

Advanced Data Management

Academic year 2023/2024

9 CFU / 6 CFU

“You can have data without information, but you cannot have information without data.”

(Daniel Keys Moran)

Who, when, and where?

Who is involved?

- Barbara Catania



- Giovanna Guerrini



- Ziad Janpih
(help during the labs and the project)

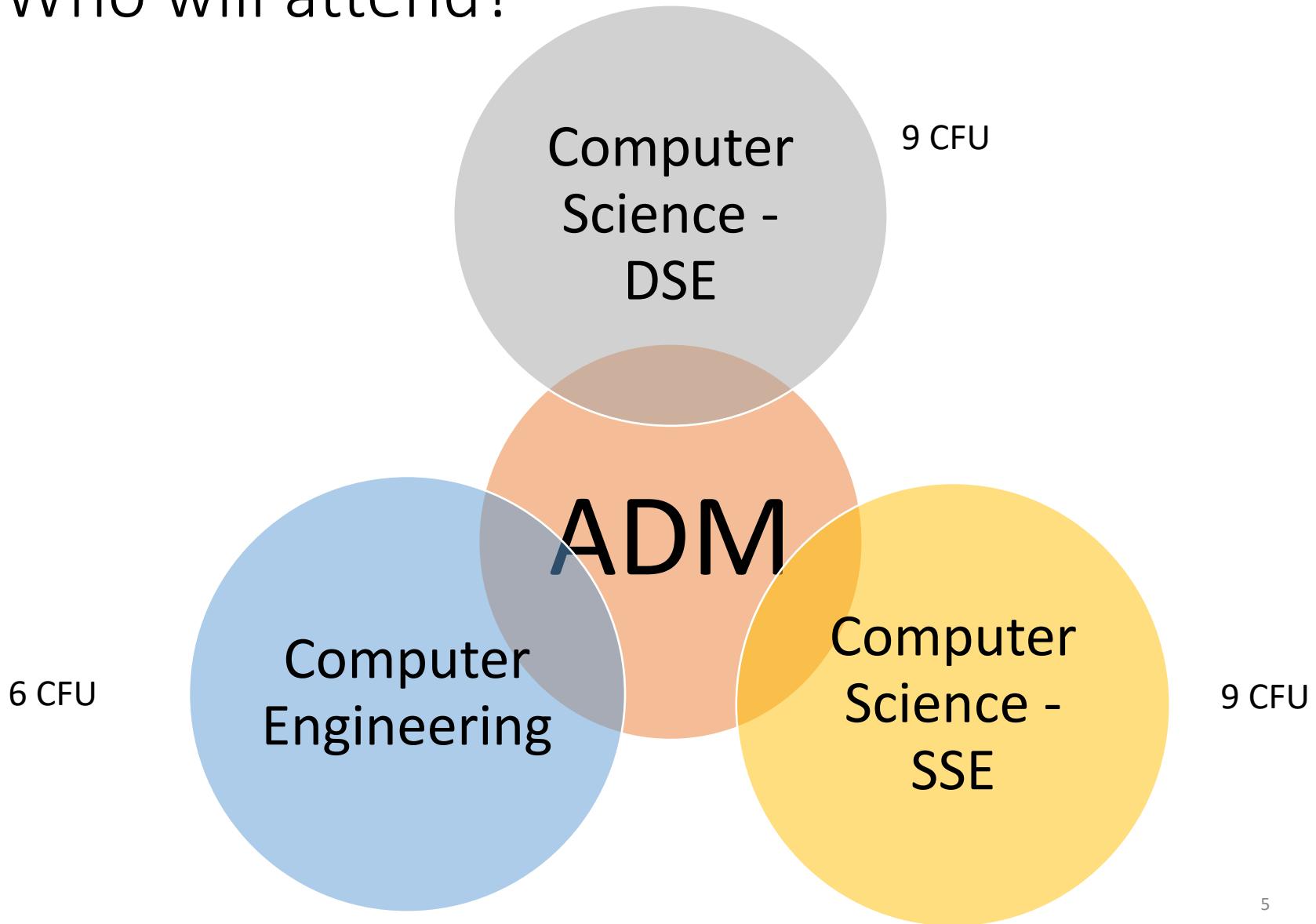
When and where will it take place?

