XML, XQuery, JSON & MongoDB

Imperial College
London

Introduction

- XML stands for Extensible Markup Language.
- It is designed to describe data and focus on what data is.
- It is used to structure store and to send information.
- It is easy to understand and is self describing.

Rules

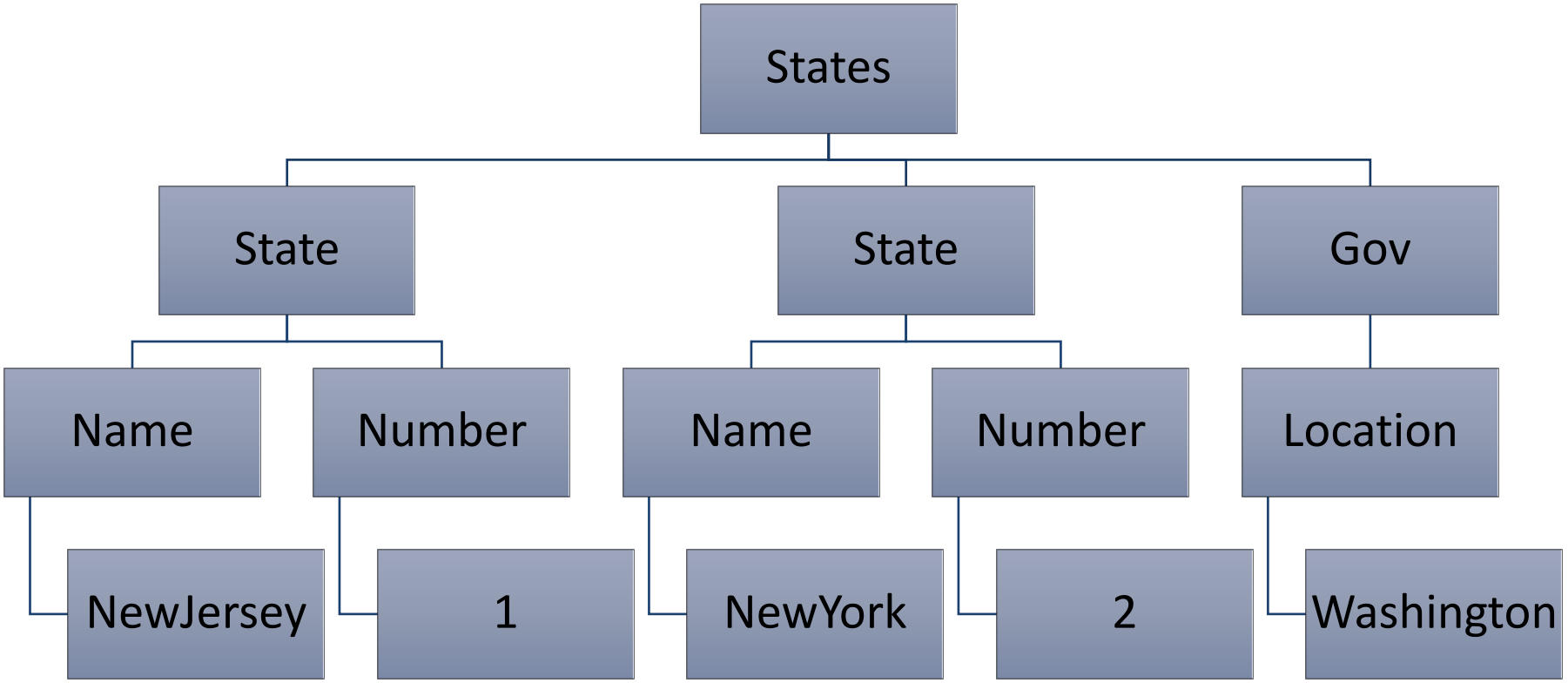
- The first tag is the root of the tree. There must be a single root.
- Every other matching pair of tags becomes one node. If a pair of tags is contained in another pair, the contained pair becomes a child of the containing pair. Children have a defined order.
- Text becomes a child of the node corresponding to the tag that encloses the text. This is always a leaf node.
- XML allows single tag. Single tag always become leaves with a box.
- XML Tags are case sensitive and they

  must be always properly nested.

XML Example

```
<States>
    <State>
        <Name>NewJersey</Name>
        <Number>1</Number>
    </State>
    <State>
        <Name>NewYork</Name>
        <Number>2<Number>
    </State>
    <Gov>
        <Location>Washington</Location>
    </Gov>
</States>
```

Tree Representation of XML



DTD

- A valid XML document is a “well formed” XML document, which also conforms to the rules of a Document Type Definition(DTD).
- A DTD is like a database schema for XML files.

Example of DTD

DTD

```
<?XML version = “1.0”?>
<!DOCTYPE note[
<!Element note(to,from,heading,notebody)>
<!Element to(#PCDATA)>
<!Element from(#PCDATA)>
<!Element heading(#PCDATA)>
<!Element notebody(#PCDATA)>
]>
```

Example XML

```
<note>
    <to>CS 731</to>
    <from>21456687</from>
    <heading>presentation</heading>
    <notebody>XML introduction</notebody>
</note>
```

Interpretation of DTD

- !ELEMENT note defines the note element as having four elements: "to, from, heading, notebody”. In this order.
- <!ELEMENT to(#PCDATA)> defines the “to” element is of type “#PCDATA”.
- PCDATA Parsed Character Data: a character string

Interpretation of DTD Cont..

