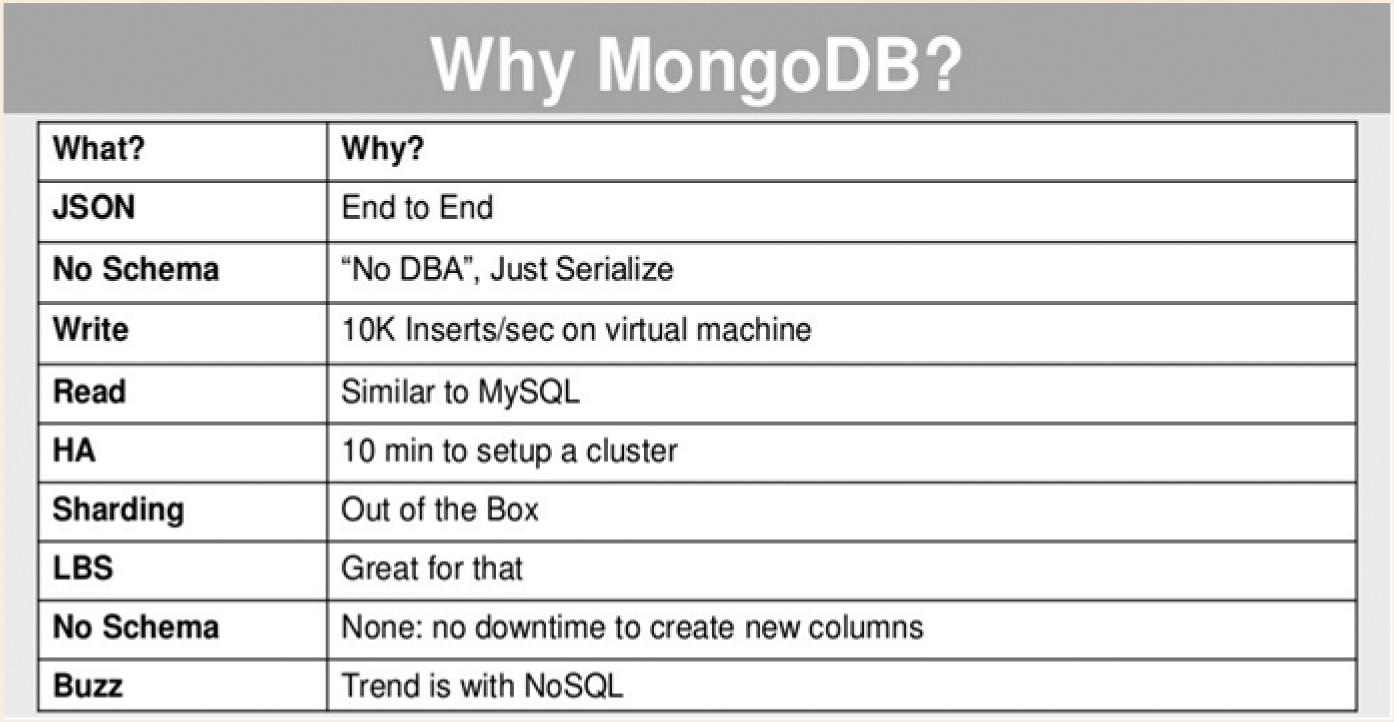
NoSQL databases

Introduction to MongoDB

MongoDB: Introduction

- The leader in the NoSQL Document-based databases
- Full of features, beyond NoSQL
  - High performance
  - High availability
  - Native scalability
  - High flexibility
  - Open source

MongoDB: Why



Why MongoDB?

What?	Why?
JSON	End to End
No Schema	"No DBA", Just Serialize
Write	10K Inserts/sec on virtual machine
Read	Similar to MySQL
HA	10 min to setup a cluster
Sharding	Out of the Box
LBS	Great for that
No Schema	None: no downtime to create new columns
Buzz	Trend is with NoSQL

