

Large Scale Data Management

Document Stores, MongoDB

Silviu Maniu, LIG, Univ. Grenoble Alpes

Data Interchange, JSON

Data Exchange Problem

- we are in an age of **Data**
- types of data: **textual**, **medical**, **legal**, **GPS**, ...
- **computers need to communicate** — data need to be readable by anyone
- how to keep **long-term access to data**?

Data Exchange

- **Why?** To ensure programs read and use data **from other programs**
- **How can we encode data?**

Data Exchange: Unstructured Data (Text/Binary)

```

Theory of Computation
Michael Sipser
Cengage Learning
2012
3

Artificial Intelligence
Stuart Russell
Peter Norvig
Pearson
2013
3
    
```

```

kXv400z0000010401g19075t XAN0
03A+000}k{0c00000;p000sHjPV0.000000S000/%ZE[000
0A0_0A00000100-01+0;01.
FHh0@01IKF
02c000-000x000 00_0}00m.010; r200)BE0000>,E000~
000
0005L0T0:x000;000c: 0100c0
0c00mX00000+0(0%
0000.0n00

osp0;Y-0000-0000Lu<+000v]00X)00]Y0400s00_a0/000+0
H90N0P_000;000H0s;0Xe0-0000N100000n!0000+00<Z
r=00v00y0:00000;p000LTI~^t00;0%:01PLR00
00)t0)00!00
x0]0000h0000040K0Y_0{aE0000-0010000-0_00sxc000!00
0s0C0}+*0-000s00\sa0E0100T400-0VA]r0-10050T1
04p00hC000000aV.000B,CLE-000f=000H0SBMA
000
0{(0000yE I0pt000000L00+00000070G0r q00~00000
0\p]0(02t0000)0+00]v0*.00R4-0000700-0000!0YK]00N
:0:FV000+10000(7N0;07000000000010000h-0-00]000v00
-000000(0m>0 H
    
```

- What is the format? What are the fields?
- Need to send the file and the parser.

Data Exchange: Relational Database

Books

| id | title | editor | year |
|----|-------------------------|---------|------|
| 1 | Theory of Computation | Cengage | 2012 |
| 2 | Artificial Intelligence | Pearson | 2013 |

Authors

| id | first | last |
|----|---------|---------|
| 1 | Sipser | Michael |
| 2 | Russell | Stuart |
| 3 | Norvig | Peter |

BookAuthor

| author | book |
|--------|------|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |

- Fixed schema, strict typing

Data Exchange

- How can we encode data for easy interchange?

Two desiderata:

1. easy to read / parse (exchange)
2. easy to detect the structure (parse)

Semi-Structured Data (SSD)

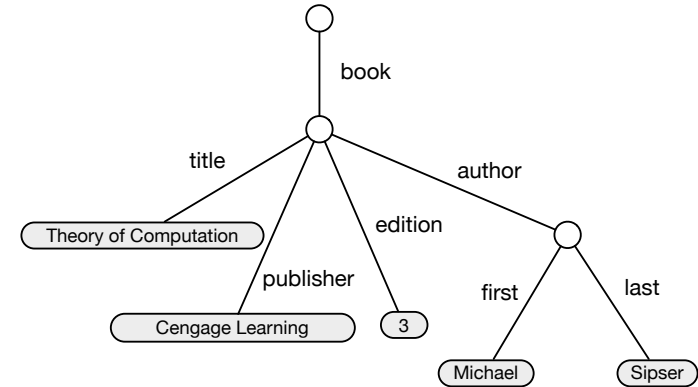
- an attempt to reconcile document view (text, HTML) and the strict structures in DB
- organized in semantic entities: similar entities grouped together, entities in the same group may not have same fields
- defined as (possibly nested) set of field-value pairs
- order of fields not important!

Semi-Structured Data (SSD)

```
{books:{  
  title: "Theory of Computation",  
  edition: 3  
  publisher: "Cengage Learning",  
  author: {first: "Michael", last: "Sipser"}  
},  
...}
```

Semi-Structured Data (SSD)

- We need an **internal representation / data model** — **trees**



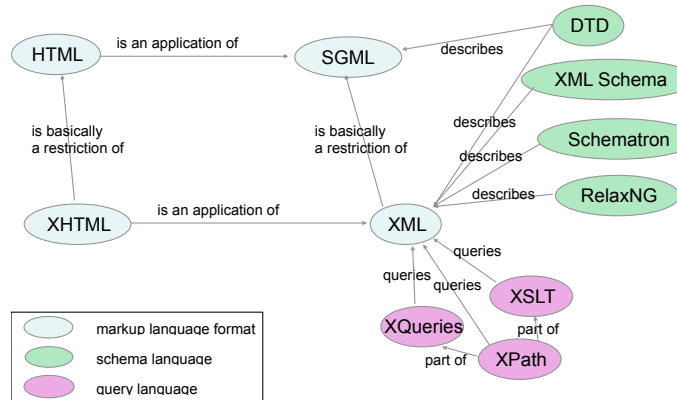
Semi-Structured Data (SSD)

