Computational Linguistics Seminar
Databases 101

Marc Verhagen
Brandeis University
Spring 2021

The banner image is a fragment of Primordial Soup at https://regenaxe.com/2017/01/17/primordial-soup/

Overview

- Assignments past and current & Flask examples
- Databases prehistory
- Relational databases
  - Relational algebra
  - SQL and embedded SQL
- Object oriented databases and NoSQL
- Flask and databases

Database

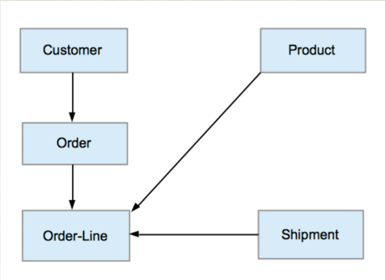
- Any organized collection of data
- Includes file systems and Python shelves
- Issues
  - Redundancy and inconsistency
  - Uniform access
  - Data integrity
  - Atomicity problems
  - Concurrency
  - Security

Prehistory

- Sequential access only, large average access times
- Can stream data very quickly
- Still used for off-line archival data storage
- Linux tar command

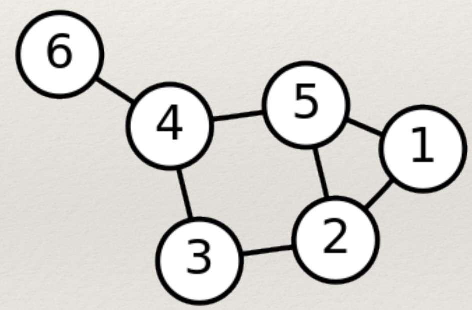
Random Access (hard drives)

- Starting in the sixties
- Hard drives provide direct access
- Batch processing $\Longrightarrow$ interactive use
- Variety of very procedural approaches
- CODASYL approach
  - Committee on Data Systems Languages
  - Network model
    - generalized graph of record types connected by relationship types



Navigational Model

- Navigation of a linked data set or network.
  - Use of a primary key (known as a CALC key, typically implemented by hashing)
  - Navigating relationships from one record to another via references from one object to the next
  - Scanning all the records in sequential order



- Point of critique: unstructured spaghetti mess

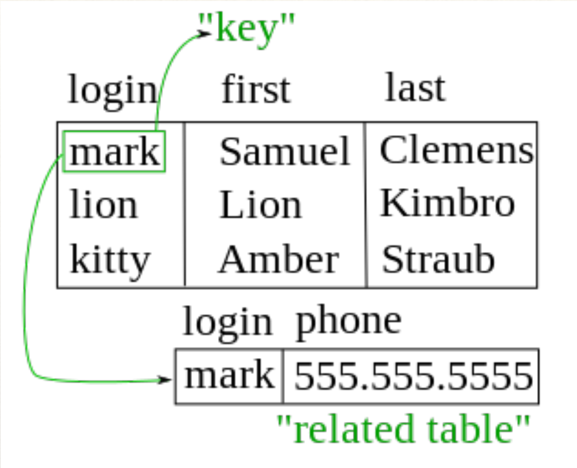
Some Notions

- Database: an organized collection of data
- Database management system (DBMS)
  - software application that interacts with the user, other applications, and the database itself to capture and analyze data.
- Schema (explicit and implicit)

Database types

- Relational Databases
  - PostgreSQL, MySQL, SQLite, Oracle, DB2, SQLServer
- Object Oriented Database
  - Gemstone
- NoSQL
  - Document-Oriented Databases, Graph Databases, Key-value databases
  - BigTable (Google), DynamoDB (Amazon), MarkLogic, Project Voldemort, MongoDB, ElasticSearch
  - Graph database: Neo4j, SPARQL (query language)
- Distributed DB, Cloud DB: HDFS
- Spatial DB, Temporal DB, Triple Stores (RDF)

