NoSQL Databases

Introduction

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]

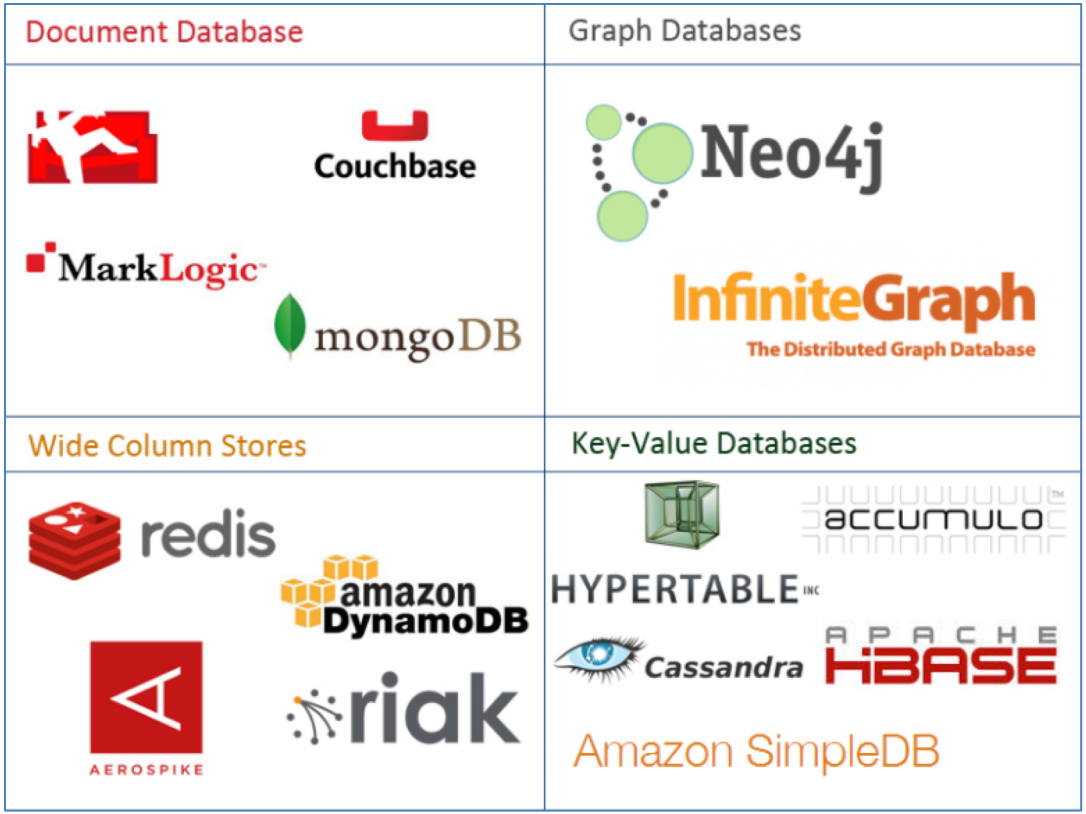
Motivations for this approach include: simplicity of design, simpler "horizontal" scaling to clusters of machines (which is a problem for relational databases),[2] and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different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

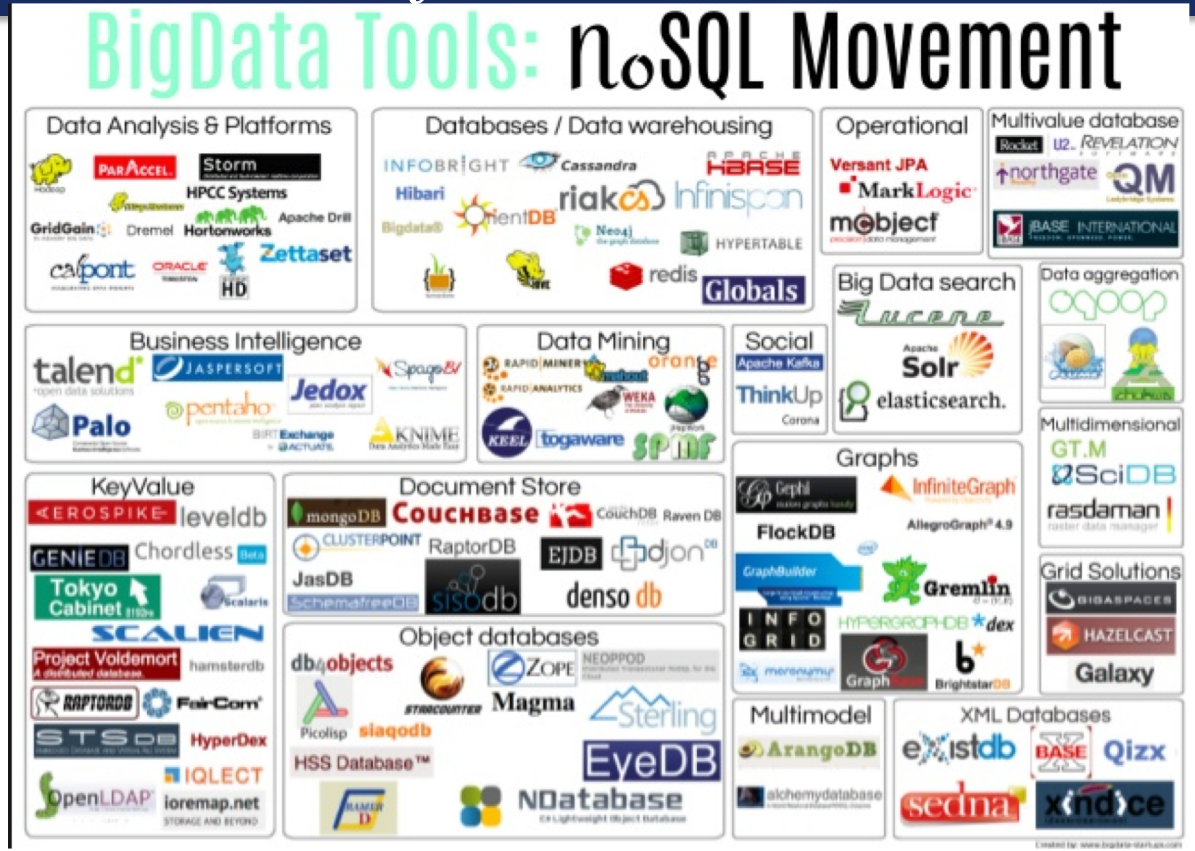
"Many NoSQL stores compromise consistency (in the sense of the CAP theorem) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins 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

