# ENABLING EXTREME SCALABILITY WITH NOSQL

# An introduction to NoSQL databases

**Demand of your DB is changing**

**Presented By:**
**Ashwani Kumar**
Updated/expanded for CS430/630 by Betty O'Neil

# What is covered in this presentation?



- A brief history of databases
- NoSQL WHY, WHAT & WHEN?
- Characteristics of NoSQL databases
- Aggregate data models
- CAP theorem

# Introduction



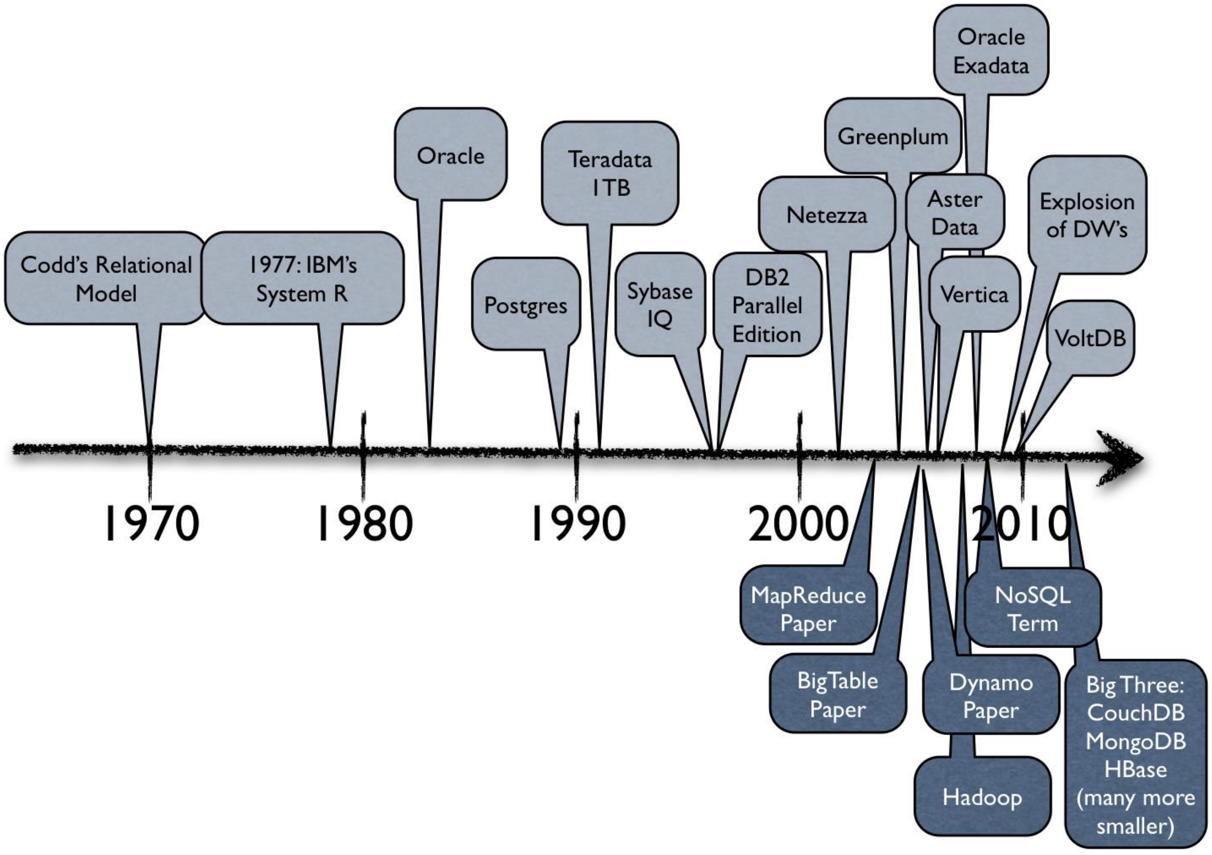
- Database - Organized collection of data

- DBMS - Database Management System: a software package with computer programs that controls the creation, maintenance and use of a database

- Databases are created to operate large quantities of information by inputting, storing, retrieving, and managing that information.

# A brief history





# Relational databases



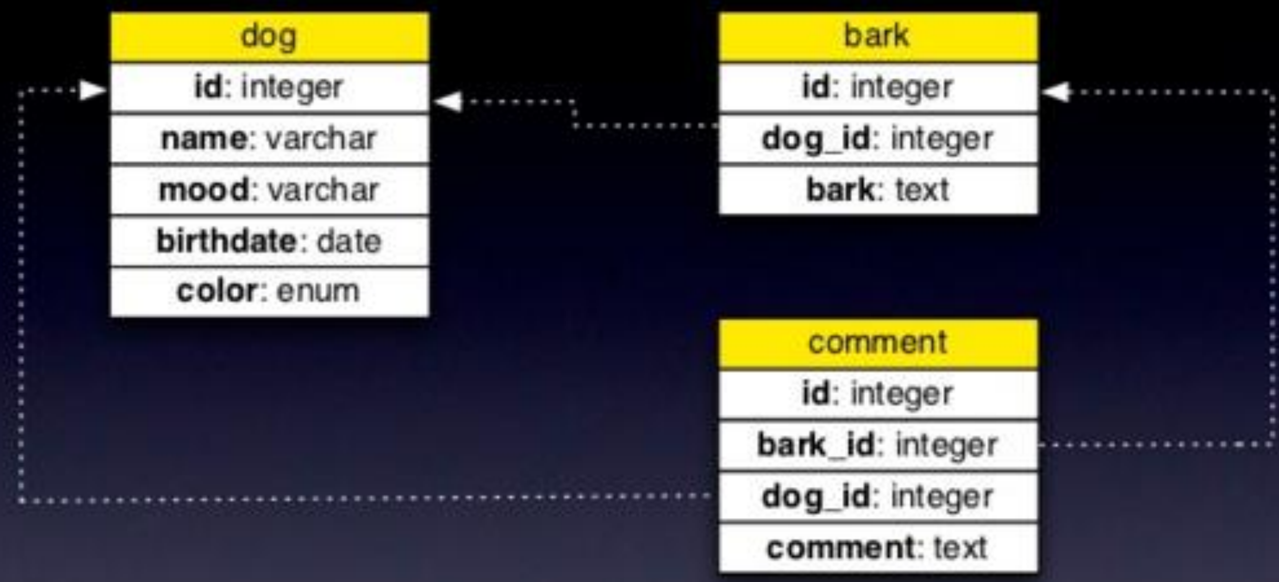
- <u>Benefits of Relational databases:</u>

- Designed for all purposes
- ACID
- Strong consistancy, concurrency, recovery
- Mathematical background (well-defined semantics)
- Standard Query language (SQL)
- Lots of tools to use with i.e: Reporting services, entity frameworks, ...