- How to **store / represent**?
- Different formalisms to represent: Object Exchange Model, Lore, XML, JSON

XML: eXtensible Markup Language

- representation of **semi-structured data**, suitable for humans and computers
- is **not for the layout of documents** (HTML, CSS, ...)
- querying still not perfect

XML: The Landscape



Credit: B. Parsia, C. Hedeler, U. Sattler; Univ. of Manchester

XML Example: Open Maps

```

    <?xml version="1.0" encoding="UTF-8"?>
    <osm version="0.14" generator="OsmLogo 0.0.2">
      <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
      <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="46882" visible="true" version="1" changeset="4766636" timestamp="2008-09-21T21:37:45Z"/>
      <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744" visible="true" version="1" changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
      <node id="183881213" version="1" changeset="12370172" lat="54.0906668" lon="12.2539381" user="ladkor" uid="78625" visible="true" timestamp="2012-07-20T09:43:19Z"/>
      <tag k="name" v="Neu Broderstorf"/>
      <tag k="traffic_sign" v="city_limit"/>
    </node>
    ...
    <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="46882" visible="true" version="1" changeset="4766636" timestamp="2008-09-21T21:37:45Z"/>
    <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changeset="4142606" timestamp="2010-03-10T11:47:08Z">
      <nd ref="292403387"/>
      <nd ref="298884289"/>
    </way>
    ...
    <nd ref="261728686"/>
    <tag k="highway" v="unclassified"/>
    <tag k="name" v="Pastover Straße"/>
  </way>
  <relation id="56688" user="xmvnr" uid="56190" visible="true" version="28" changeset="6947637" timestamp="2011-01-12T14:23:49Z">
    <member type="node" ref="294942404" role="" />
    ...
    <member type="node" ref="364933006" role="" />
    <member type="way" ref="4579143" role="" />
  </relation>
  ...
  
```

XML: Full Grammar

```

[1] document ::= prolog element Misc*
[2] Char ::= a Unicode character
[3] S ::= (' ' | '\t' | '\n' | '\r')+
[4] NameChar ::= (Letter | Digit | ':' | '-' | '_' | '.')
[5] Name ::= (Letter | '_' | ':') (NameChar)*

[22] prolog ::= XMLDecl? Misc* (doctypedecl Misc*)?
[23] XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDecl? S? '?'
[24] VersionInfo ::= S' version' Eq('"' VersionNum '"' | "'" VersionNum "'")
[25] Eq ::= S? '=' S?
[26] VersionNum ::= '1.0'

[39] element ::= EmptyElemTag
                | STag content Etag
[40] STag ::= '<' Name (S Attribute)* S? '>'
[41] Attribute ::= Name Eq AttValue
[42] Etag ::= '</' Name S? '>'
[43] content ::= (element | Reference | CharData)*
[44] EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'

[67] Reference ::= EntityRef | CharRef
[68] EntityRef ::= '&' Name ';'
[84] Letter ::= [a-zA-Z]
[88] Digit ::= [0-9]
  
```

JSON

(JavaScript Object Notation)

- another **tree data structure formalism** — simpler than XML
- coming from **JavaScript**
- details at <http://www.json.org/xml.html>

Ex-Twitter API (JSON)

Example Request

```
GET
https://api.twitter.com/1.1/statuses/show.json?
id=210462857140252672
```

Example Result

```
{
  "coordinates": null,
  "source": "Folky",
  "truncated": false,
  "created_at": "Wed Jun 06 20:07:18 +0000 2012",
  "id_str": "210462857140252672",
  "retweeted": {
    "urls": [
      {
        "expanded_url": "https://dev.twitter.com/terms/display-guidelines",
        "url": "https://t.co/8640yYc",
        "indices": [
          76,
          97
        ]
      },
      {
        "display_url": "dev.twitter.com/terms/display-1u2020"
      }
    ],
    "hashtags": [
      {
        "text": "twitterbind",
        "indices": [
          19,

```

JSON Principles

JSON - **fragment of JavaScript** from set of literals called items:

- **atomic**: numbers, bools, strings
- **arrays** (composite): [1, 2, "three", "four"]
- **objects** (composite) – sets, unordered lists, associative arrays: {"first": "Michael", "second": "Sipser"}
- **can be nested**: [{"one":1,"two":2},"name":{"first":"Michael", "second": "Sipser"}]

JSON - XML

```
{ "book": {
  "title": "Artificial Intelligence",
  "publisher": "Pearson",
  "authors": {
    "author": [
      { "first": "Peter", "last": "Norvig" },
      { "first": "Stuart", "last": "Russell" }
    ]
  }
}
}}
```

```
<book title="Artificial
Intelligence" publisher="Pearson">
  <authors>
    <author first="Peter"
last="Norvig" />
    <author first="Stuart"
last="Russell" />
  </authors>
</book>
```

Order of children matters!

