# COMP5338 – Advanced Data Models

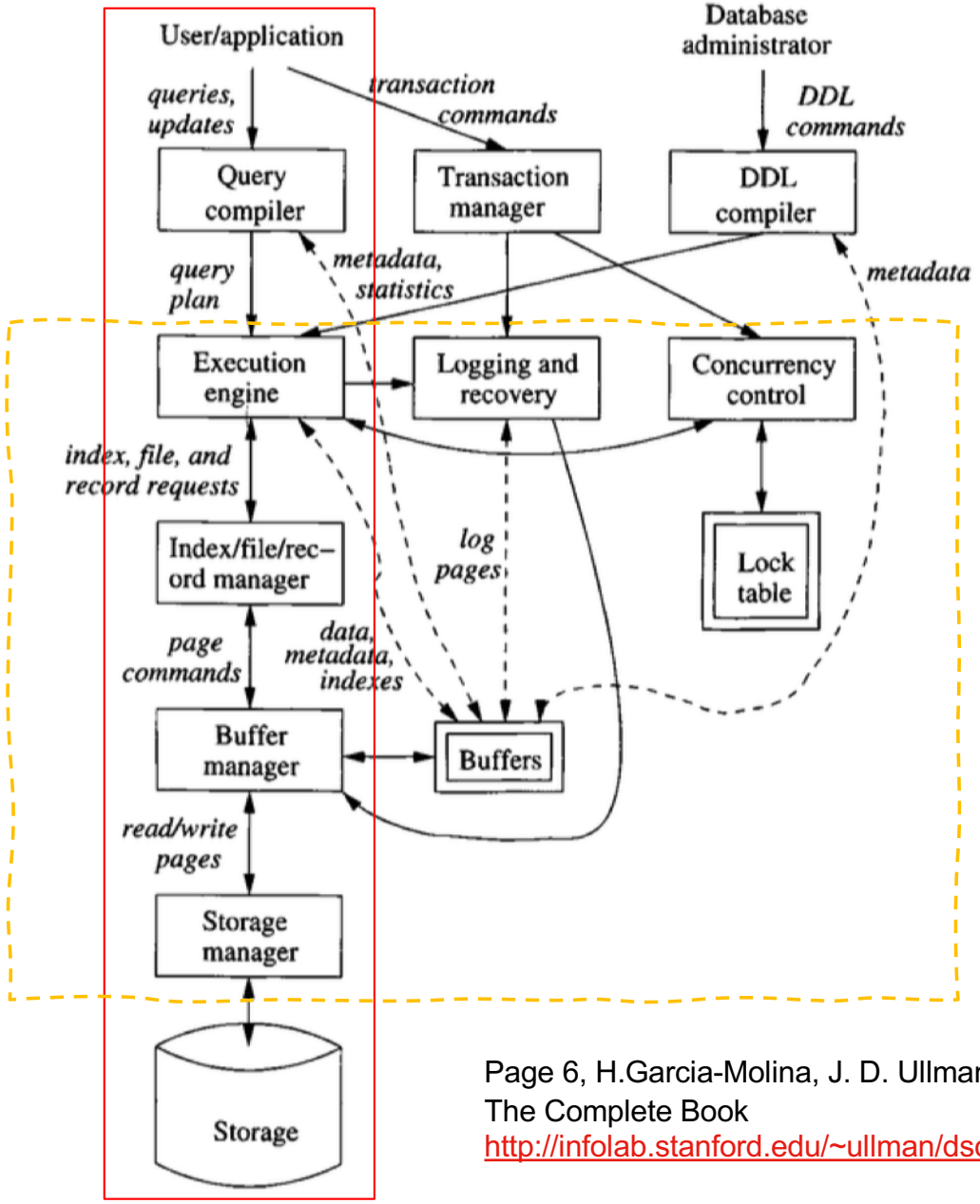
**Week 4:** MongoDB Indexing

Dr. Ying Zhou
School of Computer Science

# Outline

- Database Indexing
- **MongoDB Indexes**
- MongoDB Query Execution

# Review: DBMS Components




Disk based database system

Storage engine is responsible for managing how data is store in memory and disk

Page 6, H.Garcia-Molina, J. D. Ullman, J. Wildom, Databsae Systems The Complete Book
http://infolab.stanford.edu/~ullman/dscb.html

# The primitive operations of a query

- Read query
  - Load the element of interest from disk to main-memory buffer(s) if it is not already there
  - Read the content to client's address space
- Write query
  - The new value is created in the client's address space
  - It is copied to the appropriate buffers representing the database in the memory
  - The buffer content is flushed to the disk
- Both operations involve data movement between *disk* and *memory* and between *memory spaces*
- Typically *disk access* is the predominant performance cost in single node settings. Network communication contributes to the cost in cluster setting
- One design goal of database system is to reduce the amount of disk I/Os in read and write queries

# Typical Solutions to minimize Disk I/O

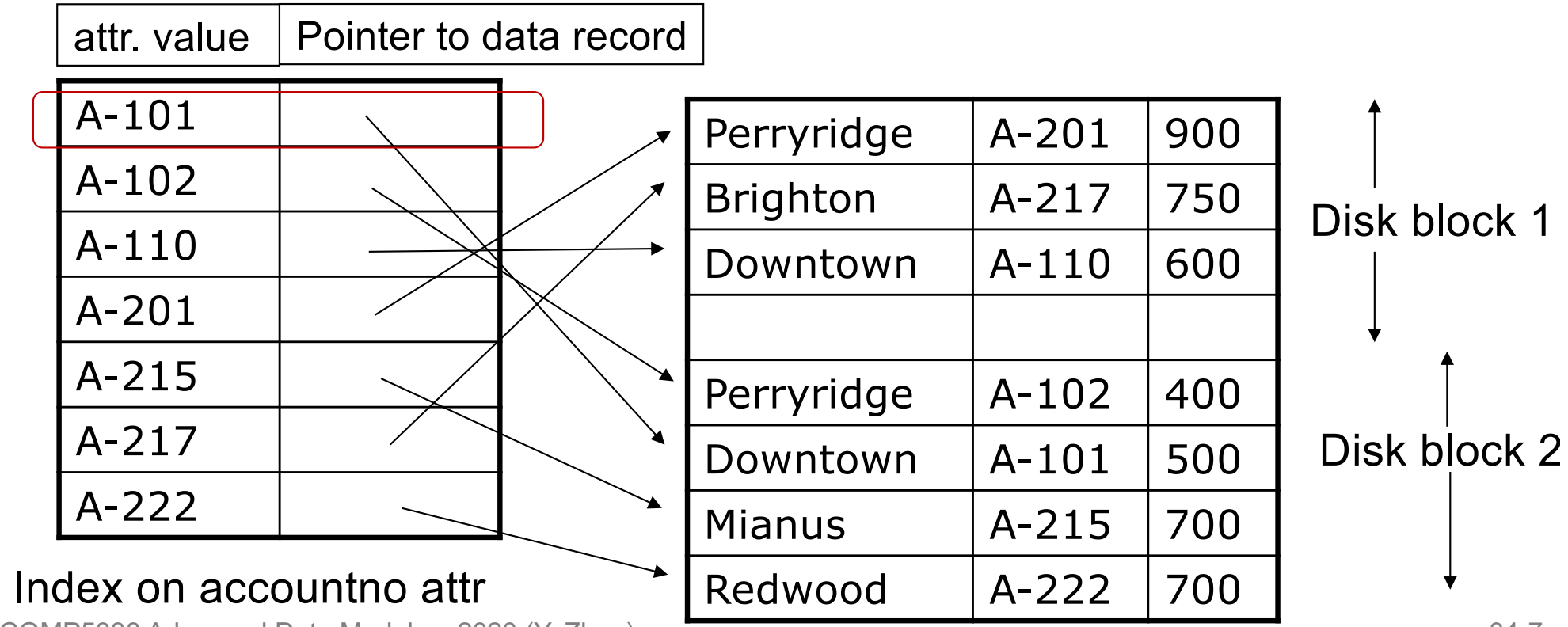
- Queries involve reading data from the database
  - Minimize the amount of data need to be moved from disk to memory
  - Use index and data distribution information to decide on a query plan
- Queries involve writing data to the database
  - Minimize the amount of disk I/O in the write path
    - Avoid flushing memory content to disk immediately after each write
    - Push non essential write out the of write path, e.g. do those asynchronously
  - To ensure durability, write ahead log/journal/operation log is always necessary
    - Appending to logs are much faster than updating the actual database file
    - The DB system may acknowledge once the data is updated in memory and appended in the WAL
    - Update to replicas can be done asynchronously, e.g. not in the write path