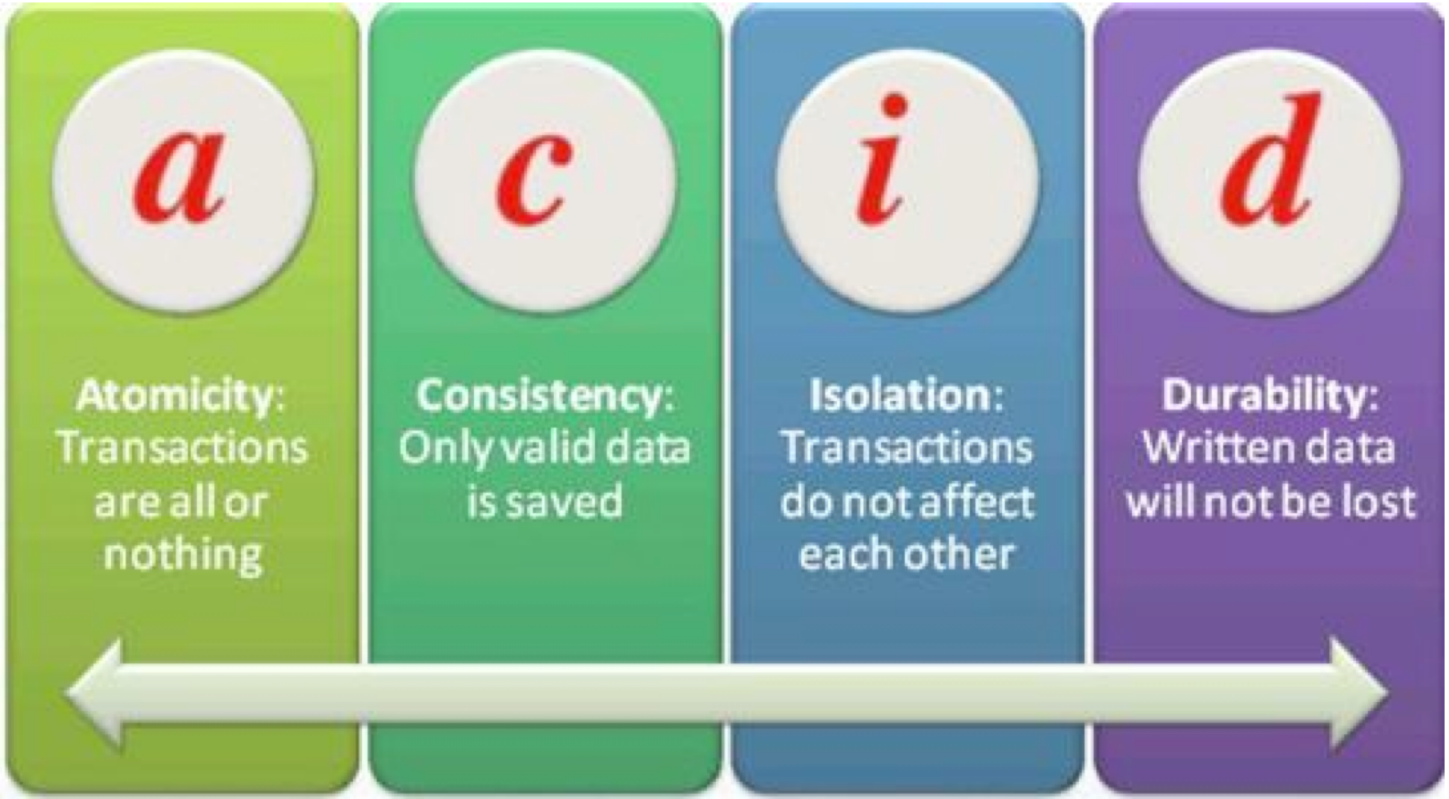


Another Taxonomy



CAP Theorem

ACID

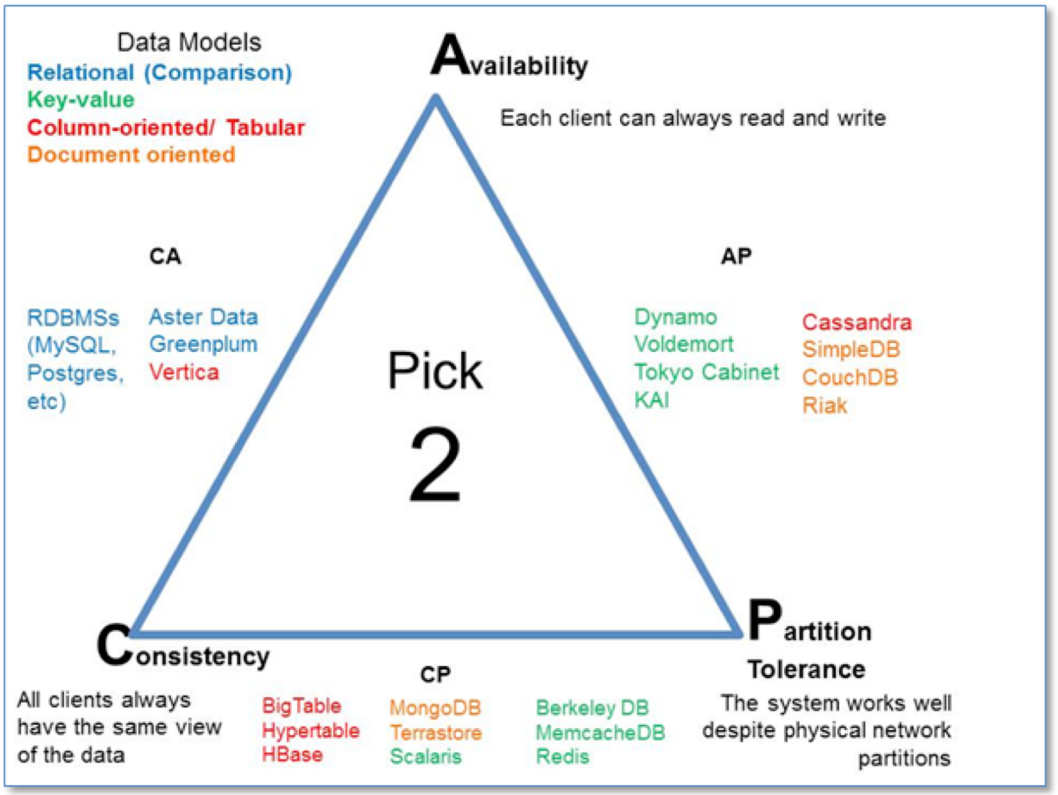


CAP Theorem

- 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

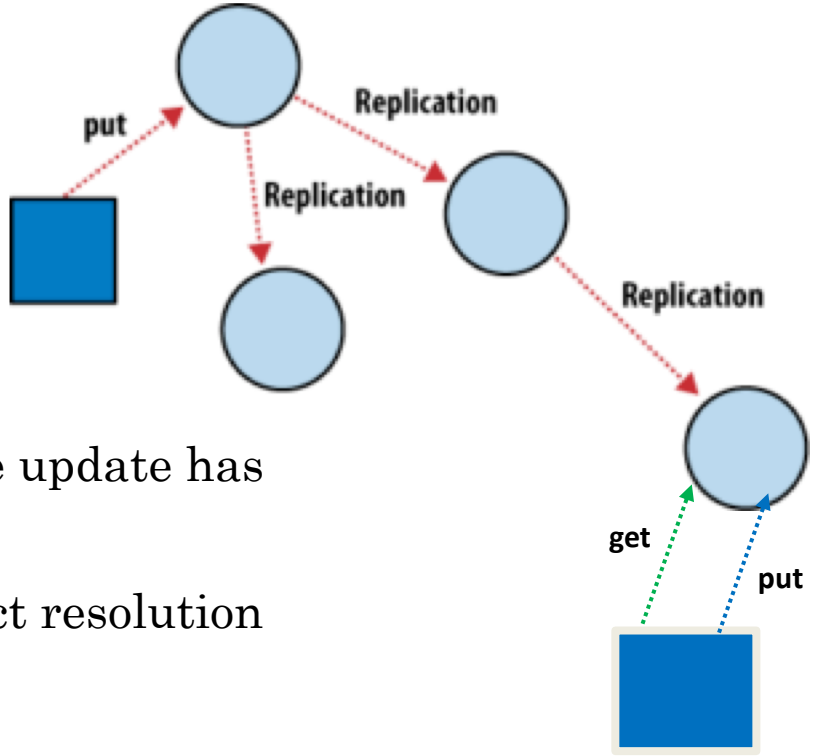


Consistency Models

- **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.

Eventual Consistency

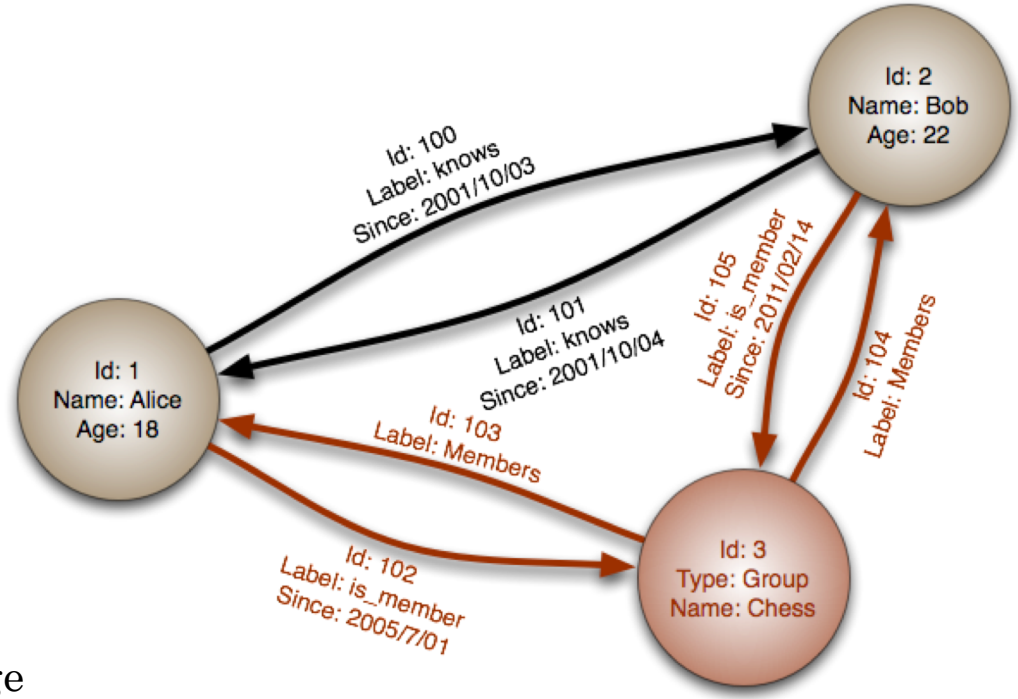
- Availability and scalability via
  - Multiple, replicated data stores.
  - Read goes to "any" replica.
  - PUT/POST/DELETE
    - Goes to any replica
    - Change propagate asynchronously
- GET may not see the latest value if the update has not propagated to the replica.
- There are several algorithms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



