Getting Started with Geospatial Data in MongoDB

Buzz Moschetti
Enterprise Architect
buzz.moschetti@mongodb.com
@buzzmoschetti

Agenda

- What is MongoDB?
- What does “geospatial capabilities” mean?
- GeoJSON
- Combining GeoJSON with non-geo data
- APIs and Use Cases
- Comparison to OGC (Open Geospatial Consortium)
- Indexing
- Using Geo Capabilities for non-Geo Things
- Esri and shapefiles

MongoDB:
The Post-Relational General Purpose Database

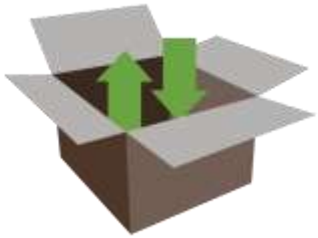
Fully Featured

High Performance

Scalable

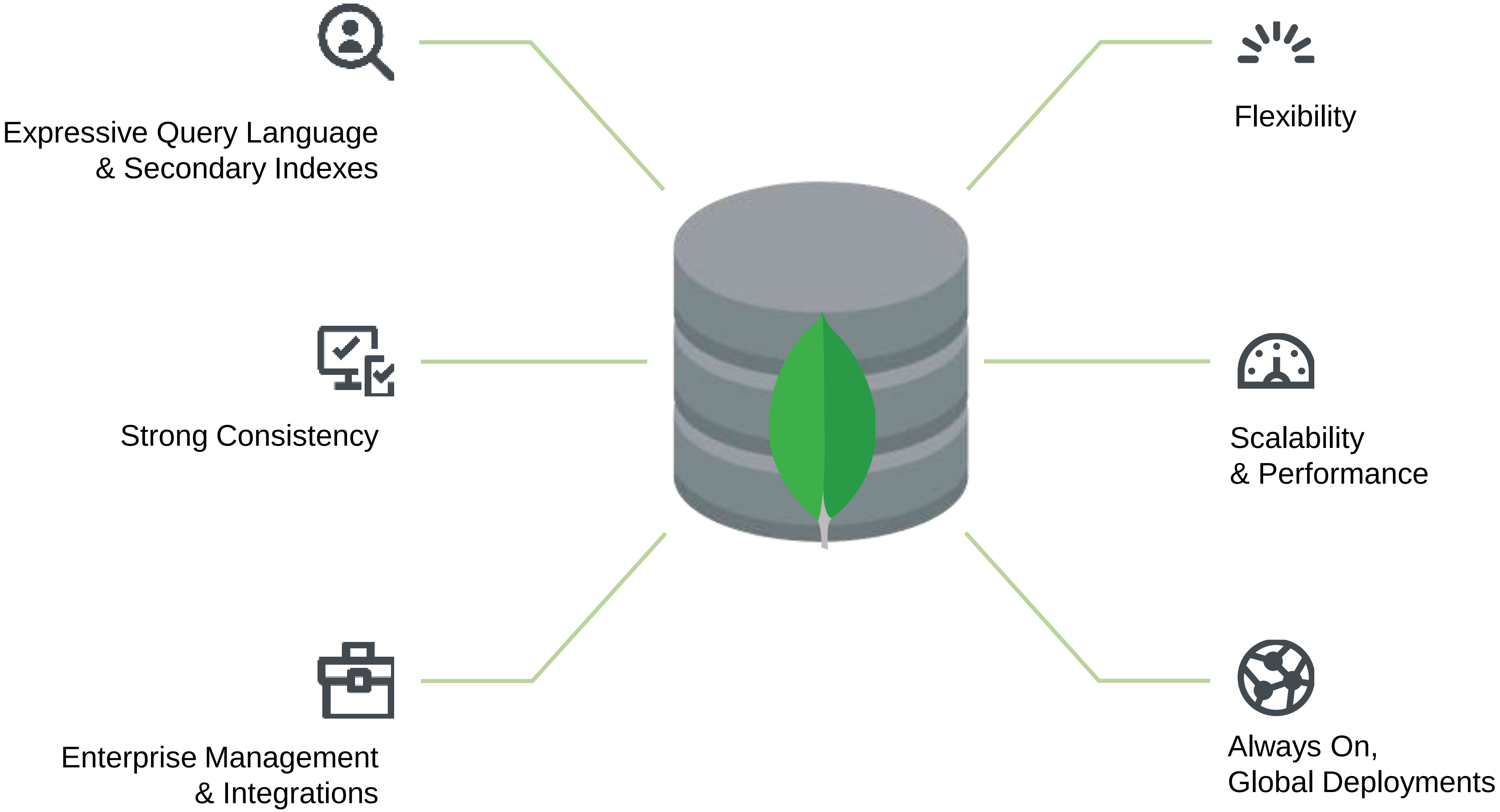
```
{
  name: "John Smith",
  pfxs: ["Dr.","Mr."],
  address: "10 3rd St.",
  phone: {
        home: 1234567890,
        mobile: 1234568138 }
}
```

