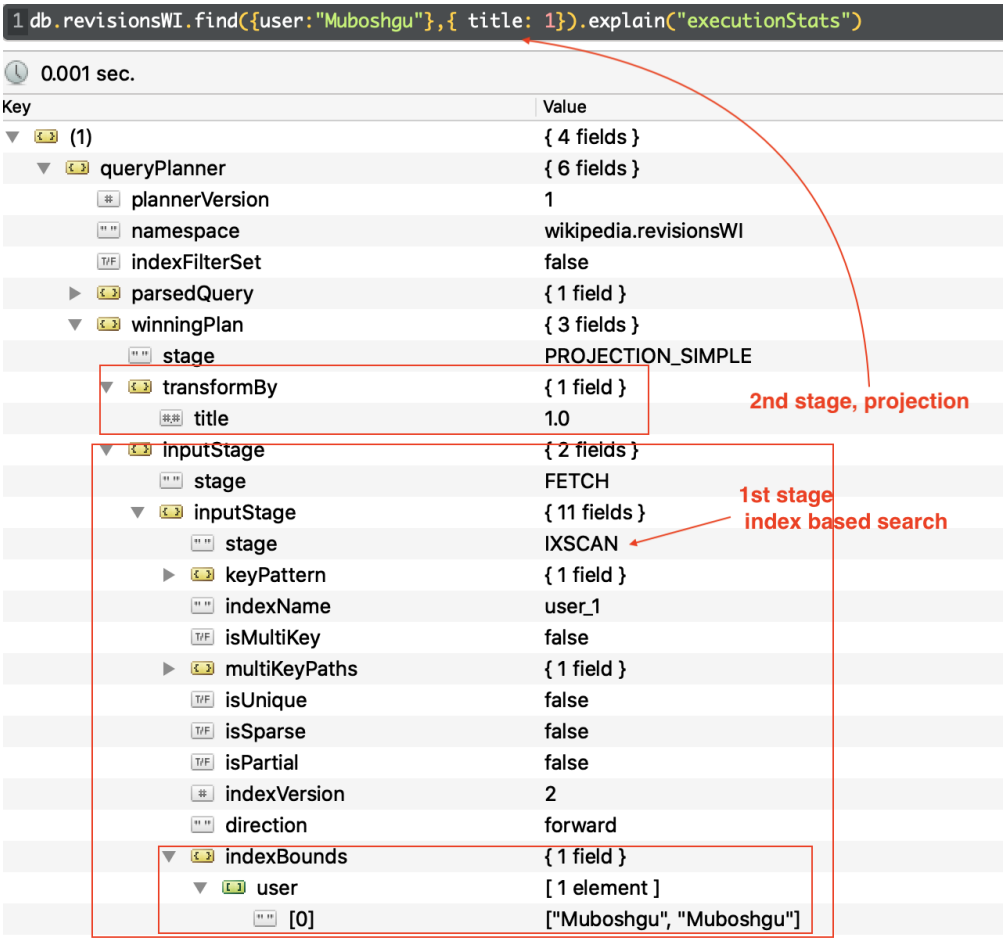

Figure 4: Single Field with Index Query Plan and Execution Statistics

In contrast, the same query running on `revisions` collection also returns 5 documents

but examined 623 documents. There is no key examined because it does not have an index on the query field (see figure 5).

