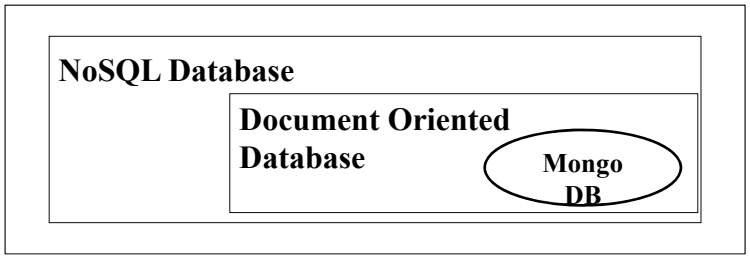
MongoDB

Overview – What is MongoDB?

[mongo.whatis]

What is MongoDB?

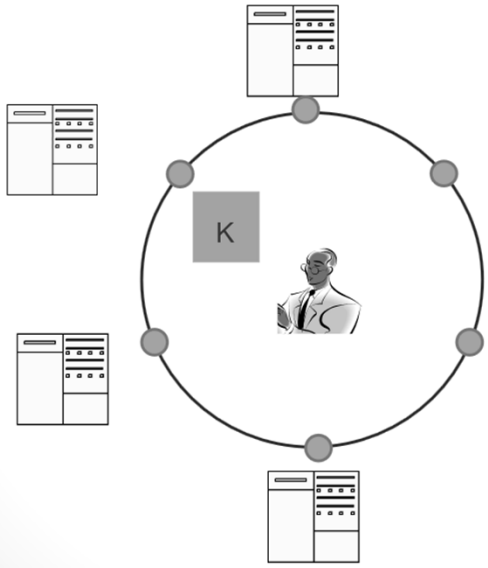
- **Mongo – humongous database**
- **Definition:** MongoDB is an open-source document oriented database that provides high performance, high availability, and automatic scaling.
- Instead of storing data in tables and rows as with a relational database, in MongoDB stores data in JSON-like documents with dynamic schemas(schema-free).



Taxonomy of NoSQL

- **Key-value** redis riak
- **Graph database** Neo4j the graph database HyperGraphDB
- **Document-oriented** mongoDB CouchDB relax
- **Column family** Cassandra HBASE

Typical NoSQL architecture



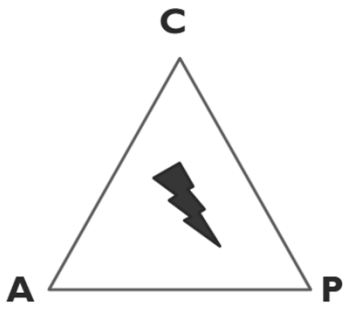
Hashing function maps each key to a server (node)

Theory of NOSQL: CAP

GIVEN:

- Many nodes
- Nodes contain ***replicas of partitions*** of the data

- **C**onsistency
  - All replicas contain the same version of data
  - Client always has the same view of the data (no matter what node)
- **A**vailability
  - System remains operational on failing nodes
  - All clients can always read and write
- **P**artition tolerance
  - multiple entry points
  - System remains operational on system split (communication malfunction)
  - System works well across physical network partitions



CAP Theorem: satisfying all three at the same time is impossible

Sharding of data

- Distributes a single logical database system across a cluster of machines
- Uses range-based partitioning to distribute documents based on a specific shard key
- Automatically balances the data associated with each shard
- Can be turned on and off per collection (table)

How does NoSQL vary from RDBMS?

- Looser schema definition
- Applications written to deal with specific documents/ data
  - Applications aware of the schema definition as opposed to the data
- Designed to handle distributed, large databases
- Trade offs:
  - No strong support for ad hoc queries but designed for speed and growth of database
    - Query language through the API
  - Relaxation of the ACID properties

Benefits of NoSQL

Elastic Scaling

- RDBMS scale up – bigger load , bigger server
- NO SQL scale out – distribute data across multiple hosts seamlessly

DBA Specialists

- RDMS require highly trained expert to monitor DB
- NoSQL require less management, automatic repair and simpler data models

Big Data

- Huge increase in data RDMS: capacity and constraints of data volumes at its limits
- NoSQL designed for big data

Drawbacks of NoSQL

- Support
  - RDBMS vendors provide a high level of support to clients
    - Stellar reputation
  - NoSQL – are open source projects with startups supporting them
    - Reputation not yet established
- Maturity
  - RDMS mature product: means stable and dependable
    - Also means old no longer cutting edge nor interesting
  - NoSQL are still implementing their basic feature set

What is MongoDB?