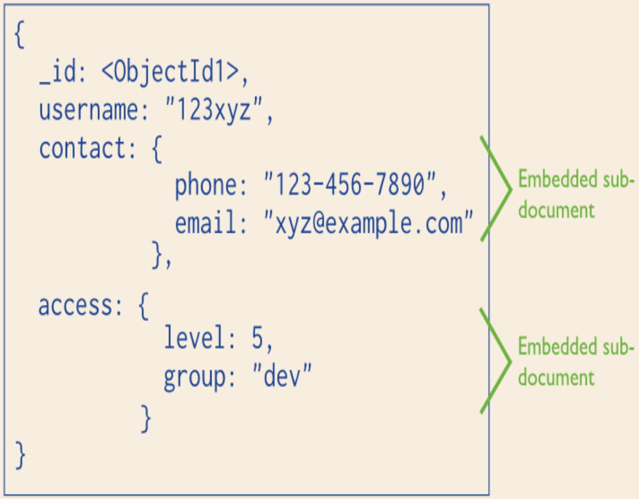
http://blogs.microsoft.co.il/blogs/vprnd
http://top-performance.blogspot.com

Terminology – Concept mapping

Relational database	MongoDB
Table	Collection
Row	Document
Column	Field

MongoDB: Document Data Design

- High-level, business-ready representation of the data
  - Records are stored into Documents
    - field-value pairs
    - similar to JSON objects
    - may be nested



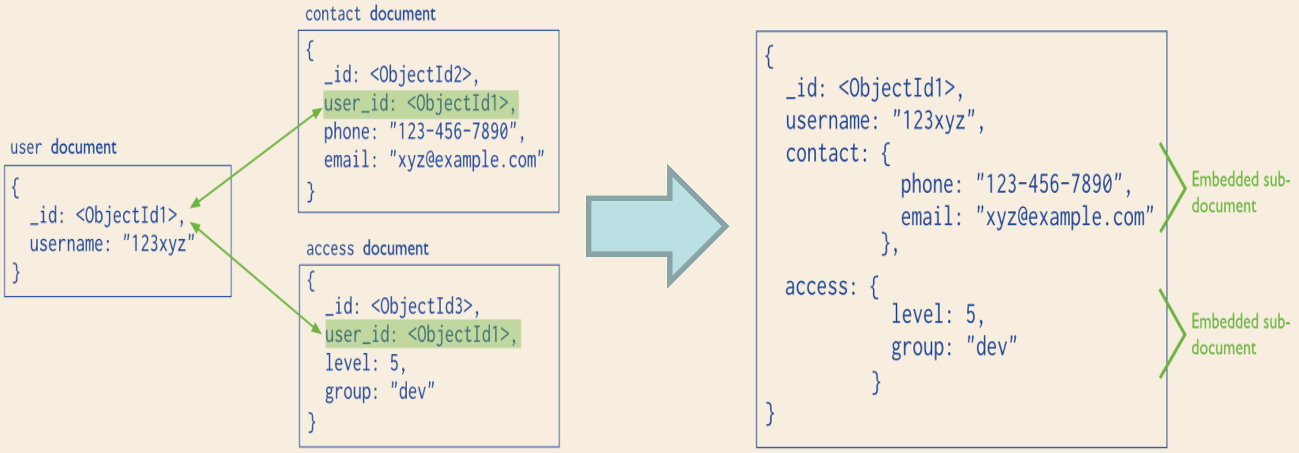
```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
          }
}
```

MongoDB: Document Data Design

- High-level, business-ready representation of the data
- Flexible and rich syntax, adapting to most use cases
- Mapping into developer-language objects
  - year, month, day, timestamp,
  - lists, sub-documents, etc.

MongoDB: Document Data Design

- **But**
  - Relations among documents/records are inefficient, and leads to de-normalization
    - Object(ID) reference, with **no native join**



```
user document
{
  _id: <ObjectId1>,
  username: "123xyz"
}

contact document
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}

access document
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
           },
  access: {
             level: 5,
             group: "dev"
          }
}
```

MongoDB: Main features

- Rich query language
  - Documents are created, read, updated and deleted by means of **CRUD operations**
  - The **SQL language** is **not supported**
  - APIs available for many programming languages
    - JavaScript, PHP, Python, Java, C#, ..

MongoDB: Main features

- **By default**, MongoDB does **not support multi-document transactions**
  - **ACID** properties only at the **document level**
- From **MongoDB 4.0**, **multi-document transactions are supported**
  - However, this feature **impacts on** its **efficiency**

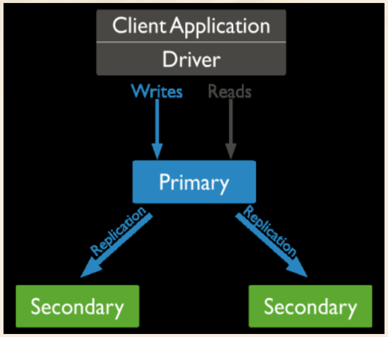
MongoDB: Main features

- **Horizontal scalability** by means of sharding
  - Each shard contains a subset of the documents
  - Pay attention to the **sharding attribute**
    - **It impacts significantly on** the **performance** of your queries
- **Indexes**
  - Support faster queries
  - Single Field, Compound Index, Multikey Index, Geospatial Index, Text Indexes, Hashed Indexes
  - By default, an index is created on the document id