Document
Data Model



Open-
Source

Nexus Architecture



Expressive Query Language
& Secondary Indexes

Strong Consistency

Enterprise Management
& Integrations

Flexibility

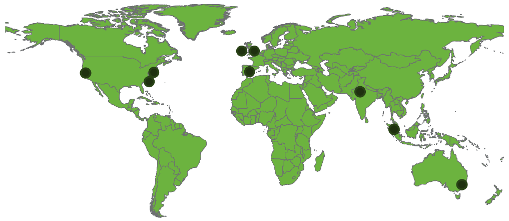
Scalability
& Performance

Always On,
Global Deployments

MongoDB Company Overview

~800 employees

2500+ customers



Offices in NY & Palo Alto and across EMEA, and APAC

NEA

SEQUOIA CAPITAL

intel Capital

union square ventures

T.RowePrice INVEST WITH CONFIDENCE

Fidelity INVESTMENTS

FLYBRIDGE CAPITAL PARTNERS

salesforce

redhat

ALTIMETER

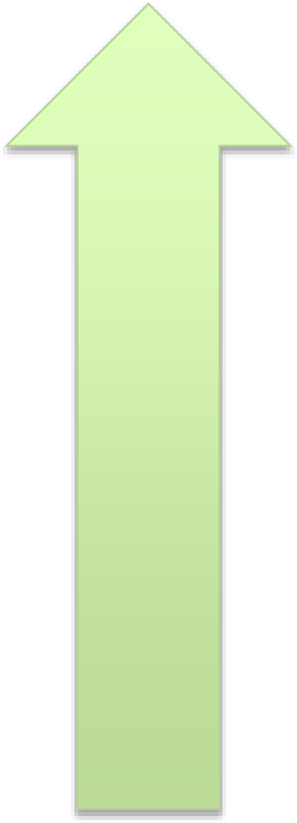
IQT IN-Q-TEL

Over **$311 million** in funding

What is “Geo”?

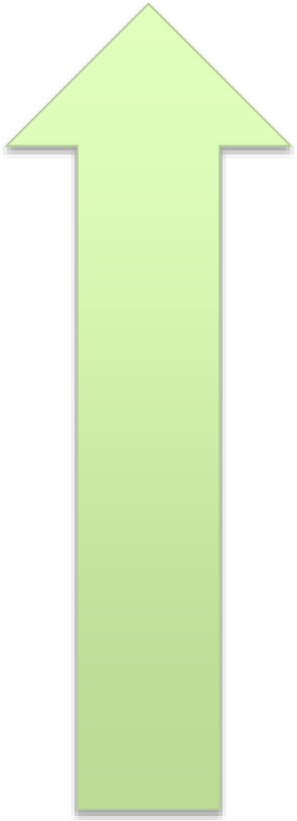
At least **4 levels** of capability

The Geo Stack



Efficiently **store, query, and index** arbitrary points, lines and **polygons** in the DB

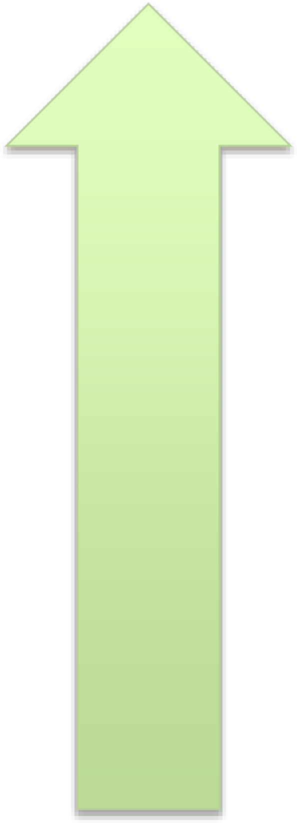
The Geo Stack



Platform for data analysis of **peer data** (trades/house value/population/sales/widgets) grouped by **geo data**

Efficiently **store, query, and index** arbitrary points, lines and polygons in the DB

The Geo Stack

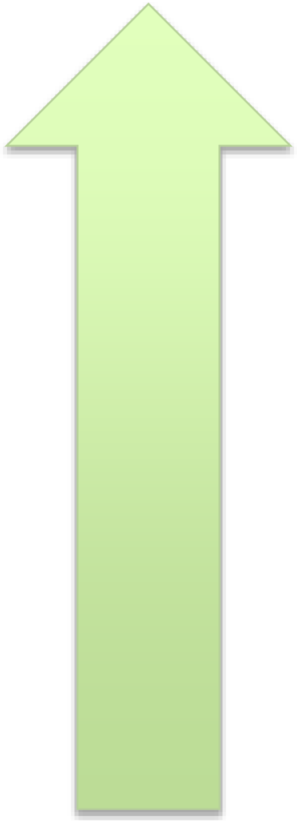


Graphical rendering of geo shapes on a **map**

Platform for data analysis of **peer data** (trades/house value/population/sales/widgets) grouped by geo data

Efficiently **store, query, and index** arbitrary points, lines and polygons in the DB

The Geo Stack



Application(s) to browse and manipulate **all** the data

Graphical rendering of geo shapes on a map

Platform for data analysis of **peer data** (trades/house value/population/sales/widgets) grouped by geo data

Efficiently **store, query, and index** arbitrary points, lines and polygons in the DB

Important: Sometimes there is NO Map

- Geo stack must support geo functions **WITHOUT** a Map
- Offline reporting
- “Nightly fleet management report”
- “Distributor loss by assigned area”
- Compute/analytical processing
- Dynamic polygon generation
- Weather catastrophe simulation
- Other geo-filtering as input to analytics

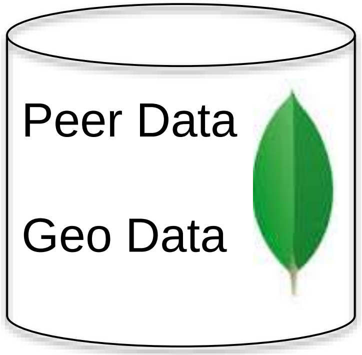
MongoDB: The Data "Base"

