Document Databases (MongoDB)

Imperial College London

Content

- Introduction & Basics
- CRUD
- Schema Design
- Indexes
- Aggregation

Motivations

Problems with SQL

- Rigid schema
- Not easily scalable
- Requires unintuitive joins

Perks of mongoDB

- Easy interface with common languages (Java, Javascript, PHP, etc.)
- DB tech should run anywhere (VM's, cloud, etc.)
- Keeps essential features of RDBMS's while learning from key-value noSQL systems

Data Model

- Document-Based (max 16 MB)
- Documents are in BSON format, consisting of field-value pairs
- Each document stored in a collection
- Collections
  - Have index set in common
  - Like tables of relational DB's.
  - Documents do not need to have uniform structure

JSON

- “JavaScript Object Notation”
- Easy for humans to write/read, easy for computers to parse/generate
- Objects can be nested
- Built on
  - name/value pairs
  - Ordered list of values

BSON

- “Binary JSON”
- Binary-encoded serialization of JSON-like docs
- Also allows “referencing”
- Embedded structure reduces need for joins
- Goals
  - Lightweight
  - Traversable
  - Efficient (decoding and encoding)

BSON Example

```
{
"_id" :         "37010"
"city" :        "ADAMS",
"pop" :         2660,
"state" :       "TN",
“councilman” : {
                    name: “John Smith”
                    address: “13 Scenic Way”
                  }
}
```

BSON Types

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

The number can be used with the $type operator to query by type!

http://docs.mongodb.org/manual/reference/bson-types/

The _id Field

By default, each document contains an _id field. This field has a number of special characteristics:

- Value serves as primary key for collection.
- Value is unique, immutable, and may be any non-array type.
- Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.” Sorting on an ObjectId value is roughly equivalent to sorting on creation time.

mongoDB vs. SQL

mongoDB	SQL
Document	Tuple
Collection	Table/View
PK: _id Field	PK: Any Attribute(s)
Uniformity not Required	Uniform Relation Schema
Index	Index
Embedded Structure	Joins

CRUD: Using the Shell

To check which db you're using

```
db
```

Show all databases

```
show dbs
```

Switch db's/make a new one

```
use <name>
```

See what collections exist

```
show collections
```

CRUD: Using the Shell (cont.)

To insert documents into a collection/make a new collection:

db.<collection>.insert(<document>)

<=>

INSERT INTO <table>

VALUES(<attributevalues>);

CRUD: Inserting Data

Insert one document

db.<collection>.insert({<field>:<value>})

Inserting a document with a field name new to the collection is inherently supported by the BSON model.

To insert multiple documents, use an array.

CRUD: Querying

- Done on collections
- Get all docs: db.<collection>.find()
  - Returns a cursor, which is iterated over shell to display first 20 results.
  - Add .limit(<number>) to limit results
  - SELECT * FROM <table>;
- Get one doc: db.<collection>.findOne()

CRUD: Querying

To match a specific value:

db.<collection>.find({<field>:<value>})

“AND”

db.<collection>.find({<field1>:<value1>,
<field2>:<value2>
})

SELECT *
FROM <table>
WHERE <field1> = <value1> AND <field2> = <value2>;

CRUD: Querying

OR

```
db.<collection>.find({ $or: [
<field>:<value1>,
<field>:<value2>        ]
})
```

```
SELECT *
FROM <table>
WHERE <field> = <value1> OR <field> = <value2>;
```

Checking for multiple values of same field

```
db.<collection>.find({<field>: {$in [<value1>, <value2>]}})
```

CRUD: Querying

Including/excluding document fields

db.<collection>.find({<field1>:<value>}, {<field2>: 0})

SELECT field1
FROM <table>
WHERE <field1> = <value>;

db.<collection>.find({<field>:<value>}, {<field2>: 1})

Find documents with or w/o field

db.<collection>.find({<field>: { $exists: true}})

CRUD: Updating

db.<collection>.update(

{<field1>:<value1>}, //all docs in which field = value

{$set: {<field2>:<value2>}}, //set field to value

{multi:true}) //update multiple docs

upsert: if true, creates a new doc when none matches search criteria.

