

CS460: Intro to Database Systems

Database Systems and Beyond

Instructor: Manos Athanassoulis

<https://midas.bu.edu/classes/CS460/>

Database Systems

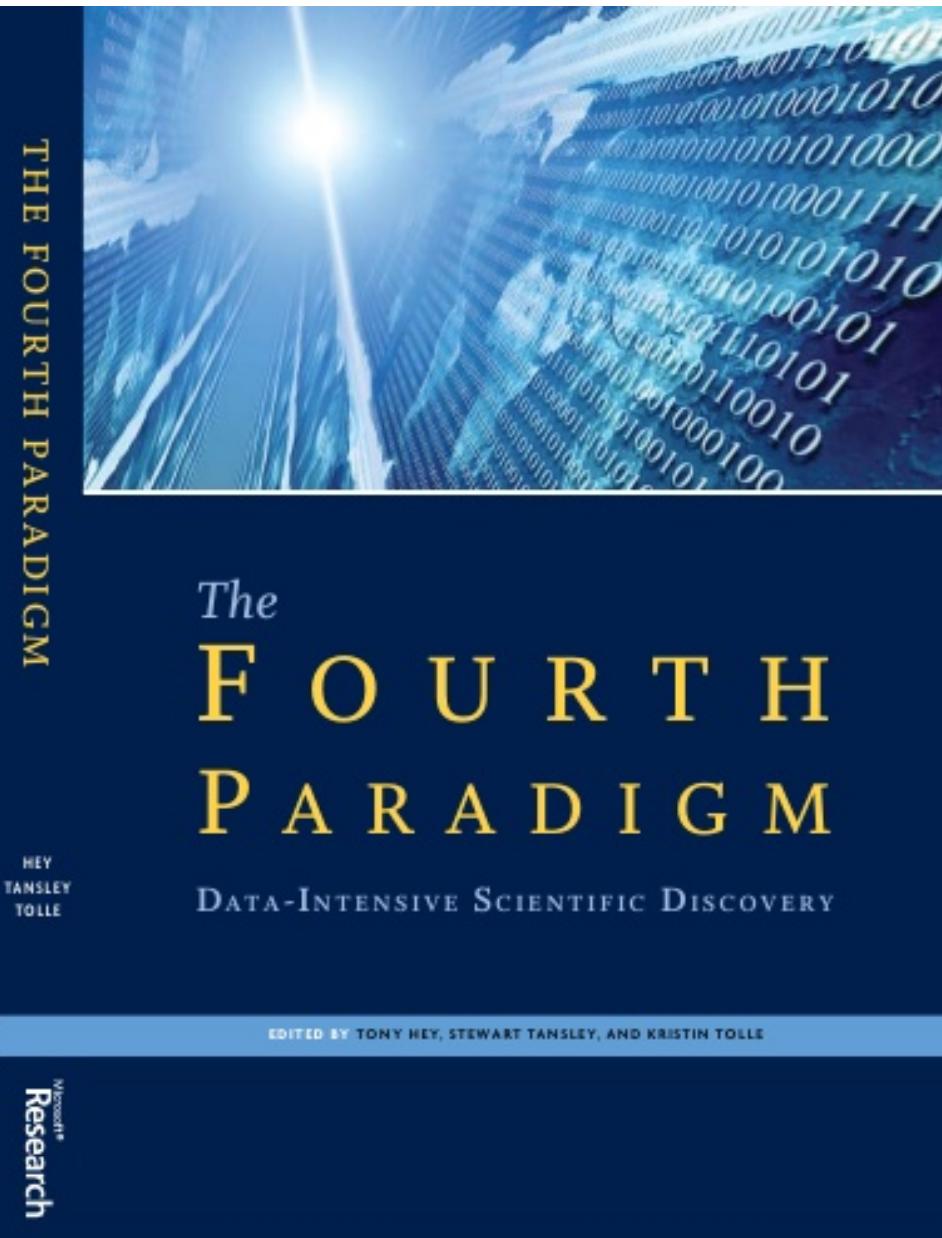
we spent a whole semester on Database Systems
what is next?

what can we do with data?

