NO SQL

what is no sql, and how to use them

Soheil Salimi - Professor Zojaji

What is SQL?

Structured Query Language is a domain-specific language used in programming and designed for managing data held in a relational database management system, or for stream processing in a relational data stream management system.

Why do we even need No SQL

- **simplicity** of design
- **simpler "horizontal" scaling** to clusters of machines (which is a problem for relational databases)
- **finer control over availability**, and limiting the object-relational impedance mismatch.
- The **data structures used by NoSQL databases** (e.g. key–value pair, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL.
- The particular suitability of a given **NoSQL database depends on the problem it must solve**. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.

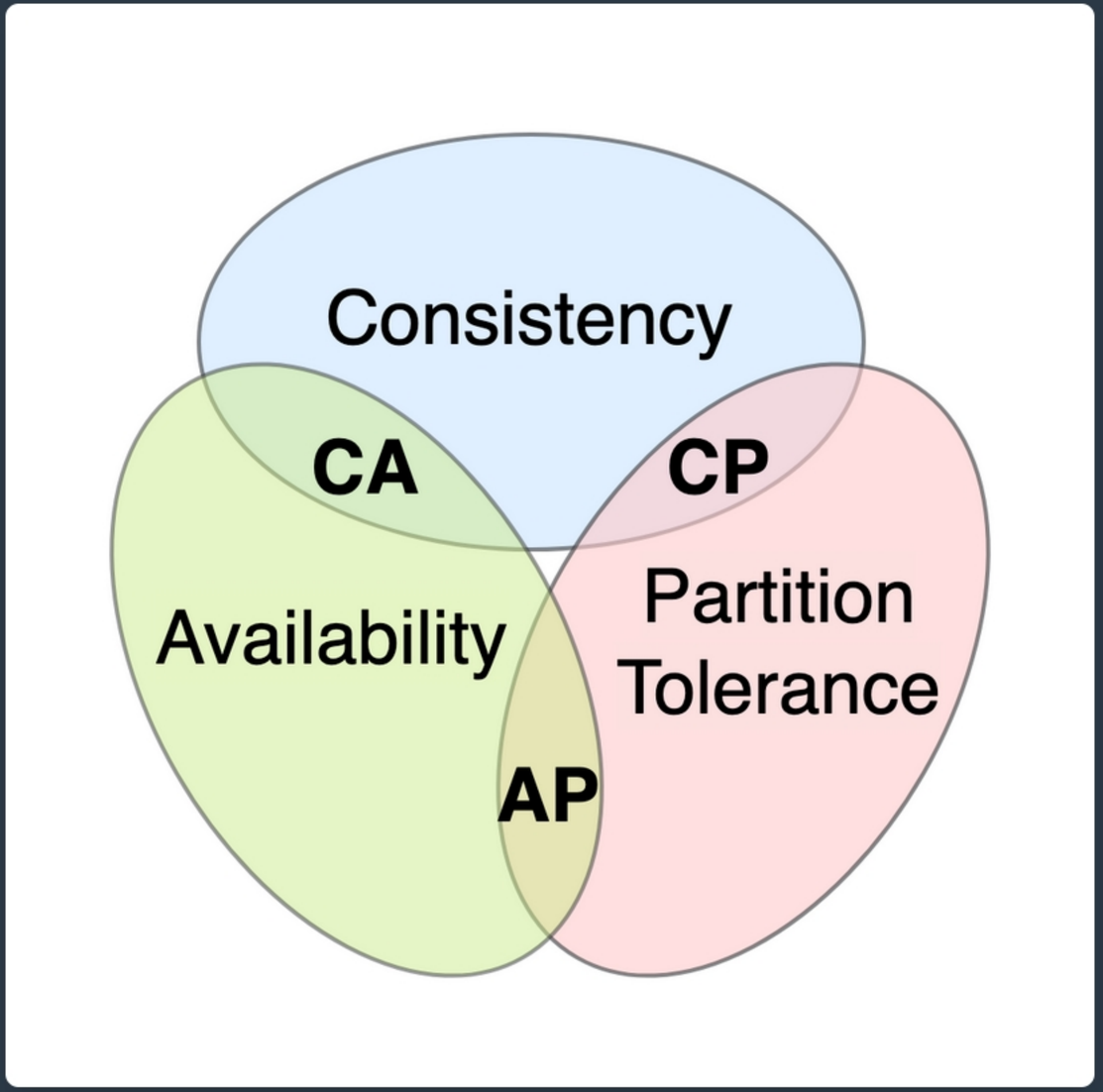
CAP theorem

the CAP theorem, states that any distributed data store can provide only two of the following three guarantees

Consistency Every read receives the most recent write or an error.