UPDATE <table>
SET <field2> = <value2>

WHERE <field1> = <value1>;

CRUD: Updating

To remove a field

```
db.<collection>.update({<field>:<value>},
                       { $unset: { <field>: 1}})
```

Replace all field-value pairs

```
db.<collection>.update({<field>:<value>},
                       { <field>:<value>, <field>:<value>})
```

NOTE: This overwrites ALL the contents of a document, even removing fields.

CRUD: Removal

Remove all records where field = value

```
db.<collection>.remove({<field>:<value>})
```

```
DELETE FROM <table>
WHERE <field> = <value>;
```

As above, but only remove first document

```
db.<collection>.remove({<field>:<value>}, true)
```

CRUD: Isolation

- By default, all writes are atomic **only** on the level of a single document.
- This means that, by default, all writes can be interleaved with other operations.

Mongo is basically schema-free

- The purpose of schema in SQL is for meeting the requirements of tables and quirky SQL implementation

- Every "*row*" in a database "*table*" is a data structure, much like a "struct" in C, or a "class" in Java. A table is then an array (or list) of such data structures

- So what mongoDB design is basically same way how we design a compound data type binding in JSON

Flexible Schemas in MongoDB

```
db.inventory.insert(
  {
    category: ”vacuum",
    details: {
      model: "14Q3",
      manufacturer: "XYZ Company"
    },
    stock: [ { size: "S", qty: 25 }]
  }
)
```

```
db.inventory.insert(
  {
    category: ”vacuum",
    details: {
      model: "14Q2",
      manufacturer: "XYZ Company"
    },
    color: ”blue"
  }
)
```

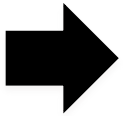
What fields does db.inventory.find ({”category" : ”vacuum”}) have?

Patterns

- Embedding
- Linking

One to One relationship

```
zip = {
 _id: 35004,
city: “ACMAR”,
 loc: [-86, 33],
pop: 6065,
State: “AL”
}

Council_person = {
zip_id = 35004,
name: “John Doe",
address: “123 Fake St.”,
Phone: 123456
}
```



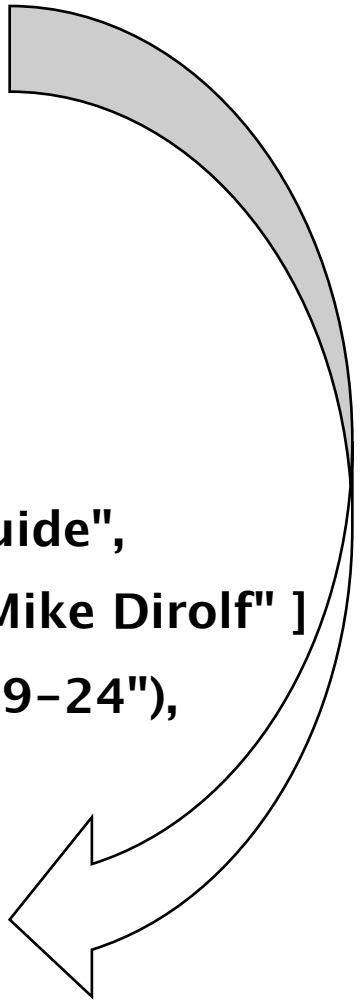
```
zip = {
        _id: 35004 ,
        city: “ACMAR”
        loc: [-86, 33],
        pop: 6065,
        State: “AL”,

        council_person: {
                name: “John Doe",
                address: “123 Fake St.”,
                Phone: 123456
        }

}
```

One to many relationship - Embedding

```
book = {
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher: {
        name: "O'Reilly Media",
        founded: "1980",
        location: "CA"
    }
}
```

One to many relationship – Linking

```
publisher = {
    _id: "oreilly",
    name: "O'Reilly Media",
    founded: "1980",
    location: "CA"
}
book = {
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English",
    publisher_id: "oreilly"
}
```



Linking vs. Embedding

- Embedding is a bit like pre-joining data
- Document level operations are easy for the server to handle
- Embed when the “many” objects always appear with (viewed in the context of) their parents.
- Linking when you need more flexibility