- Element with children (sequence)

  <!Element note(to,from,heading ,body)>

- Declaring minimum one occurrence of the same element(one or more)

  <!ELEMENT note (+)>

- Declaring zero or more occurrences of the same element

  <!ELEMENT note (*)>

- Declaring zero or one occurrences of the

  same element

  <!ELEMENT note (?)>

Advantages of XML

- XML is an open standard.
- It is human readable and not cryptic like a machine language.
- XML processing is easy.
- It can be used to integrate complex web based systems (using XML as communication).

Importance of XML

- Extensible Markup Language (XML) is fast emerging as the dominant standard for representing data on the Internet.
- Most organizations use XML as a data communication standard.
- All commercial development frameworks are XML oriented (.NET, Java).
- All modern web systems architecture is designed based on XML.

Storing XML in Databases

The primary ways to store XML data can be classified as:

- *Structure-Mapping approach*

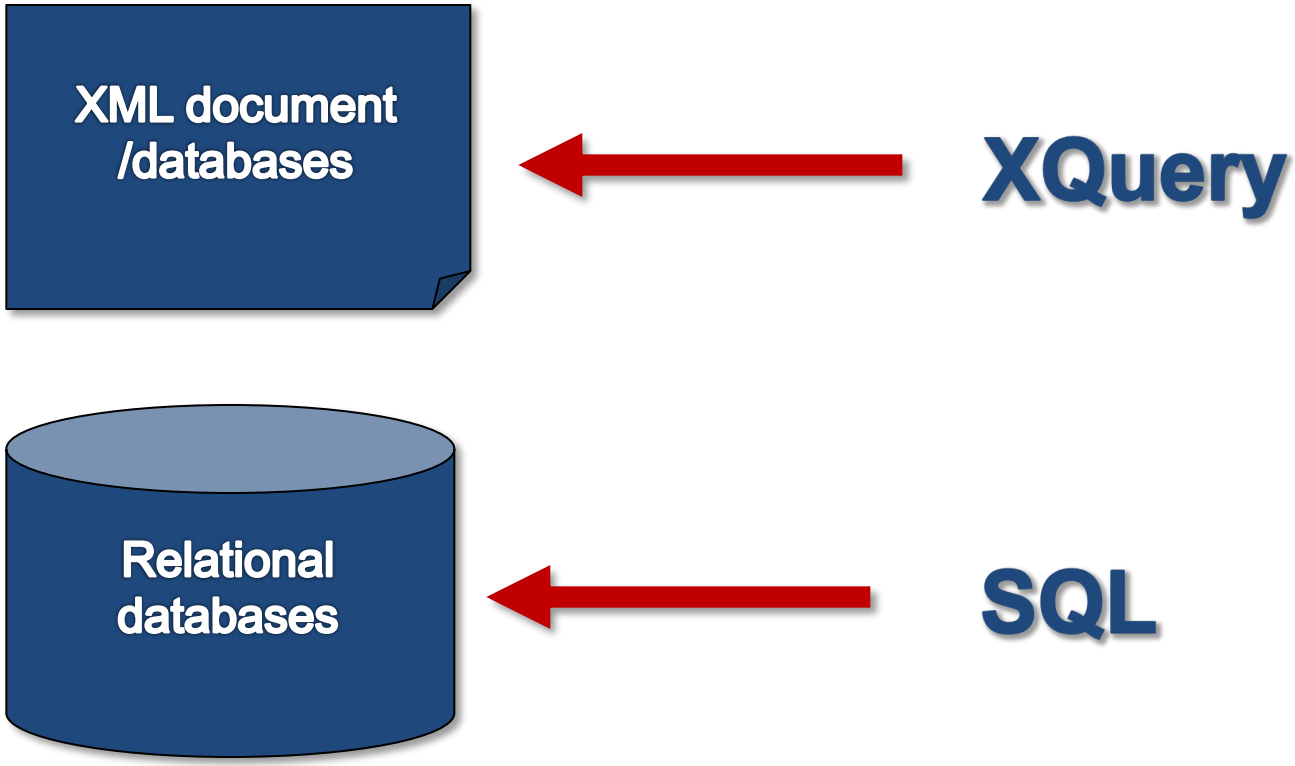
  In the Structure Mapping approach the design of database schema is based on the understanding of DTD (Document Type Descriptor) that describes the structure of XML documents.

- *Model-Mapping approach.*

  In the Model Mapping approach no DTD information is required for data storage. A fixed database schema is used to store any XML documents without assistance of DTD.

