

---

THE GEORGE WASHINGTON UNIVERSITY

---

WASHINGTON, DC

# 12. NoSQL

CSCI 2541 Database Systems & Team Projects

Wood & Chaufournier

# Logistics...

Last time: Benefits and Costs of RDBMS

This time: Not Just SQL

*NoSQL*

Wednesday:

- Data Mining / Analytics
- Project Status Checks

# Logistics...

Last time: Benefits and Costs of RDBMS

This time: Not Just SQL

Wednesday:

- Data Mining / Analytics
- Project Status Checks

If your team does not yet have a database schema deployed and accessible from a web interface, then

**you are in danger of failing the project.**

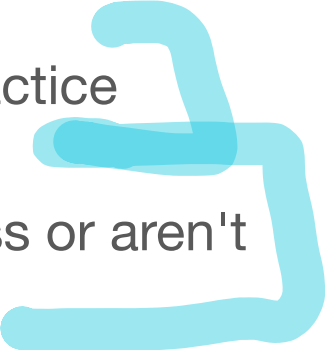
- Please come to office hours (especially mine!) if you are feeling behind and want advice on how to catch up

# Class support

We have ~12 hours of office hours every week

If you need help, come to us!

Find the balance:

- Problem solve, learn things on your own, practice debugging
  - but get help when you aren't making progress or aren't sure what to try!
- 

The TAs/UTAs/LAs are an amazing resource for you

- Maybe you should consider being one next year?!

# RDBMS Pros and Cons

## Strengths of Relational DBs?

- Well defined structure — integrity / type / constraints checking
- ACID
- Eliminate redundancy

## Weaknesses of Relational DBs?

- Fixed structure — waste space — hard to adapt

# RDBMS Pros and Cons

## Strengths

ACID properties  
(Atomic, Consistent,  
Isolated, Durable)

Widespread/  
standardized

## Weaknesses

Strong consistency  
properties are  
expensive to enforce

Strict structure is  
difficult to adapt

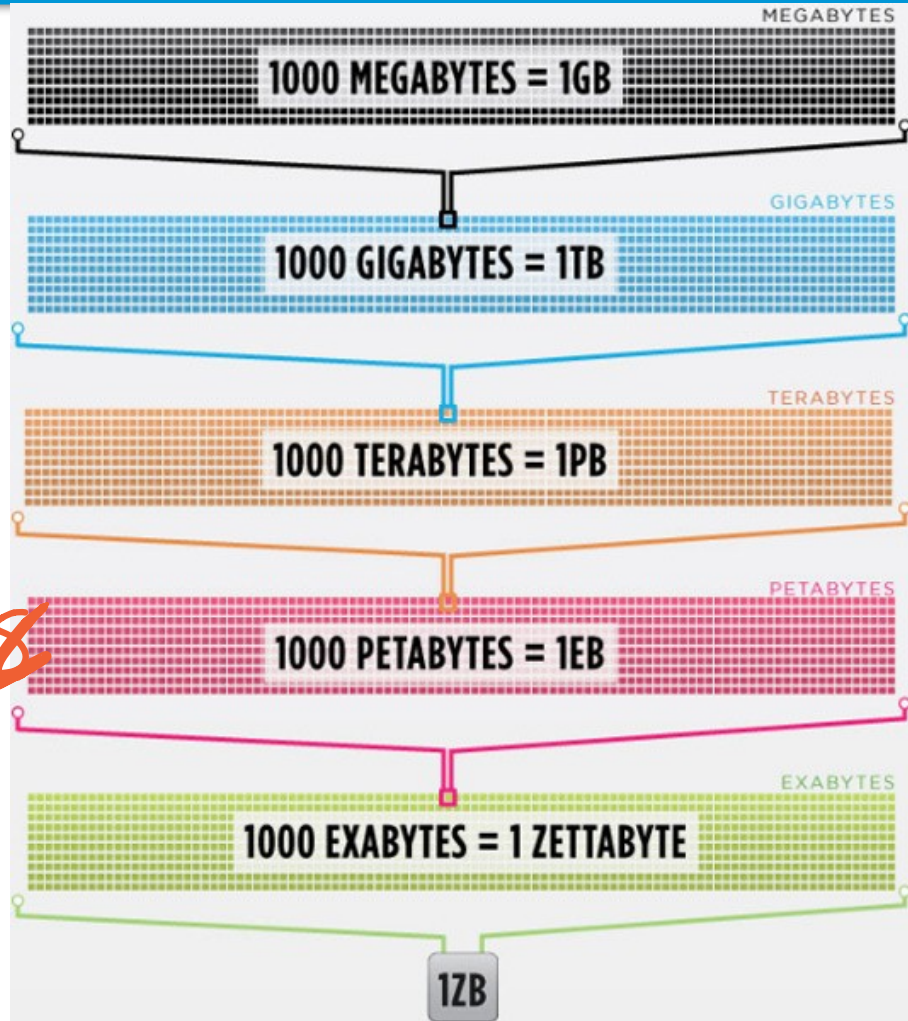
Some expensive  
features are not  
needed by some apps

