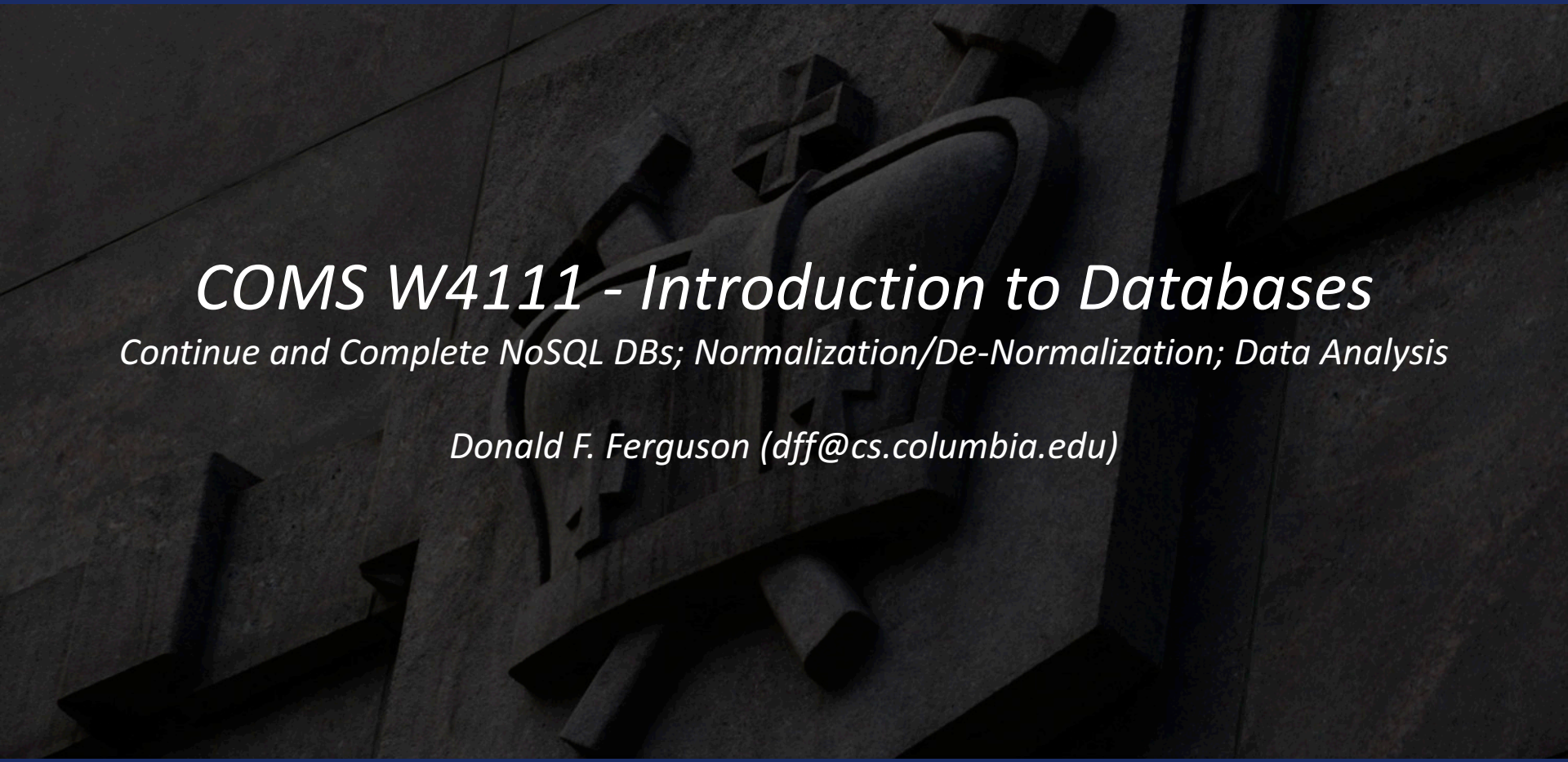


COMS W4111 - Introduction to Databases

Continue and Complete NoSQL DBs; Normalization/De-Normalization; Data Analysis

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

Contents

Contents

- A Diversion: ORM, persistence/mapping frameworks.
- Graph Database Completion – Some simple examples:
  - Neo4J and HW4
  - Facebook.
- Redis:
  - Overview.
  - API
- DynamoDB: really short overview
- Putting the pieces together: SQL and NoSQL example.
- Relational Normalization/De-normalization.
- Setup for HW5

A Diversion:
ORM, persistence /mapping frameworks

Object “Relational” Mapping

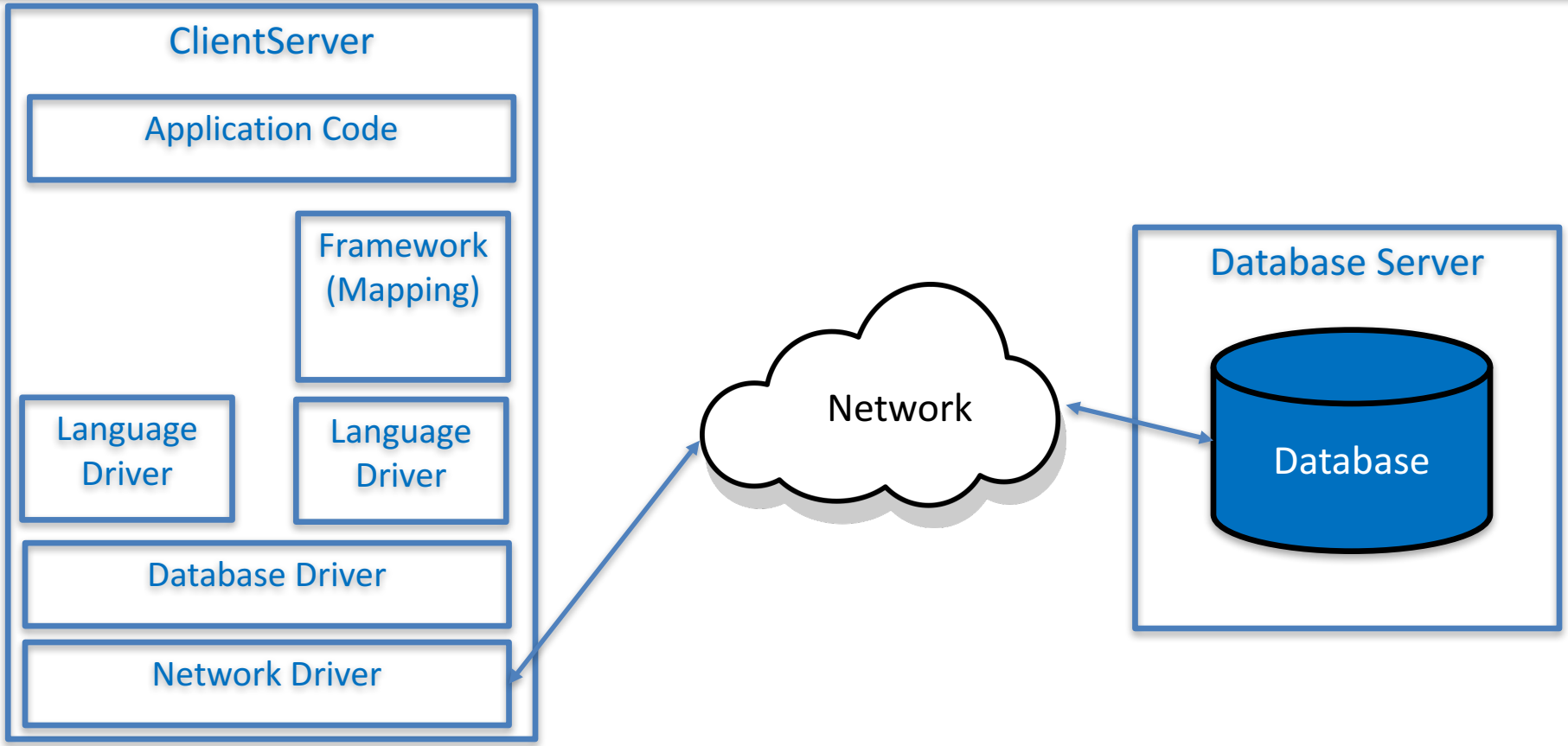
Object-relational mapping (**ORM**, **O/RM**, and **O/R mapping tool**) in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the 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.

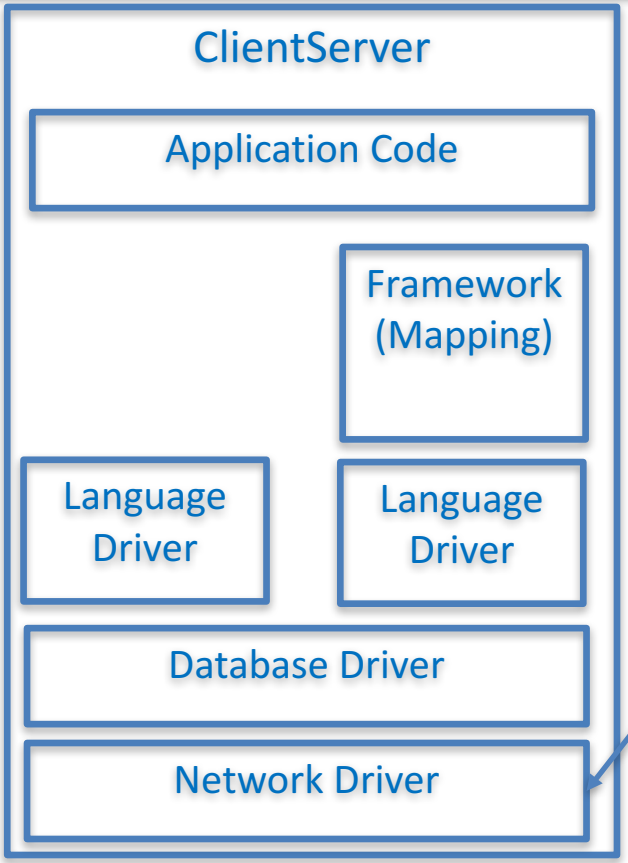
However, many popular database products such as SQL database management systems (DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (and 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