# Textbook architecture of storage engine

- Data is stored in disk blocks with row-based format
- B-Tree primary and secondary indexes
- ACID transaction support
- Row based locking
- MVCC (multi-version concurrency control)

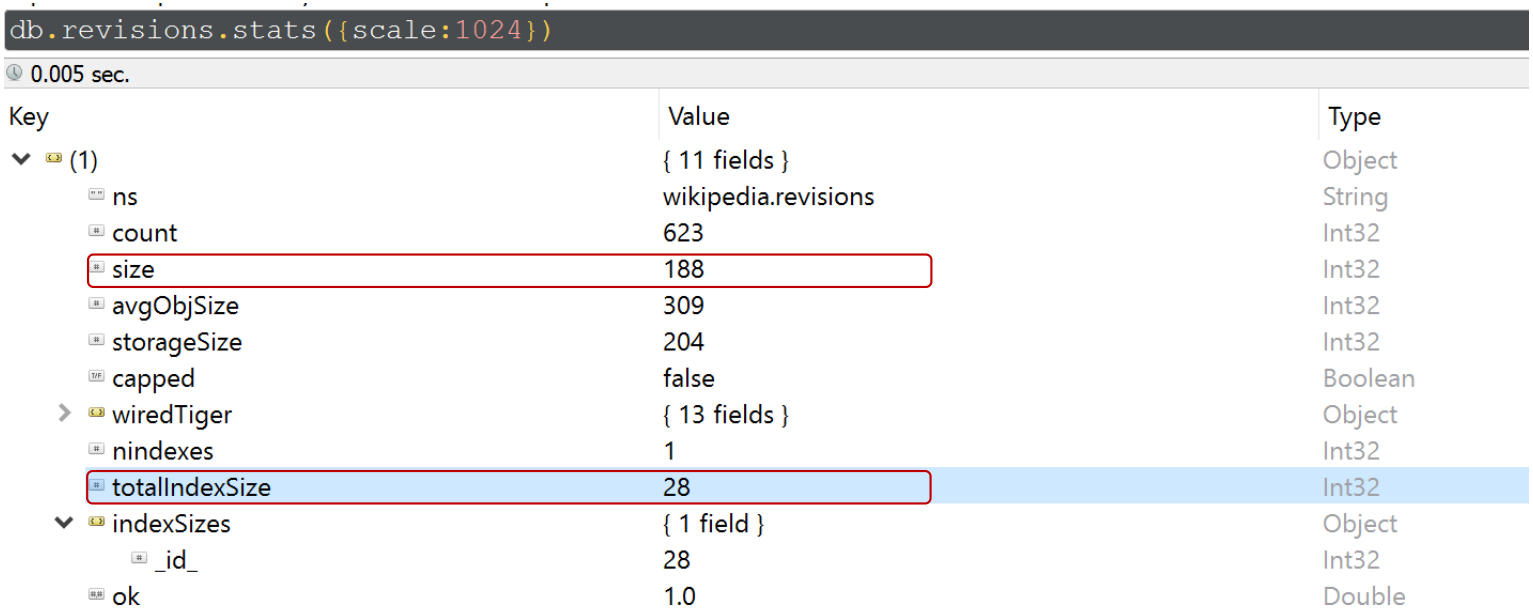
# Indexing

- An index on an attribute **A** of a table is a data structure that makes it efficient to find those rows(document) that have a required value for attribute/field **A**.
- An index consists of records (called index entries) each of which has a value for the attribute(s) of the form



# Indexing (con’td)

- Index entries are sorted by the attribute (search key) value
- Index files are typically much smaller than the original file



# Outline

- Database Indexing
- **MongoDB Indexes**
- MongoDB Query Execution

# MongoDB Storage Engine

- MongoDB supports multiple storage engines
  - **WiredTiger** is the default one since version 3.2
- Some prominent features of WiredTiger
  - Provide both B-tree and Log Structured Merge tree index
  - Document level concurrency
  - Multi Version Concurrency Control (MVCC)
    - Snapshots are provided at the start of operation using timestamp
    - Snapshots are written to disk (creating checkpoints) at intervals of 60 seconds (or 2GB of journal data)
  - Journal
    - Write-ahead transaction log
  - Compression

# MongoDB Basic Indexes

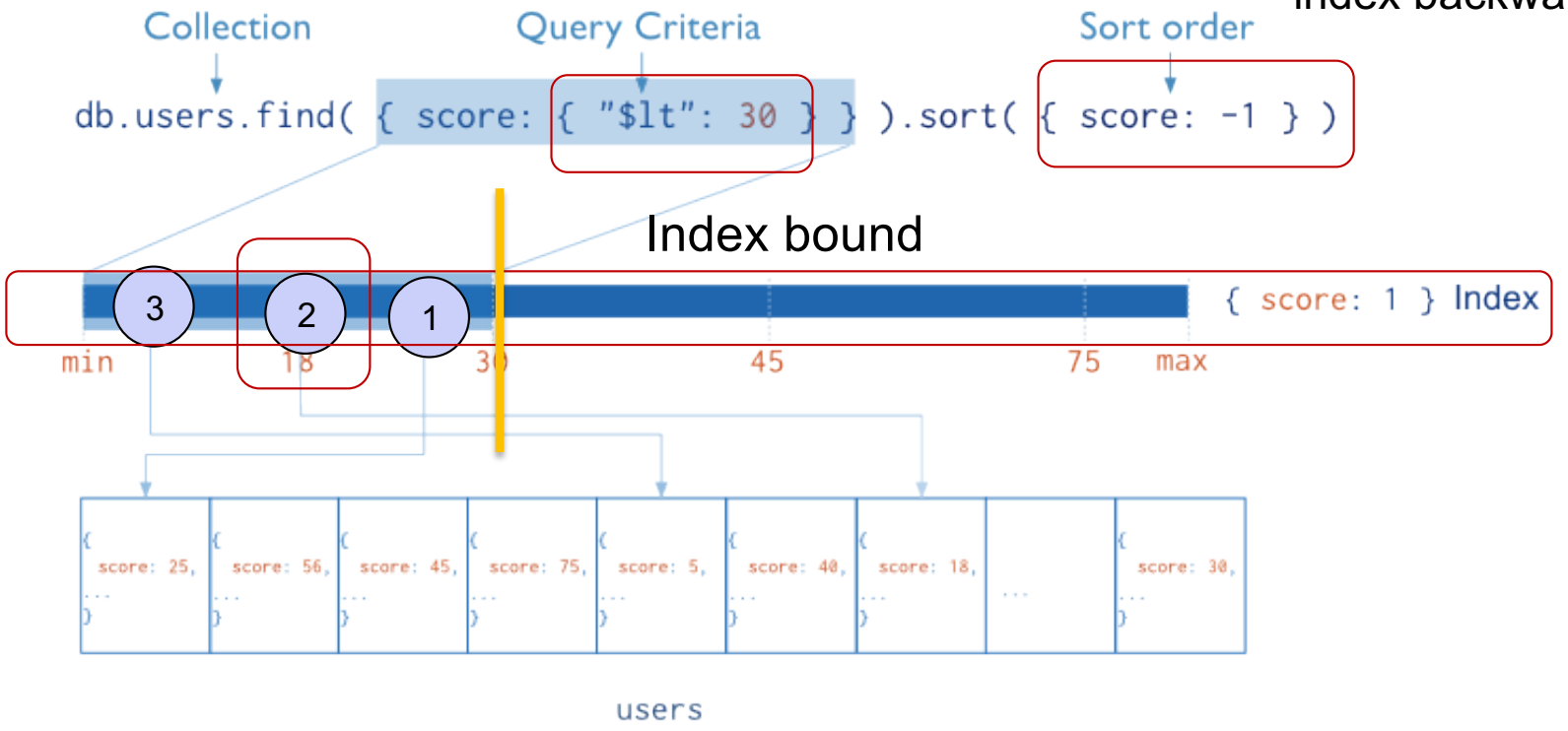
- The **`_id`** index
  - **`_id`** field is automatically indexed for all collections
  - The **`_id`** index enforces uniqueness for its keys
- Indexing on other fields
  - Index can be created on any other field or combination of fields
    - `db.`***`<collectionName>`***`.createIndex({<`***`fieldName`***`>:1});`
    - **`fieldName`** can be a simple field, array field or field of an embedded document (using dot notation)
      - db.blog.createIndex({author:1})
      - db.blog.createIndex({tags:1})
      - db.blog.createIndex({"comments.author":1})
    - the number specifies the direction of the index (1: ascending; -1: descending)
  - Additional properties can be specified for an index
    - **Sparseness**, **uniqueness**, **background**, ..
- Most MongoDB indexes are organized as **`B-Tree`** structure by default

