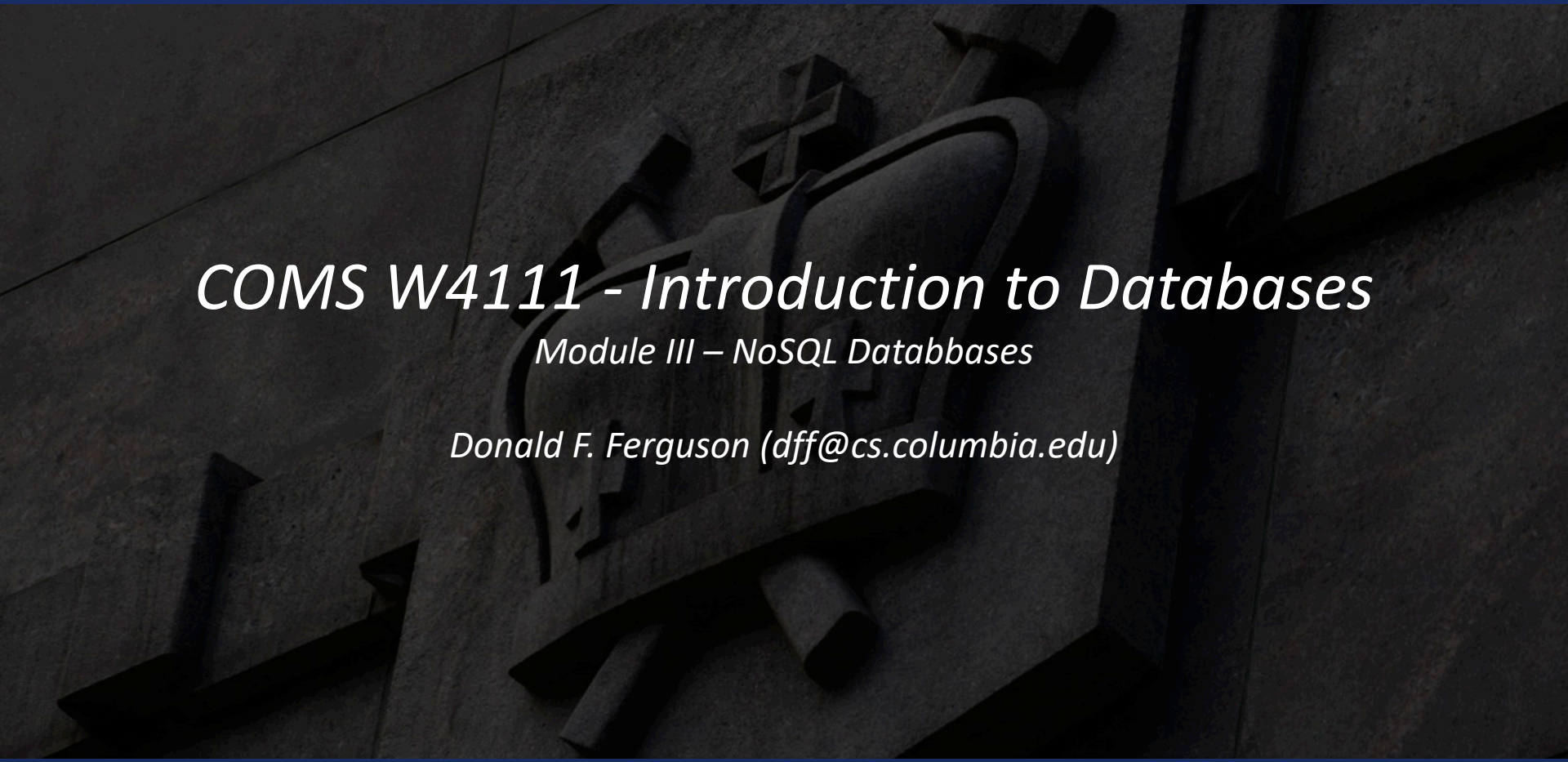


# COMS W4111 - Introduction to Databases

## Module III – NoSQL Databbases

Donald F. Ferguson (dff@cs.columbia.edu)

# *Contents*

# Contents

- Introduction
  - Course and assignment schedule update.
  - Positioning databases and the application we are not building.
  - Questions and discussion.

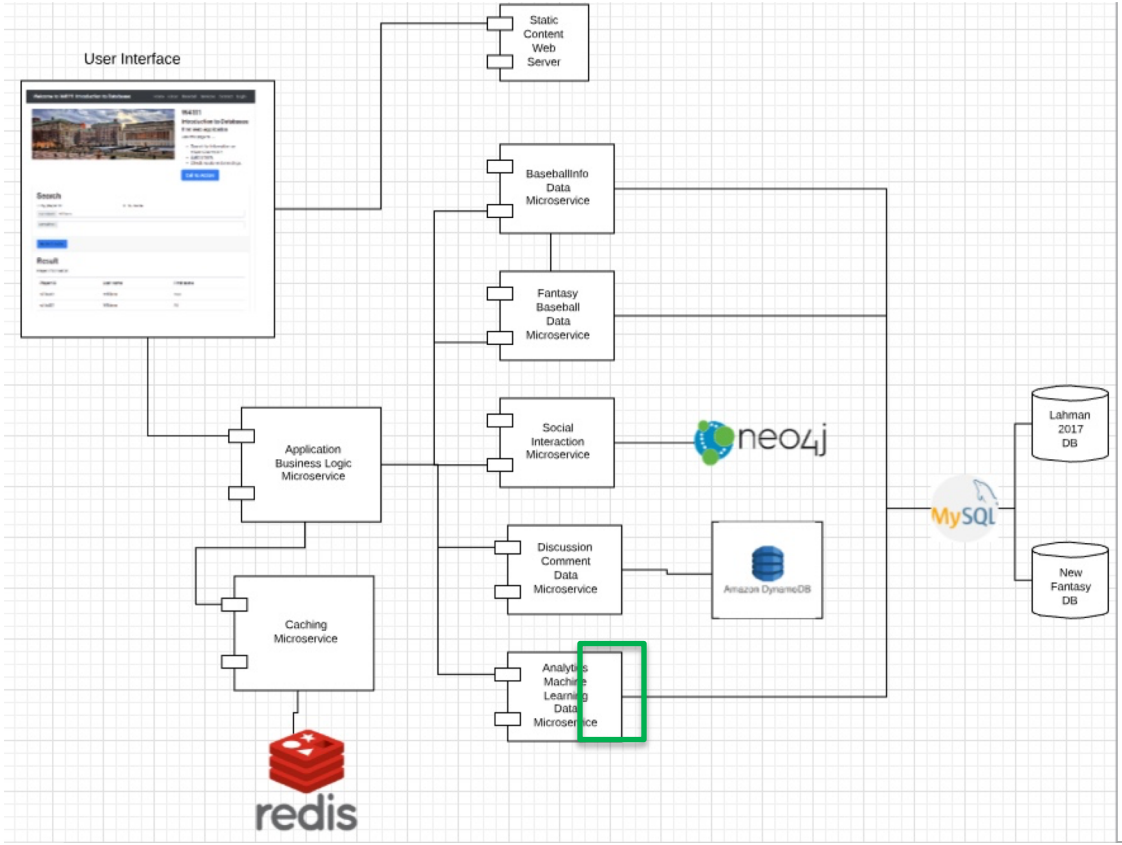
- NoSQL Databases
  - Concepts and motivation.
  - CAP “Theorem” and consistency.
  - Columnar databases. SQL done differently.
  - DynamoDB. Approaches to scalability.
  - Graph database, Neo4j, and HW4 part 1.
  - Key-Value store, Redis and HW4 part 2.

# *NoSQL Databases*

# *Introduction*

*Our Application,*
*Which we are Not Building*

# The Motivating Project

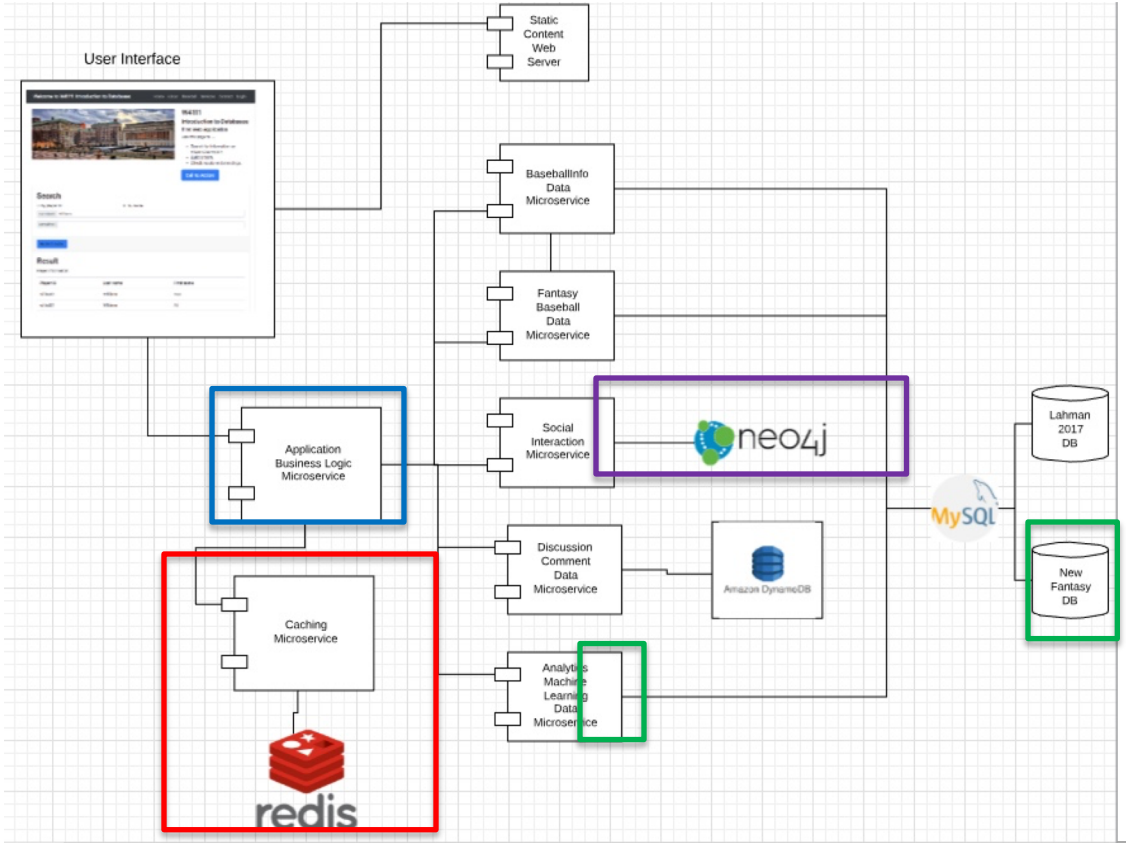



- We will not do
  - Microservices
  - REST APIs
  - Composing App. Service
- We will do:
  - HW4: Simple code that
    - Uses Neo4J
    - Uses Redis
  - ~~HW5:~~ Last lecture
    - I will give you
      - Denormalized DB
      - Python code
      - Jupyter notebook
    - ~~You will configure and run the neural network.~~

# Proposed Major Subsystems (Microservices)

- *Baseball Info Data Microse*rvice: Allows users to query and explore data about players and their performance. Date comes from Lahman2017 schema and data
- *Fantasy Baseball Data Microservice*: Allows users to create an account, define a fantasy team, define a fantasy league, assign players to teams, etc. The database is a newly designed relational database and data model
- *Social Interaction Microservice*: Allows users to follow, like, comment, etc. on teams, players, users, .. Based on Neo4J.
- *Discussion/Comment Data Microservice: S*tores comments, responses, discussion threads, etc. Based on AWS DynamoDB
- *Analytics/Machine Learning Data Microservice*: A database of analyzed and processed baseball performance interface that enables analysis and prediction using machine learning. Based on denormalized, processed Lahman 2017 data.
- *Application Business/Logic Microservice*: Implement business logic and rules, primarily governing correct operation of the fantasy league and rules.
- *Caching Microservice*: REST API and data access response cache to optimize performance.
- Static Content Web Server: Delivers HTML, CSS, images, JavaScript, etc. to web browser UI.
- *User Interface*: Browser application using AngularJS calling REST APIs/

# The Motivating Project




- We will not do
  - Microservices
  - REST APIs
  - Composing App. Service
- We will do:
  - HW4: Simple code that
    - Uses Neo4J
    - Uses Redis
  - HW5:
    - I will give you
      - Denormalized DB
      - Python code
      - Jupyter notebook
    - You will configure and run the neural network.

# *NoSQL Concepts*

# Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

A **NoSQL** (originally referring to "non SQL" or "non relational")[1] database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[2] triggered by the needs of Web 2.0 companies such as Facebook, Google, and Amazon.com.[3][4][5] NoSQL databases are increasingly used in big data and real-time web applications.[6] NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.[7][8]

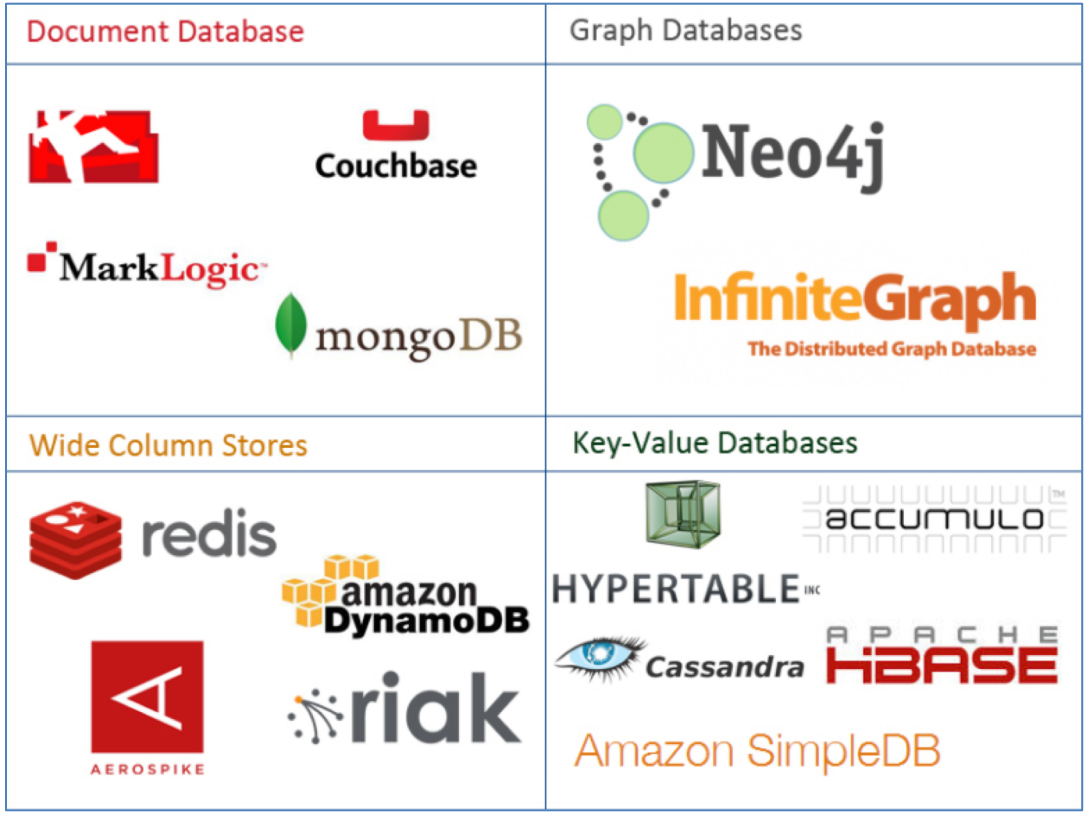
Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different 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.[9]