Application(s) to browse and visualize

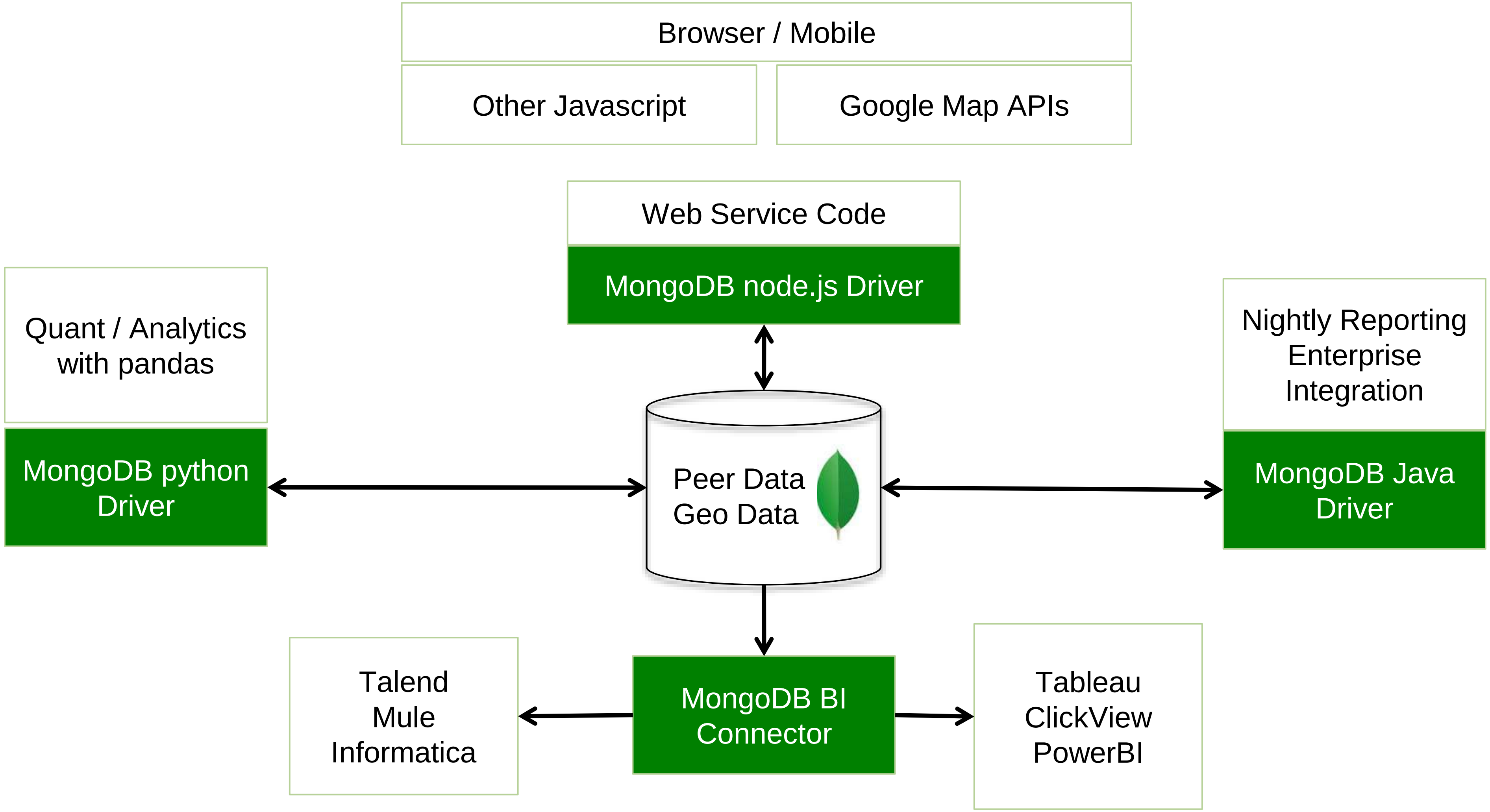
Graphical rendering of geo shapes on a map

Platform for data analysis of **peer data** (trades/value/population/sales/widgets) grouped by geo data

Efficiently **store, query, and index** arbitrary points, lines, and polygons in the DB



One Persistor for All Applications & Use Cases



Browser / Mobile

Other Javascript

Google Map APIs

Web Service Code

MongoDB node.js Driver

Quant / Analytics with pandas

MongoDB python Driver

Peer Data
Geo Data

Nightly Reporting
Enterprise
Integration

MongoDB Java Driver

MongoDB BI Connector

Talend
Mule
Informatica

Tableau
ClickView
PowerBI

Balanced Reporting

- Most other NoSQL DBs **do not** have this capability
- Oracle, Postgres, MySQL, SQLServer **do** offer it and subscribe to Open GeoS Consortium (OGC) standards

MongoDB data model is the major difference

MongoDB: Simple, parse-free, type-correct APIs and data to manipulate and interrogate geo shapes

a.k.a. arrays (of arrays (of arrays))

OpenGIS: Piles of “ST_” functions: http://postgis.net/docs/reference.html#Geometry_Accessors

```
SELECT ST_MakePolygon(
ST_GeomFromText(
'LINESTRING(75.15 29.53,77 29,77.6 29.5, 75.15 29.53)'));
```

Data & APIs

Legacy: 2D points

```
{
  name: {f: “Buzz”, l: “Moschetti”},
  favoritePets: [ “dog”, “cat” ],
  house: [ -95.12345, 43.23423 ]
}
```

Better: GeoJSON

```
{
  name: {f: “Buzz”, l: “Moschetti”},
  favoritePets: [ “dog”, “cat” ],
  house: {
    type: “Point”,
    coordinates: [ -95.12345, 43.23423 ]
  }
}
```

Better: GeoJSON

```
{
  name: “Superfund132”,
  location: {
    type: “Polygon”,
    coordinates: [
     [ [-95.12345, 43.2342],[-95.12456,43.2351],…]
     [ [-92.8381, 43.75], … ]  // “hole”
    ]
  }
}
```

The GeoJSON Family

```
{
    type: “Point”, “MultiPoint”, “LineString”,”MultiLineString”, “Polygon”, “MultiPolygon”
    coordinates: [ specific to type ]
}

{
    type: “GeometryCollection”
    geometries: [
        { type:  (one of above),
          coordinates: [ . . . ]
        }
]
```

NO COMPUTED SHAPES
(Circle, Arc, Box, etc.)