# SQL databases



# RDBMS




| id | name | mood | birth_date | color |
|---|---|---|---|---|
| 12 | Stella | Happy | 2007-04-01 | NULL |
| 13 | Wimma | Hungry | NULL | black |
| 9 | Ninja | NULL | NULL | NULL |

# NoSQL why, what and when?



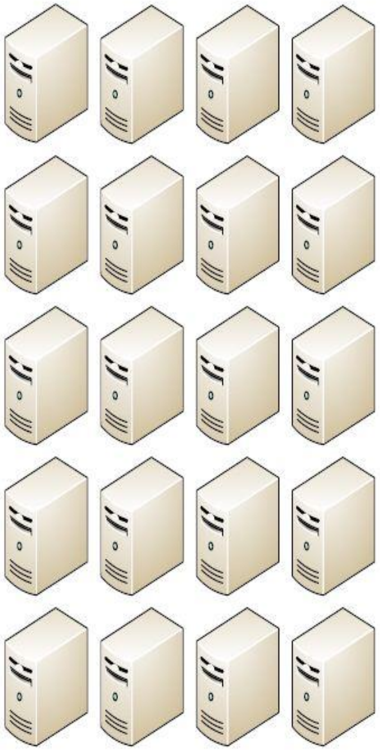
## But...

- Relational databases were not built for **distributed applications.**

## Because...

- Joins are expensive
- Hard to scale horizontally
- Impedance mismatch occurs
- Expensive (product cost, hardware, Maintenance)

## Era of Distributed Computing



# NoSQL why, what and when?

## But...

- ❑ Relational databases were not built for **distributed applications.**

Here "SQL" stands for relational DBs, not actually the query language

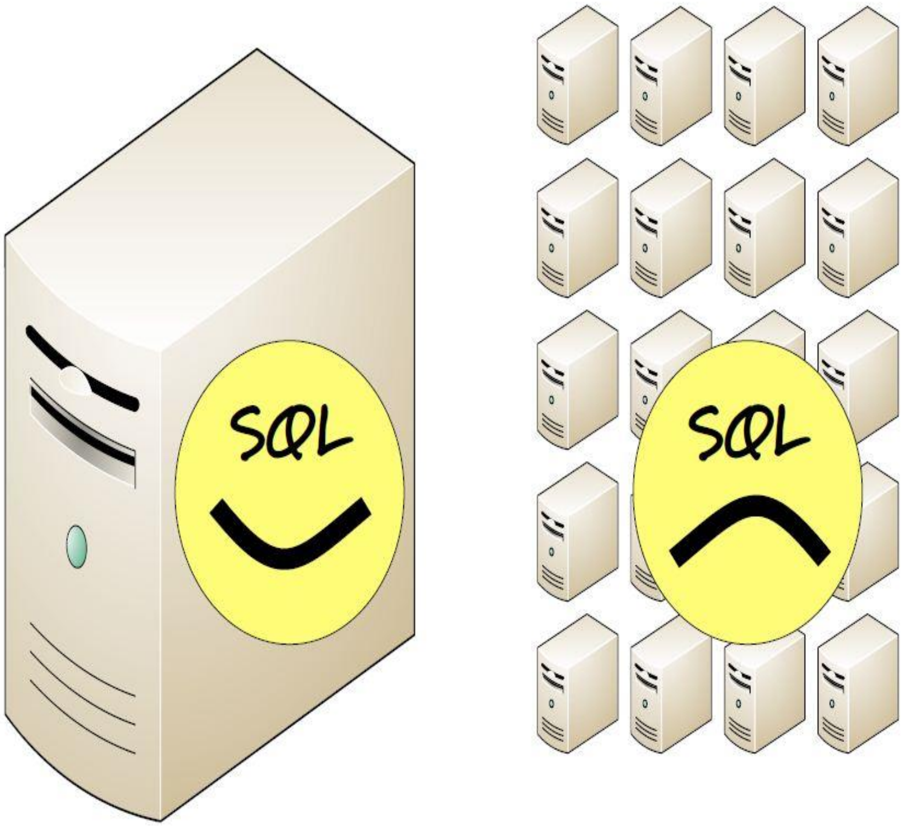
## Because...

- ❑ Joins are expensive
- ❑ Hard to scale horizontally
- ❑ Impedance mismatch occurs
- ❑ Expensive (product cost, hardware, Maintenance)

## And....

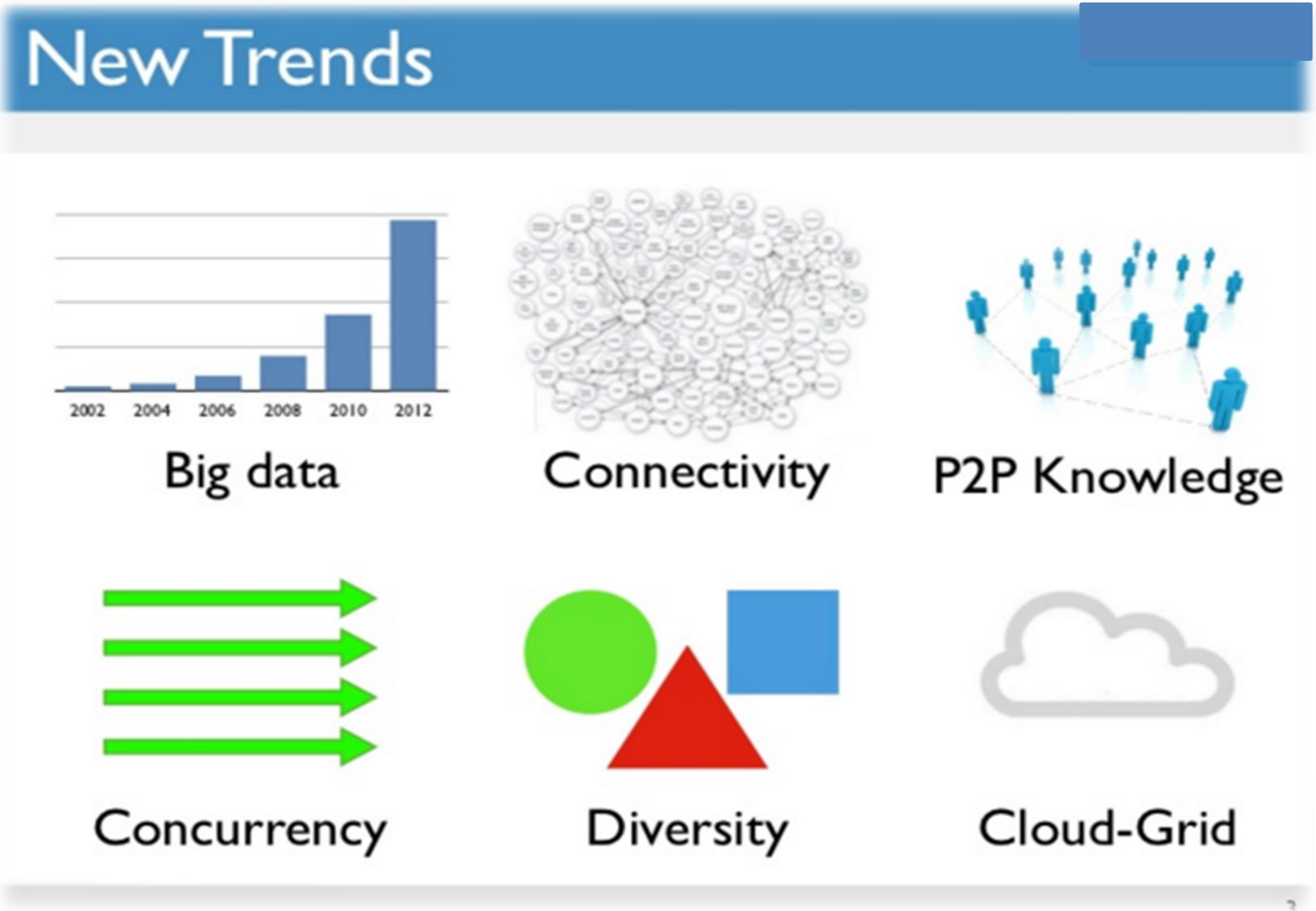
It's weak in:

