Mongo DB - Indexes

pm_jat @ daiict

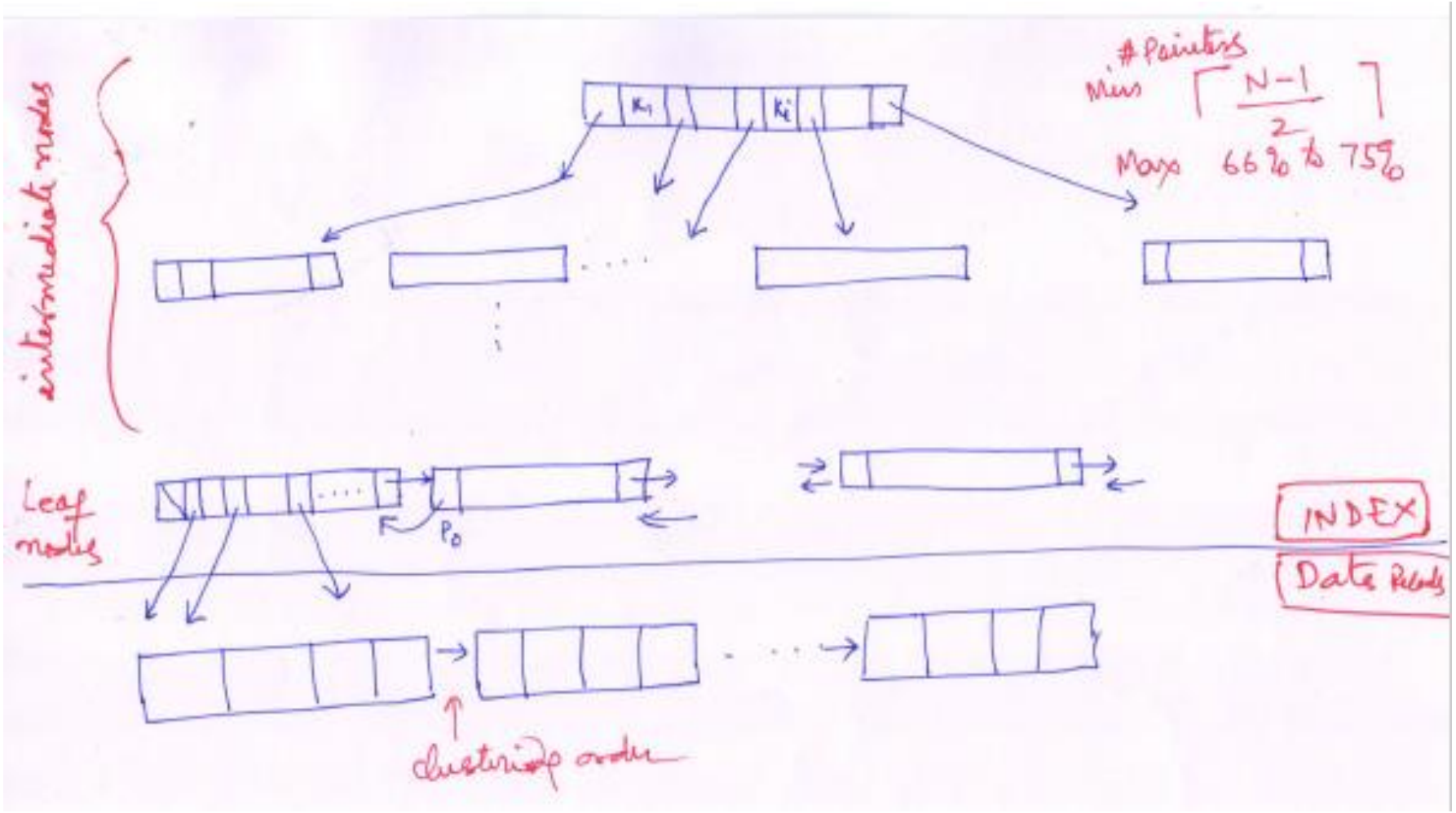
A important note!

- There are cursor methods
  - count, max, sort
- There are also $COUNT, $MAX, $SORT in aggregation pipeline.
- Both sound like similar in terms of functionality; however, are different in terms of execution space.
- Cursor methods are “mongo shell” only operations and are not available in API
- That means “cursor methods” are
  - Not distributed (executed on client)
  - Not under the scope of “mongo query optimizer”

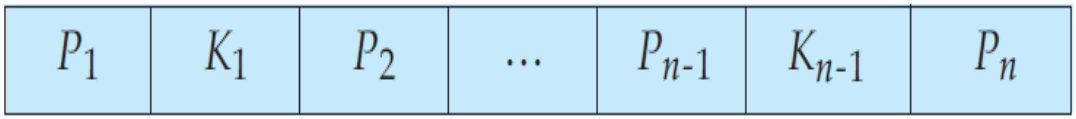
Why do we need indexes in Databases?

- Single Reason: "**to make searching most efficient**"
- Availability of indexes also helps in getting the resulted **sorted**!

B+ tree index?