data-driven science

data-driven discovery

data-driven governance



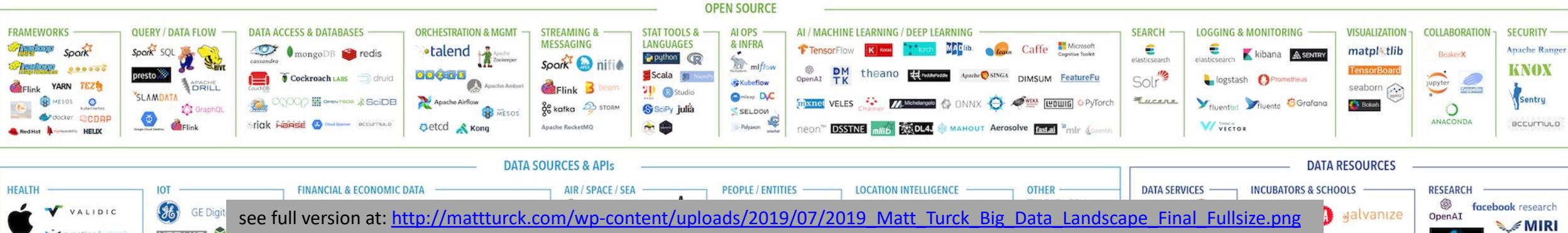
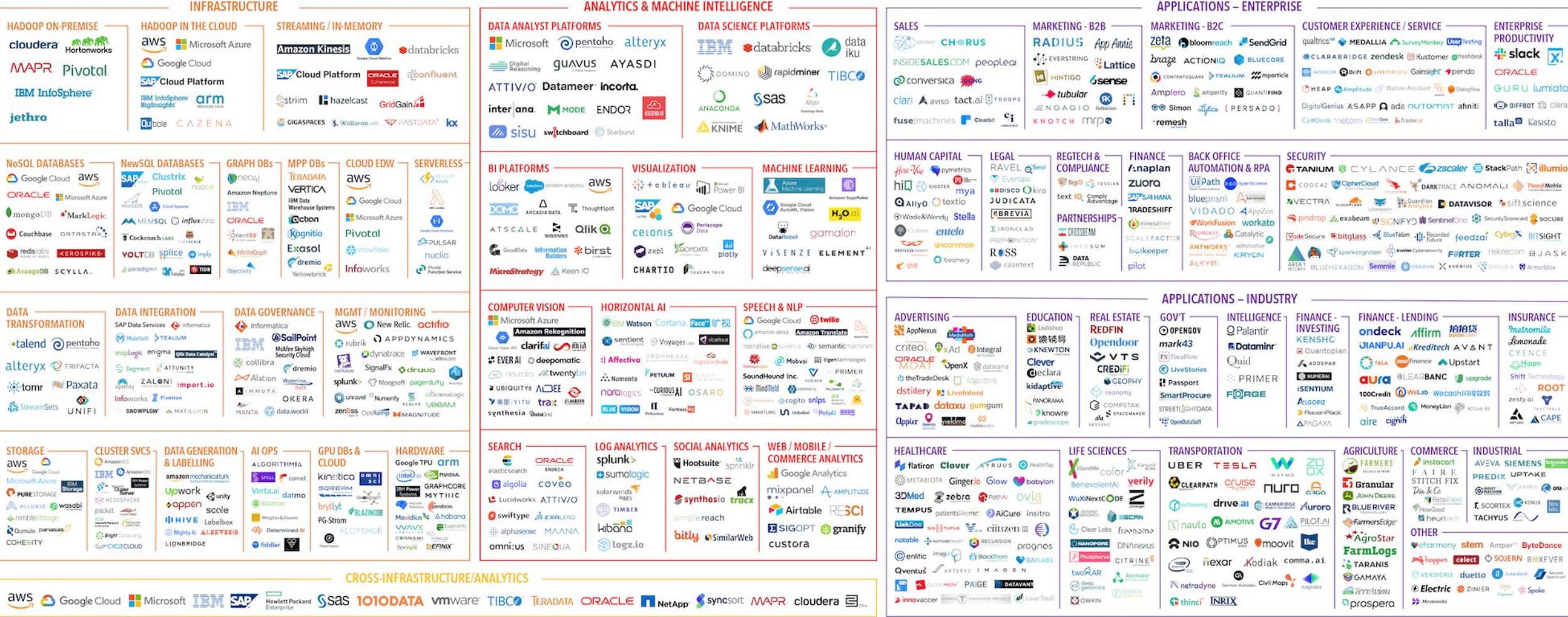
*"Experimental, theoretical, and computational science are all being affected by the data deluge, and a fourth, '**data-intensive**' science paradigm is emerging.*

