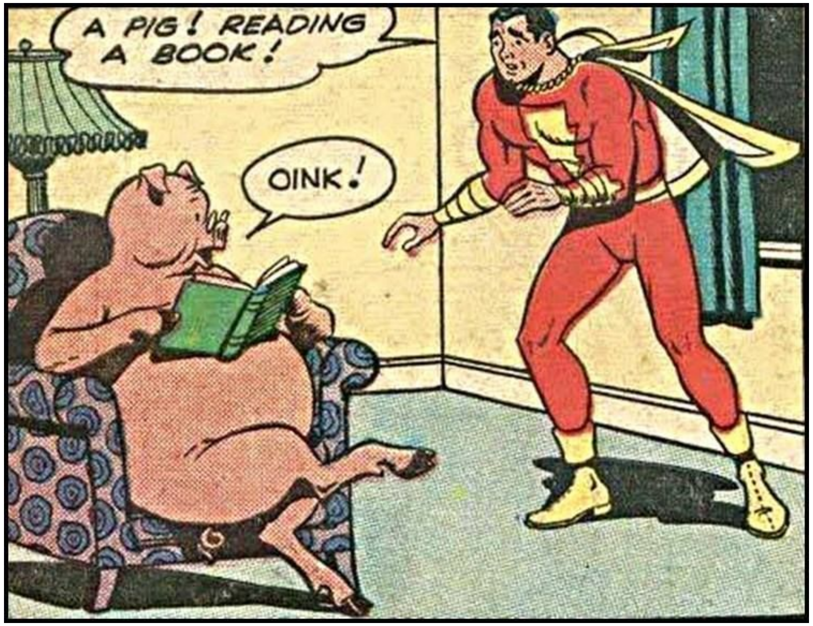
NoSQL
Document Databases

Problem Set #5 is
due on Thursday

Problem Set #6 is coming,
but getting simpler
every day it is delayed



NoSQL Databases and Data Types

1. **Key-value** stores:
    - Can store any (text or binary) data
        - often, if using JSON data, additional functionality is available
2. **Document** databases
    - Structured text data - Hierarchical tree data structures
        - typically JSON, XML
3. **Columnar** stores
    - Rows that have many columns associated with a row key
        - can be written as JSON

Unstructured Data Formats

- Binary Data
  - often, we want to store objects (class instances)
  - objects can be binary serialized (marshalled)
    - and kept in a key-value store
  - there are several popular serialization formats
    - Protocol Buffers, Apache Thrift
- Structured Text Data
  - JSON, BSON (Binary JSON)
    - JSON is currently number one data format used on the Web
  - XML: eXtensible Markup Language
  - RDF: Resource Description Framework

JSON: Basic Information

- Text-based open standard for data interchange
  - Serializing and transmitting structured data
- JSON = JavaScript Object Notation
  - Originally specified by Douglas Crockford in 2001
  - Derived from JavaScript scripting language
  - Uses conventions of the C-family of languages
- Filename: *.json
- Internet media (MIME) type: application/json
- Language independent

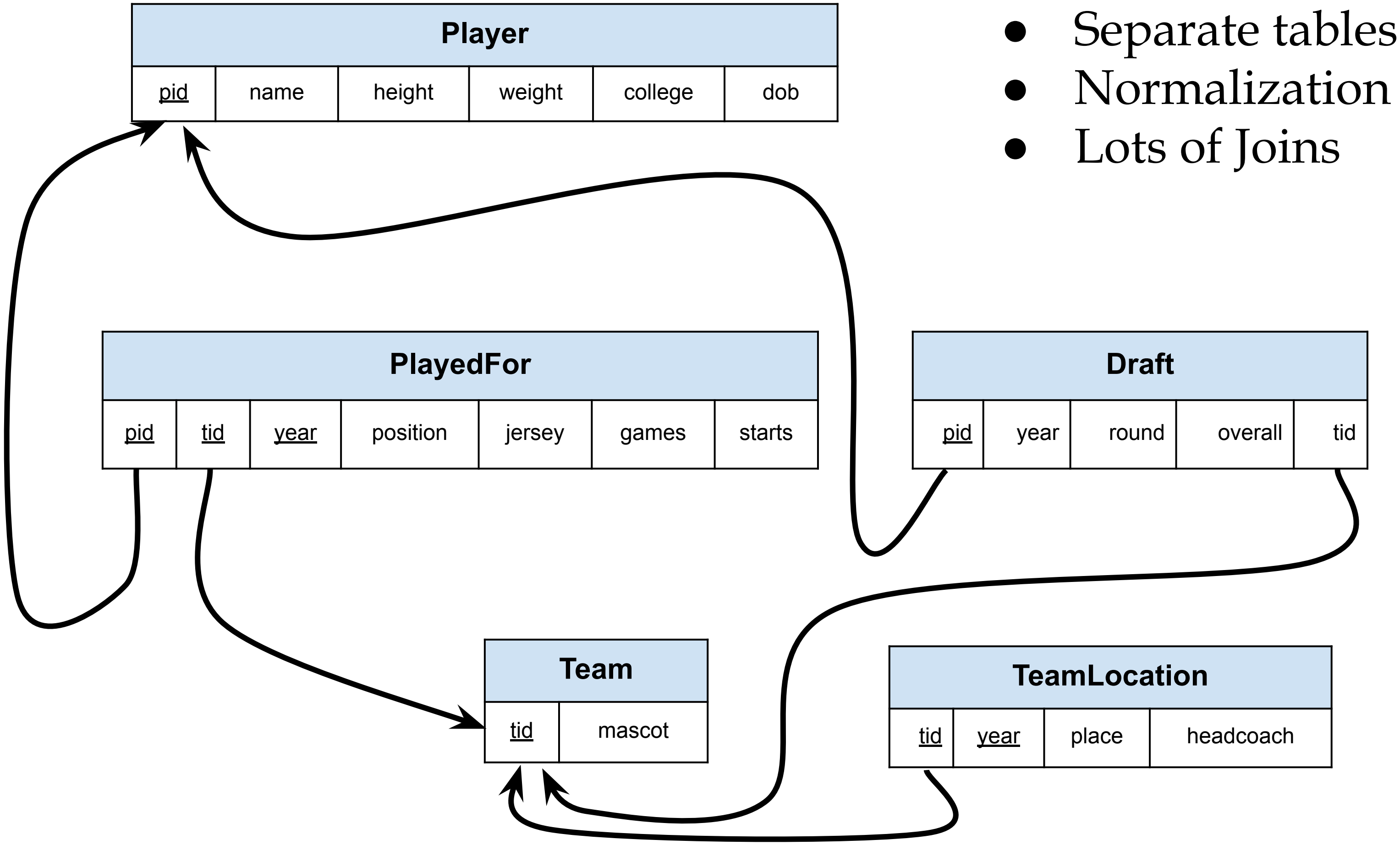
http://www.json.org

JSON:Example

```
[{  'pid': 28352, 'name': 'Brandin Cooks',
    'height': '5-10', 'weight': '189',
    'birthdate': '1993-09-25', 'college': 'Oregon St',
    'draft': {'team': 'New Orleans Saints', 'round': '1', 'order': 20, 'year': 2014 },
    'roster': [
        {'year': 2014, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 10, 'starts': 7 },
        {'year': 2015, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 16, 'starts': 12 },
        {'year': 2016, 'team': 'New Orleans Saints', 'position': 'WR', 'jersey': '10', 'games': 16, 'starts': 12 },
        {'year': 2017, 'team': 'New England Patriots', 'position': 'WR', 'jersey': '14', 'games': 16, 'starts': 15 },
        {'year': 2018, 'team': 'Los Angeles Rams', 'position': 'WR', 'jersey': '12', 'games': 16, 'starts': 16 }]
 },
 {  'pid': 22721, 'name': 'Tom Brady',
    'height': '6-4', 'weight': '225',
    'birthdate': '1977-08-03', 'college': 'Michigan',
    'draft': {'team': 'New England Patriots', 'round': '6', 'order': 199, 'year': 2000},
    'roster': [
        {'year': 2000, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 1, 'starts': 0 },
        {'year': 2001, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 15, 'starts': 14 },
        {'year': 2002, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2003, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2004, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2005, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2006, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2007, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2008, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 1, 'starts': 1 },
        {'year': 2009, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2010, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2011, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2012, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2013, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2014, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2015, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2016, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 12, 'starts': 12 },
        {'year': 2017, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 },
        {'year': 2018, 'team': 'New England Patriots', 'position': 'QB', 'jersey': '12', 'games': 16, 'starts': 16 }]
  }
]
```