# Overview (I) (https://en.wikipedia.org/wiki/NoSQL)

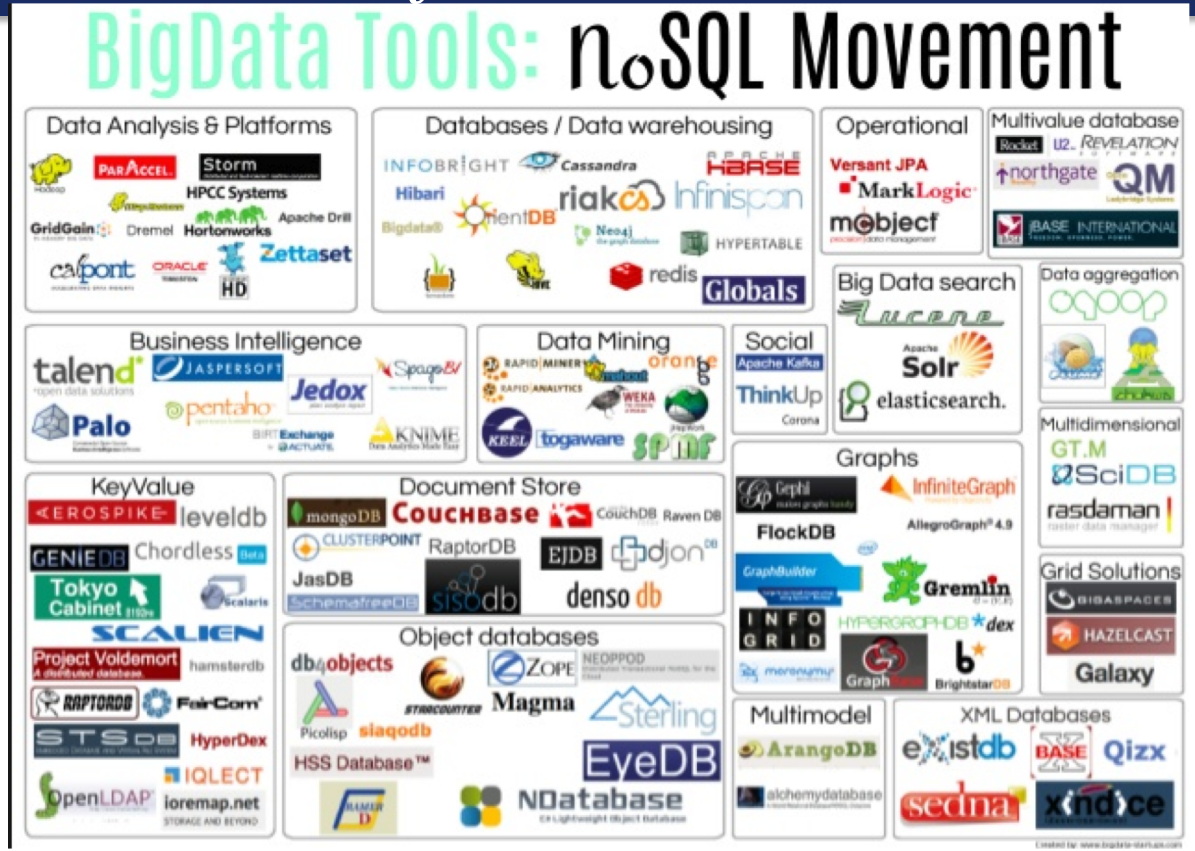
"Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[10] Most NoSQL stores lack true ACID transactions, … …

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[11] Additionally, some NoSQL systems may exhibit lost writes and other forms of data loss.[12] Fortunately, some NoSQL systems provide concepts such as write-ahead logging to avoid data loss.[13] For distributed transaction processing across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[14]

# One Taxonomy (Out-of-Date)



# Another Taxonomy



# Some Examples

https://www.slideshare.net/felixgessert/nosql-data-stores-in-research-and-practice-icde-2016-tutorial-extended-version-75275720

## The Database Explosion

### Sweetspots

IBM DB2

*RDBMS*

General-purpose ACID transactions

Greenplum

*Parallel DWH*

Aggregations/OLAP for massive data amounts

VoltDB

*NewSQL*

High throughput relational OLTP

HBASE

*Wide-Column Store*

Long scans over structured data

mongoDB

*Document Store*

Deeply nested data models

riak

*Key-Value Store*

Large-scale session storage

Neo4j the graph database

*Graph Database*

Graph algorithms & queries

redis

*In-Memory KV-Store*

Counting & statistics

cassandra

*Wide-Column Store*

Massive user-generated content

## The Database Explosion

### Cloud-Database Sweetspots

Firebase

*Realtime BaaS*

Communication and collaboration

Amazon RDS

*Managed RDBMS*

General-purpose ACID transactions

Amazon ElastiCache

*Managed Cache*

Caching and transient storage

Azure Tables

*Wide-Column Store*

Very large tables

Amazon DynamoDB

*Wide-Column Store*

Massive user-generated content

Parse

*Backend-as-a-Service*

Small Websites and Apps

bonsai

*Managed NoSQL*

Full-Text Search

Google Cloud Storage

*Object Store*

Massive File Storage

Amazon Elastic MapReduce

*Hadoop-as-a-Service*

Big Data Analytics

# Why NoSQL Databases?

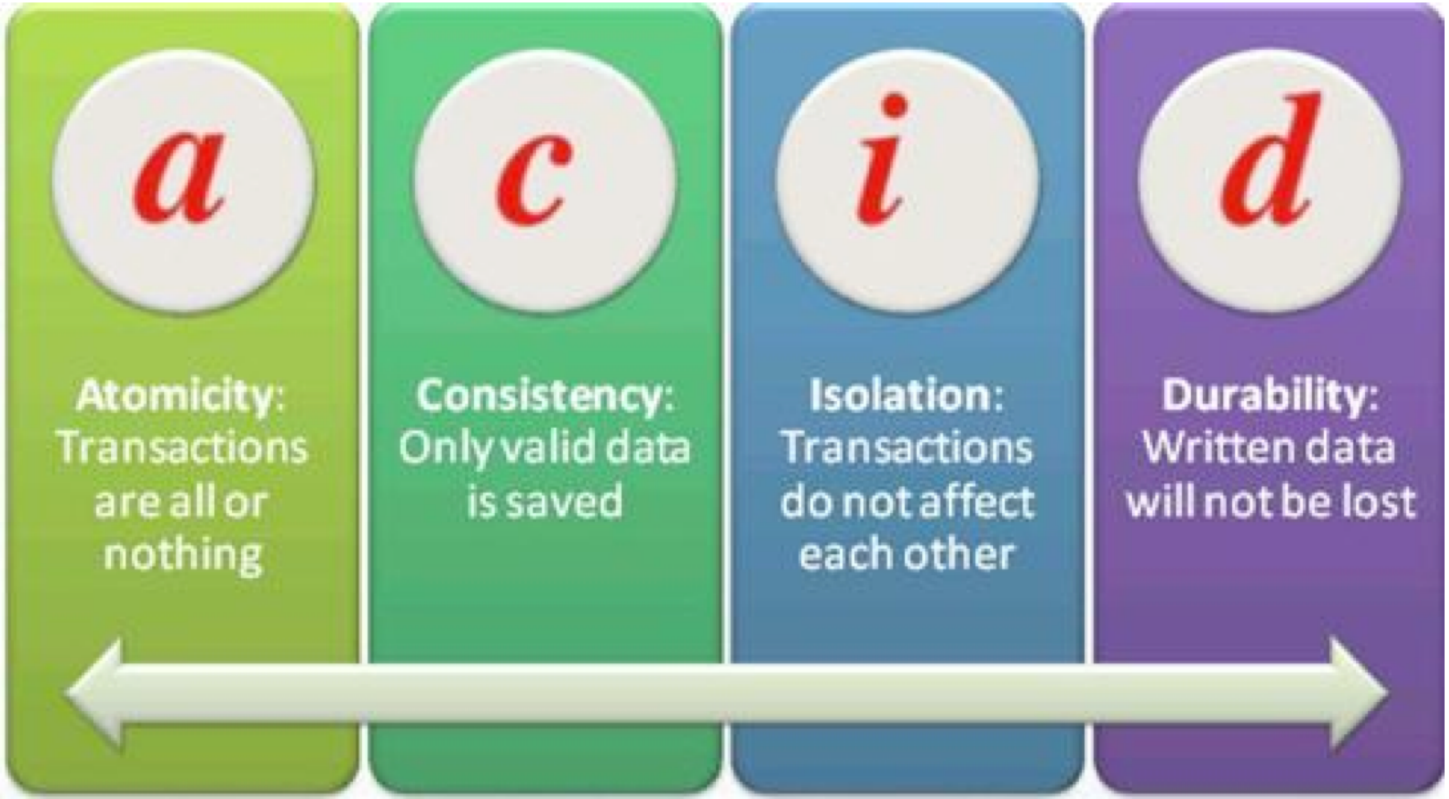
- SQL database engines provide capabilities for common application scenarios:
  - Tabular data.
  - Consistency and referential integrity.
  - Complex queries/reads.
  - Infrequent updates.

- Some application scenarios do not easily map to SQL's data model, e.g.
  - Documents (Which we have seen with simple JSON examples).
  - Social networks and graphs.
  - Highly variable, semi-structured data.

- Some applications have performance requirements that SQL engines cannot cost effectively meet.
  - Rigor of consistency and integrity.
  - Often conflicts with scalability, response time, high write rates, …

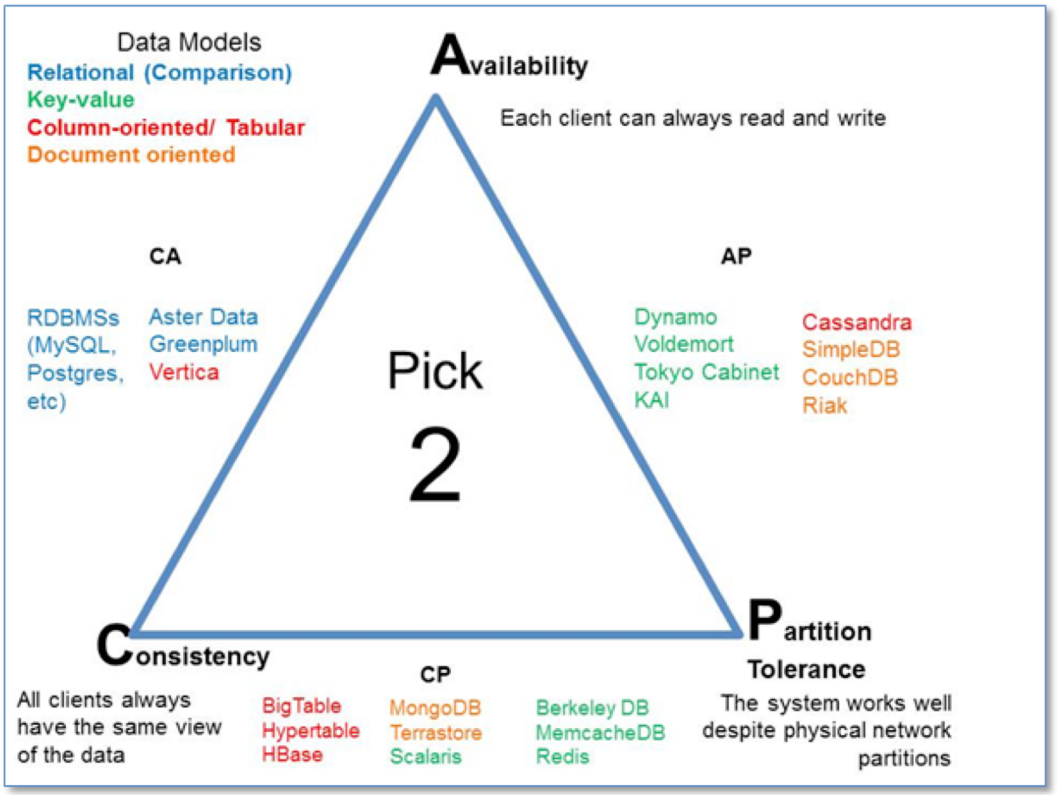
# *CAP Theorem*
# *Consistency*

# ACID



# CAP Theorem

- Consistency
  Every read receives the most recent write or an error.
- Availability
  Every request receives a (non-error) response – without 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.

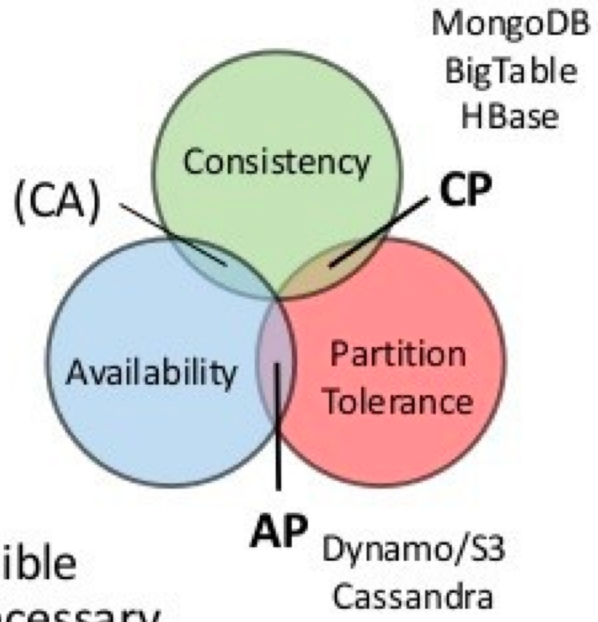


https://www.quora.com/As-per-CAP-theorem-principles-BigTable-and-MongoDB-provide-consistency-and-partition-tolerance-and-no-availability-What-does-this-mean

# CAP Theorem (2)

**CP**
- Consistent but not available under network partitions
- Lock transactions, avoid conflicts, …

**AP**
- Available but not consistent under network partitions
- Writes always possible even if no communication/synchronization is possible
- Inconsistent data, conflict resolution necessary



# CAP

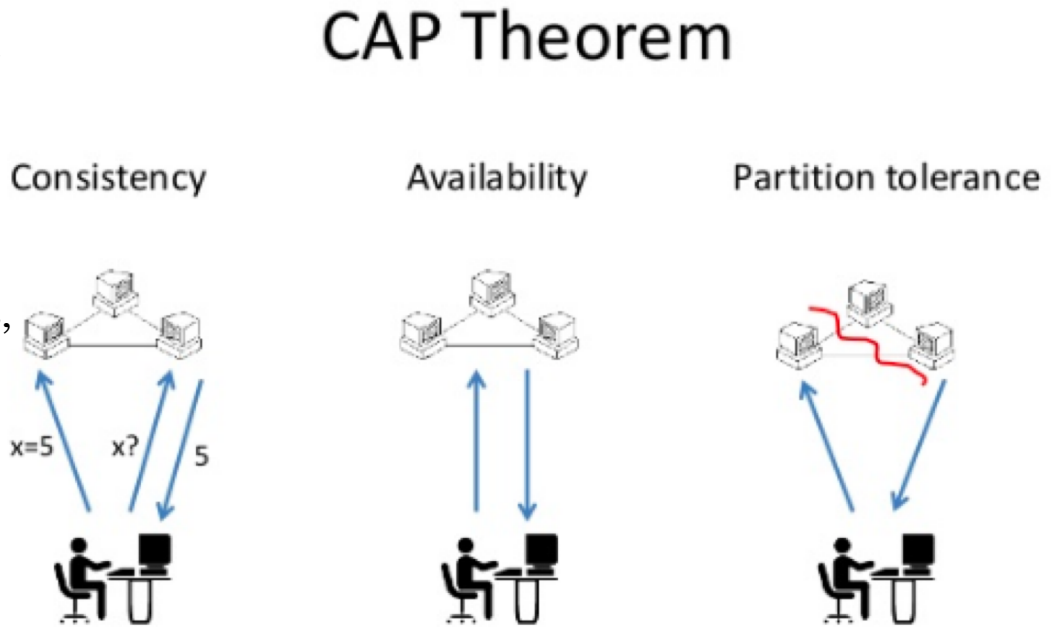
https://www.quora.com/As-per-CAP-theorem-principles-BigTable-and-MongoDB-provide-consistency-and-partition-tolerance-and-no-availability-What-does-this-mean

Not strictly “true:”

- Not formally proven in the mathematical sense of a theorem.
- The definitions of C, A and P are more complex in practice.
- Some databases have claimed to solve the problem using:
  - Very precisely synchronized clocks.
  - Near real-time conflict detection and versioning.