*The goal is to have a world in which all of the science literature is online, all of the science data is online, and they interoperate with each other. **Lots of new tools are needed to make this happen.***

Faster Innovation through Data-Intensive Approaches

Need for Innovation in Data Management!

DATA & AI LANDSCAPE 2019



increase throughput by parallelization

“scale-up”

use more powerful machines (>#CPUs, >RAM)

“scale-out”

use more machines

Scale Up Execution

how to use more cores (threads)?

inter-query parallelism

each query runs on one processor

inter-operator parallelism

each query runs on multiple processors
an operator runs on one processor

intra-operator parallelism

An operator runs on multiple processors

Scale Up Storage

needs more disks!

how to distribute data?

block partition

hash partition

range partition

how to distribute data accesses?

Scale Out

similar questions across machines

new bottlenecks?

move data across machines: network!

DATA & AI LANDSCAPE 2019



Versatile and popular infrastructure: NoSQL stores

see full version at: http://mattturck.com/wp-content/uploads/2019/07/2019_Matt_Turck_Big_Data_Landscape_Final_Fullsize.png

diving into the internals of modern data systems

cutting-edge designs / *research* projects / *engineering* projects

CS 591: Data Systems Architectures

Spring 2020

Today: more discussion on
simplified and efficient storage models
“NoSQL stores”

What is NoSQL?

HOW TO WRITE A CV

from "Geek and Poke"



Leverage the NoSQL boom

What is NoSQL?

An emerging “movement” around non-relational software for Big Data

Roots are in the Google and Amazon homegrown software stacks

Wikipedia: “A NoSQL database provides a mechanism for storage and retrieval of data that use looser consistency models than traditional [relational databases](#) in order to achieve [horizontal scaling](#) and higher availability. Some authors refer to them as "Not only SQL" to emphasize that some NoSQL systems do allow [SQL](#)-like query language to be used.”

NoSQL Stores

offer an easy to program storage model

simplification of relational

two attributes (a key and a value)

value has variable size

NoSQL features

Scalability is crucial!

- load increased rapidly for many applications

Large servers are expensive

Solution: use clusters of small commodity machines

- need to partition the data and use replication (sharding)
- cheap (usually open source!)
- cloud-based storage

NoSQL features

Sometimes not a well defined schema

Allow for semi-structured data

- still need to provide ways to query efficiently (use of index methods)
- need to express specific types of queries easily

Scalability

Often cited as the main reason for moving from DB technology to NoSQL

DB Position: there is no reason a parallel DBMS cannot scale to 1000's of nodes

NoSQL Position: a) Prove it; b) it will cost too much anyway

Flavors of NoSQL

Four main types:

- key-value stores
- document databases
- column-family (aka big-table) stores
- graph databases

Here we will talk more about “Document” databases (MongoDB)

Key-Value Stores

There are many systems like that:

Redis, MemcacheDB, Amazon's DynamoDB, Voldemort

Simple data model: key/value pairs

the DBMS does not attempt to interpret the value

Queries are limited to query by key

- get/put/update/delete a key/value pair
- iterate over key/value pairs

Document Databases

Examples include:

MongoDB, CouchDB, Terrastore

Special type of key/value that value is a document.