Availability Every request receives a (non-error) response, without the guarantee that it contains the most recent write.

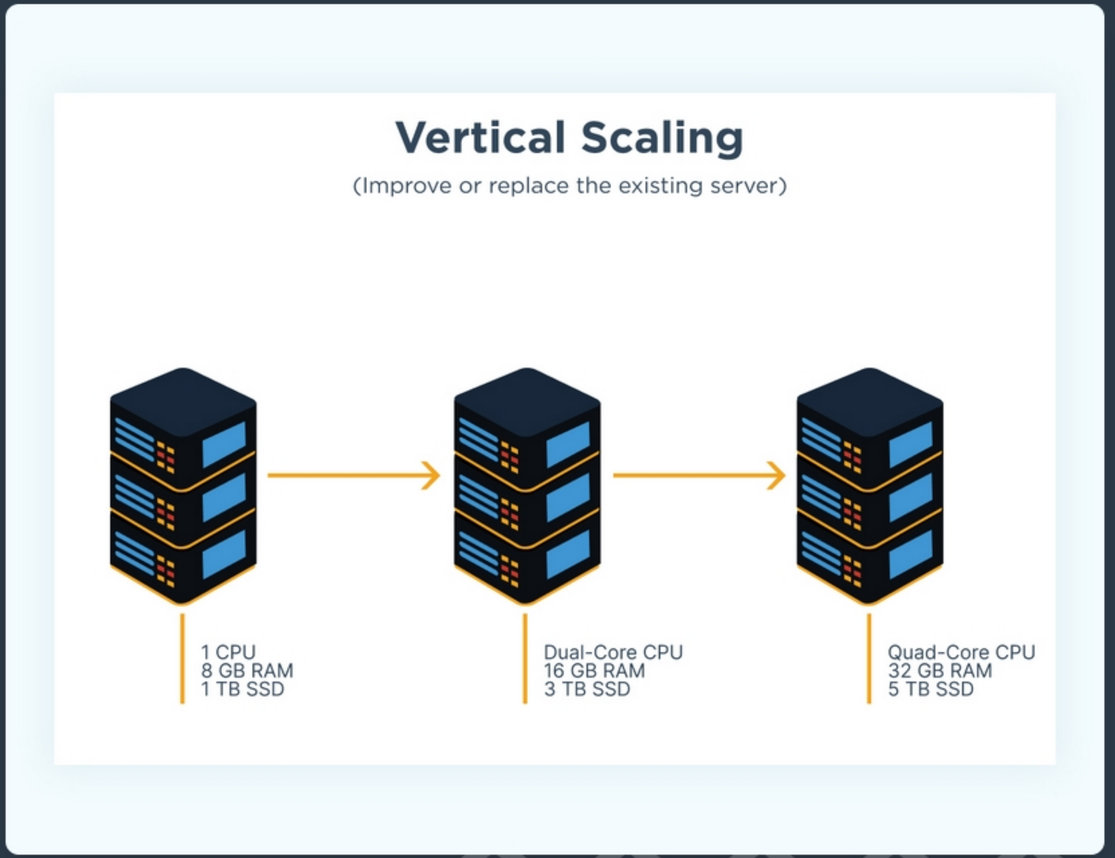
Partition tolerance The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.



What Is Vertical Scaling ?