- Developed by 10gen
  - Founded in 2007
- A document-oriented, NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - In practice, this means you can store hashes with any keys and values that you choose
      - Keys are a basic data type but in reality stored as strings
      - Document Identifiers (_id) will be created for each document, field name reserved by system
    - Application tracks the schema and mapping
    - Uses BSON format
      - Based on JSON – B stands for Binary
- Written in C++
- Supports APIs (drivers) in many computer languages
  - JavaScript, Python, Ruby, Perl, Java, Java Scala, C#, C++, Haskell, Erlang

Features

Why MongoDB?

Document-oriented

- Documents (objects) map nicely to programming language data types
- Embedded documents and arrays reduce need for joins
- Dynamically-typed (schemaless) for easy schema evolution
- No joins and **no multi-document transactions** for high performance and easy scalability

High performance

- No joins and embedding makes reads and writes fast
- Indexes including indexing of keys from embedded documents and arrays
- Optional streaming writes (no acknowledgements)

High availability

- Replicated servers with automatic master failover

Easy scalability

- Automatic sharding (auto-partitioning of data across servers)
  - Reads and writes are distributed over shards
  - No joins or multi-document transactions make distributed queries easy and fast
- Eventually-consistent reads can be distributed over replicated servers

Rich query language

What it does, how it works

- MongoDB is a **server process** that runs on Linux, Windows and OS X.
  - It can be run both as a 32 or 64-bit application. We recommend running in 64-bit mode, since Mongo is limited to a total data size of about 2GB for all databases in 32-bit mode.
- Clients **connect** to the MongoDB process, optionally authenticate themselves if security is turned on, and perform a sequence of actions, such as **inserts, queries and updates**.
- MongoDB stores its data in files (default location is /data/db/), and uses **memory mapped files** for data management for efficiency.

Platforms

- Linux
- OS X
- Windows
- Other Unix Systems

Drivers

MongoDB currently has client support for the following programming languages:
mongodb.org Supported

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- .NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

Shell:
Javascript

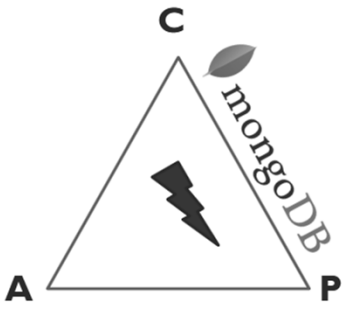
Why use MongoDB?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
  - No ERD diagram
- Not well suited for heavy and complex transactions systems

MongoDB: CAP approach

Focus on Consistency and Partition tolerance

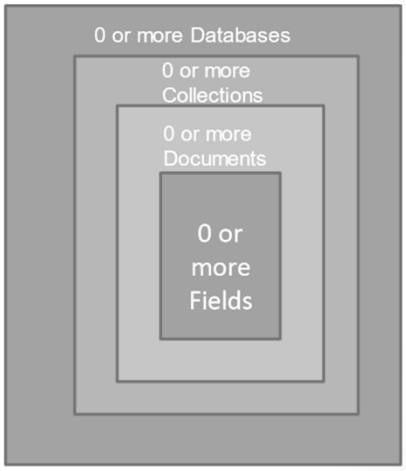
- **C**onsistency
  - all replicas contain the same version of the data
- **A**vailability
  - system remains operational on failing nodes
- **P**artition tolarence
  - multiple entry points
  - system remains operational on system split



CAP Theorem:
satisfying all three at the same time is impossible

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more 'databases'
- A database may have zero or more 'collections'.
- A collection may have zero or more 'documents'.
- A document may have one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.



RDB Concepts to NO SQL

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

Collection is not strict about what it Stores

Schema-less

Hierarchy is evident in the design

Embedded Document ?