# Basic Idea

- Availability and Partition Tolerance
  - Require multiple copies of a data object.
  - If there is a single copy: → Not A.
    - Failed node with a single copy → read error (failure).
    - Partition → I may not be able to "get to the single copy."
- If there are multiple copies of an object, Consistency requires:
  - A write must:
    - Lock enough copies.
    - Update enough copies.
    - Unlock the copies.
  - "Enough" is N+1, if there are N-copies.
  - This is not always possible if the network is partitioned.
    - If one partition has N+1 copies.
    - The writer might be in the other partition.
  - Similar analysis applies to read.

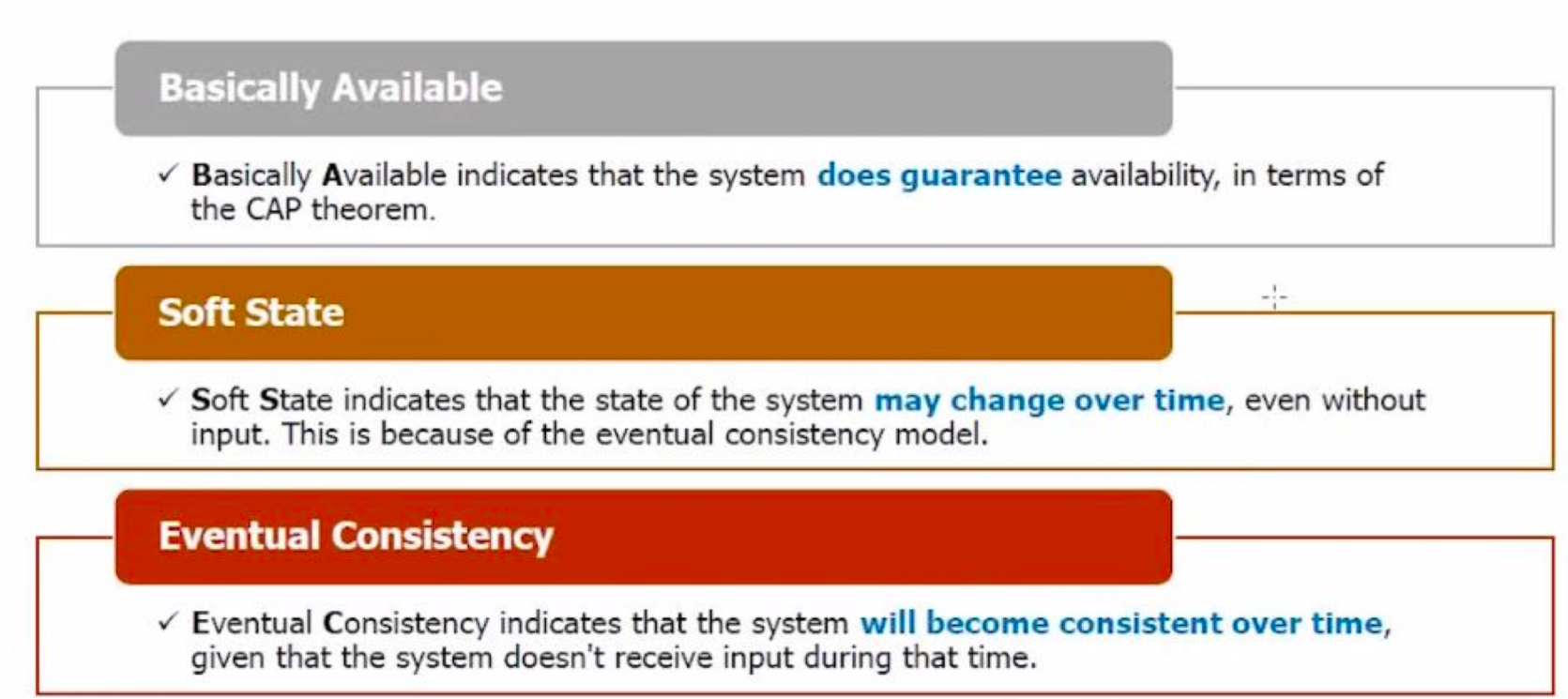


# Consistency Models

- **STRONG CONSISTENCY:** Strong consistency is a consistency model where all subsequent accesses to a distributed system will always return the updated value after the update.
- **WEAK CONSISTENCY:** It is a consistency model used in distributed computing where subsequent accesses might not always be returning the updated value. There might be inconsistent responses.
- **EVENTUAL CONSISTENCY:** Eventual consistency is a special type of weak consistency method which informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

# BASE

https://mongodbforabsolutebeginners.blogspot.com/2016/06/acid-and-cap-theroems.html



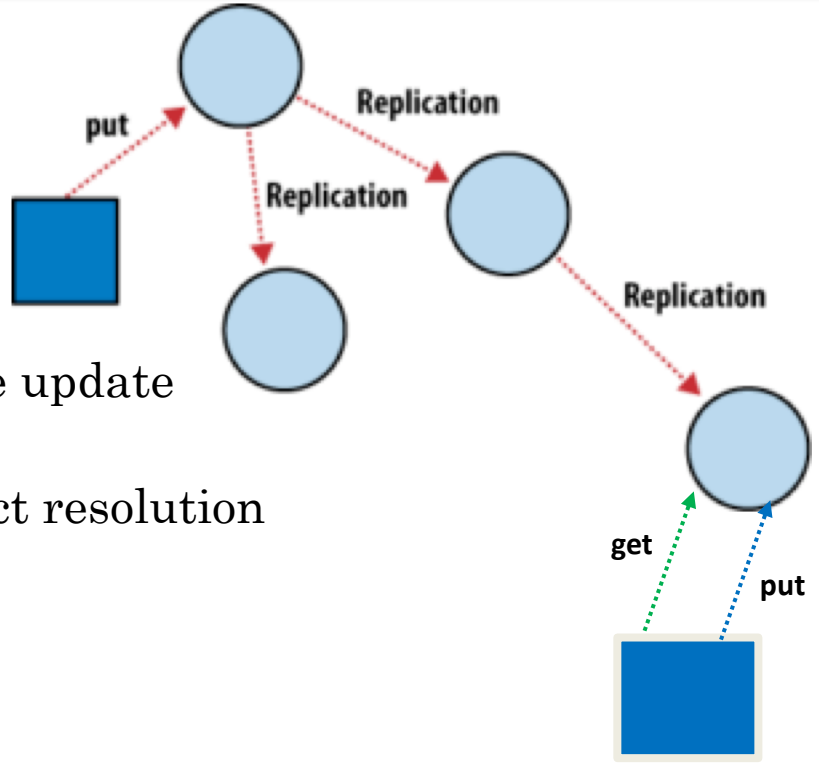
# Eventual Consistency and BASE

https://en.wikipedia.org/wiki/Eventual_consistency

- "**Eventual consistency** is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value."
- "Eventually-consistent services are often classified as providing BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) semantics, in contrast to traditional ACID(**A**tomicity, **C**onsistency, **I**solation, **D**urability) guarantees."
- " … reads eventually return the same value) and does not make safety guarantees: an eventually consistent system can return any value before it converges."
- "In order to ensure replica convergence, a system must reconcile differences between multiple copies of distributed data. This consists of two parts:
  - Exchanging versions or updates of data between servers (often known as **anti-entropy**).
  - Choosing an appropriate final state when concurrent updates have occurred, called **reconciliation**.

# Eventual Consistency

- Availability and scalability via
  - Multiple, replicated data stores.
  - Read goes to "any" replica.
  - PUT/POST/DELETE
    - Goes to any replica
    - Changes propagate asynchronously
- GET may not see the latest value if the update has not propagated to the replica.
- There are several algorithms for conflict resolution
  - Detect and handle in application.
  - Clock/change vectors/version numbers
  - Reconciliation functions.
  - Commutable operations.
  - Last writer wins.

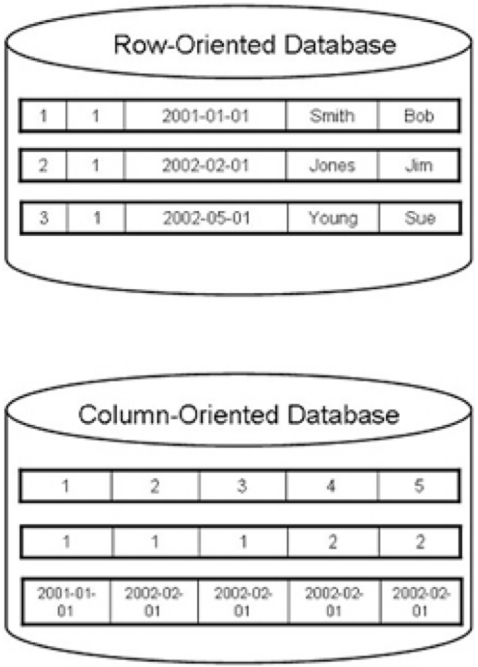


# *An Aside – Columnar*

# Columnar (Relational) Database

- Columnar and Row are both
  - Relational
  - Support SQL operations
- But differ in data storage
  - Row keeps row data together in blocks.
  - Columnar keys column data together in blocks.
- This determines performance for different types of query, e.g.
  - Columnar is extremely powerful for BI scenarios
    - Aggregation ops, e.g. SUM, AVG
    - PROJECT (do not load all of the row) to get a few columns
  - Row is powerful for OLTP. Transaction typically create and retrieve
    - One row at a time
    - All the columns of a single row.

| Emp_no | Dept_id | Hire_date | Emp_ln | Emp_fn |
|---|---|---|---|---|
| 1 | 1 | 2001-01-01 | Smith | Bob |
| 2 | 1 | 2002-02-01 | Jones | Jim |
| 3 | 1 | 2002-05-01 | Young | Sue |
| 4 | 2 | 2003-02-01 | Stemle | Bill |
| 5 | 2 | 1999-06-15 | Aurora | Jack |
| 6 | 3 | 2000-08-15 | Jung | Laura |




(https://www.dbbest.com/blog/column-oriented-database-technologies/)

# *DynamoDB*

## What is it?

*A fast and flexible NoSQL database to be used for data that requires very low latency and very large scales. Like many other NoSQL databases, it supports document and key-value data storage.*

https://medium.com/@lewisdgavin/aws-dynamodb-overview-184e53aedcd6

# Concepts

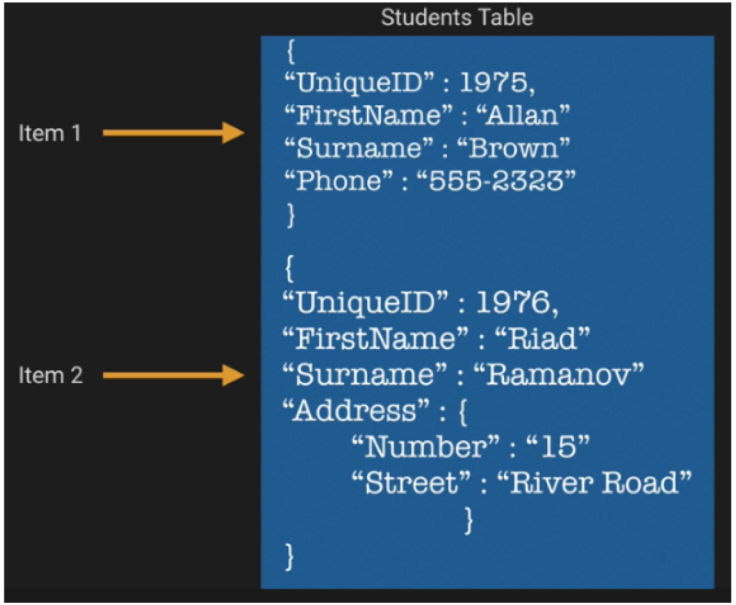
https://medium.com/@lewisdgavin/aws-dynamodb-overview-184e53aedcd6

- There are tables, “rows”, “columns:”
  - Rows can have different columns. (Flexible Schema)
  - Columns can be non-base types.
    - Arrays.
    - JSON data structures.
    - Arrays of JSON data structures containing JSON structure containing arrays, …
- Keys and indexes:
  - Primary partition key.
  - Optional sort key/index.
- Access by keys or scan.

## How does it work?

A DynamoDB database can be broken down into 3 abstractions:

- Tables: A collection of things that you want to store together
- Items: An item is just like a row in a normal database
- Attributes: A column or field in a normal database



# Primary (Partition) Key

A DynamoDB table must have a primary key. There are two possible types to choose from:

1. Partition Key—Single Attribute—this will just be a field in your data source that uniquely represents the row (e.g. an auto generated, unique product ID).
2. Partition Key & Sort Key—Composite Key—this will be a combo of two attributes that will uniquely identify the row, and how the data should naturally be sorted (e.g. Unique product ID and purchase date timestamp)

It's important that your DynamoDB partition key is unique and sparse. As this key is hashed internally and used to distribute that data for storage. This is a similar technique to Redshift and HBase that prevents hotspotting of data.