Compared to a Relational DB

- Separate tables
- Normalization
- Lots of Joins



Player

pid	name	height	weight	college	dob

PlayedFor

pid	tid	year	position	jersey	games	starts

Draft

pid	year	round	overall	tid

Team

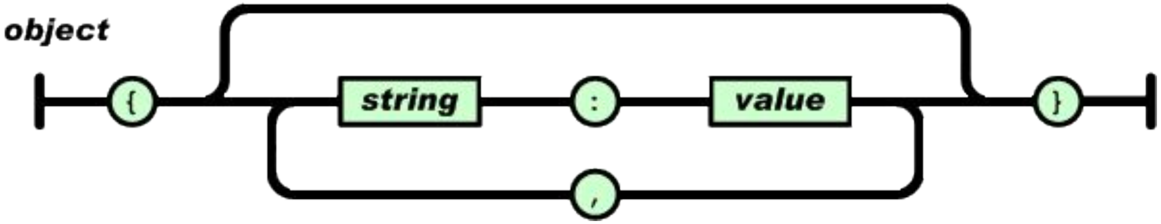
tid	mascot

TeamLocation

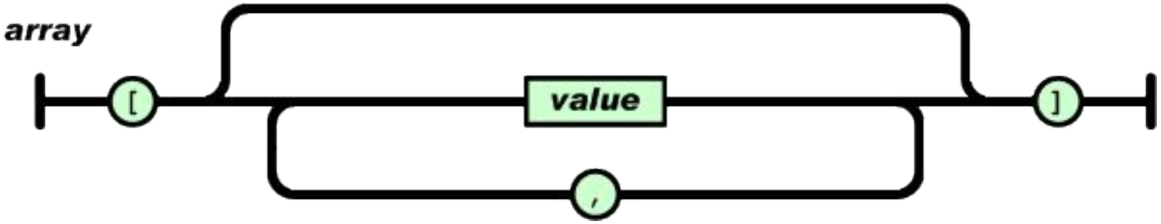
tid	year	place	headcoach

JSON: Data Types (1)

- object – an unordered set of name+value pairs
  - these pairs are called properties (members) of an object
  - syntax: { name: value, name: value, name: value, ...}

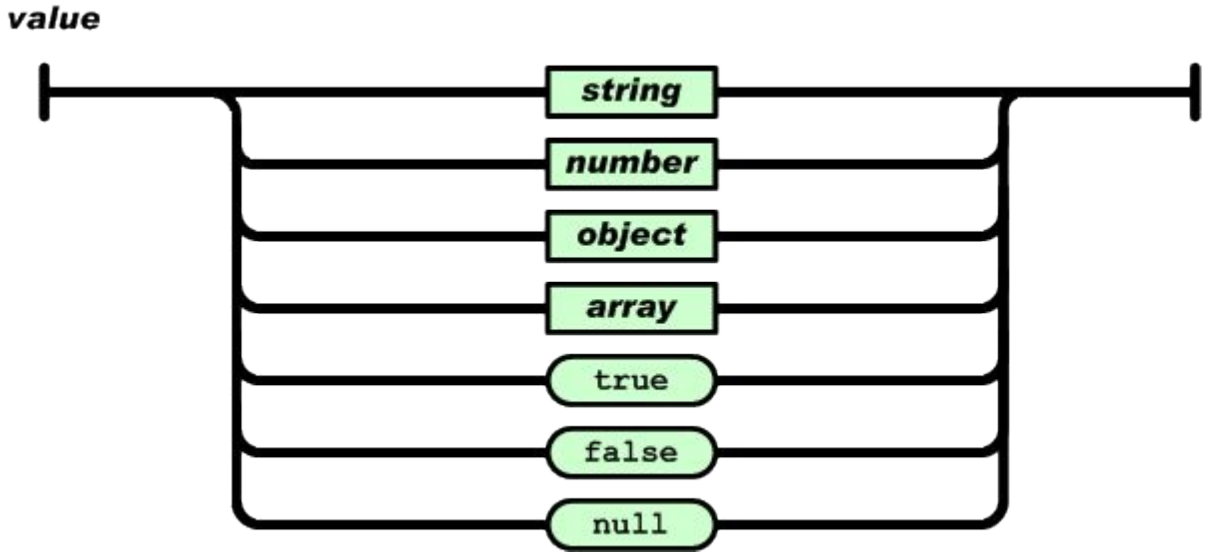


- array – an ordered collection of values (elements)
  - syntax: [comma-separated values]



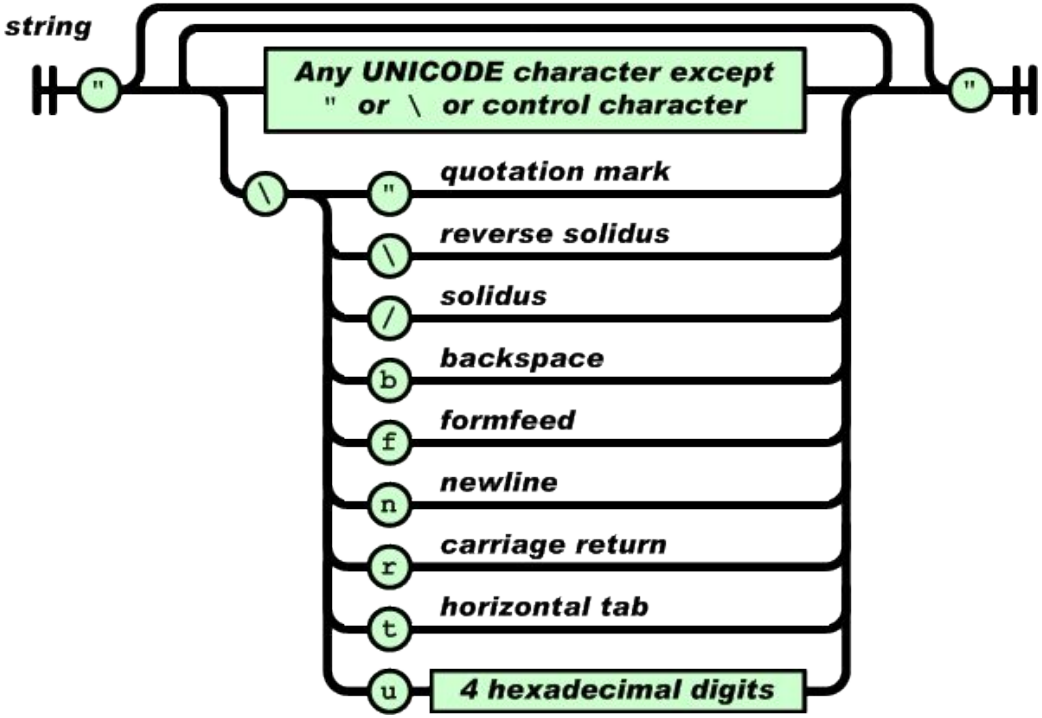
JSON: Data Types (2)

- value – string in double quotes / number / true or false (i.e., Boolean) / null / object / array