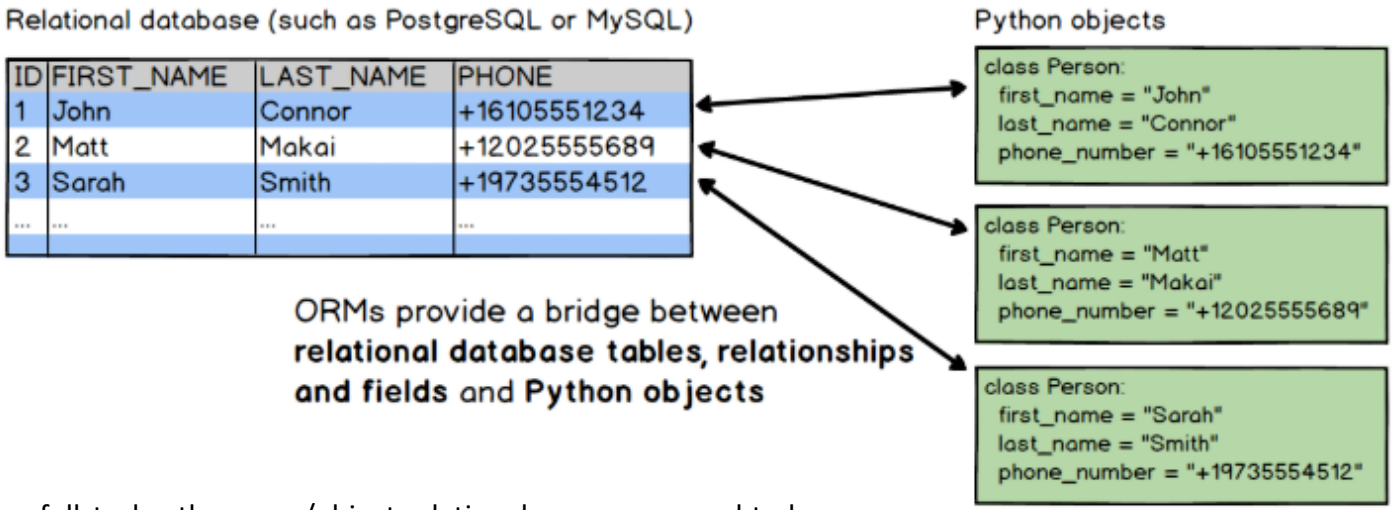


- The application code is what you write to implement your homework (project)
- Network driver is part of the operating system and sends/receives message using HTTP, TCP/IP, etc.
- Database driver uses network driver to send commands and receive response, e.g.
  - SELECT * FROM …
  - Match (n:Node) return …
- 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

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

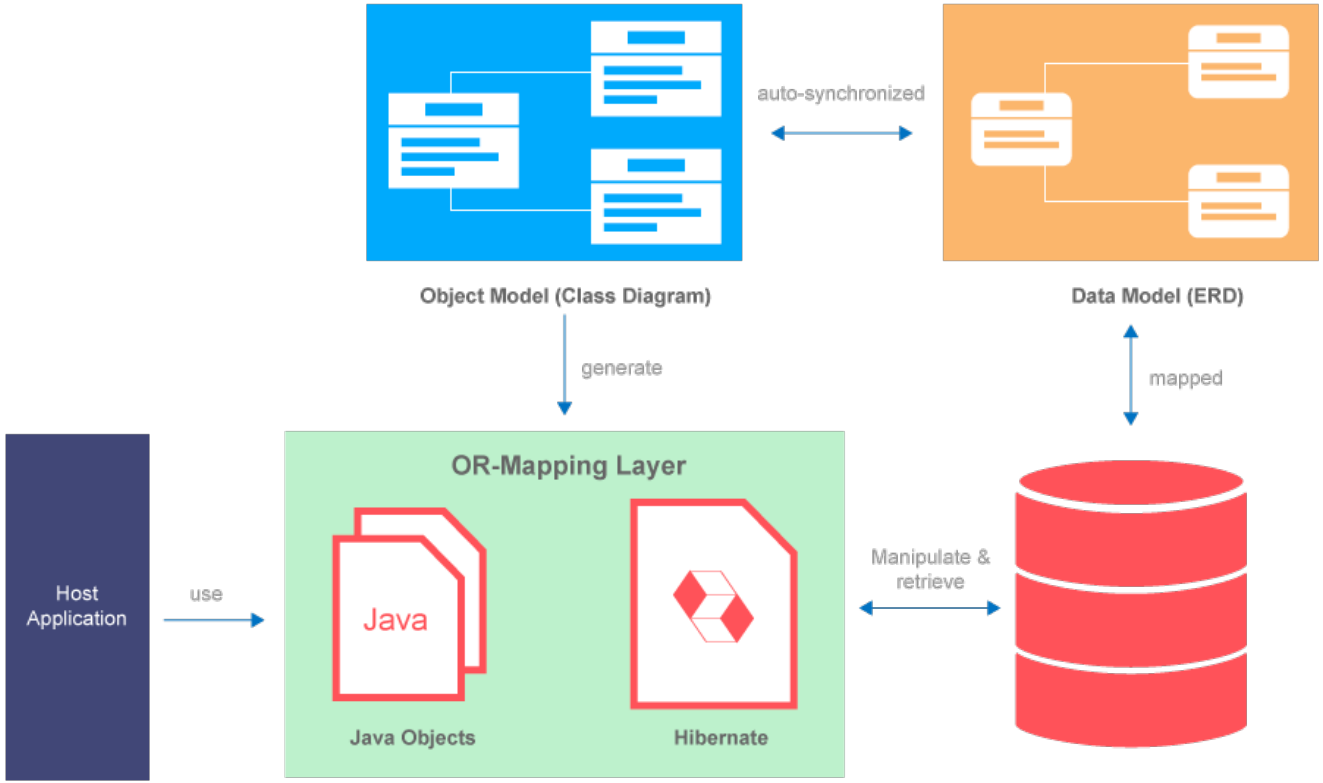
```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import sessionmaker

engine = create_engine('mysql://dbuser:dbuser@localhost:3306/University')
Session = sessionmaker(bind=engine)
session = Session()

Base = declarative_base()


class Student(Base):
    __tablename__ = 'students2'

    uni = Column(String, primary_key=True)
    last_name = Column(String)
    first_name = Column(String)
    year = Column(String)
    school = Column(String)

    def __repr__(self):
        return "<User(uni='%s', last_name='%s', first_name='%s', year='%s', school='%s')>" % (
            self.uni, self.last_name, self.first_name, self.year, self.school)
```

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

```python
def test1():
    q = session.query(Student)
    result = q.all()
    print("All students:")
    for s in result:
        print(s)


def test2():
    q = session.query(Student)
    result =\
        q.filter_by(last_name='Ferguson').all()
    print("Some students:")
    for s in result:
        print(s)


def test3():
    q = session.query(Student)
    result = q.all()
    print("Some student names:")
    for s in result:
        print(s.first_name, s.last_name)
```

```
All students:
<User(uni='FED00', last_name='Ferguson', first_name='Donald', year='2018', school='C')>
<User(uni='FED02', last_name='Ferguson', first_name='Donald', year='2019', school='C')>
<User(uni='GAWI1', last_name='Gandalf', first_name='Wizard', year='2018', school='E')>
<User(uni='GRHE1', last_name='Grainger', first_name='Hermione', year='2020', school='C')>
<User(uni='HAP01', last_name='Potter', first_name='Harry', year='2021', school='C')>
<User(uni='SMJ01', last_name='Smith', first_name='John', year='0', school='C')>


Some students:
<User(uni='FED00', last_name='Ferguson', first_name='Donald', year='2018', school='C')>
<User(uni='FED02', last_name='Ferguson', first_name='Donald', year='2019', school='C')>


Some student names:
Donald Ferguson
Donald Ferguson
Wizard Gandalf
Hermione Grainger
Harry Potter
John Smith
```

Java Hibernate



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.

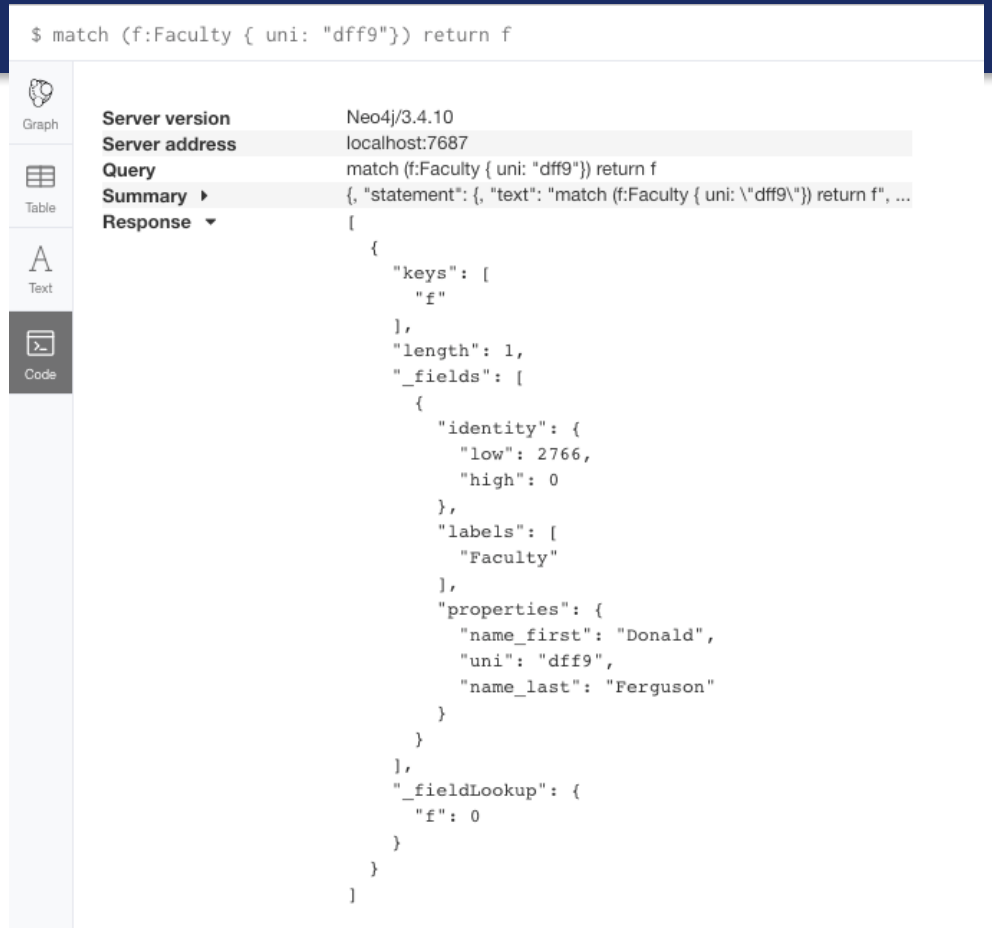
Graph Database Completion

Neo4j and HW4

A Simple Neo4j Query

A Simple Neo4j Query

- What is really happening?
  - The query is returning a complex data structure.
  - Code running in JavaScript in the browser is
    - Parsing response.
    - Laying out the graph.
- When
  - You directly use a language driver/connector.
  - You get a similarly complex data structure.
  - Mapping into the language constructs.