- ❑ Speed (performance)
- ❑ High availability
- ❑ Partition tolerance



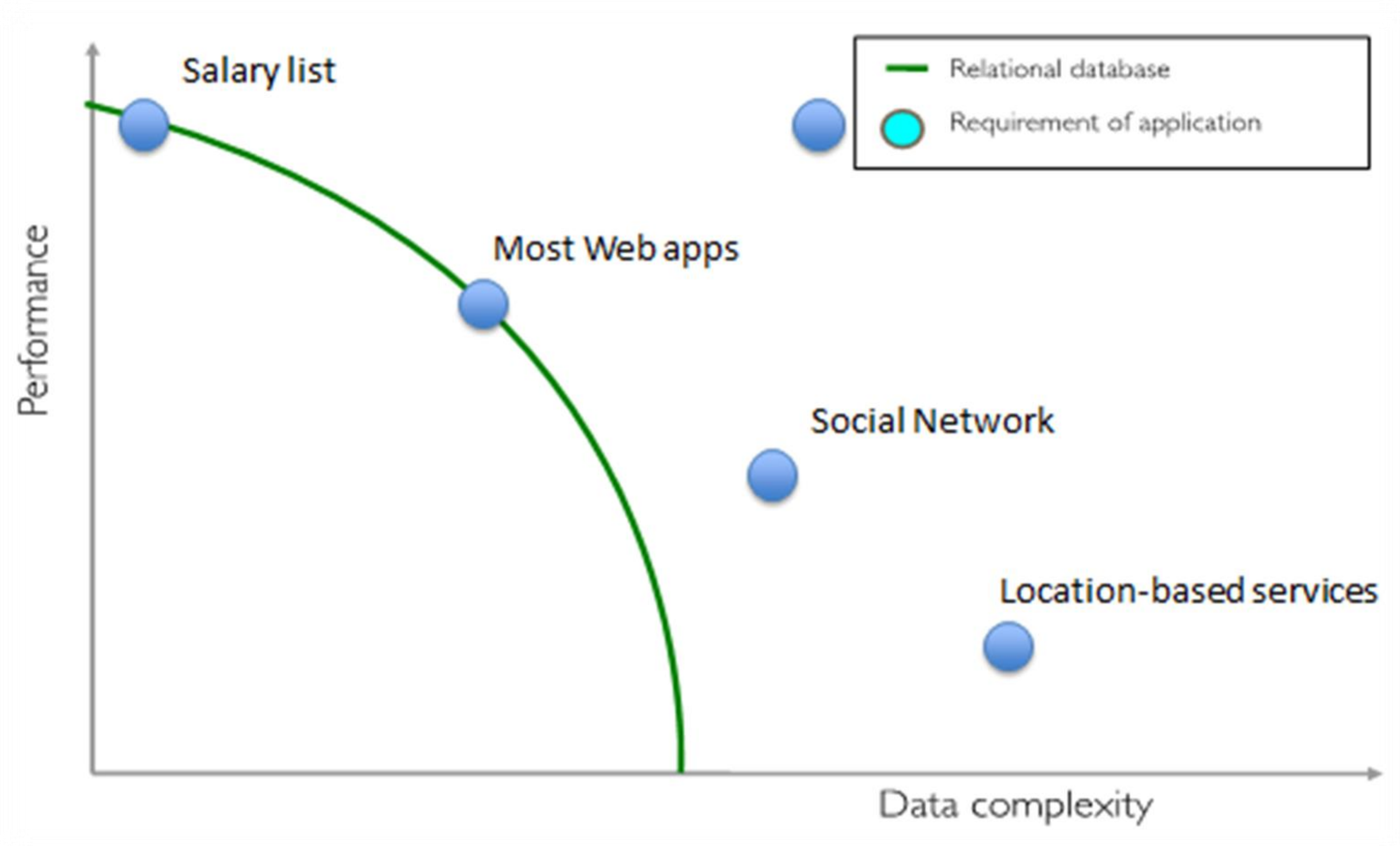
# Why NOSQL now?? Ans. Driving Trends





# Side note: RDBMS performance





# But.. What's NoSQL?


- A No SQL database provides a mechanism for storage and retrieval of data that employs less constrained consistency models than traditional relational database


- No SQL systems are also referred to as "NotonlySQL“ to emphasize that they do in fact allow SQL-like query languages to be used.

# Characteristics of NoSQL databases



NoSQL avoids:

- Overhead of ACID transactions
- Complexity of SQL query
- Burden of up-front schema design
- DBA presence
- Transactions (in many cases)

Provides:

- Easy and frequent changes to DB
- Fast development
- Large data volumes(eg.Google)
- Schema less

# NoSQL why, what and when?



## When and when not to use it?

**WHEN / WHY ?**

- When traditional RDBMS model is too restrictive (flexible schema)
- When ACID support is not "really" needed
- Object-to-Relational (O/R) impedance
- Because RDBMS is neither distributed nor scalable by nature
- Logging data from distributed sources
- Storing Events / temporal data
- Temporary Data (Shopping Carts / Wish lists / Session Data)
- Data which requires flexible schema
- **Polyglot Persistence** i.e. best data store depending on nature of data.