Relational Database



Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

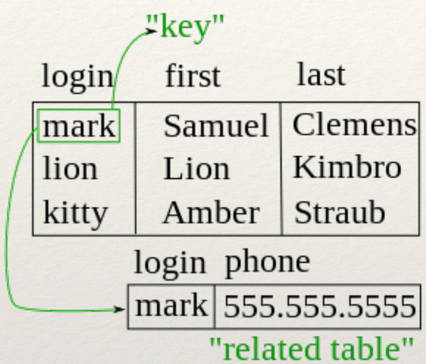
ID*	Activity1	Cost1	Activity2	Cost2
084	Tennis	$36	Swimming	$17
100	Squash	$40	Swimming	$17
182	Tennis	$36		
219	Swimming	$15	Golf	$47

- relational algebra
- primary key
- foreign key
- normalization
- index
- join
- views
- no pointers

Relational Algebra

- Procedural
- Not the same as Relational Calculus
  - a non-procedural query language which tells what to do but not how to do it.
- Operations on sets
  - select
  - project
  - union, difference and intersection
  - Cartesian product
  - natural-join
  - Others: rename, set-intersection and division

Select Operation

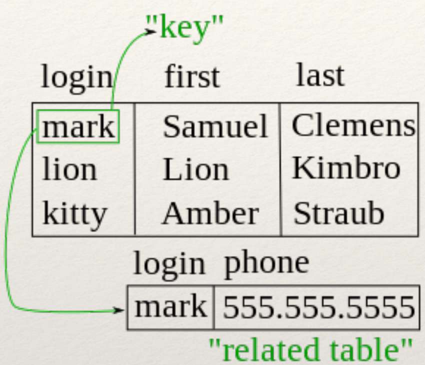


$\sigma_{\text{login=mark}}$ (names)

names		
mark	Samuel	Clemens
lion	Lion	Kimbro
kitty	Amber	Straub

$\sigma_{\text{login=mark}}$ (names)		
mark	Samuel	Clemens

Project Operation

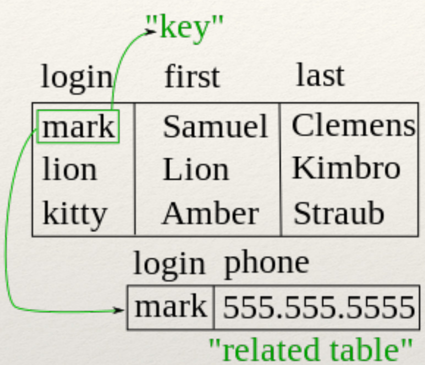


$\pi_{\text{login,first}}$ (names)

names		
mark	Samuel	Clemens
lion	Lion	Kimbro
kitty	Amber	Straub

$\pi_{\text{login,first}}$ (names)	
mark	Samuel
lion	Lion
kitty	Amber

Composition of operations



$\pi_{\text{login,first}}\ (\ \sigma_{\text{login=mark}}\ (\text{names})\ )$

names		
mark	Samuel	Clemens
lion	Lion	Kimbro
kitty	Amber	Straub

$\pi_{\text{login,first}}\ (\ \sigma_{\text{login=mark}}\ (\text{names})\ )$	
mark	Samuel

Union, Difference and Intersection

loans

name	loan
mark	L108
kitty	L224

accounts

name	account
mark	A32
lion	A90

π_{name} (loans) $\cup$ π_{name} (accounts)

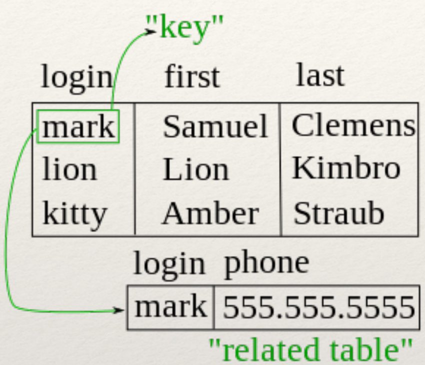
name
mark
lion
kitty

π_{name} (loans) $-$ π_{name} (accounts)