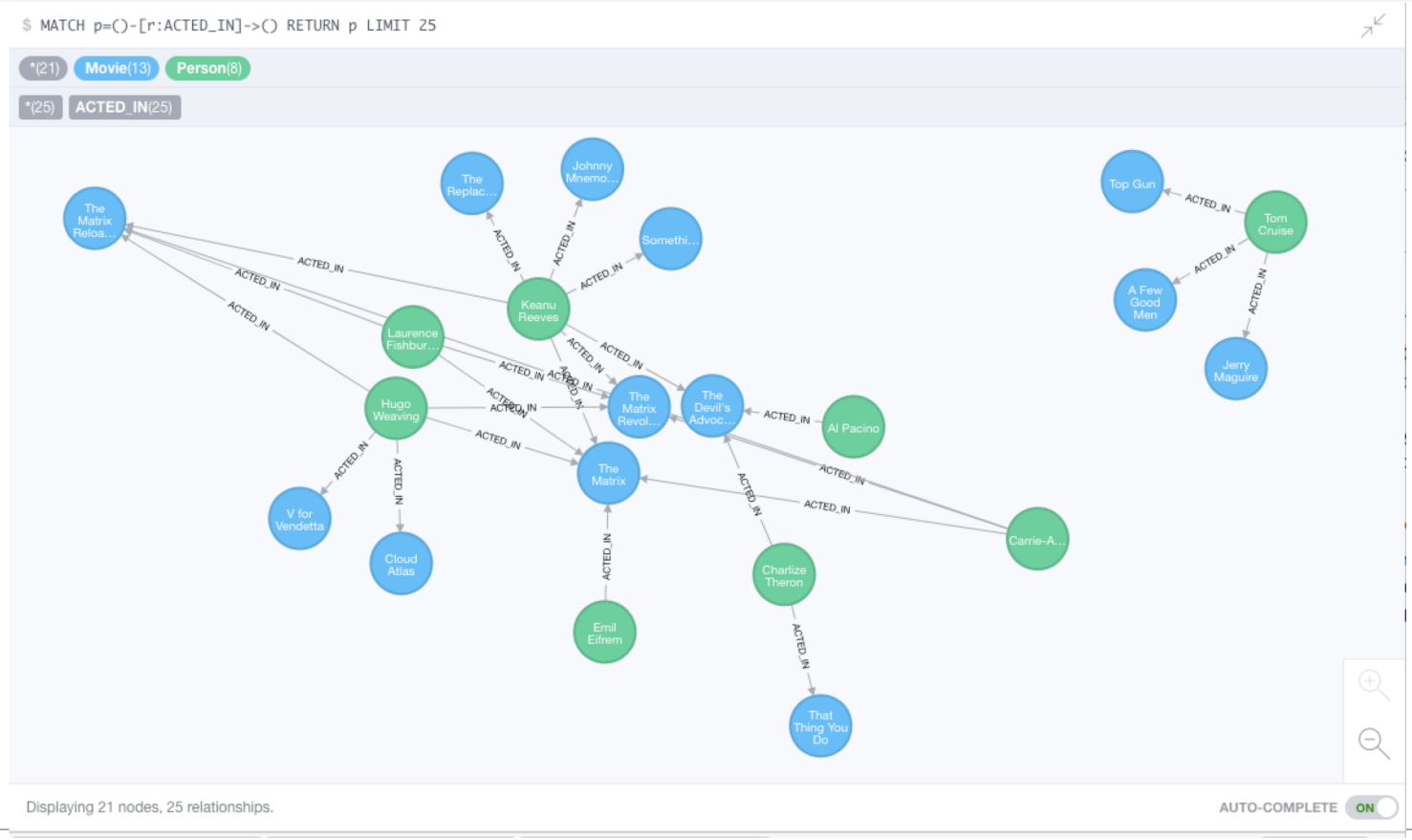
Graph Databases

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

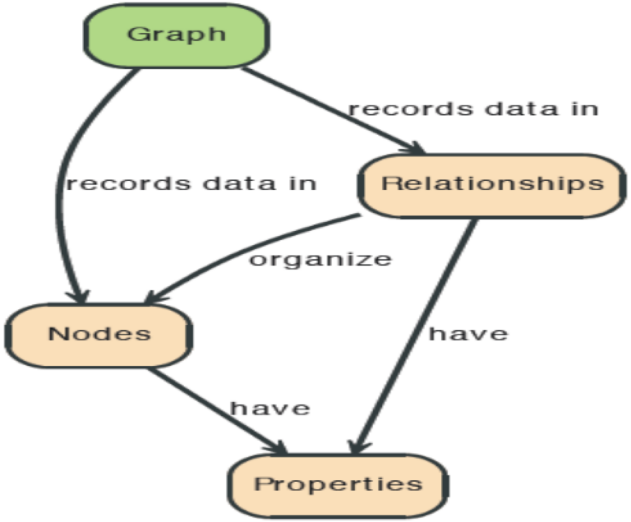
- Extremely fast for some queries and data models.
- Implement a language that vastly simplifies writing queries.

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!