We use the **WGS84** standard:
`http://spatialreference.org/ref/epsg/4326/`

MongoDB Data Types are Geo-friendly

```
var poly = [
  [ [-95.12345,43.2342],[-95.12345,43.2351],
    [-95.12211,43.2351],[-95.12211,43.2342],
    [-95.12345,43.2342]  // close the loop!
  ]
];

db.myCollection.insert(
  {name: {f: "Buzz", l: "Moschetti"},
  favoritePets: ["dog", "cat"],
  geo: { type: "Polygon“, coordinates: poly }
}));
```

… even with Java

```
Document doc = new Document();
doc.put("name", ”Superfund132");

List ring = new ArrayList();
addPoint(ring, -95.12345, 43.2342);
addPoint(ring, -95.12345, 43.2351);
addPoint(ring, -95.12211, 43.2351);
addPoint(ring, -95.12211, 43.2342);
addPoint(ring, -95.12345, 43.2342);

List poly = new ArrayList();
poly.add(ring);
Map mm = new HashMap();
mm.put("type", "Polygon");
mm.put("coordinates", poly);
doc.put("geo", mm);

coll.insertOne(doc);
```

```
static void addPoint(List ll,
                     double lng,
                     double lat) {
  List pt = new ArrayList();
  pt.add(lng);
  pt.add(lat);
  ll.add(pt);
}
```

All Types Are Preserved Correctly

```
Document doc = coll.find().first();
recursiveWalk(doc);

name: java.lang.String: Superfund132
geo: com.mongodb.BasicDBObject
  type: java.lang.String: Polygon
  coordinates: com.mongodb.BasicDBList
    0: com.mongodb.BasicDBList
      0: com.mongodb.BasicDBList
        0: java.lang.Double: -95.12345
        1: java.lang.Double: 43.2342
      1: com.mongodb.BasicDBList
        0: java.lang.Double: -95.12345
        1: java.lang.Double: 43.2351
      2: com.mongodb.BasicDBList
        0: java.lang.Double: -95.12211
        1: java.lang.Double: 43.2351
```

MongoDB Geo is just doubles and Lists!

Comparison to “Good” PostGIS

```
import org.postgis.PGgeometry;  // extended from org.postgresql.util.PGobject

((org.postgresql.Connection)conn).addDataType("geometry","org.postgis.PGgeometry"
)

String sql = "select geom from someTable”;
ResultSet r = stmt.executeQuery(sql);
while( r.next() ) {
  PGgeometry geom = (PGgeometry)r.getObject(1);
  if( geom.getType() = Geometry.POLYGON ) {
    Polygon pl = (Polygon)geom.getGeometry();
    for( int r = 0; r < pl.numRings(); r++) {
      LinearRing rng = pl.getRing(r);
      . . .
   }
}
```

Beware of bespoke types and dependencies!

Comparison to most OpenGIS

```
String sql = "select ST_AsText(geom) from someTable”;
ResultSet r = stmt.executeQuery(sql);
while( r.next() ) {
  String wkt = r.getString(1);

  // wkt is ”POLYGON((0 0,0 1,1 1,1 0,0 0))”

  // http://en.wikipedia.org/wiki/Well-known_text

  // Now we have to parse the string into
  // an array of array of doubles.
  // Don’t want to introduce a 3rd party dependency…
  // So . . .    We write our own parser.
}
```

Checkpoint

We have data in and out of the DB using basic operations (insert and find)

Now we need to make it performant!

Indexing

```
collection.createIndex({loc:"2d"})
```

When to use:

- Your database has legacy location data from MongoDB 2.2 or earlier
- You do not intend to store **any** location data as GeoJSON objects
- “Special Use Cases” e.g. arbitrary two numeric dimension indexing

```
collection.createIndex({loc:"2dsphere"})
```

When to use:

- Supports all GeoJSON objects **and legacy [x,y] pairs**

```
collection.createIndex({loc:"geoHaystack"})
```

When to use:

- Special small area flat (planar) lookup optimization

Indexing

```
collection.createIndex({loc:”2d”})
```

When to use:

- Your database has legacy location data from MongoDB 2.2 or earlier
- You do not intend to store **any** location data as GeoJSON objects
- “Special Use Cases” e.g. arbitrary two numeric dimension indexing

```
collection.createIndex({loc:”2dsphere”})
```

When to use:

- Supports all GeoJSON objects **and legacy [x,y] pairs**

```
collection.createIndex({loc:”geoHaystack”})
```

When to use:

- Special small area flat (planar) lookup optimization

find()/$match and Indexing

Operator	Geometry Arg Type	2d	2dsphere
$geoWithin	$box,$center,$polygon	Y	N
	$geometry: { type, coordinates }	N	Y
	$centerSphere: [[x,y], **radians**]	Y	Y
$geoIntersects	$geometry only	N	Y
$near,$nearSphere	[x,y]	**R**	-
(output sorted by distance)	$geometry: {type, coordinates}	-	**R**
	+ $minDistance	N	Y
	+ $maxDistance	Y	Y