π_{name} (loans) $\cap$ π_{name} (accounts)

difference and intersection

Cartesian product

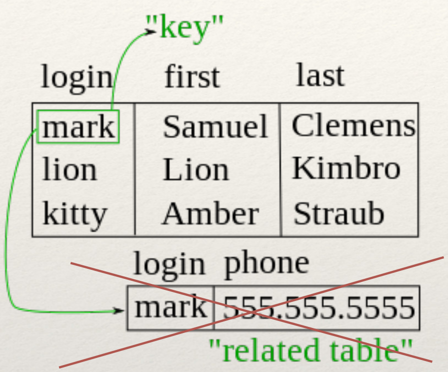


Also known as *cross product* and *cross join*

names x numbers

names × numbers				
mark	Samuel	Clemens	mark	555.555.5555
lion	Lion	Kimbro	mark	555.555.5555
kitty	Amber	Straub	mark	555.555.5555

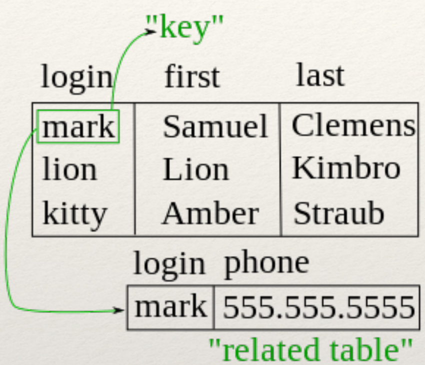
Cartesian product



mark	555.555.5555
kitty	555.666.6666

names × numbers				
mark	Samuel	Clemens	mark	555.555.5555
lion	Lion	Kimbro	mark	555.555.5555
kitty	Amber	Straub	mark	555.555.5555
mark	Samuel	Clemens	kitty	555.666.6666
lion	Lion	Kimbro	kitty	555.666.6666
kitty	Amber	Straub	kitty	555.666.6666

Join Operation



$$(\text{names} \bowtie \text{numbers}) \coloneqq \sigma_{\text{names.login} = \text{numbers.login}} (\text{names} \times \text{numbers})$$

names ⋈ numbers			
mark	Samuel	Clemens	555.555.5555

Product versus Join

- Q: why is the second faster even without a where clause
- A: because for each element from table A you only pick the elements from B with the same key, therefore O(n) versus O(n2)

Structured Query Language (SQL)

- Programming language
- Each statement is a declaration for a program
- You do not specify how to get something
- But you specify what you want and the SQL engine will generate the program
- Adding an index will change what kind of program is created
- Query optimizer picks the algorithm

Data Definition

```
CREATE TABLE people (name TEXT PRIMARY KEY,
                     address TEXT);

CREATE TABLE food (name TEXT,
                   food TEXT);

CREATE INDEX name ON food(name);
```

TABLE: people	
field:**name** type:TEXT	field:**address** type:TEXT

TABLE: food	
field:**name** type:TEXT	field:**food** type:TEXT

Data Manipulation

```
INSERT INTO people
  VALUES ('john', '1 Main Street, Springfield, MA');

INSERT INTO people
  VALUES ('jane', '1 High Street, Springfield, MA');

INSERT INTO food VALUES ('john', 'chocolate');
INSERT INTO food VALUES ('john', 'meatloaf');
INSERT INTO food VALUES ('jane', 'paella');
INSERT INTO food VALUES ('jane', 'chicken');
INSERT INTO food VALUES ('jane', 'pizza');
```

The SELECT Statement

```
SELECT * FROM people;

john|1 Main Street, Springfield, MA
jane|1 High Street, Springfield, MA
```

