NoSQL Databases

Earlier…

- We have spent most of our time with the ***relational*** DB model so far.
- There are other models:
  - **Key-value**: a hash table
  - **Graph**: stores graph-like structures efficiently
  - **Object**: good for storing OO things
  - **Document**: stores an entire “document” at a time which is usually a text-based file with some internal structure (e.g., XML, JSON).

NoSQL

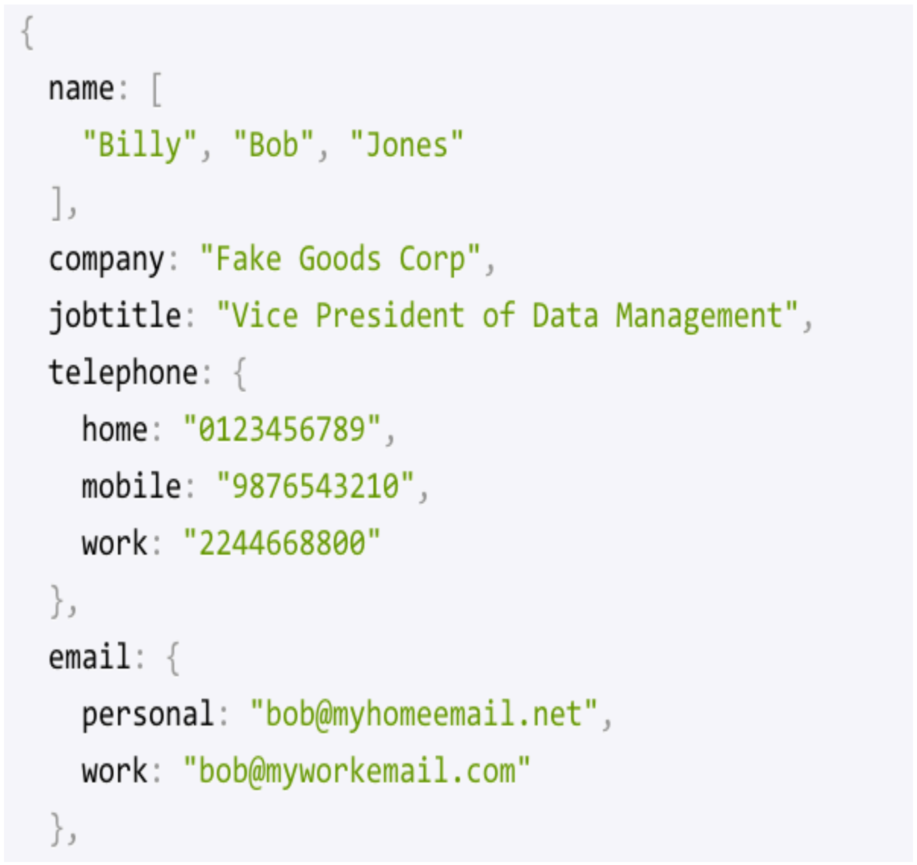
- NoSQL = “non-SQL” or “not only SQL” --- refers to anything other than the relational model.
- Around since the 60s, but the term was not popularized until these types of databases became extremely popular with companies like Facebook, Amazon, and Google.
- Increasingly used in big data and real-time web applications.
- Advantages: Simpler DB designs (no schemas), simpler scaling to clusters of machines, faster than relational in some cases.
- Disadvantages: Often no joins (low functionality), need multiple queries to answer some questions.

• **Relational Model**	• **Document Model**
• store related data in tables	• store related data in JSON-like, name-value documents
• require a schema which defines tables prior to use	• can store data without specifying a schema
• encourage normalization to reduce data redundancy	• must usually be denormalized so information about an item is contained in a single document
• support table JOINs to retrieve related data from multiple tables in a single command	• should not require JOINs (presuming denormalized documents are used)
• implement data integrity rules	• permit any data to be saved anywhere at anytime without verification
• provide transactions to guarantee two or more updates succeed or fail as an atomic unit	• guarantee updates to a single document — but not multiple documents
• can be scaled (with some effort)	• provide excellent performance and scalability
• use a powerful declarative language for querying	• use JSON data objects for querying
• offer plenty of support, expertise and tools.	• are a newer, exciting technology.