Collection Example

```
book = {
    title: "MongoDB: The Definitive Guide",
    authors : [
        { _id: "kchodorow", name: "Kristina Chodorow" },
        { _id: "mdirolf", name: "Mike Dirolf" }
    ]
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

author = {
    _id: "kchodorow",
    name: "Kristina Chodorow",
    hometown: "New York"
}

db.books.find( { authors.name : "Kristina Chodorow"
} )
```

Modelling Example

- Book can be checked out by one student at a time
- Student can check out many books

Modeling Checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader",
    join_date: ISODate("2011-10-15"),
    address: { ... }
}

book = {
    _id: "123456789"
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ],
    ...
}
```

Modeling Checkouts

```
student = {
    _id: "joe"
    name: "Joe Bookreader",
    join_date: ISODate("2011-10-15"),
    address: { ... },
    checked_out: [
        { _id: "123456789", checked_out: "2012-10-15" },
        { _id: "987654321", checked_out: "2012-09-12" },
        ...
    ]
}
```

What is good about mongoDB?

- find() is more semantically clear for programming

```
(map (lambda (b) b.title)
     (filter (lambda (p) (> p 100)) Book)
```

- De-normalization provides **Data locality, and Data locality provides speed**

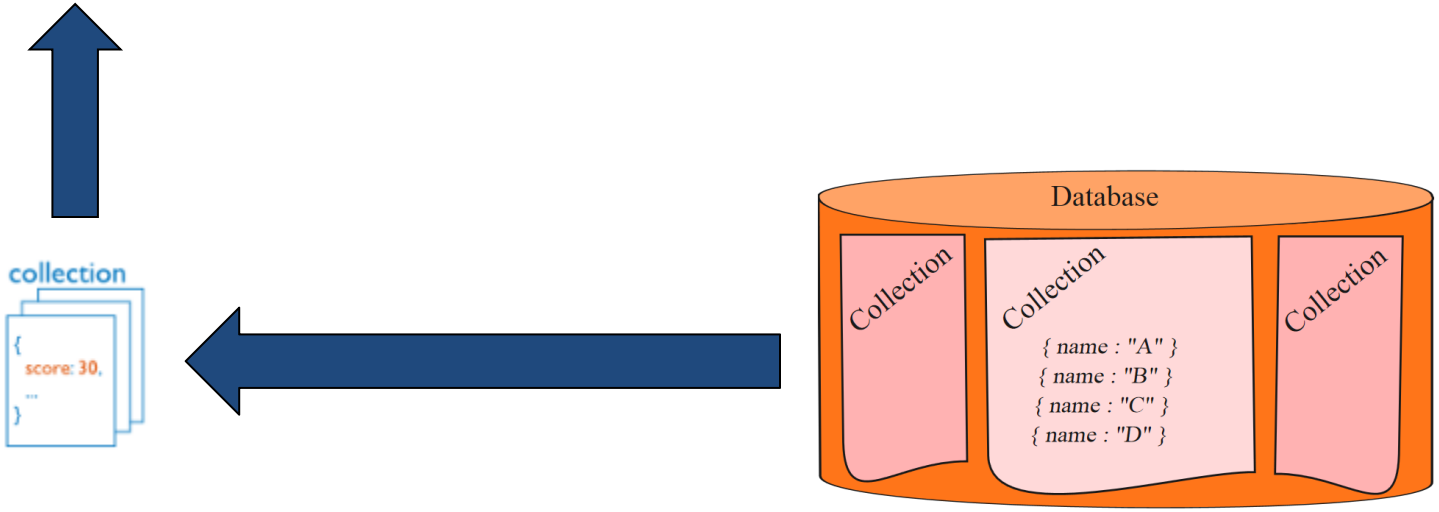
Before Index

What does database normally do when we query?

- **MongoDB must scan every document.**

  db.users.find({ score: { “$lt” : 30} })

- **Inefficient because process large volume of data**



Definition of Index

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

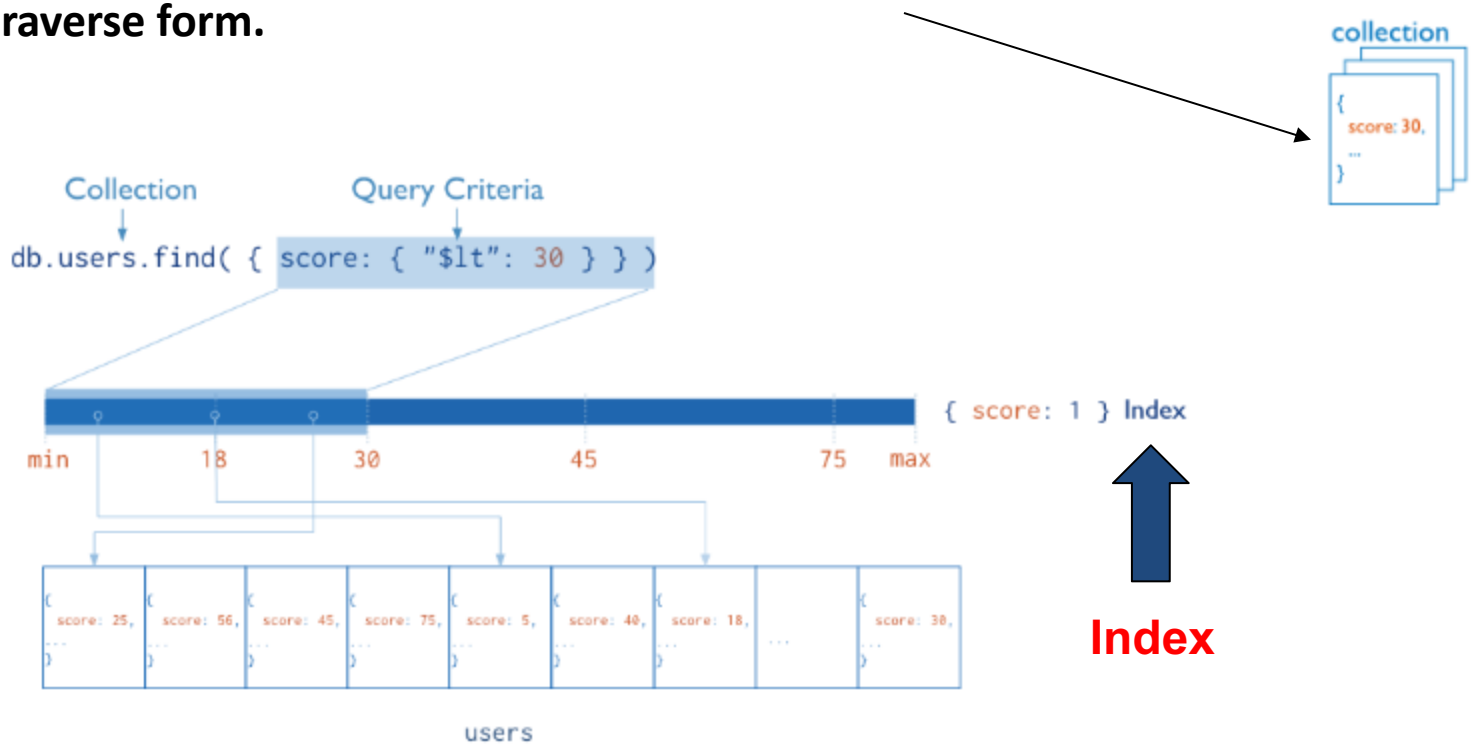


Diagram of a query that uses an index to select

Index in MongoDB

Creation index

db.users.ensureIndex({ score: 1 })

Show existing indexes

db.users.getIndexes()

Drop index

db.users.dropIndex({score: 1})

Explain—Explain

db.users.find().explain()

Returns a document that describes the process and indexes

Hint

db.users.find().hint({score: 1})

Overide MongoDB's default index selection

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Single Field Indexes**
