CS 5/7330

NoSQL : Document Databases / MongoDB

Motivation: Semi-structured data

- Relational model is very rigid in terms of structure
- For example, consider this table:

SSN	Name	Phone-#
123456789	John Doe	123-4567
234567890	Mary Doe	456-8901

- But what if someone have more/less than one phone?

Motivation: Semi-structured data

- No phone

SSN	Name	Phone-#
123456789	Brad Doe	NULL
234567890	Mary Doe	456-8901

- What can NULL means?
  - Brad has no phone
  - Brad has a phone but I don’t know his number
  - Possible ambiguity

Motivation: Semi-structured data

- Many phones

SSN	Name	Phone-#
123456789	Tai-Kwan Doe	555-5555
123456789	Tai-Kwan Doe	689-0777

- Cause duplication
- Anomaly and/or inefficiency

Motivation: Semi-structured Data

- Many problems can be resolved by normalization (breaking into multiple tables)
- But queries will require potentially many joins
    - Can be inefficient
- Solution:
    - Semi-structured data

Semi-structured Data

- Some data have a bit of a structure
- E.g. item have a set of attributes
- However they can still differs
  - some attributes may be present in some items but not all of them
  - Some attributes for an item may have multiple values
  - … or even multiple possible types
- So new models are proposed. Examples:
  - XML
  - JSON

JSON

- JavaScript Object Notation
- Originally decided as a way for information interchange
  - Text-based
  - Self describing
    - The “schema” is described within the object
- While the schema for an object is flexible, the way of describing the schema is fixed
  - Thus one can write standard tool(s) to interpret any JSON object, without knowing its schema beforehand

JSON

- Each JSON object is represented as a set of name/value pairs
  - {"name":"John", "age":30, "car":null}
- Name is a string in quotes (“name”, “age”, “car”)
- Each name has a value associated with it.
- Valid value types:
  - String
  - Number
  - Array
  - Boolean
  - Null
  - Another JSON object

JSON (Example of complex case)

https://www.sitepoint.com/twitter-json-example/

General idea of document databases

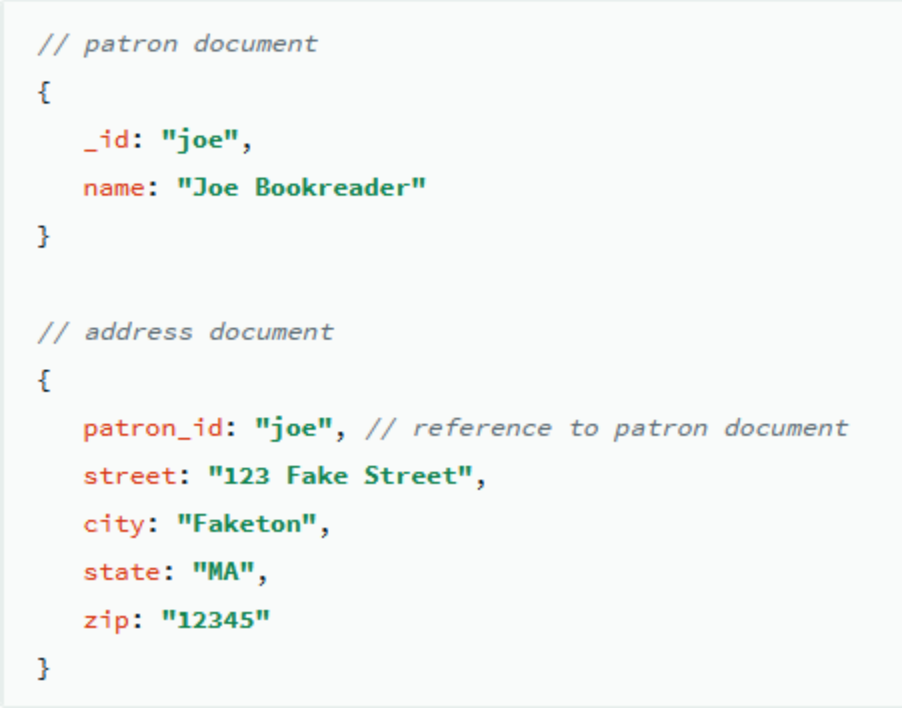
- Database store objects
  - Objects are known as documents
  - Name/value pairs are usually termed as key/value pairs
- Objects have unique IDs
  - Typically as Entity in E-R Model
- Relations between objects can be enforced by
  - Nested Objects/Documents (embedding)
  - Key/value pairs where values are IDs of documents to be related (references)