- Monday, 11.00-13.00
[room 711, Valle Puggia]
- Wednesday, 11.00-13.00
[room 710, Valle Puggia]
- Thursday, 14.00-16.00
[room 711, Valle Puggia]

In presence, no streaming

Recording of lectures proposed in previous a. y. available on Aulaweb

Who will attend?

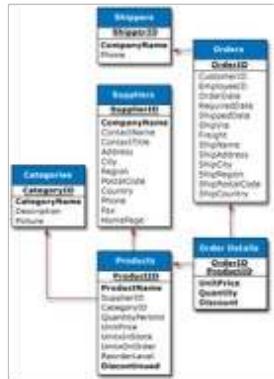


Why this course?



Data Management in the Big-Data era

Why this course?



Transactional data
(standard data, internal
to one organization)



Scientific instruments
(collecting all sorts of data)



Social media and networks
(all of us are generating data)



Mobile devices
(tracking all objects all the time)



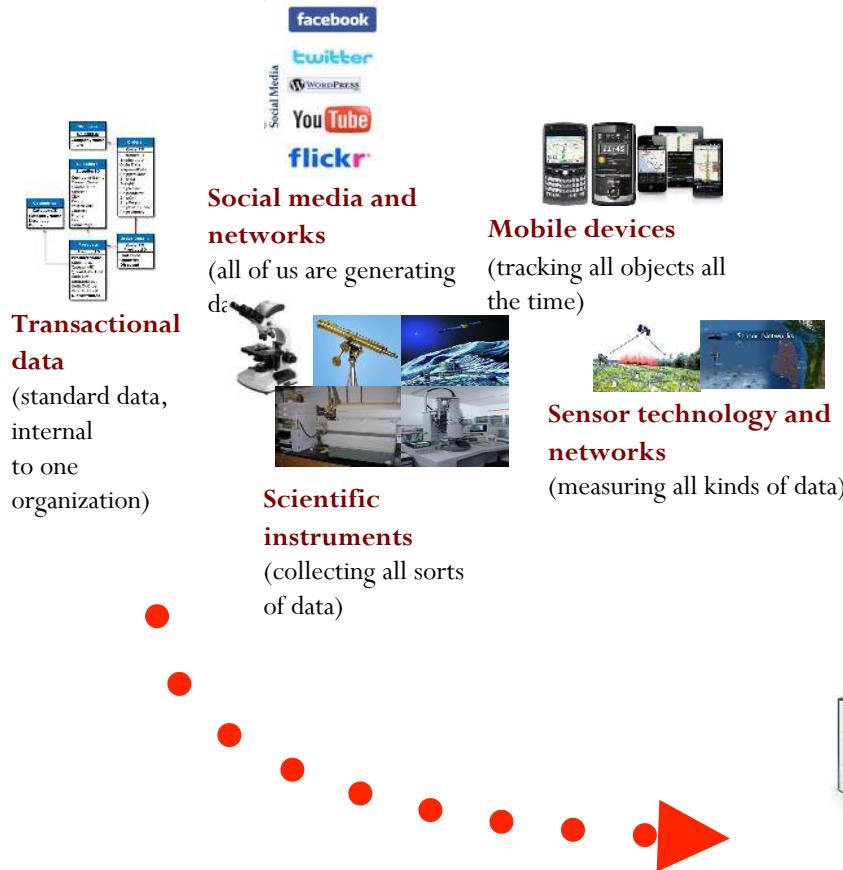
Sensor technology and networks
(measuring all kinds of data)



Why this course?