# Trend 1

Data is getting bigger:

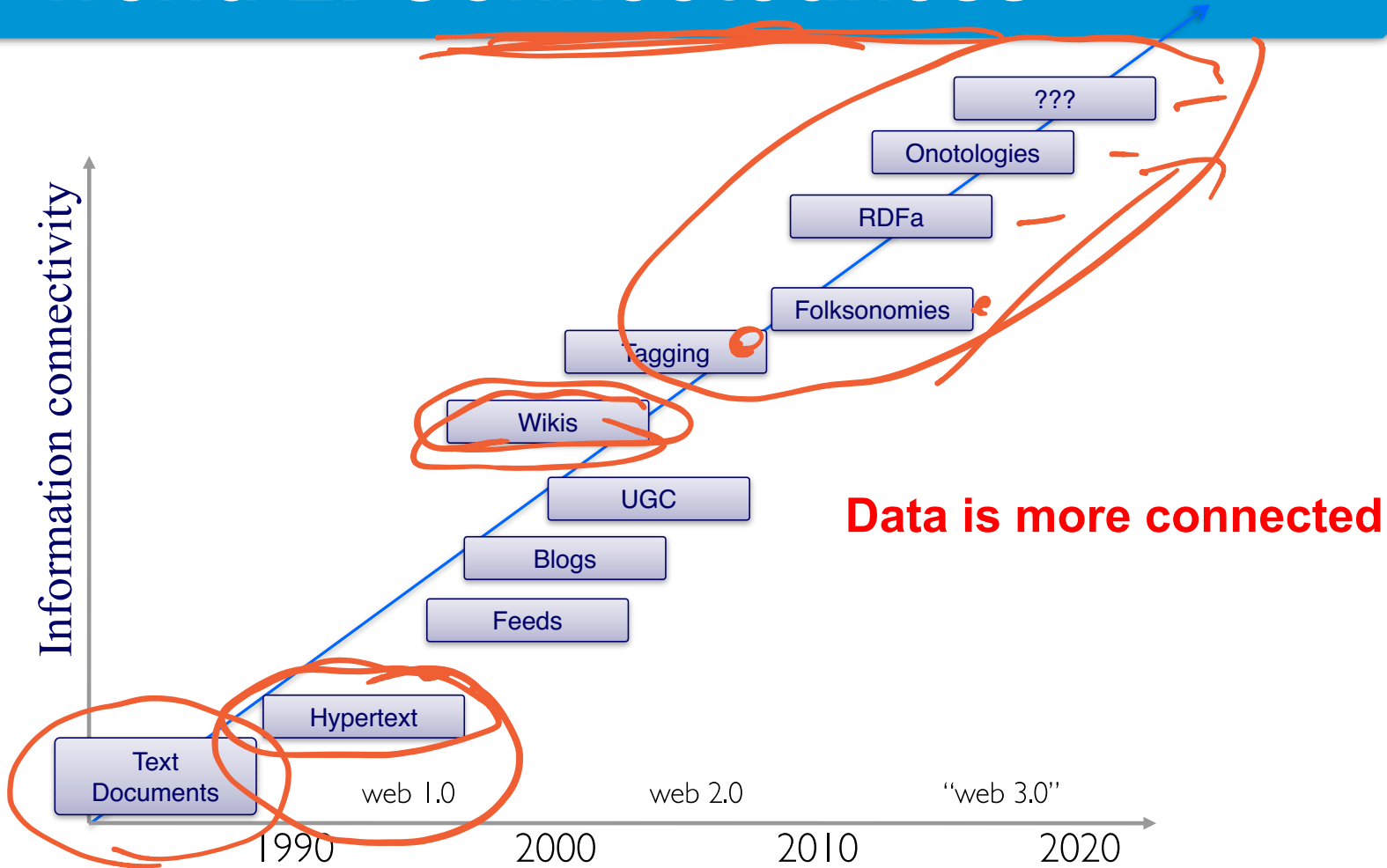
“Every 2 days we create as much information as we did up to 2003”  
– Eric Schmidt, Google in 2010

Facebook generates 4 Petabytes per day! (2020)



2018

# Trend 2: Connectedness





## Trend 3: Data is often Semi-Structured (or no structure)

If you tried to collect all the data of every movie ever made, how would you model it?