BSON format

- Binary-encoded serialization of JSON-like documents
- Zero or more key/value pairs are stored as a single entity
- Each entry consists of a field name, a data type, and a value
- Large elements in a BSON document are prefixed with a length field to facilitate scanning

Sample Document

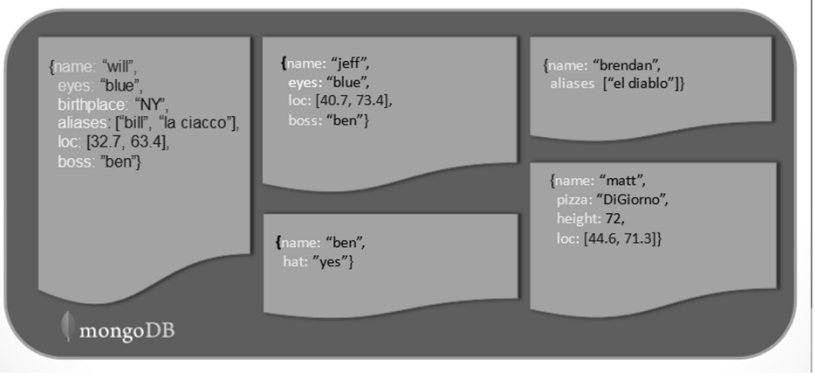
Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection could have different data
  - Addresses NULL data fields



JSON format

- Data is in name / value pairs
- A name/value pair consists of a field name followed by a colon, followed by a value:
  - Example: "name": "R2-D2"
- Data is separated by commas
  - Example: "name": "R2-D2", race : "Droid"
- Curly braces hold objects
  - Example: {"name": "R2-D2", race : "Droid", affiliation: "rebels"}
- An array is stored in brackets []
  - Example [{"name": "R2-D2", race : "Droid", affiliation: "rebels"},
  - {"name": "Yoda", affiliation: "rebels"}]

Index Functionality

- B+ tree indexes
- An index is automatically created on the _id field (the primary key)
- Users can create other indexes to improve query performance or to enforce Unique values for a particular field
- Supports single field index as well as Compound index
  - Like SQL order of the fields in a compound index matters
  - If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array
- Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (so ignore records that do not have the field defined)
- If an index is both unique and sparse – then the system will reject records that have a duplicate key value but allow records that do not have the indexed field defined

CRUD operations

- Create
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- Update
  - db.collection.update(<query>, <update>, <options>)
- Delete
  - db.collection.remove(<query>, <justOne>)

Collection specifies the collection or the 'table' to store the document

CRUD examples

```
> db.user.insert({
      first: "John",
      last : "Doe",
      age: 39
})
```

```
> db.user.find ()
{ "_id" : ObjectId("51"),
      "first" : "John",
      "last" : "Doe",
      "age" : 39
}
```

```
> db.user.update(
      {"_id" : ObjectId("51")},
      {
          $set:{
              age: 40,
              salary: 7000}
      }
)
```

```
> db.user.remove({
      "first": /^J/
})
```

Replication of data

- Ensures redundancy, backup, and automatic failover
  - Recovery manager in the RDMS
- Replication occurs through groups of servers known as replica sets
  - Primary set – set of servers that client tasks direct updates to
  - Secondary set – set of servers used for duplication of data
  - At the most can have 12 replica sets
    - Many different properties can be associated with a secondary set i.e. secondary-only, hidden delayed, arbiters, non-voting
  - If the primary set fails the secondary sets 'vote' to elect the new primary set

Consistency of data

- All read operations issued to the primary of a replica set are consistent with the last write operation
  - Reads to a primary have **strict consistency**
    - Reads reflect the latest changes to the data
  - Reads to a secondary have **eventual consistency**
    - Updates propagate gradually
  - If clients permit reads from secondary sets – then client may read a previous state of the database
  - Failure occurs before the secondary nodes are updated
    - System identifies when a rollback needs to occur
    - Users are responsible for manually applying rollback changes

Document

- MongoDB store record/data in the form of documents, which is a data structure composed of **field** and **value pairs.**
- MongoDB documents are similar to JSON objects.
- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.