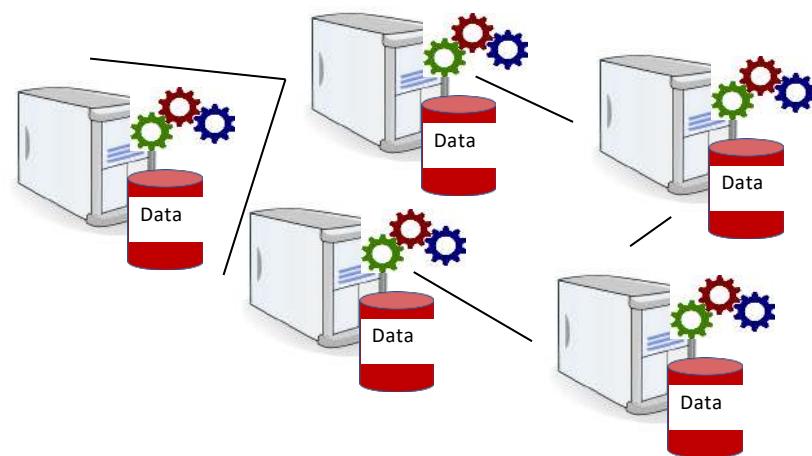
- big datasets (**Volume**)
- heterogeneous, often incomplete, and highly interconnected data (**Variety** and **Veracity**)
- data can be generated at a very high rate (very fast - **Velocity**)
- the volume of such data requires programming environments which exploit **parallelism** in order to cope with such huge volumes in an efficient way
- many previously unknown information can be extracted from them (high **Value**)

Why this course?

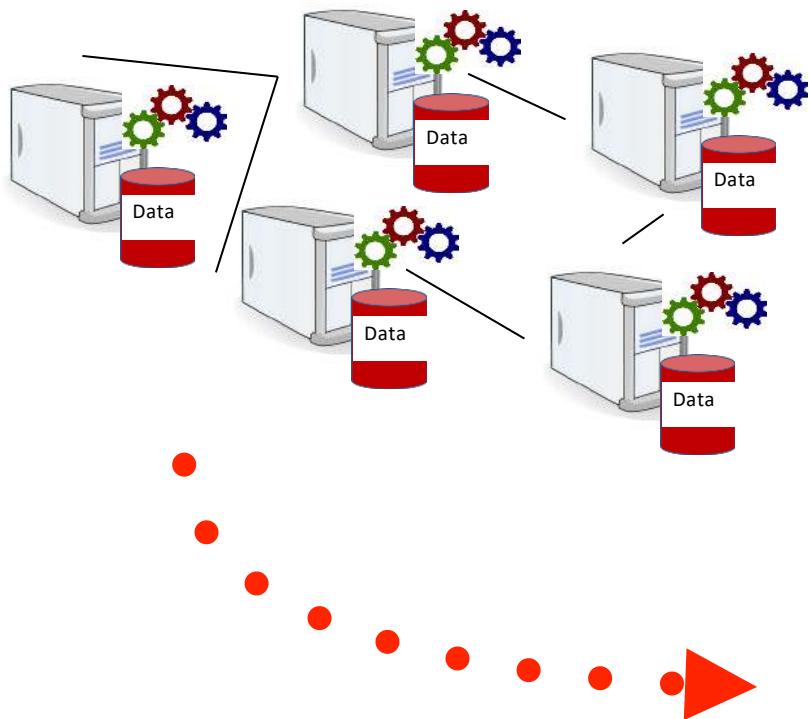


Data Management

- collect data
- integrate data
- clean data
- represent and store data
- query & process data



Why this course?



Data Analysis

- analyse your data
- interpret the obtained results
- take your decisions



Why this course?

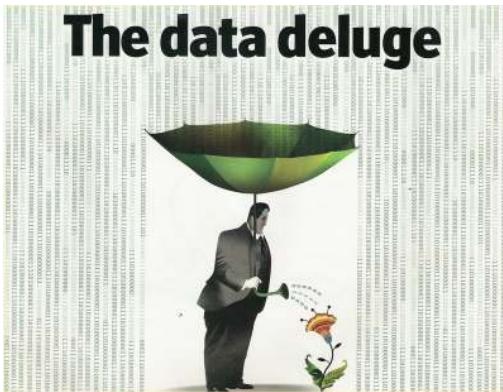
the way to the «value», through data analysis, could be very dangerous, be careful!