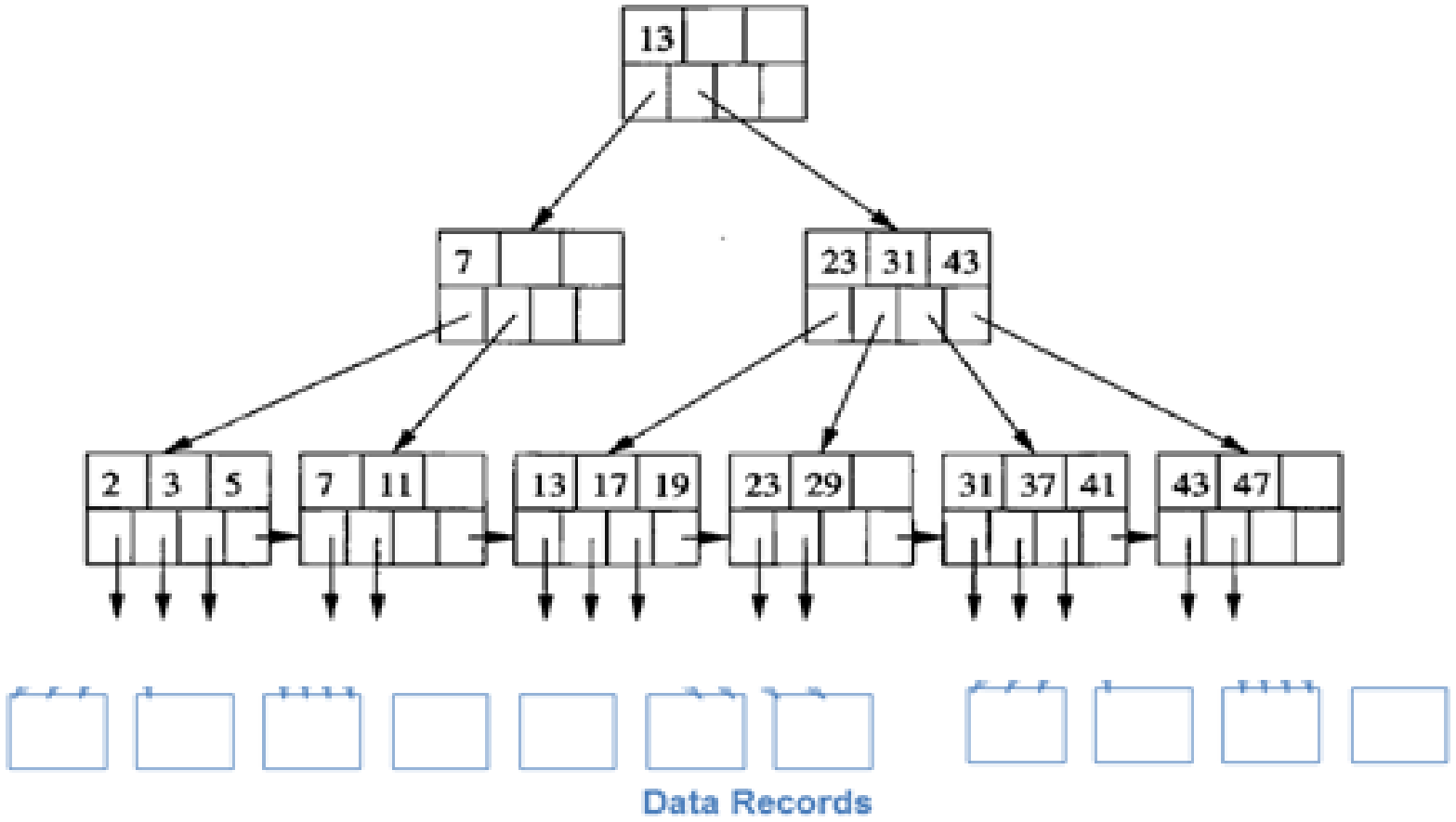

intermediate nodes
Pointers
Min
N-1
2
Max 66% to 75%
Leaf nodes
INDEX
Data Records
clustering order

B+ tree “Node”



- Non-leaf node
  - Non leaf nodes contain pointers to its children nodes
  - A pointer Pi in a node points to record with search key < Ki and >= Ki-1
  - A search value appearing parent node will also appear in child node.
- Leaf nodes
  - Leaf nodes only store address for data records
  - All leaf nodes combined will be in the order of search key.
  - The pointer Pn (that is last pointer) of a leaf node point to next leaf node, and maintains a linked list of leaf nodes.
  - Leaf node may also be linked doubly; may have a pointer like P0 referring to previous node.