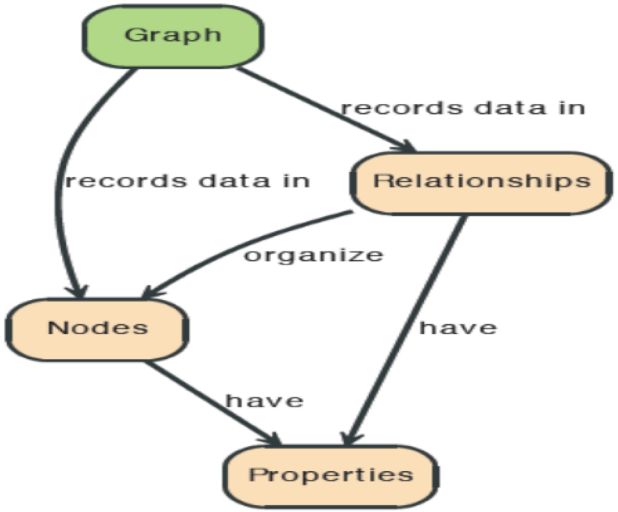
Graphs

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



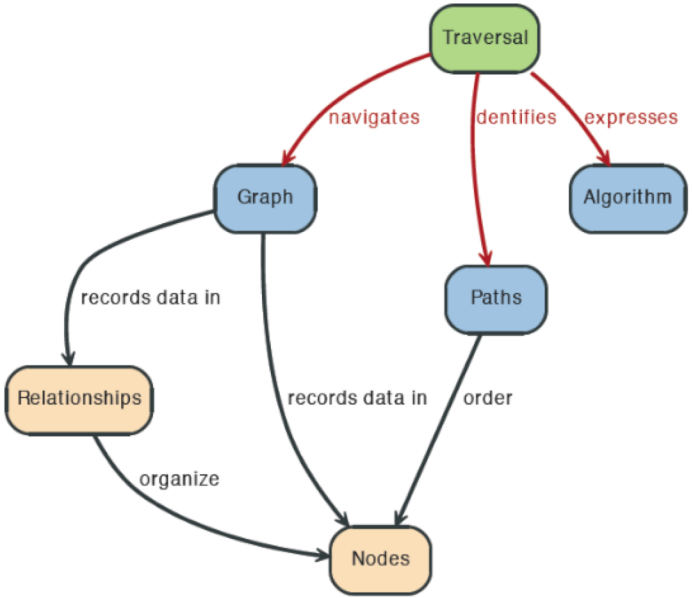
Graphs

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



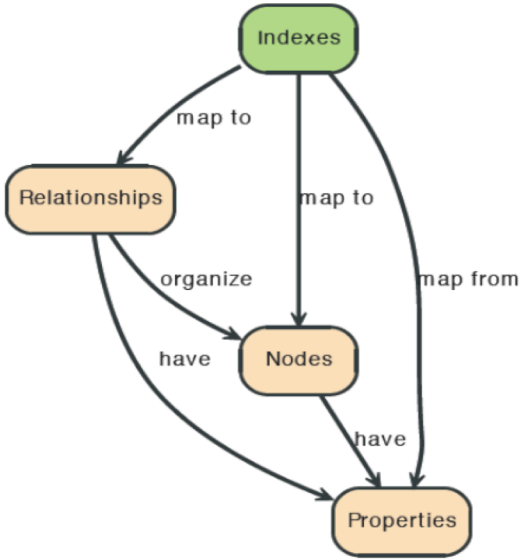
Query a graph with Traversal

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

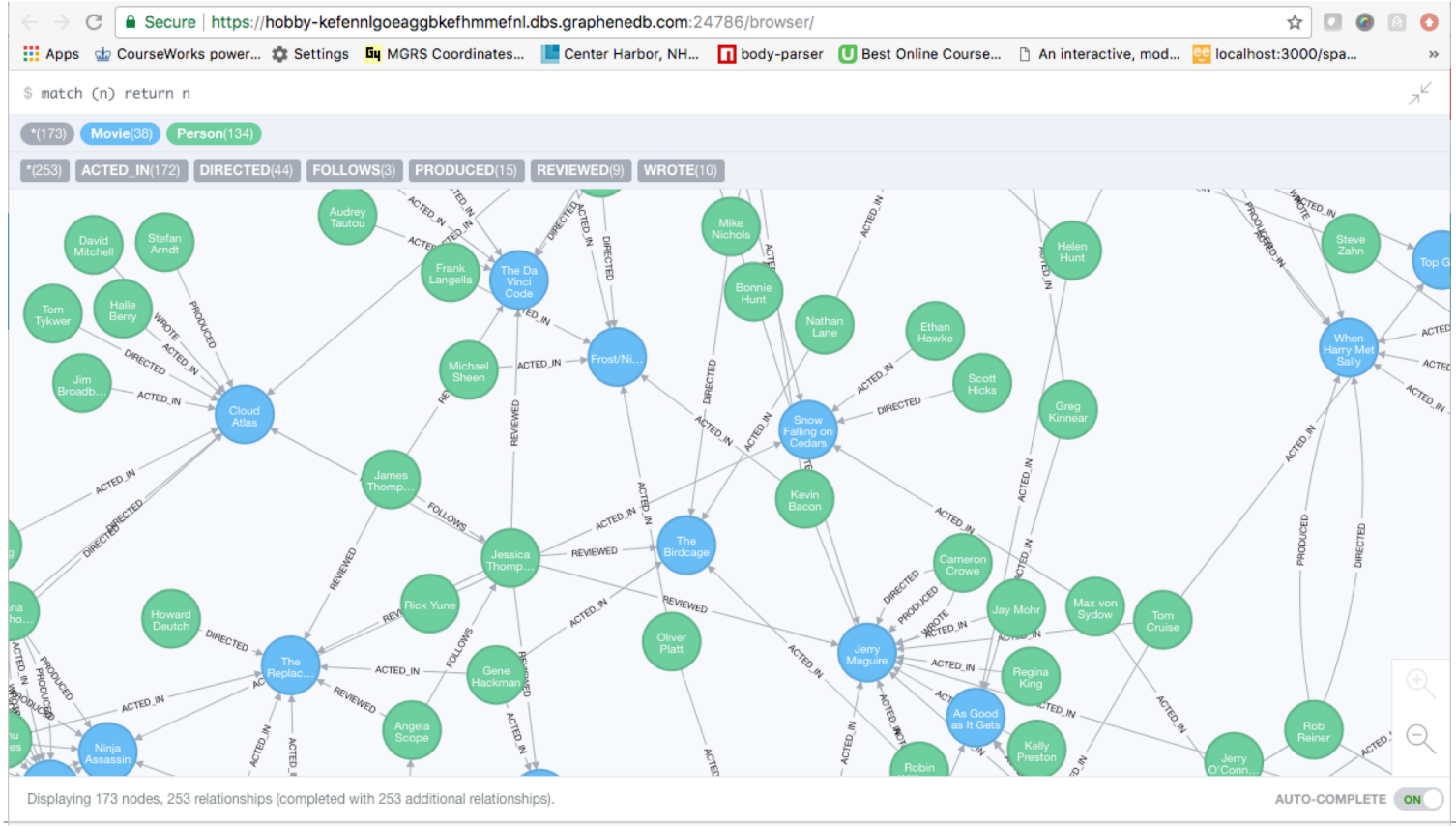


Indexes

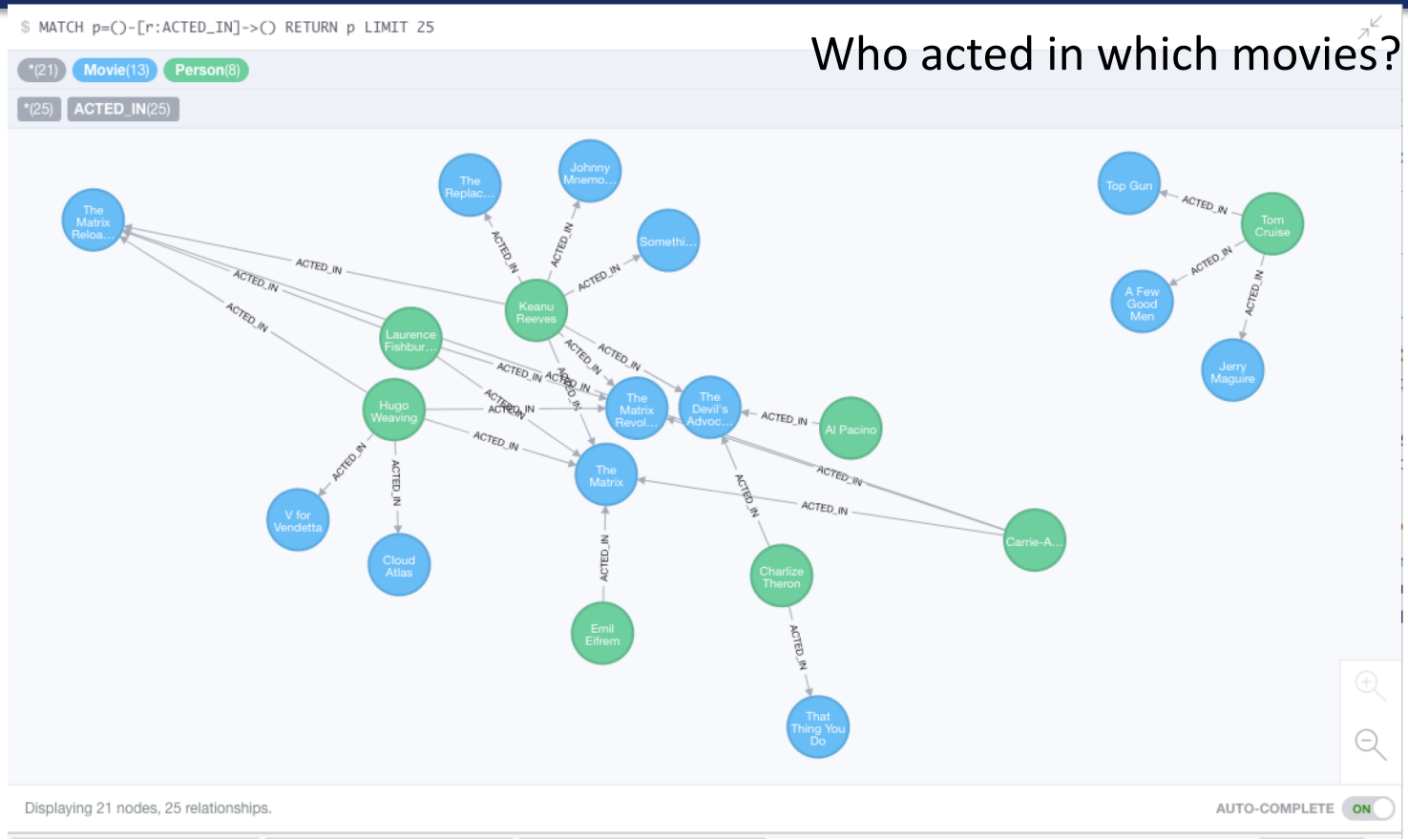
- "An Index —maps from→ Properties —to either→ Nodes or Relationships"



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...

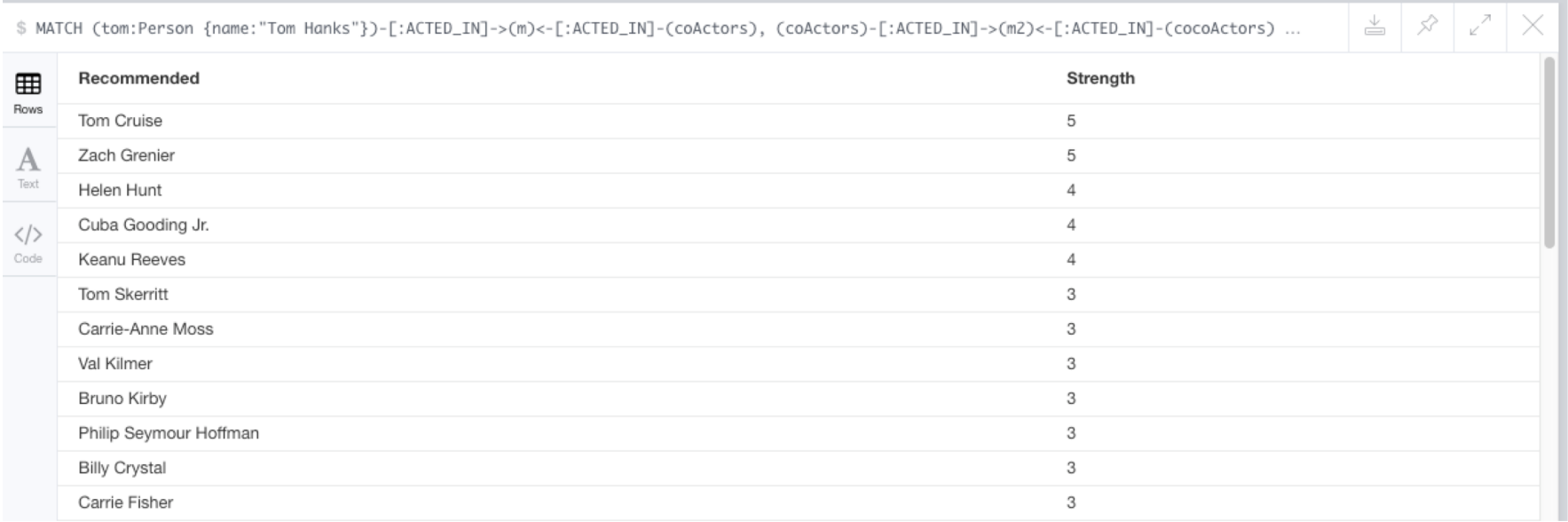
```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

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
```

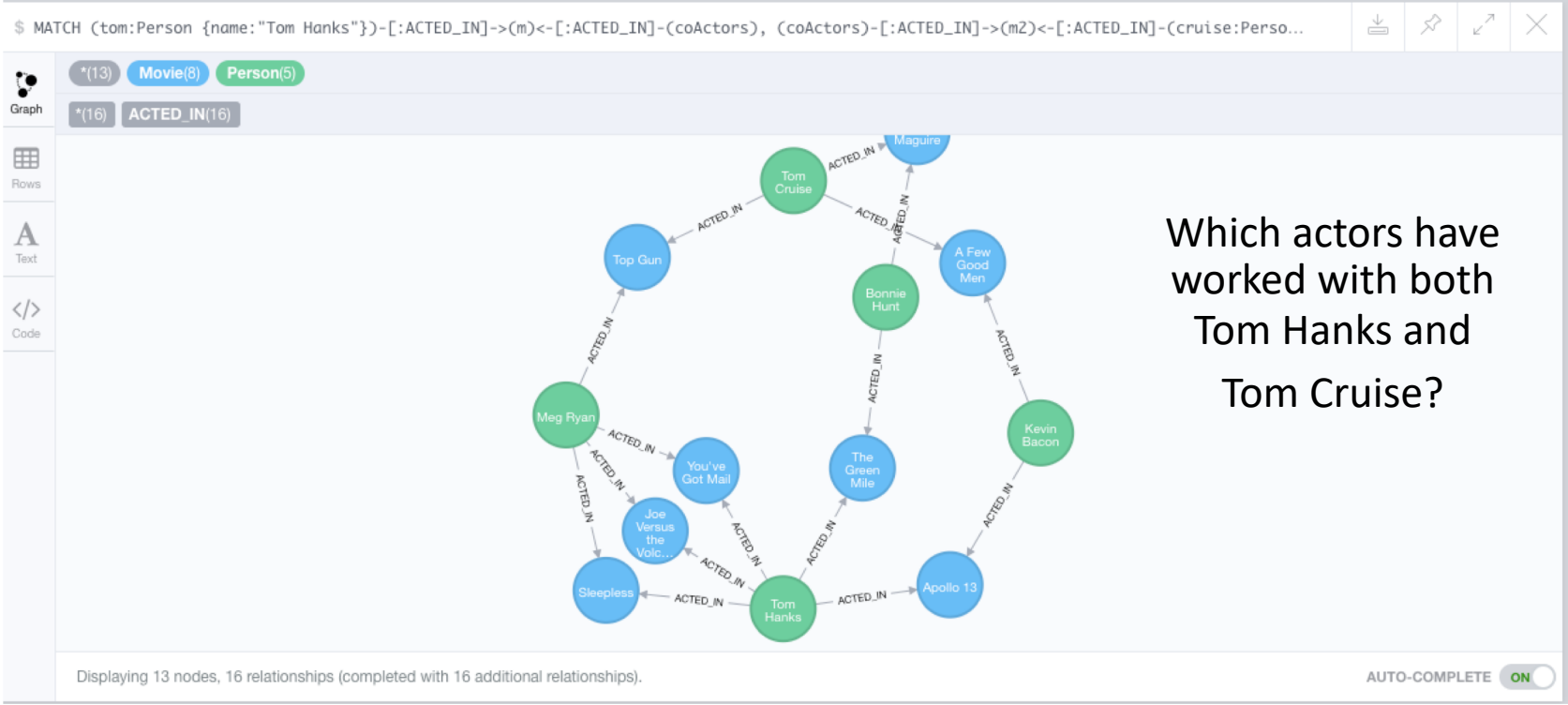
Recommend

```
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
```