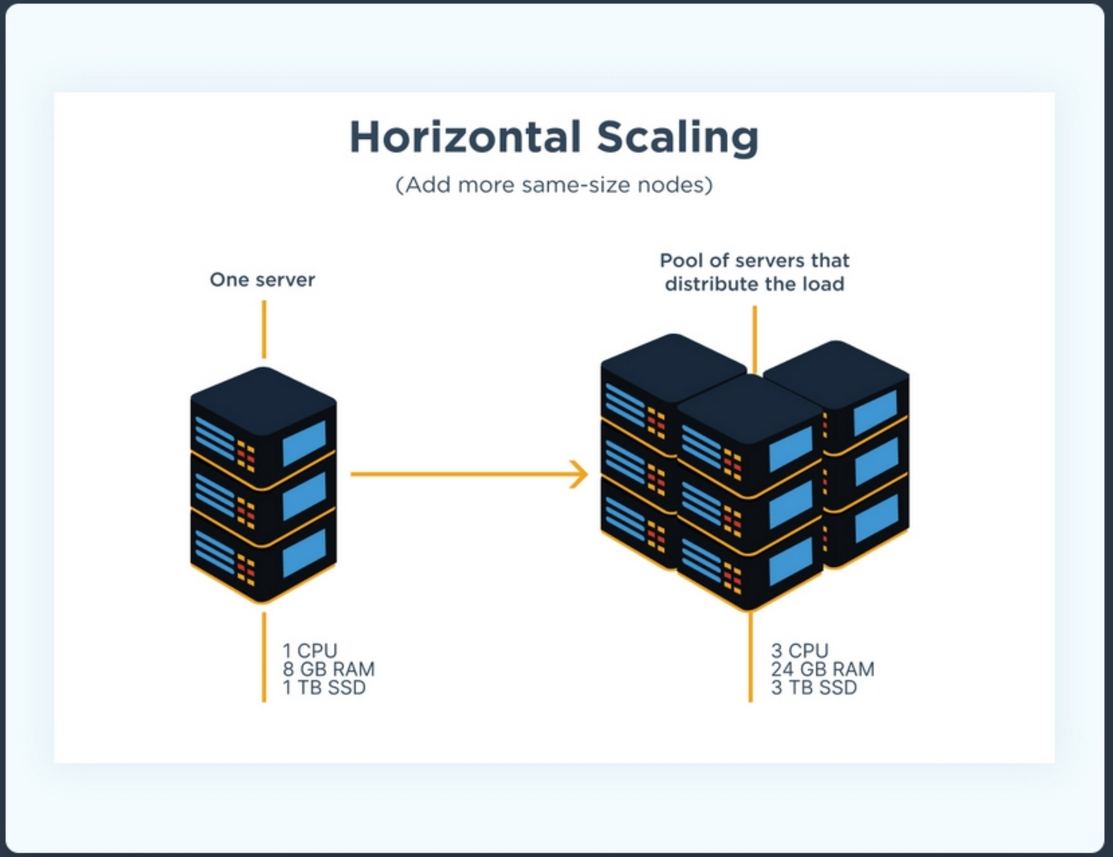
Vertical scaling (aka scaling up) describes adding additional resources to a system so that it meets demand.

While horizontal scaling refers to adding additional nodes, vertical scaling describes adding more power to your current machines. For instance, if your server requires more processing power, vertical scaling would mean upgrading the CPUs. You can also vertically scale the memory, storage, or network speed.



What Is Horizontal Scaling ?

Horizontal scaling (aka scaling out) refers to adding additional nodes or machines to your infrastructure to cope with new demands. If you are hosting an application on a server and find that it no longer has the capacity or capabilities to handle traffic, adding a server may be your solution.