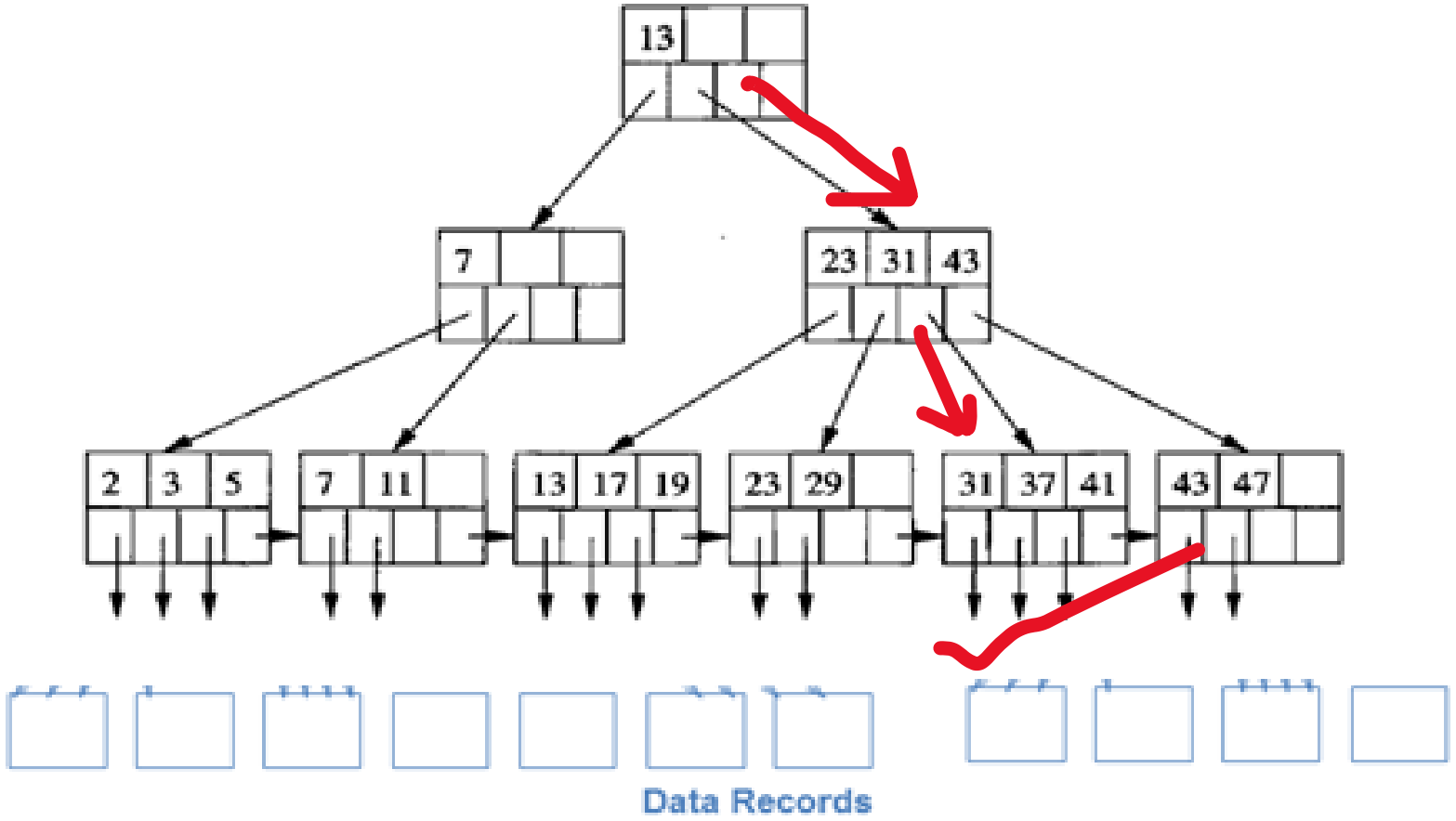
B+ tree index: toy example



Let us say this is index for items. Let search key here be item-id.

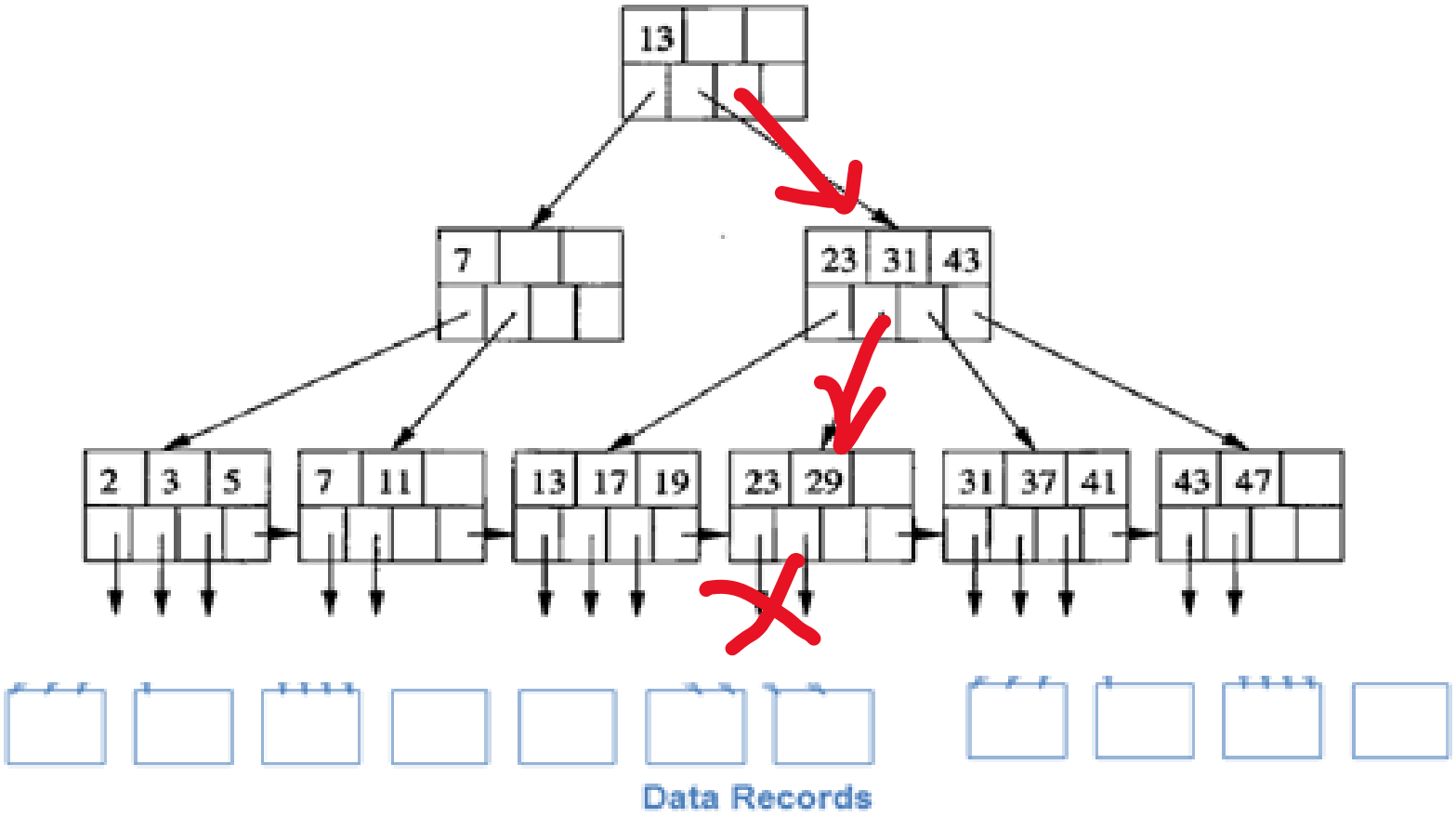
Can you trace how item id = 31 be searched?