**WHEN NOT ?**

- Financial Data
- Data requiring strict ACID compliance
- Business Critical Data

# NoSQL is getting more & more popular



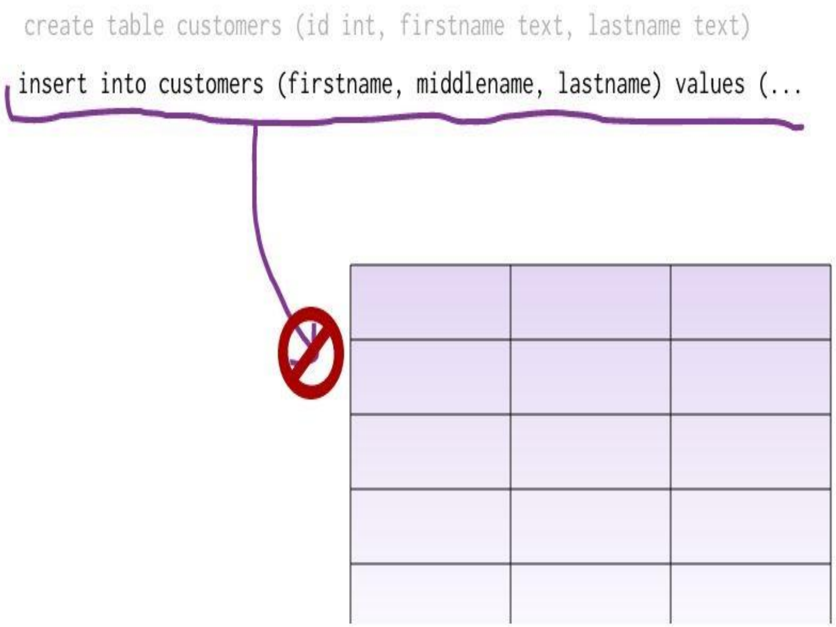
Actually Facebook uses mysql, but in a no-SQL way (key-value store)

# What is a schema-less datamodel?



In relational Databases:

- You can't add a record which does not fit the schema
- You need to add NULLs to unused items in a row
- We should consider the datatypes. i.e : you can't add a string to an integer field
- You can't add multiple items in a field (You should create another table: primary-key, foreign key, joins, normalization, ... !!!)



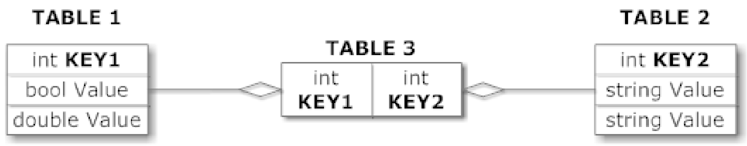
# What is a schema-less datamodel?



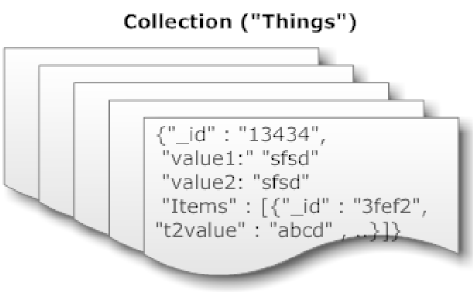
In NoSQL Databases, typically:

- There is no schema to consider
- There is no unused cell
- There is no datatype (implicit)
- We gather all items in an aggregate (docu

Relational Model




Document Model

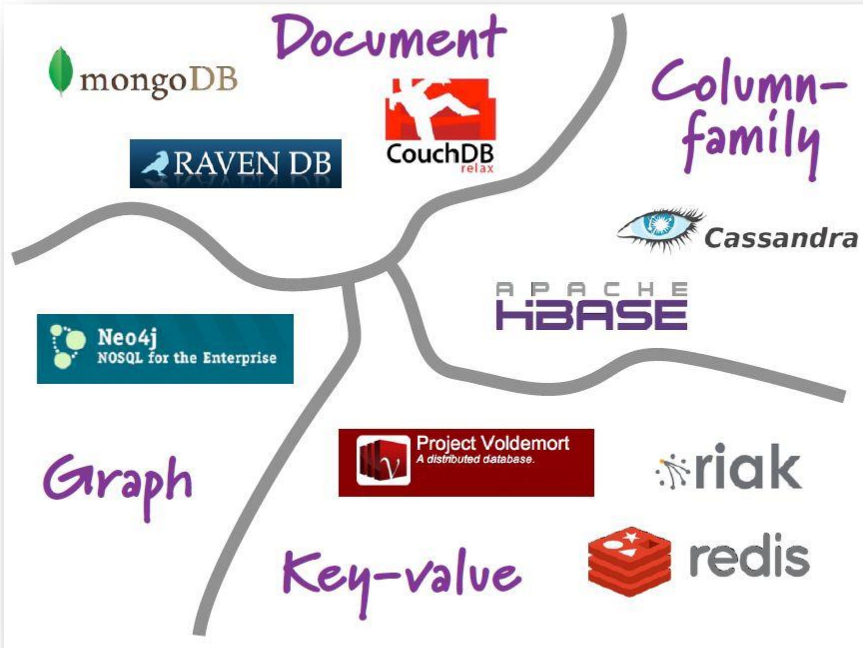


# Aggregate Data Models



NoSQL databases are classified in four major datamodels:

- Key-value
- Document
- Column family
- Graph

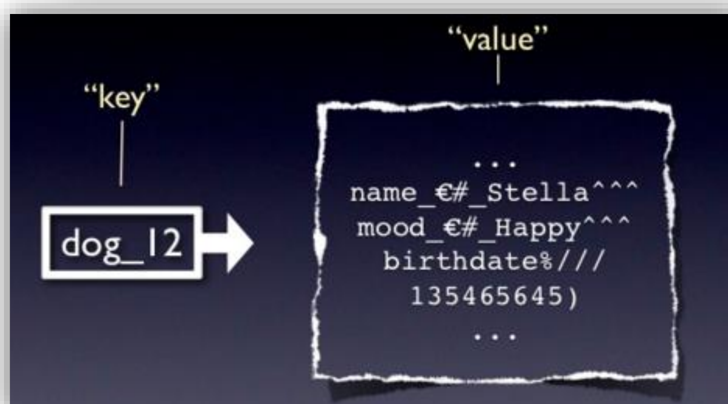



<u>Each DB has its own query language</u>

# Key-value data model

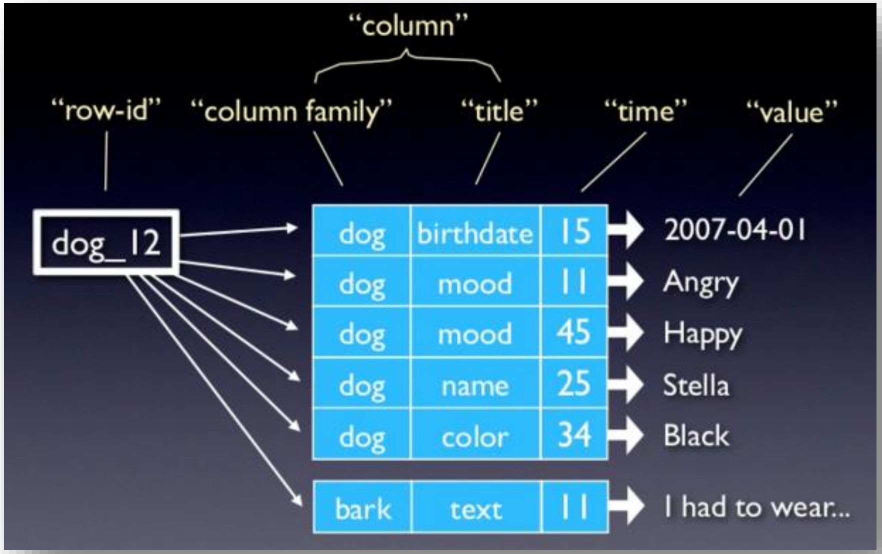
- Simplest NOSQL databases
- The main idea is the use of a hash table
- Access data (values) by strings called keys
- Data has no required format data may have any format
- Data model: (key, value) pairs
- Basic Operations:
  Insert(key,value),
  Fetch(key),
  Update(key),
  Delete(key)

| Car | |
|---|---|
| Key | Attributes |
| 1 | Make: Nissan<br>Model: Pathfinder<br>Color: Green<br>Year: 2003 |
| 2 | Make: Nissan<br>Model: Pathfinder<br>Color: Blue<br>Color: Green<br>Year:2005<br>Transmission: Auto |



# Column family data model

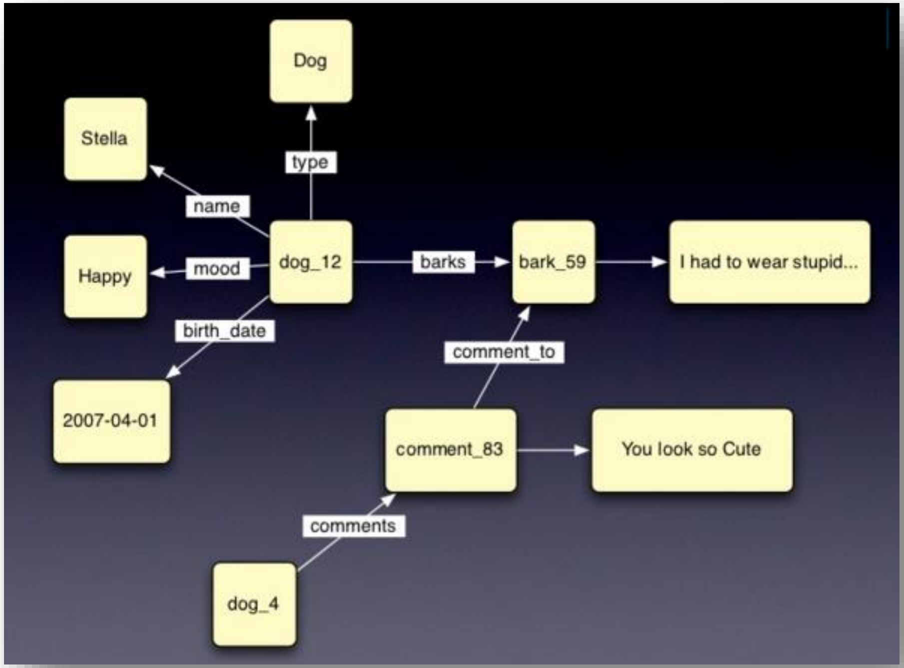
- The column is lowest/smallest instance of data.
- It is a tuple that contains a name, a value and a timestamp
- This is HBASE design
- We'll skip this case



# Graph data model



- Based on Graph Theory.
- Scale vertically, no clustering.
- You can use graph algorithms easily
- Transactions
- ACID