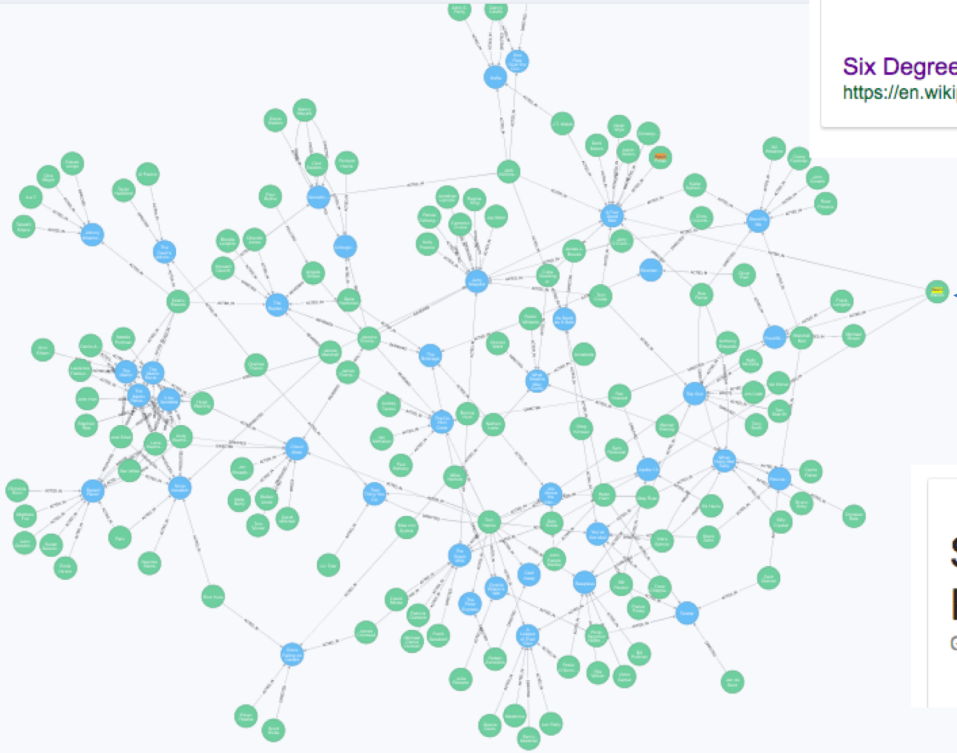


Which actors have worked with both Tom Hanks and Tom Cruise?

```
$ 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)



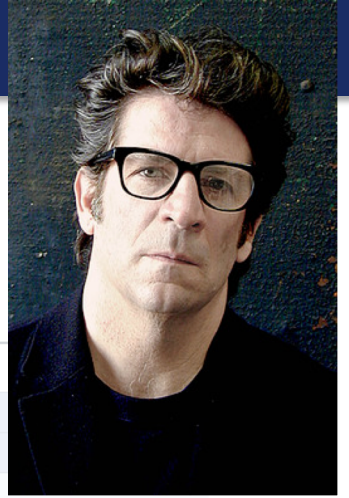
Six **Degrees of Kevin Bacon** is a parlour game based on the "six **degrees** of separation" concept, which posits that any two people on Earth are six or fewer acquaintance links apart. Movie buffs challenge each other to find the shortest path between an arbitrary actor and prolific actor **Kevin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

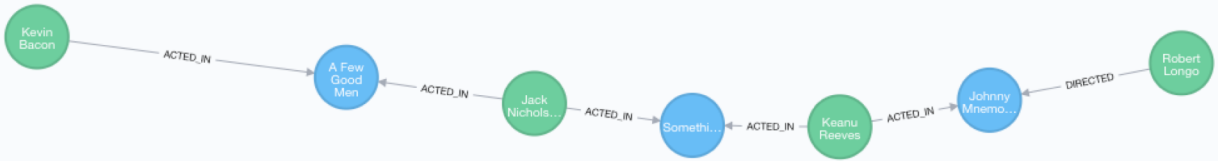


```
$ MATCH (kevin:Person { name: 'Kevin Bacon' }),(robert:Person { name: 'Robert Longo' }), p = shortestPath((kevin)-[*..15]-(robert)) RETURN p
```

Graph | Rows | Text | Code

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

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



Facebook Graph API

Facebook Graph API

Overview

https://developers.facebook.com/docs/graph-api/overview

The Graph API is the primary way to get data into and out of the Facebook platform. It's an HTTP-based API that apps can use to programmatically query data, post new stories, manage ads, upload photos, and perform a wide variety of other tasks.

The Basics

The Graph API is named after the idea of a "social graph" — a representation of the information on Facebook. It's composed of:

- **nodes** — basically individual objects, such as a User, a Photo, a Page, or a Comment
- **edges** — connections between a collection of objects and a single object, such as Photos on a Page or Comments on a Photo
- **fields** — data about an object, such as a User's birthday, or a Page's name

Typically you use nodes to get data about a specific object, use edges to get collections of objects on a single object, and use fields to get data about a single object or each object in a collection.

Facebook Graph API

Graph API Root Nodes

This is a full list of the Graph API root nodes. The main difference between a root node and a non-root node is that root nodes can be queried directly, while non-root nodes can be queried via root nodes or edges. If you want to learn how to use the Graph API, read our Using Graph API guide, and if you want to know which APIs can solve some frequent issues, try our Common Scenarios guide.

Node	Description
Achievement	Instance for an achievement for a user.
Achievement Type	Graph API Reference Achievement Type /achievement-type
Album	A photo album
App Link Host	An individual app link host object created by an app
App Request	An individual app request received by someone, sent by an app or another person
Application	A Facebook app
Application Context	Provides access to available social context edges for this app
Async Session	Represents an async job request to Graph API
Audience Insights Rule	Definition of a rule
CTCert Domain	A domain name that has been issued new certificates.
Canvas	A canvas document
Canvas Button	A button inside the canvas
Canvas Carousel	A carousel inside a canvas

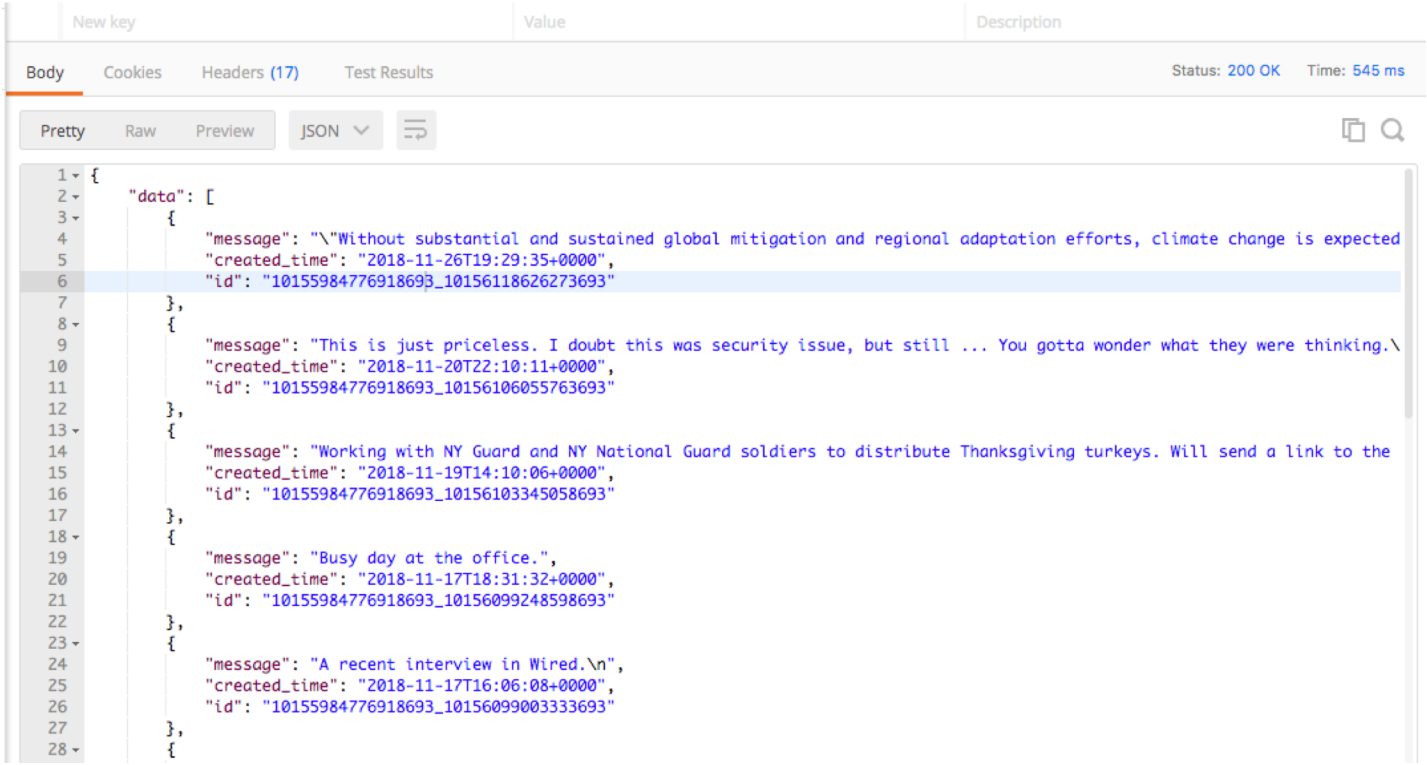
Full list at:
https://developers.facebook.com/docs/graph-api/reference

... ...

Life Event	Page milestone information
Link	A link shared on Facebook
Live Encoder	An EntLiveEncoder is for the live encoders that can be associated with video broadcasts. This is part of the reference live encoder API
Live Video	A live video
Mailing Address	A mailing address object
Message	An individual message in the Facebook messaging system.
Milestone	Graph API Reference Milestone /milestone
Native Offer	A native offer represents an Offer on Facebook. The /{offer_id} node returns a single native offer. Each native offer requires a view to be rendered to users.
Notification	Graph API Reference Notification /notification
Object Comments	This reference describes the /comments edge that is common to multiple Graph API nodes. The structure and operations are the same for each node.

Facebook Graph API – Examples

https://graph.facebook.com/v3.2/me/posts?access_token=xxxxxxx



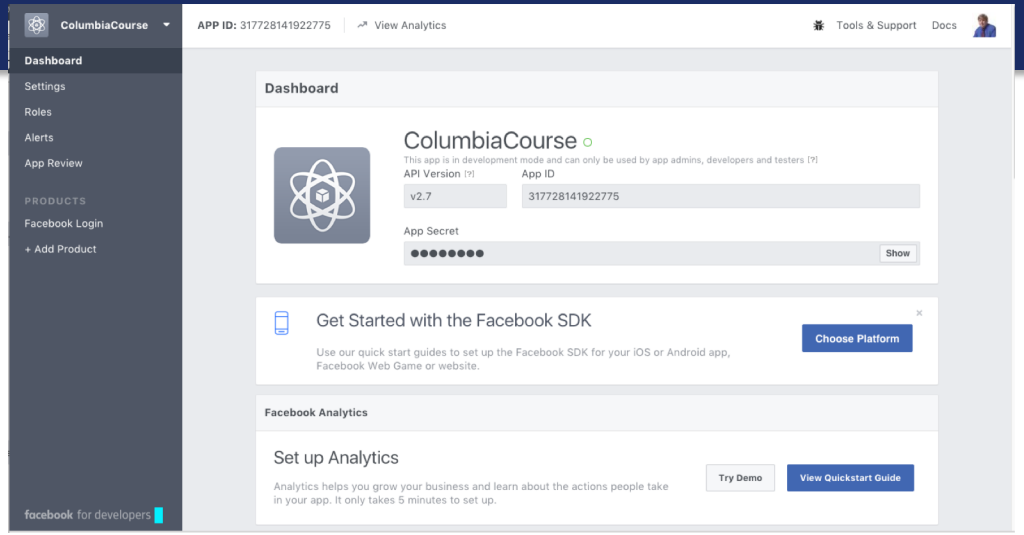