http://www.mongodb.org/display/DOCS/Indexes

# Single field Index

```
db.users.createIndex({ score: 1 })
```

Descending order means traversing the index backwards

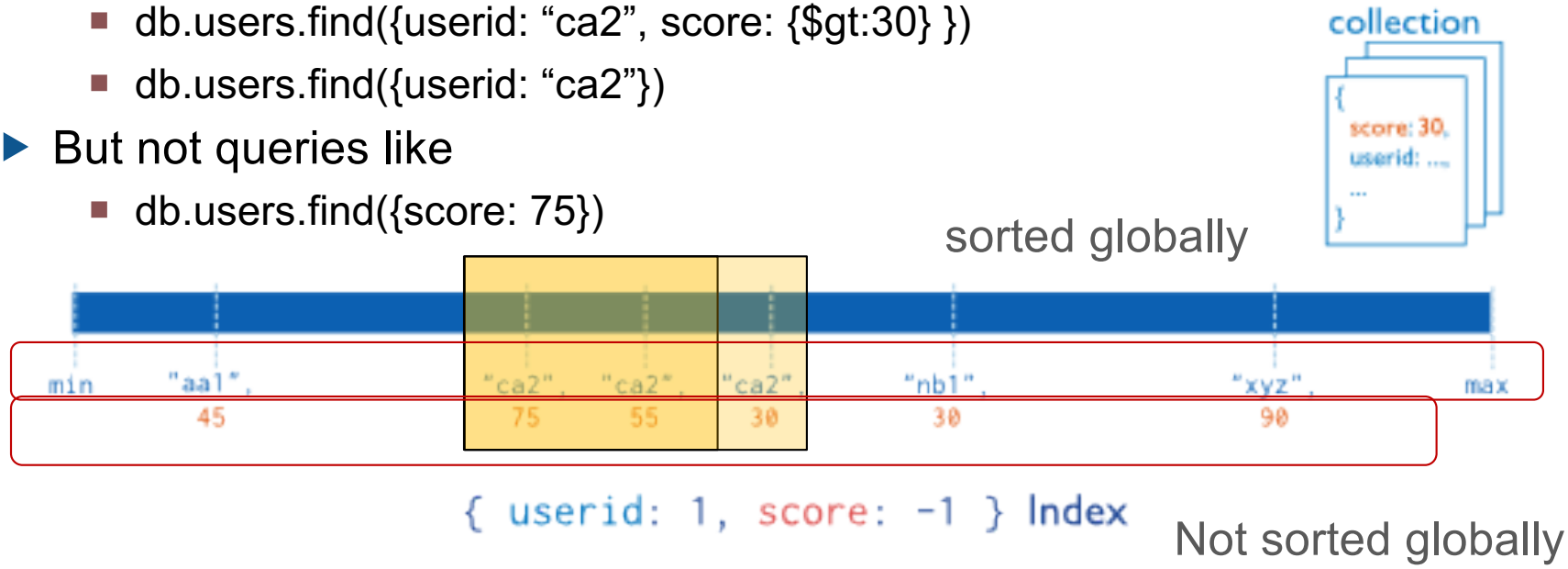



Single entry with filed value and pointer to document

https://docs.mongodb.com/manual/core/index-single/

# Compound Index

- Compound Index is a single index structure that holds references to multiple fields within a collection
- The order of field in a compound index is very important
  - The index entries are sorted by the value of the first field, then second, third…
  - If we have a compound index: `{userid:1, score:-1}`
  - It supports queries like
    - db.users.find({userid: “ca2”, score: {$gt:30} })
    - db.users.find({userid: “ca2”})
  - But not queries like
    - db.users.find({score: 75})




https://docs.mongodb.com/manual/core/index-compound/

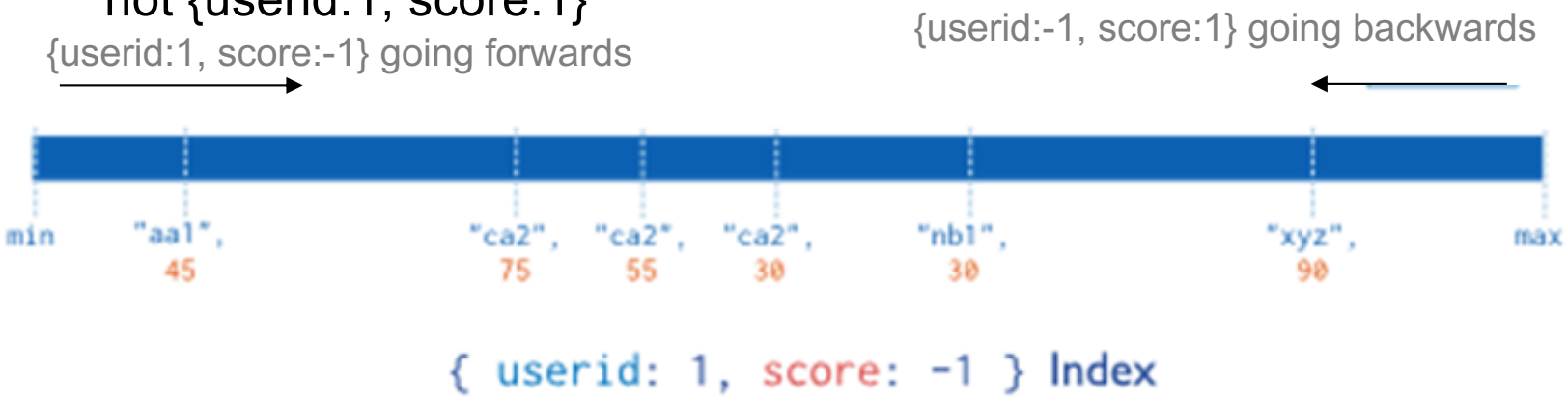
# Use Index to Sort (single field)

- Sort operation may *obtain the order from index* or *sort the result in memory*
- Index can be traversed in either direction
- Sort with a single field index
  - For single field index, sorting by that field can always use the index regardless of the sort direction
  - E.g. `db.records.createIndex( { a: 1 } )` supports both
    - `db.records.find().sort({a:1})` and
    - `db.records.find().sort({a: -1})`

https://docs.mongodb.com/manual/tutorial/sort-results-with-indexes/

# Use Index to Sort (multiple fields)

- Sort on multiple fields
  - Compound index may be used on sorting multiple fields.
  - There are constrains on fields and direction
    - Sort key should have the same order as they appear in the index
    - All field sort have same sort direction, either going forwards or backwards the index
    - E.g. {userid:1, score:-1} and {userid:-1, score:1} can use the index, but not {userid:1, score:1}



# Use Index to Sort (multiple fields)

- Sort and Index Prefix
  - If the sort keys correspond to the index keys or an index *prefix*, MongoDB can use the index to sort the query results.
    - E.g. db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )
    - Supported query:
    - db.data.find().sort({ a: -1 })
    - db.data.find().sort({ a: 1, b: 1 } )
    - db.data.find({a:{ $gt: 4}}).sort( { a: 1, b: 1 } )