- performance,
performance,
performance!
- effectiveness
- heterogeneity
- flexibility

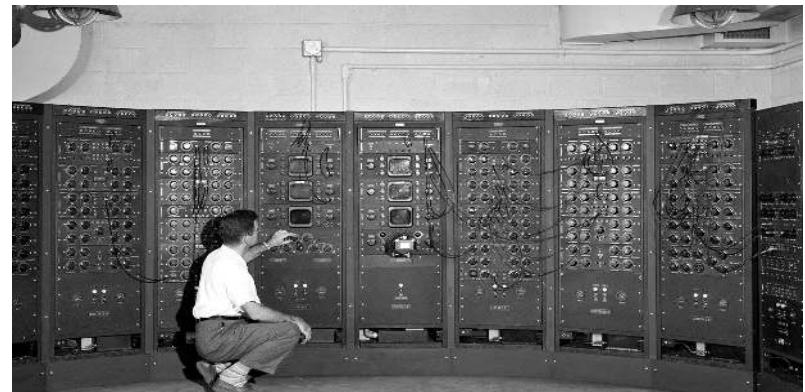


Why this course?

big data



big machines / big architectures



data management

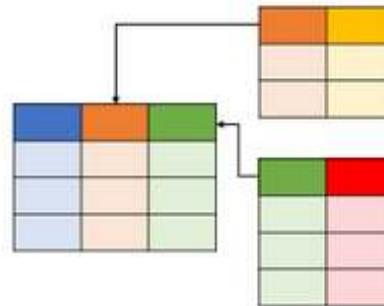
Why this course?

- unfortunately, traditional database management techniques do not scale to such huge datasets and do not effectively take into account issues raised by large-scale environments
- new solutions have therefore been devised

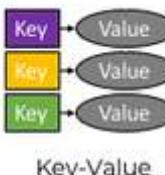
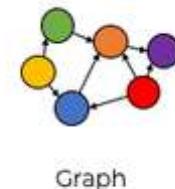
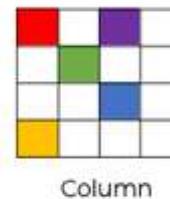
Why this course?

- high volume
- flexible data structure
- strong connection of data and applications
- moving away from using databases as integration points towards encapsulating databases within applications
- systems for large-scale data management, NoSQL systems