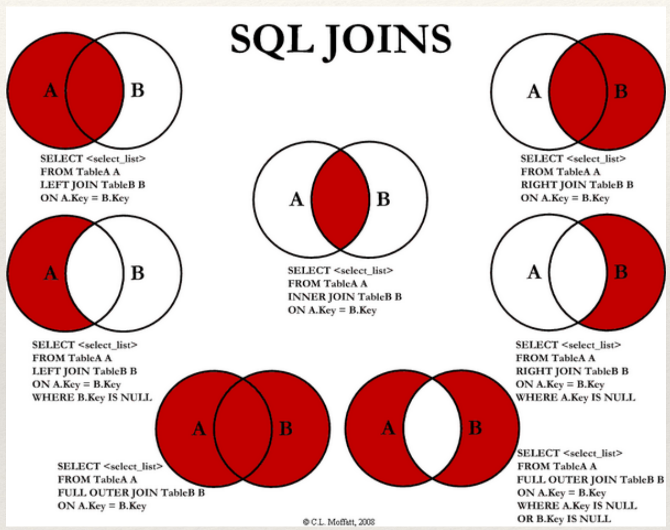
```
SELECT people.name, address, food
FROM people, food
WHERE people.name = food.name;

john|1 Main Street, Springfield, MA|pizza
john|1 Main Street, Springfield, MA|meatloaf
jane|1 High Street, Springfield, MA|paella
jane|1 High Street, Springfield, MA|chicken
jane|1 High Street, Springfield, MA|chocolate
```

inner join

The world of Joins

- Cross join
- **Inner join**
- Left join
- Right join
- Outer join
- Left outer join
- Right outer join
- Natural join
- Equi join



The world of Joins

A
1
2
3
4

B
3
4
5
6

Inner join

A	B
3	3
4	4

Left join

A	B
1	-
2	-

Right join

A	B
-	5
-	6

Left outer join

A	B
1	-
2	-
3	3
4	4

Right outer join

A	B
3	3
4	4
-	5
-	6

Full outer join

A	B
1	-
2	-
3	3
4	4
-	5
-	6

Inner versus Natural

A `NATURAL` join is just short syntax for a *specific* `INNER` join -- or "equi-join" -- and, once the syntax is unwrapped, both represent the same Relational Algebra operation. It's not a "different kind" of join, as with the case of `OUTER` (`LEFT` / `RIGHT`) or `CROSS` joins.

See the equi-join section on Wikipedia:

> A natural join offers a further specialization of equi-joins. **The join predicate arises implicitly by comparing all columns in both tables *that have the same column-names* in the joined tables.** The resulting joined table contains only one column for each pair of equally-named columns.
>
> Most experts agree *that NATURAL JOINs are dangerous and therefore strongly discourage their use.* The danger comes from inadvertently adding a new column, named the same as another column ...

https://stackoverflow.com/questions/8696383/difference-between-natural-join-and-inner-join

Statement to program

- SELECT * FROM people WHERE name=“jane";
- Program:

```
table = open(people)
answer = []
for record in table:
    if record.name = "jane":
        answer.append(record)
return answer
```

Note: this is not how it works in real database life,
its just an illustration

Statement to program

- SELECT * FROM people WHERE name="jane";
- But now we also have an index
- Program:

```
table = open(people)
answer = []
for record in table["jane"]:
    answer.append(record)
return answer
```

- Gets complicated with more complex queries

Some More Notions

- Transaction, commit, rollback
- Concurrency
- ACID
  - Atomicity: each transaction is all or nothing
  - Consistency: transactions result in valid state (schema and constraints are met)
  - Isolation: transactions are independent
  - Durability: once committed, stuff stays