B+ tree index: toy example



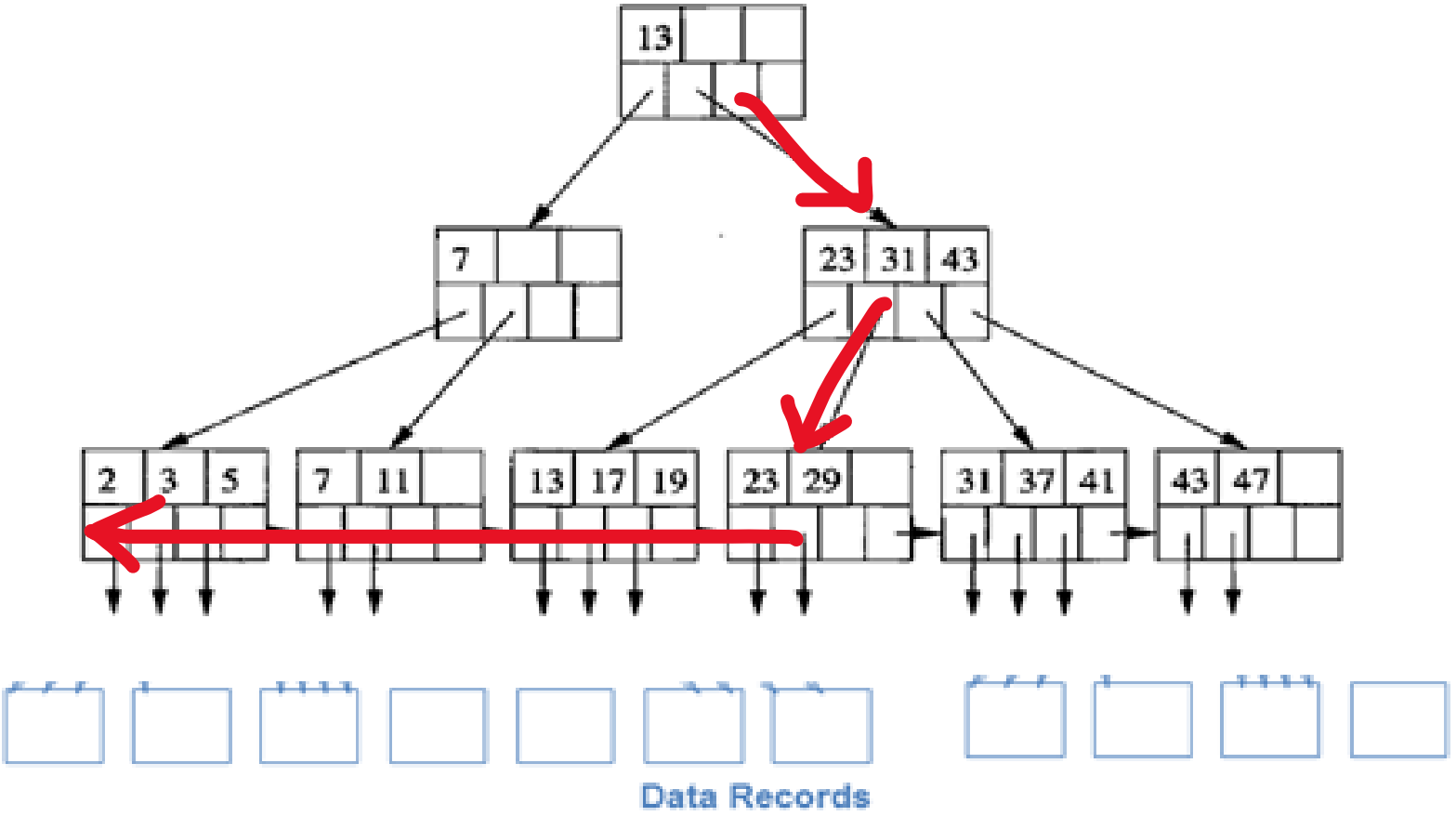
Access path for item id = 31

B+ tree index: toy example



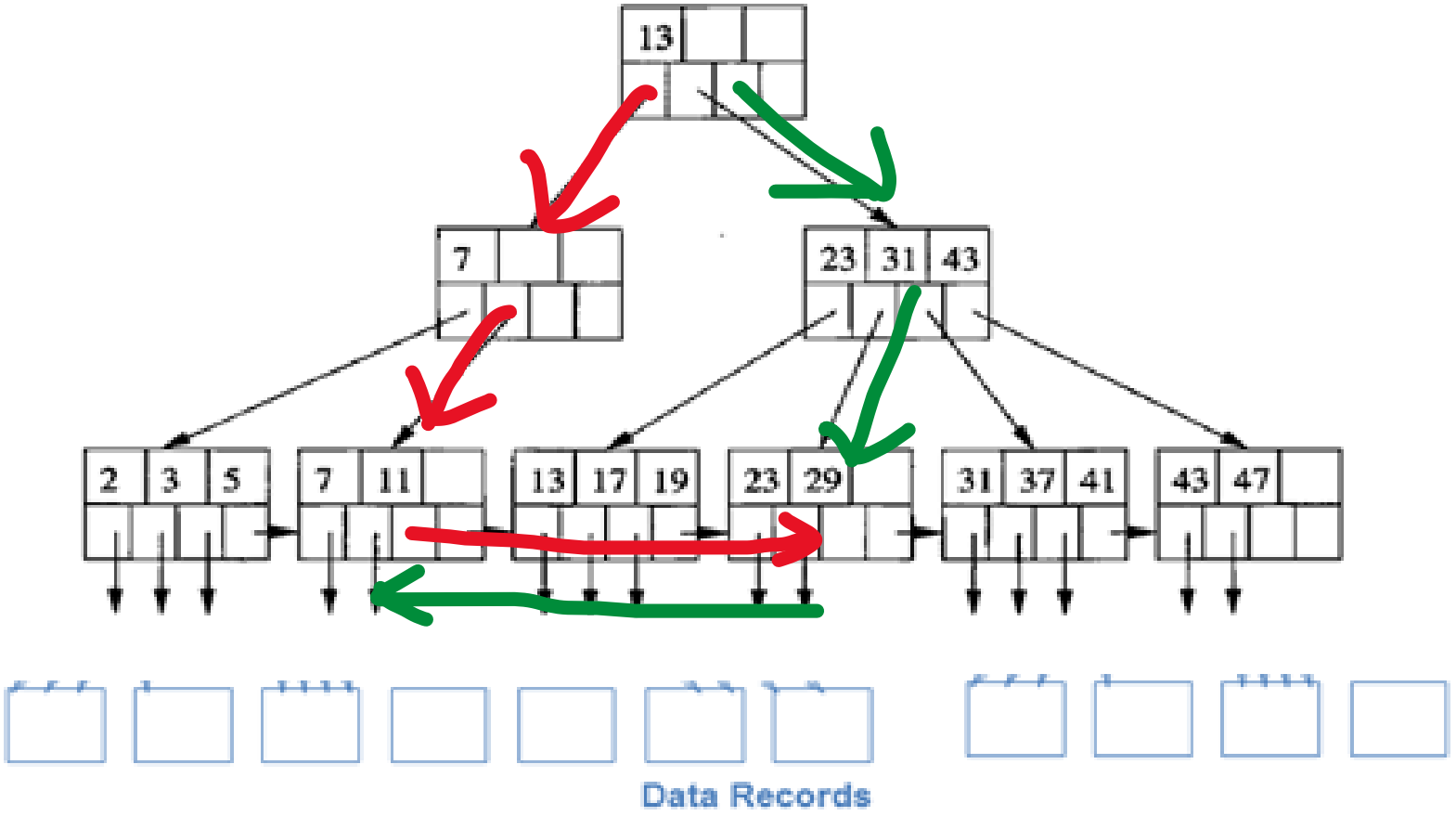
Access path for item id = 30?

B+ tree index: toy example



Traversal path for item id <= 30?

B+ tree index: toy example



Traversal for 10 <=x<= 30 … ascending order

Traversal for 10 <=x<= 30 … descending order

Indexes and Query execution

- Hope you know indexes are used in Relational Databases?
- How will indexes affect execution of following query?
  1. SELECT * FROM PRODUCTS WHERE CAT=5
  2. SELECT * FROM PRODUCTS WHERE CAT=5 and PRICE >=2000 and PRICE < 5000
  3. SELECT * FROM CUSTOMERS WHERE STATE="ABCD"
  4. SELECT * FROM CUSTOMERS WHERE ZIP="122345"
- Availability of indexes on attributes in predicates makes query execution efficient?

Indexes and Query execution

- For instance: SELECT * FROM PRODUCTS WHERE CAT=5
- For getting PRODUCTS of given CAT, we have following options
  - First: Sequentially scan all products and select the one that have CAT=5. BUT this is brute force and not efficient.
  - Two: Sort and Search on CAT
  - Third: TABLE PRODUCTS have index on CAT.
- An index on CAT makes this query executing this query most efficient.
- Interestingly DBMS uses indexes transparently, and we do not have to tell if index is to be used.