Results can be obtained using the same index

- Sort and Non-prefix Subset of an Index
  - An index can support sort operations on a non-prefix subset of the index key pattern if the query include **equality** conditions on all the prefix keys that precede the sort keys.
    - db.data.find( { a: 5 } ).sort( { b: 1, c: 1 } )
    - db.data.find( { a: 5, b: { $lt: 3} } ).sort( { b: 1 } )

Sort keys

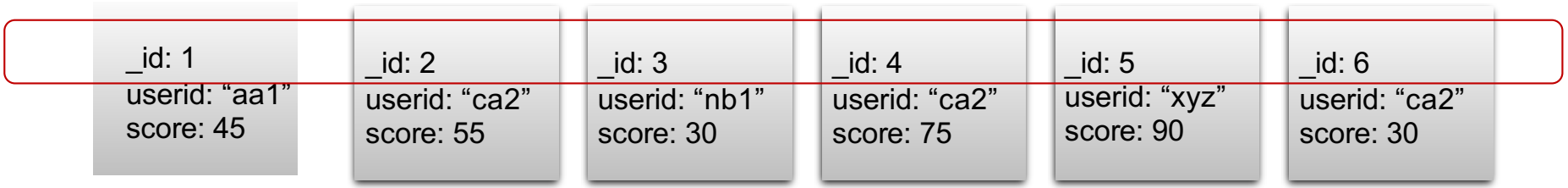
Sort key

Equality condition on a

Equality condition on a

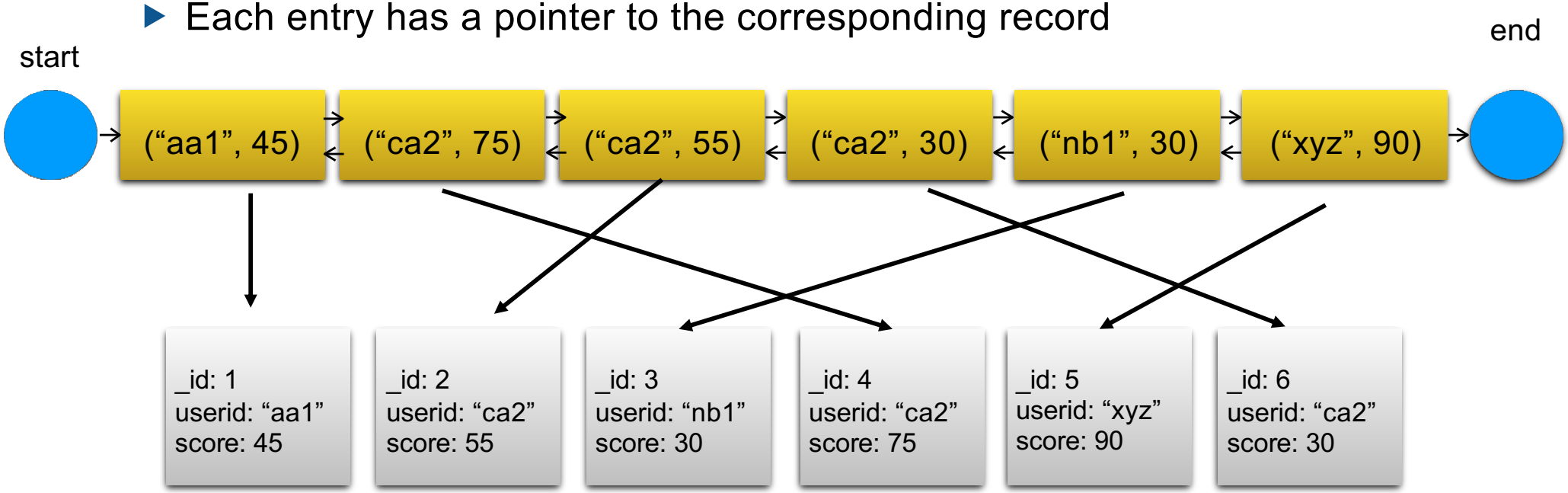
# Running Example

- Suppose we have a **users** collection with the following 6 documents stored in the order of `_id` values



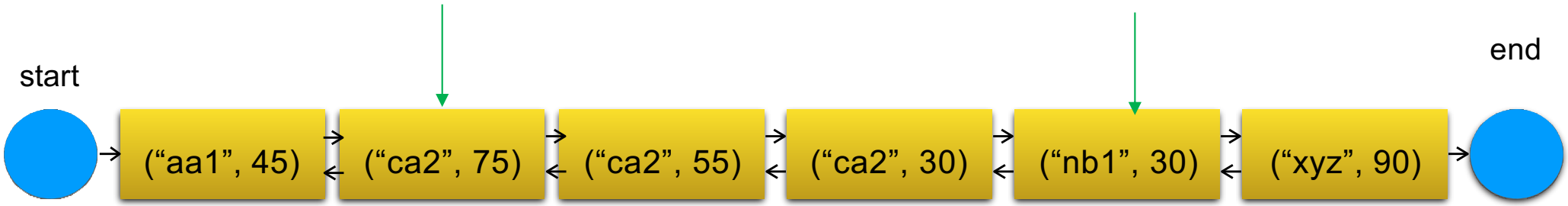
# Index Entries

- Now we create a compound index on **userid** and **score** fields : **`db.users.createIndex(userid:1, score:-1)`**
- With the current data, the index has six entries because we have 6 records in the collection
  - The entries are sorted in descending `(userid,score)` value
  - the index entries usually form a doubly linked list to facilitate bi-directional traversal
  - Each entry has a pointer to the corresponding record



# Using index to find documents

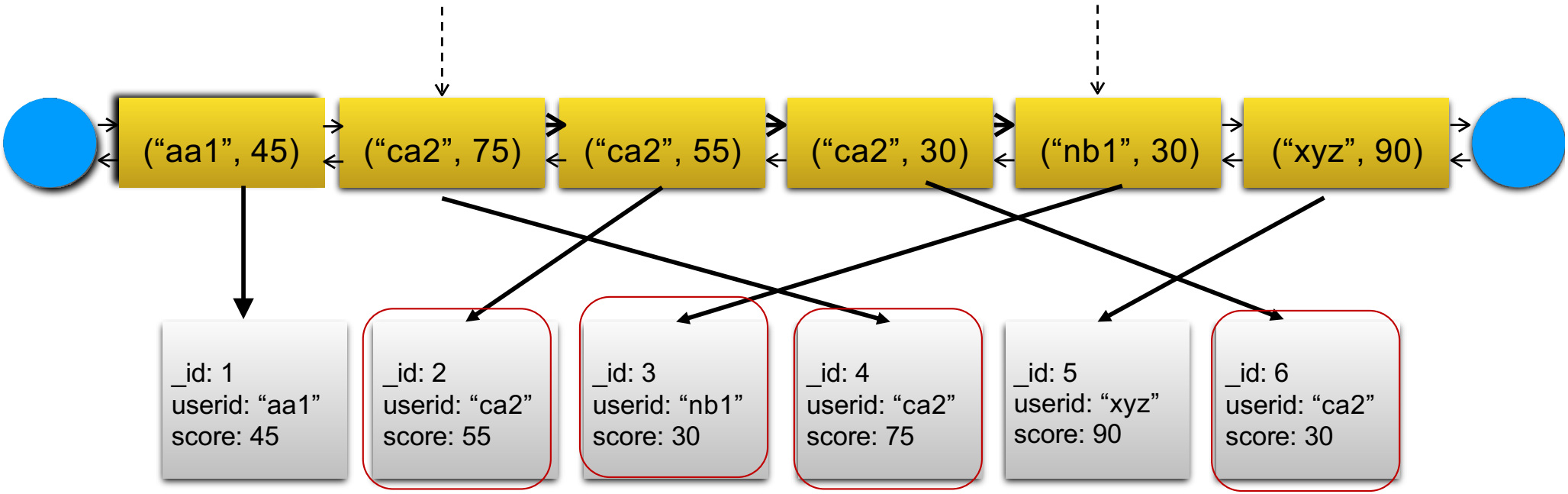
- For queries that are able to use index, the first step is to find the boundary entries on the list based on given query condition
- Example query
  - `db.users.find({userid:{$gt: “b”, $lt:“s”}})`
- This query is able to use the compound index and the two bounds are:(“ca1”, 75) and (“nb1”,30) inclusive at both ends