Using a *driver*

```
import json
from HW4Template.utils import utils as ut

from neo4j.v1 import GraphDatabase

ut.set_debug_mode(False)

_driver = GraphDatabase.driver(uri="bolt://localhost:7687", auth=("neo4j",
                    "XXXXXXX"))

def close(self):
    self._driver.close()
```

Using a *driver*

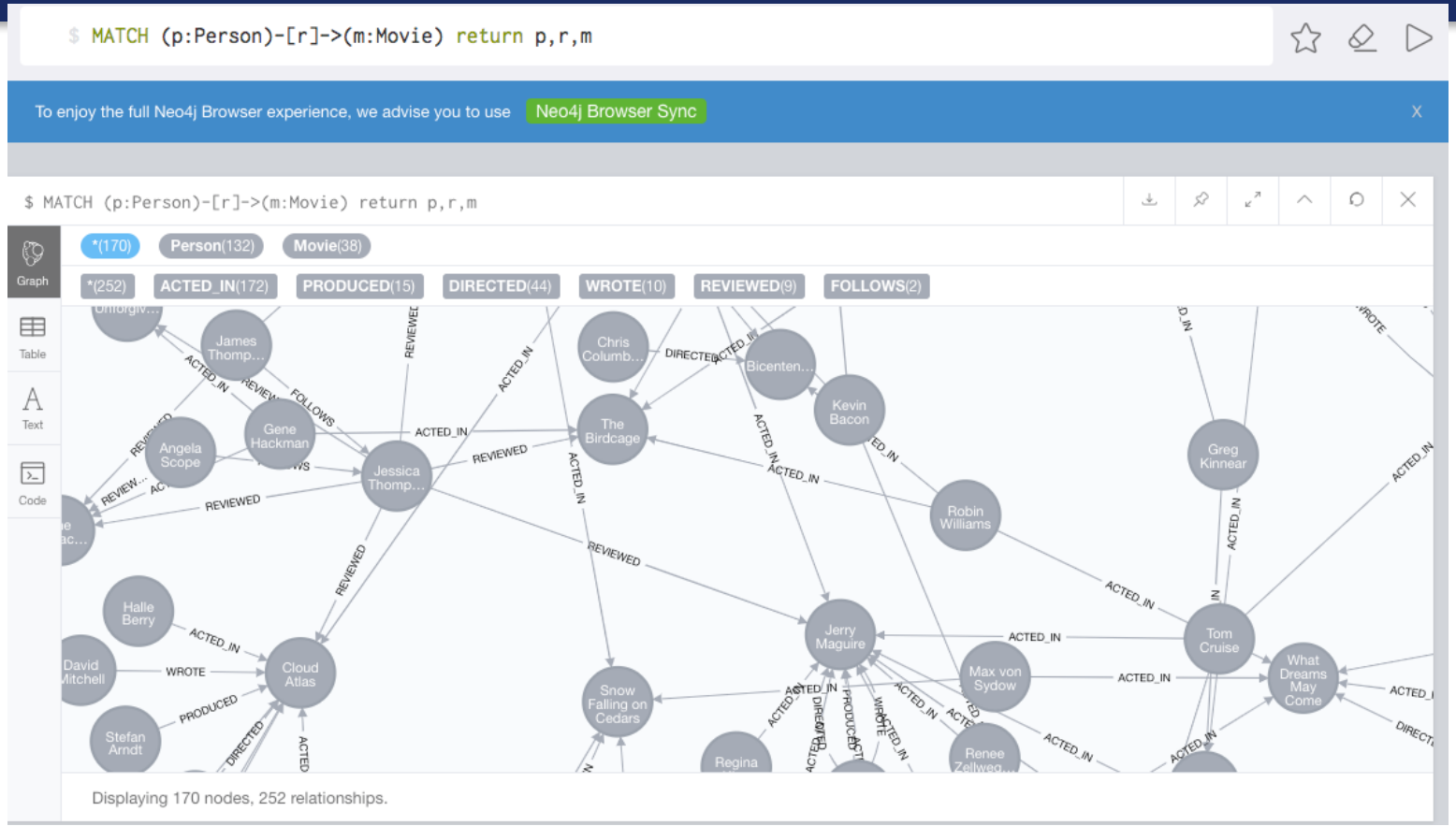
```
def get_faculty(uni):
    with _driver.session() as session:
        q = "match (n:Faculty {uni : '" + uni + "'}) return n"
        ut.debug_message("get_faculty: query  = ", q)
        qr = session.run(q)
        ut.debug_message("get_faculty: qr = ", qr)
        result = []
        for r in qr:
            ut.debug_message("get_faculty: r = ", r)
            r0 = r[0]
            ut.debug_message("get_faculty: r[0] = ", r0)
            ut.debug_message("get_faculty: r[0].labels = ", r0.labels)
            ut.debug_message("get_faculty: r[0].items() = ", r0.items())
            dd = dict(r0.items())
            ut.debug_message("get_faculty: dict(r[0].items()) = ", dict(r0.items()))

            # Convert to something more intuitive.
            temp_r = {"labels": list(r0.labels), "properties": dd}
            result.append(temp_r)
        return result
```

Using a *driver*

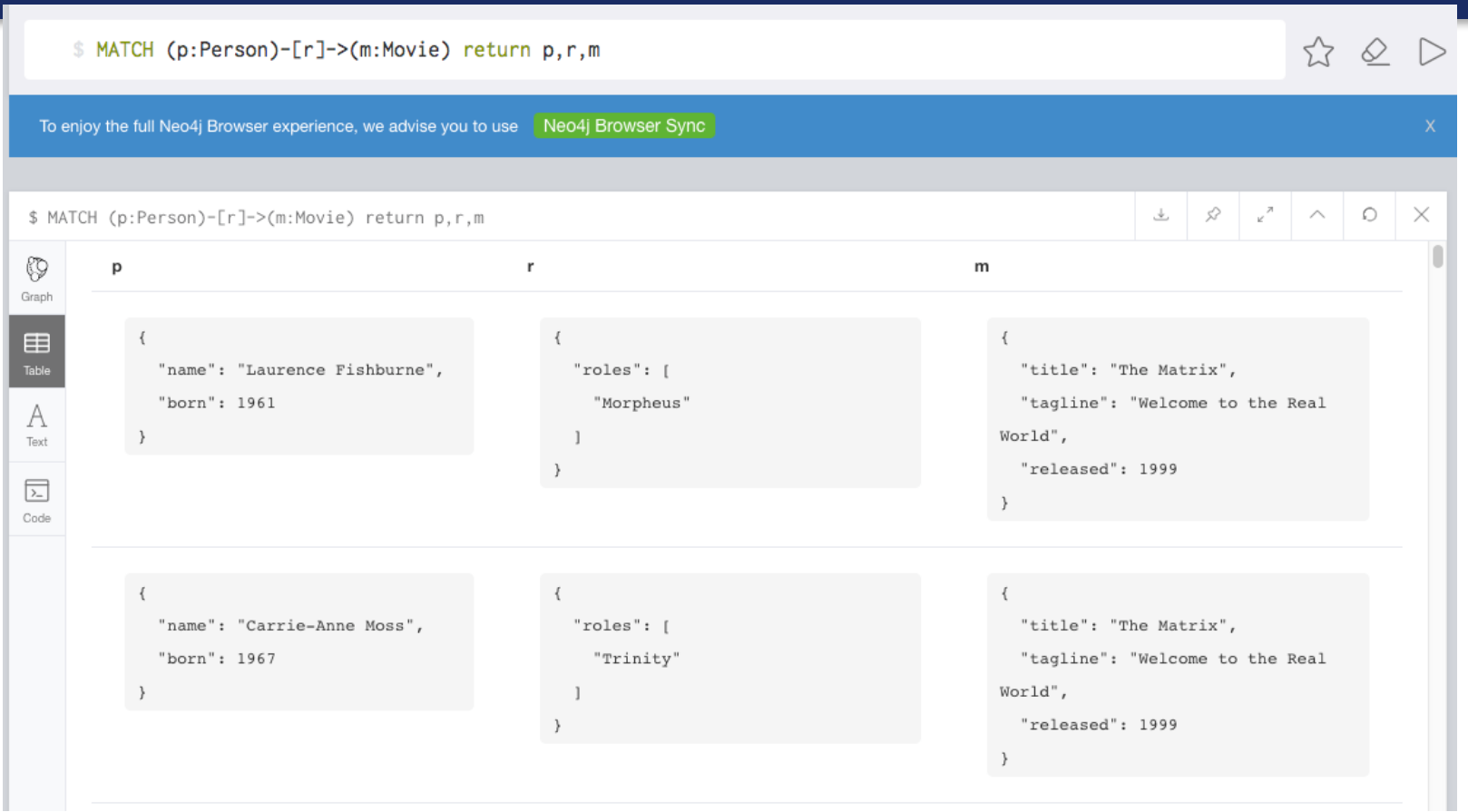
```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/donaldferguson/Dropbox/ColumbiaCourse/Courses/Fall2018/W4111/HW4T
get_faculty: query  =  "match (n:Faculty {uni : 'dff9'}) return n"
get_faculty: qr =  "<neo4j.v1.result.BoltStatementResult object at 0x10729e0b8>"
get_faculty: r =  [
  "<Node id=2766 labels={'Faculty'} properties={'name_first': 'Donald', 'uni': 'dff9', 'name_last': 'Ferguson'}>"
]
get_faculty: r[0] =  "<Node id=2766 labels={'Faculty'} properties={'name_first': 'Donald', 'uni': 'dff9', 'name_last': 'Ferguson'}>"
get_faculty: r[0].labels =  "frozenset({'Faculty'})"
get_faculty: r[0].items() =  "dict_items([('name_first', 'Donald'), ('uni', 'dff9'), ('name_last', 'Ferguson')])"
get_faculty: dict(r[0].items()) =  {
  "name_first": "Donald",
  "uni": "dff9",
  "name_last": "Ferguson"
}
Test1 result =  [{"labels": ["Faculty"], "properties": {"name_first": "Donald", "uni": "dff9", "name_last": "Ferguson"}}]
```