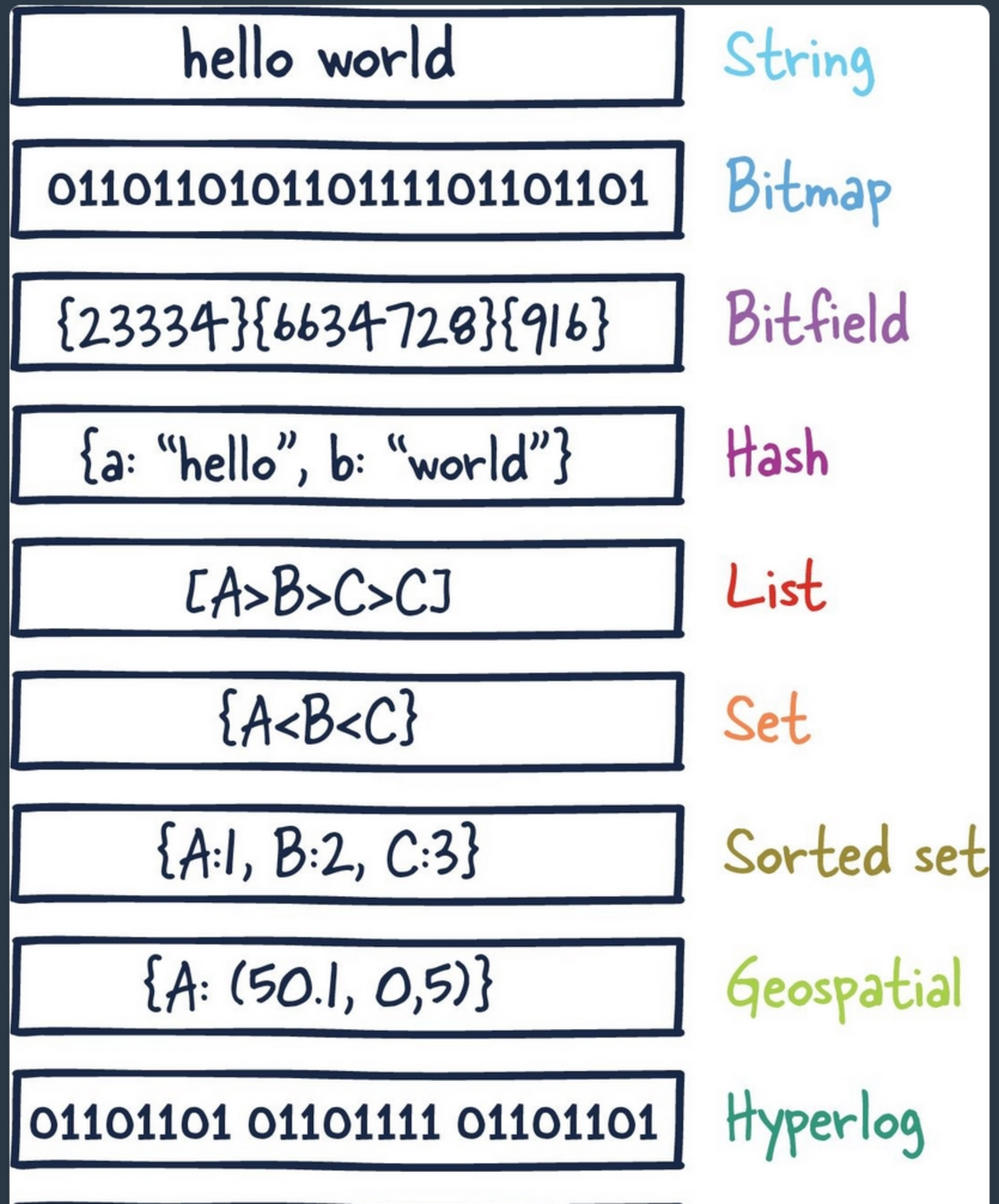
There is multiple kind of No SQL

Type	Notable examples of this type
Key–value	Redis
Tuple store	Apache River, GigaSpaces, Tarantool, TIBCO ActiveSpaces, OpenLink Virtuoso
Object database	Objectivity/DB, Perst, ZODB
Document store	Azure Cosmos DB, CouchDB, MongoDB
Wide-column store	Azure Cosmos DB, Cassandra, Google Cloud Datastore
Graph database	Azure Cosmos DB, Neo4J