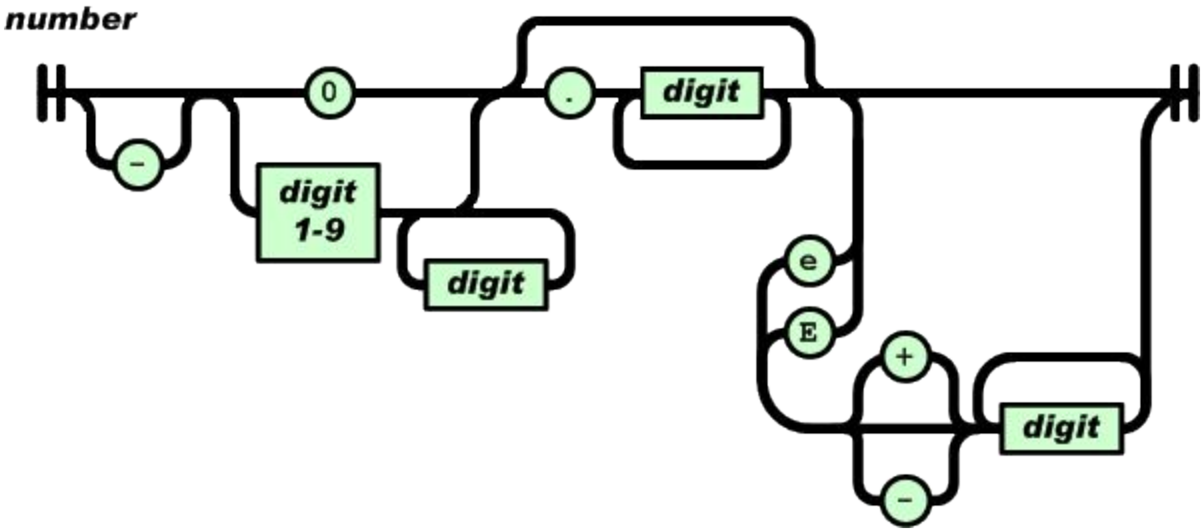
JSON: Data Types (3)

- string – sequence of zero or more Unicode characters, wrapped in double quotes
    - Backslash escaping



JSON: Data Types (4)

- number – like a C, Python, or Java number
  - Integer or float
  - Octal and hexadecimal formats are not used



JSON Properties

- There are no comments in JSON
  - Originally, there was but they were removed for security
- No way to specify precision/size of numbers
  - It depends on the parser and the programming language
- There exists a standard “JSON Schema”
  - A way to specify the schema of the data
  - Field names, field types, required/optional fields, etc.
  - JSON Schema is written in JSON, of course
    - see example below

JSON Schema: Example

```
{
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "properties": {
    "conferences": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": { "type": "string" },
          "start": { "type": "string", "format": "date" },
          "end": { "type": "string", "format": "date" },
          "web": { "type": "string" },
          "price": { "type": "number" },
          "currency": { "type" : "string",
                  "enum": ["CZK", "USD", "EUR", "GBP"] },
          "topics": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
```

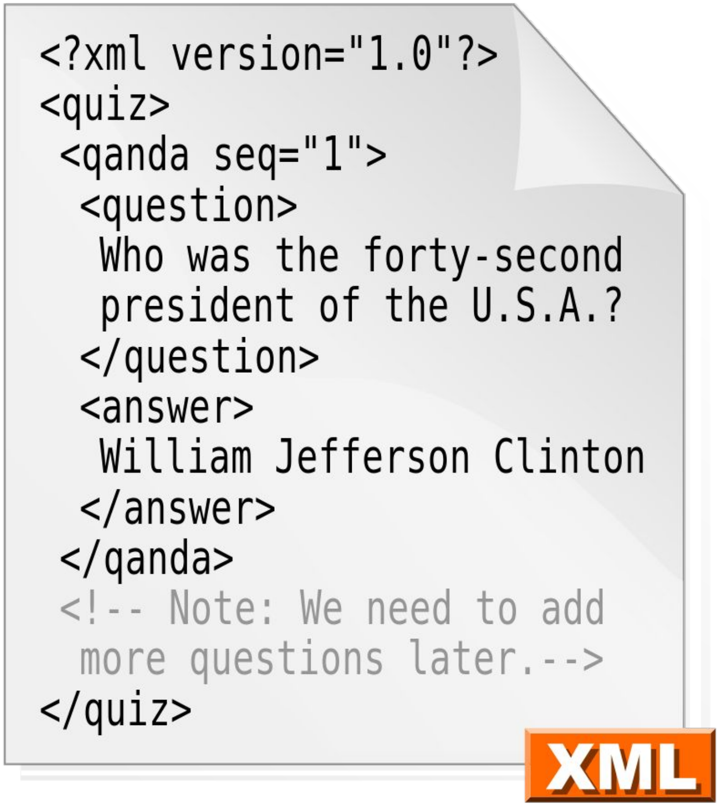
```
          "venue": {
            "type": "object",
            "properties": {
              "name": { "type": "string" },
              "location": {
                "type": "object",
                "properties": {
                  "lat": { "type": "number" },
                  "lon": { "type": "number" }
                }
              }
            },
            "required": ["name"]
          }
        },
        "required": ["name", "start", "end",
                     "web", "price", "topics"]
      }
    }
  }
}
```

Document with JSON Schema

```
{
  "conferences":
  [
    {
      "name": "XML Prague 2015",
      "start": "2015-02-13",
      "end": "2015-02-15",
      "web": "http://xmlprague.cz/",
      "price": 120,
      "currency": "EUR",
      "topics": ["XML", "XSLT", "XQuery", "Big Data"],
      "venue": {
        "name": "VŠE Praha",
        "location": {
          "lat": 50.084291,
          "lon": 14.441185
        }
      }
    },
    {
      "name": "DATAKON 2014",
      "start": "2014-09-25",
      "end": "2014-09-29",
      "web": "http://www.datakon.cz/",
      "price": 290,
      "currency": "EUR",
      "topics": ["Big Data", "Linked Data", "Open Data"]
    }
  ]
}
```

XML: Basic Information

- XML: eXtensible Markup Language
    - W3C standard (since 1996)
- Designed to be both human and machine readable
- example:



```xml
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

source: http://en.wikipedia.org/wiki/XML

XML: Features and Comparison

- Standard ways to specify XML document schema:
  - DTD, XML Schema, etc.
  - concept of Namespaces; XML editors (for given schema)
- Technologies for parsing: DOM, SAX
- Many associated technologies:
  - XPath, XQuery, XSLT (transformation)
- XML is good for configurations, meta-data, etc.
- XML databases are mature, not considered NoSQL
- Currently, JSON format rules:
  - compact, easier to write, meets most needs

NoSQL Document Databases

- Basic concept of data: *Document*
- Documents are self-describing pieces of data
  - Hierarchical tree data structures
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, …
- Documents in a collection should be “similar”
  - Their schema can differ
- Often: Documents stored as values of key-value
  - Key-value stores where the values are examinable
  - Building search indexes on various keys/fields

Why Document Databases

- XML and JSON are popular for data exchange
  - Recently mainly JSON
- Data stored in document DB can be used directly

- Databases often store objects from memory
  - Using RDBMS, we must do Object Relational Mapping (ORM)
    - ORM is relatively demanding
  - JSON is much closer to structure of memory objects
    - It was originally for JavaScript objects
    - Object Document Mapping (ODM) is faster

Document Databases

mongoDB

Apache CouchDB relax

OrientDB

RAVENDB

terrastore

PostgreSQL

amazon DynamoDB

MS Azure DocumentDB

Example: MongoDB

- Initial release: 2009
    - Written in C++
    - Open-source
    - Cross-platform
- JSON documents

```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

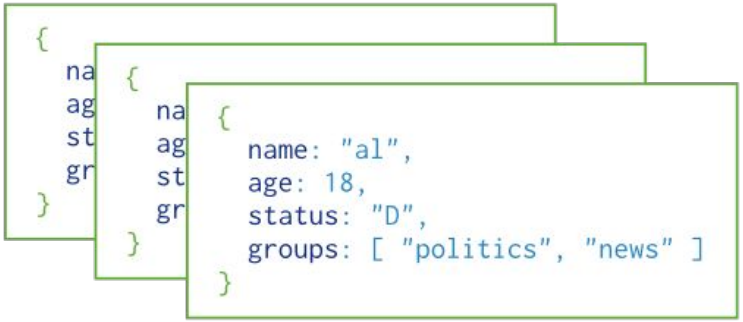
- Basic features:
    - High performance – many indexes
    - High availability – replication + eventual consistency + automatic failover
    - Automatic scaling – automatic sharding across the cluster
    - MapReduce support

http://www.mongodb.org/

MongoDB: Terminology

RDBMS	**MongoDB**
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id

- each JSON document:
  - belongs to a collection
  - has a field _id
    - unique within the collection
- each collection:
  - belongs to a “database”



```
{
   name: "al",
   age: 18,
   status: "D",
   groups: [ "politics", "news" ]
}
```