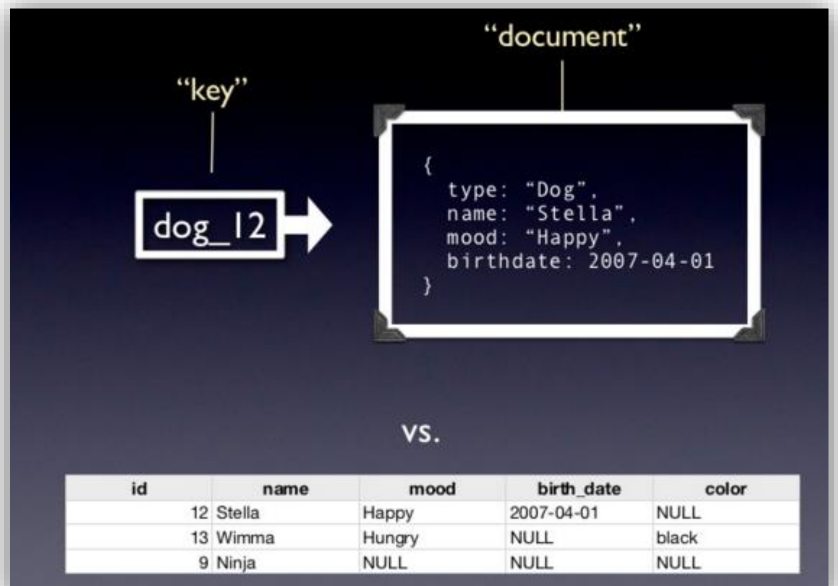

neo4j

# Document based data model

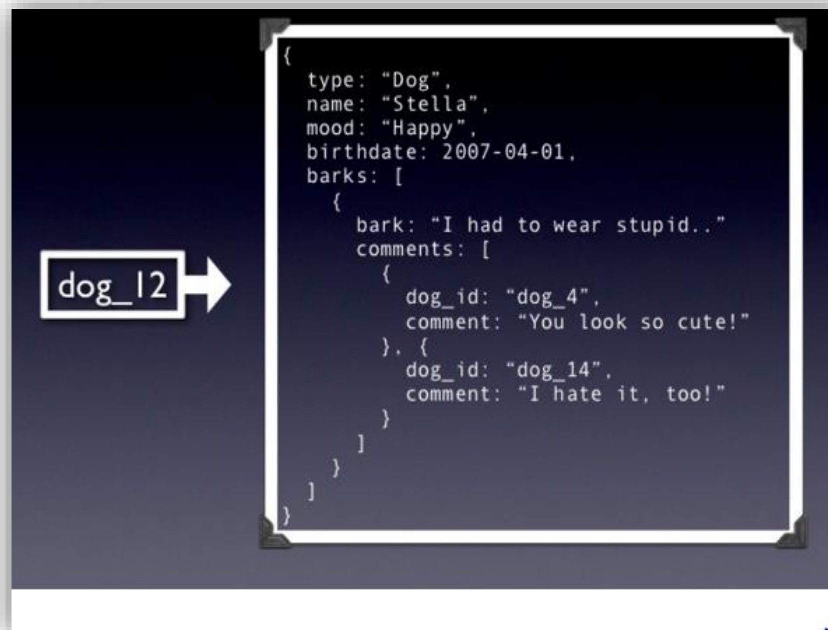


- Pair each key with complex data structure known as a document.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
- We'll look further into this type…

```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```



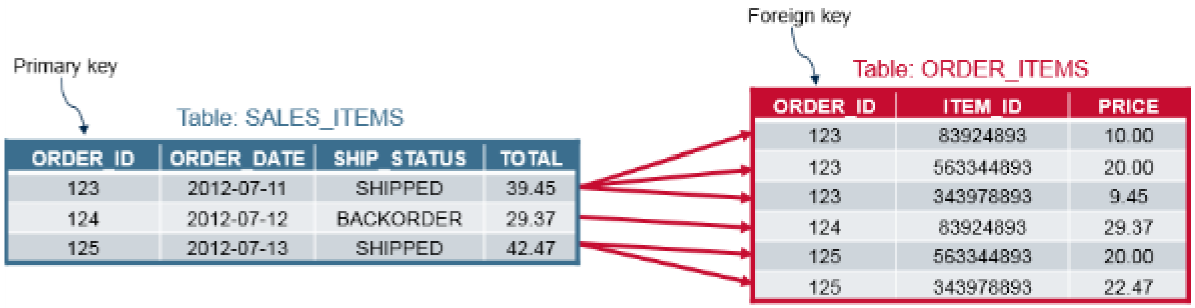

| id | name | mood | birth_date | color |
|---|---|---|---|---|
| 12 | Stella | Happy | 2007-04-01 | NULL |
| 13 | Wimma | Hungry | NULL | black |
| 9 | Ninja | NULL | NULL | NULL |



# Document-based data modeling

- No E-R, no normalization theory
- Goal: data that is accessed together is stored together
- Avoids joins, which are very expensive in a distributed system
- Query-centric: typically used in cases where data is read more often than updated
- Data duplication is tolerated
- Let’s look at examples…

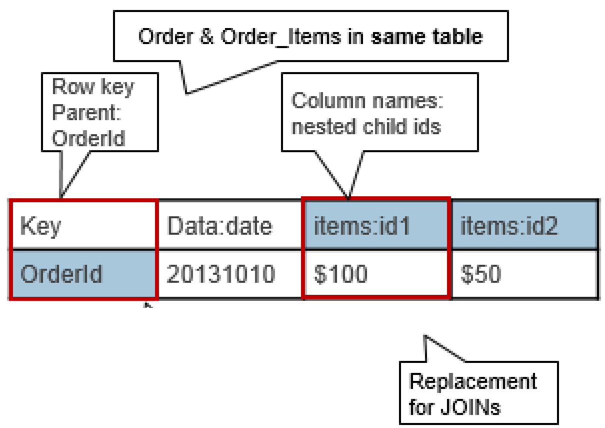
# Invoice-lineitem (one to many)




| ORDER_ID | ORDER_DATE | SHIP_STATUS | TOTAL |
|---|---|---|---|
| 123 | 2012-07-11 | SHIPPED | 39.45 |
| 124 | 2012-07-12 | BACKORDER | 29.37 |
| 125 | 2012-07-13 | SHIPPED | 42.47 |

| ORDER_ID | ITEM_ID | PRICE |
|---|---|---|
| 123 | 83924893 | 10.00 |
| 123 | 563344893 | 20.00 |
| 123 | 343978893 | 9.45 |
| 124 | 83924893 | 29.37 |
| 125 | 563344893 | 20.00 |
| 125 | 343978893 | 22.47 |

Relational Database: two tables, foreign key between them

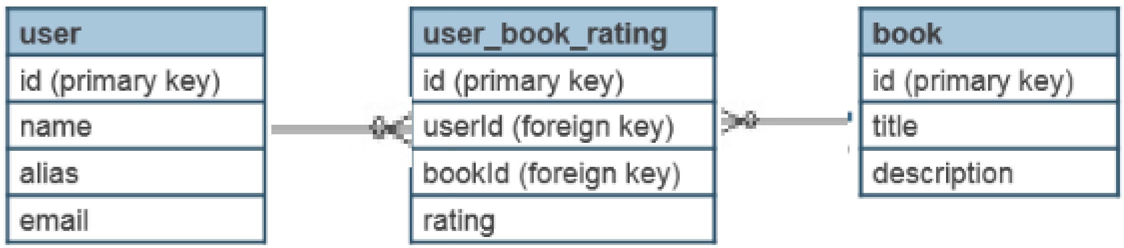



| Key | Data:date | items:id1 | items:id2 |
|---|---|---|---|
| OrderId | 20131010 | $100 | $50 |

No-Sql Database: one table, with lineitems for each order in its document

# Users-books: many-to-many

Online book store




Relational DB: with relationship table: if each user-book combo has a certain rating, the PK should be (userid, bookid)

User table    Column family for book ratings by userid for bookids

| Key | **data**:fname | ... | **rating:bookid1** | rating:bookid2 |
|---|---|---|---|---|
| **userid1** | | | 5 | 4 |

Book table    Column family for ratings for bookid by userid

| Key | **data:title** | ... | **rating:userid1** | rating:userid2 |
|---|---|---|---|---|
| **bookid1** | | | 5 | 4 |

NoSQL DB: one table/collection to look up ratings by userid, another to look up ratings by bookid (something like what we do in Java, etc.)
Note duplication of all rating data

- A document store, allowing embedded documents (unlike DynamoDB)
- Started in 2007
- Targeting semi-structured data in JSON
- Designed to be easy to "scale out" in traditional data centers or in the cloud
    - runs on Linux, OSX, Windows, Solaris
- Good support for indexing, partitioning, replication
- Nice integration in Web development stacks
- Not-so-great support for joins (or complex queries) or transactions

# a MongoDB database

- Database = a number of “collections”
- Collection = a list of “documents”
- Document = a JSON object (tree-like datastructure with arrays too)
  - Must have an _id attribute whose value can uniquely identify a document within the collection

☞In other words, a database has collections of similarly structured “documents”

# Querying MongoDB

- find() and sort() – Analogous to single-table selection/projection/sort
- “Aggregation” pipeline – With “stages” analogous to relational operators – Join, group-by, restructuring, etc.
- MapReduce: big data capabilities

# Find() examples

- All books
```
db.bib.find()
```
- Books with title "Foundations of Databases"
```
db.bib.find({ title: "Foundations of Databases" })
```
- Books whose title contains "Database" or "database" and whose price is lower than $50
```
db.bib.find({ title:/[dD]atabase/, price:{$lt:50} })
```
- Books with price between $70 and $100
```
db.bib.find({$and:[{price:{$gte:70}},
                   {price:{$lte:100}}]})
```

# MongoDB Document Example