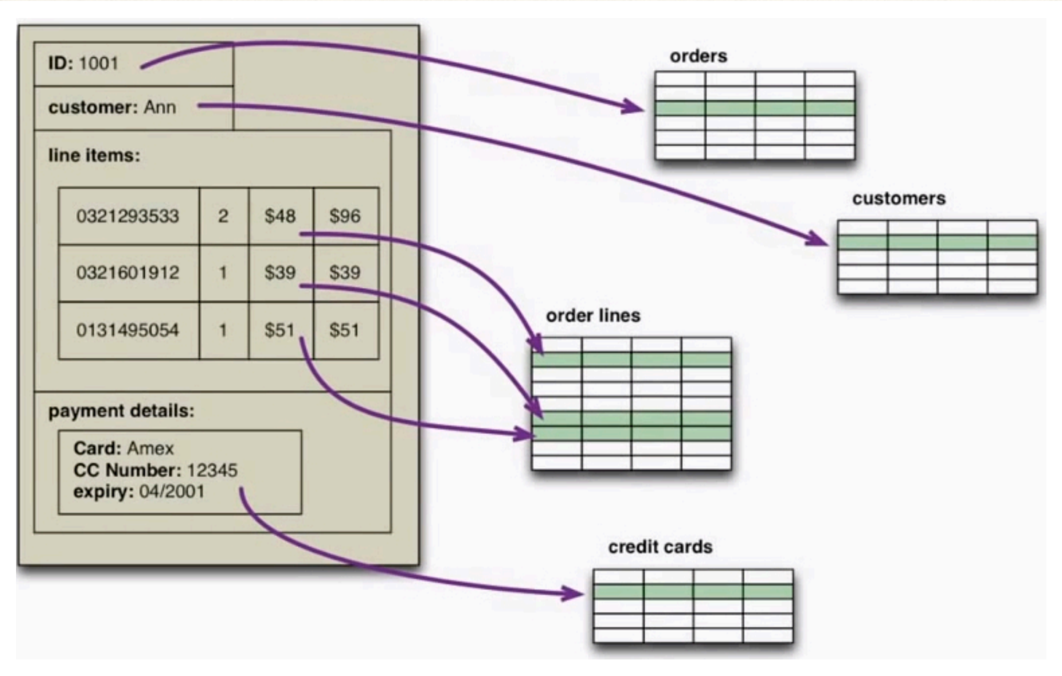
SQLite

- D. Richard Hipp
  - (D. for Dwayne, not Dr.)
- Standalone, small executable
- Available everywhere, public domain
- No configuration, lightweight
- Can serve as an interchange format
  - Easier than parsing a file
- http://sqlite.org/

Embedded SQLite

```
>>> import sqlite3
>>> connection = sqlite3.connect('test')
>>> c = conn.cursor()
>>> c.execute("CREATE TABLE people (name TEXT PRIMARY KEY, address TEXT);")
<sqlite3.Cursor object at 0x10fded8f0>
>>> c.execute("INSERT INTO people VALUES (?, ?)", ('john', 'here'))
<sqlite3.Cursor object at 0x10fded8f0>
>>> c.execute("INSERT INTO people VALUES (?, ?)", ('john', 'here'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
sqlite3.IntegrityError: column name is not unique
>>> c.execute("SELECT * FROM people")
<sqlite3.Cursor object at 0x10fded8f0>
>>> c.fetchall()
[(u'john', u'here')]
>>> c.fetchall()
[]
>>>
```