```
{
    "data": [
        {
            "message": "\"Without substantial and sustained global mitigation and regional adaptation efforts, climate change is expected
            "created_time": "2018-11-26T19:29:35+0000",
            "id": "10155984776918693_10156118626273693"
        },
        {
            "message": "This is just priceless. I doubt this was security issue, but still ... You gotta wonder what they were thinking.\
            "created_time": "2018-11-20T22:10:11+0000",
            "id": "10155984776918693_10156106055763693"
        },
        {
            "message": "Working with NY Guard and NY National Guard soldiers to distribute Thanksgiving turkeys. Will send a link to the
            "created_time": "2018-11-19T14:10:06+0000",
            "id": "10155984776918693_10156103345058693"
        },
        {
            "message": "Busy day at the office.",
            "created_time": "2018-11-17T18:31:32+0000",
            "id": "10155984776918693_10156099248598693"
        },
        {
            "message": "A recent interview in Wired.\n",
            "created_time": "2018-11-17T16:06:08+0000",
            "id": "10155984776918693_10156099003333693"
        },
        {
```

Facebook Graph API – Examples

https://graph.facebook.com/v3.2/me/likes?access_token=xxxxxxx

Notice use of paging links.

Facebook Applications



- Facebook Application
  - Define an application at developers.facebook.com/apps/
  - Users log into your application via Facebook.
  - This grants your application permission to access the user's Facebook content.
  - User must approve access on login and Facebook must "approve" the application for non-profile information.
- Once logged onto your application via Facebook, you can
  - Get basic information about people.
  - Enable people to share, etc. information from your application to Facebook.
  - Perform analytics on users and their networks, subject to granted permissions.

- Approaches to creating a “primary key.”
- UUID

- MATCH (n {uni: "al1"})-[r:FOLLOWS *0..4]->(q) return n,r,q