# Features

https://www.techtrainees.com/overview-of-amazon-web-services-non-relational-database-amazon-dynamodb/

## Features of Amazon DynamoDB

**Writing & Reading** – DynamoDB writes to three **Availability Zones (AZs)**, but if it is confirmed that writing to two AZs is completed, it is regarded as writing completion. It is a reading model of results consistency. If reading is done immediately after writing, old data may be returned.

**High availability & High scalability** – Since data is stored in three AZs, both availability and robustness will be high. There is no capacity limitation in storage, accounting will occur for the used capacity. Because it is not charging with the capacity secured like EBS, cost performance is also high.

**Simple Request** – One cannot implement complicated queries like RDBMS. DynamoDB is not a data manipulation in the SQL statement but a simple API operation.

**One Can Specify Throughput** – It specifies the throughput with the capacity unit (processing capacity per second).

# Versus RDS

https://www.techtrainees.com/overview-of-amazon-web-services-non-relational-database-amazon-dynamodb/

## RDS vs DynamoDB

Also, no transactions across multiple reads and writes.

1. **Consistency** – RDS has strong consistency, while DynamoDB's consistency is basically weak because of result matching (strong consistency specification is also possible).

2. **Atomicity** – Atomicity is there in RDS, while the same is Not (possible if updating within the same item) with DynamoDB.

3. **Availability** – DynamoDB is always available, while in RDS it is available with maintenance window.

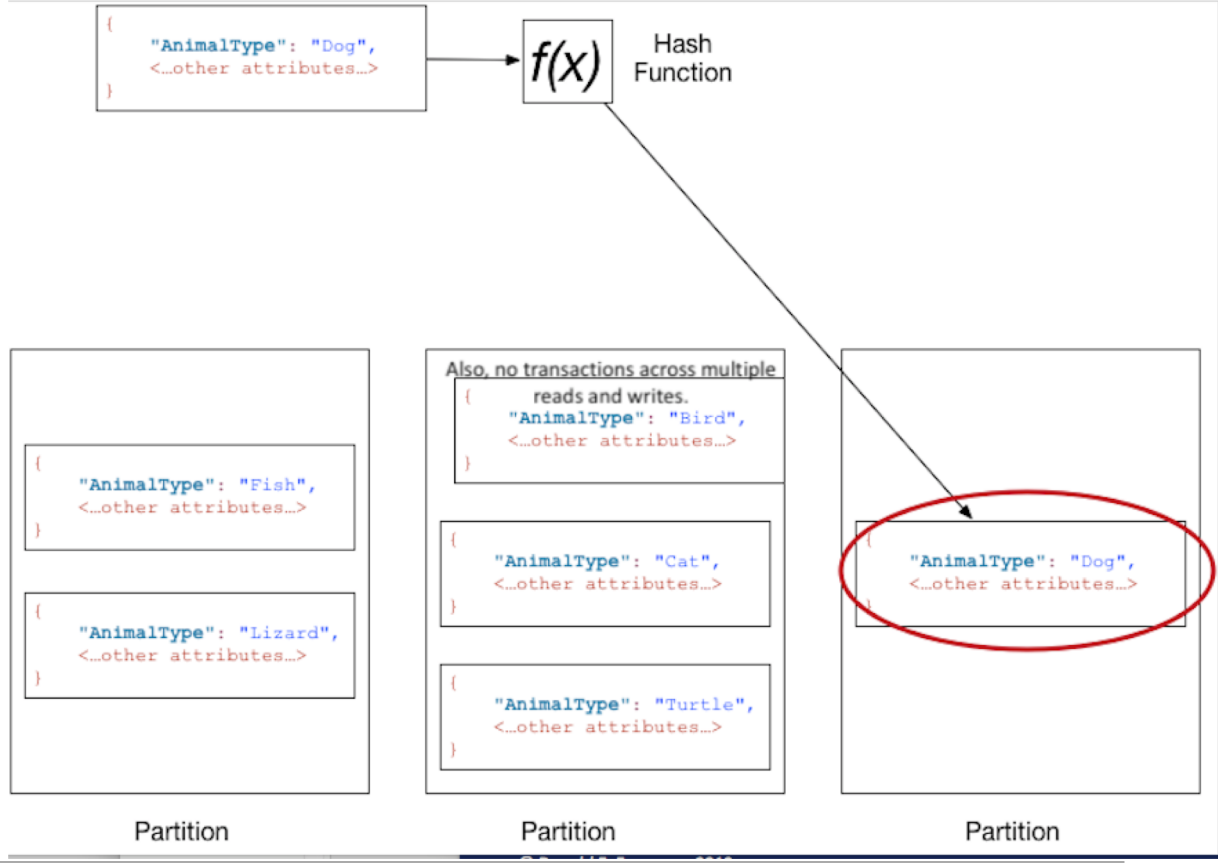
4. **Scalability** – In RDS ceiling is low due to scale-up only, while in DynamoDB scaling is out by sharding.

5. **Search Condition** – In RDS SQL where clause freely. Pre-specified key or index only in DynamoDB.

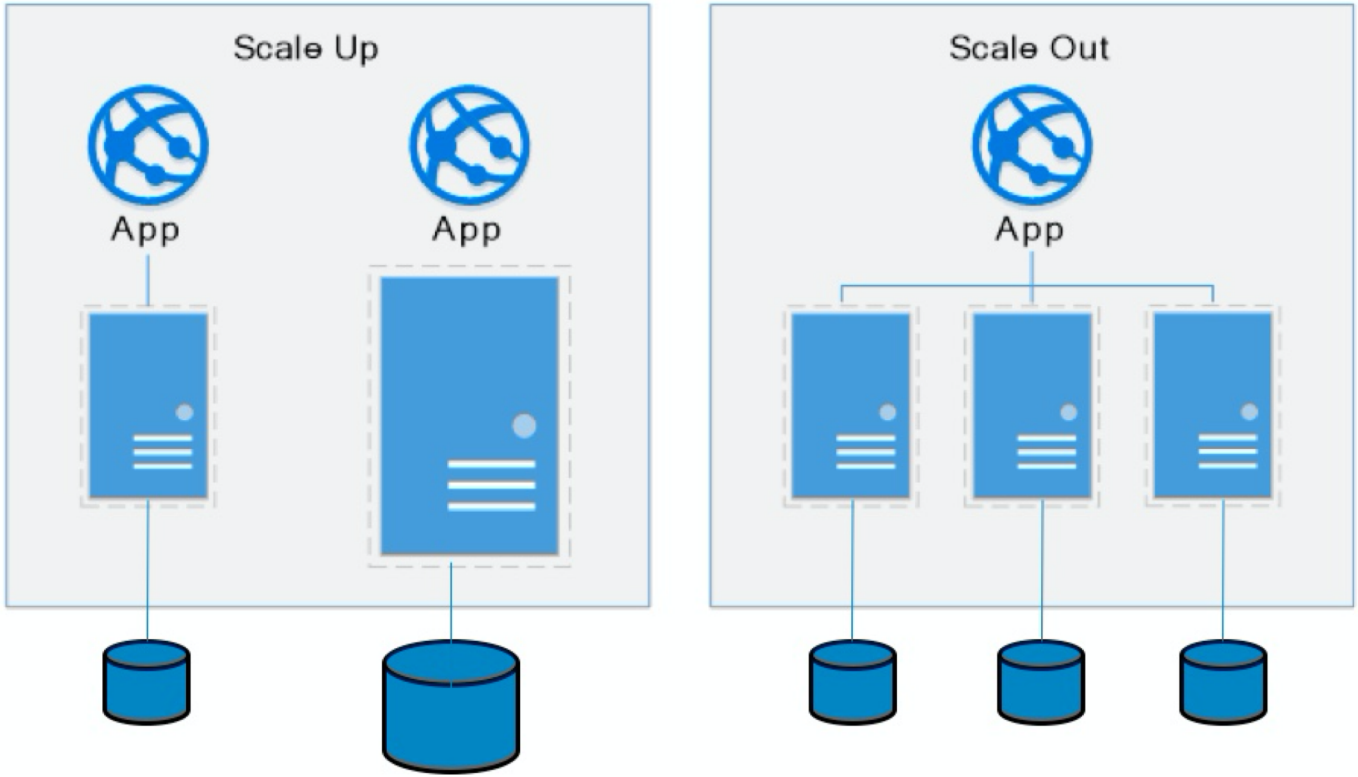
Supports scans.

# Partitions

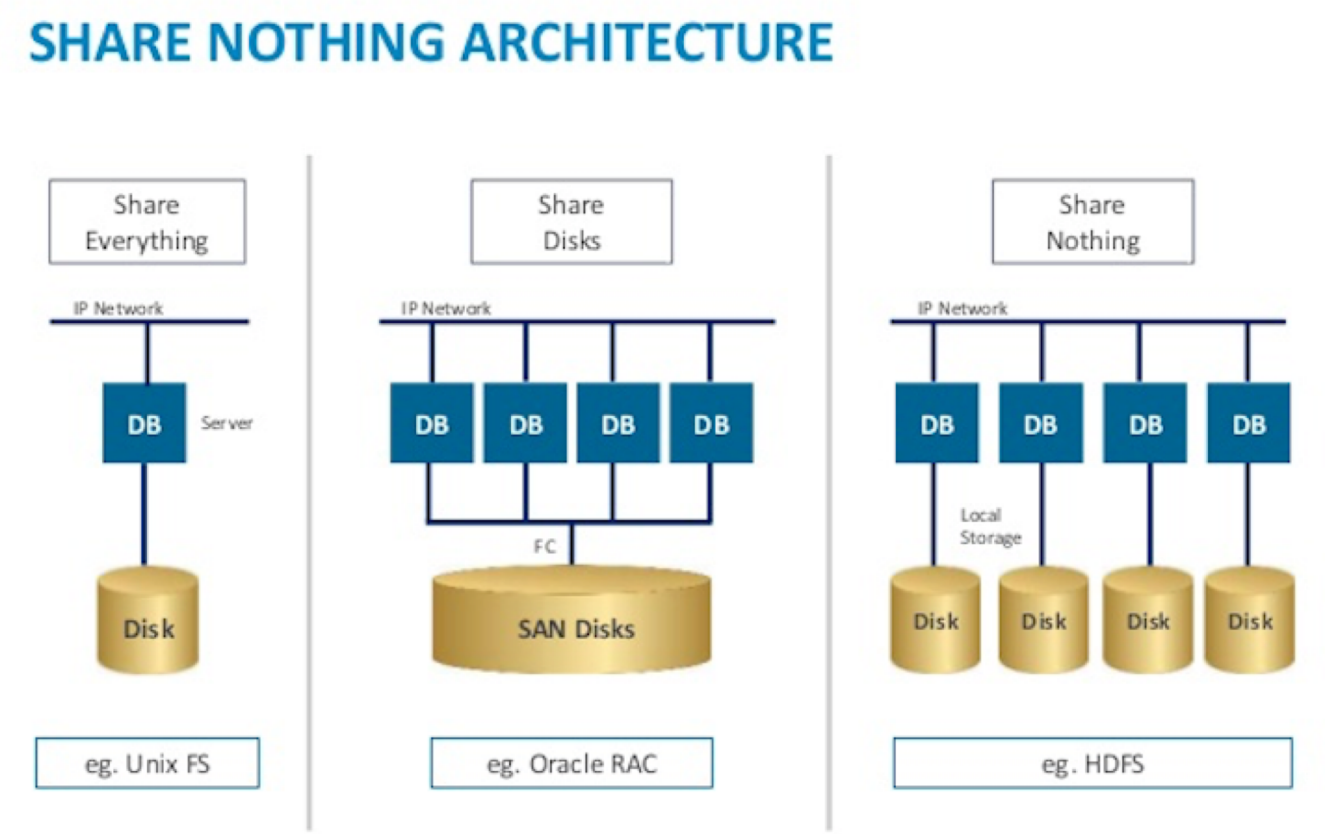
- A DynamoDB "table" has one or more *partitions*, and must define a *partition key* that is a data field on ALL entries in the table.
- DynamoDB processes operations by:
  - Getting the partition key value from the request data.
  - Hashing to determine the partition.
  - Routing the request to the partition that contains (may contain) the data.



# Scale Up versus Scale Out



# Shared Nothing – Shared Everything



# Comments

- Shared nothing/scale out works
  - Extremely well for:
    - Scans (parallel without indexes)
    - Individual item read/write/update.
  - Poorly for:
    - Referential integrity.
    - Multi-table/type operations, e.g. JOIN.
- Shared everything/scale up works
  - Extremely well for:
    - Complex, multi-table/type queries.
    - Referential integrity.
  - Poorly for:
    - Massive scalability.
    - Availability.

I conflated scale up/versus scale out with shared everything/shared nothing. There is some ability to mix and match, but in general the two patterns are scale out/shared nothing and scale up/share everything.

# Midterm Question

- You will remember from midterm that reading and saving this structure was awkward.
- This is an extremely simple document. Realistic ones are more awkward and painful.
- DynamoDB can read, write and scan these documents.
- But, but the application is responsible for referential integrity for customerNumber and productCode,

```
o_all_1 = {
            "order_number": "0123",
            "orderDate": "2003-01-06",
            "requiredDate": "2003-01-13",
            "shippedDate": "2003-01-10",
            "status": "Shipped",
            "comments": None,
            "customerNumber": 363,
    "orderdetails":
        [
        {
            "productCode": "S24_3969",
            "quantityOrdered": 49,
            "priceEach": "35.29",
            "orderLineNumber": 1
        },
        {
            "productCode": "S18_2248",
            "quantityOrdered": 50,
            "priceEach": "55.09",
            "orderLineNumber": 2
        },
        {
            "productCode": "S18_1749",
            "quantityOrdered": 30,
            "priceEach": "136.00",
            "orderLineNumber": 3
        },
        {
            "productCode": "S18_4409",
            "quantityOrdered": 22,
            "priceEach": "75.46",
            "orderLineNumber": 4
        }
    ]
}
```

# DynamoDB Code

```
def do_a_put(table_name, item):
    table = dynamodb.Table(table_name)

    response = table.put_item(
        Item=item
    )

    #print("Put response = ", json.dumps(response))
    return response


result = do_a_put('orders', o_all_1)

print(json.dumps(result, indent=4))
```