XQuery

XQuery



What is XQuery

- Designed to meet the requirements identified by the W3C XML Query Working Group
  - "XML Query 1.0 Requirements"
  - "XML Query Use Cases".
- Designed to be a small, easily implementable language.
- Flexible enough to query a broad spectrum of XML sources (both databases and documents).
- Defines a human-readable syntax for that language.
- A query in XQuery is an expression that:
  - Reads a number of XML documents or fragments
  - Returns a sequence of well-formed XML fragments

The Principal Forms of XQuery

- Path
  - Locates nodes within a tree, and returns a sequence of distinct nodes in document order.
- Sequence
  - An ordered collection of zero or more items, where an item may be an atomic value or a node.
- Arithmetic
  - Arithmetic operators for addition, subtraction, multiplication, division, and modulus.
- Comparison
  - Four kinds of comparisons: value, general, node, and order comparisons.
- Logical
  - A logical expression is either an AND-expression or an OR-expression.
  - The value of a logical expression is always a Boolean value.

The Principal Forms of XQuery

- Constructor
  - Constructors can create XML structures within a query.
  - There are constructors for elements, attributes, CDATA sections, processing instructions, and comments.
- FLWR
  - Expression for iteration and for binding variables to intermediate results.
  - Useful for computing joins between two or more documents and for restructuring data.
  - Pronounced "flower", stands for the keywords FOR, LET, WHERE, and RETURN, the four clauses found in a FLWR expression.

The Principal Forms of XQuery

- Sorting expressions
  - Provides a way to control the order of items in a sequence.
- Conditional expressions
  - Based on the keywords IF, THEN, and ELSE.
- Quantified expressions
  - support existential and universal quantification.
  - The value of a quantified expression is always true or false.

Example XML Document

```
<bib>
  <book year="2000">
        <title>Data on the Web</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <author><last>Suciu</last><first>Dan</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>39.95</price>
 </book>
 <book year=”1995">
        <title>XML</title>
        <author><last>Abiteboul</last><first>Serge</first></author>
        <author><last>Buneman</last><first>Peter</first></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price>45</price>
 </book>
</bib>
```

XQuery Example 1

Find all books with a price of $39.95

XQuery:

```
document("bib.xml")/bib/book[price = 39.95]
```

Result:

```
<book year="2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price> 39.95</price>

</book>
```

XQuery Example 2

Find the title of all books published before 2017

XQuery:

```
document("bib.xml")/bib/book[@year < 1995]/title
```

Result:

```
<title>XML</title>
<title>Data on the Web</title>
```

XQuery Example 3 (For Loop)

List books published by Addison-Wesley after 1991, including their year and title.

XQuery:

```
<bib>
 {
  for $b in document("bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

XQuery Example 3 (For Loop)

- List books published by Addison-Wesley after 1991, including their year and title…

Result:

```
<bib>
 <book year="1995">
     <title>XML</title>
 </book>
 <book year=”2000">
     <title>Data on the Web</title>
 </book>
</bib>
```

XQuery Example 4 (Join)

For each book found at both bn.com and amazon.com, list the title of the book and its price from each source.

XQuery:

```
<books-with-prices>
  {
    for $b in document("bib.xml")//book,
        $a in document("reviews.xml")//entry
    where $b/title = $a/title
    return
        <book-with-prices>
            { $b/title }
            <price-amazon>{ $a/price }</price-amazon>
            <price-bn>{ $b/price }</price-bn>
        </book-with-prices>
  }
</books-with-prices>
```

XQuery Example 4 (Join)

For each book found at both bn.com and amazon.com, list the title of the book and its price from each source.

Result:

```
<books-with-prices>
 <book-with-prices>
       <title>XML</title>
       <price-amazon><price>65.95</price></price-amazon>
       <price-bn><price> 65.95</price></price-bn>
 </book-with-prices>
 <book-with-prices>
       <title>Data on the Web</title>
       <price-amazon><price>34.95</price>
       </price-amazon>
       <price-bn><price> 39.95</price></price-bn>
 </book-with-prices>
</books-with-prices>
```

XQuery Support on RDBMSs

- Oracle XQuery Engine
  - http://www.oracle.com/technology/tech/xml/xquery/index.html
- Introduction to XQuery in SQL Server 2005
  - http://msdn.microsoft.com/en-us/library/ms345122(SQL.90).aspx
- Query DB2 XML data with XQuery
  - http://www.ibm.com/developerworks/data/library/techarticle/dm-0604saracco/
- DataDirect: Data Integration Suite – MySQL Database Support
  - http://www.datadirect.com/products/data-integration/datasources/databases/mysql/index.ssp

JSON & MongoDB

MongoDB Contents

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

Motivations

Challenges of SQL

- Rigid schema
- Not easily scalable (transactions are the road block)
- Requires unintuitive joins

Advantages 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
Shard	Partition

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 [<value>, <value>]}})
```

CRUD: Querying

Including/excluding document fields

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

SELECT field1
FROM <table>;

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 we what we design in mongoDB 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"
  }
)
```

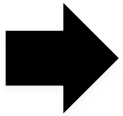
What fields does db.user.find ({”category" : ”vacuum”}) have?

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
}
```