JSON Advantages

- **much simpler to understand**: less verbose, more humanly readable
- **simpler processing**: XML needs DOM/SAX, JSON operates on dictionaries / lists
- good for **config files** (name-value pairs)

MongoDB

MongoDB

Generalities

- Developed since 2007, initially for ad backend, now NoSQL document store
- More flexible than relational schema, easy to distribute on the cloud
- Most popular document store, most popular NoSQL db
- Free to use and API connectors

MongoDB

Document store

Document store = records JSON documents into **collections**.

- each doc has a unique field "`_id`", automatically generated = **primary key**
- the docs in a collection **do not always follow the same schema** (schema-less).
- the docs are **stored in MongoDB's binary format** for JSON: BSON

| Relational | MongoDB |
|-----------------|--------------------------------------|
| table | collection |
| line | document |
| column | field |
| join | nested doc, lookup |
| aggregation | aggregation pipeline |
| secondary index | secondary index |
| SQL | calling methods on a document object |

MongoDB

Document insertion

```
> help
> show dbs
> show collections
// See also: use madatabase, db.dropDatabase(), db.movies.drop(),

// insertions, creates collection if needed
> db.movies.insertOne({"nom": "Les 7 Samurais" })

// inserting many docs at once :
> db.movies.insertMany(
  [
    {"nom": "Citizen Kane"},
    {"nom": "The Godfather",
      "acteurs": [
        {"prenom": "Marlon", "nom": "Brando"},
        {"prenom": "Al", "nom": "Pacino"} ]}
  ], {
    writeconcern : { w: "majority", wtimeout: 100 },
    j: true,
    ordered : false
  })
// if all required (a majority) replica nodes have not acknowledged
// within 100ms and been written on journal
```

MongoDB

Document querying

```
> db.movies.find()
// result: ... to indent: db.movies.find().pretty()
// { "_id" : ObjectId("5a982ac71e32c4e0f52641be"), "nom" : "Les 7 Samurais" }
// { "_id" : ObjectId("5a982b3f1e32c4e0f52641bf"), "nom" : "Citizen Kane" }
// { ... }

// selection: (returns all movie docs (complete doc))
> db.movies.find({"nom": "The Godfather"})
> db.movies.find({"acteurs.nom": "Pacino"}).sort({"nom":1}).skip(0).limit(1)
//                                     ↗ -1 for descending

/* db.collection.find(query) */

// Selection operators: regex, in, nin (not in), all, $lt...
> db.movies.find({"nom": {$regex: "[c]", $options: "i"}})
> db.movies.find({"acteurs.nom": {$in: ["Pacino", "Brando"]}})
> db.movies.find({"acteurs.nom": {$nin: ["Pacino", "Brando"]}})
> db.movies.find({"acteurs.nom": {$all: ["Pacino", "Brando"]}})

// boolean combinations of conditions
> db.movies.find({"nom": {$gt: "D", $lt: "M"}})
> db.movies.find({$and: [{"acteurs.nom": "Pacino"}, {"acteurs.nom": "Brando"}]})
> db.movies.find({$or: [{"acteurs.nom": "Pacino"}, {"nom": {$lt: "D"}}]})
> db.movies.find({"acteurs.nom": {$gt: "D", $lt: "C"}})
> db.movies.find({"acteurs.nom": {$elemMatch: {$gt: "D", $lt: "C"}}})

/* db.collection.find(query, projection) */

// projection: field is eluded if 0 or null, returned on other values
// but _id always in unless explicitly out.
// Cannot specify simultaneously in and out (except _id)
> db.movies.find({}, {"acteurs.prenom": 0}) → champs prenom est exclu
> db.movies.find({"acteurs.nom": {$all: ["Pacino", "Brando"]}}, {"acteurs": 1})
```

MongoDB

Aggregation pipeline

```
// aggregation - easy cases:
> db.movies.distinct("nom") // returns: [ "Les 7 Samurais", "Citizen Kane"... ]
> db.movies.count() // returns : 3

// aggregation pipeline : aggregate( [ <step1>, <step2> ... ] )
> db.movies.aggregate(
  [ {
    $group: { "_id": "$acteurs.nom", "nb_films": { $sum: 1 } }
  } ] )

> db.movies.aggregate(
  [
    { $match: { "nom": { $ne: "Citizen Kane"} } }, → match : to filter
    { $group: { "_id": "$acteurs.nom", "nb_films": { $sum: 1 } } }
  ] )
// ... and other operations to transform, sample
```