Collection

Documents

- Use JSON for API communication
- Internally: BSON
  - Binary representation of JSON
  - For storage and inter-server communication

- Document has a maximum size: 16MB (in BSON)
  - Not to use too much RAM
  - GridFS tool can divide larger files into fragments

Document Fields

- Every document must have field _id
  - Used as a primary key
  - Unique within the collection
  - Immutable
  - Any type other than an array
  - Can be generated automatically

- Restrictions on field names:
  - The field names cannot start with the $ character
    - Reserved for operators
  - The field names cannot contain the . character
    - Reserved for accessing sub-fields

Database Schema

- Documents have flexible schema
    - Collections do not enforce specific data structure
    - In practice, documents in a collection are similar

- Key decision of data modeling:
    - References vs. embedded documents

    - In other words: Where to draw lines between aggregates
        - Structure of data
        - Relationships between data

Schema: Embedded Docs

- Related data in a single document structure
    - Documents can have subdocuments (in a field or array)

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

Embedded sub-document

Embedded sub-document

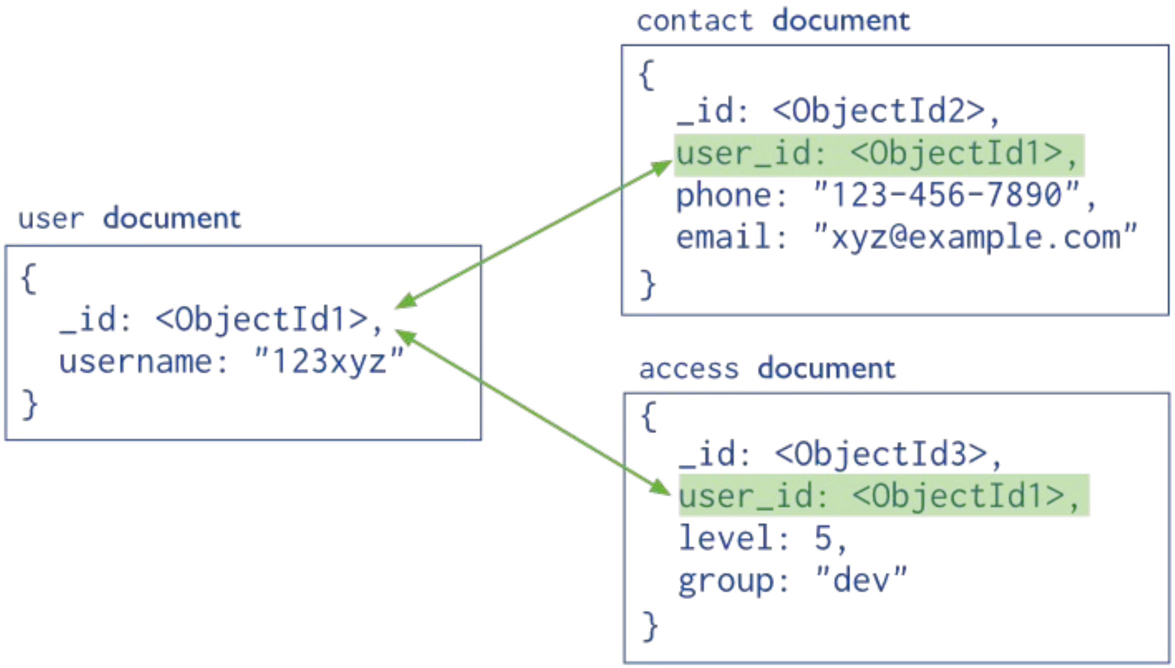
Schema: Embedded Docs (2)

- Denormalized schema
- Main advantage:
  Manipulate related data in a single operation
- Use this schema when:
  - One-to-one relationships: one doc “contains” the other
  - One-to-many: if children docs have one parent document

- Disadvantages:
  - Documents may grow significantly during the time
  - Impacts both read/write performance
    - Document must be relocated on disk if its size exceeds allocated space
    - May lead to data fragmentation on the disk

Schema: References

- Links/references from one document to another
- Normalization of the schema



contact **document**

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

user **document**

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

access **document**

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

Schema: References (2)

- More flexibility than embedding
- Use references:
  - When embedding would result in duplication of data
    - and only insignificant boost of read performance
  - To represent more complex many-to-many relationships
  - To model large hierarchical data sets

- Disadvantages:
  - Can require more roundtrips to the server
    - Documents are accessed one by one

Querying: Basics

- Mongo query language
- A MongoDB query:
  - Targets a specific collection of documents
  - Specifies criteria that identify the returned documents
  - May include a projection to specify returned fields
  - May impose limits, sort, orders, …
- Basic query - all documents in the collection:

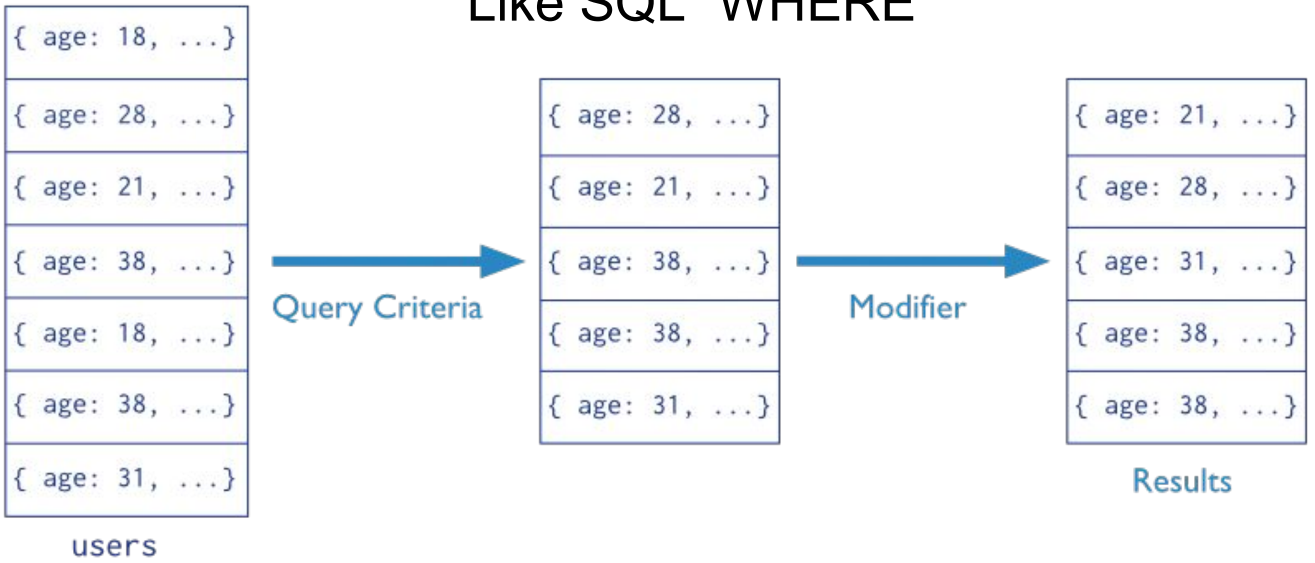
```
db.users.find()      -- Like SELECT *
db.users.find( {} )
```

Querying: Example

Collection — Query Criteria — Modifier

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```

Like SQL "WHERE"



users		Results
{ age: 18, ...}	{ age: 28, ...}	{ age: 21, ...}
{ age: 28, ...}	{ age: 21, ...}	{ age: 28, ...}
{ age: 21, ...}	{ age: 38, ...}	{ age: 31, ...}
{ age: 38, ...}	{ age: 38, ...}	{ age: 38, ...}
{ age: 18, ...}	{ age: 31, ...}	{ age: 38, ...}
{ age: 38, ...}		
{ age: 31, ...}		

Querying: Selection

```
db.inventory.find({ type: "snacks" })
```

- All documents from collection inventory where the type field has the value snacks

```
db.inventory.find(
   { type: { $in: [ 'food', 'snacks' ] } } )
```