```
{
first_name: “Paul”,
surname: “Miller”,
city: “London”,
location: [45.123,47.232],
cars: [
  { model: “Bentley”, year: 1973, value:100000},
  { model: “Rolls Royce”, year: 1965, value: 330000, },
  ]
}
```

This is representing a one-to-many relationship between persons and cars.

This is from the “RDBMS to MongoDB Migration Guide” available at mongodb.com.

# A more complex example

```
{ "_id": ObjectId("5ad88534e3632e1a35a58d00"),
 "name": { "first": "John", "last": "Doe" },
 "address": [
   { "location": "work",
     "address": { "street": "16 Hatfields",
           "city":  "London",
           "postal_code": "SE1 8DJ"},
     "geo": { "type": "Point",
              "coord": [ 51.5065752,-0.109081]}
   }
  ],
  "phone": [ { "location": "work", "number": "+44-1234567890"},…]
}
```

Here the relational design would have an streetaddress table, a geopoints table, location table, phoneno table, and person table.

# Yelp db for MongoDB

**Business:**

```
{"business_id": "tnhfDv5Il8EaGSXZGiuQGg",
"name": "Garaje", …
"categories": [ "Mexican", "Burgers", "Gastropubs" ], …}
```

**User:**

```
{"user_id": "Ha3iJu77CxlrFm-vQRs_8g", "name": "Sebastien",
"review_count": 56, "yelping_since": "2011-01-01", …}
```

**Review:**