Indexes and Query execution

- Query: SELECT * FROM PRODUCTS WHERE CAT=5 and PRICE >=2000 and PRICE < 5000
- Best case is when we have composite index on both attributes
- It can still be efficient if there is index on one of attribute (say CAT)
  - Index lookup on indexing field (CAT in this case) and scan for other field
- (Sequential | Sort and Search) when no index

Good and Downside of Indexes

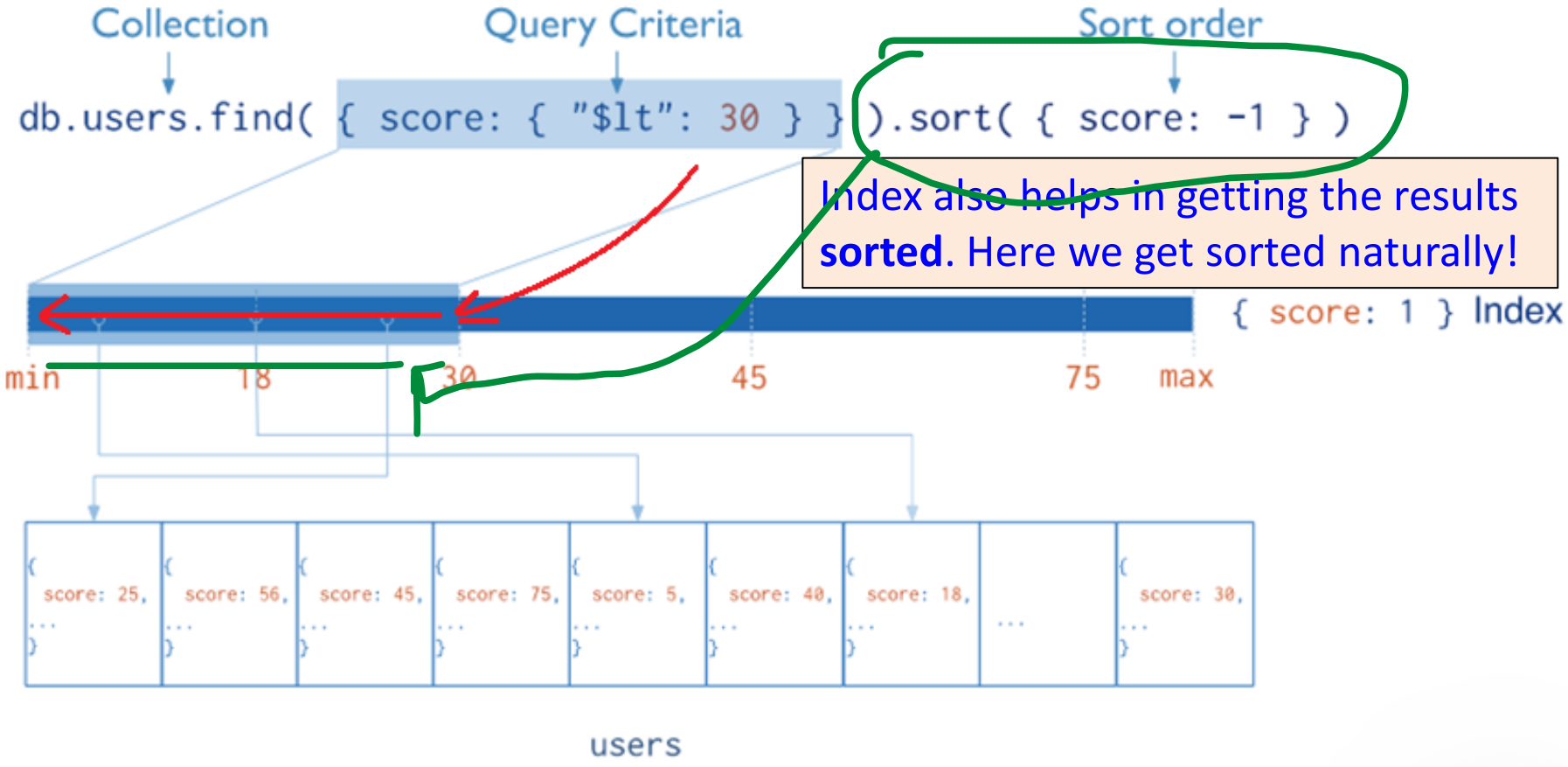
- Good:
  - Helps in improving searches
- Downside:
  - Since DBMS require updating indexes also when data items are getting added and updated.
  - This affects the performance of update operations and hence update operations become slower.
- Therefore, we can not have index on every field and we can not afford having no index at all?

Mongo DB Indexes

- Mongo provides very comprehensive support for Indexes
- Mongo DB indexes are B+ tree based.
  - However, you can also create Hashed Indexes. Hashed Indexes are only good for equality searches
- Mongo DB automatically makes use of indexes wherever indexes are available!

Mongo DB Indexes

- How availability of index on score (ascending) on users will be used in answering following query?



Source: https://docs.mongodb.com/manual/indexes/

Default _id Index

- MongoDB creates a unique index on the _id field during the creation of a collection.
- The _id index prevents clients from inserting two documents with the same value for the _id field.
- You cannot drop this index on the _id field.

Source: https://docs.mongodb.com/manual/indexes/

Index Properties

Unique Indexes

- A unique index causes MongoDB to reject duplicate values for the indexed field.

Partial Indexes

- Partial indexes only index the documents in a collection that meet a specified filter expression.
- By indexing a subset of the documents in a collection, partial indexes have lower storage requirements and reduced performance costs for index creation and maintenance.

Source: https://docs.mongodb.com/manual/indexes/

Index Properties

TTL Indexes

- TTL (Time To Live) Indexes lives only for specified time.