MongoDB: Replication

- A **replica set** is a group of mongoDB instances that maintain the same data set
  - Replica sets = Multiple copies of data
- Replication provides **redundancy** and **increases** data **availability**
  - fault tolerance against the loss of a single server
- Replication **can provide increased read capacity** as we can read from different servers
  - It is **not the default behavior** in MongoDB

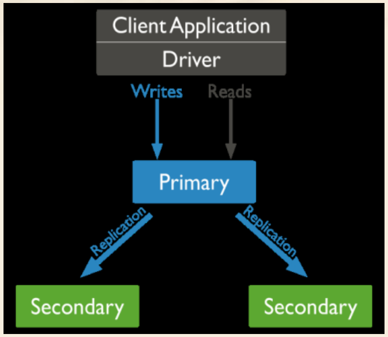
MongoDB: Replication



- Replica set
  - Primary node
    - It receives all write/update operations
  - Secondary nodes
    - They replicate the same operations of the primary node on their data sets (synch operation)
- Asynchronous Replication
- Automatic Failover
  - When the primary becomes unavailable, an eligible secondary will hold an election to elect itself the new primary

MongoDB: Replication

- Read operations
  - All nodes of the replica set can accept read operations
    - But MongoDB replication is based on asynchronous replication → Reads from **secondary nodes may** return data that does **not reflect the state of the data on the primary**
  - By **default**, an application directs its **read operations to the primary** node
    - To avoid inconsistency



Use cases: MongoDB vs Oracle

- The most common use cases for MongoDB include
  - Single View, Internet of Things, Mobile, Real-Time Analytics, Personalization, Geospatial data, Catalog and Content Management
- Oracle would be better suited
  - Applications that require many complex and multi-row transactions (e.g., a double-entry bookkeeping system), tightly coupled systems

From https://www.mongodb.com/compare/mongodb-oracle

Use cases: MongoDB + Oracle

- Booking engine behind a travel reservation system
  - The core booking engine might run on Oracle
  - Those parts of the app that engage with users – serving up content, integrating with social networks, managing sessions – would be better placed in MongoDB

MongoDB

MongoDB: How to insert, update and read data

MongoDB: Databases and Collections

- Each instance of MongoDB can manage multiple databases
- Each database is composed of a set of collections
- Each collection contains a set of documents
  - The documents of each collection represent similar "objects"
    - However, remember that MongoDB is schema-less
    - You are not required to define the schema of the documents a-priori and objects of the same collections can be characterized by different fields

MongoDB: Databases and Collections

- Show the list of available databases
  - show databases;
- Select the database you are interested in
  - use <database name>;
- E.g.,

```
use deliverydb;
```

MongoDB: Databases and Collections

- Create a database
  - Select the database by using use <database name>
  - Create a collection
    - MongoDB will also automatically create the database
- Delete/Drop a database
  - Select the database by using use <database name>
  - Execute the command db.dropDatabase()
- E.g.,

```
use deliverydb;
db.dropDatabase();
```

MongoDB: Databases and Collections

- A collection stores documents, uniquely identified by a document "_id"
- Create collections
  - db.createCollection(<collection name>, <options>);
  - The collection is associated with the current database. Always select the database before creating a collection.
  - Options related to the collection size and indexing
- E.g.,

```
db.createCollection("authors");
```

MongoDB: Databases and Collections

- Show collections

```
show collections;
```

- Drop collections

```
db.<collection name>.drop();
```

- E.g.,

```
db.authors.drop();
```

MongoDB: Read/Insert/Update data

MongoDB	Relational database
db.users.find()	SELECT * FROM users
db.users.insert({ user_id: 'bcd001', age: 45, status: 'A’})	INSERT INTO users (user_id, age, status) VALUES ('bcd001', 45, 'A')
db.users.update({ age: { $gt: 25 } }, { $set: { status: 'C' } }, { multi: true })	UPDATE users SET status = 'C' WHERE age > 25

MongoDB: Insert documents

- Insert a single document in a collection
  - db.<collection name>.insertOne({<set of the field:value pairs of the new document>});
- E.g.,