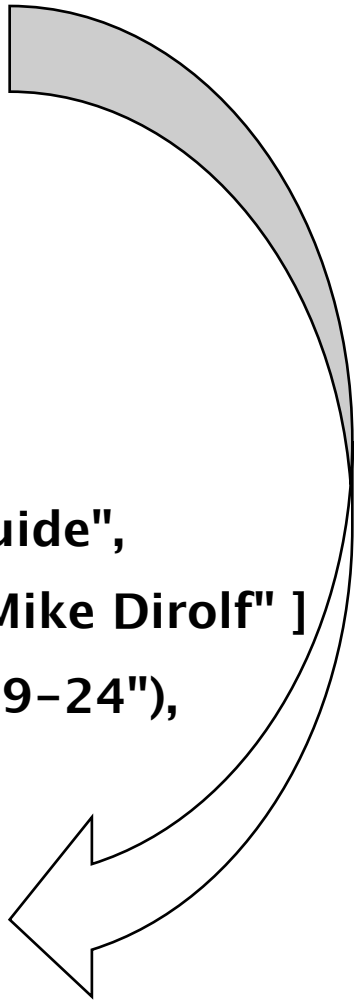
```
zip = {
        _id: 35004 ,
        city: “ACMAR”
        loc: [-86, 33],
        pop: 6065,
        State: “AL”,

        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

Many to many relationship

- Can put relation in either one of the documents (embedding in one of the documents)
- Focus how data is accessed queried

SQL Challenges

- “Primary keys” of a database table are in essence persistent memory addresses for the object. The address may not be the same when the object is reloaded into memory. This is why we need primary keys.
- Foreign key functions just like a pointer in C, persistently point to the primary key.