Document Structure

- MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
field1: value1,
field2: value2,
...
fieldN: valueN
}
```

- The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.
- Field names are Strings.

Fields

The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For example, the following document contains values of varying types:

copy

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

The above fields have the following data types:

- `_id` holds an ObjectId.
- `name` holds an *embedded document* that contains the fields `first` and `last`.
- `birth` and `death` hold values of the *Date* type.
- `contribs` holds an *array of strings*.
- `views` holds a value of the *NumberLong* type.

The `_id` Field

In MongoDB, each document stored in a collection requires a unique _id field that acts as a primary key. If an inserted document omits the `_id` field, the MongoDB driver automatically generates an ObjectId for the `_id` field.

This also applies to documents inserted through update operations with upsert: true.

The `_id` field has the following behavior and constraints:

- By default, MongoDB creates a unique index on the `_id` field during the creation of a collection.
- The `_id` field is always the first field in the documents. If the server receives a document that does not have the `_id` field first, then the server will move the field to the beginning.
- The `_id` field may contain values of any BSON data type, other than an array.

Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of an embedded document.

Arrays

To specify or access an element of an array by the zero-based index position, concatenate the array name with the dot (`.`) and zero-based index position, and enclose in quotes:

copy

```
"<array>.<index>"
```

For example, given the following field in a document:

copy

```
{
   ...
   contribs: [ "Turing machine", "Turing test", "Turingery" ],
   ...
}
```

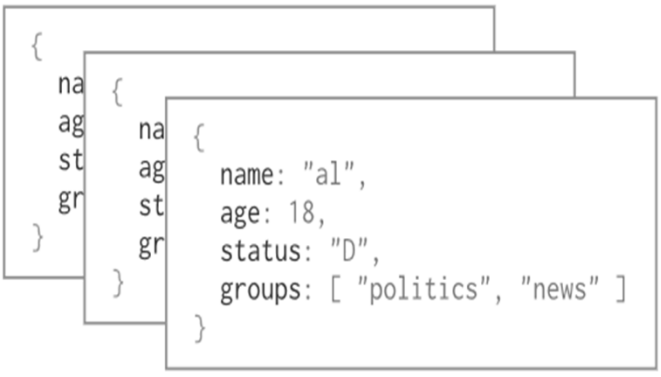
To specify the third element in the `contribs` array, use the dot notation `"contribs.2"`.

Collection

- MongoDB stores all documents in collections.
- A collection is a group of related documents.
- Collections are analogous to a table in relational databases.

Collection

MongoDB stores BSON documents, i.e. data records, in collections; the collections in databases.



Collection

In short We Can say:

MongoDB-Datatypes

- Documents in MongoDB can be thought of as "JSON-like" in that they are conceptually similar to objects in JavaScript.
- JSON is a simple representation of data:

- **Null:** Use to represent both a null value and a nonexistent field:

{field : **null**}

- **Boolean:** which can be used for the values true and false:

{field : **true**}

- **String:** Any string of characters can be represented using the string type:

{field : "foodbar"}

MongoDB-Datatypes contd…

- **Number:** The shell defaults to using 64-bit floating point numbers. Thus, these numbers look "normal" in the shell:

{field : 3.14} or: {field : 3}

- For integers, use the NumberInt or NumberLong classes, which represent 4-byte or 8-byte signed integers, respectively.

{field : NumberInt(3)}

{field : NumberLong(3)}

- **Dates:** are stored as milliseconds. The time zone is not stored:

{field : **new** Date()}

MongoDB-Datatypes contd…

- **Array:** Sets or lists of values can be represented as arrays:
  {field : ["a", "b", "c"]}
- **Embedded document:** Documents can contain entire documents embedded as values in a parent document:
  {field : {field : "abc"}}
- **object id:** An object id is a 12-byte ID for documents.
  {_id : ObjectId()}

Collections

MongoDB stores documents in collections. Collections are analogous to tables in relational databases.

Create a Collection

If a collection does not exist, MongoDB creates the collection when you first store data for that collection.

copy