  - db.users.ensureIndex({ score: 1 })

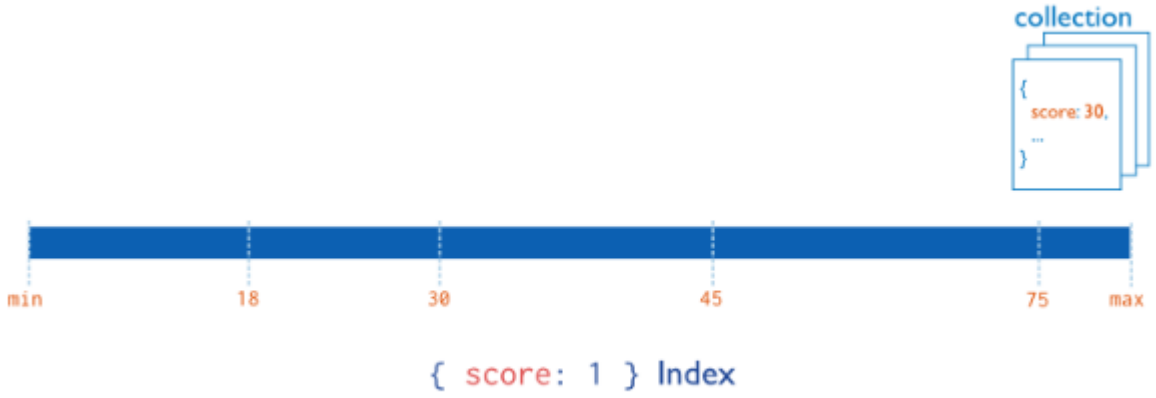


Diagram of an index on the score field (ascending).

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Compound Field Indexes**
  - db.users.ensureIndex({ userid:1, score: -1 })

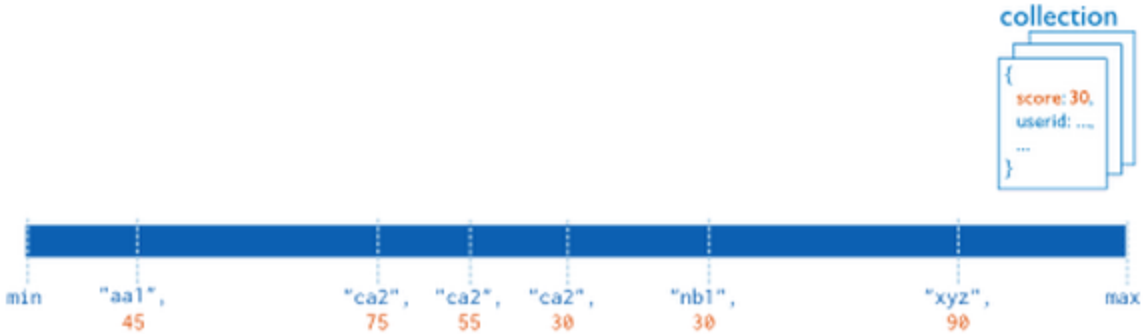


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

Index in MongoDB

Types

- **Single Field Indexes**
- **Compound Field Indexes**
- **Multikey Indexes**

- **Multikey Indexes**
  - db.users.ensureIndex({ addr.zip:1})

{ "addr.zip": 1 } Index

Diagram of a multikey index on the `addr.zip` field. The `addr` field contains an array of address documents. The address documents contain the `zip` field.

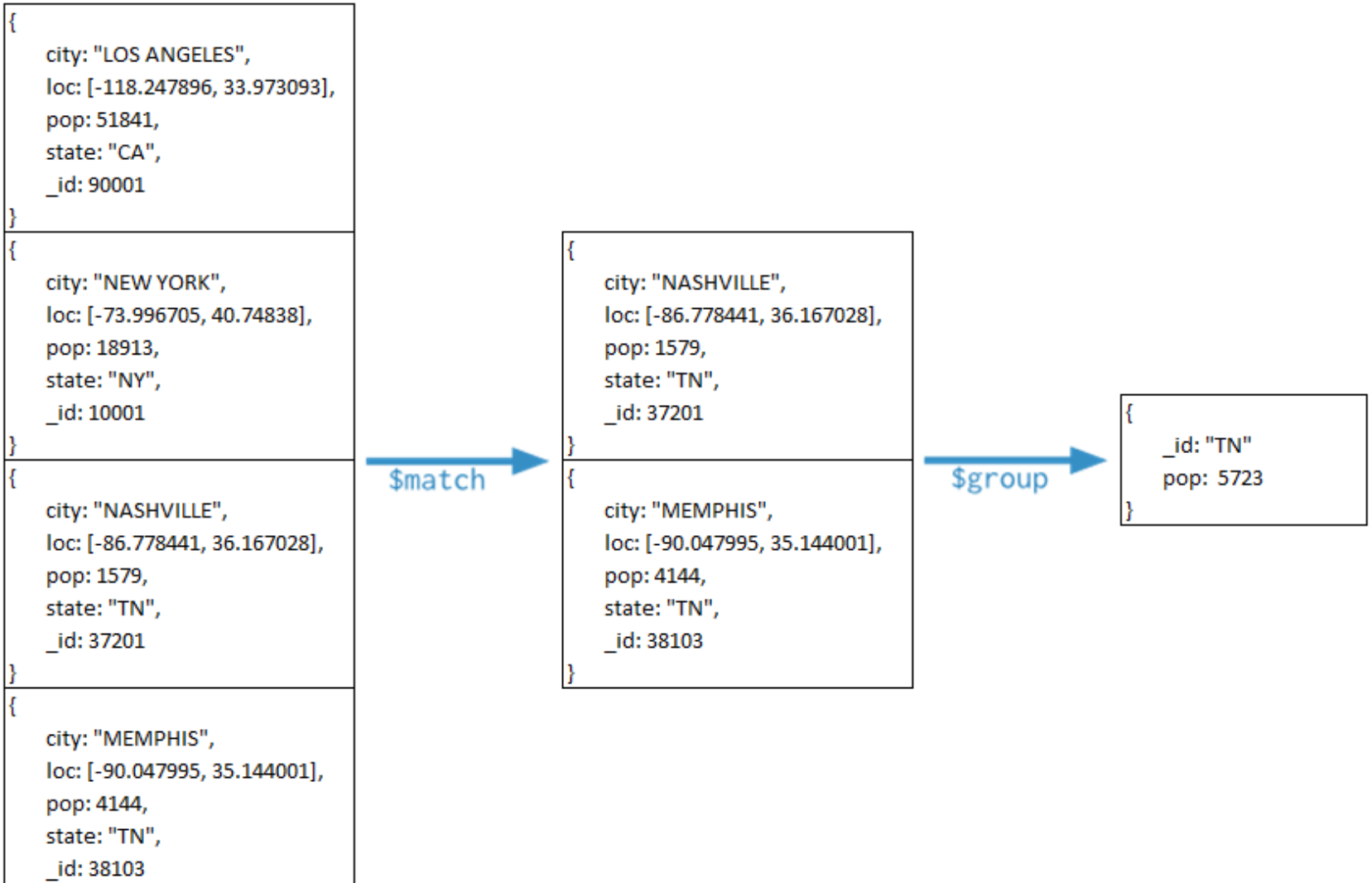
Aggregation

- Operations that process data records and return computed results.
- MongoDB provides aggregation operations
- Running data aggregation on the mongod instance simplifies application code and limits resource requirements.

Pipelines

- Modeled on the concept of data processing pipelines.
- Provides:
  - *filters* that operate like queries
  - *document transformations* that modify the form of the output document.
- Provides tools for:
  - grouping and sorting by field
  - aggregating the contents of arrays, including arrays of documents
- Can use operators for tasks such as calculating the average or concatenating a string.

```
db.zips.aggregate(
                    { $match: { state: "TN" } },
                    { $group: {_id: "TN", pop: { $sum: "$pop" } } }
                  );
```



Pipelines

- $limit
- $skip
- $sort

```
db.zips.distinct( "state" );
```

```
{
    city: "LOS ANGELES",
    loc: [-118.247896, 33.973093],
    pop: 51841,
    state: "CA",
    _id: 90001
}
{
    city: "NEW YORK",
    loc: [-73.996705, 40.74838],
    pop: 18913,
    state: "NY",
    _id: 10001
}
{
    city: "NASHVILLE",
    loc: [-86.778441, 36.167028],
    pop: 1579,
    state: "TN",
    _id: 37201
}
{
    city: "MEMPHIS",
    loc: [-90.047995, 35.144001],
    pop: 4144,
    state: "TN",
    _id: 38103
}
```