- Whenever we need to deference a pointer, we do JOIN
- It is not intuitive for programming and

  also JOIN is time consuming

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**

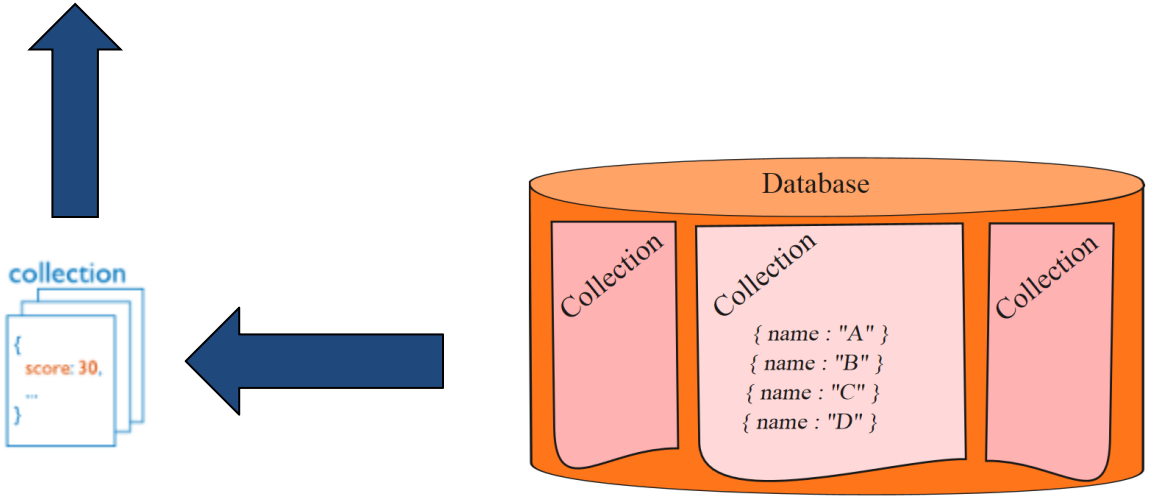
Before Index

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.

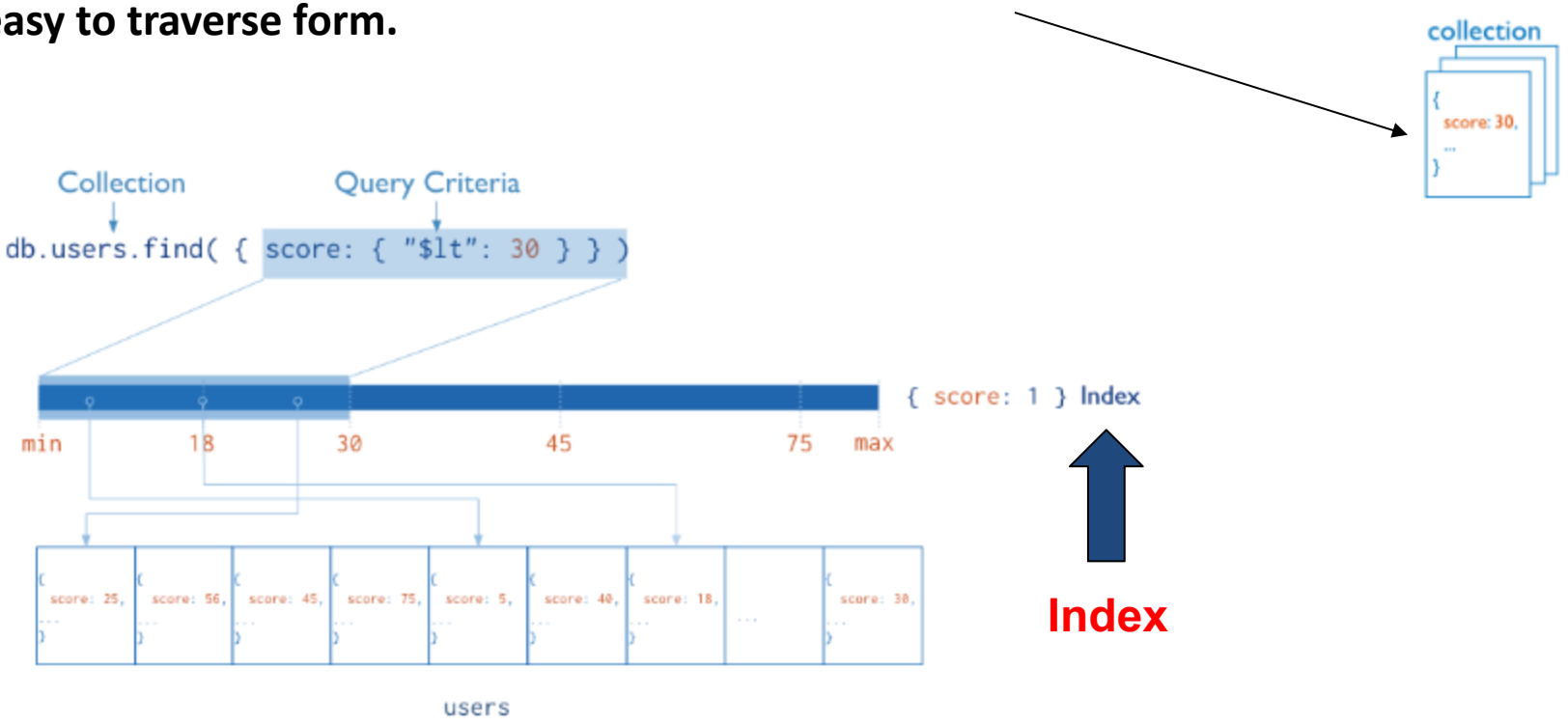


Diagram of a query that uses an index to select

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 })

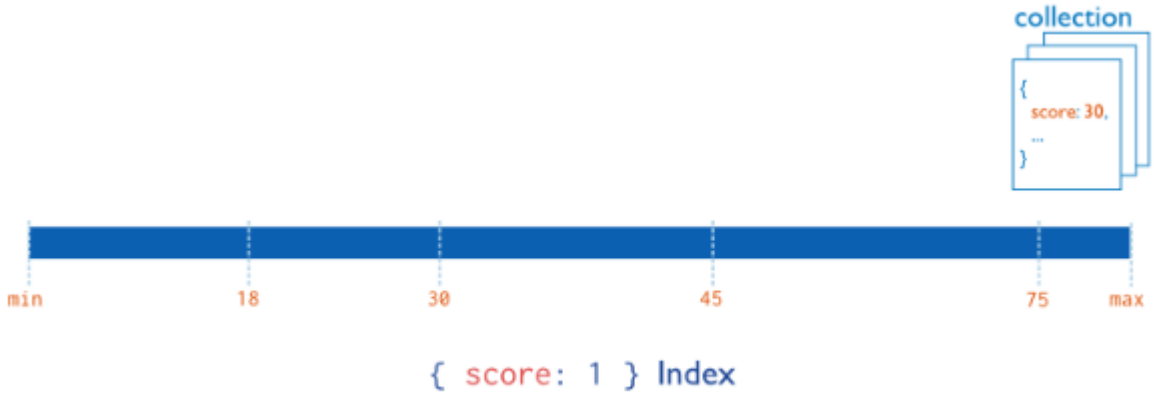


Diagram of an index on the score field (ascending).

Index in MongoDB

Types

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

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