- All inventory docs where the type field is either food or snacks

```
db.inventory.find(
   { type: 'food', price: { $lt: 9.95 } } )
```

- All ... where the type field is food and the price is less than 9.95

Inserts

```
db.inventory.insert( { _id: 10, type: "misc",
item: "card", qty: 15 } )
```

- Inserts a document with three fields into collection inventory
    - User-specified _id field

```
db.inventory.insert(
   { type: "book", item: "journal" } )
```

- The database generates `_id` field

```
$ db.inventory.find()
{ "_id": ObjectId("58e209ecb3e168f1d3915300"),
type: "book", item: "journal" }
```

Updates

```
db.inventory.update(
    { type: "book", item : "journal" },
    { $set: { qty: 10 } },
    { upsert: true }  )
```

- Finds all docs matching query
  `{ type: "book", item : "journal" }`
- and sets the field `{ qty: 10 }`

- `upsert: true`
  - if no document in the inventory collection matches
  - creates a new document (generated _id)
    - it contains fields _id, type, item, qty

MapReduce

```
collection "accesses":
{
  "user_id": <ObjectId>,
  "login_time": <time_the_user_entered_the_system>,
  "logout_time": <time_the_user_left_the_system>,
  "access_type": <type_of_the_access>
}
```

- How much time did each user spend logged in
  - Counting just accesses of type “regular”

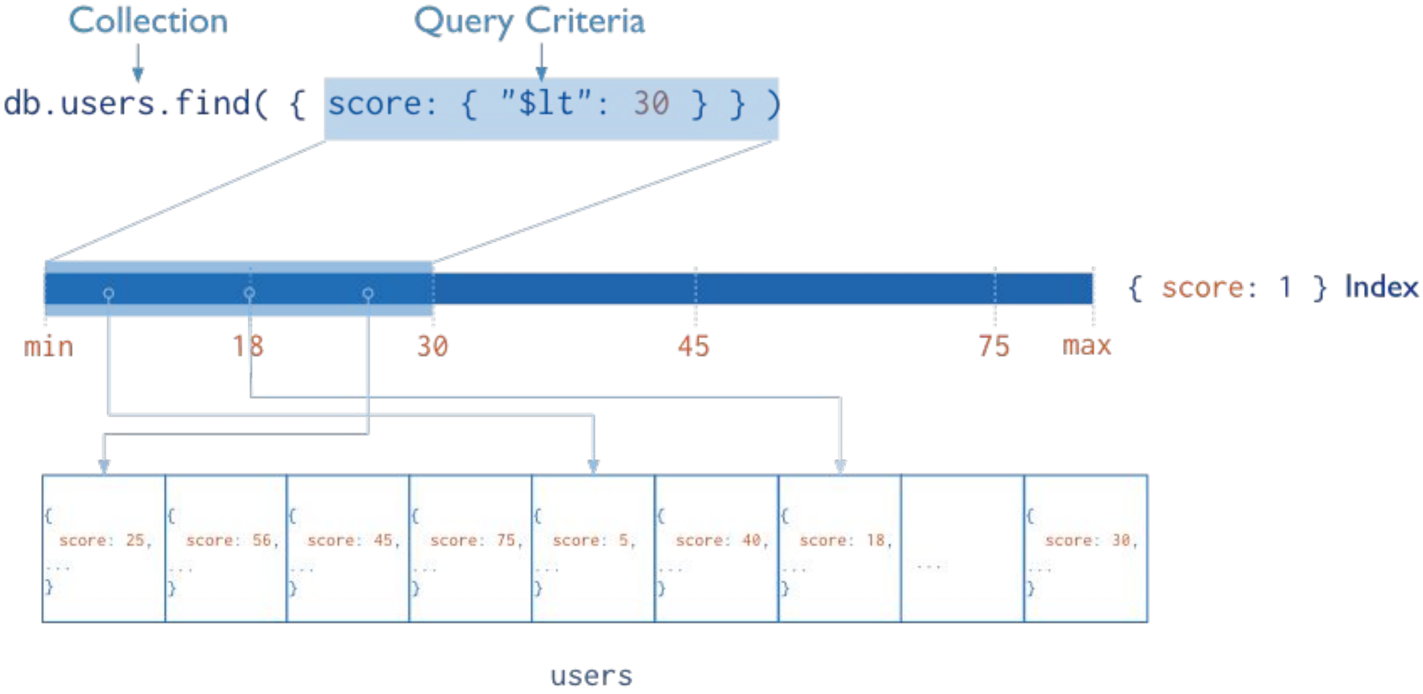
```
db.accesses.mapReduce(
  function() { emit (this.user_id, this.logout_time - this.login_time); },
  function(key, values) { return Array.sum( values ); },
  {
    query: { access_type: "regular" },
    out: "access_times"
  }
)
```

MongoDB Indexes

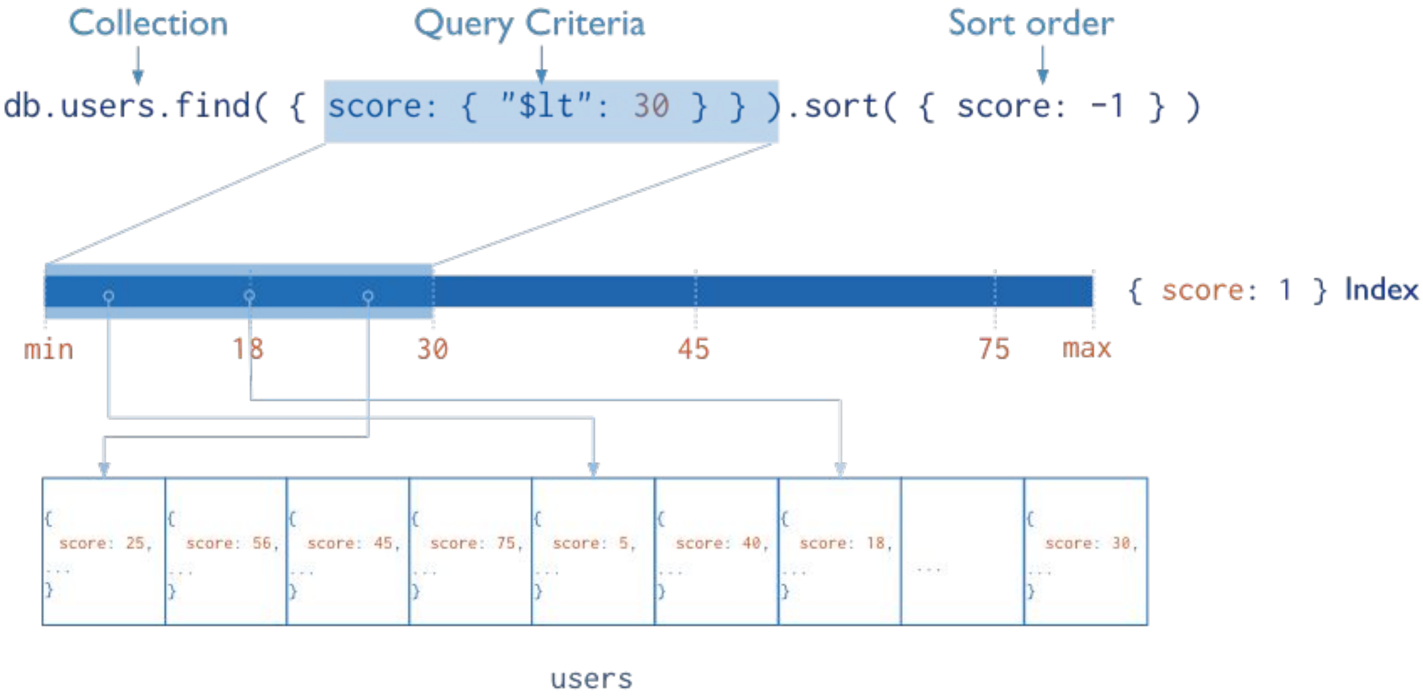
- Indexes are the key for MongoDB performance
  - Without indexes, MongoDB must scan every document in a collection to select matching documents
- Indexes store some fields in easily accessible form
  - Stores values of a specific field(s) ordered by the value

- Defined per collection
- Purpose:
  - To speed up common queries
  - To optimize performance of other specific operations