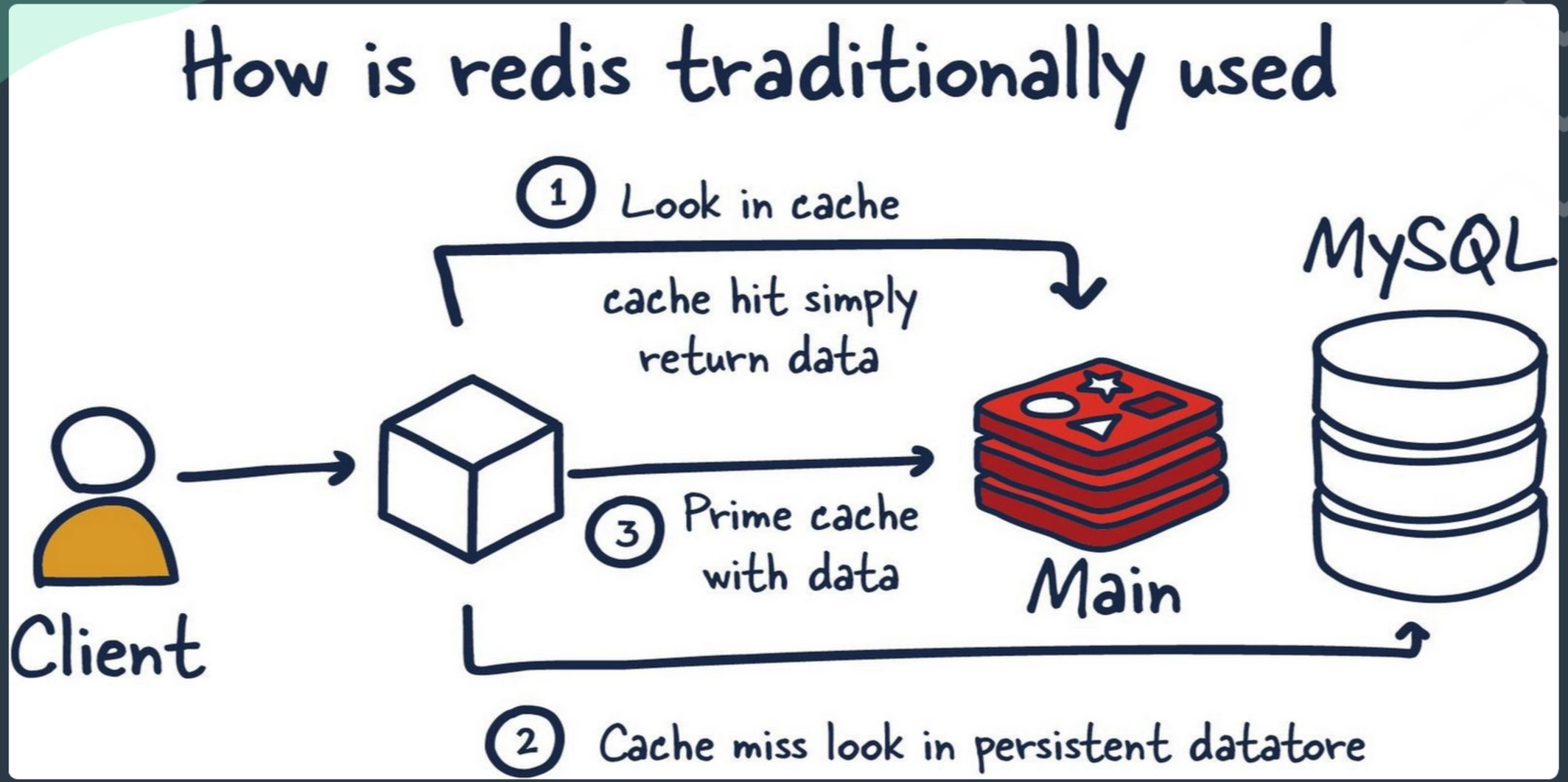
Key–value

Key–value (KV) stores use the associative array (also called a map or dictionary) as their fundamental data model. In this model, data is represented as a collection of key–value pairs, such that each possible key appears at most once in the collection.

Mostly used for caching since most of theme are in-memory databases



caches with redis



Document store

The central concept of a document store is that of a "document". While the details of this definition differ among document-oriented databases, they all assume that documents encapsulate and encode data (or information) in some standard formats or encodings. Encodings in use include XML, YAML, and JSON and binary forms like BSON. Documents are addressed in the database via a unique key that represents that document. Another defining characteristic of a document-oriented database is an API or query language to retrieve documents based on their contents.

```
{
    "_id": 1,
    "first_name": "Tom",
    "email": "tom@example.com",
    "cell": "765-555-5555",
    "likes": [
       "fashion",
       "spas",
       "shopping"
    ],
    "businesses": [
       {
          "name": "Entertainment 1080",
          "partner": "Jean",
          "status": "Bankrupt",
          "date_founded": {
             "$date": "2012-05-19T04:00:00Z"
          }
       },
       {
          "name": "Swag for Tweens",
          "date_founded": {
             "$date": "2012-11-01T04:00:00Z"
          }
```