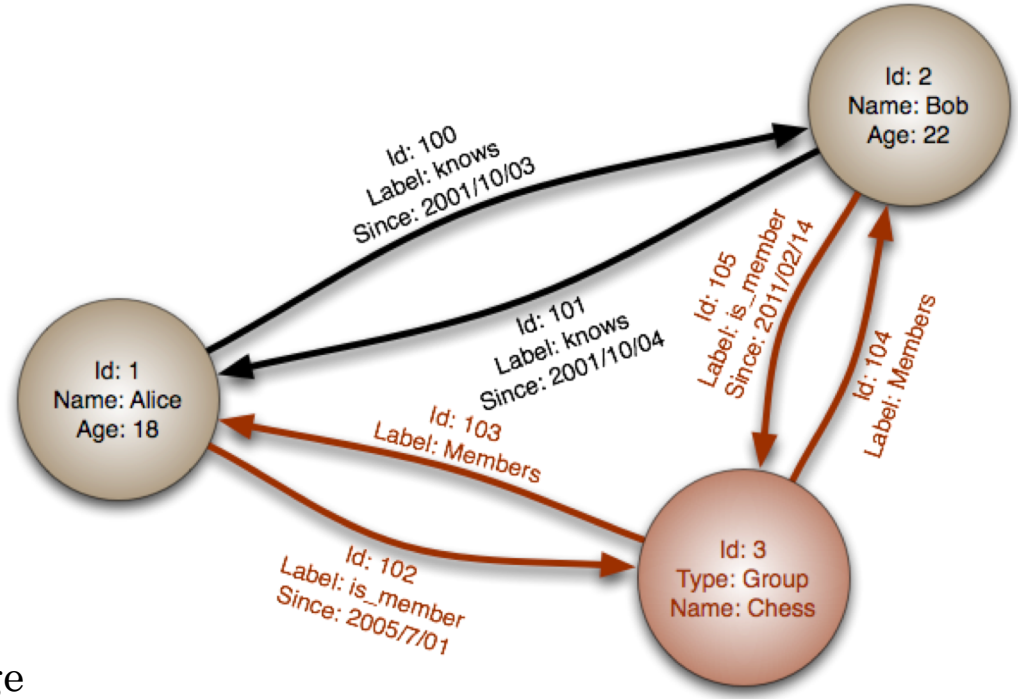
```
o_all_1 = {
            "order_number": "0123",
            "orderDate": "2003-01-06",
            "requiredDate": "2003-01-13",
            "shippedDate": "2003-01-10",
            "status": "Shipped",
            "comments": None,
            "customerNumber": 363,
    "orderdetails":
        [
        {
            "productCode": "S24_3969",
            "quantityOrdered": 49,
            "priceEach": "35.29",
            "orderLineNumber": 1
        },
        {
            "productCode": "S18_2248",
            "quantityOrdered": 50,
            "priceEach": "55.09",
            "orderLineNumber": 2
        },
        {
            "productCode": "S18_1749",
            "quantityOrdered": 30,
            "priceEach": "136.00",
            "orderLineNumber": 3
        },
        {
            "productCode": "S18_4409",
            "quantityOrdered": 22,
            "priceEach": "75.46",
            "orderLineNumber": 4
        }
    ]
}
```

- There are ways to scan to find orders based on *status, customerNumber, …*
- No easy way to find orders based on *orderDetails* properties.

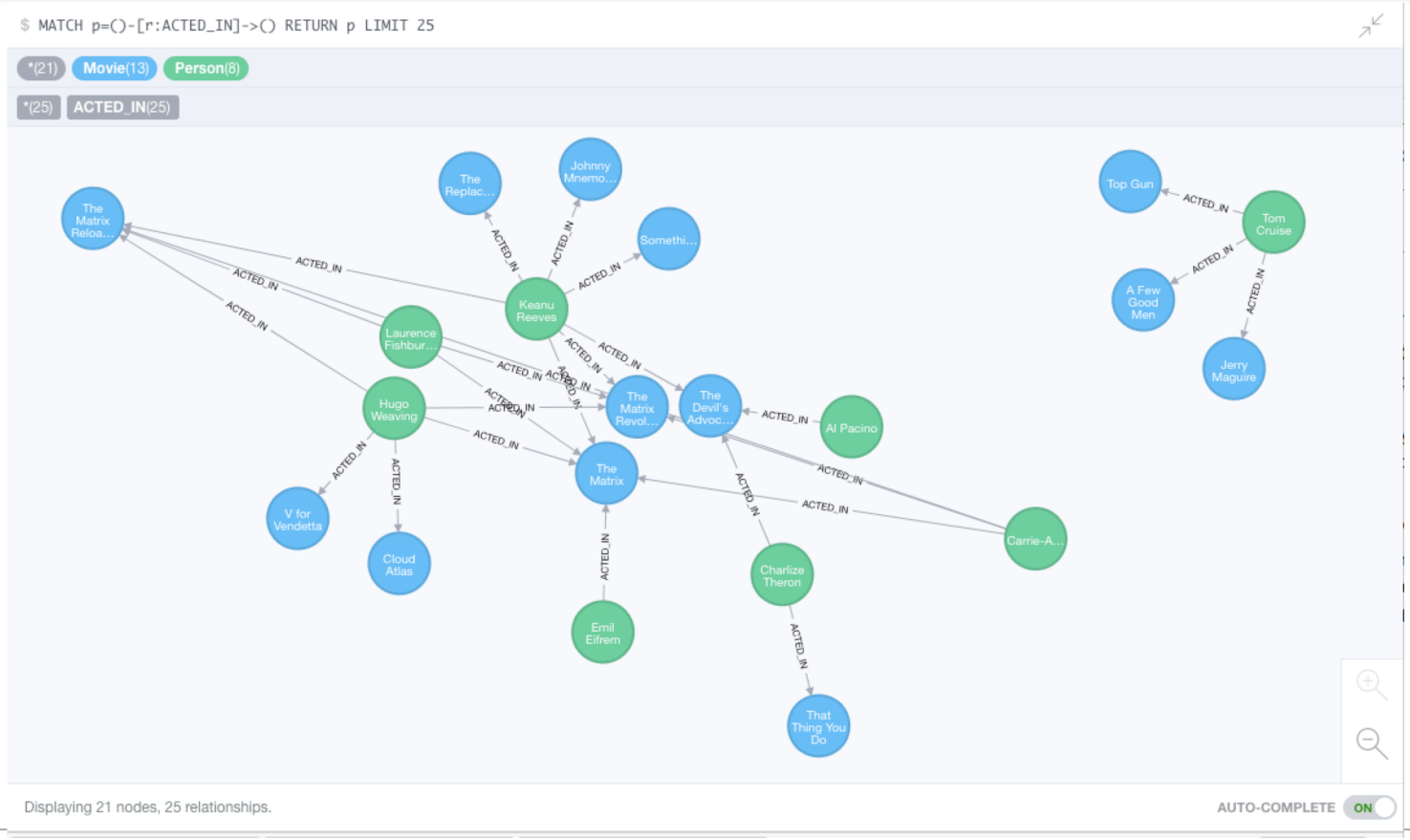
# *Graph Databases*

# Graph Database

- Exactly what it sounds like
- Two core types
  - Node
  - Edge (link)
- Nodes and Edges have
  - Label(s) = "Kind"
  - Properties  (free form)
- Query is of the form
  - p1(n)-p2(e)-p3(m)
  - n, m are nodes; e is an edge
  - p1, p2, p3 are predicates on labels



# Neo4J Graph Query



# Why Graph Databases?

- Schema Less and Efficient storage of Semi Structured Information
- No O/R mismatch – very natural to map a graph to an Object Oriented language like Ruby.
- Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- Very natural to express graph related problem with traversals (recommendation engine, find shortest parth etc..)
- Seamless integration with various existing programming languages.
- ACID Transaction with rollbacks support.
- Whiteboard friendly – you use the language of node,properties and relationship to describe your domain (instead of e.g. UML) and there is no need to have a complicated O/R mapping tool to implement it in your database. You can say that Neo4j is "Whiteboard friendly" !(http://video.neo4j.org/JHU6F/live-graph-session-how-allison-knows-james/)

# Social Network "path exists" Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4

| | # persons | query time |
|---|---|---|
| Relational database | 1000 | 2000ms |
| Neo4j | 1000 | 2ms |
| Neo4j | 1000000 | 2ms |

Graph databases are

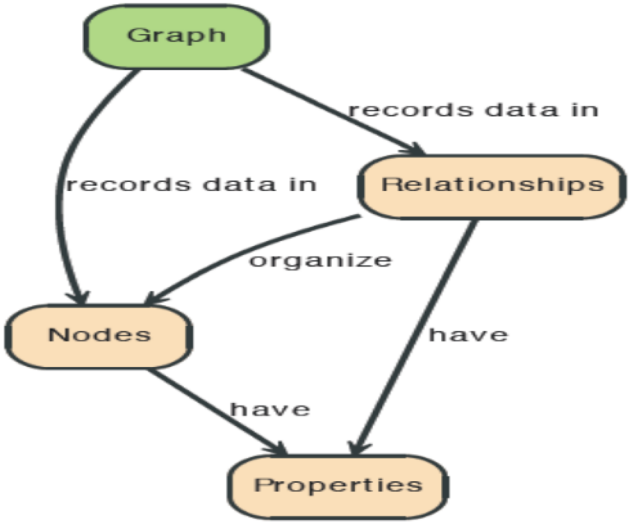
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

# What are graphs good for?

- Recommendations
- Business intelligence
- Social computing
- Geospatial
- Systems management
- Web of things
- Genealogy
- Time series data
- Product catalogue
- Web analytics
- Scientific computing (especially bioinformatics)
- Indexing your *slow* RDBMS
- And much more!

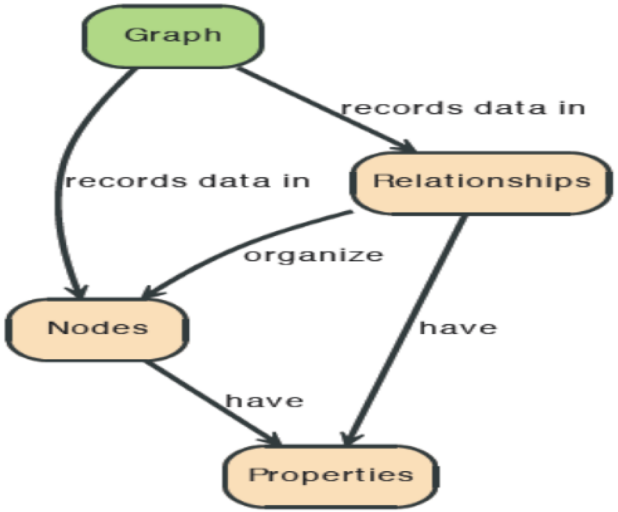
# Graphs

- "A Graph —records data in→ Nodes —which have→ Properties"



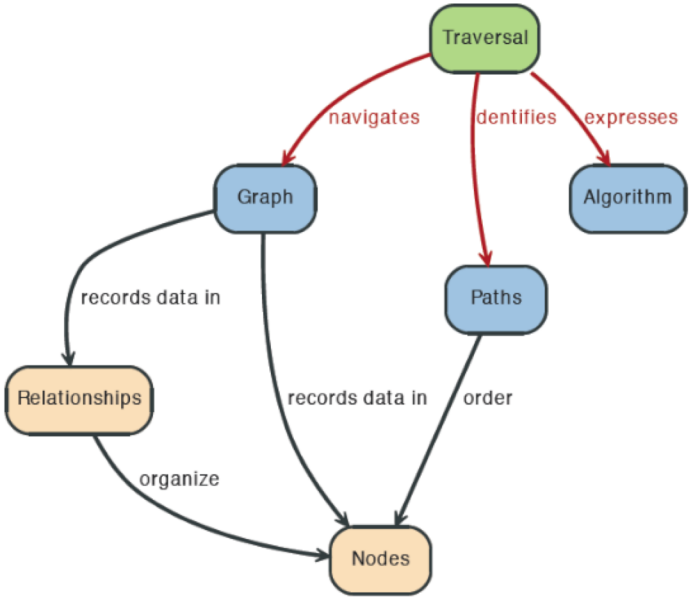
# Graphs

- "Nodes —are organized by→ Relationships — which also have→ Properties"



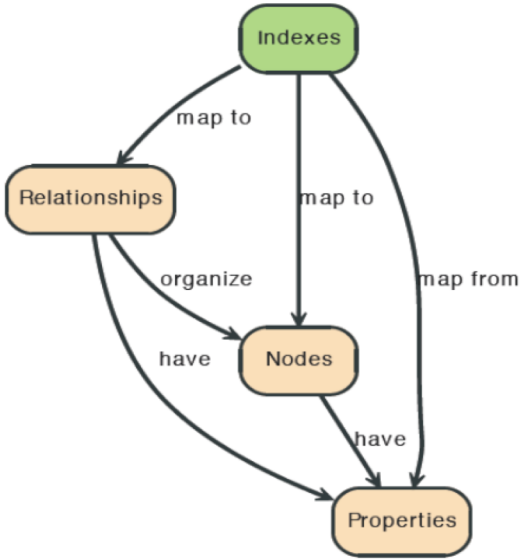
# Query a graph with Traversal

- "A Traversal —navigates→ a Graph; it — identifies→ Paths —which order→ Nodes"