Y = will assist
N = will not assist
R = REQUIRED

Syntax helper:
find(“loc”:{$geoWithin: {$box: [[x0,y0], [x1,y2]]}});
find(“loc”:{$geoWithin: {$geometry: { type: “Polygon”, coordinates: [….] }}});

Aggregation Framework: $geoNear

	Option	2D	2dsphere
$geoNear (output sorted by distance)	near: { type: “Point”, coordinates }	-	R - and spherical:true
	near: [x, y]	R (or)	R
	query: { expression INCL geo find() on previous page **EXCEPT $near**}	N	N

Important Considerations:

1. You can only use $geoNear as the first stage of a pipeline.
2. You must include the distanceField option.
3. The collection **must have only one geospatial index**: one 2d index or one 2dsphere index.
4. You do not need to specify which field in the documents hold the coordinate pair or point. Because $geoNear requires that the collection have a single geospatial index, $geoNear implicitly uses the indexed field.

Y = will assist
N = will not assist
R = REQUIRED

Use Cases

Case #1: Find Things in a Given Area + More

- Docs contain Points (or possibly “small” polygons)
- $geoWithin

```
db.site.aggregate([
    {$match: {  "loc": { $geoWithin: { $geometry:
                   { type: "Polygon", coordinates: [ coords ] }}}
               ,"portfolio_id": portid
               ,“insuredValue“: {$gt: 1000000}
               ,“insuredDate“: {$gt: new ISODate(„2016-01-01“) }}
    ,{$bucket: {groupBy: „$insuredValue“,
        boundaries: [ 1000000, 2000000, 5000000, 10000000,
                      20000000, Infinity] }}
     . . .
```

Case #2: Find Things in an Area Stored in DB

- Get the shape from the “shapes” collection via query:

  ```
  db.shapes.findOne({predicate},{theShape:1});
  ```

- Turn around and query the target collection, e.g. `buildingSites` with shape:

  ```
  db.buildingSites.find({loc:{$geoWithin: theShape}})
  ```

Case #3: Find Things Closest to where I am

```
db.buildingSites.aggregate([{$geoNear: { point … }}]);
```

- Results returned **already in sorted order by closeness**

Case #3.5: Find Things Closest to where I am but within some bounds

-
  ```
  db.buildingSites.aggregate([
  {$geoNear: {
     query: { “loc”: {$geoWithin:
  {$centerSphere: … }} }

  (or)

     query: { “loc”: {$geoWithin: {$geometry:
  GeoJSON }} }
     }}  ])
  ```

When the Database isn’t enough

When the Database isn’t enough

- VERY fast intersection/within for many objects given probes at high velocity (10000s/sec).
- Geo manipulation: unions, deltas, layering
- Dynamic/programmatic geo construction
- Advanced features: smoothing, simplifiers, centroids, …

You Need Three Things

- Basic geo objects
- Geo operators like intersects, within, etc.
- Algos and smoothers, etc.

Don’t forget
The Geo Stack!

com.vividsolutions.jts

```
Map m = (Map) dbo.get("loc");  // get a GeoJSON object from MongoDB
List coords = (List) m.get("coordinates");
List outerRing = (List) coords.get(0); // polygon is array of array of pts

CoordinateSequence pseq = new CoordinateGeoJSONSequence(outerRing, true);
LinearRing outerlr = new LinearRing(pseq, gf);
int numHoles = coords.size() - 1; // -1 to skip outer ring;
LinearRing[] holes = null;
if(numHoles > 0) {
    holes = new LinearRing[numHoles];
    for(int k = 0; k < numHoles; k++) {
        List innerRing = (List) coords.get(k+1); // +1 adj for outer ring
        CoordinateSequence psx = new CoordinateGeoJSONSequence(innerRing, true);
        holes[k] = new LinearRing(psx, gf);
    }
}
Polygon poly1 = new Polygon(outerlr, holes, gf); // ok if holes was null
Point pt1 = new Point(pseq2, gf);
boolean a = pt1.intersects(poly1);
Geometry simplified = TopologyPreservingSimplifier.simplify(poly1, tolerance);
```

The Ecosystem

- OpenGeo runs over MongoDB!
  http://suite.opengeo.org/docs/latest/dataadmin/mongodb/index.html

- **BoundlessGeo**: Commerical support for OpenGeo over MongoDB
  * Provides top 2 tiers (viz, analysis)
  * https://boundlessgeo.com

Geo Capabilities beyond “Simple Geo”

Geo as Date Range

```
{
        who: ‘john’
        where: ‘mongodb’
        what: ‘lightning talk’
        start:    ISODate(“2016-06-30T15:00:00”)
        end:      ISODate(“2016-06-30T15:05:00”)
}
```

<u>What events were happening at 15:03?</u>

```
collection.find({
    start : { $lte:ISODate(“2016-06-30T15:05:03”)},
    end:    { $gte:ISODate(“2016-06-30T15:05:03”)}
})
```