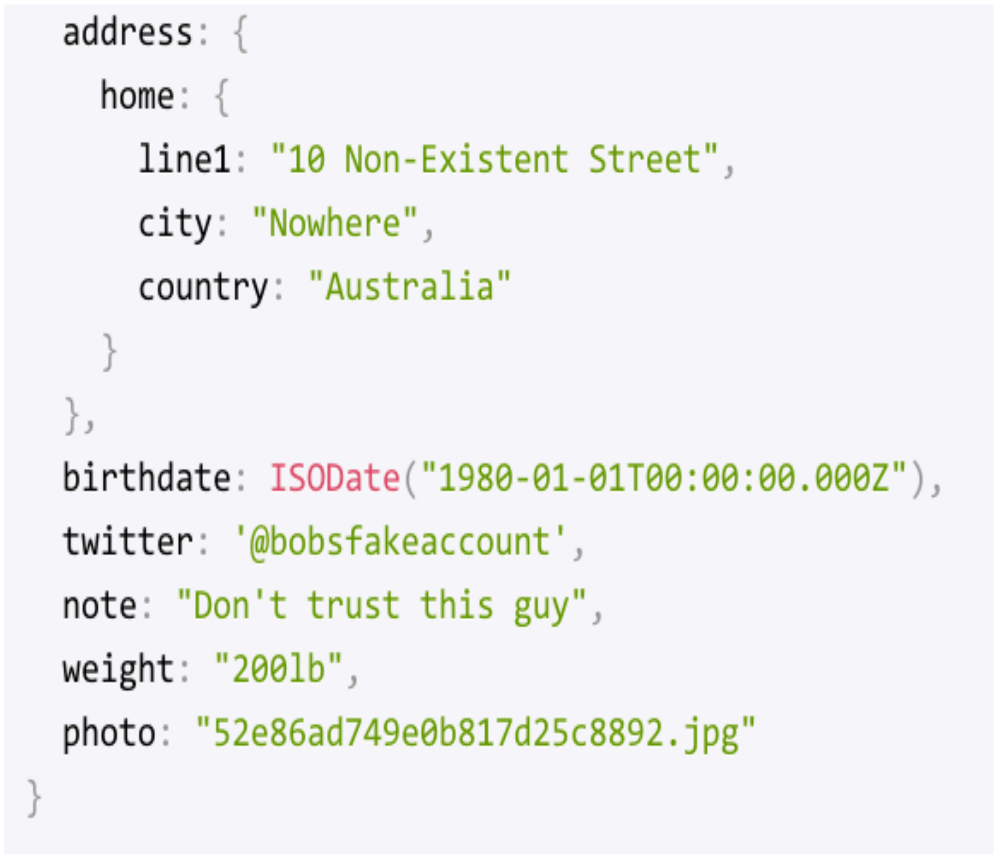
Scenario: Address book

- First attempt: id, title, firstname, lastname, telephone, email, address, city, state, zipcode.
- Problem – multiple telephone numbers, addresses, emails
  - Solution – Create separate tables for each of these.
- New relations: original relation is now just (id, title, first, last). Three new relations for Telephones, Addresses, Emails.
- Problems
  - Rigid schema – what if we want to add middle names, birthdays, company name, job title, anniversary, social media accounts?
  - Data is fragmented – split across multiple tables. Not easy to retrieve all of someone’s email addresses, telephone numbers, and postal addresses at once in an easy-to-read format.

NoSQL alternative



```
{
  name: [
    "Billy", "Bob", "Jones"
  ],
  company: "Fake Goods Corp",
  jobtitle: "Vice President of Data Management",
  telephone: {
    home: "0123456789",
    mobile: "9876543210",
    work: "2244668800"
  },
  email: {
    personal: "bob@myhomeemail.net",
    work: "bob@myworkemail.com"
  },
```



```
  address: {
    home: {
      line1: "10 Non-Existent Street",
      city: "Nowhere",
      country: "Australia"
    }
  },
  birthdate: ISODate("1980-01-01T00:00:00.000Z"),
  twitter: '@bobsfakeaccount',
  note: "Don't trust this guy",
  weight: "200lb",
  photo: "52e86ad749e0b817d25c8892.jpg"
}
```

Scenario: Twitter

- Suppose we want to implement a relational DB to store tweets.
- The overhead of a relational DB may be overkill here --- we will rarely need transactions, for instance.
- A failed update is unlikely to cause a global meltdown or financial loss. We can sacrifice a little ACID for some speed.

NoSQL Twitter

```
{
  user_id: ObjectID("65f82bda42e7b8c76f5c1969"),
  update: [
    {
      date: ISODate("2015-09-18T10:02:47.620Z"),
      text: "feeling more positive today"
    },
    {
      date: ISODate("2015-09-17T13:14:20.789Z"),
      text: "spending far too much time here"
    }
    {
      date: ISODate("2015-09-17T12:33:02.132Z"),
      text: "considering my life choices"
    }
  ]
}
```

MongoDB

- NoSQL document-oriented database.
- Every document is represented by JSON
  - Javascript Object Notation
- Free and open-source.
- According to their website, used by Expedia, Forbes, AstraZeneca, MetLife, Facebook, Urban Outfitters, and Comcast.

JSON

- Data interchange format, not a programming language.
- In other words, used to represent and store data, not give commands.
- Data types:
    - Number (integer or float), String (double quoted), Boolean (true/false)
    - Arrays: uses square bracket notation
    - Objects: Uses curly bracket notation
- Spacing doesn’t matter.

JSON Example (Object)

```
{
    “crn”: 12345
    “title”: “Databases”
    “department”: “Math and CS”
}
```

Name-value pairs. Name is quoted (must be a string). Value is quoted if it's a string, but it can be any data type.

JSON Example (Array)

```
[1, “hello world”, 2.76]
```

JSON Example (Array of Objects)

```
[
    {
        “crn”: 12345
        “title”: “Databases”
        “department”: “Math and CS”
    },
    {
        “crn”: 45897
        “title”: “Discrete Structures”
        “department”: “Math and CS”
    }
]
```

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [ { "type": "home", "number": "212 555-1234" },
                    { "type": "mobile", "number": "123 456-7890" }
  ],
  "children": [],
  "spouse": null
}
```

MongoDB concepts

- In a RDBMS, we often think of rows of a table as individual records.
- In MongoDB (and other document-oriented DBs), records are (JSON) ***documents***.
- A group of documents (with presumably similar structures) is called a ***collection*** in MongoDB.

- Table <-> Collection
- Row <-> Document
- Column <-> Field