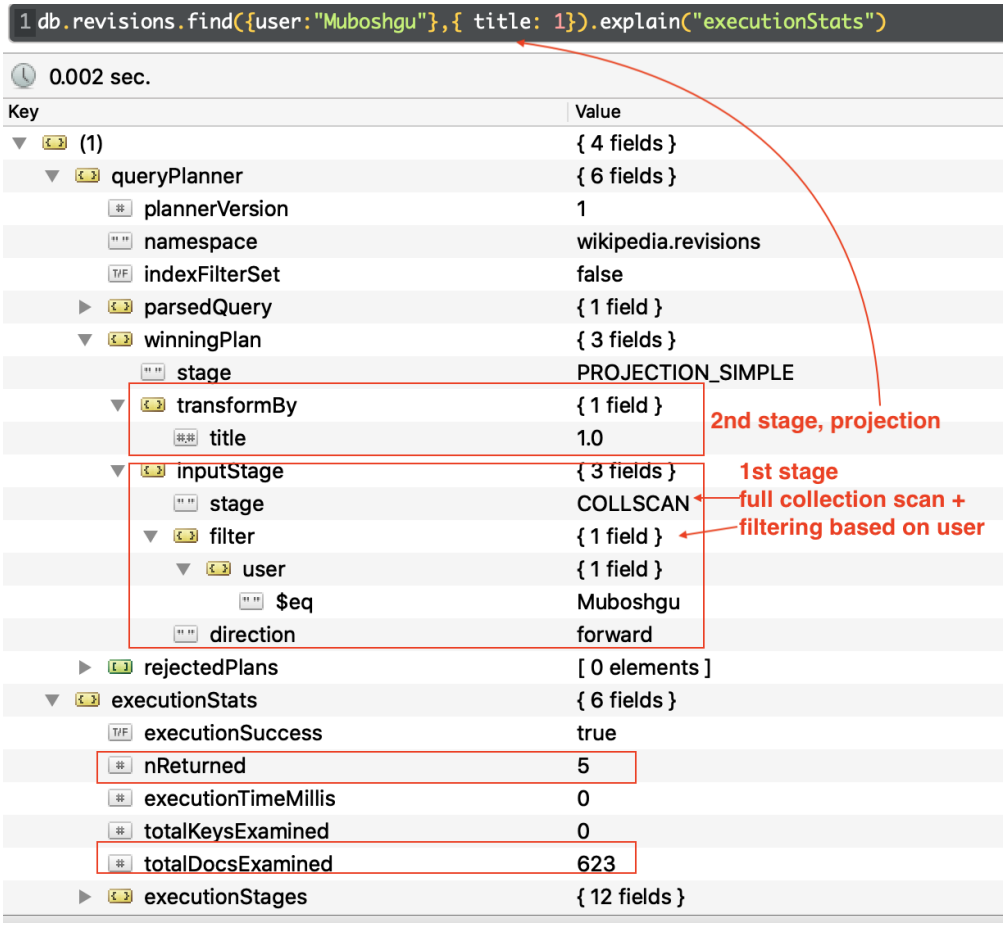

Figure 5: Single Field No Index Query Plan and Execution Statistics

c) Querying two indexed fields.

```
db.revisions.find(
    {
        "title": "Donald_Trump",
        "timestamp":{
            $gte: ISODate("2016-07-01"),
            $lte:ISODate("2016-07-02")
        }
    }
).explain("executionStats")

db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "timestamp":{
            $gte: ISODate("2016-07-01"),
            $lte:ISODate("2016-07-02")
        }
    }
).explain("executionStats")
```

The third pair searches for revisions made on Donald Trump page on a particular day ("2016-07-01"). The query condition involves two fields: `title` and `timestamp`, both have index set up on collection `revisionsWI`. Query executing on `revisionsWI` collection uses only **one index** to find an initial document set. Documents in the initial set are examined to extract those satisfying the condition on the **second field**. MongoDB evaluates the plans of using either index and picks one with better performance, e.g., returning a smaller sized initial document set.

Inspect the `explain` output of the `revisionsWI` collection, you will find there is a `winningPlan` as well as a `rejectedPlans` under `queryPlanner`. The `winningPlan` has a `inputStage` and `filter` stage. The `inputStage` applies index scan (`IXSCAN`) on the `indexName` "timestamp_1". The `filter` stage filtering documents based on `title` field (see figure 6).