General idea of document databases

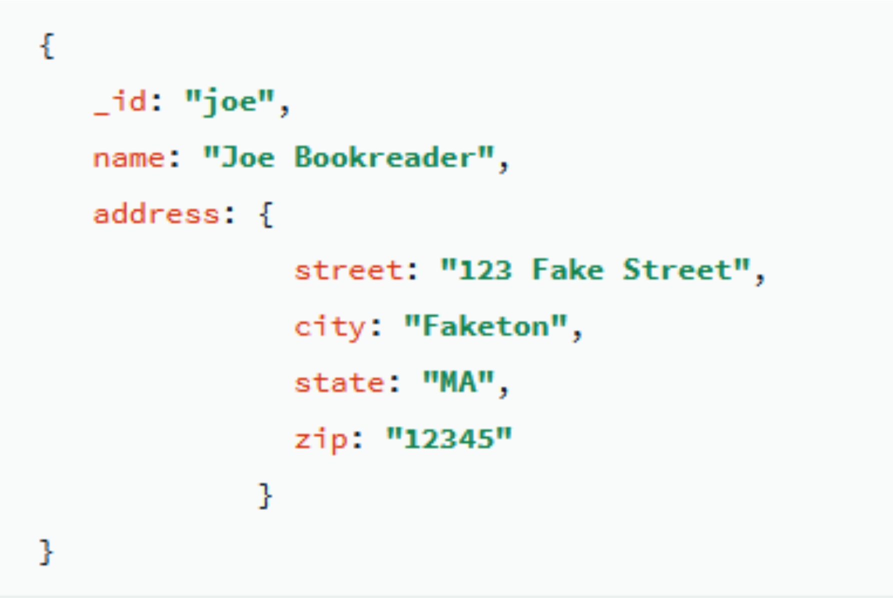
- Other properties
  - Data are often grouped into buckets (like tables)
  - Provide indexes for document attributes
  - Provide query language support to retrieve (part of) a document

Modeling using Document Databases

- One-to-one relationship
- Either embedding or relationship will work
- Example : (assume patron and address is a 1-1 relationship)