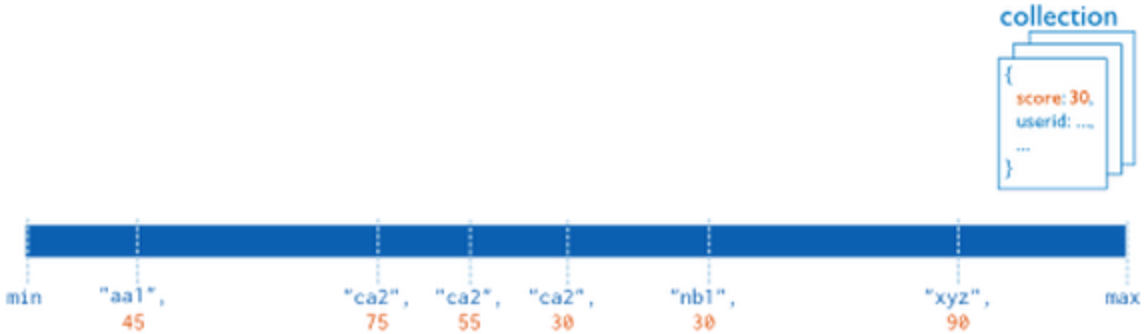


Diagram of a compound index on the `userid` field (ascending) and the `score` field (descending). The index sorts first by the `userid` field and then by the `score` field.

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

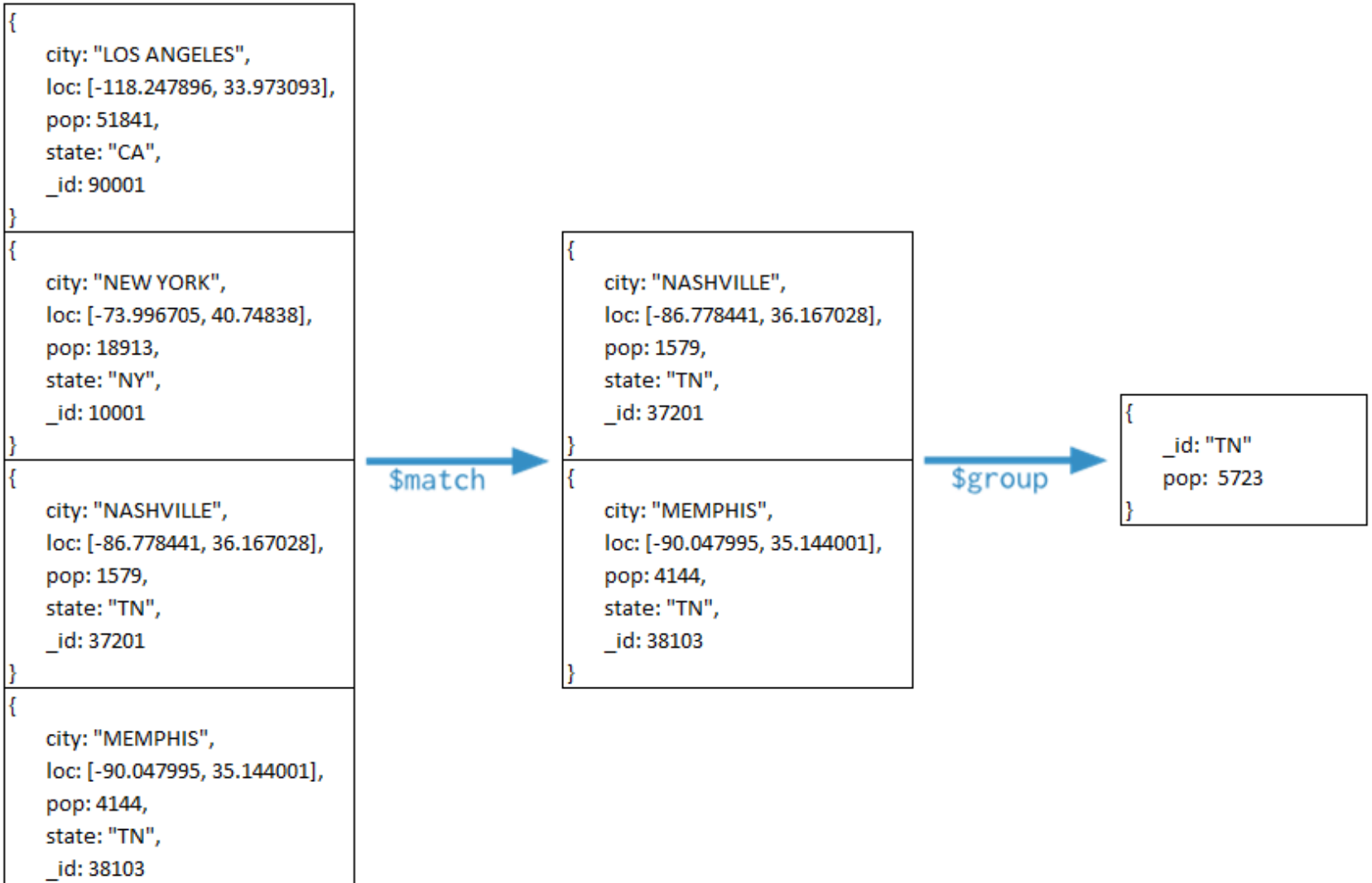
- Operations that process data records and return computed results.
- MongoDB provides aggregation operations
- Running data aggregation on the mongod instance simplifies application code and limits resource requirements.

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.

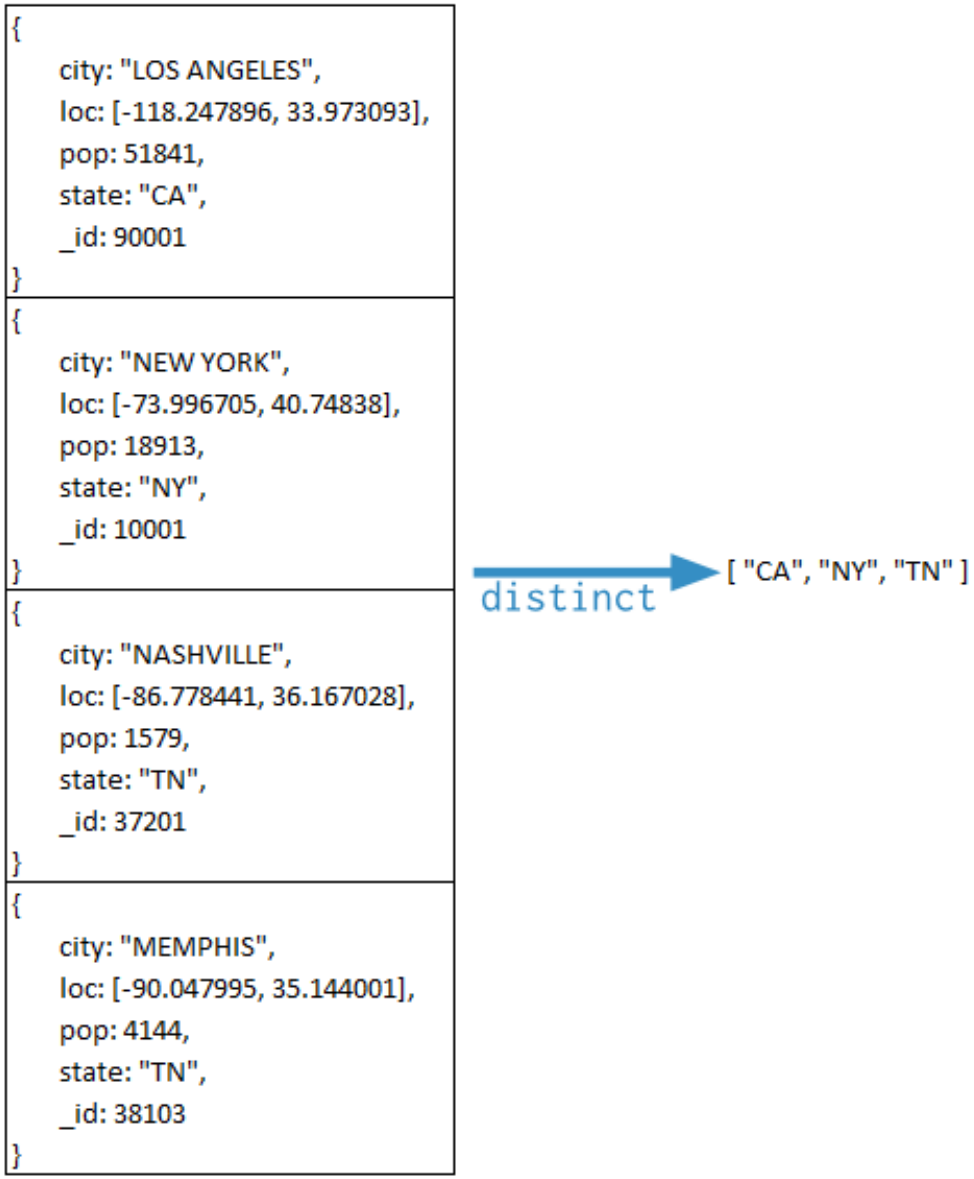
```
db.zips.aggregate(
                    { $match: { state: "TN" } },