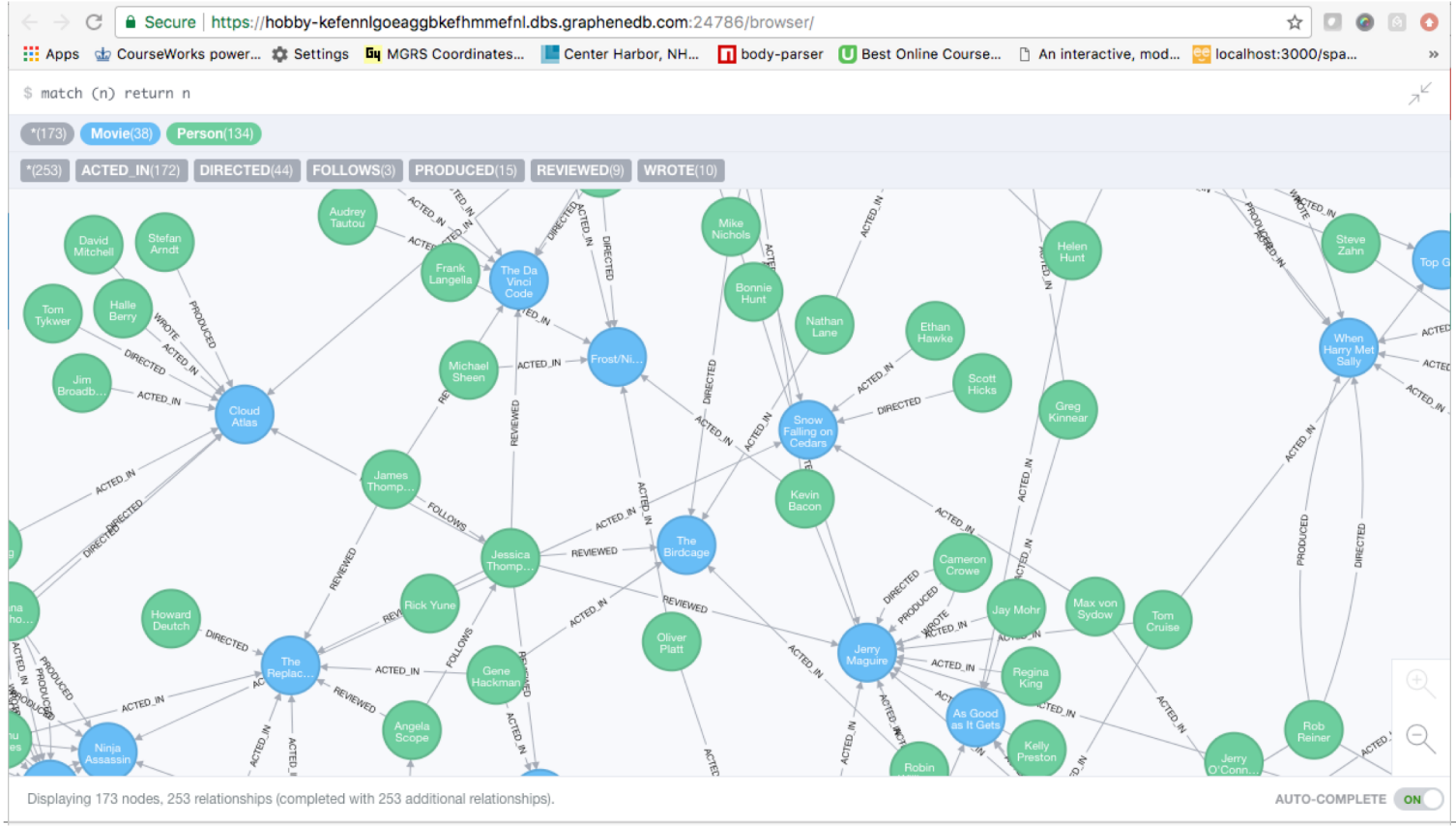


# Indexes

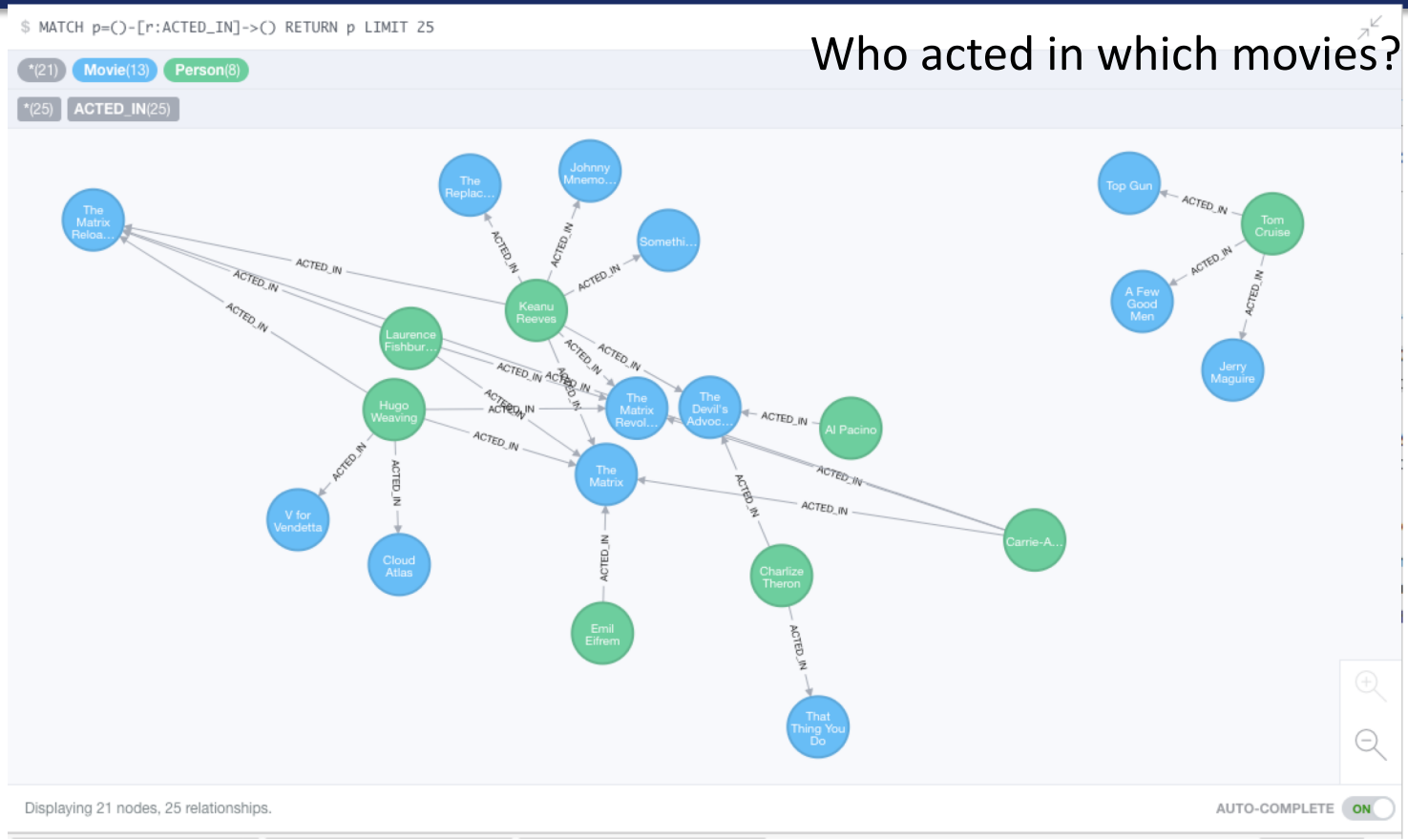
- "An Index —maps from→ Properties —to either→ Nodes or Relationships"



# A Graph Database (Sample)



# Neo4J Graph Query



# Big Deal. That is just a JOIN.

- Yup. But that is simple.
- Try writing the queries below in SQL.

The Movie Graph

## Recommend

Let's recommend new co-actors for Tom Hanks. A basic recommendation approach is to find connections past an immediate neighborhood which are themselves well connected.

For Tom Hanks, that means:

1. Find actors that Tom Hanks hasn't yet worked with, but his co-actors have.
2. Find someone who can introduce Tom to his potential co-actor.

Extend Tom Hanks co-actors, to find co-co-actors who haven't work with Tom Hanks...

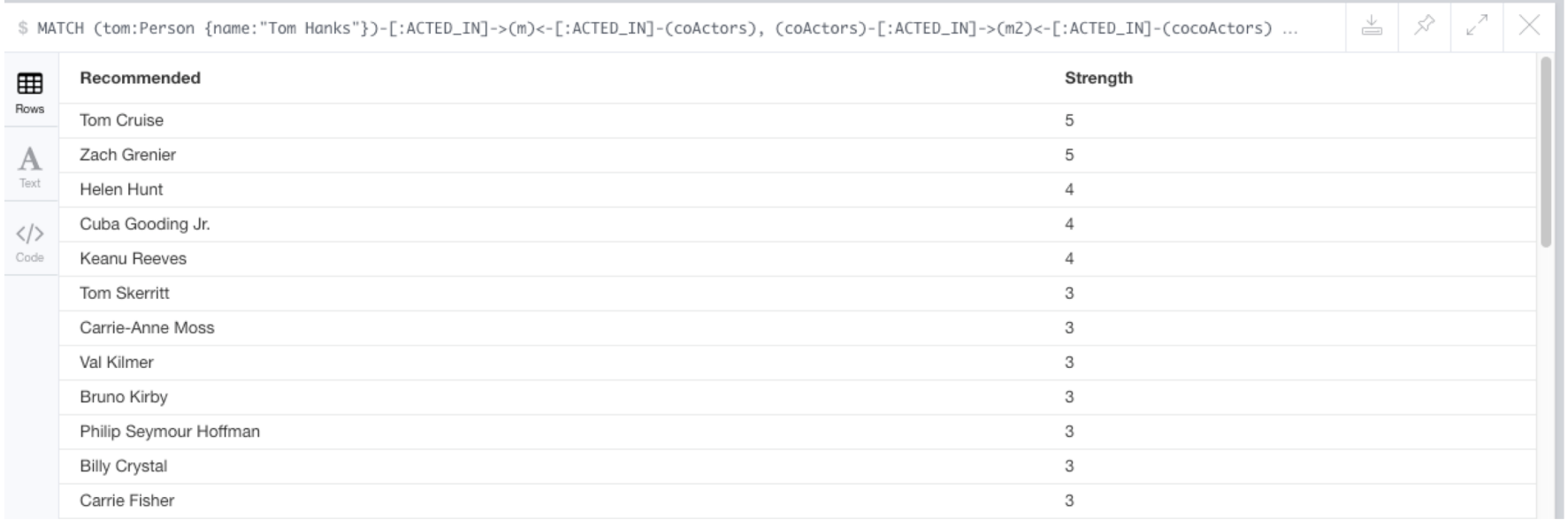
```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Find someone to introduce Tom Hanks to Tom Cruise

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

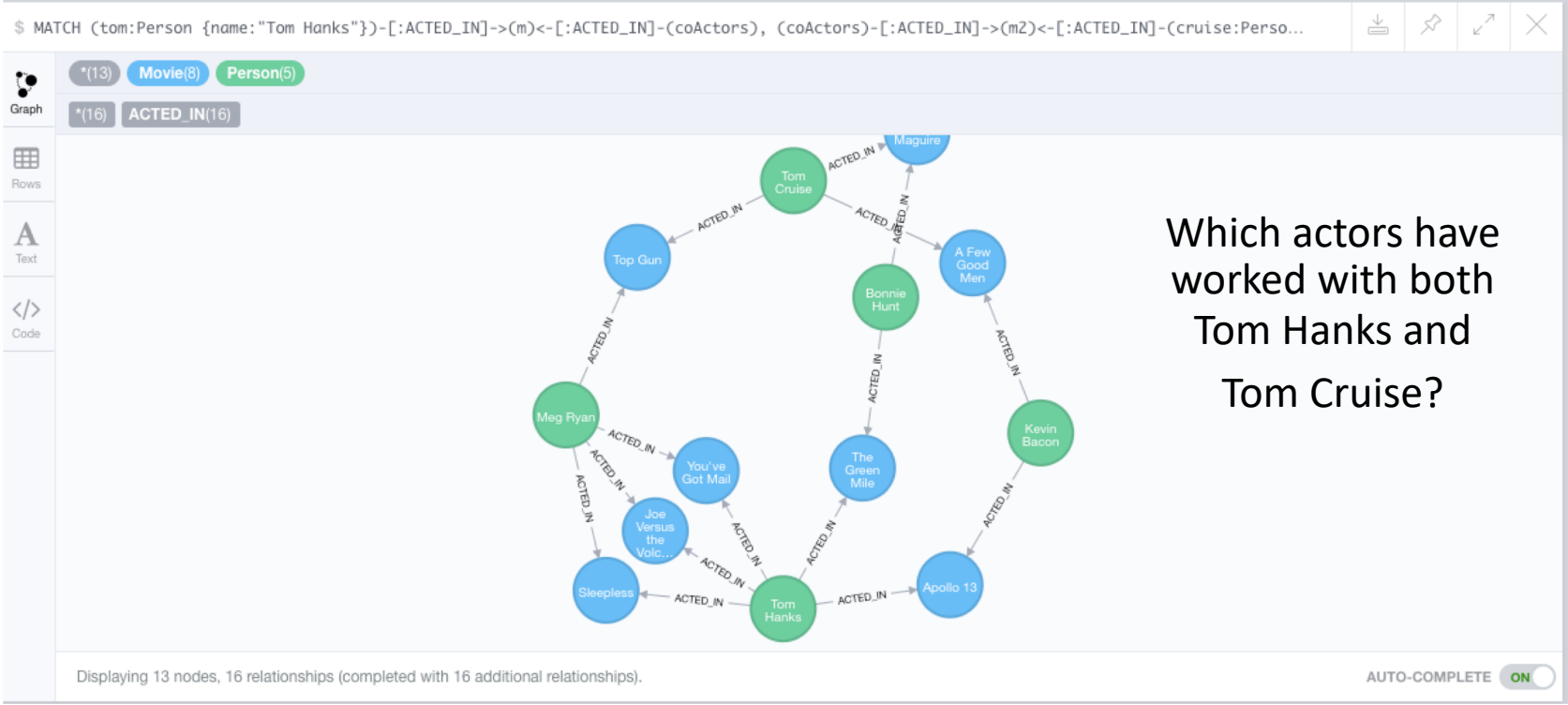
# Recommend

```
1 MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
2       (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
3 WHERE NOT (tom)-[:ACTED_IN]->(m2)
4 RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```




| Recommended | Strength |
|---|---|
| Tom Cruise | 5 |
| Zach Grenier | 5 |
| Helen Hunt | 4 |
| Cuba Gooding Jr. | 4 |
| Keanu Reeves | 4 |
| Tom Skerritt | 3 |
| Carrie-Anne Moss | 3 |
| Val Kilmer | 3 |
| Bruno Kirby | 3 |
| Philip Seymour Hoffman | 3 |
| Billy Crystal | 3 |
| Carrie Fisher | 3 |

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

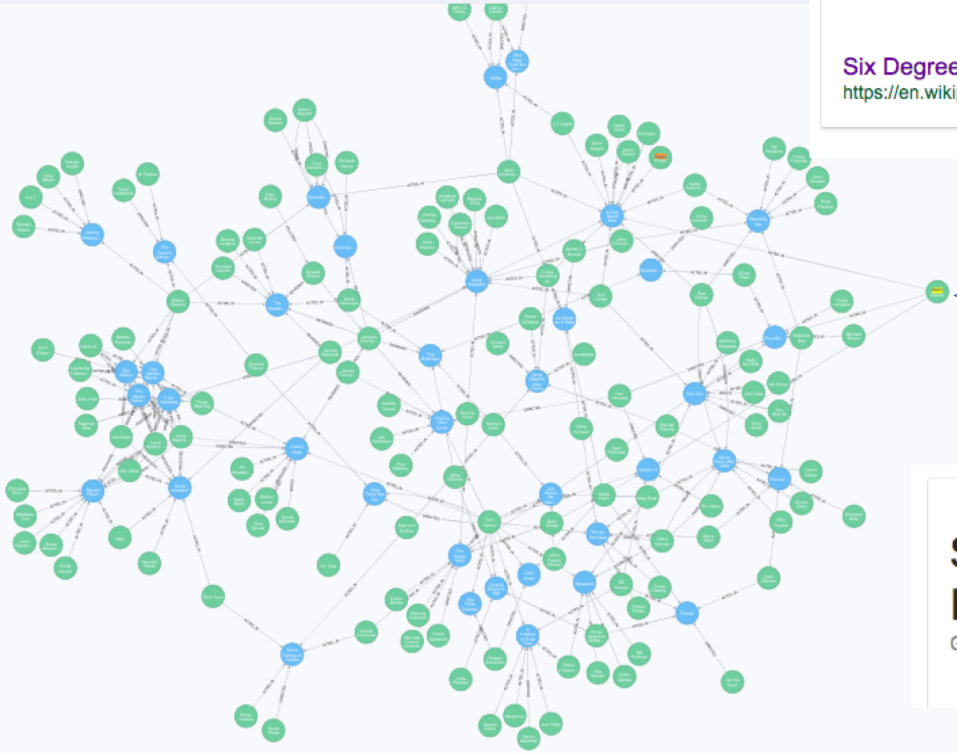



Which actors have worked with both Tom Hanks and Tom Cruise?

```
$ MATCH (s:Person { name: 'Kevin Bacon' })-[*0..6]-(m) return s,m
```

*(171) Movie(38) Person(133)

*(253) ACTED_IN(172) DIRECTED(44) FOLLOWS(3) PRODUCED(15) REVIEWED(9) WROTE(10)

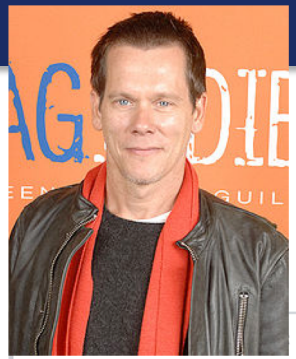


Six **Degrees of Kevin Bacon** is a parlour game based on the "six **degrees** of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin Bacon**.