- Relational database queries operate on tables and produce tables.
- Neo4j queries *operate* on *graphs* and (usually) *product graphs*.
- Programming languages do not directly support tables or graphs.
- So, the drivers convert the query results into the programming languages types and constructs:
  - cursor() and list[] or OrderedDict{} for relational and Python.
  - *BoltStatementResult* for Neo4j and Python

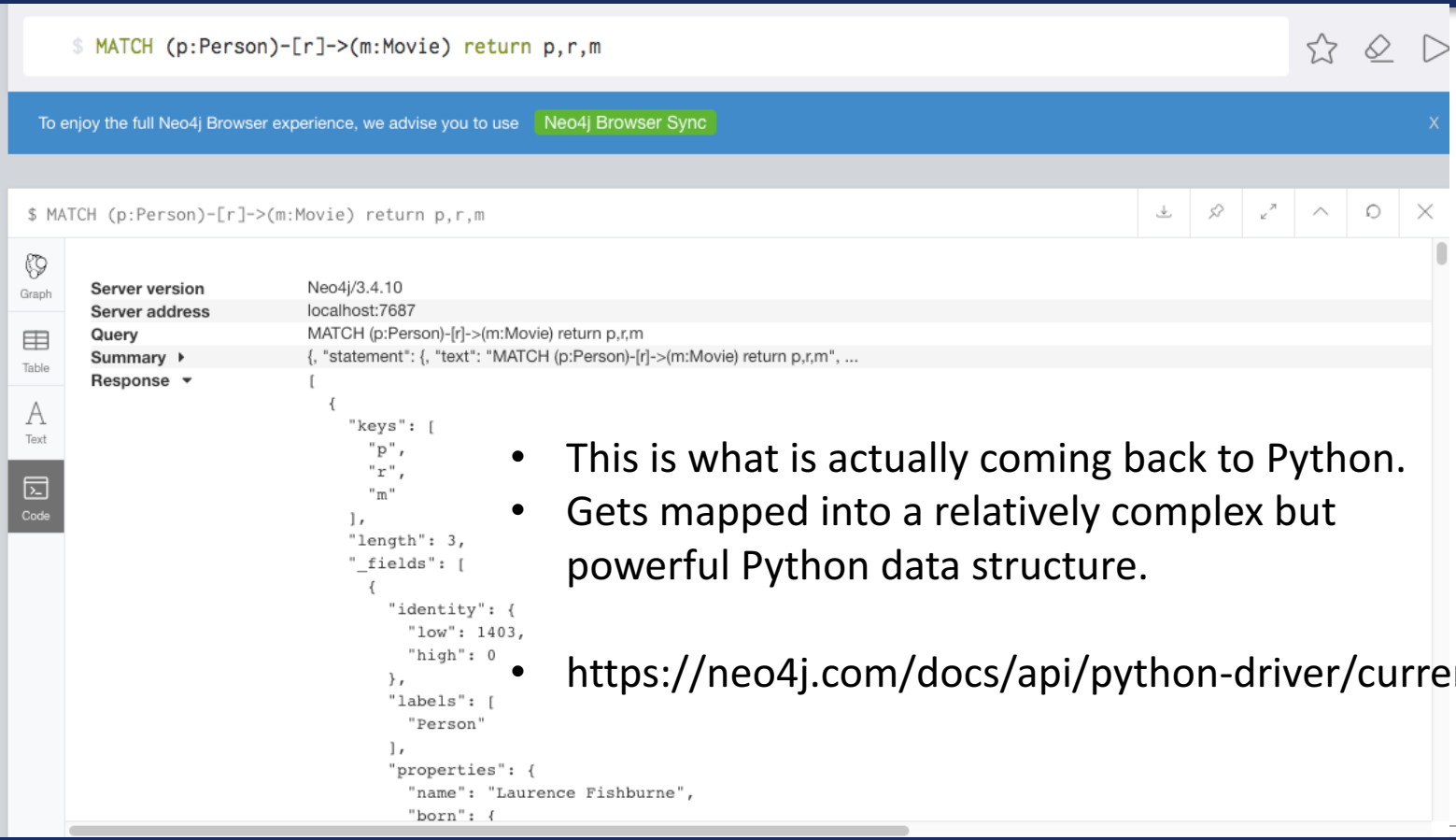
Graphs



Graphs



Graphs



- This is what is actually coming back to Python.
- Gets mapped into a relatively complex but powerful Python data structure.
- https://neo4j.com/docs/api/python-driver/current/results.html

py2neo, queries and HW4

```python
from py2neo import Graph, NodeMatcher, \
    Node, Relationship, RelationshipMatcher
import json
from HW4Template.utils import utils as ut
import uuid

_graph = Graph(secure=False,
                    auth=("dbuser", "dbuser"),
                    bolt=True,
                    host="localhost",
                    port=7687)

_node_matcher = NodeMatcher(_graph)
_relationship_matcher = RelationshipMatcher(_graph)


def test2():
    some_actors = _node_matcher.match("Person", born=1962)
    for a in some_actors:
        print("Actor = ", a)
        movies = _relationship_matcher.match([a], "ACTED_IN")
        for m in movies:
            print("\t", m)
            print("\t", m.end_node)
        print("\n")
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 /Users/donaldferguson/Dropbox/ColumbiaCourse/Cour
Actor =  (_2761:Person {born: 1962, name: 'Tom Cruise'})
(Tom Cruise)-[:ACTED_IN {roles: ['Lt. Daniel Kaffee']}]->(_2760)
     (_2760:Movie {released: 1992, tagline: "In the heart of the nation's capital, in a courthouse of the U.S. go
(Tom Cruise)-[:ACTED_IN {roles: ['Jerry Maguire']}]->(_6256)
     (_6256:Movie {released: 2000, tagline: 'The rest of his life begins now.', title: 'Jerry Maguire'})
(Tom Cruise)-[:ACTED_IN {roles: ['Maverick']}]->(_6248)
     (_6248:Movie {released: 1986, tagline: 'I feel the need, the need for speed.', title: 'Top Gun'})


Actor =  (_2763:Person {born: 1962, name: 'Demi Moore'})
(Demi Moore)-[:ACTED_IN {roles: ['Lt. Cdr. JoAnne Galloway']}]->(_2760)
     (_2760:Movie {released: 1992, tagline: "In the heart of the nation's capital, in a courthouse of the U.S. go


Actor =  (_6251:Person {born: 1962, name: 'Anthony Edwards'})
(Anthony Edwards)-[:ACTED_IN {roles: ['Goose']}]->(_6248)
     (_6248:Movie {released: 1986, tagline: 'I feel the need, the need for speed.', title: 'Top Gun'})


Actor =  (_6258:Person {born: 1962, name: 'Kelly Preston'})
(Kelly Preston)-[:ACTED_IN {roles: ['Avery Bishop']}]->(_6256)
     (_6256:Movie {released: 2000, tagline: 'The rest of his life begins now.', title: 'Jerry Maguire'})


Actor =  (_6296:Person {born: 1962, name: "Rosie O'Donnell"})
(Rosie O'Donnell)-[:ACTED_IN {roles: ['Doris Murphy']}]->(_6381)
     (_6381:Movie {released: 1992, tagline: 'Once in a lifetime you get a chance to do something different.', tit
(Rosie O'Donnell)-[:ACTED_IN {roles: ['Becky']}]->(_6292)
     (_6292:Movie {released: 1993, tagline: 'What if someone you never met, someone you never saw, someone you ne
```

Without py2neo

```
def test2():
    with _driver.session() as session:
        q = "match (n:Person {born : 1962})-[r:ACTED_IN]->(m) return n,m,r"
        qr = session.run(q)
        current_actor = None
        print("Raw result = ", qr)
        for q in qr:
            a = q['n']
            if a != current_actor:
                current_actor = a
                print("Actor = ", json.dumps(dict(q['n'])), "acte
            print("\tRole: ", json.dumps(dict(q['r'])))
            print("\tMovie: ", json.dumps(dict(q['m'])['title']))
```

```
Actor =  {"name": "Tom Cruise", "born": 1962} acted in:
    Role:  {"roles": ["Lt. Daniel Kaffee"]}
    Movie:  "A Few Good Men"
    Role:  {"roles": ["Jerry Maguire"]}
    Movie:  "Jerry Maguire"
    Role:  {"roles": ["Maverick"]}
    Movie:  "Top Gun"
Actor =  {"name": "Demi Moore", "born": 1962} acted in:
    Role:  {"roles": ["Lt. Cdr. JoAnne Galloway"]}
    Movie:  "A Few Good Men"
Actor =  {"name": "Anthony Edwards", "born": 1962} acted in:
    Role:  {"roles": ["Goose"]}
    Movie:  "Top Gun"
Actor =  {"name": "Kelly Preston", "born": 1962} acted in:
    Role:  {"roles": ["Avery Bishop"]}
    Movie:  "Jerry Maguire"
Actor =  {"name": "Rosie O'Donnell", "born": 1962} acted in:
    Role:  {"roles": ["Doris Murphy"]}
    Movie:  "A League of Their Own"
    Role:  {"roles": ["Becky"]}
    Movie:  "Sleepless in Seattle"
```

Comparison – Which is Easier?

```
def test2():
    with _driver.session() as session:
        q = "match (n:Person {born : 1962})-[r:ACTED_IN]->(m) return n,m,r"
        qr = session.run(q)
        current_actor = None
        print("Raw result = ", qr)
        for q in qr:
            a = q['n']
            if a != current_actor:
                current_actor = a
                print("Actor = ", json.dumps(dict(q['n'])), "acted in:")
            print("\tRole: ", json.dumps(dict(q['r'])))
            print("\tMovie: ", json.dumps(dict(q['m'])['title']))
```