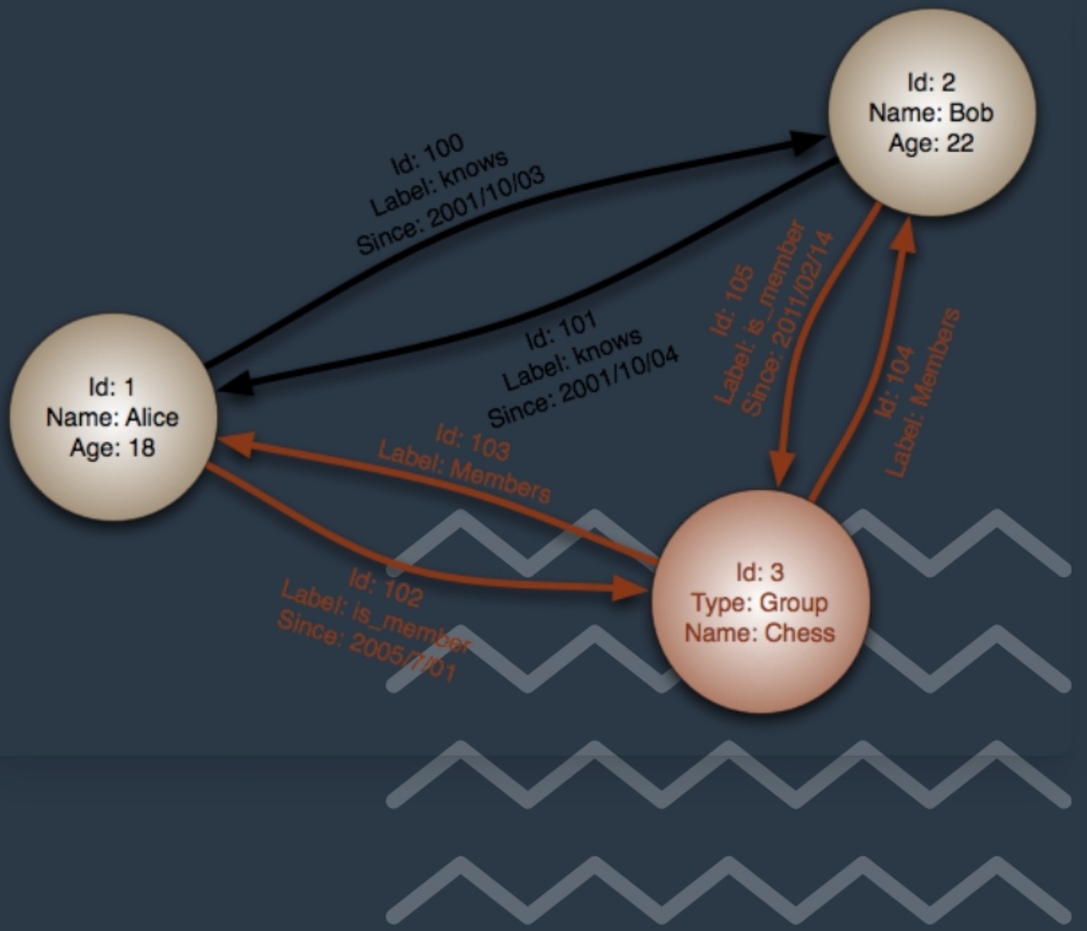
Graph

Graph databases are designed for data whose relations are well represented as a graph consisting of elements connected by a finite number of relations. Examples of data include social relations, public transport links, road maps, network topologies, etc.

Nodes represent entities or instances such as people, businesses. They are roughly the equivalent of a record, or a document in a document-store database.

Edges, also termed graphs or relationships, are the lines that connect nodes to other nodes

Properties are information associated to nodes.



When is a NoSQL database the best option?

When deciding which database to use, decision-makers typically find one or more of the following factors lead them to selecting a NoSQL database:

- Fast-paced Agile development
- Storage of structured and semi-structured data
- Huge volumes of data
- Requirements for scale-out architecture
- Modern application paradigms like microservices and real-time streaming

How do I choose a NoSQL database?

Database Type	Document database	Key-value store	Wide column store	Graph database
Data Models	Best when data is modeled by a set of interrelated objects, with its flexibility toward data structure making it a good general purpose database. Documents can contain nested structures for capturing complex data.	Excellent for frequent high-speed access to the same chunks of data, even if those chunks of data are large.	Best for extremely large sets of data, where querying patterns are predictable, often for supporting aggregation and analytics.	Suitable when there is a need to store and query data about the connections between related data, such as in social network contexts.