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } );
```

  - Now people contains a new document representing a user with user_id="abc123", age=55 and status: "A"

MongoDB: Insert documents

- Insert a single document in a collection
  - db.<collection name>.insertOne({<set of the field:value pairs of the new document>});
- E.g.,

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } );
```

Field name

  - Now people contains a new document representing a user with user_id="abc123", age=55 and status: "A"

MongoDB: Insert documents

- Insert a single document in a collection
  - db.<collection name>.insertOne({<set of the field:value pairs of the new document>});
- E.g.,

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } );
```

Field value

  - Now people contains a new document representing a user with user_id="abc123", age=55 and status: "A"

MongoDB: Insert documents

- E.g.,

```
db.people.insertOne( {
    user_id: "abc124",
    age: 45,
    favorite_colors: ["blue", "green"]
 } );
```

Favorite_colors is an array

- Now people contains a new document representing a user with user_id="abc124", age=45 and an array favorite_colors containing the values "blue" and "green"

MongoDB: Insert documents

- E.g.,

```
db.people.insertOne( {
    user_id: "abc124",
    age: 45,
    address: { street: "my street",
               city: "my city"}
 } );
```

Nested document

- Example of a document containing a nested document

MongoDB: Insert documents

- Insert many documents with one single command
  - db.<collection name>.insertMany([<comma separated list of documents>]);
- E.g.,

```
db.people.insertMany([
{user_id: "abc123", age: 55, status: "A"},
{user_id: "abc124",         age: 45, favorite_colors:
["blue", "green"]}
] );
```

MongoDB: Document update

- Documents can be updated by using
  - db.collection.updateOne(<filter>, <update>, <options>)
  - db.collection.updateMany(<filter>, <update>, <options>)
  - filter = filter condition. It specifies which documents must be updated
  - update = specifies which fields must be updated and their new values
  - options = specific update options

MongoDB: Document update

- E.g.,

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

  - This operation updates all documents with qty<50
  - It sets the value of the size.uom field to "in‟, the value of the status field to "P", and the value of the lastModified field to the current date.

MongoDB: Read data from documents

- Select documents
  - db.<collection name>.find({<conditions>}, {<fields of interest>});
- E.g.,

```
db.people.find();
```

  - Returns all documents contained in the people collection

MongoDB: Read data from documents

- Select documents
  - db.<collection name>.find({<conditions>}, {<fields of interest>});
- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - <conditions> are optional
    - conditions take a document with the form: {field1 : <value>, field2 : <value> ... }
    - Conditions may specify a value or a regular expression

MongoDB: Read data from documents

- Select documents
  - db.<collection name>.find({<conditions>}, {<fields of interest>});
- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - <fields of interest> are optional
    - projections take a document with the form: {field1 : <value>, field2 : <value> ... }
    - 1/true to include the field, 0/false to exclude the field

MongoDB: Read data from documents

- E.g.,

```
db.people.find().pretty();
```

  - No conditions and no fields of interest
    - Returns all documents contained in the people collection
    - pretty() displays the results in an easy-to-read format

```
db.people.find({age:55})
```

  - One condition on the value of age
    - Returns all documents having *age* equal to 55

MongoDB: Read data from documents

```
db.people.find({ }, { user_id: 1, status: 1 })
```

- No conditions, but returns a specific set of fields of interest
  - Returns only *user_id* and *status* of all documents contained in the people collection
  - Default of fields is false, except for _id

```
db.people.find({ status: "A",  age: 55})
```

- Status = "A" and age = 55
  - Returns all documents having *status*="A" and *age*=55

MongoDB: Read data from documents

db.people.find({ age: { $gt: 25, $lte: 50 } })

- Age greater than 25 and less than or equal to 50
  - Returns all documents having *age*>25 and *age*<=50

db.people.find({ $or: [{ status: "A"}, {age: 55}] })

- Status = “A” or age = 55
  - Returns all documents having status=“A” or age=55

db.people.find({ status: {$in:[“A”, “B”]}})

- Status = “A” or status = B
  - Returns all documents where the status field value is either “A” or “B”

MongoDB: Read data from documents

- Select a single document
  - db.<collection name>.findOne({<conditions>}, {<fields of interest>});
- Select one document that satisfies the specified query criteria.
  - If multiple documents satisfy the query, it returns the first one according to the natural order which reflects the order of documents on the disk.

MongoDB: Read data from documents

- There are other operators for selecting data from MongoDB collections
- However, there is not a join operator
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using the loop statement provided by the programming language you are using, and executes one query for each of them to retrieve the corresponding document(s) in the other collection

MongoDB: Comparison query operators

Name	Description
$eq or :	Matches values that are equal to a specified value
$gt	Matches values that are greater than a specified value
$gte	Matches values that are greater than or equal to a specified value
$in	Matches any of the values specified in an array
$lt	Matches values that are less than a specified value
$lte	Matches values that are less than or equal to a specified value
$ne	Matches all values that are not equal to a specified value
$nin	Matches none of the values specified in an array

MongoDB: Cursor

- db.collection.find() gives back a cursor. It can be used to iterate over the result or as input for next operations.
- E.g.,
  - cursor.sort()
  - cursor.count()
  - cursor.forEach()
  - cursor.limit()
  - cursor.max()
  - cursor.min()
  - cursor.pretty()

MongoDB: Cursor

- Cursor examples:

```
db.people.find({ status: "A"}).count()
```

- Select documents with status="A" and count them.

```
db.people.find({ status: "A"}).forEach(
    function(myDoc) { print( "user: "+myDoc.name );
    })
```

- forEach applies a JavaScript function to apply to each document from the cursor.
    - Select documents with status="A" and print the document name.

MongoDB: Sorting data

- Sort is a cursor method
- Sort documents
  - sort({<list of field:value pairs>});
  - field specifies which filed is used to sort the returned documents
  - value = -1 descending order
  - Value = 1 ascending order
- Multiple field: value pairs can be specified
  - Documents are sort based on the first field
  - In case of ties, the second specified field is considered

MongoDB: Sorting data

- E.g.,

  ```
  db.people.find({ status: "A"}).sort({age:1})
  ```

  - Select documents with status="A" and sort them in ascending order based on the age value
    - Returns all documents having status="A". The result is sorted in ascending age order

MongoDB

MongoDB: How to aggregate data

MongoDB: Aggregation

- Aggregate functions can be applied to collections to group documents
  - db.collection.aggregate({<set of stages>})
  - Common stages: $match, $group ..
  - The aggregate function allows applying aggregating functions (e.g. sum, average, ..)
  - It can be combined with an initial definition of groups based on the grouping fields

MongoDB: Aggregation

- E.g.,

```
db.people.aggregate( [
  { $group: { _id: null,
              count: { $sum: 1 }
            }
  }
] )
```

  - Counts the number of documents in people
    - Returns an integer value that is equal to the number of documents
    - The solution counts by summing a set of ones (one for each document)

MongoDB: Aggregation

```
db.people.aggregate( [
  { $group: { _id: null,
                    total: { $sum: "$age" }
                 }
  }
] )
```

- Considers all documents of people and sum the values of their age
- The returned value is associated with a field called total

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: null,
                     average: { $avg: "$age" },
                     total: { $sum: "$age" }
                  }
   }
] )
```

- Considers all documents of people and computes
  - sum of age
  - average of age

MongoDB: Aggregation

```
db.people.aggregate( [
   { $match: {status:"A"} } ,
   { $group: { _id: null,
                    count: { $sum: 1 }
                 }
   }
] )
```

- Counts the number of documents in people with status equal to “A”

MongoDB: Aggregation

```
db.people.aggregate( [
   { $match: {status:"A"} } ,
   { $group: { _id: null,
                  count: { $sum: 1 }
               }
   }
] )
```

Where conditions

- Counts the number of documents in people with status equal to “A”

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: "$status",
                     count: { $sum: 1 }
                  }
   }
] )
```

- Creates one group of documents for each value of status and counts the number of documents per group
  - Returns one value for each group containing the value of the grouping field and an integer representing the number of documents

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: "$status",
                     count: { $sum: 1 }
                  }
   }
] )
```

Aggregation field

- Creates one group of documents for each value of status and counts the number of documents per group
  - Returns one value for each group containing the value of the grouping field and an integer representing the number of documents

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: "$status",
                    count: { $sum: 1 }
                  }
   },
   { $match: { count: { $gte: 3 } } }
] )
```

- Creates one group of documents for each value of status and counts the number of documents per group. Returns only the groups with at least 3 documents

MongoDB: Aggregation