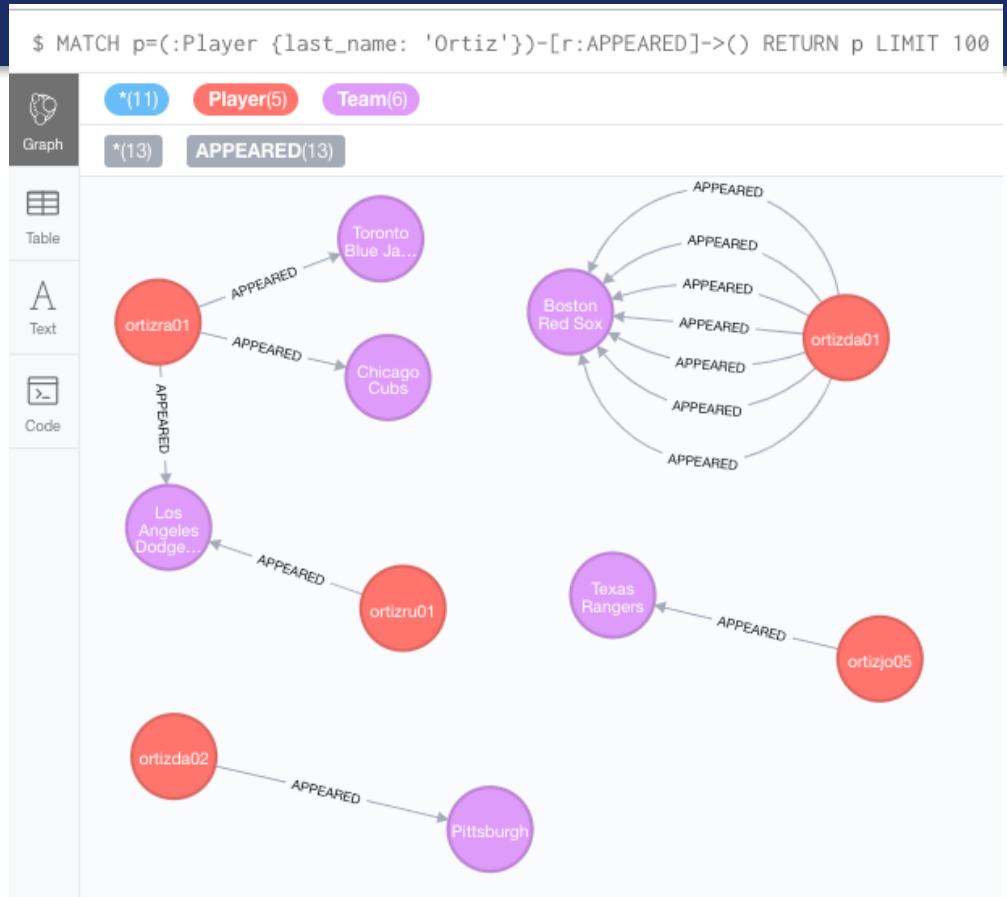
Direct use of language driver.

py2neo

```
def test2():
    some_actors = _node_matcher.match("Person", born=1962)
    for a in some_actors:
        print("Actor = ", a)
        movies = _relationship_matcher.match([a], "ACTED_IN")
        for m in movies:
            print(m)
            print("\t", m.end_node)
        print("\n")
```

Interesting Query 1

- Note: I loaded
  - All appearances since 2010.
  - Players that appeared.
- Find all of the appearances by a player with last_name ‘Ortiz’
- Note 2:
  - Cannot create multiple relationships with same name using py2neo.
  - I had to use raw Cypher



Interesting Query 2 – Loading All Appearances

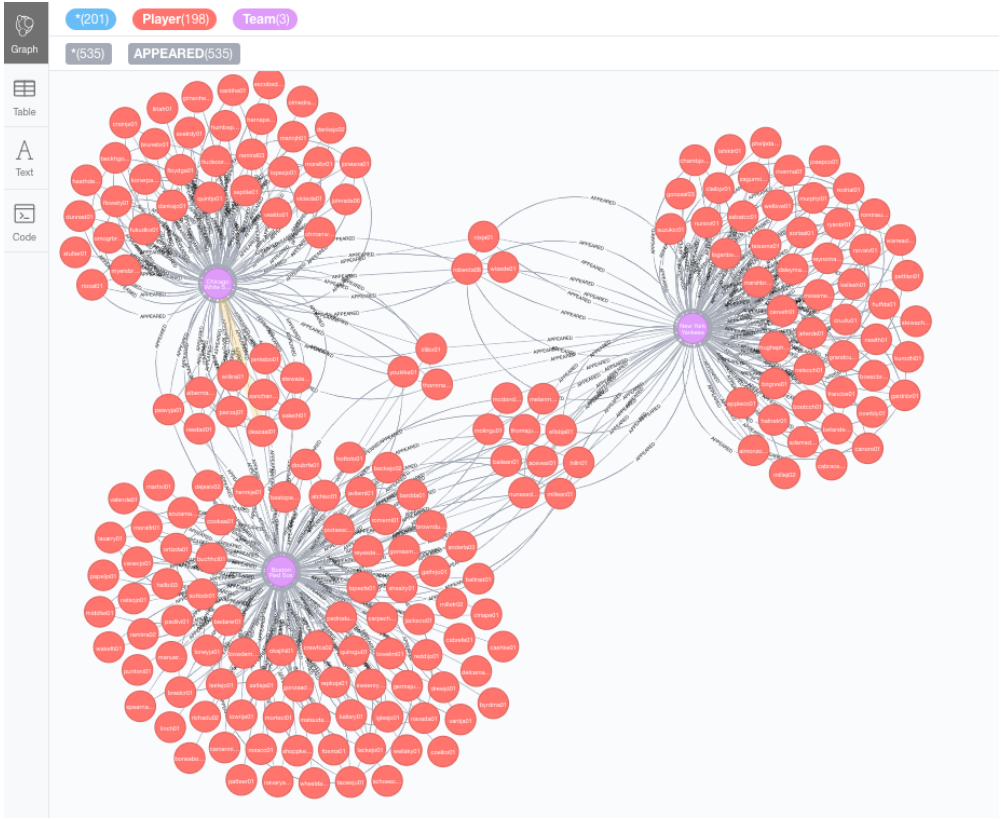
```
def load_appearances():

    q = "SELECT distinct playerid, teamid, yearid, g_all as games from appearances where yearid >= 2010"

    curs = cnx.cursor()
    curs.execute(q)

    r = curs.fetchone()
    cnt = 0
    while r is not None:
        print(r)
        cnt += 1

        if r is not None:
            try:
                p = fg.create_appearance_all(team_id=r['teamid'], player_id=r['playerid'], \
                                             games=r['games'], year=r['yearid'])
                print("Created appearances = ", json.dumps(p))
            except Exception as e:
                print("Could not create.")
        r = curs.fetchone()

    print("Loaded ", cnt, "records.")
```

Interesting Query 2 – Loading All Appearances

```
# Create an APPEARED relationship from a player to a Team
def create_appearance_all(self, player_id, team_id, year, games):
    try:
        tx = self._graph.begin(autocommit=False)
        q = "match (n:Player {player_id: '" + player_id + "'}), " + \
            "(t:Team {team_id: '" + team_id + "'}) " + \
            "create (n)-[r:APPEARED { games: " + str(games) + ", year : " + str(year) +
            "}]->(t)"
        result = self._graph.run(q)
        tx.commit()
    except Exception as e:
        print("create_appearances: exception = ", e)
```

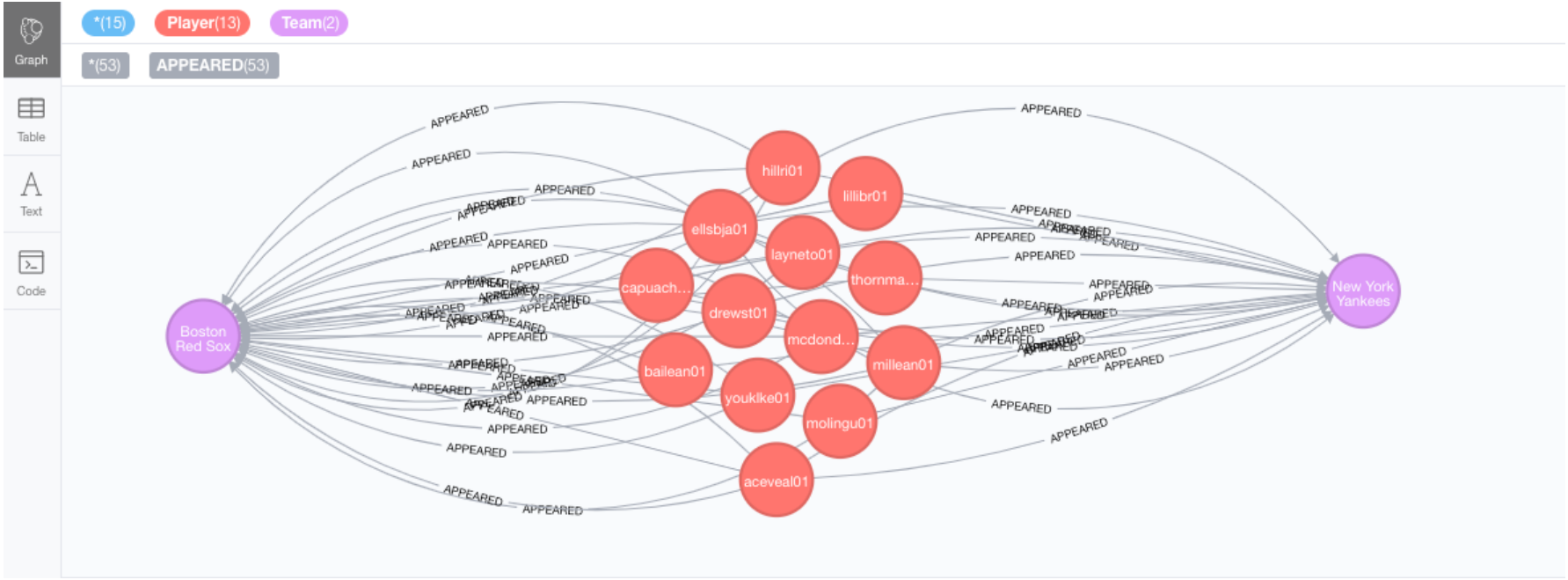
Kevin Youkilis' Teammates

```
MATCH  c=(p:Player {last_name: 'Youkilis'})
-[r1:APPEARED]->
(t:Team)
<-[r2:APPEARED]-
(p2:Player)
where not p = p2  and r1.year = r2.year
RETURN  c
```



Find Recent Traitors

```
MATCH  (p:Player)-[r1:APPEARED]->(t:Team {team_id: "NYA"}),
(p:Player)-[r2:APPEARED]->(t2:Team {team_id: "BOS"})
where r2.year < r1.year return p,r1,t,r2,t2
```