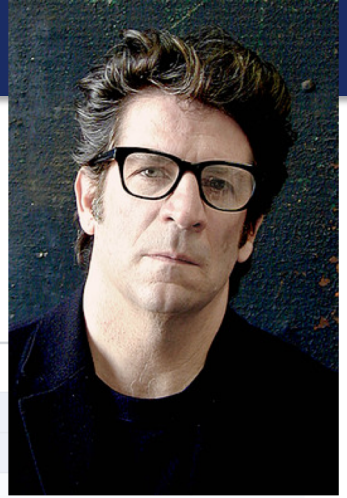
Six Degrees of Kevin Bacon - Wikipedia
https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon

About this result Feedback

Six Degrees of Kevin Bacon
Game



# How do you get from Kevin Bacon to Robert Longo?

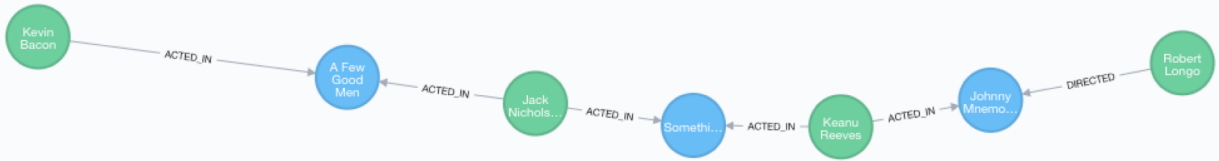


```
'Kevin Bacon' }),(robert:Person { name: 'Robert Longo' }), p = shortestPath((kevin)-[*..15]-(robert)) RETURN p
```

Graph | Rows | Text | Code

*(7) Movie(3) Person(4)

*(6) ACTED_IN(5) DIRECTED(1)



# *An Interlude on Frameworks*

# Object “Relational” Mapping

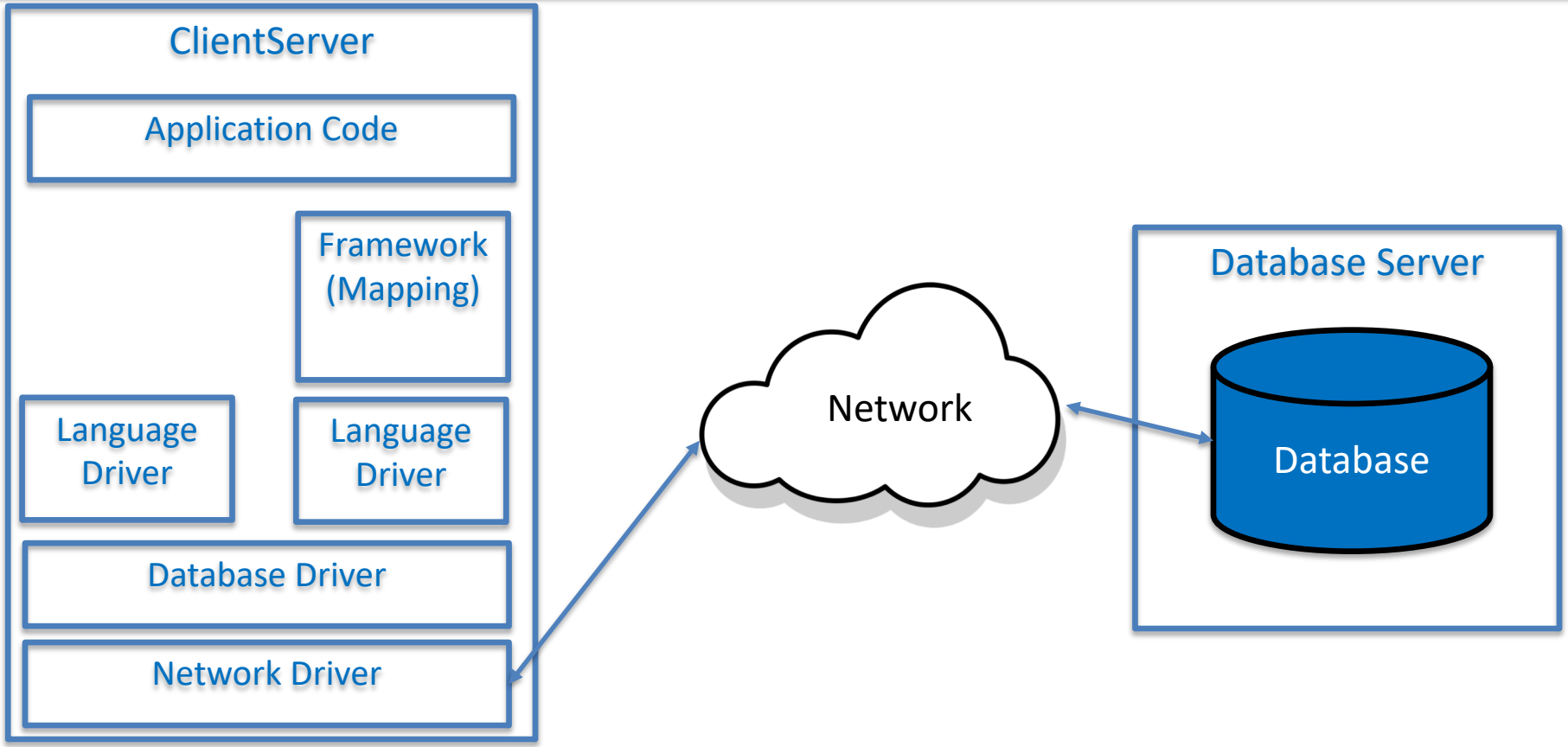
**Object-relational mapping** (**ORM**, **O/RM**, and **O/R mapping tool**) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to construct their own ORM tools.

In object-oriented programming, data-management tasks act on objects that are almost always non-scalar values. For example, an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address-book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

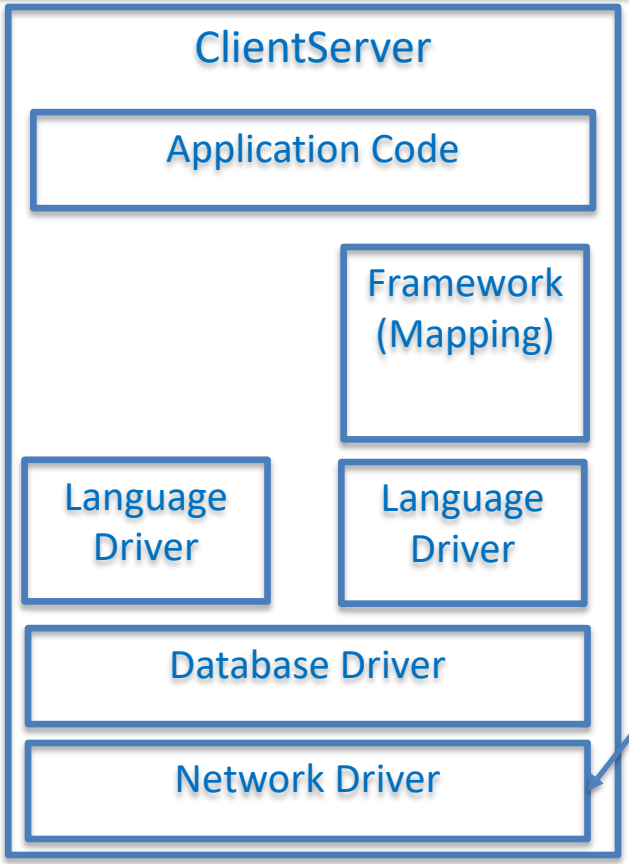
However, many popular database products such as SQL database management systems (DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping implements the first approach.[1]

The heart of the problem involves translating the logical representation of the objects into an atomized form that is capable of being stored in the database while preserving the properties of the objects and their relationships so that they can be reloaded as objects when needed. If this storage and retrieval functionality is implemented, the objects are said to be persistent.

# Simplistic Overview



# Simplistic Overview

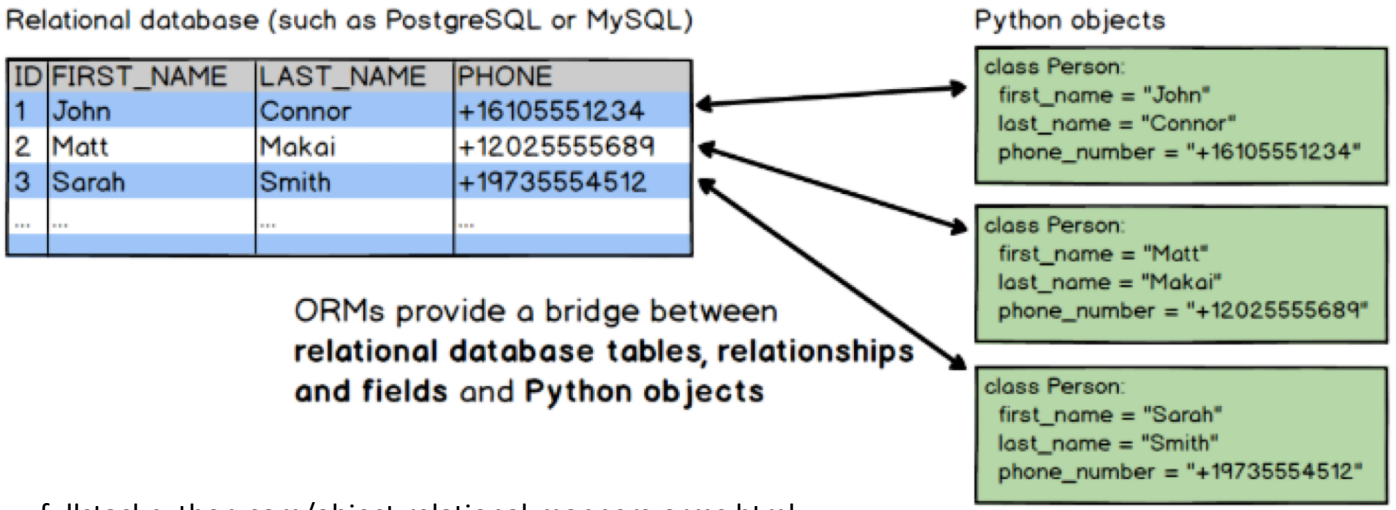



- The application code is what you write to implement your homework (project)
- Network driver is part of the operating system and sends/receives message using HTTP, TCP/IP, etc.
- Database driver uses network driver to send commands and receive response, e.g.
  - SELECT * FROM …
  - Match (n:Node) return …
- The commands and responses are in string/byte format.
- The language driver connects to database driver to language concepts in a library, e.g. functions, object.
- Framework/Mapping "reduces" the mismatch between
  - Language model.
  - Database model.

# Object "Relational" Mapping – **Simple** Python Example

## Object-relational mappers (ORMs)

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code.




https://www.fullstackpython.com/object-relational-mappers-orms.html

## SQLAlchemy

http://www.sqlalchemy.org/

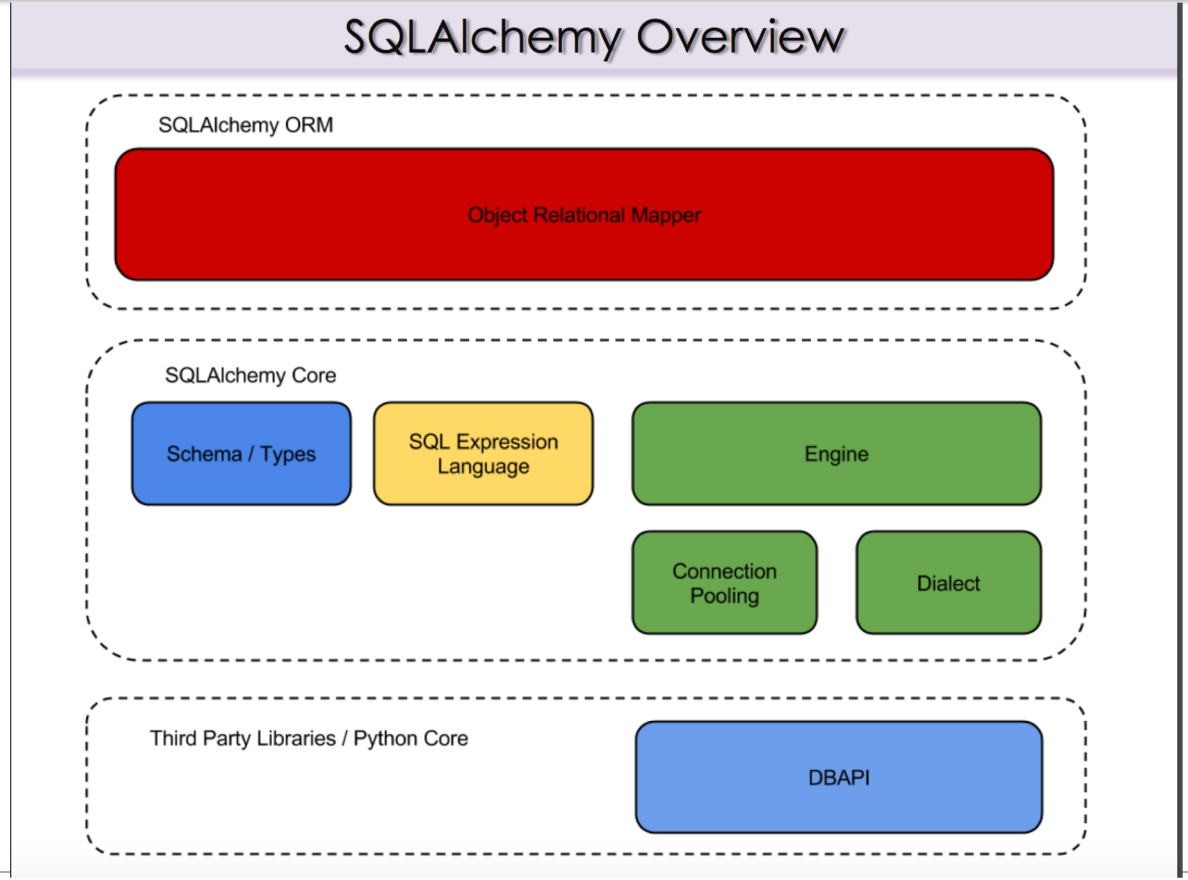
First release in 2005

Now at version 1.0.8

What is it

- Provides helpers, tools & components to assist with database access
- Provides a consisdent and full featured façade over the Python DBAPI
- Provides an optional object relational mapper(ORM)
- Foundation for many Python third party libraries & tools
- It doesn't hide the database, you need understand SQL

https://www.slideshare.net/hexdump42/relational-database-access-with-python?from_action=save



# Some Code Examples

- SQLAlchemy Example
- W4111-Databases/Examples/sqlalchemy_example.py

# Some Questions

- Why did I take this diversion?
  - I have alluded to various frameworks, but never covered.
    You should be aware of the option of using the frameworks.
  - *py2neo is an example of one of these frameworks.*
- There are pros and cons to using the frameworks:
  - Pros: significantly improved productivity for simple applications and mappings.
  - Cons: Anything complex can be virtually impossible if you use a framework.
  - The Wikipedia article is a good start on pros and cons.
- Why didn't I tell you about/let you use these frameworks?
  - You would not have learned the fundamental concepts in databases and models.
  - Frameworks are not helpful for many application scenarios.