# Using index to find documents

- The four documents with **`_id`** equals: 4, 2, 6 and 3 are the result of the above query

```
db.users.find({userid:{$gt: “b”, $lt:“s”}})
```

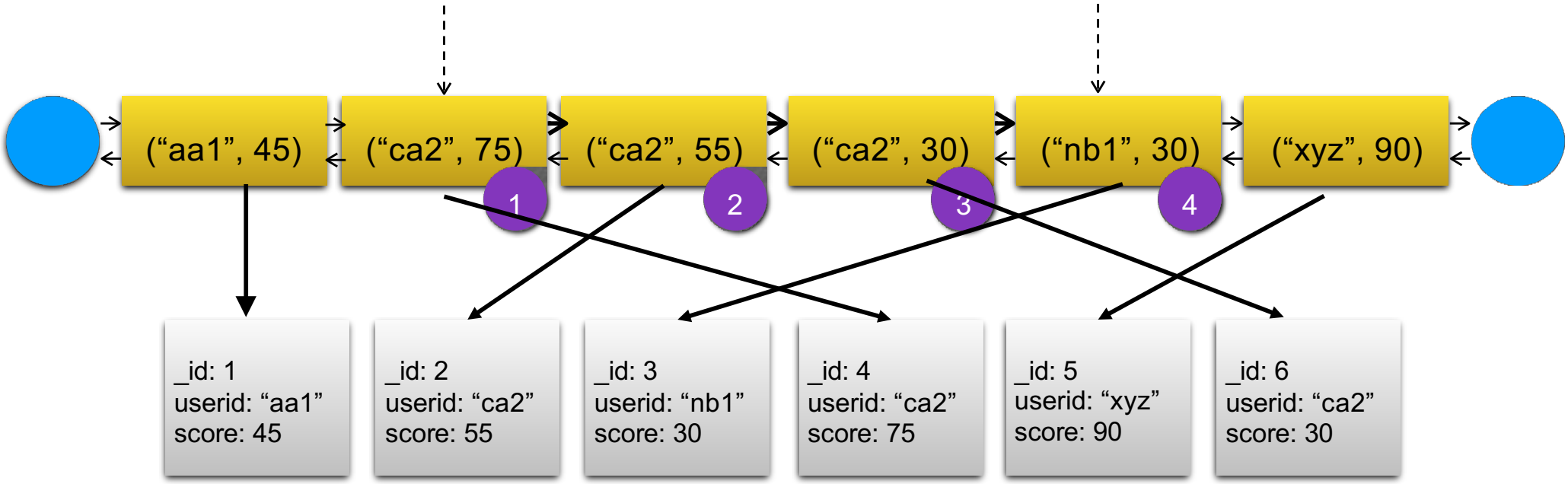


# Using Index to sort

- If our queries include a sorting criteria

```
db.users.find(
    {userid:{$gt: “b”, $lt:“s”}}
).sort({userid:1, score:-1})
```

- Theengine will start from the **lower** bound, following the **forward links** to the **upper** bound and return all documents pointed by the entries



# Sorting that cannot use index

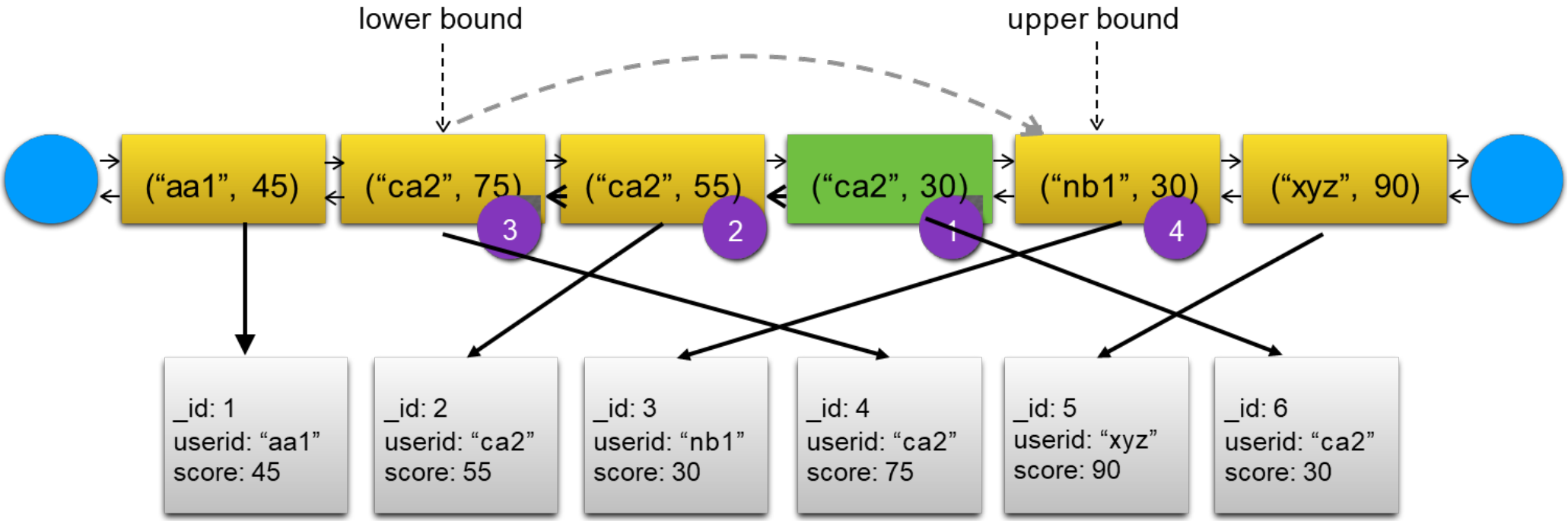
- If our query includes yet another sorting criteria

```
db.users.find(
    {userid:{$gt: “b”, $lt:“s”}}
).sort({userid:1, score:1})
```

- We can still use the index to find the bounds and the four documents satisfying the query condition, but we are not able to follow a single forward or backward link to get the correct order of the documents

# Sorting that cannot use index

- If we want to use the index entry list to obtain the correct, we would start from a mysterious position (“ca2”,30), follow the backward links to (“ca2”,75), and make a magic jump to the entry (“nb1”, 30).
  - complexity involved:
    - how do we find the start point in between lower and upper bound?
    - how do we decide when and where to jump in another direction?
  - The complexity of such algorithm makes it less optimal than a memory sort of the actual documents.