Actors, Characters, Locations, Dates, Costs, Ratings, Showings, Ticket Sales, etc.



# Relational Databases Challenges

Features of relational databases make them "challenging" for certain problems:

1. Fixed schemas – defined ahead of time, changes are difficult, and lots of real-world data is "messy". Relational design requires lots of Joins. **So get rid of schemas**
2. Complicated queries – SQL is declarative and powerful but may be overkill. **Instead, do the work in application code**
3. Transaction overhead – Not all data and query answers need to be perfect. **Close enough is sometimes good enough**
4. Scalability – Relational databases may not scale sufficiently to handle high data and query loads or this scalability comes with a very high cost. **Find new ways to scale**

# Database Scaling

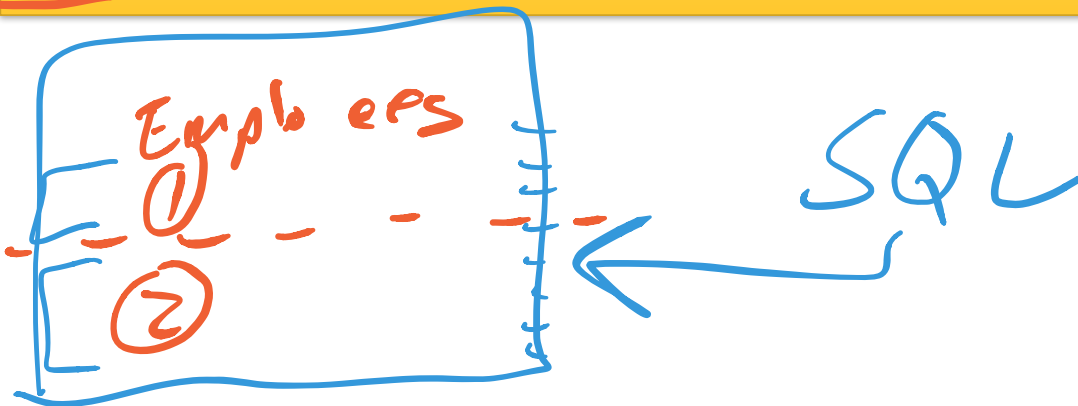
RDBMS are “scaled up” by adding hardware processing power *to a single computer*

*+ RAM  
+ CPU*

- Need more performance? upgrade your machine!

*Server*

Why is it difficult to **replicate** or **partition** an RDBMS to improve performance by using multiple computers?



# Let's consider the Python Dictionary

*DB*  
*man*  
myDict = { *get / set*

```
"name": "Maya",  
"address": "156 East 24th street",  
"city": "New York",  
"state": "New York",  
"cars": ["Ford", "Honda"]
```

*Keys* *Values*

Access any **Value** from the dictionary using its **Key**

- Dictionary = Key/Value Store = Hash Table / Map

Suppose we have to add lots and lots more fields...

How could we scale this "database"?

# Scaling a Dictionary (KV Store)

A Dictionary (or Key-Value store) can be:

**Scaled UP** by getting a more powerful server

- Just like RDBMS

**Scaled OUT** by adding another server and *partitioning* the data

- KV store doesn't need to support queries across objects!
- Consistency is not a problem, easy to exploit parallelism from many servers

# Dictionaries can be Nested

A "value" can be a complex data structure of its own!

Each Employee can have several fields within its own dictionary

We can partition the KV store so each server holds a set of Employees

```
employees = {}
employees["Brenda"] = {
    "name": "Brenda Kali",
    "address": "156 East 24th St",
    "city": "New York",
    "state": "New York",
    "cars": ["Ford", "Honda"]
}
employees["Jose"] = {
    "name": "Jose Constantino",
    "address": "231 West 181st St",
    "city": "New York",
    "state": "New York",
    "cars": ["Tesla"]
}
...
```

Be careful - key must be unique!

# Employee Database

Which is better?!

Two possible structures

*App*  
**RDBMS / SQL**

ID	name	address	...
Brenda	Brenda Kali	156 E. 24th St	...
Jose	Jose Constantino	231 W. 181st St	...
...	...	...	...