*Back to Graphs and HW4*
*(Switch to Notebook)*

# *Redis*

# Redis

## Redis Key-Value Database: Practical Introduction


**SoftUni Team**

**Technical Trainers**

**Software University**

http://softuni.bg

# Table of Contents



SOFTWARE UNIVERSITY FOUNDATION

# Redis

## Ultra-Fast Data Structure Server

# What is Redis?



- Redis is:
  - Ultra-fast in-memory key-value data store
  - Powerful data structure server
  - Open-source software: http://redis.io
- Redis stores data structures:
  - Strings, lists, hashes, sets, sorted sets
  - Publish / subscribe messaging

# Redis: Features



- Redis is really fast
  - Non-blocking I/O, single threaded
  - 100,000+ read / writes per second


- Redis is not a database
  - It complements your existing data storage layer
  - E.g. StackOverflow uses Redis for data caching
- For small labs Redis may replace entirely the database
  - Trade performance for durability → data is persisted immediately

# Installing Redis


- To install Redis on Linux / Mac OS X:
  - Download from http://redis.io/download
  - Install it and run it
- To install Redis on Windows:
  - Download https://github.com/MSOpenTech/redis
    - Compile the solution in the **`msvs`** folder and build it with VS
  - The easier way: use the Windows package manager Chocolatey
    - http://chocolatey.org

```
choco install redis
```

# Running Redis


- On Linux / Mac OS X start / stop the Redis service

```
sudo service redis_6379 start
```

```
sudo service redis_6379 stop
```

  - The default Redis port is 6379 (the service name is **redis_6379**)

- On Windows start / stop the "**redis**" service

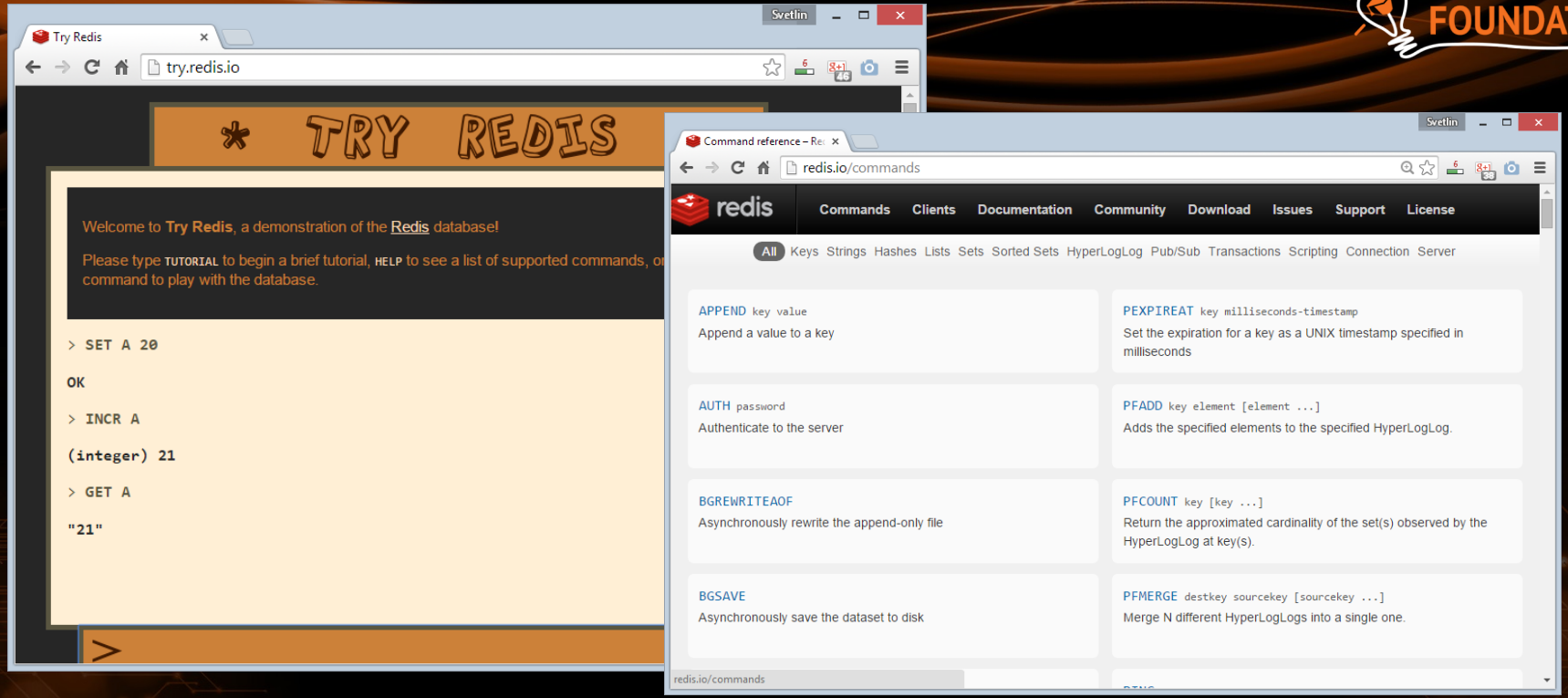
```
sc start redis
```

```
sc stop redis
```

- Use the console-based (CLI) client or just connect with Telnet:

```
redis-cli
```

```
telnet localhost 6379
```




# Redis: Basic Commands

# Redis: Data Model

- Redis keeps key-value pairs
  - Every item is stored as **`key`** + **`value`**
- Keys are unique identifiers
- Values can be different data structures:
  - Strings (numbers are stored as strings)
  - Lists (of strings)
  - Hash tables: string → string
  - Sets / sorted sets (of strings)


| key | value |
|---|---|
| firstName | Bugs |
| lastName | Bunny |
| location | Earth |

# Redis: Commands

- Redis works as interpreter of commands:

```
SET name "Nakov"
OK
GET name
"Nakov"
DEL name
(integer) 1
GET name
(nil)
```

- Play with the command-line client **`redis-cli`**
- Or play with Redis online at http://try.redis.io

# String Commands


- `SET [key] [value]`
  - Assigns a string value in a key
- `GET [key]` / `MGET [keys]`
  - Returns the value by key / keys
- `INCR [key]` / `DECR [key]`
  - Increments / decrements a key
- `STRLEN [key]`
  - Returns the length of a string

```
SET name
"hi"
OK

GET name
"hi"

SET name abc
OK

GET "name"
"abc"

GET Name
(nil)
```

```
SET a 1234
OK

INCR a
(integer) 1235

GET a
"12345"

MGET a name
1) "1235"
2) "asdda"

STRLEN a
(integer) 4
```

# Working with Keys


- `EXISTS [key]`
  - Checks whether a key exists
- `TYPE [key]`
  - Returns the type of a key
- `DEL [key]`
  - Deletes a key
- `EXPIRE [key] [t]`
  - Deletes a key after `t` seconds

```
SET count 5
OK
TYPE count
string
EXISTS count
(integer) 1
DEL count
(integer) 1
EXISTS count
(integer) 0
```

```
SET a 1234
OK
GET a
"1234"
EXPIRE a 5
(integer) 1
EXISTS a
(integer) 1
EXISTS a
(integer) 0
```

# Working with Hashes (Hash Tables)

- `HSET [key] [field] [value]`
  - Assigns a value for given field
- `HKEYS [key]`
  - Returns the fields (keys) is in a hash
- `HGET [key] [field]`
  - Returns a value by fields from a hash
- `HDEL [key] [field]`
  - Deletes a fields from a hash

```
HSET user name "peter"
(integer) 1
HSET user age 23
(integer) 1
HKEYS user
1) "name"
2) "age"
HGET user age
"23"
HDEL user age
(integer) 1
```

# Working with Lists


- `RPUSH` / `LPUSH [list] [value]`
  - Appends / prepend an value to a list
- `LINDEX [list] [index]`
  - Returns a value given index in a list
- `LLEN [list]`
  - Returns the length of a list
- `LRANGE [list] [start] [count]`
  - Returns a sub-list (range of values)

```
RPUSH names "peter"
(integer) 1
LPUSH names Nakov
(integer) 1
LINDEX names 0
"Nakov"
LLEN names
(integer) 2
LRANGE names 0 100
1) "Nakov"
2) "peter"
```

# Working with Sets

- `SADD [set] [value]`
  - Appends a value to a set
- `SMEMBERS [set]`
  - Returns the values from a set
- `SREM [set] [value]`
  - Deletes a value form a set
- `SCARD [set]`
  - Returns the stack size (items count)

```
SADD users "peter"
(integer) 1
SADD users "peter"
(integer) 0
SADD users maria
(integer) 1
SMEMBERS users
1) "peter"
2) "maria"
SREM users maria
(integer) 1
```

# Working with Sorted Sets


```
ZADD myzset 1 "one"
(integer) 1

ZADD myzset 1 "uno"
(integer) 1

ZADD myzset 2 "two" 3 "three"
(integer) 2

ZRANGE myzset 0 -1 WITHSCORES
1) "one"
2) "1"
…
```

- Learn more at http://redis.io/commands#sorted_set

# Publish / Subscribe Commands

SOFTWARE UNIVERSITY FOUNDATION

- First user subscribes to certain channel "**news**"

```
SUBSCRIBE news
1) "subscribe"
2) "news"
3) (integer) 1
```

- Another user sends messages to the same channel "**news**"

```
PUBLISH news "hello"
(integer) 1
```

- Learn more at http://redis.io/commands#pubsub

# Using Redis as a Database

SOFTWARE UNIVERSITY FOUNDATION

- Special naming can help using Redis as database

```
SADD users:names peter
HSET users:peter name "Peter Petrov"
HSET users:peter email "pp@gmail.com"

SADD users:names maria
HSET users:maria name "Maria Ivanova"
HSET users:maria email "maria@yahoo.com"

SMEMBERS users:names
HGETALL users:peter
```

Add a user "**peter**".

Use "**users:peter**" as key to hold user data

Add a user "**maria**".

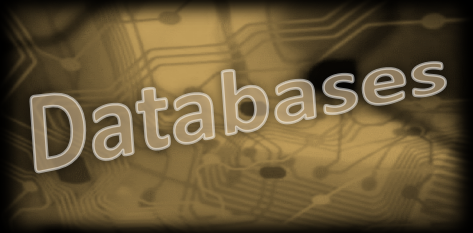
List all users

List all properties for user "**peter**"

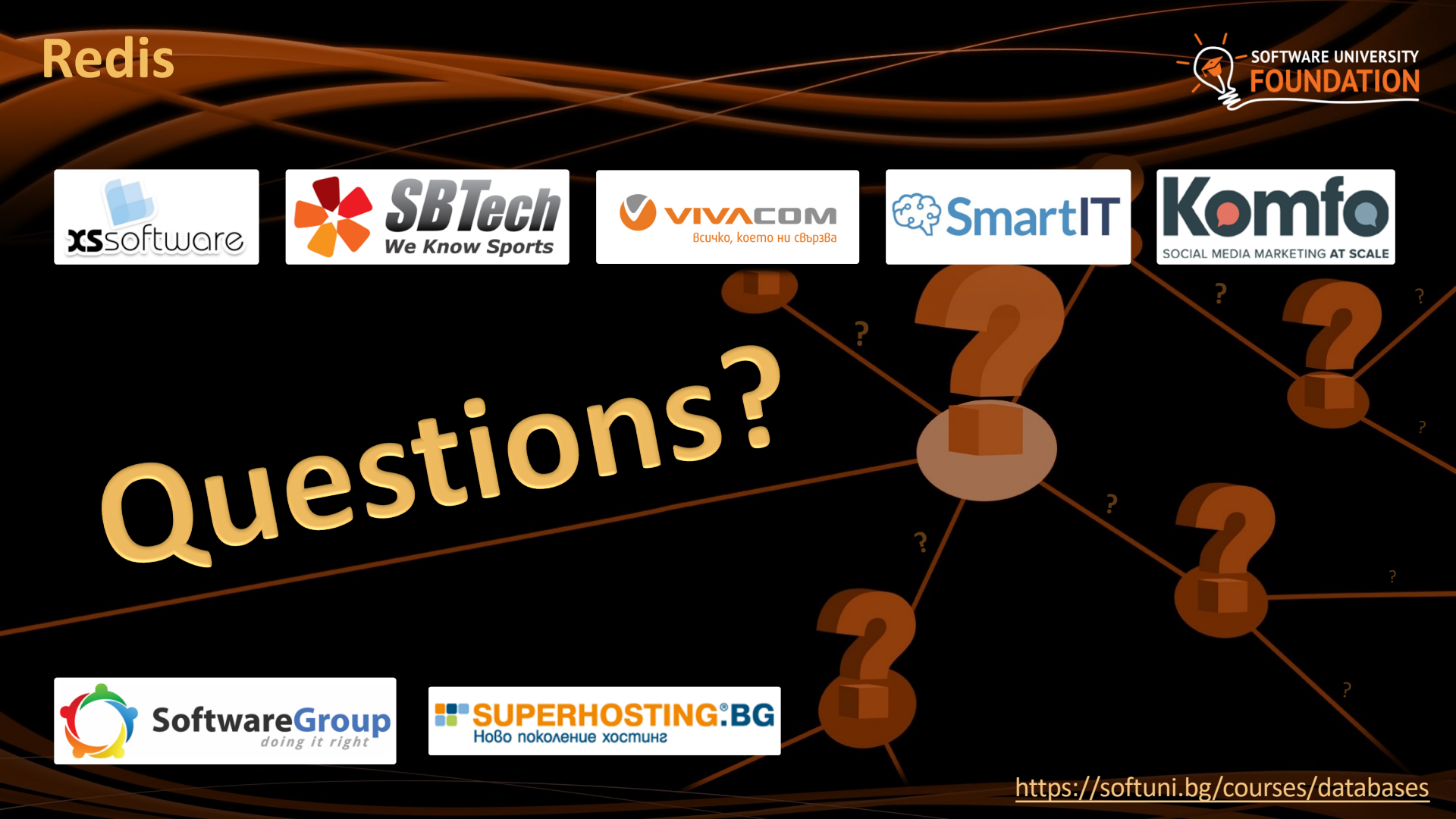
# Summary



1. Redis is ultra-fast in-memory data store
   - Not a database, used along with databases
2. Supports strings, numbers, lists, hashes, sets, sorted sets, publish / subscribe messaging
3. Used for caching / simple apps

# Redis



# License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license


- Attribution: this work may contain portions from
  - "Databases" course by Telerik Academy under CC-BY-NC-SA license

# Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg




*Redis and HW4*
*(Switch to Notebook)*