- match (n:Fan)-[r:FOLLOWS]->(p)-[r2:SUPPORTS]->(t) return n,r,p,r2,t
- match (n:Fan {uni: 'al1'})-[r:FOLLOWS]->(p)-[r2:SUPPORTS]->(t) return n,r,p,r2,t
- match (n:Fan {uni: 'al1'})-[r:FOLLOWS *0..2]->(p)-[r2:SUPPORTS]->(t) return n,r,p,r2,t
- create (p:Faculty {uni: 'dff9', name_last: "Ferguson", name_first: "Donald"}) return p

Graph Database Summary

Graph Databases

- Popularity and usage has exploded in the past few years.
  - Facebook Graph API (https://developers.facebook.com/docs/graph-api/)
  - LinkedIn (https://developer.linkedin.com/docs/rest-api)
  - Instagram (https://developers.facebook.com/docs/instagram-api)
  - Google Knowledge Graph (https://developers.google.com/knowledge-graph/)

- Two major motivations:
  1. Many common types of data are inherently graph oriented, and representing in RDB or some other mechanism results in icky code.
  2. Performs of common graph operations is extremely slow in RDB.

Redis

Redis

Redis Key-Value Database: Practical Introduction

SoftUni Team

Technical Trainers

Software University

http://softuni.bg

SOFTWARE UNIVERSITY FOUNDATION

Redis

Ultra-Fast Data Structure Server

What is Redis?

SOFTWARE UNIVERSITY FOUNDATION

- 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

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

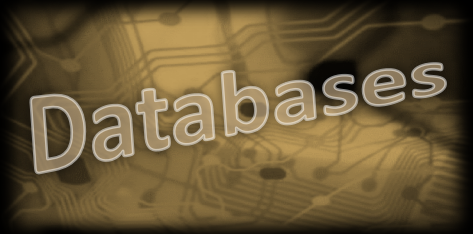
Add a user "**maria**".

List all users

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

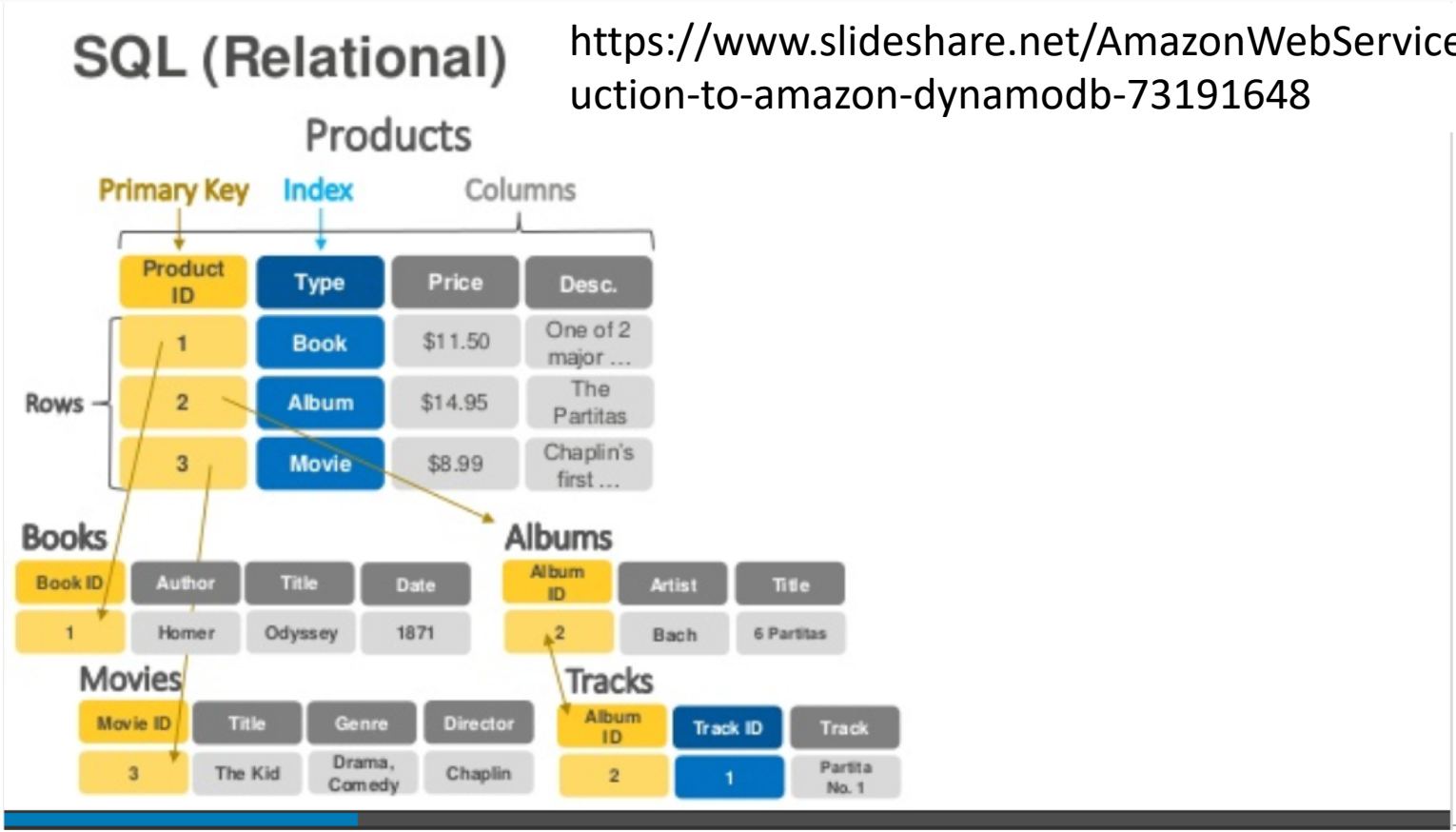
Summary

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



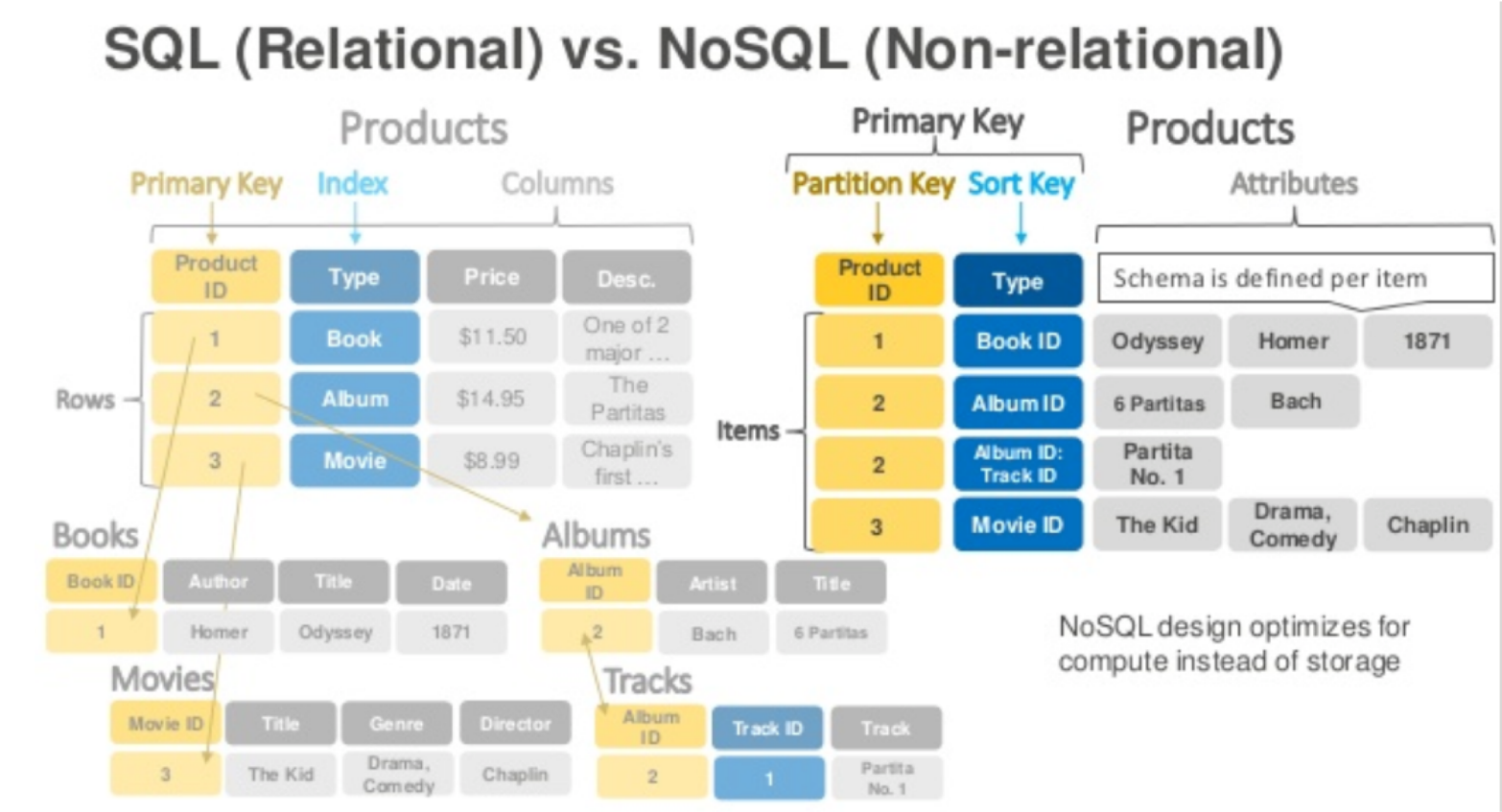
DynamoDB

DynamoDB Data Model



https://www.slideshare.net/AmazonWebServices/introduction-to-amazon-dynamodb-73191648

DynamoDB Data Model



DynamoDB Benefits


Fully managed


Fast, consistent performance

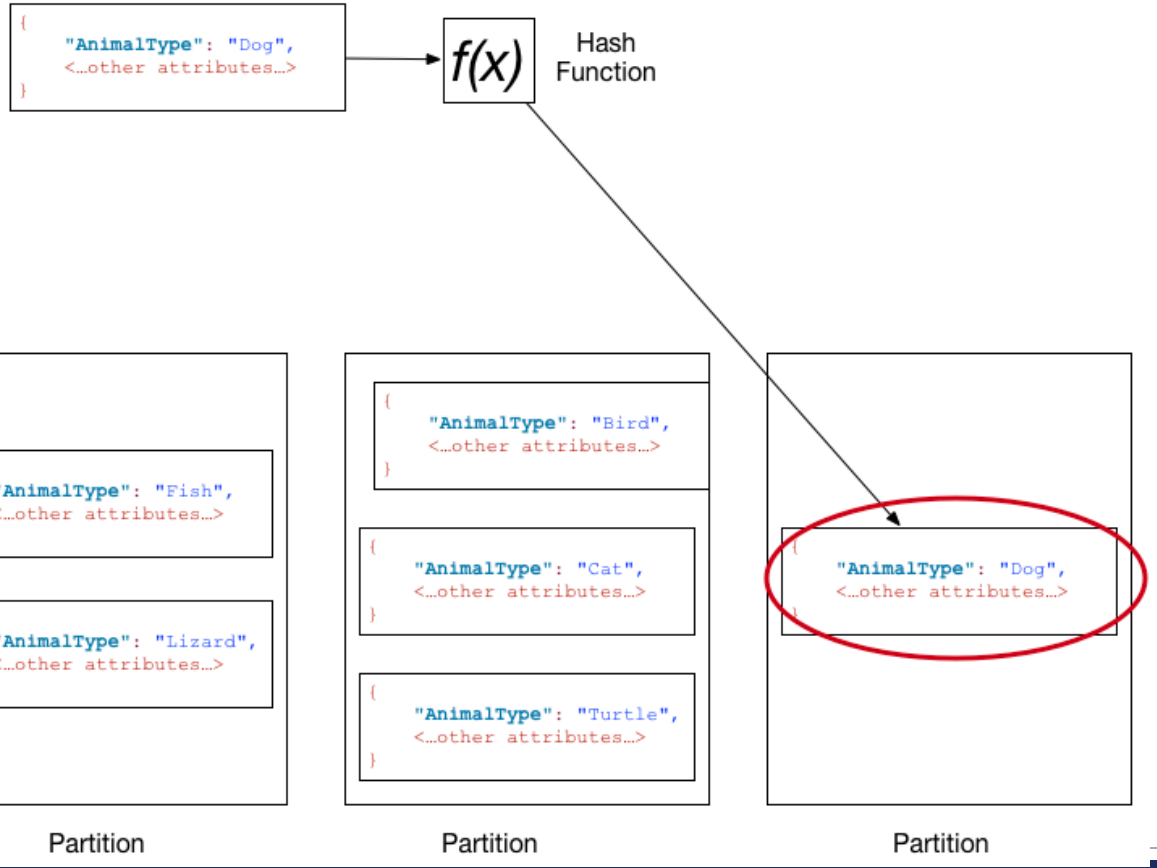

Highly scalable


Flexible

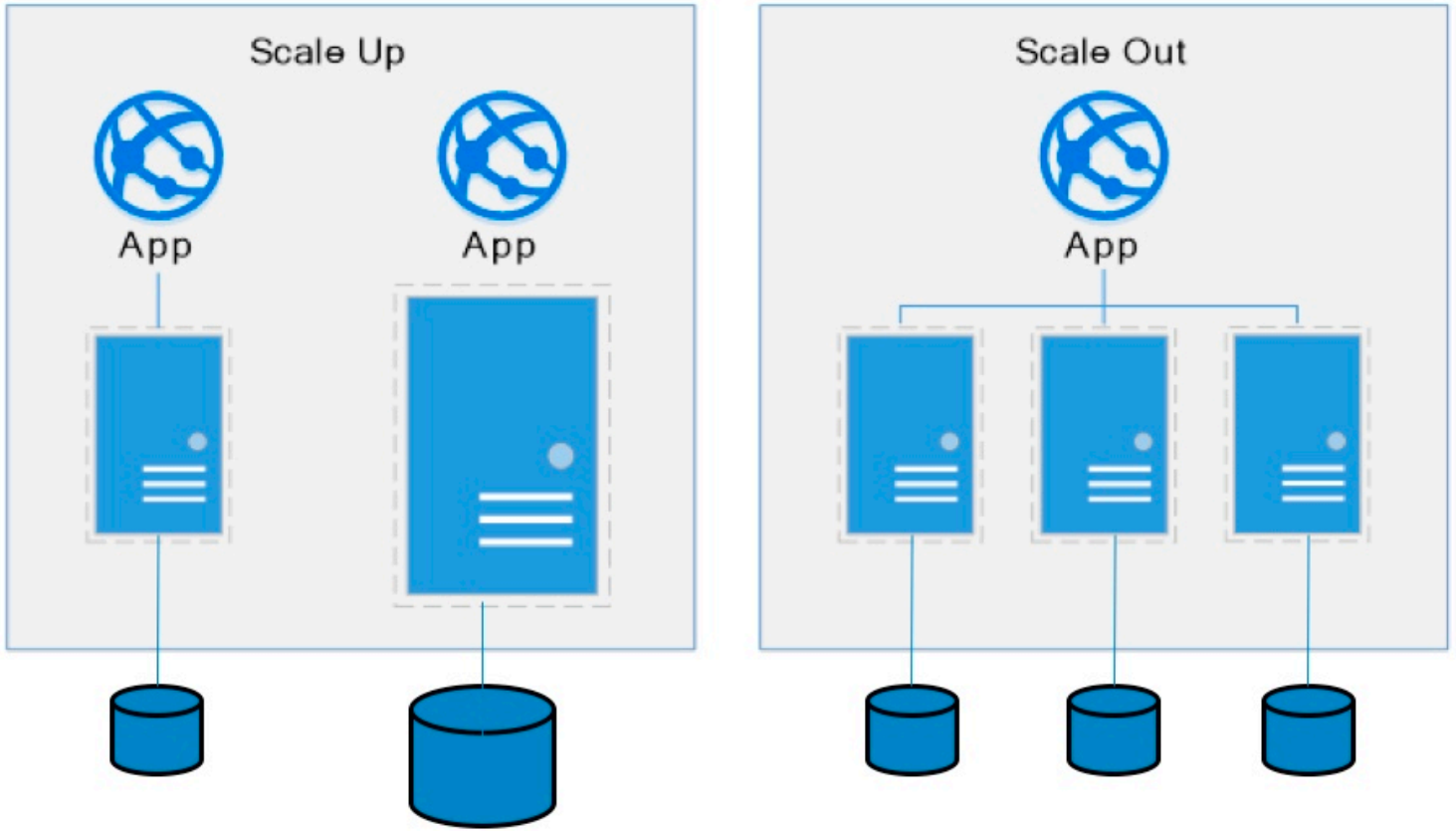

Event-driven programming


Fine-grained access control

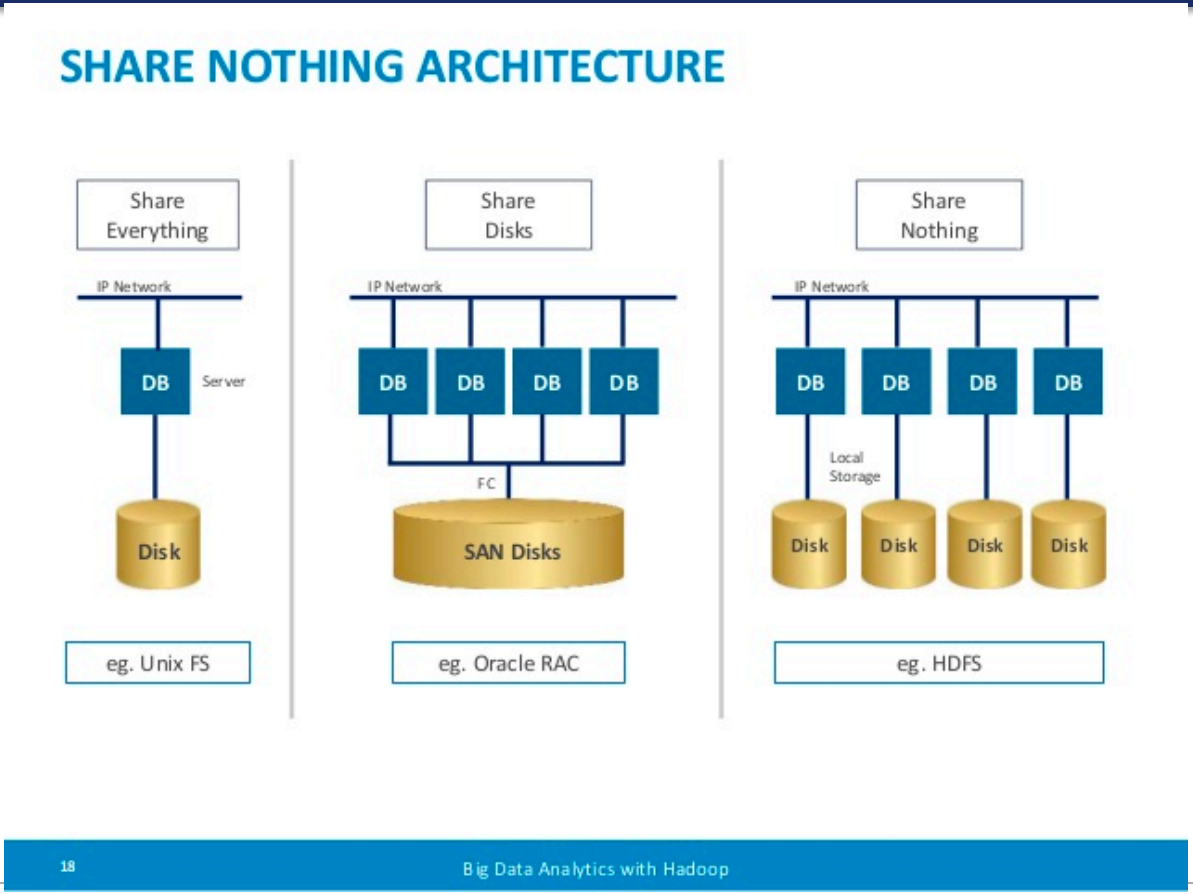
DynamoDB Hashing



Approaches to Scalability



Share Nothing Architecture



Sample Code