```
{ "review_id": "zdSx_SD6obEhz9VrW9uAWA",
  "user_id": "Ha3iJu77CxlrFm-vQRs_8g", ←ref to user
  "business_id": "tnhfDv5Il8EaGSXZGiuQGg", ← ref to business
  "stars": 4, date": "2016-03-09", text": "Great place to
hang out after work … ", …}
```

**Also checkin, tip, photo. Many fewer tables.**

# Referencing in MongoDB

- Referencing enables data normalization, and can give more flexibility than embedding.
  - But the application will issue follow-up queries to resolve the reference, requiring additional round-trips to the server
  - or require a JOIN operation using the $lookup aggregation stage.
- References are usually implemented by saving the _id field1 of one document in the related document as a reference.
  - A second query is then executed by the application to return the referenced data
  - In yelp_db data, the user_id and business_id are used in the review to provide user and business details when needed

# Design Considerations on Refs

- MongoDB refs should be used where the object is referenced from many different sources, especially if those objects are changing over time.
  - In yelp_db, a business may have 30 reviews, so 30 reviews ref that business object, itself changeable.
  - A user may create 20 reviews, so then there are 20 reviews that ref that user object, itself changeable.
  - If these business and user objects are embedded in the review, it blows up the storage for this business by a factory of 30 and the storage for this user by factor of 20.
    - When a user object changes, it means 20 changes…
    - Also note MongoDB limits document size to 16MB
- Clearly this is a big design decision: more storage, harder updates, or more secondary access.

# AWS DynamoDB

- Only available on AWS (Amazon Web Services) cloud
- Similar DB on Google cloud: Cloud Datastore
- In between key-value store and MongoDB-style document store in data structures
- As cloud services, fully managed by cloud provider: just define it, start using it, scale it up, pay for faster access, … ("elastic")
- Replicated with automatic fail-over.
- Idea "cloud is the database", no traditional DBA needed (in theory, anyway)
- Great for huge jobs: supported Amazon prime days

# DynamoDB data

- Tables, items in tables, attributes in items, though attribute value could be arbitrary JSON as well as integer, string, etc.
- Attributes for a table are predefined, so not schema-free. One attribute is PK.
- The PK determines data location (the "partition")
- A secondary key can be used ("Sort key") to access data in a partition
  - Supports efficient access to one-to-many data such as invoice-lineitems
  - Called a "sort key" because the partition's data is effectively sorted by this key, allowing some tricks on access
- Can use refs efficiently with use of index
- No easy access to subdocuments as in MongoDB: here pull out whole JSON doc, take it apart.

# SQL vs NOSQL



## Differences

| | SQL Databases | No SQL Database |
|---|---|---|
| Example | Oracle , mysql | Mondo DB, CouchDB, Neo4J |
| Storage Model | Rows and tables | Key-value. Data stored as single document in JSON, XML |
| Schemas | Static | Dynamic |
| Scaling | Vertical & Horizontal | Horizontal |
| Transactions | Yes | Certain levels |
| Data Manipulation | Select, Insert , Update | Through Object Oriented API's |

# What we need ?



- We need a distributed database system having such features:
- **– Fault tolerance**
- **– High availability**
- **– Consistency**
- **– Scalability**

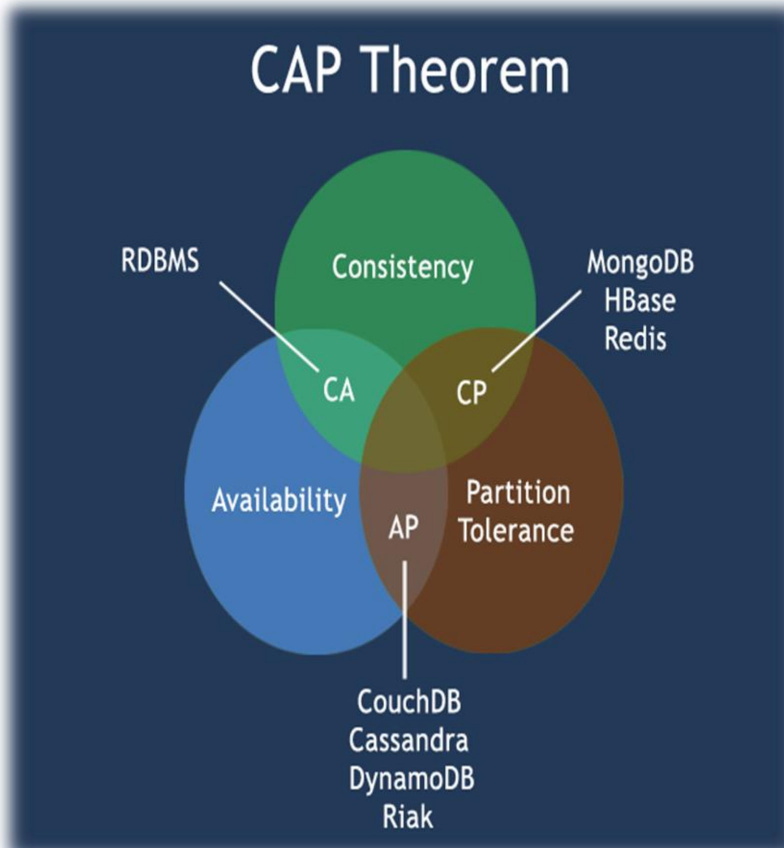
**Which is impossible!!!**

**According to CAP theorem**

# CAP theorem

## The CAP Theorem

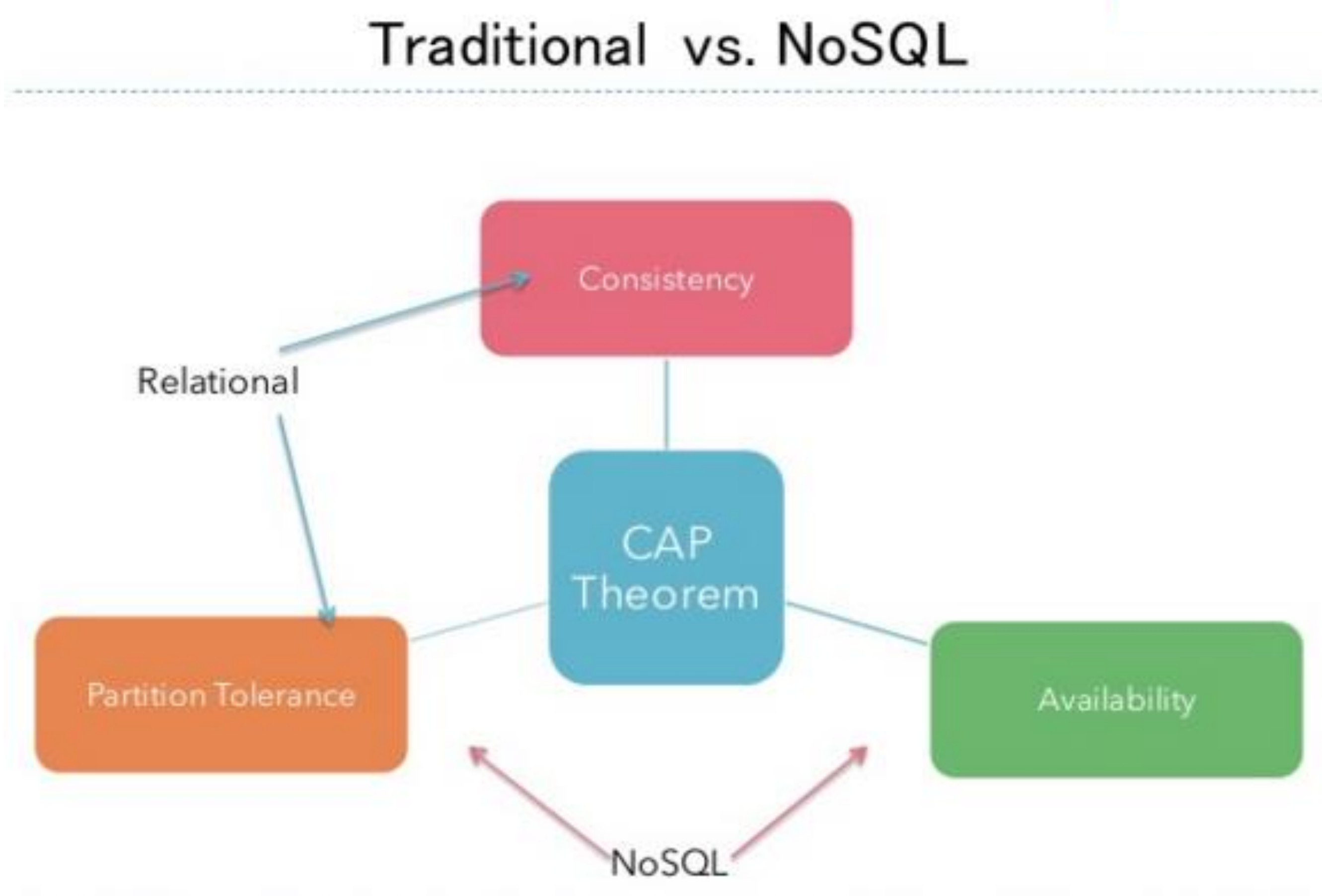
- Impossible for any shared data-system to guarantee simultaneously all of the following three properties:
  - Consistency – once data is written, all future read requests will contain that data
  - Availability – the database is always available and responsive
  - Partition Tolerance – if part of the database is unavailable, other parts are unaffected




We can not achieve all the three items
In distributed database systems (center)

# CAP theorem




Traditional vs. NoSQL
Consistency
Relational
CAP Theorem
Partition Tolerance
Availability
NoSQL

# NoSQL with consistency

- AWS DynamoDB, MongoDB, and Google Cloud Datastore offer strong consistency for certain operations (single key-value lookups, for example) vs. “eventual consistency” for others.
- In these strong-consistency cases, availability suffers, as the system returns errors related to lack of access to data, or takes a long time to respond.

# Conclusion....



## In Conclusion!

- RDBMS is a great tool for solving ACID problems
  - When data validity is super important
  - When you need to support dynamic queries
- NoSQL is a great tool for solving data availability problems
  - When it's more important to have fast data than right data
  - When you need to scale based on changing requirements
- Pick the right tool for the job

# References..




- **nosql**-database.org/
- https://www.mongodb.com/**nosql**-explained, also their RDBMS to MongoDB Migration Guide (available after registration of email)
- www.couchbase.com/**nosql**-resources/what-is-**no-sql**
- http://nosql-database.org/ "NoSQL DEFINITION: Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable“
- NoSQL distilled, Martin Fowler
- The basis of the intro part, and end parts of this presentation: https://www.slideshare.net/AshwaniKumar274/introduction-to-nosql-databases-57925674 and its author page: www.slideshare.net/AshwaniKumar274

# Thanks…

# Any Questions??