Geo as Date Range

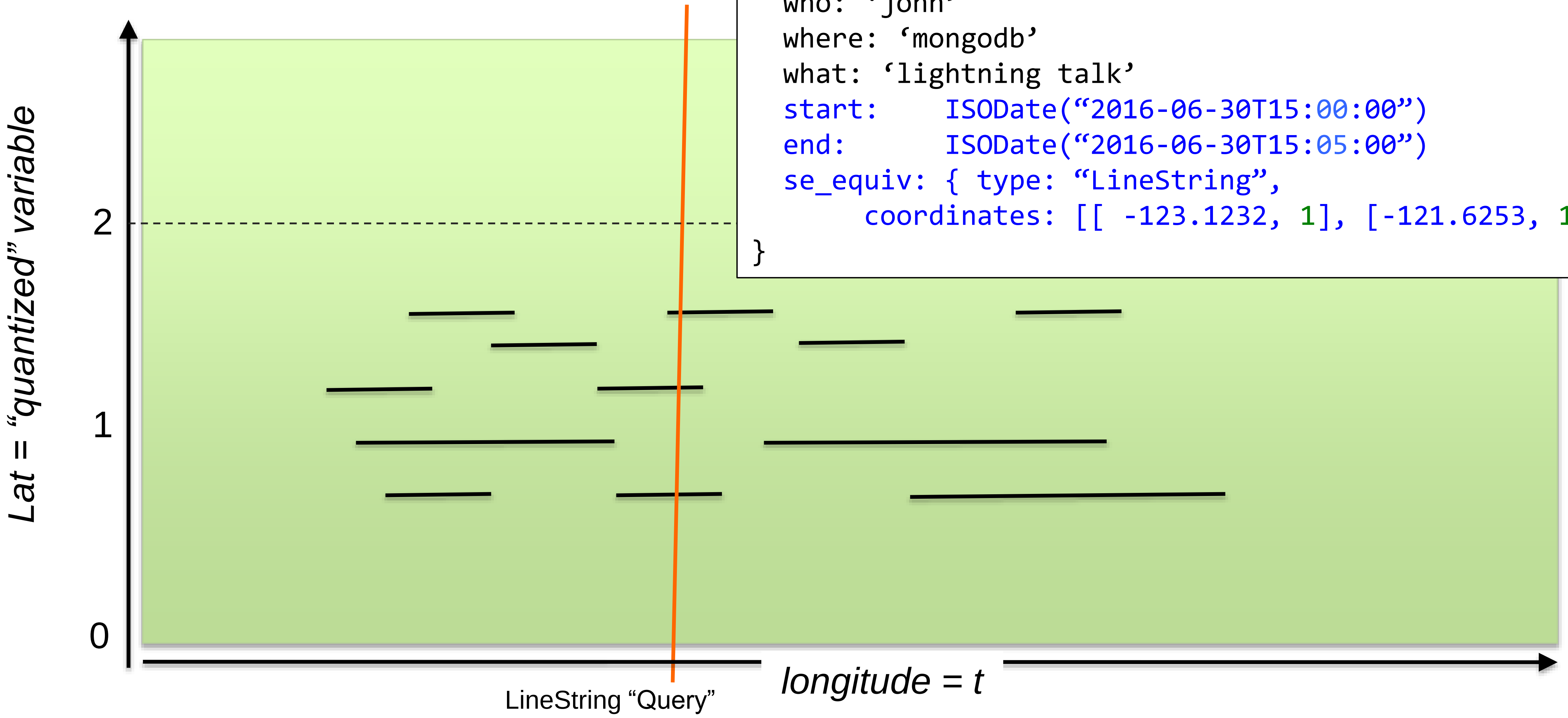
- Ranges on 2 attributes – Two BTree walks (intersection)
- Assuming time can be anywhere in range of records, index walk is average 50% of index
- Test: Macbook Pro, i5, 16GB RAM, data fits in WT Cache easily. Warmed up. Average of 100 runs

694ms /query using index

487ms /query – COLLSCAN!

MUST
DO
BETTER!

(StartDate,EndDate) → Range Type using Geo



```
{
  who: ‘john’
  where: ‘mongodb’
  what: ‘lightning talk’
  start:     ISODate(“2016-06-30T15:00:00”)
  end:       ISODate(“2016-06-30T15:05:00”)
  se_equiv: { type: “LineString”,
       coordinates: [[ -123.1232, 1], [-121.6253, 1]]
}
```

Over 10X performance improvement!

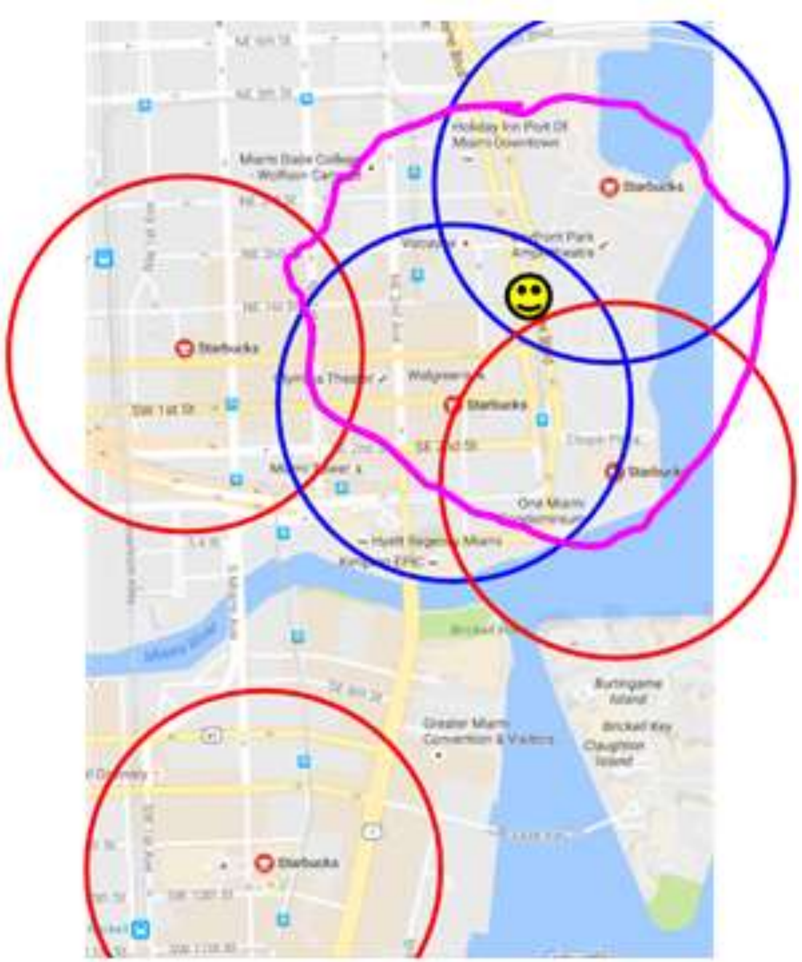
```
start2 = (((start / yearsecs) - 45) *90) – 90
end2 = (((end / yearsecs) - 45) *90) – 90
event = { type: "LineString", coordinates: [ [ start2, 1 ], [end2, 1 ] ] }

// dx = is the LineString “query”
query = {
  g: {
     $geoIntersects: {
        $geometry: { type: "LineString",
                     coordinates: [ [ dx, 0 ], [dx, 2 ] ] }
     }
  }
}
```