Source: https://docs.mongodb.com/manual/indexes/

Creating Index

- Mongo DB provides method `createIndex`
- Parameters: key and other index specifications, and options

```
db.collection.createIndex( <key and index type specification>, <options> )
```

- Index on zip (ascending) for zips collection:
  db.zips.createIndex({ zip: 1 })
- Index on state, population for zips collection:

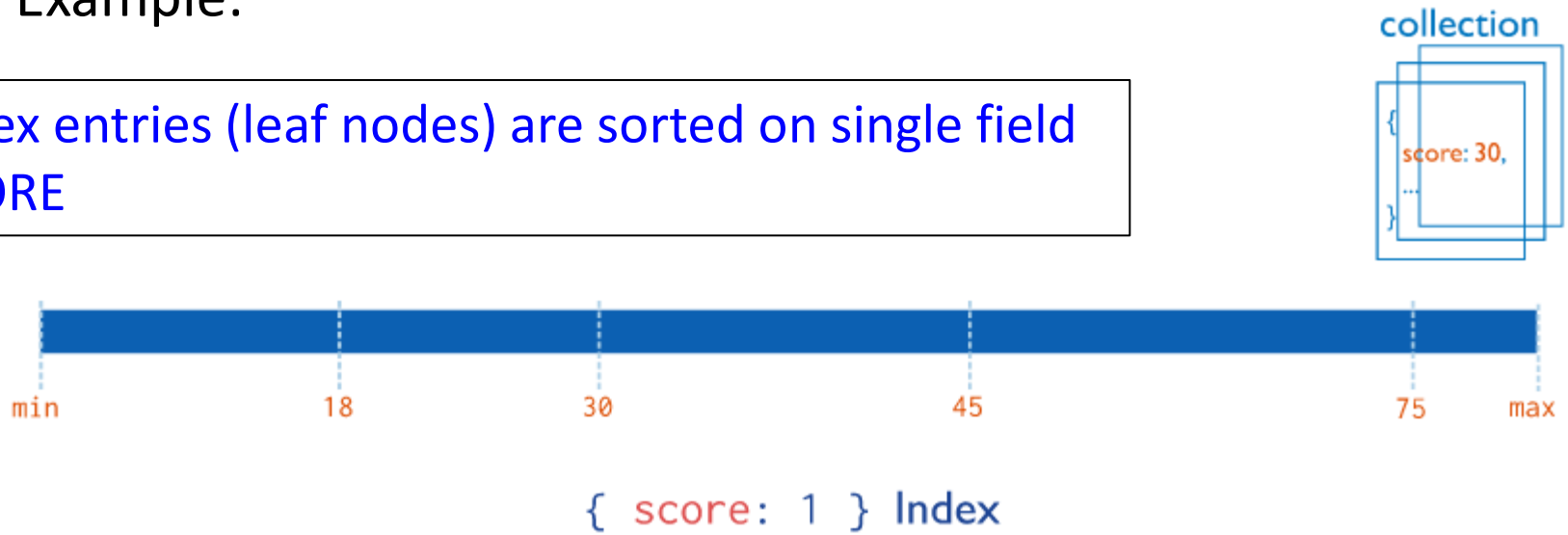
```
db.zips.createIndex(
  { state: 1, pop: 1 },
  { name: "stateindex"}
)
```

Source: https://docs.mongodb.com/manual/indexes/

Index Types

- MongoDB provides a number of index types to support specific types of data and queries.
- Single Field: On a single field: ascending/descending
- Example:

Index entries (leaf nodes) are sorted on single field SCORE

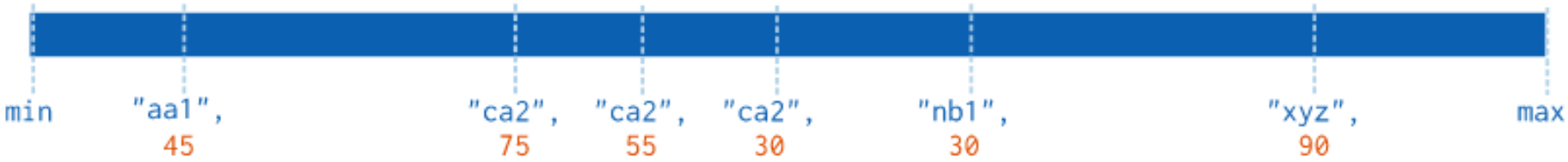


Source: https://docs.mongodb.com/manual/indexes/

Index Types: Compound Index

- Indexes on multiple fields.

Index entries (leaf nodes) are sorted on two field
User, Score



{ userid: 1, score: -1 } Index

Note this index is no more efficient for query

```
db.users.find( { score: { "$lt": 30 } } )
```

Source: https://docs.mongodb.com/manual/indexes/

Index Types: Compound Index

- So index on User and Score on Collection Users will be useful for query like
  - find users where userid=101 and score > 50, and
- Not for query like
  - find users with score > 50

"Covered Queries"

- Wishfully all queries are "covered"
- **A covered query is a query that can be satisfied entirely using an index** and does not require looking into documents.
- An index covers a query when all of the following apply:
  - all the fields in the query are part of an index, and
  - all the fields returned in the results are in the same index.
  - no fields in the query are equal to null (i.e. {"field" : null} or {"field" : {$eq : null}}).

Source: https://docs.mongodb.com/manual/indexes/

Some more index types

- **Geospatial Index**: To support efficient queries of geospatial coordinate data
- **Text Indexes**: to support full text search. Index/Searching is done based root words, etc.
- **Hashed Indexes**: To support hash based sharding, MongoDB provides a hashed index type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries.

Source: https://docs.mongodb.com/manual/indexes/

Dropping indexes

- By field specifications specified in create index, as
  db.zips.dropIndex({ zip: 1 })
- Or by names if we specified name in create index:
  db.zips.dropIndex("stateindex")

Full Text Search | Text Search

- Full Text Searching (or just text search) provides the capability to search in natural-language documents
- Keyword based search is quite popular looking into textual descriptions. Consider following two examples
- (1) Searching in Book titles based on input keywords:
  - Suppose we want to search books with following two keywords in title “Mongo+Java”
  - How do we perform? Probably we can use some RegEx but regular expressions are not index friendly!
- (2) Keyword based search in product descriptions?