```
item1 ={
        'product_id': 'od1',
        'kind': 'book',
        'title': 'Database Manager Systems',
        'isbn': "978-0072465631",
        'authors': [
            {
                'last_name': 'Gherke',
                'first_name': 'Johannes'
            },
            {
                'last_name': 'Ramakrishnan',
                'first_name': 'Raghu'
            }
        ],
        'edition': '3rd',
        'categories': ['books', 'software', 'd
    }
```

```
item2 = {
        'product_id': 'molm',
        'kind': 'Movie',
        'formats': ['online', 'dvd', 'vhs'],
        'title': 'Man of La Mancha',
        'artists': [
            {
                "directors": [
                    {
                        'last_name': 'Hiller',
                        'first_name': 'Arther'
                    }
                ],
                "actors": [
                    {
                        'last_name': "O'Toole",
                        'first_name': 'Peter'
                    },
                    {
                        'last_name': "Loren",
                        'first_name': 'Sophia'
                    }
                ]
            }
        ],
        'genres': ['musical', 'broadway', 'culture'],
        'running time': 128,
        'languages': ['english']
}
```

```
import boto3
import json
import dynamodb_json
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('products')

table.put_item(
    Item=item1
)
table.put_item(
    Item=item2
)
```

Sample Code

```python
import boto3
import json

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('products')

response = table.get_item(
        Key={
            'product_id': 'molm'
        }
    )
print(response)
response = response['Item']

response['running time'] = \
    str(response['running time'])

print('Response with Decimal = ',
      json.dumps(response, indent=2))
```

```
{'Item': {'running time': Decimal('128'), 'artists': [{'actors': [{'last_
Response without Decimal =  {
  "running time": "128",
  "artists": [
    {
      "actors": [
        {
          "last_name": "O'Toole",
          "first_name": "Peter"
        },
        {
          "last_name": "Loren",
          "first_name": "Sophia"
        }
      ],
      "directors": [
        {
          "last_name": "Hiller",
          "first_name": "Arther"
        }
      ]
    }
  ],
  "languages": [
    "english"
  ],
  "kind": "Movie",
  "formats": [
    "online",
    "dvd",
    "vhs"
  ],
  "genres": [
    "musical",
    "broadway",
    "culture"
  ],
  "product_id": "molm",
  "title": "Man of La Mancha"
}
```

DynamoDB Summary

- Achieves much greater scalability and performance than RDBMs
- Does not support some RDBMs capabilities:
  - Referential integrity.
  - JOIN
  - Queries limited to key fields.
  - Non-key field queries are always scans.
- Data model better fit for *semi-structured* data models and good fit for JSON
  - Maps
  - Lists

batch_get_item()	get_waiter()
batch_write_item()	list_backups()
can_paginate()	list_global_tables()
create_backup()	list_tables()
create_global_table()	list_tags_of_resource()
create_table()	put_item()
delete_backup()	query()
delete_item()	restore_table_from_backup()
delete_table()	restore_table_to_point_in_time()
describe_backup()	scan()
describe_continuous_backups()	tag_resource()
describe_endpoints()	untag_resource()
describe_global_table()	update_continuous_backups()
describe_global_table_settings()	update_global_table()
describe_limits()	update_global_table_settings()
describe_table()	update_item()
describe_time_to_live()	update_table()
generate_presigned_url()	update_time_to_live()
get_item()	get_paginator()

DynamoDB Python API