Indexes: Example of Use

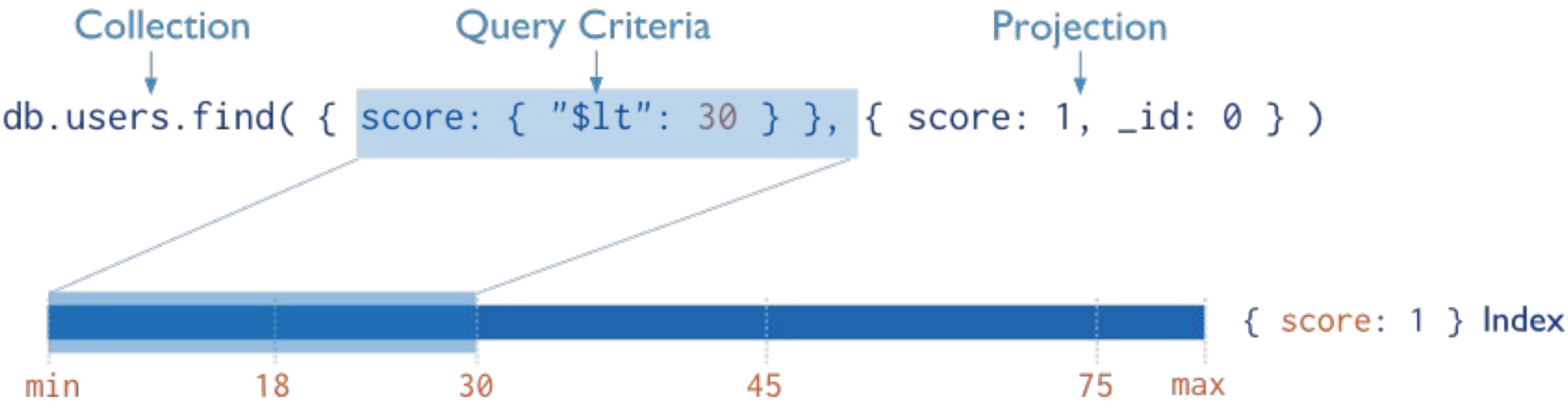


Indexes: Example of Use (2)



- The index can be traversed in order to return sorted results (without sorting)

Indexes: Example of Use (3)



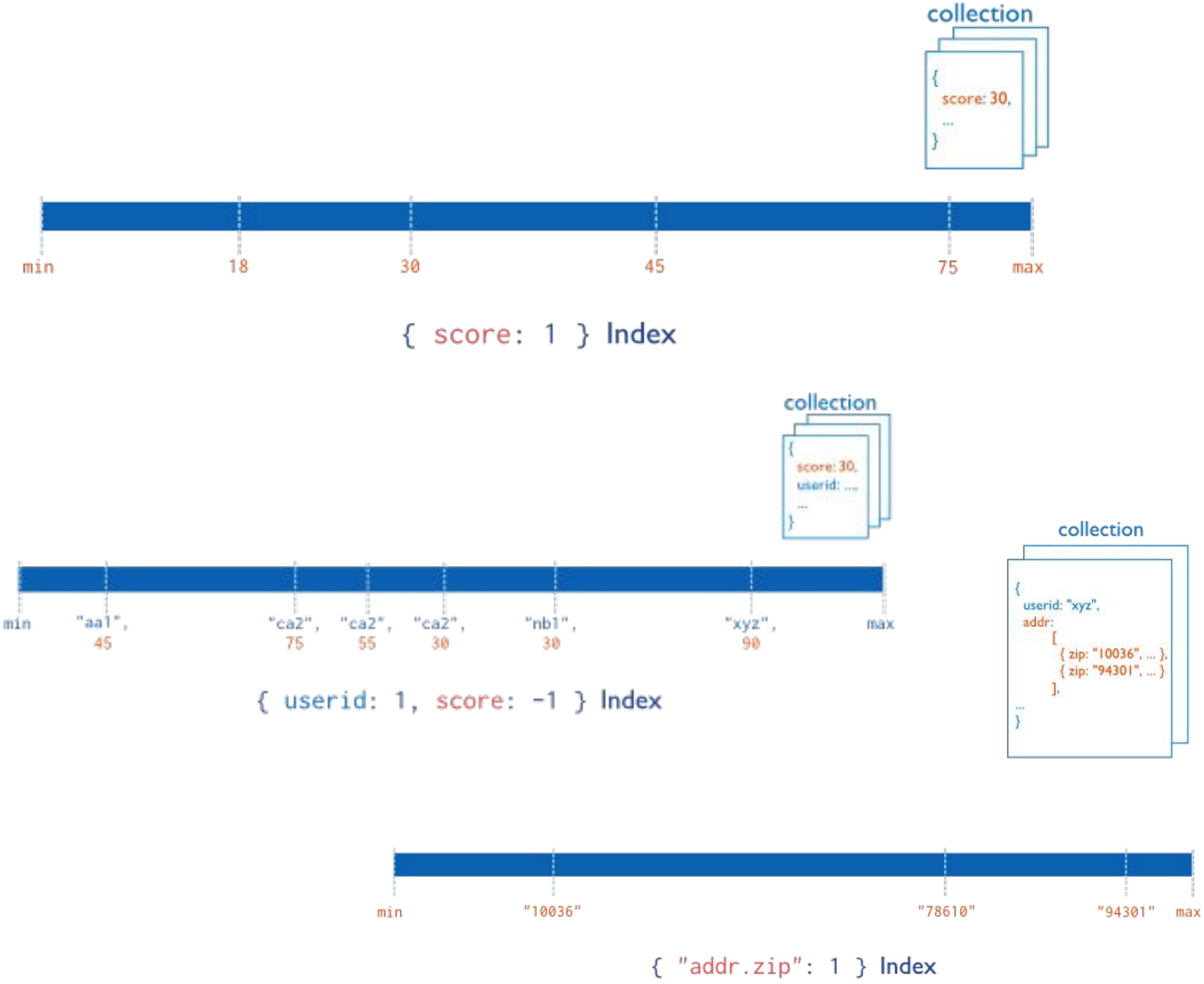
```
db.users.find( { score: { "$lt": 30 } }, { score: 1, _id: 0 } )
```

- MongoDB does not need to inspect data outside of the index to fulfill the query

Index Types

- **Default:** _id
  - Exists by default
    - If applications do not specify _id, it is created.
  - Unique
- **Single Field**
  - User-defined indexes on a single field of a document
- **Compound**
  - User-defined indexes on multiple fields
- **Multikey index**
  - To index the content stored in arrays
  - Creates separate index entry for each array element

Index Types (2)



- Index on score field (ascending)
- Compound Index on userid (ascending) AND score field (descending)
- Multikey index on the addr.zip field

Index Types (3)

- **Ordered** Index
  - B-Tree (see above)
- **Hash** Indexes
  - Fast *O*(1) indexes the hash of the value of a field
    - Only equality matches
- **Geospatial** Index
  - 2d indexes = use planar geometry when returning results
    - For data representing points on a two-dimensional plane
  - 2sphere indexes = spherical (Earth-like) geometry
    - For data representing longitude, latitude
- **Text** Indexes
  - Searching for string content in a collection

MongoDB: Behind the Curtain

- BSON format
- Distribution models
  - Replication
  - Sharding
  - Balancing
- MapReduce
- Transactions
- Journaling

BSON (Binary JSON) Format

- Binary-encoded serialization of JSON documents
    - Representation of documents, arrays, JSON simple data types + other types (e.g., date)

```
                                "\x16\x00\x00\x00\x02hello\x00
{"hello": "world"}          →    \x06\x00\x00\x00world\x00\x00"

                                "\x31\x00\x00\x00\x04BSON\x00\x26\x00
                                 \x00\x00\x020\x00\x08\x00\x00
{"BSON": ["awesome",             \x00awesome\x00\x011\x00\x33\x33\x33
5.05, 1986]}                →   \x33\x33\x33
                                 \x14\x40\x102\x00\xc2\x07\x00\x00
                                 \x00\x00"
```

http://www.bsonspec.org/

BSON: Basic Types

- `byte` – 1 byte (8-bits)
- `int32` – 4 bytes (32-bit signed integer)
- `int64` – 8 bytes (64-bit signed integer)
- `double` – 8 bytes (64-bit IEEE 754 floating point)

http://www.bsonspec.org/

BSON Grammar

```
document ::= int32 e_list "\x00"
```

- BSON document
- `int32` = total number of bytes in document

```
e_list ::= element e_list | ""
```