                    { $group: {_id: "TN", pop: { $sum: "$pop" } } }
                  );
```



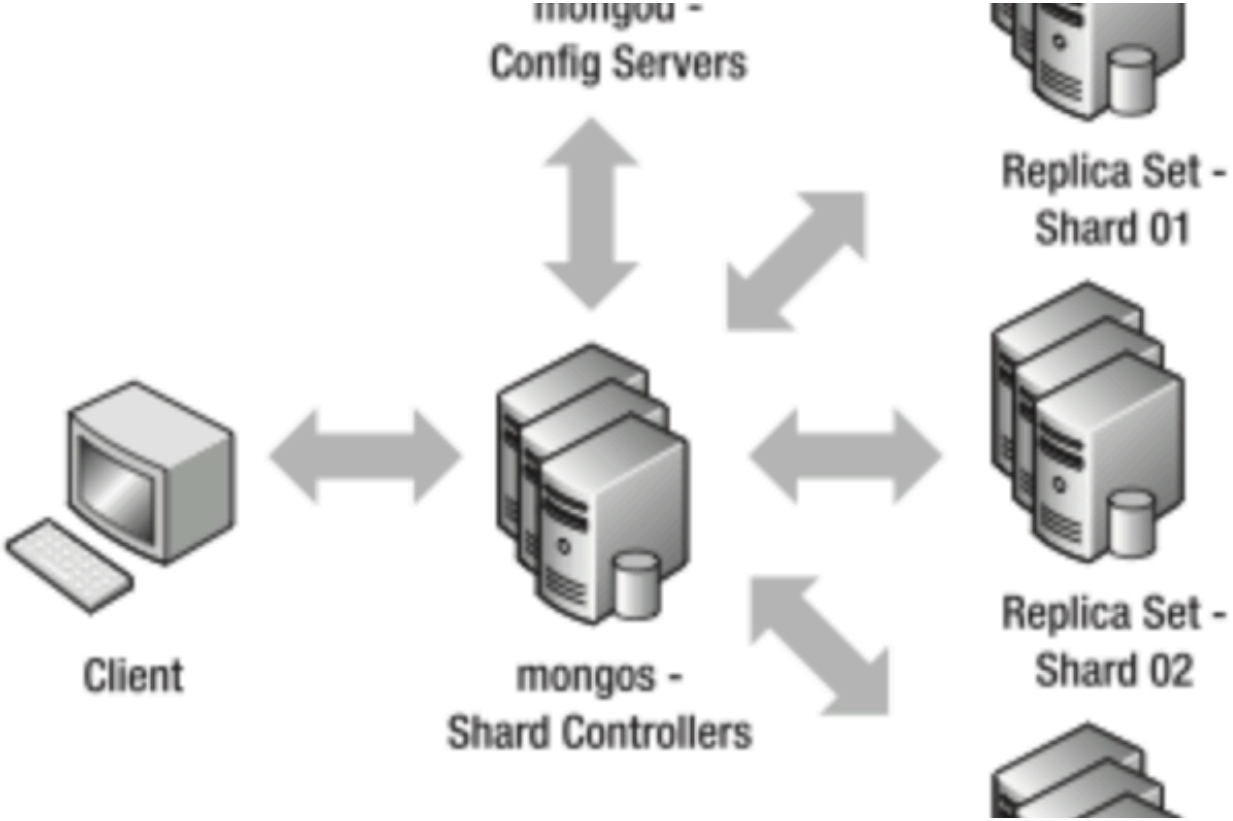
Pipelines

- $limit
- $skip
- $sort

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



Replication & Sharding



Replication

- What is replication?
- Purpose of replication/redundancy
  - Fault tolerance
  - Availability
    - Increase read capacity

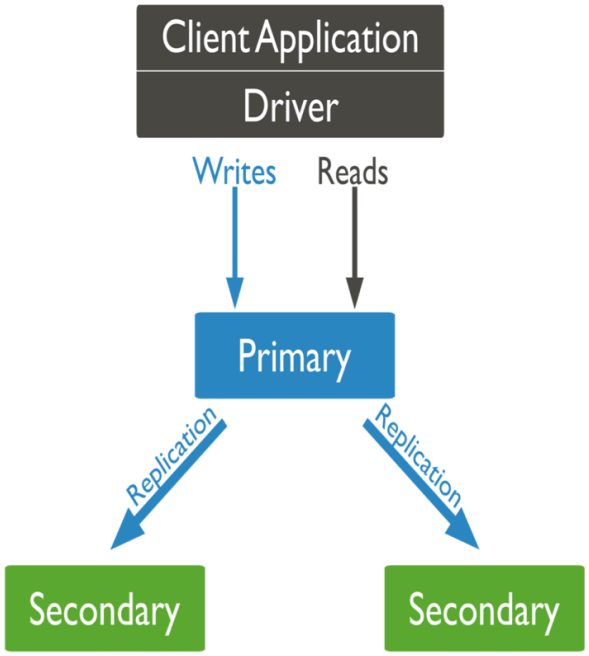


Figure 1: Diagram of default routing of reads and writes to the primary.

Replication in MongoDB

Replica Set Members

- Primary
  - Read, Write operations
- Secondary
  - Asynchronous Replication
  - Can be primary
- Arbiter