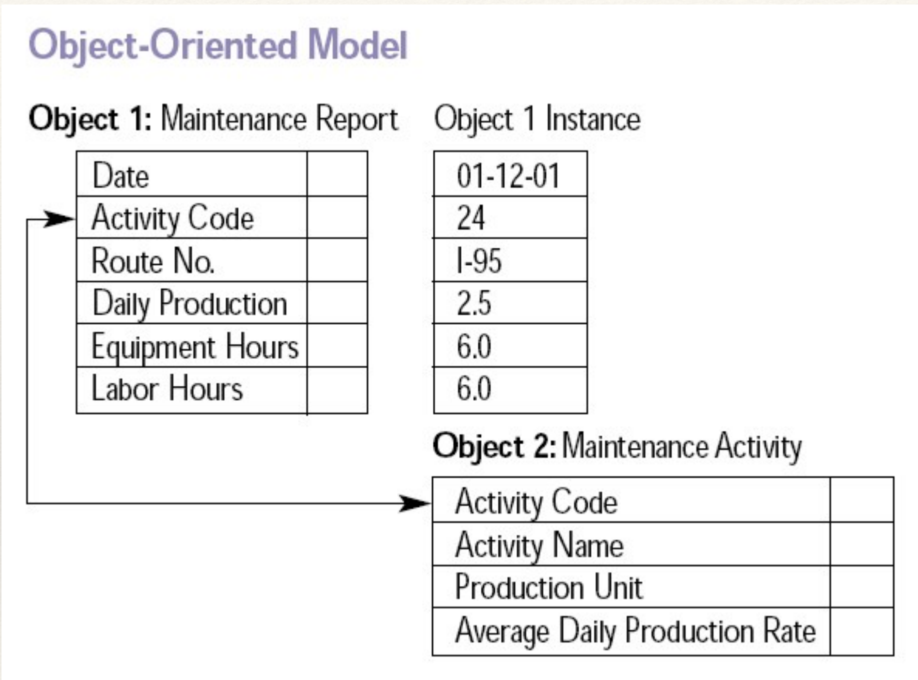
Object-Relational Impedance Mismatch



A natural mapping is much easier to achieve with a document or object database.

Screenshot from Martin Fowler's talk at the GOTO Conference, see https://www.youtube.com/watch?v=qI_g07C_Q5I

Object-Oriented Database



Uses pointers for relations between objects

Integrated with an object-oriented programming language

NoSQL

- Term was coined in 1998, now usually means “Not only SQL”
- Object-oriented databases in the eighties did not gain traction
  - relational databases were mature
  - tables often used for integration
- The internet and Big Data changed this
  - relational databases are not a natural fit to run on many small machines
  - Amazon and Google designed non-relational DBs

Challenges

- Data volume
  - Large datasets can become unwieldy when stored in relational databases. In particular, query execution times increase as the size of tables and the number of JOINs grow (aka JOIN pain).

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

- Data velocity
  - The rate at which data changes over time
    - Need to handle high levels of edits AND deal with surging peaks of database activity.
    - Relational databases cannot handle a sustained level of write loads (edits) and can crash during peak activity if not properly tuned (mind you, this is a claim from a NoSQL database provider).
  - The rate at which the data structure changes. In other words, it's not just about the rapid change of specific data points but also the rapid change of the data model itself.

https://neo4j.com/blog/why-nosql-databases/?ref=blog

Challenges

- Data variety
    - Data can be dense or sparse, connected or disconnected, regularly or irregularly structured.
- Data valence (connectedness)
    - The tendency of individual data to connect as well as the overall connectedness of datasets
    - Densely yet unevenly connected data is difficult to unpack and explore with traditional analytics (such as those based on RDBMS data stores)

https://neo4j.com/blog/why-nosql-databases/?ref=blog

Document Database

- Documents can be XML , but JSON has become standard
- Documents inside a document-oriented database are similar, in some ways, to records or rows in relational databases, but they are less rigid. No standard schema.

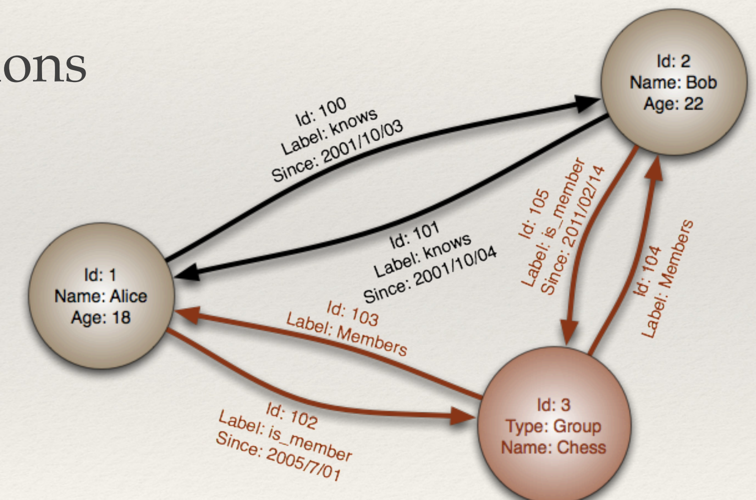
```
{
    FirstName: "Bob",
    Address: "5 Oak St.",
    Hobby: "sailing”
}
```