```
db.myNewCollection2.insertOne( { x: 1 } )
db.myNewCollection3.createIndex( { y: 1 } )
```

Both the `insertOne()` and the `createIndex()` operations create their respective collection if they do not already exist.

BSON

- BSON is a bin-ary-en-coded seri-al-iz-a-tion of JSON-like doc-u-ments. BSON is designed to be lightweight, traversable, and efficient. BSON, like JSON, supports the embedding of objects and arrays within other objects and arrays
- **BSON and MongoDB**
- MongoDB uses BSON as the data storage and network transfer format for "documents".
- BSON at first seems BLOB-like, but there exists an important difference: the **Mongo database understands BSON internals**. This means that MongoDB can "reach inside " BSON objects, even nested ones. Among other things, this allows MongoDB to build indexes and match objects against query expressions on both top-level and nested BSON keys.
- To be efficient, MongoDB uses a format called BSON which is a **binary representation** of this data. BSON **is faster to scan for specific fields** than JSON. Also BSON adds some additional types such as a date data type and a byte-array (bindata) datatype. BSON maps readily to and from JSON and also to various data structures in many programming languages.
- **Client** drivers **serialize data to BSON**, then transmit the data over the wire to the db. **Data is stored on disk in BSON format**. Thus, on a retrieval, the database does very little translation to send an object out, allowing high efficiency. The client driver unserialized a received BSON object to its native language format.

Goals of BSON

- So what are the goals of BSON? They are:

1. **Fast scan-ability.** For very large JSON documents, scanning can be slow. To skip a nested document or array we have to scan through the intervening field completely. In addition as we go we must count nestings of braces, brackets, and quotation marks. In BSON, the size of these elements is at the beginning of the field's value, which makes skipping an element easy.
2. **Easy manipulation.** We want to be able to modify information within a document efficiently. For example, incrementing a field's value from 9 to 10 in a JSON text document might require shifting all the bytes for the remainder of the document, which if the document is large could be quite costly. (Albeit this benefit is not comprehensive: adding an element to an array mid-document would result in shifting.) It's also helpful to not need to translate the string "9" to a numeric type, increment, and then translate back.
3. **Additional data types.** JSON is potentially a great interchange format, but it would be nice to have a few more data types. Most importantly is the addition of a "byte array" data type. This avoids any need to do base64 encoding, which is awkward.

   In addition to the basic JSON types of string, integer, boolean, double, null, array, and object, these types include date, object id, binary data, regular expression, and code.

A BSON document

```
{ author: 'joe',
 created : new Date('03/28/2009'),
 title : 'Yet another blog post',
 text : 'Here is the text...',
 tags : [ 'example', 'joe' ],
 comments : [
            { author: 'jim',
              comment: 'I disagree'
            },
            { author: 'nancy',
              comment: 'Good post'
            }
 ]
}
```

This document is a blog post, so we can store in a "posts" collection using the shell:

```
> doc = { author : 'joe', created : new Date('03/28/2009'), ... }
> db.posts.insert(doc);
```

Use: mongoimport To Load CSV data

```
$ cat > locations.csv
Name,Address,City,State,ZIP
Jane Doe,123 Main St,Whereverville,CA,90210
John Doe,555 Broadway Ave,New York,NY,10010
 ctrl-d
$ mongoimport -d mydb -c things --type csv --file locations.csv --headerline
connected to: 127.0.0.1
imported 3 objects
$ mongo
MongoDB shell version: 1.7.3
connecting to: test
> use mydb
switched to db mydb
> db.things.find()
{ "_id" : ObjectId("4d32a36ed63d057130c08fca"), "Name" : "Jane Doe", "Address" : "123 Main St
{ "_id" : ObjectId("4d32a36ed63d057130c08fcb"), "Name" : "John Doe", "Address" : "555 Broadwa
```

Databases

In MongoDB, databases hold collections of documents.

To select a database to use, in the mongo shell, issue the `use <db>` statement, as in the following example:

```
use myDB
```

Create a Database

If a database does not exist, MongoDB creates the database when you first store data for that database. As such, you can switch to a non-existent database and perform the following operation in the mongo shell:

copy

```
use myNewDB

db.myNewCollection1.insertOne( { x: 1 } )
```

The `insertOne()` operation creates both the database **myNewDB** and the collection **myNewCollection1** if they do not already exist.

END