45ms!

☺

Mr. Smiley

```
//  Assume Mr. Smiley has these params:
var mslng = -90.00;
var mslat = 42.00;
var msr   = 0.005;  // ~1500 ft radius around him
```



```
db.participatingVendors.aggregate([

// Stage 1: The Bounding Box
{$match: { "loc": { $geoWithin: { $geometry:
          { type: "Polygon", coordinates: mapBoundingBox}}}
    }}

// Stage 2: Compute distance from Mr. Smiley to the points: Pythagorean theorem:
,{$addFields: {dist: {$sqrt: {$add: [
{$pow:[ {$subtract: [ {$arrayElemAt:[ "$loc.coordinates",0]}, mslng]} , 2]}
,{$pow:[ {$subtract:[ {$arrayElemAt:[ "$loc.coordinates", 1]}, mslat]} , 2]}
                         ]}
                 }}}

// Stage 3: If the distance between points is LTE the sum of the radii, then
// Mr. Smiley's circle intersects that of the participant:
// Project 0 (no touch) or 1 (touch)
,{$addFields: {"touch": {$cond: [
     {$lte: [ {$add: [ "$promoRadius", msr ]}, "$dist" ]}, 0, 1
                              ]}}}

,{$match: {"touch": 1}}
                    ]);
```

The Pusher

Don’t forget
$slice!

```
var pts = [ [-74.01,40.70], [-73.99,40.71], . . .];

db.foo.insert({_id:1,
       loc: { type:"LineString", coordinates: [ pts[0], pts[1] ]}});

// Push points onto LineString to "extend it” in an index optimized way!
for(i = 2; i < pts.length; i++) {
   db.foo.update({_id:1},{$push: {"loc.coordinates”: pts[i]}});

   // Perform other functions, e.g.
   c=db.foo.find({loc: {$geoIntersects:
               {$geometry: { type: ”Polygon", coordinates: … } } });
}
```

Perimeter of Simple Polygon

```
> db.foo.insert({_id:1, "poly": [ [0,0], [2,12], [4,0], [2,5], [0,0] ] });
> db.foo.insert({_id:2, "poly": [ [2,2], [5,8],  [6,0], [3,1], [2,2] ] });

> db.foo.aggregate([
 {$project: {"conv": {$map: { input: "$poly", as: "z", in: {
                    x: {$arrayElemAt: ["$$z”,0]},  y: {$arrayElemAt: ["$$z”,1]}
                    ,len: {$literal: 0}  }}}}}
,{$addFields: {first: {$arrayElemAt: [ "$conv", 0 ]} }}
,{$project: {"qqq":
    {$reduce: { input: "$conv",  initialValue: "$first",  in: {
                x: "$$this.x”, y: "$$this.y"
                ,len: {$add: ["$$value.len",  // len = oldlen + newLen
               {$sqrt: {$add: [
                          {$pow:[ {$subtract:["$$value.x","$$this.x"]}, 2]}
                          ,{$pow:[ {$subtract:["$$value.y","$$this.y"]}, 2]}
                         ] }} ] } }}
,{$project: {"len": "$qqq.len"}}

{ "_id" : 1, “len" : 35.10137973546188 }
{ "_id" : 2, "len" : 19.346952903339393 }
```

Geospatial = 2D Numeric Indexable Space

Find all branches close to my location:

```
target = [ someLatitude, someLongitude ];
radians = 10 / 3963.2;     // 10 miles
db.coll.find({"location": { $geoWithin :
            { $center : [ target, radians ] }}}});
```

Find nearest investments on efficient frontier:

```
target = [ risk, reward ];
closeness = someFunction(risk, reward);
db.coll.find({”investmentValue": { $geoWithin :
            { $center : [ target, closeness]}}}});
```

Basic Tips & Tricks

- We use [long,lat], not [lat,long] like Google Maps
- Use 2dsphere for geo; avoid legacy $box/$circle/$polygon
- Use 2d for true 2d numeric hacks
- 5 digits beyond decimal is accurate to 1.1m:
- var loc = [-92.12345, 42.56789] // FINE
- var loc = [-92.123459261145, 42.567891378902] // ABSURD
- $geoWithin and $geoIntersects do not REQUIRE index

- Remember to close loops (it’s GeoJSON!)

esri-related Tips & Tricks

- Shapefiles are **everywhere**; google shapefile <whatever>
- Crack shapefiles to GeoJSON with python pyshp module

```
import shapefile
import sys
from json import dumps

reader = shapefile.Reader(sys.argv[1])

field_names = [field[0] for field in reader.fields[1:] ]
buffer = []

for sr in reader.shapeRecords():
    buffer.append(dict(geometry=sr.shape.__geo_interface__,
                       properties=dict(zip(field_names, sr.record))))

sys.stdout.write(dumps({"type": "FeatureCollection”, "features": buffer},
indent=2) + "\n”)
```

Q & A

mongoDB®

Thank You!