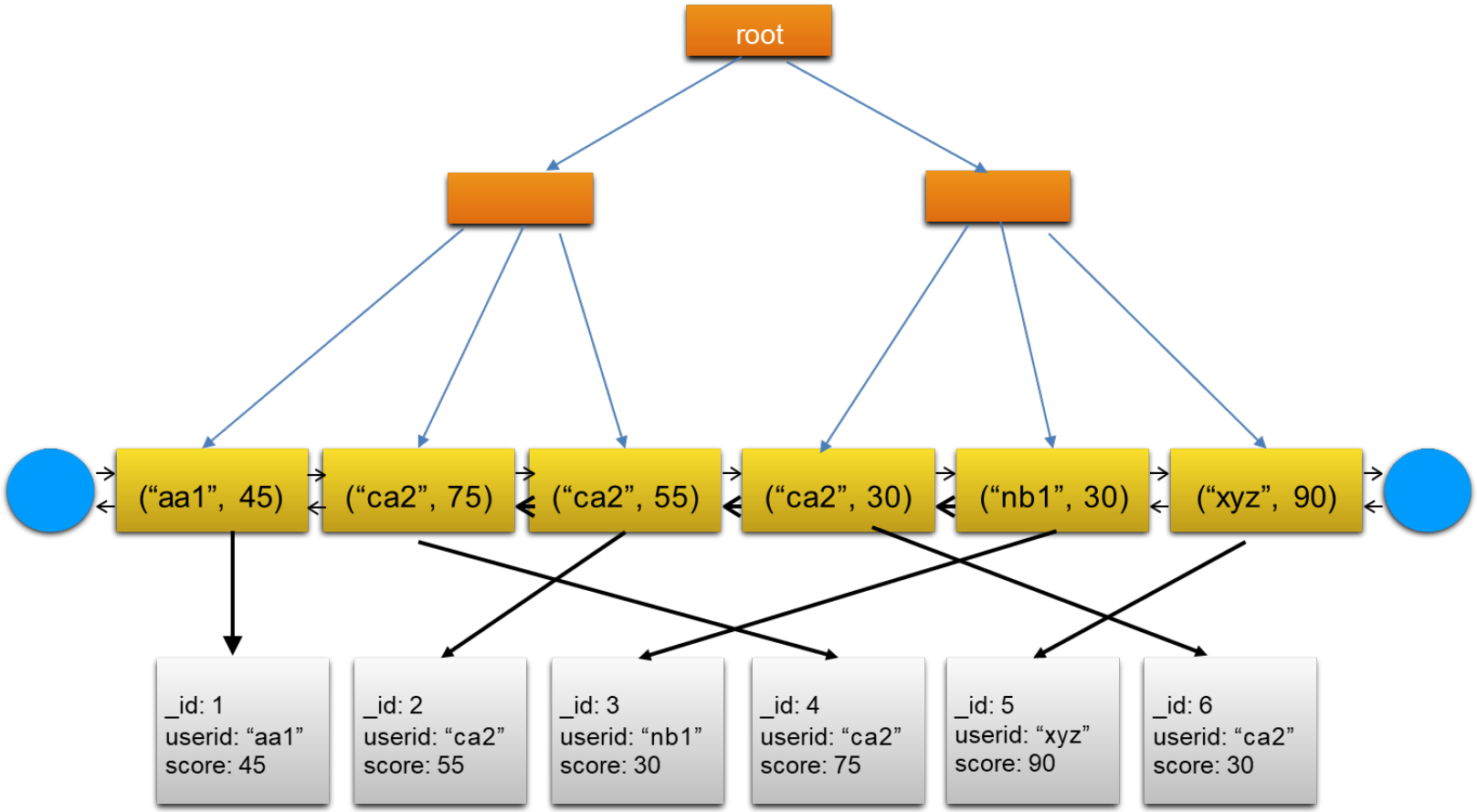


# General rules

- If you are able to traverse the list between the upper and lower bounds as determined by your query condition in one direction to obtain the correct order as specified in the sort condition, the index will be used to sort the result
- Otherwise you may still use index to obtain the results but have to sort them in memory

# BTree motivation

- Finding the boundaries could be time consuming if we only have the list structure and can only start from one of the two ends
- B-Tree structure is built on top of the index values to accelerate the process of locating the boundary.



# Multi key index

- Index can be created on array field, the key set includes each element in the array. It behaves the same as single index field otherwise
- There are restrictions on including multi key index in compound index



{ "addr.zip": 1 } Index

# Text Indexes

- Text indexes support efficient text search of string content in documents of a collection
- To create a text index
  - `db.<collectionName>.createIndex({<fieldName>:"text"});`
  - text index tokenizes and stems the terms in the indexed fields for the index entries.
- To perform text query
  - `db.find($text:{$search:<search string>}})`
    - No field name is specified
- Restrictions:
  - A collection can have at most one text index, but it can include text from multiple fields
  - Different field can have different weights in the index, results can be sorted using text score based on weights
  - Sort operations cannot obtain sort order from a text index

# Other Indexes

- Geospatial Index
  - MongoDB can store and query spatial data in a flat or spherical surface
    - 2d indexes and 2dsphere indexes
- Hash indexes
  - Index the hash value of a field
  - Only support equality match, but not range query
  - Mainly used in hash based sharding

# Indexing properties

- Similar to index in RDBMS, extra properties can be specified for index
- We can enforce the *uniqueness* of a field by create a unique indexes
  - `db.members.createIndex( { "user_id": 1 }, { unique: true } )`
- We can reduce the index storage by specifying index as *sparse*
  - Only documents with the indexed field will have entries in the index
  - By default, non-sparse index contain entries for all documents. Documents without the indexed field will be considered as having **`null`** value.
- MongoDB also supports TTL indexes and partial index
  - Not all documents will be indexed, based on time or based on given condition

# Indexing strategy

- Indexing cost
  - Storage, memory, write latency
- Performance consideration
  - In general, MongoDB only uses one index to fulfil specific queries
    - `$or` query on different fields may use different indexes
    - MongoDB may use intersection of multiple indexes
  - When index fits in memory, you get the most performance gain
- Build index if the performance gain can justify the cost
  - Understand the query
  - Understand the index behaviour

# Outline

- Database Indexing
- MongoDB Indexes
- **MongoDB Query Execution**

# Performance Monitoring Tools

- Profiler
  - Collects execution information about *queries* running on a database
  - It can be used to identify various underperforming queries
    - Slowest queries
    - Queries not using any index
    - Queries running slower than some threshold
    - Custom tagged queries, e.g. by commenting
    - And more
- Explain method
  - Collect detailed information about *a particular query*
    - How the query is executed
    - What execution plans are evaluated
    - Detailed execution statistics, e.g. how many index entries or documents have been examined

https://studio3t.com/knowledge-base/articles/mongodb-query-performance/

# MongoDB Query Execution

- MongoDB supports querying any field in a collection
  - Including non-existent field
- When index exists on a query field
  - It uses the index to find intermediate or final results
- Otherwise
  - It performs a full collection scan and exams every document to find the results
- When multiple indexes can be used for a query
  - The query optimizer evaluates different plans and determine the best one
    - Usually the one with high selectivity that can narrow the results most using index
- The `explain` method output many information about a particular query.

# Using the `explain` method

- The method can be added on both **`find`** and **`aggregate`** command
- Explain find command:
  - **db.collection.find(…).explain(…)** or
  - db.collection.explain(…).find(…)
- Explain aggregation command:
  - db.collection.aggregate(…).explain(…)
  - **db.collection.explain(…).aggregate(…)**
- The Robo 3T shell only supports **`explain`** *after* **`find`** and **`explain`** *before* **`aggregate`**
- Other shell (e.g. VS + Mongo Ext) supports all four options.

# Explain Verbosity Modes

- Showing only the query plan
  - Use it without any parameter
- Showing also the execution statistics of the chosen plan
  - **`explain("executionStats")`**
- Showing execution statistics of all candidate plans
  - **`explain("allPlansExecution" )`**

```
1 db.revisions.find({"_id" : ObjectId("5f53204cc89636b3c92e0851")}).explain("executionStats")
```

0.002 sec.

| Key | Value | Type |
|---|---|---|
| ▼ (1) | { 4 fields } | Object |
| ▶ queryPlanner | { 6 fields } | Object |
| ▶ executionStats | { 6 fields } | Object |
| ▶ serverInfo | { 4 fields } | Object |
| ok | 1.0 | Double |

# No Index Query Plan

```
1 db.revisions.find({random: {$exists: true}}).explain("executionStats")
```

0.009 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisions |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 3 fields } |
| stage | COLLSCAN |
| filter | { 1 field } |
| random | { 1 field } |
| $exists | true |
| direction | forward |
| rejectedPlans | [ 0 elements ] |
| executionStats | { 6 fields } |
| executionSuccess | true |
| nReturned | 0 |
| executionTimeMillis | 5 |
| totalKeysExamined | 0 |
| totalDocsExamined | 623 |
| executionStages | { 13 fields } |
| serverInfo | { 4 fields } |
| ok | 1.0 |

There is only one candidate plan: whole collection scan!