  - Voting
  - Can't be primary
- Delayed Secondary
  - Can't be primary

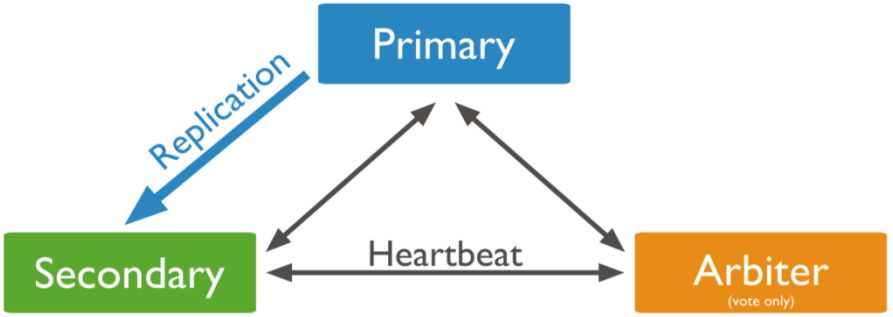


Figure 3: Diagram of a replica set that consists of a primary, a secondary, and an arbiter.

Replication in MongoDB

- Automatic Failover
  - Heartbeats
  - Elections
- The Standard Replica Set Deployment
- Deploy an Odd Number of Members
- Rollback
- Security
  - SSL/TLS

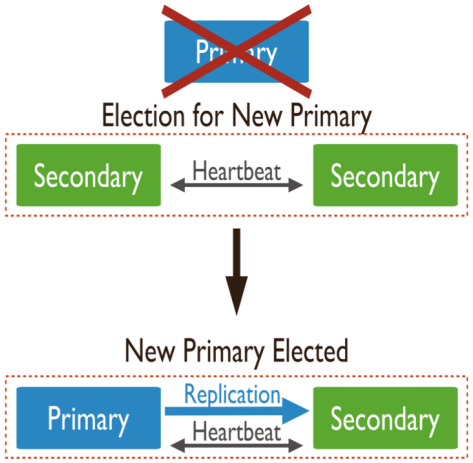


Figure 21: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Sharding

- What is sharding?
- Purpose of sharding
  - Horizontal scaling out
- Query Routers
  - mongos
- Shard keys
  - Range based sharding
  - Cardinality
  - Avoid hotspotting