Mongodb

MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. MongoDB is developed by MongoDB Inc. and current versions are licensed under the Server Side Public License.

```
users={
 "name": {
         "first": "string",
         "last": "string"
     },
     "birth": "Date",
     "death": "Date",
     "views": "Long",
  "age": "Int"
}
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

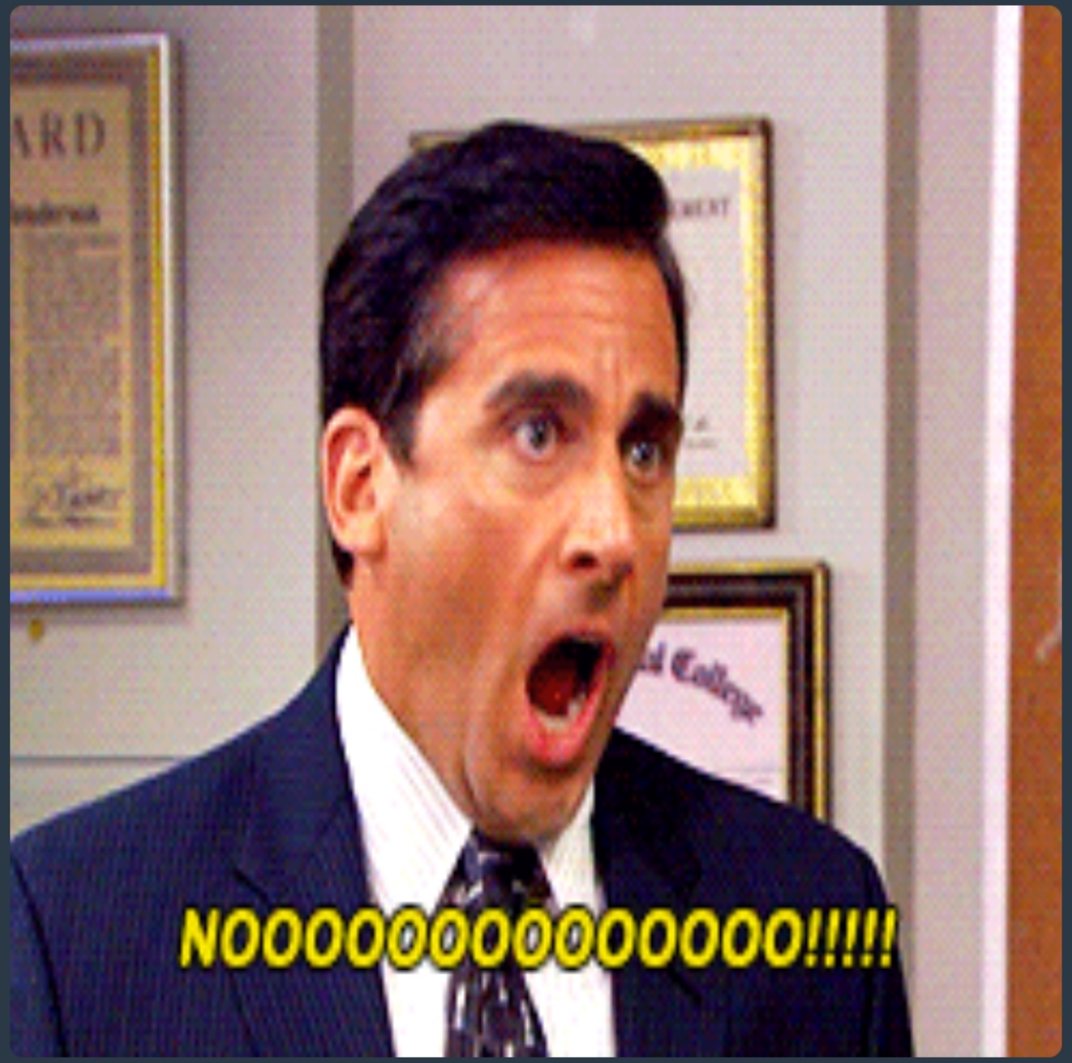
```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

```
db.users.insertOne({
     name: {
          first: 'Alan',
          last: 'Turing'
     },
     birth: new Date('Jun 23, 1912'),
     death: new Date('Jun 07, 1954'),
contribs: ['Turing machine', 'Turing test'],
     views: Long(1250000)
})
```

```
db.users.find({
     {
          birth: {
               $gt: new Date('Jun 23, 1900')
          }
     },
     {
          death: {
               $lt: new Date('Jun 23, 2000')
          }
     }
})
```

Schema Design – Relational vs. No SQL!