In MongoDB and ElasticSearch you can add either of these to your database, without ever defining schema

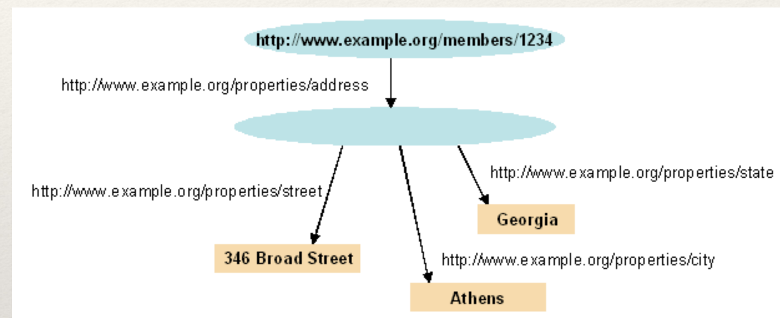
```
{
    FirstName: "Jonathan",
    Address: "15 Wanamassa Point Road",
    Children: [
         { Name: "Michael", Age: 10 },
         { Name: "Jennifer", Age: 8 },
         { Name: "Samantha", Age: 5 },
         { Name: "Elena", Age: 2 }
    ]
}
```

Graph Database

- Nodes
- Properties
- Relations



https://commons.wikimedia.org/w/index.php?curid=19279472

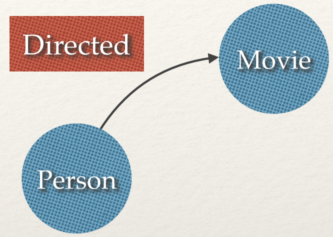


https://commons.wikimedia.org/w/index.php?curid=17096

Graph Database

Example Neo4j query:

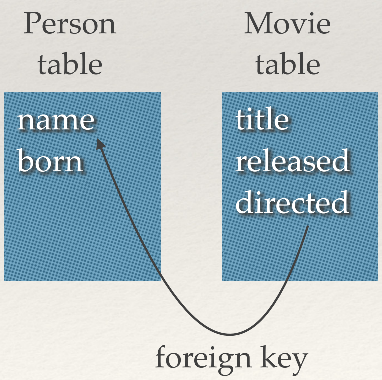
```
MATCH (tom:Person {name:'Tom Hanks'})-[rel:DIRECTED]-(movie:Movie)
RETURN tom.name, tom.born, movie.title, movie.released
```



Equivalent SQL query:

```
SELECT Person.name, Person.born, Movie.title, Movie.released
FROM Person, Movie
WHERE Person.name = 'Tom Hanks' AND Movie.directed = Person.name
```

This is assuming there are just two tables with the Movie table referring back to the Person table. In reality there is likely to be a table named Director that links to foreign keys in the Person and Movie tables. So the join will be more complex.



Databases with Flask

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SECRET_KEY'] = 'fc3bb2a43ff1103895a4ee315ee27740'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

class User(db.Model):

    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return '<User %r %r>' % (self.username, self.email)
```

Databases with Flask

- **SQLAlchemy** is a
  - Library that facilitates the communication between Python programs and databases.
  - Used as an Object Relational Mapper (ORM) tool that translates Python classes to tables on relational databases and automatically converts function calls to SQL statements.
- Flask SQLAlchemy
  - https://flask-sqlalchemy.palletsprojects.com/en/2.x/