Full Text Search | Text Search

- The most common type of search is to find all documents containing given query terms and
  - (1) return them in order of their similarity to the query, or
  - (2) Return only top k in terms of similarity
- Notion of query and similarity are very flexible and depend on the specific application.
- The simplest search considers query as a set of words and similarity as the frequency of query words in the document (popular metric is cosine similarity based on Term Frequency and TFIDF)

Text Indexes in Mongo DB

- MongoDB provides text indexes to support basic support for full text searches.
- Text indexes can include any field whose value is a string or an array of string elements.
- Here is an example creating text index on reviews collection on comments fields:

```
db.reviews.createIndex({comments:"text"})
```

Searching Text index : Toy Example

- Suppose we have collection `comments` with following documents

```
{ "_id" : "1", "statement" : "MongoDB is the worst." }
{ "_id" : "2", "statement" : "MongoDB is the best" }
{ "_id" : "3", "statement" : "DynamoDB is the worst" }
{ "_id" : "4", "statement" : "DynamoDB is the best" }
```

- And let us create text index as `db.comments.createIndex({statement: "text"})`
- And let us try search for keywords “MongoDB Best”?

Searching Text index : Toy Example

- Now if we attempt searching:

```
db.comments.find(
{ $text: { $search: "MongoDB Best" } }
)
```

- Returns following. Does it surprizes?

```
{ "_id" : "4", "statement" : "DynamoDB is the best" }
{ "_id" : "2", "statement" : "MongoDB is the best" }
{ "_id" : "1", "statement" : "MongoDB is the worst." }
```

Searching Text index : Toy Example

- Following query results as given below:

```
db.textExample.find({
    $text: { $search : "MongoDB best" } },
    { score: { $meta: "textScore" } })
.sort({ score: { $meta: "textScore" } })
```

```
{ "_id" : "2", "statement" : "MongoDB is the best", "score" : 1.5 }
{ "_id" : "4", "statement" : "DynamoDB is the best", "score" : 0.75 }
{ "_id" : "1", "statement" : "MongoDB is the worst.", "score" : 0.75 }
```

- We have text index built on field on which we want perform full text search
- Also note that
- Success lies in the computation of similarity

More characteristics of Text Indexes

- We can specify language for removing “stop words” and “stemming”.
- A collection can have at most one text index.
- However we can include multiple fields in an index.
- We can also specify weights to each field match
- We can also create “wildcard text” indexes that means index gets built on all text fields,
- So forth

More powerful full text search on Mongo DB

- Full text becomes more important in semi structured data where we find many fields are in text forms.
- Elasticsearch or Apache Solr are popular large scale full text search engines with many machine intelligence features.
- MongoDB allows us integrating with such external full text search engines.
- “MongoDB Atlas” already provides integrated solution with lucene

Getting started with MongoDB Atlas Full-Text Search
https://www.mongodb.com/blog/post/getting-started-with-mongodb-atlas-fulltext-search

Mongo DB Query Planner

- In database world “Query Optimization” often refers to putting an efficient “**query plan**” for executing a query.
- What we mean by “query plan”?
  - Basically means having a execution pipeline.
  - Order of database operations (at logical level): selection (filter), projection, grouping, aggregation, etc.
  - Order and “method” of operations at physical level. Often there are multiple methods of performing a logical operations, and one is more suited over the other in certain situations. For instance “collection scan”, sort-scan, index-scan for searching through a collection.
- Mongo DB also provides an underlying query optimizer.

Mongo DB Query Planner

- For a query, the MongoDB query optimizer chooses and caches the most efficient query plan given the available indexes.
- The evaluation of the most efficient query plan is based on the number of “work units” (works) performed by the query execution plan when the query planner evaluates candidate plans.
- The associated plan cache entry is used for subsequent queries with the same “query shape”.
- To view query plan for a query, we can use `db.collection.explain()` or the `cursor.explain()`. Here we see what plan is winner and its details.

db.collection.explain()

- Helps you finding out
  - What plan has been finally executed - what has been various stages?
  - What other query plans
  - Is query using index for sort, and so
  - What part of query

Examples: db...explain()

- Here are some examples

```
db.zips.explain().find({ "zip": "35004"})

db.zips.explain("executionStats").find({ "zip": "35004"} })

db.zips.explain().find(
  { "state": "PA", pop: { $gte: 50000 } },{ "_id": 0, "city": 1 }
)

db.zips.explain().aggregate( [
  { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
  { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

References and Further Readings

[1] https://docs.mongodb.com/manual/

[2] Most content put here is from:
https://docs.mongodb.com/manual/indexes/

[3] MongoDB University course “Mongo DB Performance”:
https://university.mongodb.com/mercury/M201/2021_March_16

[4] Getting started with MongoDB Atlas Full-Text Search
https://www.mongodb.com/blog/post/getting-started-with-mongodb-atlas-fulltext-search

[5] Apache Solr: Because your Database is not a Search: Engine
https://programmaticponderings.com/2019/02/25/apache-solr-because-your-database-is-not-a-search-engine/

[6] Solr vs. Elasticsearch: Who’s The Leading Open Source Search Engine
https://logz.io/blog/solr-vs-elasticsearch/