And every document will be checked to see if a field called **`random`** exists

The query searches for documents with a field "random", The current collection does not have any document with that field

# Two Collections with same data

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

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

# Query with one indexed field

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

0.001 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisionsWI |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 3 fields } |
| stage | PROJECTION_SIMPLE |
| transformBy | { 1 field } |
| title | 1.0 |
| inputStage | { 2 fields } |
| stage | FETCH |
| inputStage | { 11 fields } |
| stage | IXSCAN |
| keyPattern | { 1 field } |
| indexName | user_1 |
| isMultiKey | false |
| multiKeyPaths | { 1 field } |
| isUnique | false |
| isSparse | false |
| isPartial | false |
| indexVersion | 2 |
| direction | forward |
| indexBounds | { 1 field } |
| user | [ 1 element ] |
| [0] | ["Muboshgu", "Muboshgu"] |

2nd stage, projection

1st stage
index based search

| | |
|---|---|
| executionStats | { 6 fields } |
| executionSuccess | true |
| nReturned | 5 |
| executionTimeMillis | 0 |
| totalKeysExamined | 5 |
| totalDocsExamined | 5 |

# Multiple Query Plans

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "user":"ThiefOfBagdad"
    }
).explain("executionStats")
```

The query document contains two condition on two fields

There are three possible plans to execute the query

We can find all revision documents for “Donald Trump” page using the **`title`** index; then check if the revision is maded by “ThiefOfBagdad”

We can find all revision documents made by “ThiefOfBagdad” using the **`user`** index; then check if the `title` is “Donald Trump”

We can find revision documents made by “ThiefOfBagdad” using the **`user`** index; then find revision documents for “Donald Trump” page using the **`title`** index. The intersection of these two sets is the query result.

# Query Plans by MongoDB

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "user":"ThiefOfBagdad"
    }
).explain("executionStats")
```

0.001 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisionsWI |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 3 fields } |
| rejectedPlans | [ 2 elements ] |

# Winning Plan

```
db.revisionsWI.find(
    {
        "title": "Donald_Trump",
        "user":"ThiefOfBagdad"
    }
).explain("executionStats")
```

0.001 sec.

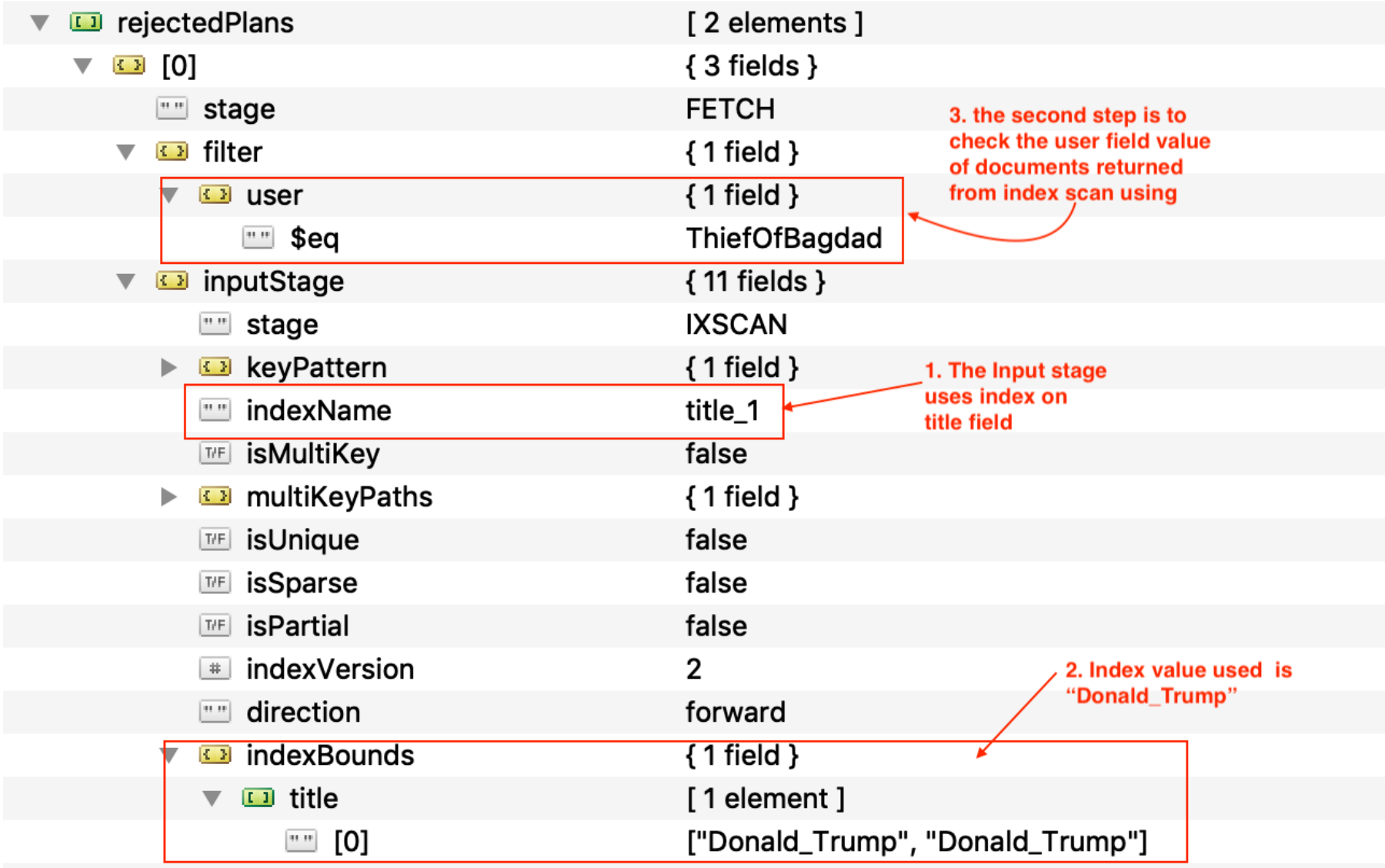
| Key | Value |
|---|---|
| winningPlan | { 3 fields } |
| stage | FETCH |
| filter | { 1 field } |
| title | { 1 field } |
| inputStage | { 11 fields } |
| stage | IXSCAN |
| keyPattern | { 1 field } |
| indexName | user_1 |
| isMultiKey | false |
| multiKeyPaths | { 1 field } |
| isUnique | false |
| isSparse | false |
| isPartial | false |
| indexVersion | 2 |
| direction | forward |
| indexBounds | { 1 field } |
| user | [ 1 element ] |
| [0] | ["ThiefOfBagdad", "ThiefOfBagdad"] |