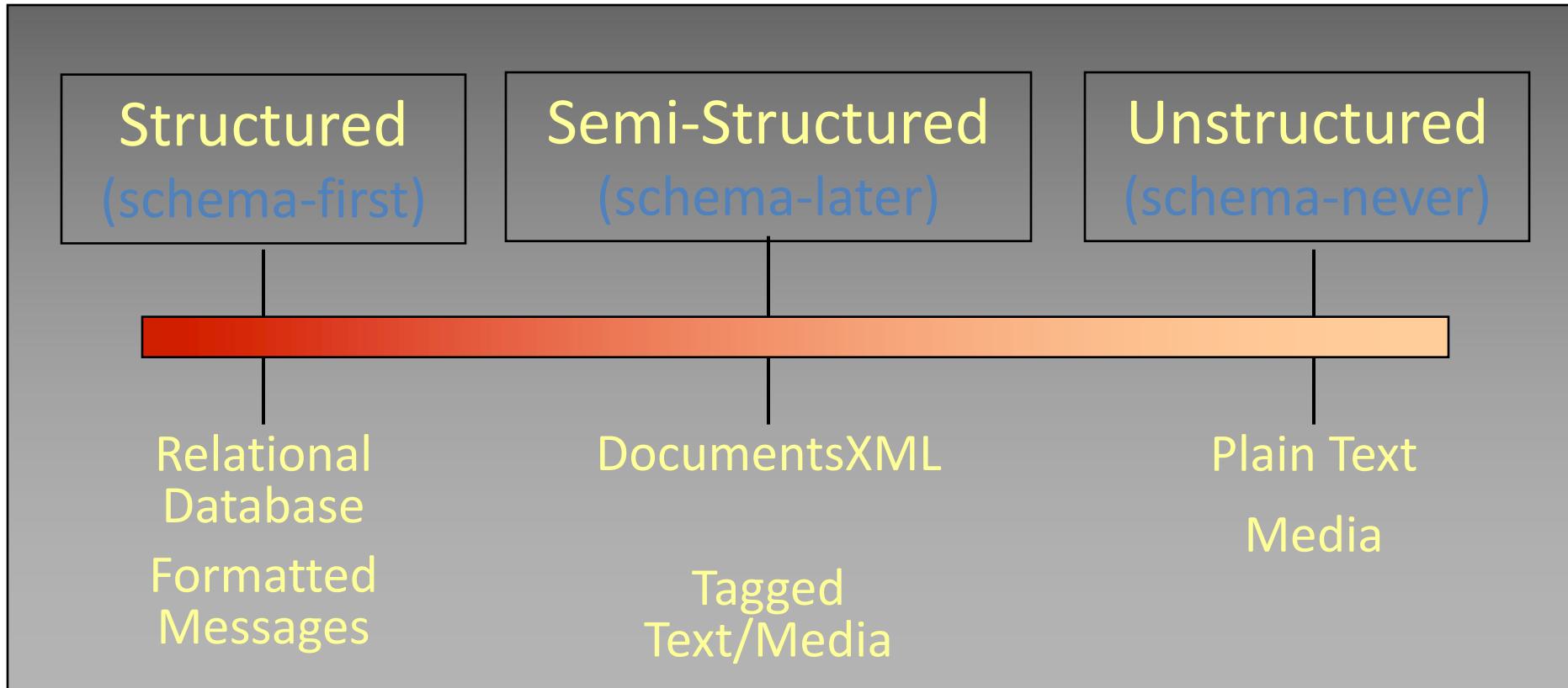
- use some sort of semi-structured data model: XML/JSON
- the value can be examined and used by the system (unlike in key/data stores)

Queries based on key (as in key/value stores), but also on the document (value).

Here again, there is support for sharding and replication.

- the sharding can be based on values within the document

The Structure Spectrum



MongoDB (An example of a Document Database)

Data are organized in collections. A collection stores a set of documents.

Collection (like table) and document (like record)

- but: each document can have a different set of attributes even in the same collection
- Semi-structured schema!

Only requirement: every document should have an “_id” field

- **humongous** => Mongo

Example mongodb

```
{  "_id":ObjectId("4efa8d2b7d284dad101e4bc9"),  
  "Last Name": " Cousteau",  
  "First Name": " Jacques-Yves",  
  "Date of Birth": "06-1-1910" },
```

```
{  "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),  
  "Last Name": "PELLERIN",  
  "First Name": "Franck",  
  "Date of Birth": "09-19-1983",  
  "Address": "1 chemin des Loges",  
  "City": "VERSAILLES" }
```

Example Document Database: MongoDB

Key features include:

JSON-style documents

- actually uses BSON (JSON's binary format)

replication for high availability

auto-sharding for scalability

document-based queries

can create an index on any attribute for faster reads

under the hood, a simple key-value store called WiredTiger!
design based on LSM-trees

MongoDB Terminology

relational term <==> MongoDB equivalent

database <==> database

table <==> collection

row <==> document

attributes <==> fields (field-name:value pairs)

primary key <==> the `_id` field, which is the key associated with the document

JSON

JSON is an alternative data model for semi-structured data

- JavaScript Object Notation

Built on two key structures:

- an object, which is a sequence of name/value pairs
`{ "_id": "1000", "name": "Sanders Theatre", "capacity": 1000 }`
- an array of values ["123", "222", "333"]

A value can be:

- an atomic value: string, number, true, false, null
- an object
- an array

The `_id` Field

Every MongoDB document must have an `_id` field.

- its value must be unique within the collection

- acts as the primary key of the collection

- it is the key in the key/value pair

If you create a document without an `_id` field:

- MongoDB adds the field for you

- assigns it a unique BSON (binary JSON) ObjectID

- example from the MongoDB shell:

```
> db.test.save({ rating: "PG-13" })
> db.test.find() { "_id" : ObjectId("528bf38ce6d3df97b49a0569") , "rating" : "PG-13" }
```

Note: quoting field names is optional (see rating above)

Capturing Relationships in MongoDB

Two options:

1. store references to other documents using their `_id` values
2. embed documents within other documents

Example relationships

```
{
  "_id": ObjectId("52ffc33cd85242f436000001"),
  "name": "Tom Hanks",
  "contact": "987654321",
  "dob": "01-01-1991"
}
```

Here is an example of embedded relationship:

```
{
  "_id": ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address": [
    {
      "building": "22 A, Indiana Apt",
      "pincode": 123456,
      "city": "Los Angeles",
      "state": "California"
    },
    {
      "building": "170 A, Acropolis Apt",
      "pincode": 456789,
      "city": "Chicago",
      "state": "Illinois"
    }
  ]
}
```

```
{
  "_id": ObjectId("52ffc4a5d85242602e000000"),
  "building": "22 A, Indiana Apt",
  "pincode": 123456,
  "city": "Los Angeles",
  "state": "California"
}
```

And here an example of reference based

```
{
  "_id": ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

Queries in MongoDB

Each query can only access a single collection of documents.

Use a method called

```
> db.collection.find(<selection>, <projection>)
```

Example: find the names of all R-rated movies:

```
> db.movies.find({ rating: 'R' }, { name: 1 })
```

Projection

Specify the name of the fields that you want in the output with 1 (0 hides the value)

Example:

```
> db.movies.find( {}, { "title":1, _id:0 } )
```

(will report the title but not the id)

Selection

You can specify the condition on the corresponding attributes using the find:

```
> db.movies.find({ rating: "R", year: 2000 }, { name: 1, runtime: 1 })
```

Operators for other types of comparisons:

MongoDB	SQL equivalent
\$gt, \$gte	>, >=
\$lt, \$lte	<, <=
\$ne	!=

Example: find the names of movies with an earnings <= 200000

```
> db.movies.find({ earnings: { $lte: 200000 } })
```

For logical operators \$and, \$or, \$nor

use an array of conditions and apply the logical operator among the array conditions:

```
> db.movies.find({ $or: [ { rating: "R" }, { rating: "PG-13" } ] })
```

Aggregation

Recall the aggregate operators in SQL: AVG(), SUM(), etc.

More generally, aggregation involves computing a result from a collection of data.

MongoDB supports several approaches to aggregation:

- single-purpose aggregation methods
- an aggregation pipeline
- map-reduce

Aggregation pipelines are more flexible and useful (see next):

<https://docs.mongodb.com/manual/core/aggregation-pipeline/>

Simple Aggregations

db.collection.count(<selection>)

returns the number of documents in the collection
that satisfy the specified selection document

Example: how many R-rated movies are shorter than 90 minutes?

```
> db.movies.count({ rating: "R", runtime: { $lt: 90 } })
```

db.collection.distinct(<field>, <selection>)

returns an array with the distinct values of the specified field
in documents that satisfy the specified selection document
if omit the query, get all distinct values of that field

Example: which actors have been in one or more of the top 10 grossing movies?