*Cars*

ID	car
Brenda	Ford
Brenda	Honda
Jose	Tesla

*App*  
**KV Store / Not SQL**

```
employees = {}
employees["Brenda"] = {
  "name": "Brenda Kali",
  "address": "156 East 24th St",
  "city": "New York",
  "state": "New York",
  "cars": ["Ford", "Honda"]
}
employees["Jose"] = {
  "name": "Jose Constantino",
  "address": "231 West 181st St",
  "city": "New York",
  "cars": ["Tesla"]
}
...
```

# It depends!

Do you need to filter employees by where they live?

- Use RDBMS! KV store just knows about the key!

What if each employee has **unique set of fields** that must be stored?

- Use KV store since internals of an employee are entirely customizable

What if scale of data is really really big?

- Use KV store **IF** you don't need to worry about cross-record consistency or queries



# Does this look familiar to anyone?

(Reformatted slightly)

JSON

```
{ 'Brenda': {  
  'name': 'Brenda Kali',  
  'address': '156 East 24th St',  
  'city': 'New York',  
  'state': 'New York',  
  'cars': ['Ford', 'Honda']},  
  'Jose': {  
    'name': 'Jose Constantino',  
    'address': '231 West 181st St',  
    'city': 'New York',  
    'state': 'New York',  
    'cars': ['Tesla']}  
}
```

# JSON, XML, etc

'Schema-less' data structure definitions

- Data format, not a full DBMS!

JavaScript Object Notation (**JSON**, pronounced "Jason")

- Serializes (saves) data objects into text form
- Human-readable
- Semi-structured
- Pervasively used in many languages (beyond JS)

Used to transmit most data to/between web services over AJAX/REST interfaces

- Client-side javascript makes a request to server, server responds with JSON data, client updates local browser view

# JSON Example

## JSON constructs:

- **Values:** number, strings (double quoted), true, false, null
- **Objects:** enclosed in { } and consist of set of key-value pairs (dictionary)
- **Arrays:** enclosed in [ ] and are lists of values
- Objects and arrays can be nested

## Example:

```
{
  "Employees": [
    {
      "eno": "E1",
      "ename": "J. Doe",
      "title": "EE",
      "salary": 30000,
      "WorksOn": ["P1"]
    },
    {
      "eno": "E2",
      "ename": "M. Smith",
      "title": "SA",
      "salary": 50000,
      "WorksOn": ["P1", "P2"]
    },
    {
      "eno": "E3",
      "ename": "A. Lee",
      "title": "ME",
      "salary": 40000,
      "WorksOn": ["P3"]
    }
  ],
  "Projects": [
    {
      "pno": "P1",
      "pname": "Instruments",
      "budget": 150000
    },
    {
      "pno": "P2",
      "pname": "DB Develop",
      "budget": 135000
    },
    {
      "pno": "P3",
      "pname": "Budget",
      "budget": 250000
    }
  ]
}
```

# JSON Parsers

JSON parser converts JSON file (or string) into program objects (checks syntax)

- In javascript, can call eval() method on variable containing a JSON string

Many languages have APIs to allow for creation and manipulation of JSON objects

Common use:

- JSON data provided from a server (NoSQL or relational) and sent to web client
- Web client uses javascript to convert JSON into objects and manipulate as required

Converters for csv to json

# What is NoSQL?

Stands for No-SQL or Not Only SQL??

What is definition....No definition!! But common some characteristics:

Class of non-relational data storage systems



Schema-less: usually do not require a fixed schema nor do they use the concept of joins

Cluster friendliness – ability to run on large number of servers (distributed system / cluster) Scalability

All NoSQL offerings relax one or more of the ACID properties

# NoSQL - advantages

NoSQL databases are useful for several problems not well-suited for relational databases:

- Variable data: semi-structured, evolving, or has no schema
- Massive data: terabytes or petabytes of data from new applications (web analysis, sensors, social graphs)
- Parallelism: large data requires architectures to handle massive parallelism, scalability, and reliability
- Simpler queries: may not need full SQL expressiveness
- Relaxed consistency: more tolerant of errors, delays, or inconsistent results ("eventual consistency")
- Easier/cheaper: less initial cost to get started

NoSQL is not really about SQL but instead developing data management systems that are not relational.

# CAP Theorem..getting around ACID

The CAP Theorem (proposed by Eric Brewer) states that there are three properties of a data system:

- Consistency
- Availability
- Partitions

but you can have at most two of the three properties at a time

- Since scaling out requires partitioning, many NoSQL systems sacrifice consistency for availability/partitioning.

Eventual Consistency - weaker than ACID

- Kind of what it sounds like
- Does not guarantee updates are immediately visible
- But eventually all nodes will agree on a final value