- Sequence of elements

http://www.bsonspec.org/

BSON Grammar (2)

`element ::= "\x01" e_name double`	Floating point
`| "\x02" e_name string`	UTF-8 string
`| "\x03" e_name document`	Embedded document
`| "\x04" e_name document`	Array
`| "\x05" e_name binary`	Binary data
`| …`	…

```
e_name ::= cstring
```

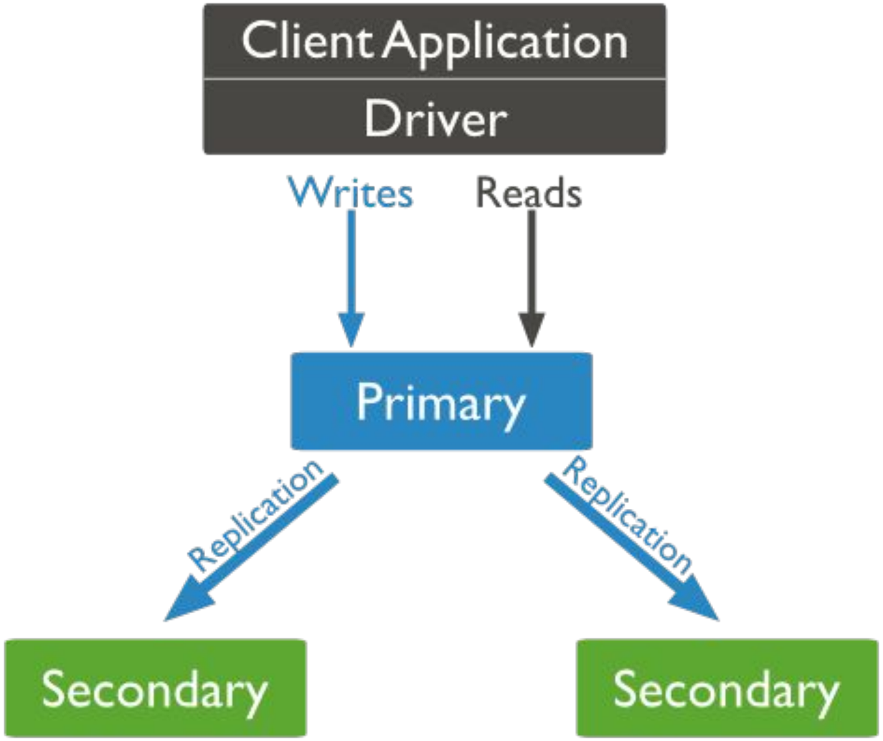
- Field key

```
cstring ::= (byte*) "\x00"
string ::= int32 (byte*) "\x00"
```

etc.…

Data Replication

- Master/slave replication
- Replica set = group of instances that host the same data set
  - **primary** (master) – handles all write operations
  - **secondaries** (slaves) – apply operations from the primary so that they have the same data set



Replication: Read & Write

- **Write operation:**
  1. Write operation is applied on the primary
  2. Operation is recorded to primary's oplog (operation log)
  3. Secondaries replicate the oplog + apply the operations to their data sets

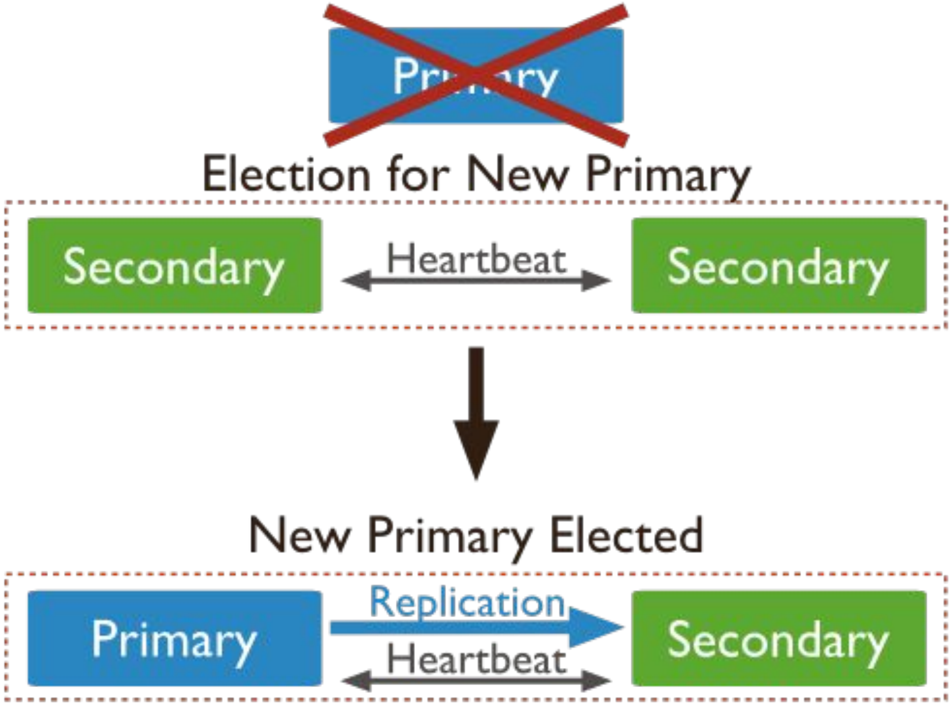
- **Read:** All replica set members can accept reads
  - By default, application directs its reads to the primary
    - Guaranties the latest version of a document
    - Decreases read throughput
  - Read preference mode can be set
    - See below

Replication: Read Modes

Read Preference Mode	**Description**
`primary`	operations read from the primary of the replica set
`primaryPreferred`	operations read from the primary, but if unavailable, operations read from secondary members
`secondary`	operations read from the secondary members
`secondaryPreferred`	operations read from secondary members, but if none is available, operations read from the primary
`nearest`	operations read from the nearest member (= shortest ping time) of the replica set

Replica Set Elections

- If the primary becomes unavailable, an election determines a new primary
  - Elections need some time
  - No primary => no writes