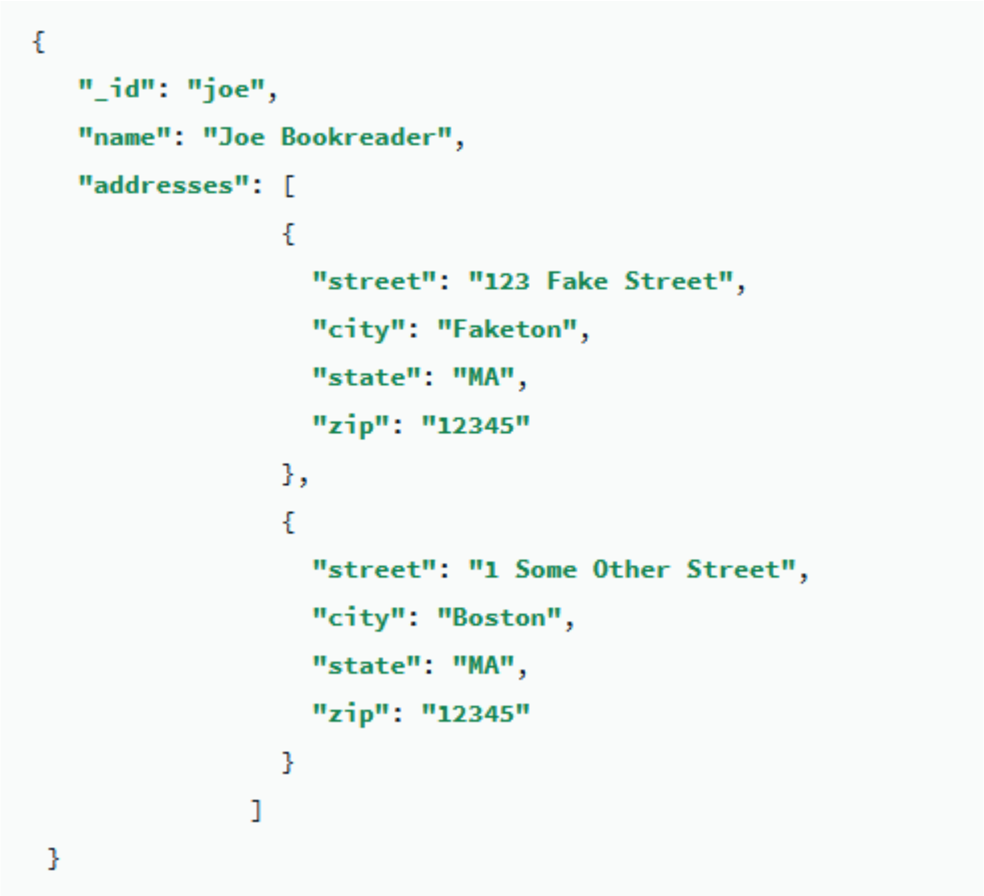
```
// patron document
{
   _id: "joe",
   name: "Joe Bookreader"
}

// address document
{
   patron_id: "joe", // reference to patron document
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}
```



```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA",
              zip: "12345"
            }
}
```

Modeling using Document Databases

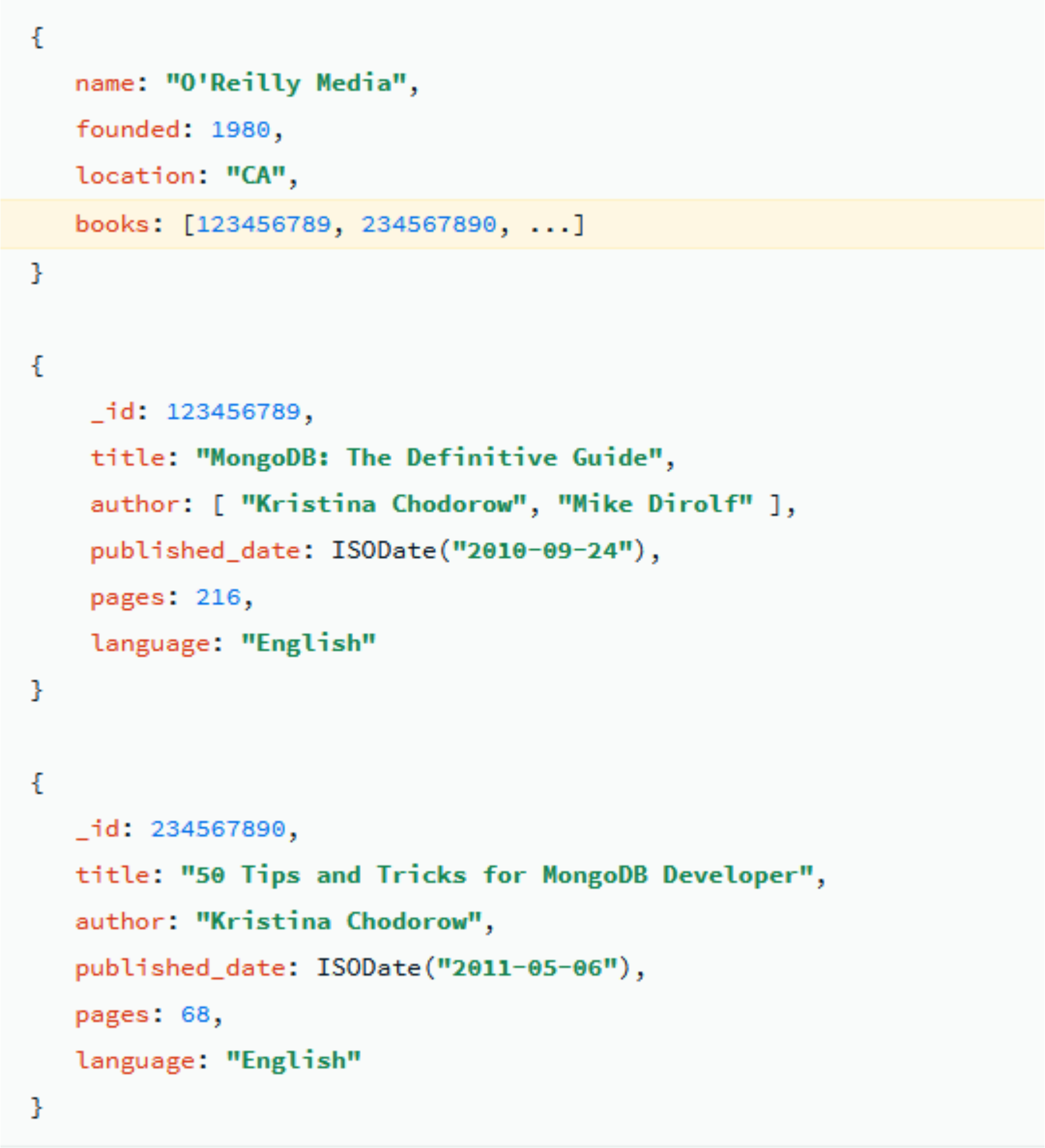
- One-to-many relationship
- Example : (assume patron and address is a 1-m relationship)
- Embedding



```
{
   "_id": "joe",
   "name": "Joe Bookreader",
   "addresses": [
                {
                  "street": "123 Fake Street",
                  "city": "Faketon",
                  "state": "MA",
                  "zip": "12345"
                },
                {
                  "street": "1 Some Other Street",
                  "city": "Boston",
                  "state": "MA",
                  "zip": "12345"
                }
              ]
 }
```