```
> db.movies.distinct("actors.name", { earnings_rank: { $lte: 10 } })
```

Aggregation Pipeline

A very powerful approach to write queries in MongoDB is to use pipelines.

We execute the query in stages.

Every stage gets as input some documents, applies filters/aggregations/projections and outputs some new documents.

These documents are the input to the next stage (next operator) and so on

Similar to a traditional query plan. But always with one child (no joins!)

Aggregation Pipeline example

Example for the zipcodes database:

```
> db.zipcodes.aggregate( [  
  { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },  
  { $match: { totalPop: { $gte: 10*1000*1000 } } }  
] )
```

```
{  
  "_id": "10280",  
  "city": "NEW YORK",  
  "state": "NY",  
  "pop": 5574,  
  "loc": [  
    -74.016323,  
    40.710537  
  ]  
}
```

Here we use group_by to group documents per state, compute sum of population and output documents with _id, totalPop (_id has the name of the state). The next stage finds a match for all states the have more than 10M population and outputs the state and total population.

More here: <https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/>

continued:

Output example:

```
{  
  "_id" : "AK",  
  "totalPop" : 550043  
}
```

In SQL:

```
SELECT state, SUM(pop) AS totalPop  
FROM zipcodes  
GROUP BY state  
HAVING totalPop >= (10*1000*1000)
```

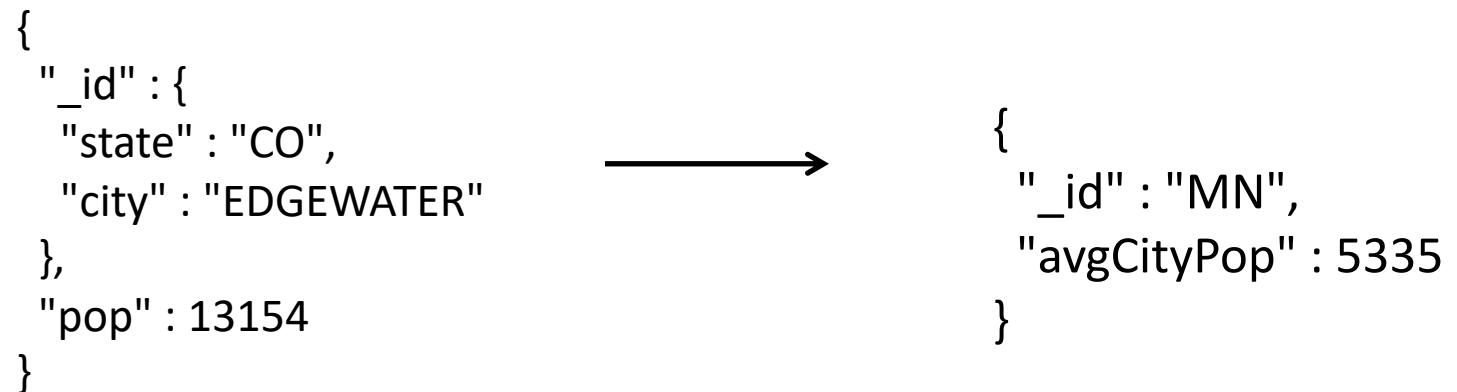
```
db.zipcodes.aggregate( [  
  { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },  
  { $match: { totalPop: { $gte: 10*1000*1000 } } }  
] )
```

more examples:

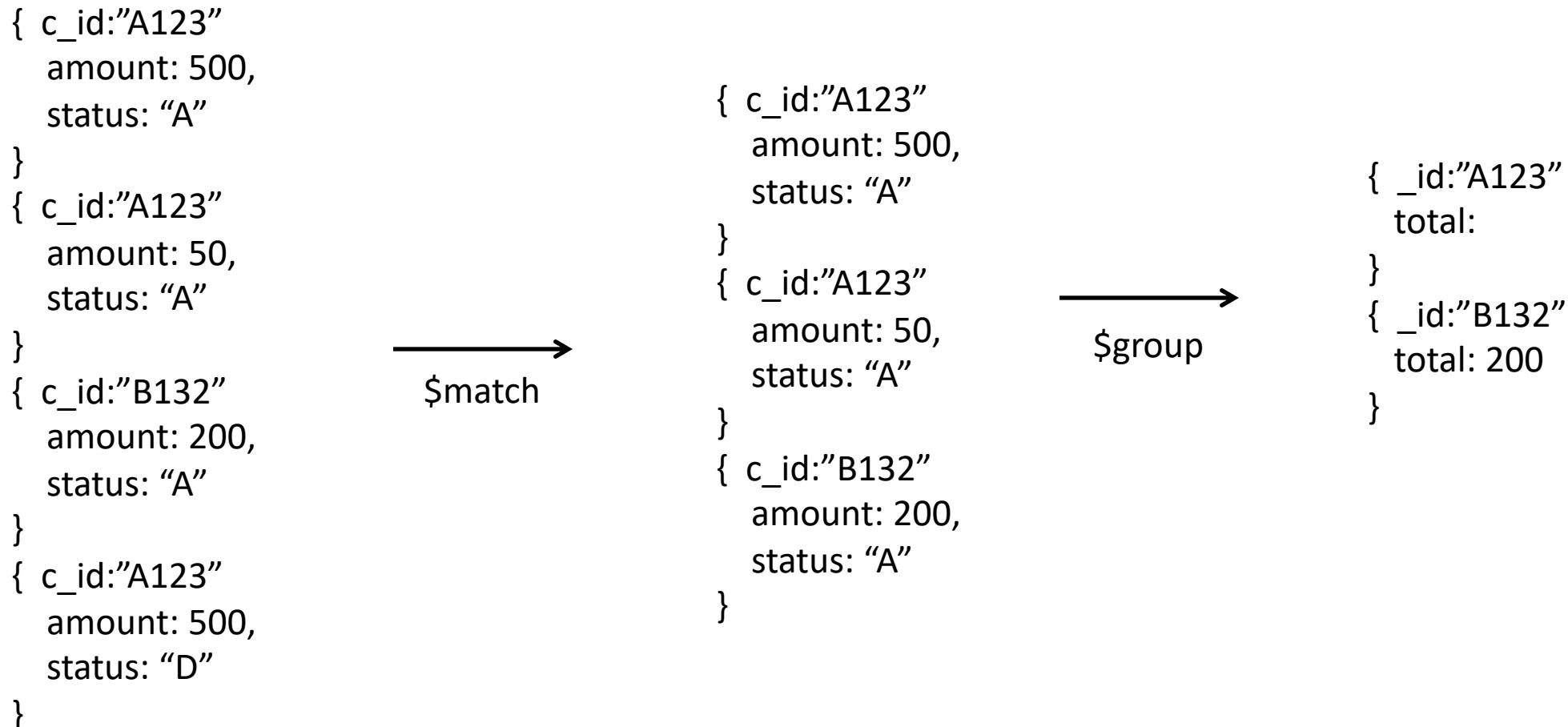
```
db.zipcodes.aggregate( [  
    { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },  
    { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }  
] )
```

What we compute here?

First we get groups by city and state and for each group we compute the population.
Then we get groups by state and compute the average city population



Aggregation Pipeline example