```
db.people.aggregate( [
  { $group: { _id: "$status",
                count: { $sum: 1 }
              }
  },
  { $match: { count: { $gte: 3 } } }
] )
```

Having condition

- Creates one group of documents for each value of status and counts the number of documents per group. Returns only the groups with at least 3 documents

MongoDB

MongoDB: Map-Reduce operations

MongoDB: Map-Reduce

- MongoDB supports the Map-Reduce paradigm to process and reduce data into aggregated results.
  - MapReduce uses custom JavaScript function to perform the map and the reduce operations
  - db.collection.mapReduce({<map Func>, <Reduce funct>, <finalize>, <query>, <out>, <sort>, <limit>, ..})

MongoDB: Map-Reduce

- **Map** requires *emit(key, value)* to map each value with a key. It refers to the current document as *this*
- **Reduce** groups all document with the same key. These functions must be associative and commutative and must return an object of the same type of value emitted by *Map*
- **Out** specifies where to output the map-reduce query results (either a collection or a inline result)

MongoDB: Map-Reduce

- **Finalize** (optional) Follows the *reduce* method and modifies the output
- **Query** (optional) specifies the selection criteria for selecting the input documents to the *map* function
- **Sort** (optional) specifies the sort criteria for the input documents
- **Limit** (optional) specifies the maximum number of input documents

MongoDB: Map-Reduce

- E.g.,

```
db.orders.mapReduce(
    function() {emit(this.cust_id, this.amount);},
    function(key, values) {return Array.sum(values)};
    {
      query: {status: “A”},
      out: “order_totals”
    }
)
```

MongoDB: Map-Reduce

```
db.orders.mapReduce(
    function() {emit(this.cust_id, this.amount);},
    function(key, values) {return Array.sum(values)};
   {
      query: {status: "A"},
      out: "order_totals"
    }
)
```

Map function

Reduce function

- Only for orders with status: "A" and for each cust_id, sum all the orders values into a the "order_totals" collection

MongoDB

MongoDB: Indexing

MongoDB: Indexes

- Indexes are data structures that store a small portion of the collection’s data set in a form easy to traverse.
- They store ordered values of a specific field, or set of fields, in order to efficiently support equality matches, range-based queries and sorting operations.

MongoDB: Indexes

- MongoDB provides different data-type indexes
  - Single field indexes
  - Compound field indexes
  - Multikey indexes
  - Geospatial indexes
  - Text indexes
  - Hashed indexes

MongoDB: Create new indexes

- From MongoDB v. 3.0
  - db.collection.createIndex(<index keys>, <options>)
  - Before v. 3.0 use db.collection.ensureIndex()
  - Options include: *name*, *unique* (whether to accept or not insertion of documents with duplicate index keys), *background*, *dropDups*, ..

MongoDB: Indexes

- Single field indexes
  - Support user-defined ascending/descending indexes on a single field of a document
- E.g.,
  - db.orders.createIndex({orderDate: 1})
- Compound field indexes
  - Support user-defined indexes on a set of fields
- E.g.,
  - db.orders.createIndex({orderDate: 1,
    zipcode: -1})

MongoDB: Indexes

- Geospatial support
  - MongoDB supports efficient queries of geospatial data
  - Geospatial data are stored as:
    - GeoJSON objects: embedded document { <type>, <coordinate> }
      - E.g., location: {type: "Point", coordinates: [-73.856, 40.848]}
    - Legacy coordinate pairs: array or embedded document
      - point: [-73.856, 40.848]

MongoDB: Indexes

- Geospatial indexes
  - Two type of geospatial indexes are provided:
    - 2d: use and return planar geometry
    - 2dsphere: use and return spherical geometry
- E.g.,
  - db.places.createIndex({location: “2dsphere”})
- Geospatial query operators
  - $geoIntersects, $geoWithin, $near, $nearSphere
- Geospatial aggregation stage
  - $geoNear
    - { $geoNear: { <geoNear options> } }

MongoDB: Indexes

- E.g.,

  -
    ```
    db.places.find({location:
      {$near:
         {$geometry: {type: "Point",
                      coordinates: [ -73.96, 40.78 ] },
          $maxDistance: 5000}
      }})
    ```
  - Find all the places within 5000 meters from the specified GeoJSON point, sorted in order from nearest to furthest

MongoDB: Indexes

- Text indexes
  - Support efficient searching for string content in a collection
  - Text indexes store only *root words* (no language-specific *stop words* or *stem*)
- E.g.,
  - db.reviews.createIndex({comment: “text”})
  - Wildcard ($**) allows MongoDB to index every field that contains string data
  - E.g., db.reviews.createIndex({“$**”: “text”})