Buzz Moschetti
Enterprise Architect
buzz.moschetti@mongodb.com
@buzzmoschetti

Agenda

- What does geospatial capabilities mean?
- The "levels": DB with geo types, rendering, analytics
- MongoDB brings together geo AND non-geo data

- Geo Data model
- GeoJSON
- Combining GeoJSON with non-geo data

- APIs and Use Cases
- Looking up things contained in a polygon
- Finding things near a point
- Summary of geo ops e.g. $center
- $geoNear and the agg framework
- The power of the document model and MongoDB APIs
- Arrays and rich shapes as first class types
- Comparison to Postgres (PostGIS)

\

Indexing

- Geospatial queries do not necessarily require an index
- 2d vs. 2dsphere

- Geo stacks and the Ecosystem
- MongoDB and OpenGIS and OpenGEO
- Google Maps
- MEAN

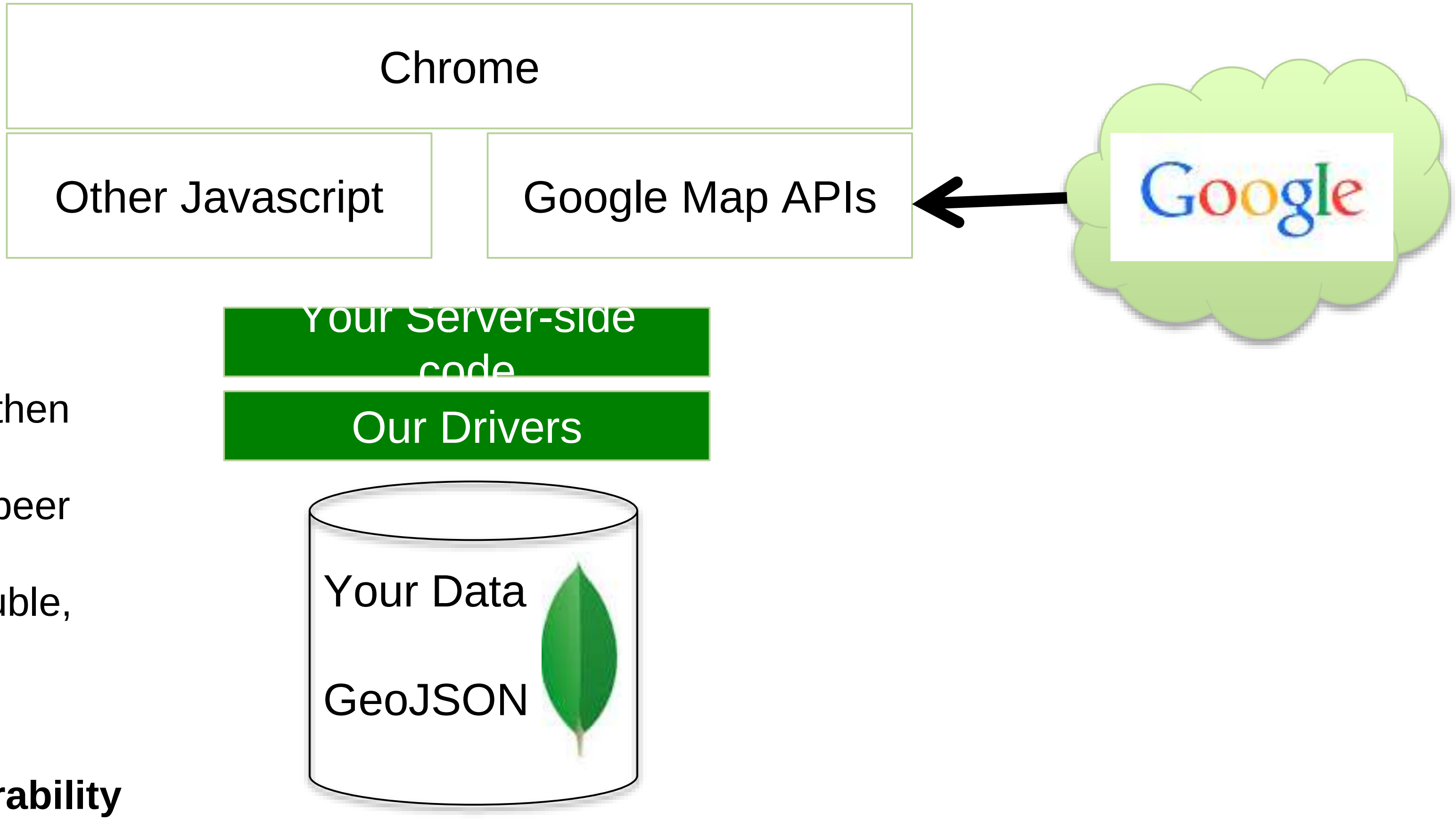
- A Sampling of Geo Solutions
- Mr. Smiley, etc.

- Integration with esri and shapefiles
- esri shapefile cracking

Clever Hacks

- John Page date thing
- Mr. Smiley
- Wildfire
- Push pts on a LineString and check for intersects
- Perimeter & Area of simple polygon
- makeNgon

MongoDB handles your data + geo
Google handles the maps



- Organization unit is document, then collection
- Geo data can contain arbitrary peer data or higher scope within doc
- Proper handling of int, long, double, and decimal128
- Dates are REAL datetimes

- **Uniform indexability and querability across geo and "regular" data**