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "timestamp":{$gte: ISODate("2016-07-01"), $lte:ISODate("2016-07-02")}
    }
).explain("executionStats")
```

0.022 sec.

Key	Value	
queryPlanner	{ 6 fields }	
plannerVersion	1	
namespace	wikipedia.revisionsWI	
indexFilterSet	false	
parsedQuery	{ 1 field }	
winningPlan	{ 3 fields }	One winning plan
stage	FETCH	
filter	{ 1 field }	2nd stge, filter by title
title	{ 1 field }	
$eq	Donald_Trump	
inputStage	{ 11 fields }	1st stage search timestamp index
stage	IXSCAN	
keyPattern	{ 1 field }	
indexName	timestamp_1	
isMultiKey	false	
multiKeyPaths	{ 1 field }	
isUnique	false	
isSparse	false	
isPartial	false	
indexVersion	2	
direction	forward	
indexBounds	{ 1 field }	
rejectedPlans	[1 element]	One rejected plan

Figure 6: Query Two Fields with Index: Winning Plan

The rejected plan has the stages reversed (see figure 7). The `executionStats` shows that with the winning plan, 7 documents are returned with 11 documents examined (see figure 8). If you are not sure if Mongodb has picked the best plan, you may force it to run with index on `title` field with the following query.

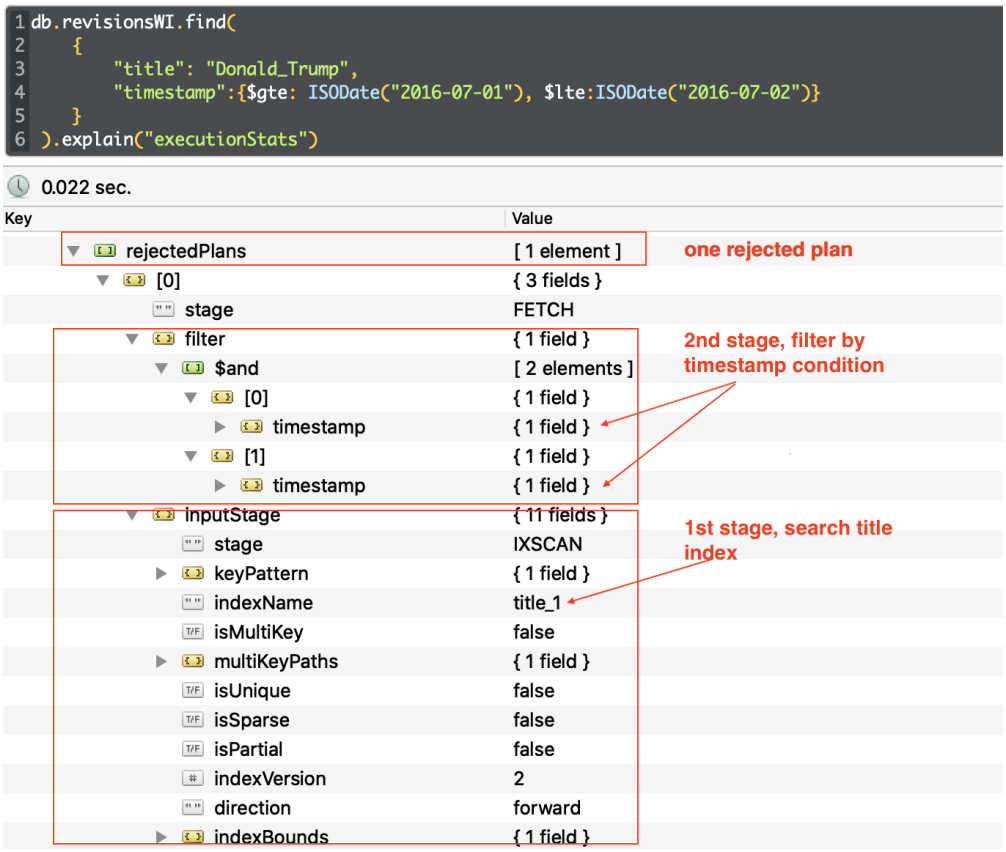

Figure 7: Query Two Fields with Index: Rejected Plan

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "timestamp":{
            $gte: ISODate("2016-07-01"),
            $lte:ISODate("2016-07-02")
        }
    }
).hint({title: 1}).explain("executionStats")
```

The `hint({title: 1})` method suggests Mongodb to answer the query with index on `title` field. The `executionStats` shows that with the `title` index, it needs to examine 434 documents to return the 7 documents as result. This is much worse than the planner picked up by MongoDB. MongoDB is able to compare performance of different indexes.