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

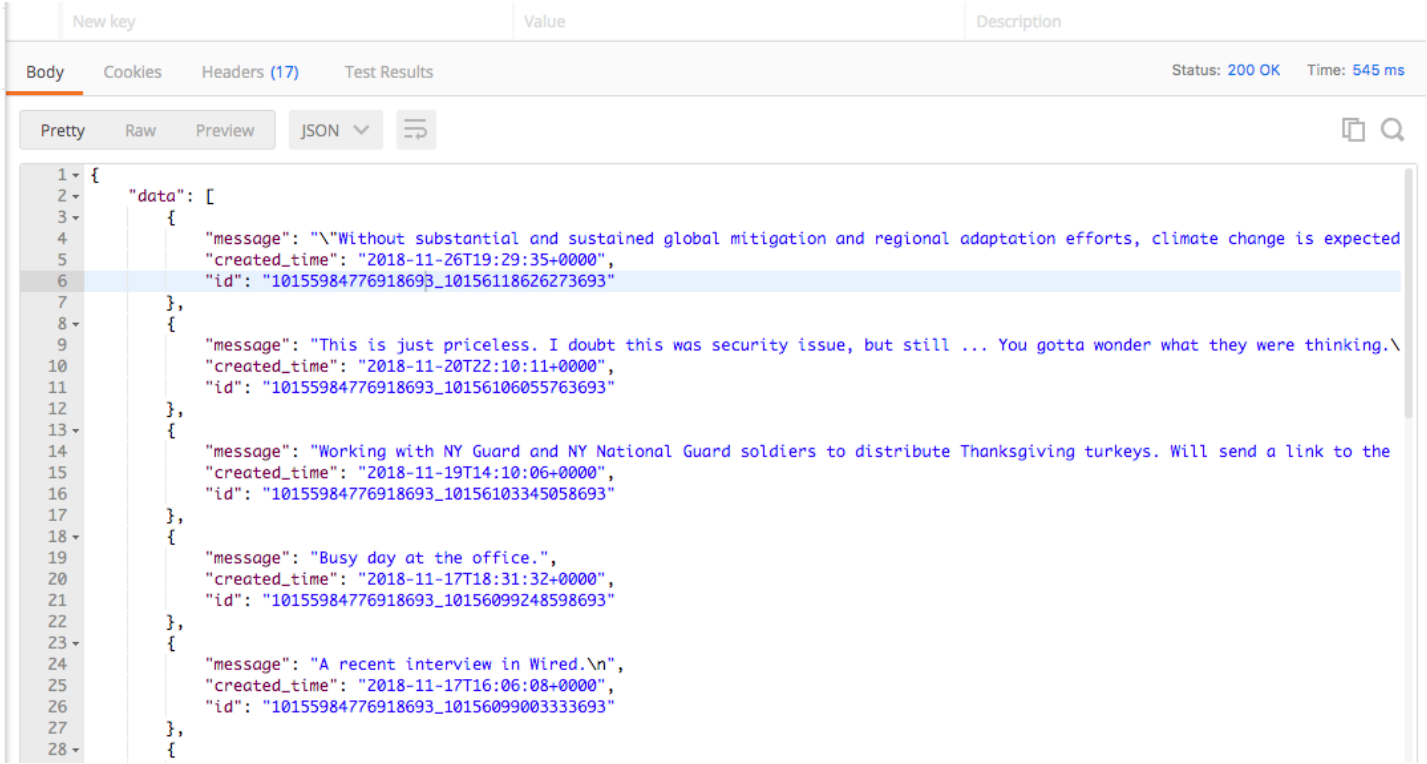
Full list at:
https://developers.facebook.com/docs/graph-api/reference

... ...

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

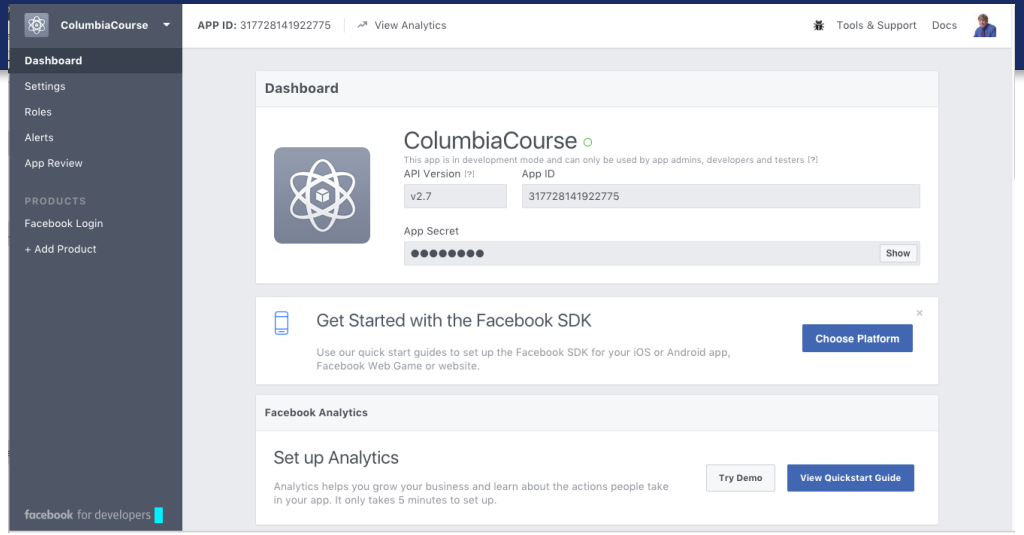
GET https://graph.facebook.com/v3.2/me/likes?access_token=EAAE2HWkr2kIBANW8p4YZCiZA82WiRkqabbKnw5GEA... Params Send Save

Pretty Raw Preview JSON


```
            "id": "120885117945175",
            "created_time": "2010-09-04T18:57:10+0000"
        },
        {
            "name": "Making it Big in Software: Get the job. Work the org. Become great.",
            "id": "337756296552",
            "created_time": "2010-03-27T10:54:20+0000"
        },
        {
            "name": "Tim's Sounds",
            "id": "311405670729",
            "created_time": "2010-02-06T14:56:33+0000"
        },
        {
            "name": "Erasmus Mundus IMSE",
            "id": "186881751274",
            "created_time": "2009-12-06T14:18:14+0000"
        }
    ],
    "paging": {
        "cursors": {
            "before": "MTE1MDM3NTQ4NTE0NzMx",
            "after": "MTg2ODgxNzUxMjc0"
        },
        "next": "https://graph.facebook.com/v3.2/10155984776918693/likes?access_token=EAAE2HWkr2kIBANW8p4YZCiZA8ZWiRkqabbKnw5GEAMZAORuqJ
    }
}
```


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

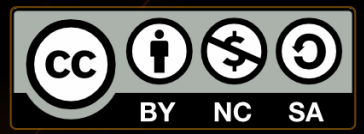
- Popularity and usage has exploded in the past few years.
  - Facebook Graph API (https://developers.facebook.com/docs/graph-api/)
  - LinkedIn (https://developer.linkedin.com/docs/rest-api)
  - Instagram (https://developers.facebook.com/docs/instagram-api)
  - Google Knowledge Graph (https://developers.google.com/knowledge-graph/)

- 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?

- 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

- 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

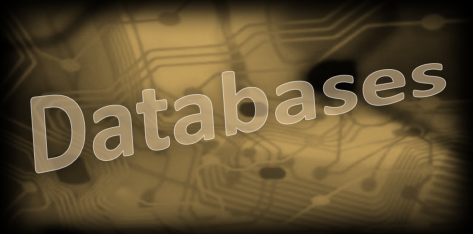
Add a user "**maria**".