MongoDB

Joins

Since version 3.2 = implements lookup

- Similar to LEFT OUTER JOIN in SQL

```
> db.movies.aggregate(
  [ {
    $lookup:
      {
        from: "seances", → the other collection we wish to join
        localField: "nom",
        foreignField: "nom_film",
        as: "infos_cine" → field containing the array of "seance objects"
      }
    } ] )
```

MongoDB

Map-Reduce

Slower

```
var mapfunction = function () {
  emit (this._id, 1);
};

var reducefunction = function (key, values) {
  return Array.sum(values);
};

db.movies.mapReduce(
  mapfunction,
  reducefunction,
  { out : "fichier_nb" }
)

db.fichier_nb.find()
```

MongoDB Architecture

Replication

Replication: multiple instances (forming what MongoDB calls a replica set) maintain the same data: 1 primary node (master) receives all operations (esp. writes), and secondary nodes (slaves) copy the primary node's data.

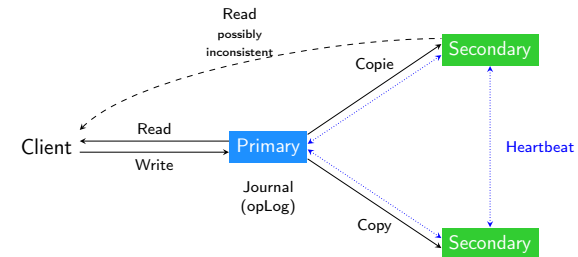
Generally, replication distributes the load (reads). But not in MongoDB with default setting.

Periodically (ex: heartbeat=10s), nodes exchange short messages (heartbeat) to check availability. If primary fails, secondaries elect a new one

MongoDB Architecture

Replication

- opLog (operations log) copied, asynchronous copy



MongoDB Architecture

Sharding

MongoDB Sharding:

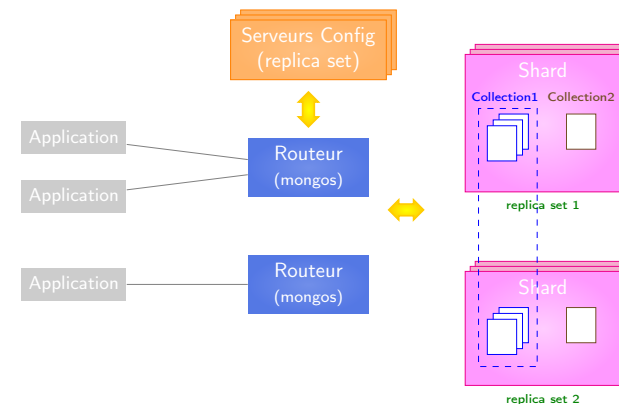
- partitions a collection
- by default, range partitioning
- can use hash partitioning (just hashed through MD5 each key, then interval partitioning on hashes)
- specify partitioning key and nodes (cf architecture): chunk is split if exceeds a given size (=real time), chunks balanced when number chunks too large

Architecture:

- each chunk stored independently on some replica set
- routers called "mongos" distribute queries, merge results

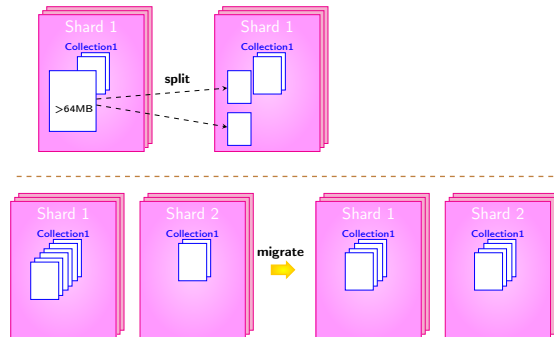
MongoDB Architecture

Sharding



MongoDB Architecture

Sharding: balancing



MongoDB Architecture

Concurrency, Failure Recovery

Operations on a document are **atomic**.

Engine uses **locks** (on collections, db, globally).

Internally, engine uses MVCC and multidoc transactions

Checkpoint every 60s: data saved on disk

Journal : a **write-ahead log** records transactions between 2 checkpoints.

MongoDB Architecture

Indexes

Types of index:

- **simple or composite**: B-trees (can also enforce unicity)
- **(geo)spatial index** (2d) for queries such as : closest points, points in a given area
- **text**: word list, stemming
- **hash**: hashing a field (and its contents)

```
// Composite indexes use lexicographic order
db.movies.createIndex( { "actors.nom": 1, "actors.prenom": -1 } )
```

To Read Further

Specs and documentation

- XML specification: <https://www.w3.org/TR/xml/>
- JSON specification: <https://www.json.org/json-en.html>
- MongoDB: <https://www.mongodb.com/docs/>