Modeling using Document Databases

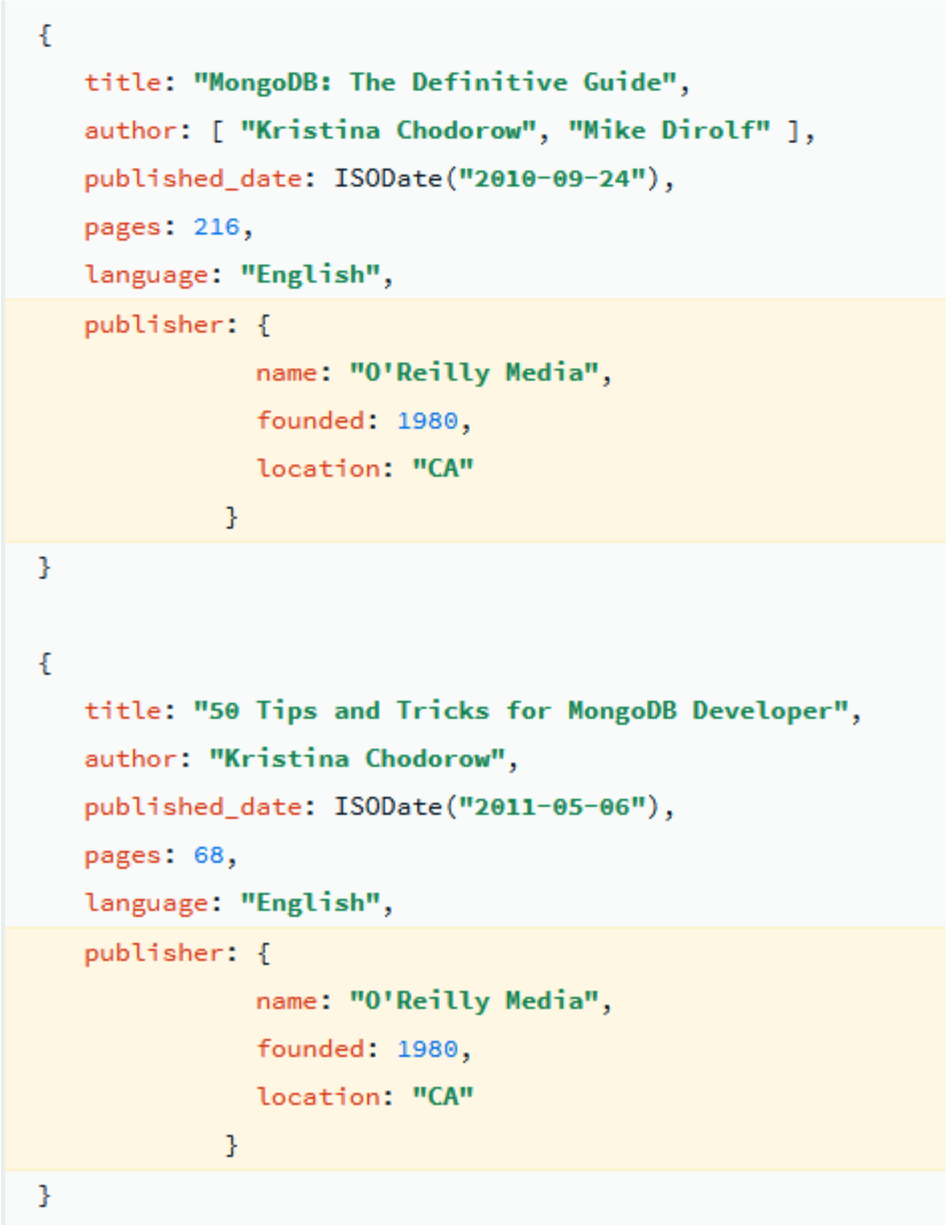
- One-to-many relationship
- Example : (assume publisher and books is a 1-m relationship)
- relationship



```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [123456789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

Modeling using Document Databases

- What you should AVOID
- Duplication of data
- Same problem as in relational databases



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

{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

Modeling using Document Databases

- One thing you should never do
- Whenever there is a relationship between two objects, using relationship should only be from one object to the other, BUT NOT BOTH WAYS
  - Why not?
- Which way to go is database and application dependent