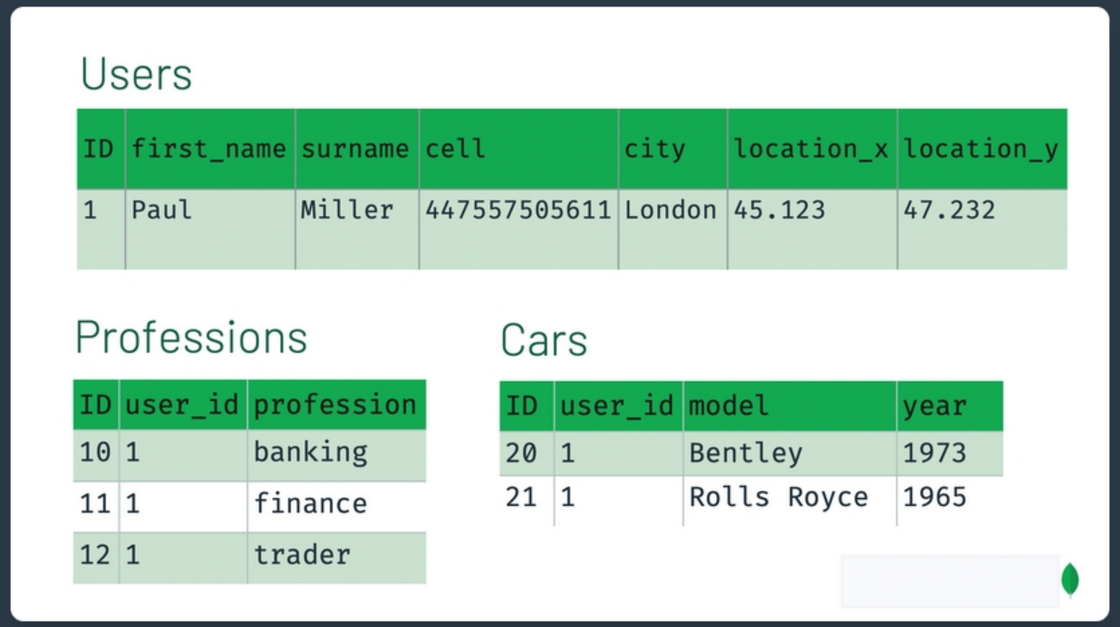
Schema Design – Relational vs. No SQL!



Schema Design for SQL

When designing a relational schema, typically, devs model their schema independent of queries. They ask themselves, "What data do I have?" Then, by using prescribed approaches, they will normalize (typically in 3rd normal form). The tl;dr of normalization is to split up your data into tables, so you don't duplicate data. Let's take a look at an example of how you would model some user data in a relational database.

In this example, you can see that the `user` data is split into separate tables and it can be JOINED together using foreign keys in the user_id column of the Professions and Cars table. Now, let's take a look at how we might model this same data in MongoDB.



MongoDB Schema Design

When you are designing your MongoDB schema design, the only thing that matters is that you design a schema that will work well for your application. Two different apps that use the same exact data might have very different schemas if the applications are used differently. When designing a schema, we want to take into consideration the following:

- Store the data
- Provide good query performance
- Require reasonable amount of hardware

```
{
    "first_name": "Paul",
    "surname": "Miller",
    "cell": "447557505611",
    "city": "London",
    "location": [45.123, 47.232],
"profession": ["banking", "finance", "trader"],
    "cars": [
        {
            "model": "Bentley",
            "year": 1973
        },
        {
            "model": "Rolls Royce",
            "year": 1965
        }
    ]
}
```

```
{
    "first_name": "Paul",
    "surname": "Miller",
    "cell": "447557505611",
    "city": "London",
    "location": [45.123, 47.232],
"profession": ["banking", "finance", "trader"],
    "cars": [
        {
            "model": "Bentley",
            "year": 1973
        },
        {
            "model": "Rolls Royce",
            "year": 1965
        }
    ]
}
```

```
{
    "first_name": "Paul",
    "surname": "Miller",
    "cell": "447557505611",
    "city": "London",
    "location": [45.123, 47.232],
"profession": ["banking", "finance", "trader"],
    "cars": [ObjectId(AAAA) , ObjectId(AAAB)]
}
```

```
[
  {
    "_id": ObjectId(AAAA),
    "model": "Bentley",
    "year": 1973
  },
  {
    "_id": ObjectId(AAAB),
    "model": "Rolls Royce",
    "year": 1965
  }
]
```

Embedding vs. Referencing

Embedding

Advantages

- You can retrieve all relevant information in a single query.

Limitations

- Large documents mean more overhead if most fields are not relevant.

Referencing

We can reference another document using document's object ID and join them together.

Advantages

- By splitting up data, you will have smaller documents.

Limitations

- a minimum of two queries required

Referencing

- One-to-One - Prefer key value pairs within the document
- One-to-Few - Prefer embedding
- One-to-Many - Prefer embedding
- One-to-Squillions - Prefer Referencing
- Many-to-Many - Prefer Referencing

NEW SQL

New SQL

NewSQL is a class of relational database management systems that seek to provide the scalability of NoSQL systems for online transaction processing (OLTP) workloads while maintaining the ACID guarantees of a traditional database system.

- surrealdb
- cockroachdb

Thank You

Hope you have good day