Then filter by **`title`**

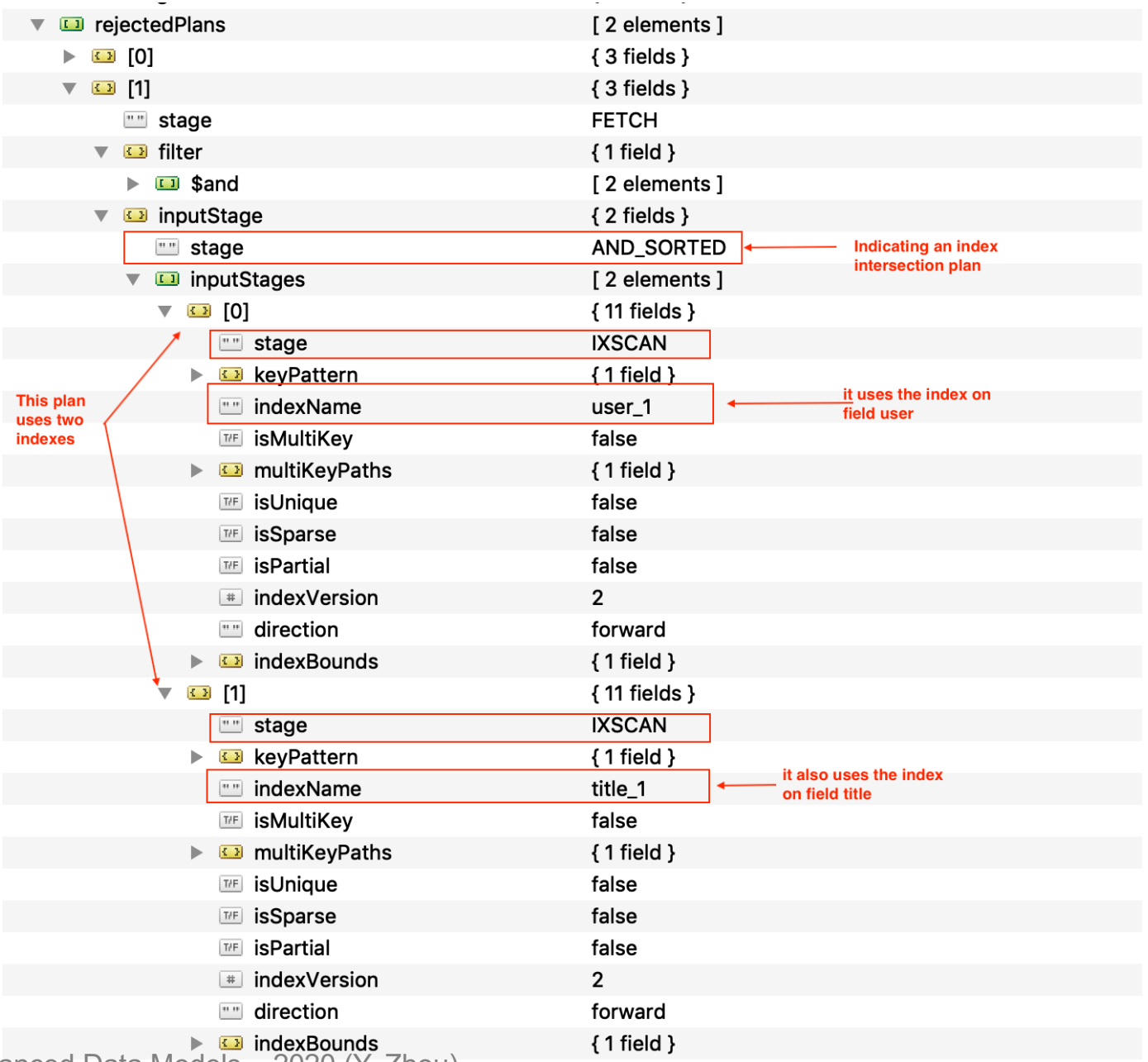
Search **`user`** index

Index bound is the given **`user`** value

# First Rejected Plan



# Second Rejected Plan



# Candidate Plan Selection

- Theoretically it depends on selectivity of plan
  - Using `user` index can narrow the result to 68 documents, while using `title` index can only narrow the result to 434 documents
- How does the database build such knowledge
  - Using various statistics to calculate cost (most RDBMS)
  - Run each plan partially to estimate cost and cache the plan for future use (MongDB's current approach)

# Check index usage on sort

- Index usage on SORT is not *explicitly* included in `explain()` result and the indication of sort usage in the `explain()` result varies in different MongoDB version
- In the current version, the inclusion of a stage called SORT means we cannot obtain sort order from index and sort need to be handled separately

# Sort not supported by index

```
1 db.revisionsWI.find(
2     {
3         "user":"ThiefOfBagdad"
4     }
5 ).sort({"timestamp":1}).explain("executionStats")
```

0.001 sec.

| Key | Value |
|---|---|
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisionsWI |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 3 fields } |
| stage | SORT |
| sortPattern | { 1 field } |
| timestamp | 1.0 |
| inputStage | { 2 fields } |
| stage | SORT_KEY_GENERATOR |
| inputStage | { 2 fields } |
| stage | FETCH |
| inputStage | { 11 fields } |
| stage | IXSCAN |
| keyPattern | { 1 field } |
| indexName | user_1 |
| isMultiKey | false |
| multiKeyPaths | { 1 field } |
| isUnique | false |
| isSparse | false |
| isPartial | false |
| indexVersion | 2 |
| direction | forward |
| indexBounds | { 1 field } |
| user | [ 1 element ] |
| [0] | ["ThiefOfBagdad", "ThiefOfBagdad"] |

Search field is ‘`user`’, sort field is ‘`timestamp`’

Extra SORT stage in the query plan, Sort by ‘`timestamp`’

The results to be sorted are obtained using index on ‘`user`’ field.

Index entry to be examined

# Sort not supported by index

```
db.revisionsWI.find(
    {
        "user":"ThiefOfBagdad"
    }
).sort({"timestamp":1}).explain("executionStats")
```

0.001 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| executionStats | { 6 fields } |
| executionSuccess | true |
| nReturned | 68 |
| executionTimeMillis | 0 |
| totalKeysExamined | 68 |
| totalDocsExamined | 68 |
| executionStages | { 14 fields } |
| stage | SORT |
| nReturned | 68 |
| executionTimeMillisEstimate | 0 |
| works | 140 |
| advanced | 68 |
| needTime | 70 |
| needYield | 0 |
| saveState | 2 |
| restoreState | 2 |
| isEOF | 1 |
| sortPattern | { 1 field } |
| memUsage | 17298 |
| memLimit | 33554432 |
| inputStage | { 11 fields } |

Extra execution stage for sort and its memory usage

# No sort comparison

```
db.revisionsWI.find(
    {
        "user":"ThiefOfBagdad"
    }
).explain("executionStats")
```

0.001 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisionsWI |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 2 fields } |
| stage | FETCH |
| inputStage | { 11 fields } |
| rejectedPlans | [ 0 elements ] |
| executionStats | { 6 fields } |
| executionSuccess | true |
| nReturned | 68 |
| executionTimeMillis | 0 |
| totalKeysExamined | 68 |
| totalDocsExamined | 68 |
| executionStages | { 13 fields } |
| stage | FETCH |
| nReturned | 68 |
| executionTimeMillisEstimate | 0 |
| works | 69 |
| advanced | 68 |
| needTime | 0 |
| needYield | 0 |
| saveState | 0 |
| restoreState | 0 |
| isEOF | 1 |
| docsExamined | 68 |
| alreadyHasObj | 0 |
| inputStage | { 24 fields } |

No sort modifier in the query, only search by ‘user’ field

Only a FETCH stage

# SORT supported by index

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

0.001 sec.

| Key | Value |
|---|---|
| (1) | { 4 fields } |
| queryPlanner | { 6 fields } |
| plannerVersion | 1 |
| namespace | wikipedia.revisionsWI |
| indexFilterSet | false |
| parsedQuery | { 1 field } |
| winningPlan | { 3 fields } |
| stage | FETCH |
| filter | { 1 field } |
| title | { 1 field } |
| $eq | Donald_Trump |
| inputStage | { 11 fields } |
| 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 } |
| timestamp | [ 1 element ] |
| [0] | [new Date(1467331200000), new Date(1467417600000)] |
| rejectedPlans | [ 1 element ] |

Search fields are '`title`' and '`timestamp`'
sort field is '`timestamp`'

Theoretically, if we use index on "`timestamp`" to find the results, we can obtain sort order from the index

This is the chosen plan with a FETCH stage. No extra sort is needed

In fact, the winning plan is exactly the same as the one without sort modifier. Try it yourself.

Only one rejected plan in this case

# SORT supported by index (rejected plan)

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

0.001 sec.

| Key | Value |
|---|---|
| winningPlan | { 3 fields } |
| rejectedPlans | [ 1 element ] |
| [0] | { 3 fields } |
| stage | SORT |
| sortPattern | { 1 field } |
| timestamp | 1.0 |
| inputStage | { 2 fields } |
| stage | SORT_KEY_GENERATOR |
| inputStage | { 3 fields } |
| stage | FETCH |
| filter | { 1 field } |
| inputStage | { 11 fields } |
| stage | IXSCAN |
| keyPattern | { 1 field } |
| indexName | title_1 |
| isMultiKey | false |
| multiKeyPaths | { 1 field } |
| isUnique | false |
| isSparse | false |
| isPartial | false |
| indexVersion | 2 |
| direction | forward |
| indexBounds | { 1 field } |

Use index on "`title`" to find the result, and sort based on "`timestamp`"

# Index Usage in Aggregation Pipeline

- Index can be used in some pipeline stages under certain conditions:
  - **`$match`** stage if it is the first in the pipeline
  - **`$sort`** stage if the original document has not been changed, e.g. there is no **`$project`**, **`$unwind`** or **`$group`** stage in front
  - **`$group`** stage if the grouping key is sorted right before and if the grouping accumulator is **`$first`**
  - A few other stages under respective conditions
- The output of explain on information on pipe stage is limited
- See lab question for details

https://docs.mongodb.com/manual/core/aggregation-pipeline/#aggregation-pipeline-operators-and-performance

# References

- MongoDB documentation on indexes
  - https://docs.mongodb.com/manual/indexes/
- MongoDB documentation on **`explain()`** method
  - https://docs.mongodb.com/manual/reference/explain-results/