List all users

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

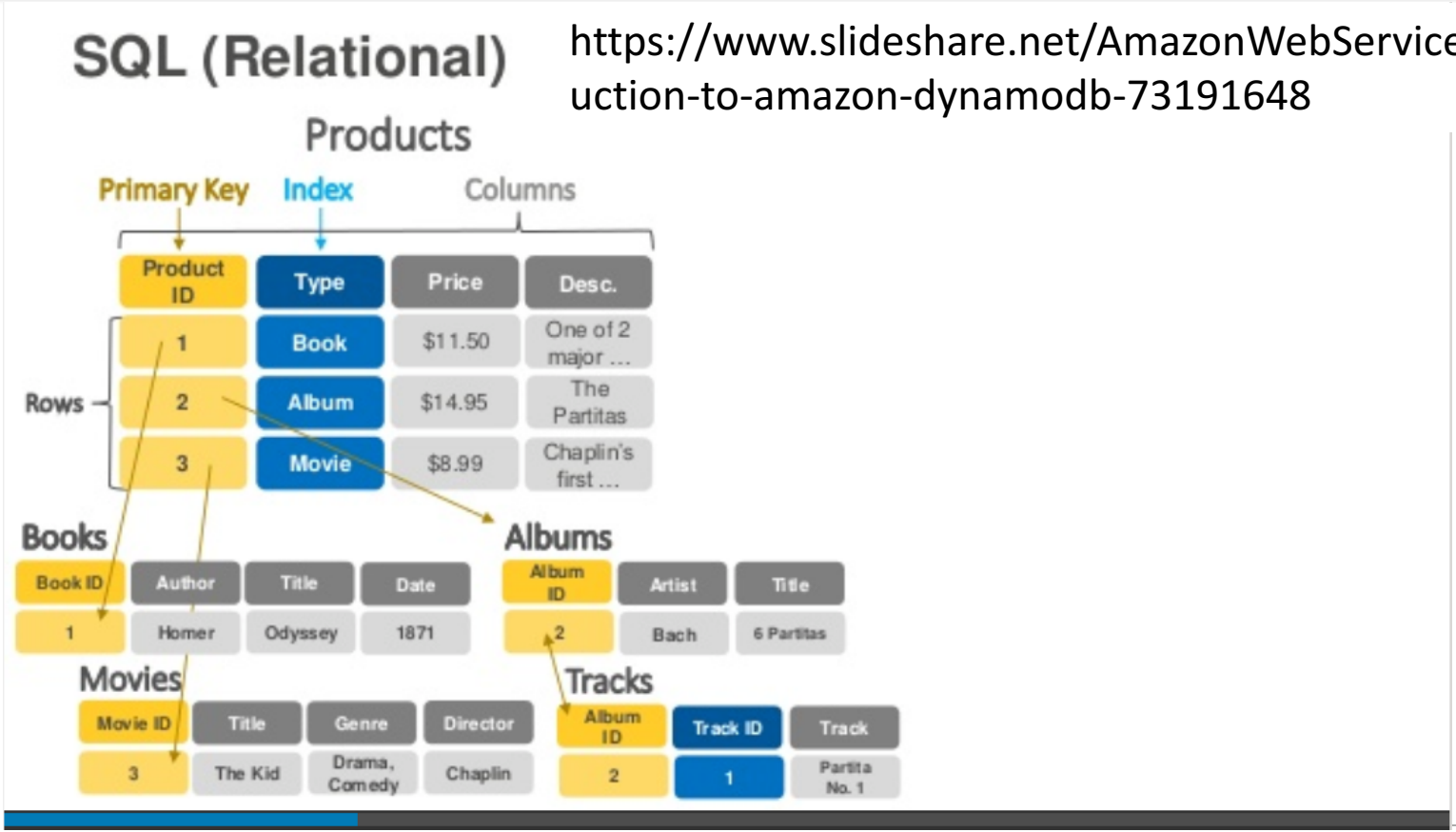
Summary

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



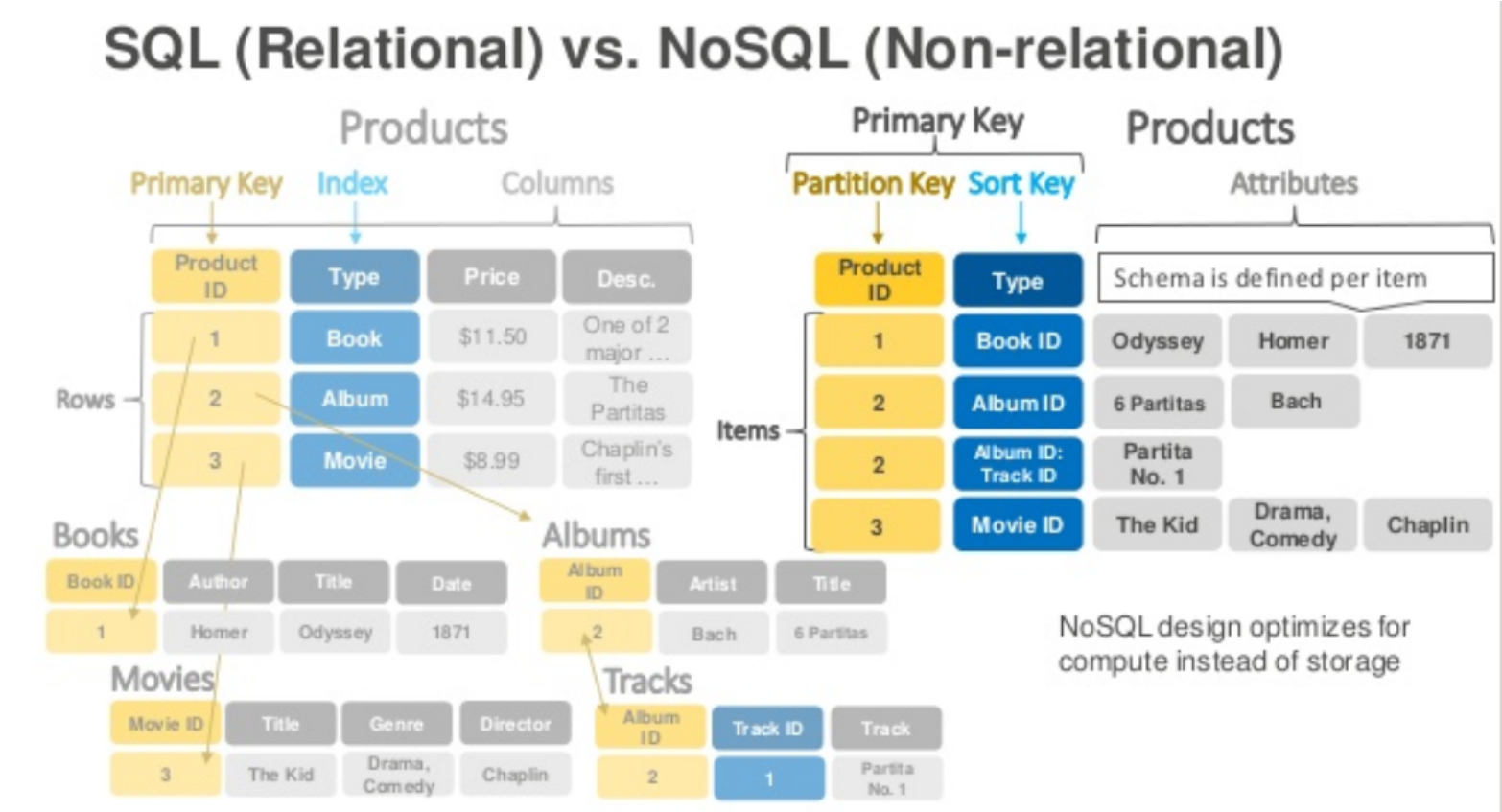
DynamoDB

DynamoDB Data Model



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

DynamoDB Data Model



DynamoDB Benefits


Fully managed


Fast, consistent performance

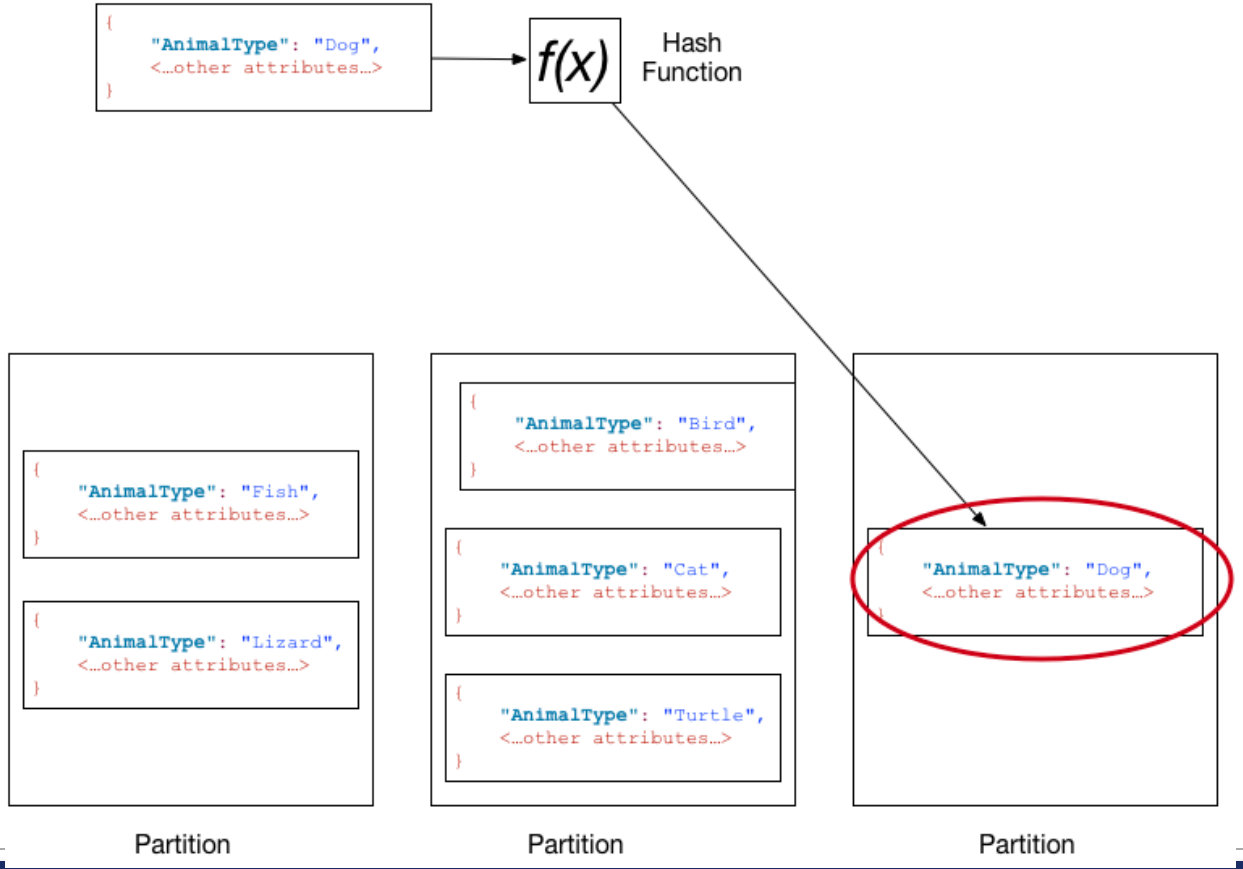

Highly scalable


Flexible

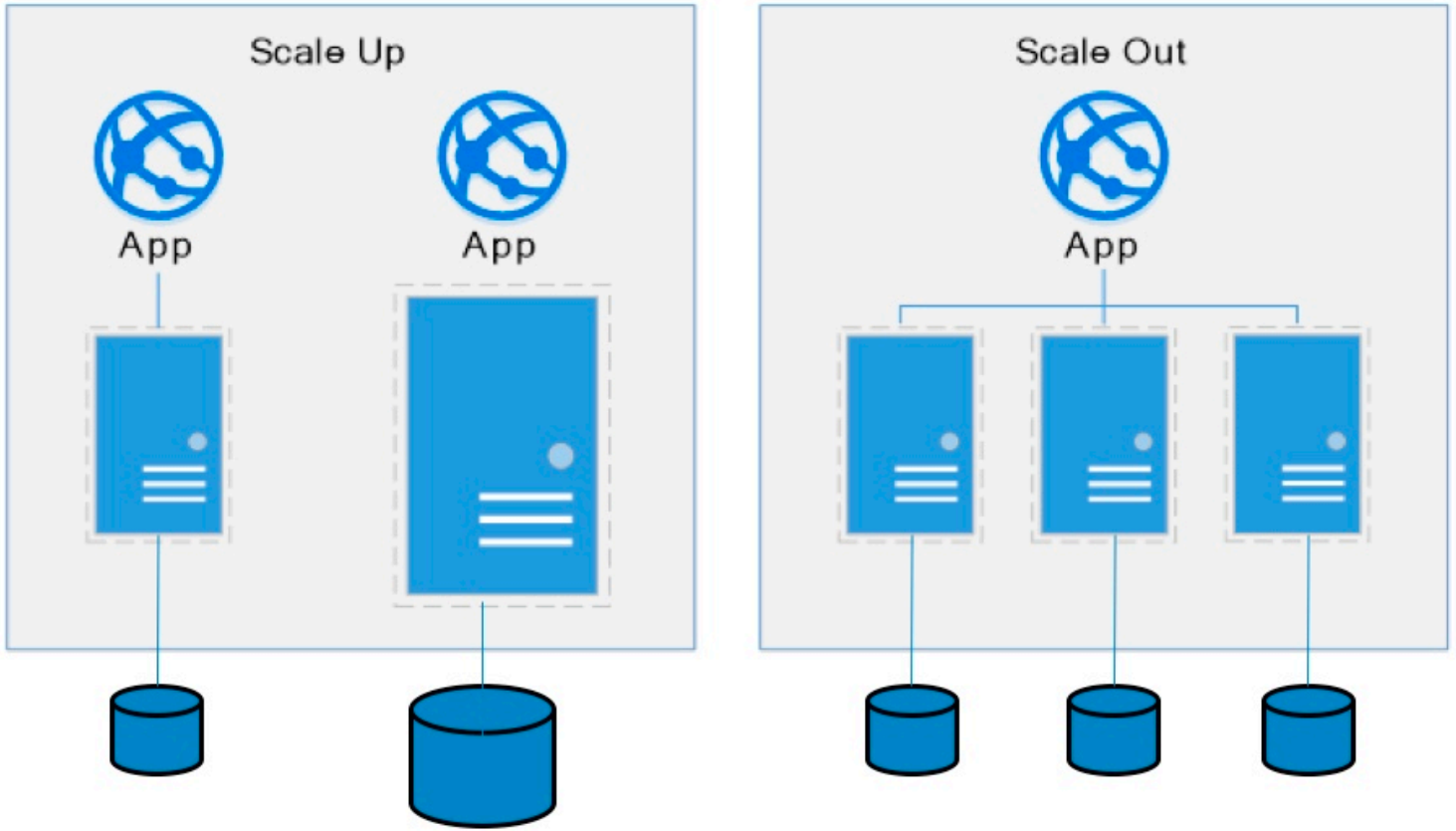

Event-driven programming


Fine-grained access control

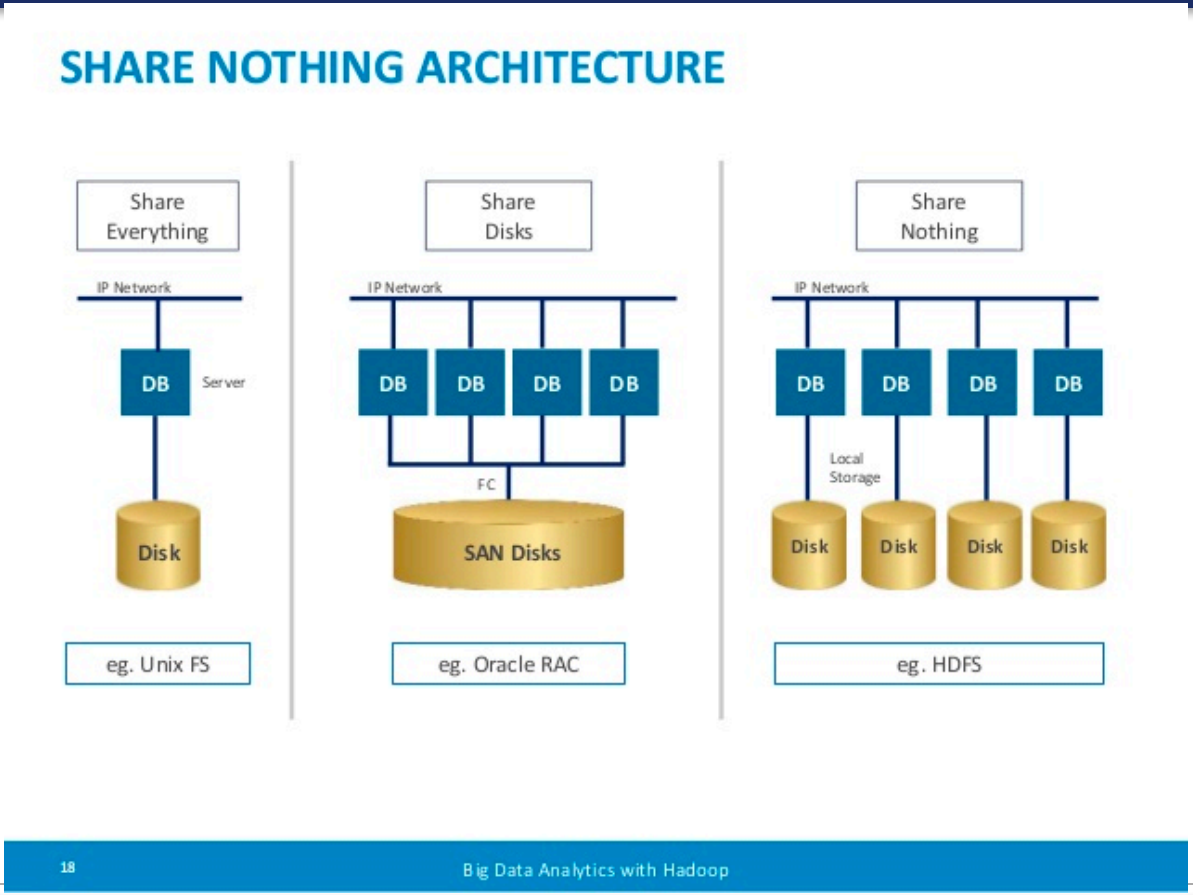
DynamoDB Hashing



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"
}
```

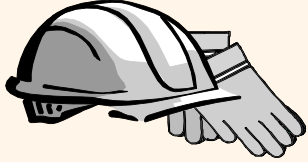
DynamoDB Summary

- 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

Normalization/De-normalization



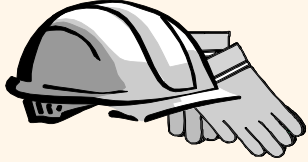
Schema Refinement and Normal Forms

Chapter 19

Note:

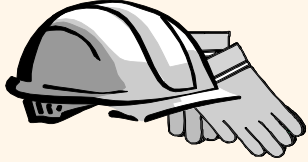
- There is a theoretical/formal basis for normalization in the relational model.
- Has practical application to real scenarios and data models.
- Primarily applied intuitively and based on common sense, not theory.
- But, like relational algebra, I have to give you an overview of the theory.

The Evils of Redundancy



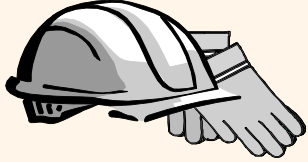
- *Redundancy* is at the root of several problems associated with relational schemas:
  - redundant storage, insert/delete/update anomalies
- Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.
- Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).
- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?

Functional Dependencies (FDs)



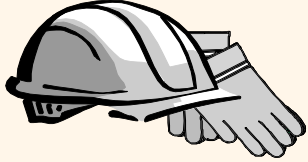
- A functional dependency X→ Y holds over relation R if, for every allowable instance *r* of R:
  - $t1 \in r,\ t2 \in r, \pi_X(t1) = \pi_X(t2)$ implies $\pi_Y(t1) = \pi_Y(t2)$
  - i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree. (X and Y are *sets* of attributes.)
- An FD is a statement about *all* allowable relations.
  - Must be identified based on semantics of application.
  - Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R!
- K is a candidate key for R means that K→ R
  - However, K→ R does not require K to be *minimal*!

Example: Constraints on Entity Set



- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (*ssn, name, lot, rating, hrly_wages, hrs_worked*)
- *Notation*: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the *set* of attributes {S,N,L,R,W,H}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)
- Some FDs on Hourly_Emps:
  - *ssn* is the key: S $\rightarrow$ SNLRWH
  - *rating* determines *hrly_wages*: R $\rightarrow$ W

Example (Contd.)



Wages

R	W
8	10
5	7

Hourly_Emps2

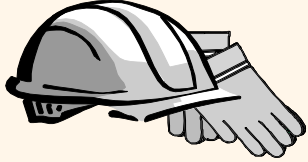
S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

- Problems due to R → W :
  - *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
  - *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