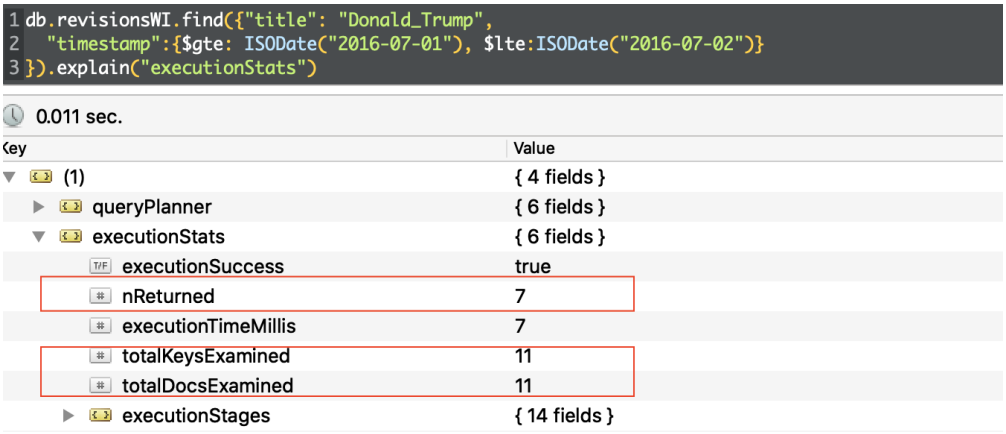

Figure 8: Query Two Fields with Index: Execution Statistics

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "timestamp":{
            $gte: ISODate("2016-07-01"),
            $lte:ISODate("2016-07-02")
        }
    }
).explain("allPlansExecution")
```

The query on collection `revisions` needs to examine all documents in the collection to find the results.

d) Query plan for aggregation pipeline Index has limited usage in aggregation pipeline. Some pipeline stage may use index under certain condition. Typically, the `$match` stage can use index if it is the first stage in the pipeline. There are also cases when `$group` and `$sort` stage may use index. In the following aggregation commands, the `$match` stage running on `revisionsWI` apply IXSCAN on index 'title' and the other two stages do not use index (see figure 9 and figure 10). A similar query on `revisions` collection does COLLSCAN during the `$match` stage (see figure 11).

```
db.revisionsWI.explain("executionStats").aggregate(
    [
        {$match:{title:"Donald_Trump"}},
        {$group:{_id:"$user", numOfEdits: {$sum:1}}},
        {$sort:{numOfEdits:-1}},
        {$limit:5}
    ]
)
```

and