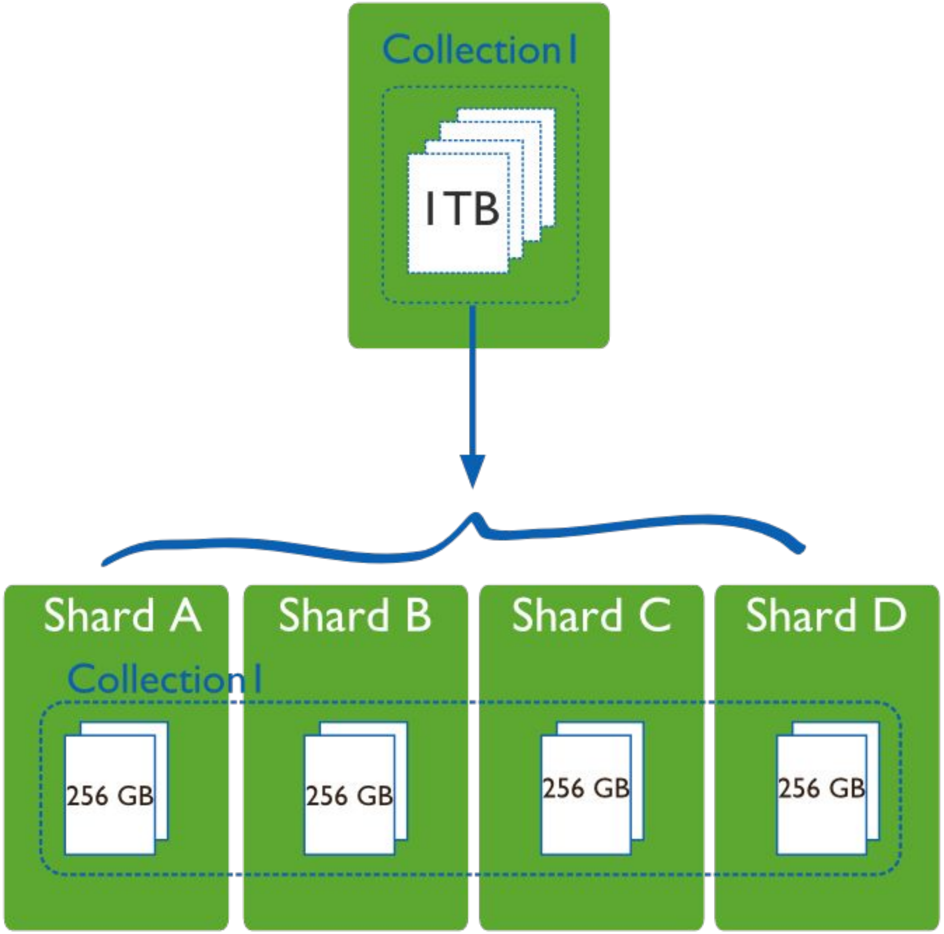


Replica Set: CAP

- Let us have three nodes in the replica set
  - Let's say that the master is disconnected from the other two
    - The distributed system is partitioned
  - The master finds out, that it is alone
    - Specifically, that can communicate with less than half of the nodes
    - And it steps down from being master (handles just reads)
  - The other two slaves "think" that the master failed
    - Because they form a partition with more than half of the nodes
    - And elect a new master
- In case of just two nodes in RS
  - Both partitions will become read-only
    - Similar case can occur with any even number of nodes in RS
  - Therefore, we can always add an arbiter node to an even RS

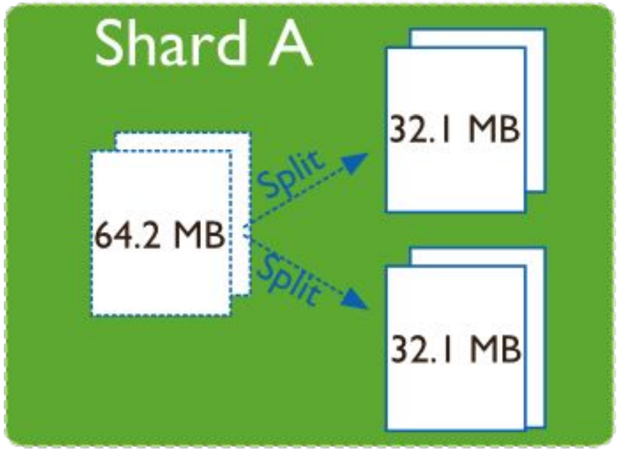
Sharding

- MongoDB enables collection partitioning (sharding)



Collection Partitioning

- Mongo partitions collection's data by the shard key
  - Indexed field(s) that exist in each document in the collection
    - Immutable
  - Divided into chunks, distributed across shards
    - **Range-based** partitioning
    - **Hash-based** partitioning
  - When a chunk grows beyond the size limit, it is split
    - Metadata change, no data migration

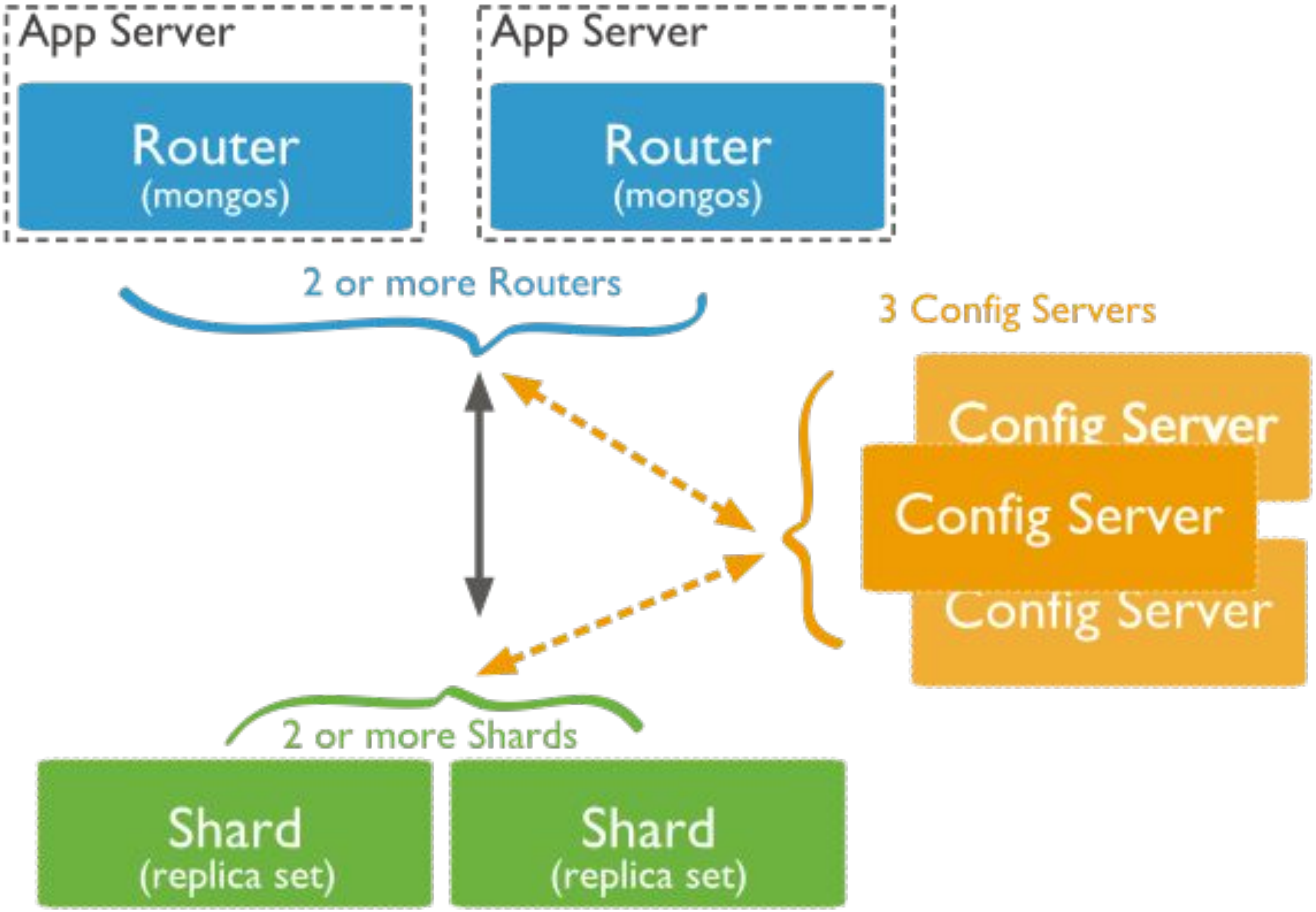


- Data balancing:
  - Background chunk migration

Sharding: Components

- MongoDB runs in cluster of different node types:
- Shards – store the data
  - Each shard is a replica set
    - Can be a single node
- Query routers – interface with client applications
  - Direct operations to the relevant shard(s)
    - + return the result to the client
  - More than one => to divide the client request load
- Config servers – store the cluster's metadata
  - Mapping of the cluster's data set to the shards
  - Recommended number: 3

Sharding: Diagram



Journaling

- Write operations are applied in memory and into a journal before done in the data files (on disk)
  - To restore consistent state after a hard shutdown
  - Can be switched on/off

- **Journal directory** – holds journal files
- **Journal file** = write-ahead redo logs
  - Append only file
  - Deleted when all the writes are durable
  - When size > 1GB of data, MongoDB creates a new file
    - The size can be modified
- Clean shutdown removes all journal files

Transactions

- Write ops: atomic at the level of single document
  - Including nested documents
  - Sufficient for many cases, but not all
  - When a write operation modifies multiple documents, other operations may interleave

- **Transactions**:
  - Isolation of a write operation that affects multiple docs

```
db.foo.update( { field1 : 1 , $isolated : 1 }, { $inc
: { field2 : 1 } } , { multi: true } )
```

  - Two-phase commit
    - Multi-document updates