SQL DATABASES



NoSQL DATABASES



Key-Value



Document

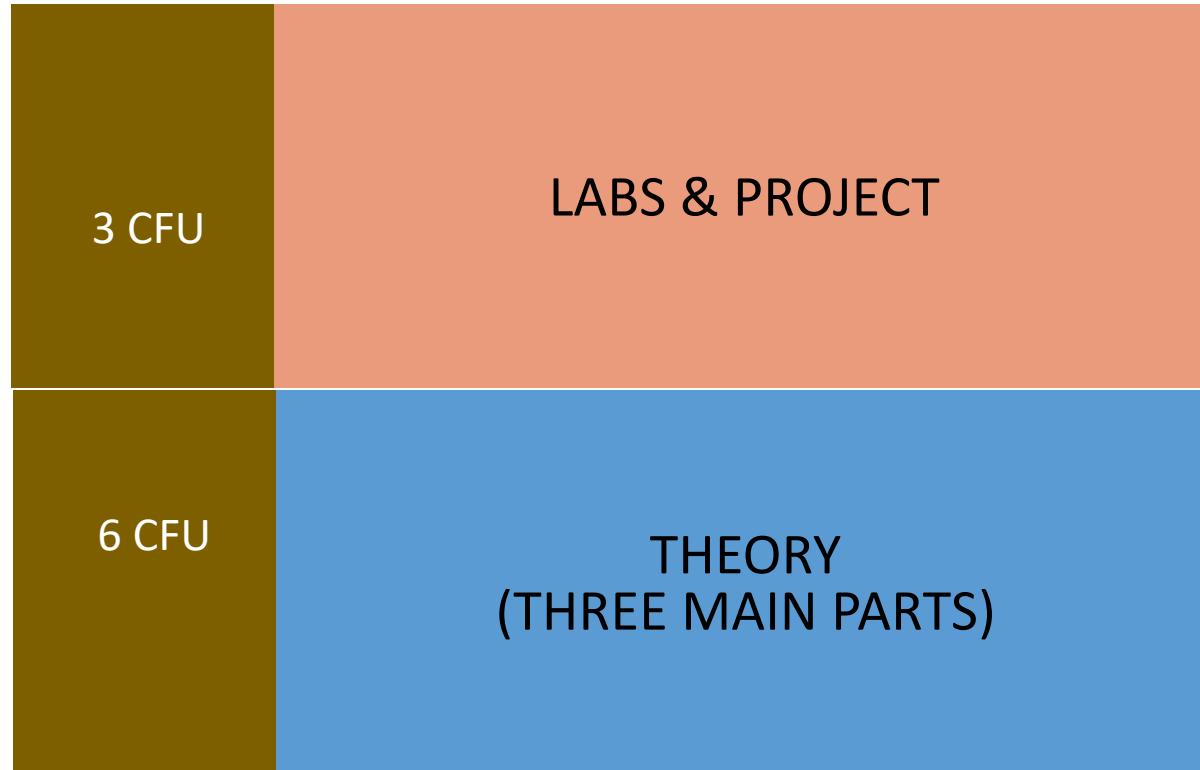
Why this course?

- data exchanged between systems and applications
- heterogeneous, possibly unknown and mostly uncontrolled sources
- need to agree on the data meaning in order to avoid dangerous misunderstandings
- semantic elicitation and the availability of appropriate semantic metadata are the key to the meaningful use of information in modern distributed environments



What will you learn?

What will you learn?



What will you learn?

REFERENCE
PART

THEORY

LABS

What will you learn?

PART I
Only 6 CFU

Recap on large scale distributed architectures
and data-intensive computing

REFERENCE
PART

THEORY

LABS

What will you learn?

Learning outcome - part I

- **DESCRIBE** the principles for data management in distributed systems, environments for large-scale data processing, systems for large-scale data management
- **UNDERSTAND** the differences between traditional data processing and management and large-scale (semantic) data processing and management

What will you learn?

| | | |
|-----------------------------|--|------------------------------------|
| PART II | Systems for large-scale data management (NoSQL systems) | Riak, Cassandra, MongoDB, Neo4J |
| PART I Only 6 CFU | Recap on large scale distributed architectures and data-intensive computing | |

REFERENCE
PART

THEORY

LABS

What will you learn?

| | | |
|-------------------------------|--|------------------------------------|
| PART III Only 9 CFU | Semantic data management | RDF, SPARQL, OWL |
| PART II | Systems for large-scale data management (NoSQL systems) | Riak, Cassandra, MongoDB, Neo4J |
| PART I Only 6 CFU | Recap on large scale distributed architectures and data-intensive computing | |

REFERENCE
PART

THEORY

LABS

What will you learn?

Learning outcome - parts II & III

- **UNDERSTAND** the differences between the presented approaches for large-scale (semantic) data management
- **SELECT** the system and the methodology for large-scale (semantic) data management, suitable in a given application context
- **USE** some of the presented systems for large-scale (semantic) data management, for solving simple problems
- **USE** at least one of the presented systems for large-scale (semantic) data management for solving non-trivial problems
- **ANSWER** questions related to large-scale (semantic) data management
- **SOLVE** exercises related to the data design in some of the presented systems and the interaction with such systems, through the available languages

Prerequisites

Prerequisites

- basics on large-scale distributed systems and computing
 - *DSE-Computer Science students: Distributed Computing, I year*
 - *all the other students: in PART I of the course*
 - *partial overlap with Distributed Computing for SSE-Computer Science students*
- solid foundation in database design and querying (see *AulaWeb for references*)
 - **you must pass the test for taking the exam**
 - *first test on Monday, October 2*
 - *before any exam date*

How is the course organized?