```
db.orders.aggregate([ { $match: {status: "A"} }  
                      { $group: {_id:c_id, total: {$sum: $amount}} }  
                    ] )
```

Other Structure Issues

NoSQL

- a) Tables are unnatural
- b) “joins” are evil
- c) need to be able to “grep” my data

DB

- a) Tables are a natural/neutral structure
- b) data independence lets you precompute joins under the covers
- c) this is a price of all the DBMS goodness you get

This is an Old Debate – Object-oriented databases, XML DBs, Hierarchical, ...

Fault Tolerance

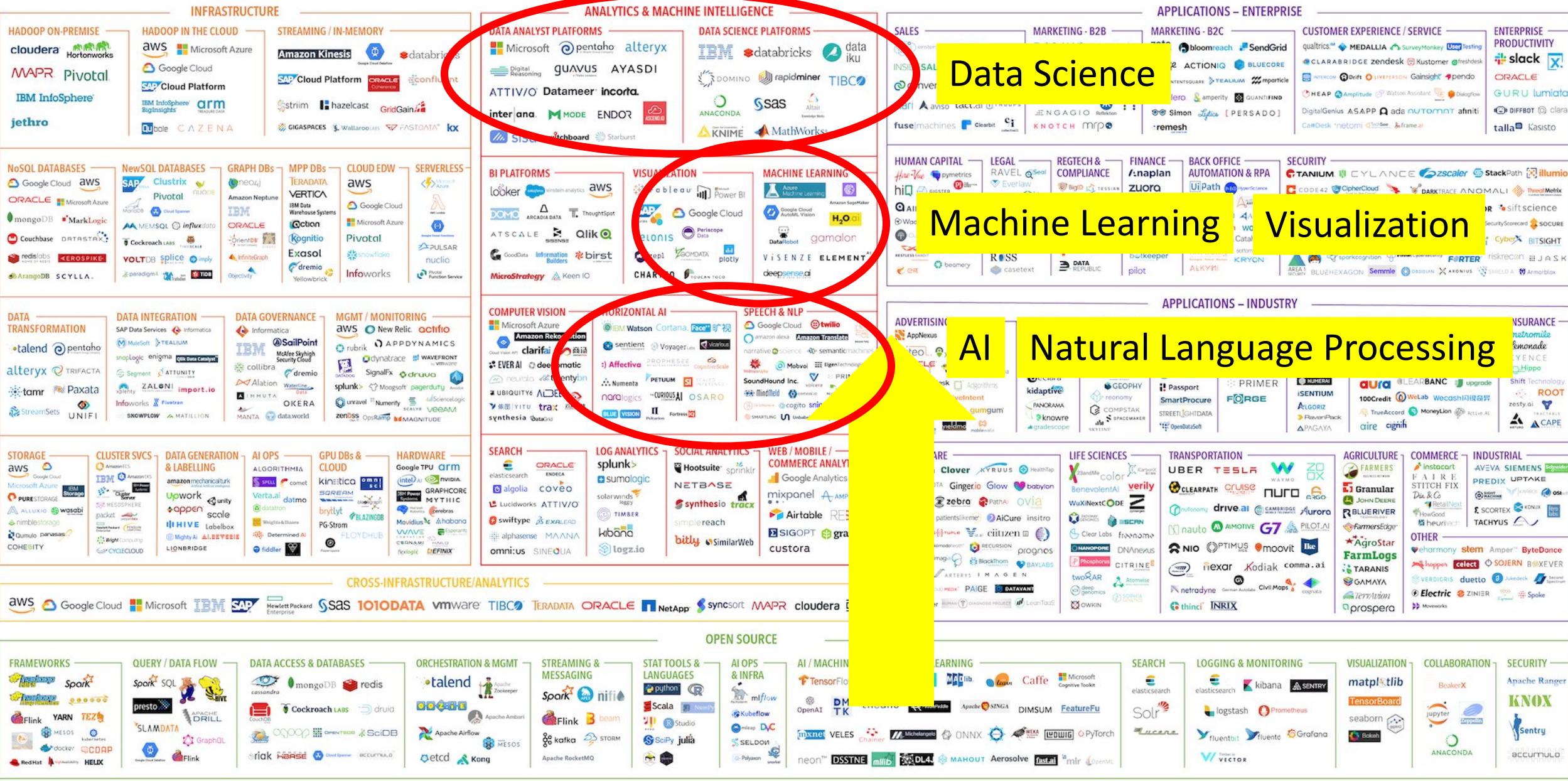
DBs: coarse-grained FT – if trouble, restart transaction

- Fewer, Better nodes, so failures are rare
- Transactions allow you to kill a job and easily restart it

NoSQL: Massive amounts of cheap HW, **failures are the norm** and massive data means **long running jobs**

- So must be able to do mini-recoveries
- This causes some overhead (file writes)

DATA & AI LANDSCAPE 2019



see full version at: http://mattturck.com/wp-content/uploads/2019/07/2019_Matt_Turck_Big_Data_Landscape_Final_Fullsize.png

A path in data science

(1) strong data systems skills

(i) coding skills

(ii) system architecture insights

performance tradeoffs

(2) application domain knowledge

(3) statistics, machine learning, math tools

Academic Research

Industry