```
db.revisions.explain("executionStats").aggregate(
    [
        {$match:{title:"Donald_Trump"}},
        {$group:{_id:"$user", numOfEdits: {$sum:1}}},
        {$sort:{numOfEdits:-1}},
        {$limit:5}
    ]
)
```

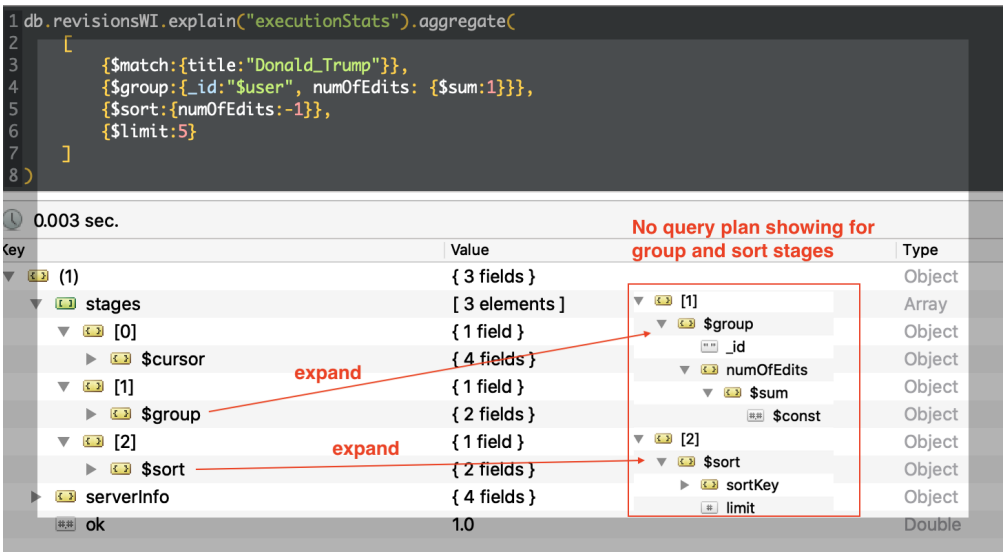

Figure 9: Aggregation High level Query Plan on Indexed Collection

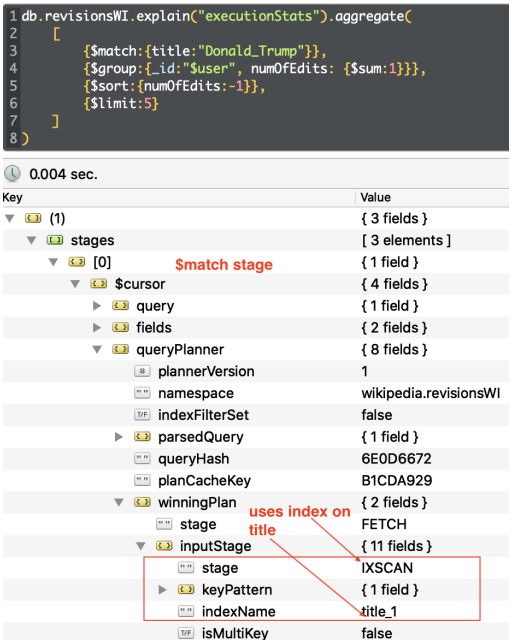

Figure 10: `$match` stage on indexed collection

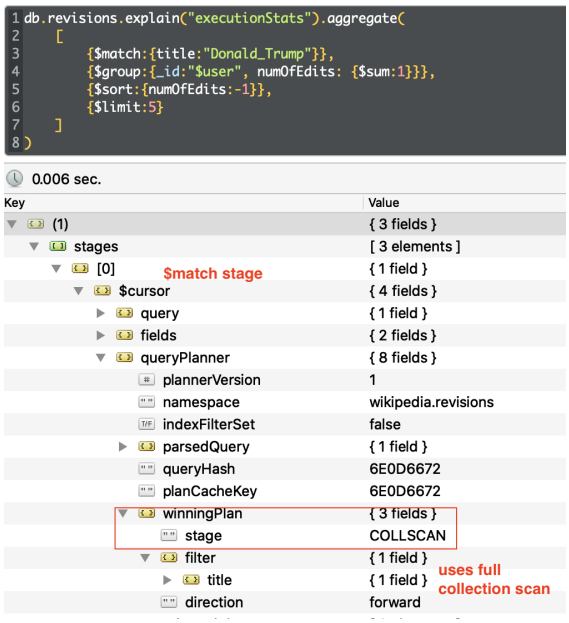

Figure 11: `$match` stage on non-indexed collection

Question 4: Compound Index and Sorting

Now create the following indexes on `users` collection:

```
db.users.createIndex({registration:1,editcount:1})
db.users.createIndex({gender:1})
```

Note that you need to change the 'registration' filed to ISODate type before creating the index.

Write your own queries to answer the following questions. In particular,use `explain()` to inspect how index is used in each query

1. Find out the female editors who made over 20000 edit. The number of edit is stored in field "editcount". Check if any index is used in this query? If yes, which one? How many query plans are evaluated?
2. Find out the number of female editors registered before "2007-01-01"'.
3. Find out the number of gender 'unknown' editors registered before "2007-01-01".
4. Compare the `explain` output of the last two queries. Do they adopt similar plan? If not, why?
5. Find out all users registered before "2007-01-01" and has made more than 30000 revisions, sort the result based on registration time. Inspect the query plan and

execution status. Does this query use any index? Do you think the sort stage can utilize the index as well? This query examined more index than documents, why?

Question 5: Multikey Index

Create an index on the array field “categories” of collection “pagecat”. Find all pages with “film” in its categories. Does this query use index, how is it used?

```
db.pagecat.find(
    {
        categories: {$regex: "film"}
    }
)
```

Compare the index usage in the following two queries. In particular check the key boundaries and the number of keys and documents examined. Are you able to describe how index is used to find the document satisfying the query condition

```
db.pagecat.find(
    {
        categories:
        {
            $all:
                [
                    "Category:Good articles",
                    "Category:American films"
                ]
        }
    }
).explain("executionStats")


db.pagecat.find(
    {
        categories:
        {
            $in:
           [
                "Category:Good articles",
                "Category:American films"
           ]
    }
 }
).explain("executionStats")
```