  - *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Will 2 smaller tables be better?

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Example: Constraints on Entity Set

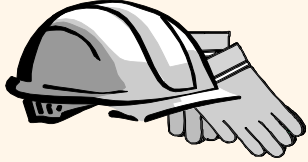


- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (*ssn, name, lot, rating, hrly_wages, hrs_worked*)
- *Notation*: We will denote this relation schema by

A more intuitive example – Consider a table of addresses.
- Addresses(id, street_no, street_name, city, country, zipcode)
- (city, country) are *functionally dependent* on *zipcode*.

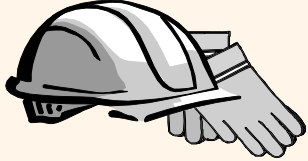
- Some FDs on Hourly_Emps:
  - *ssn* is the key: S → SNLRWH
  - *rating* determines *hrly_wages*: R → W

Reasoning About FDs



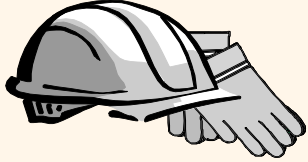
- Given some FDs, we can usually infer additional FDs:
  - *ssn* $\rightarrow$ *did*, *did* $\rightarrow$ *lot* implies *ssn* $\rightarrow$ *lot*
- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - F^+ = *closure of F* is the set of all FDs that are implied by *F*.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - *Reflexivity*: If $X \subseteq Y$, then $Y \rightarrow X$
  - *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These are *sound* and *complete* inference rules for FDs!

Normal Forms



- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given A → B: Several tuples could have the same A value, and if so, they'll all have the same B value!

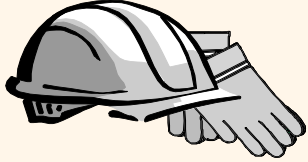
Boyce-Codd Normal Form (BCNF)



- Reln R with FDs *F* is in BCNF if, for all X→ A in F^+
  - A ∈ X (called a *trivial* FD), or
  - X contains a key for R.
- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If example relation is in BCNF, the 2 tuples must be identical (since X is a key).

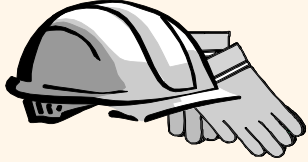
X	Y	A
x	y1	a
x	y2	?

Third Normal Form (3NF)

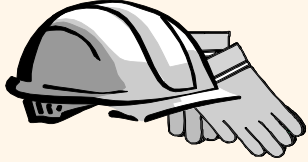


- Reln R with FDs *F* is in 3NF if, for all X→ A in F^+
  - A ∈ X (called a *trivial* FD), or
  - X contains a key for R, or
  - A is part of some key for R.
- *Minimality* of a key is crucial in third condition above!
- If R is in BCNF, obviously in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
  - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

Example Decomposition



- Decompositions should be used only when needed.
  - SNLRWH has FDs S → SNLRWH and R → W
  - Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
    - i.e., we decompose SNLRWH into SNLRH and RW
- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?



Problems with Decompositions

- There are three potential problems to consider:
  - Some queries become more expensive.
    - e.g., How much did sailor Joe earn? (salary = W*H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
    - Fortunately, not in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations.
    - Fortunately, not in the SNLRWH example.
- *Tradeoff*: Must consider these issues vs. redundancy.