```
{ _id = “_1”;
   name = “john doe”
   address = “_add1”
}
{
   _id = “_add1”
   address = “SMU Blvd”
   patron_id = “1”
}
```

Modeling using Document Databases

- Document models tends to works well with binary relationship.
- For relationship with multiple entity (e.g: supplier-restaurant-food), it may need either:
    - Deep embedding
    - Creating separate objects to store the relationships (in terms of E-R modelling, treating relationship as entity)

Example: MongoDB

- Document-based database system
- Basic unique of storage: Document
  - Very similar to JSON
  - With some additional data type support (e.g. date, binary data, code)
  - Each document need to have an "_id" field (can be auto-generated)
  - Each document have a size limit : 16 MB (in Mongo DB 4.0)
- Collections: A group of Documents
  - No requirement that each document have the same schema
  - But there are advantages (in terms of implementation, and potential efficiency)
  - One can name sub-collections of a collection
- Database: A group of collections
  - Typically an application will use one database

MongoDB: insert

- Insertions:
  - db.collection.insertOne({…})
  - db.collection.insertMany([{…}, {…}, …, {…}])
- Notice that the only automatic check is unique _id
  - NO notion of referential integrity
  - NO checking of name of keys (since we do not assume any scheme [even within a collection])
  - It’s the (virtually all) developer’s responsibility to ensure everything works.

MongoDB: Query -- selection

- Basic method: find()
- db.collection.find({<selection>}. {<projection>})
  - Describe the documents to be retrieved
  - Each of this has the same format as a document (but without the _id).
- db.collection.find({})
  - Return all documents in the collection
- db.collection.find({“name” : “john doe”})
  - Return all documents that have key value pair (name : john doe)
- db.collection.find({“name” : “john doe”, “age” : 30})
  - Return all documents that have BOTH key value pairs
  - Logical AND
- Note:
  - Value in the key/value pair in find must be a constant.
  - Need other constructs to relate different documents from the same/different collections

MongoDB: Query -- projection

- db.collection.find({“name” : “john doe”}, {“name” : 1, “age”: 1})
  - Return all documents that have key value pair (name : john doe), and output the name and the age
- db.collection.find({“name” : “john doe”}, {“name” : 0, “age”: 1})
  - Return all documents that have key value pair (name : john doe), and output age, but not the name
- Note:
  - _id : is output by default
  - Can suppress it by including {“_id” : 0} in the projection clause

MongoDB: Query – selection clauses

- db.collection.find({“name” : “john doe”, “age” : {“$gte”: 18}})
  - Return all documents that have name john doe, and is at least 18 years old (gte = greater than or equal to)
  - Clause: a document, the key is a reserved word starting with “$”, to denote special condition or functions,
- db.collection.find({“name” : “john doe”, “age” : {“$gte”: 18, “$lte” : 22 } })
  - Return all documents that have name john doe, and is between 18 and 22 (lte = less than or equal to)

MongoDB: Query – OR / NOT

- db.collection.find({“dept” : “CS”, {$or, [{“age” : {“$gte”: 18}}, {“gpa” : 4.0}] } })
  - Return all documents that have dept = “CS” and either age is >= 18, or gpa = 4.0
  - Notice that the term following or is an list (array) of conditions.
- db.collection.find({“dept” : “CS”, {$not, {“gpa” : 4.0} } })
  - Return all documents that have dept = “CS” and either age is >= 18, or gpa ≠ 4.0

MongoDB: Query – NULL values

- BE VERY CAREFUL!
- Suppose your collection has the following documents:
  - { “_id” : “1”, “name” : “john doe”, “phone” : null}
  - { “_id” : “2”, “name” : “jack doe”, “age” : 28}
- db.collection.find({“phone” : null}) will return BOTH documents
  - Null matches either “having the value null”, and “no such key exists in the document”
  - Use the “$exists” clause to compensate

MongoDB: Query – Arrays

  - { “_id” : “1”, “name” : “john doe”,   “x” : [1, 3, 5, 11]}
- The following queries will return this document
  - db.collection.find({“x” : 5})
  - db.collection.find({“x” : {“$gte”, 4, “$lte”, 11}})
  - db.collection.find({“x” : {“$gte”, 7, “$lte”, 9}})
    - Why?
      Notice that matching an array means that every clause has to be matched, but each clause can be matched by a different element in the array
    - Use $elemMatch to overcome this

MongoDB: Query – Embedded documents

{ “_id” : “1”, “name” : “john doe”,
“course” : {“cid” : “CS 5330”, “grade”: “A”} }

- db.collection.find({“course” : {“cid” : “CS 5330”, “grade”: “A”} })
  - will return the document
- db.collection.find({“course” : {“cid” : “CS 5330”} })
  - Will NOT
  - For embedding documents, require perfect match (“_id” can be left out)
- db.collection.find({“course.cid” : “CS 5330”} })
  - Will return the document again
  - Use dot notation

MongoDB: Query – More complicated issues

(See MongoDB manual / books for details)

- Matching array of embedded documents can be very tricky
- No inherit way to “join” multiple collections using the query language
  - Need to write program and using API
  - There is an option of a “$WHERE” clause where you can embed code inside a query statement

Mongo DB : Updates and Delete

- db.collection.updateOne(), updateAll(), replaceOne(), replaceAll()
  - Two parameters
    - First one is the query condition to specify which document(s) in the collection is to be updated.
    - Second one specify how the document is to be updated / or what to replace the document with
    - For replacement, the “_id” will not change, but everything else will be replaced.
    - Also an “upsert” option: try to update, if the document to be updated is not found, then insert a new document.

Other aspects

- Indexing – per collection basis, similar to relational database indices
  - Specific type of indices for specific data types (e.g. text, geospital)
- Aggregation – group by, and computation based on groups
- Replication – allow for duplication of data