Resources (see Aulaweb)

- books
 - manuals
 - scientific papers
 - software links
-
- slides
 - recording of lectures proposed in previous a.y.

Lectures and labs

- Lectures (in presence)
 - on the main theoretical and methodological issues
 - exercises on the main theoretical and methodological issues
 - talks from prominent experts in the field
- labs on the main technologies presented in the course
 - each lab = one groupwork assignment (2 persons each)
 - we will communicate soon the proposed organization
- quiz (online) on the main concepts proposed in the course
- bonus based on labs and quiz
 - no oral exam (see later)
 - up to 2 points bonus

Project (for both 9 CFU and 6 CFU)

- groupwork (up to 2 persons)
- two options
 - A. design and development
 - B. research-based
 - *only for students that pass the assignments proposed during the course with a grade higher than a given threshold (more information later)*

Project (for both 9 CFU and 6 CFU)

A - Design & development project (both 9 CFU and 6 CFU)

- choose one application domain
- write a requirement analysis document
- choose one technology, among those studied
- deliver a document explaining your choices
- design and develop your solution
- deliver it as soon as you want to take the exam

B- Research-based project (both 9 CFU and 6 CFU)

only for students that pass the prerequisite test and the assignments proposed during the course with a grade higher than a given threshold (more information later)

- select one topic among a set of available ones
- review the literature on the selected topic
- prepare a report summarizing the investigated topic
- deliver it as soon as you want to take the exam

Exam modalities

- written exam (questions/exercises)
- project delivery + video presentation
- oral exam [only for those that do not delivered the assignments]
 - theoretical questions and / or practices of the course topics
 - up to 2 points
- the **project** has to be submitted **before** the **written exam**
- the **oral exam** [if needed] will be scheduled **after** the **written exam**



**First of all enroll to the
course on Aulaweb!**

Introduction to Big Data and Large Scale Architectures

What kind of data?

Classical data management applications



GWSS - Editech - Programma di Gestione delle Attività Meccaniche (programma basato)

Da: ROMA - ROMA - ROMA

Al: 01/01/2013 - 31/12/2013

Registrazione: 01/01/2013 - 31/12/2013

Portata: 01/01/2013 - 31/12/2013

Rag. sociale: DOLMUS SRL

Nominativo: ROMA/2013/000

CAP: 00041 Città: ROMA Provv: 100

Necessario:

Tipo: servizio

Da: 01/01/2013 - 31/12/2013

Nr. ord. via stampante: P.A.

Dati Contatti: Contatti | Ordini di accounto | DDE di entrata

Condizioni:

Categoria: MPOA | Mentre Dopo: ALTA TOOL S.

Punto: A | PORTO ACCIAIATO

Viale: 104 | 00140

Cond. pag.: | P. D'AGOSTINA - GRG 7 M

Residenza:

ABN: 07/025 | DAV PAOLO - IMI SPA

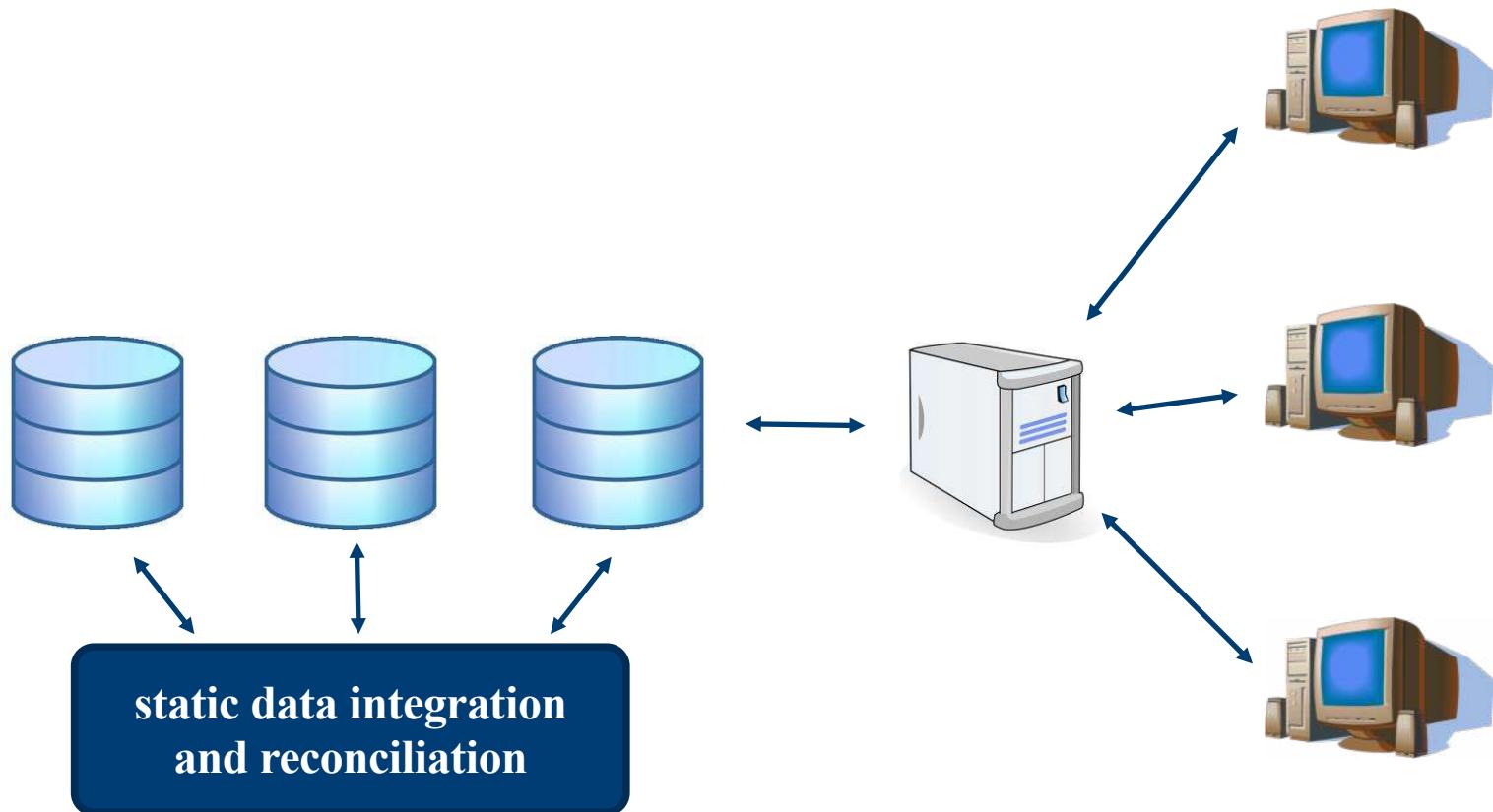
CAD: 07/001 | AG 3

QDG: 07/005 | VIA PERETTI 13R

B. Emissione: 00/00/0000

Stampa: 00/00/0000

Reference architecture



Data models

- Relational model: set of tuples (relations) sharing relationships

- Tu
sa

Small enough data
Well-defined structure
A new kind of software:

- Re
markt
- DataBase Management System (DBMS)**

| Employee Table | | | |
|----------------|------------|---------|----------|
| Last Name | First Name | Empl ID | Salary |
| Jones | Jeff | 0001 | 30000.00 |
| | | | 40000.00 |

| Location |
|----------|
| York |

| Dept ID | Empl ID |
|---------|---------|
| 0010 | 0001 |
| 0010 | 0002 |

Typical interaction

- Declarative standard language - SQL
- Operational data retrieval operations

| Employee Table | | | |
|----------------|------------|---------|----------|
| Last Name | First Name | Empl ID | Salary |
| Jones | Jeff | 0001 | 30000.00 |
| Smith | Jane | 0002 | 40000.00 |

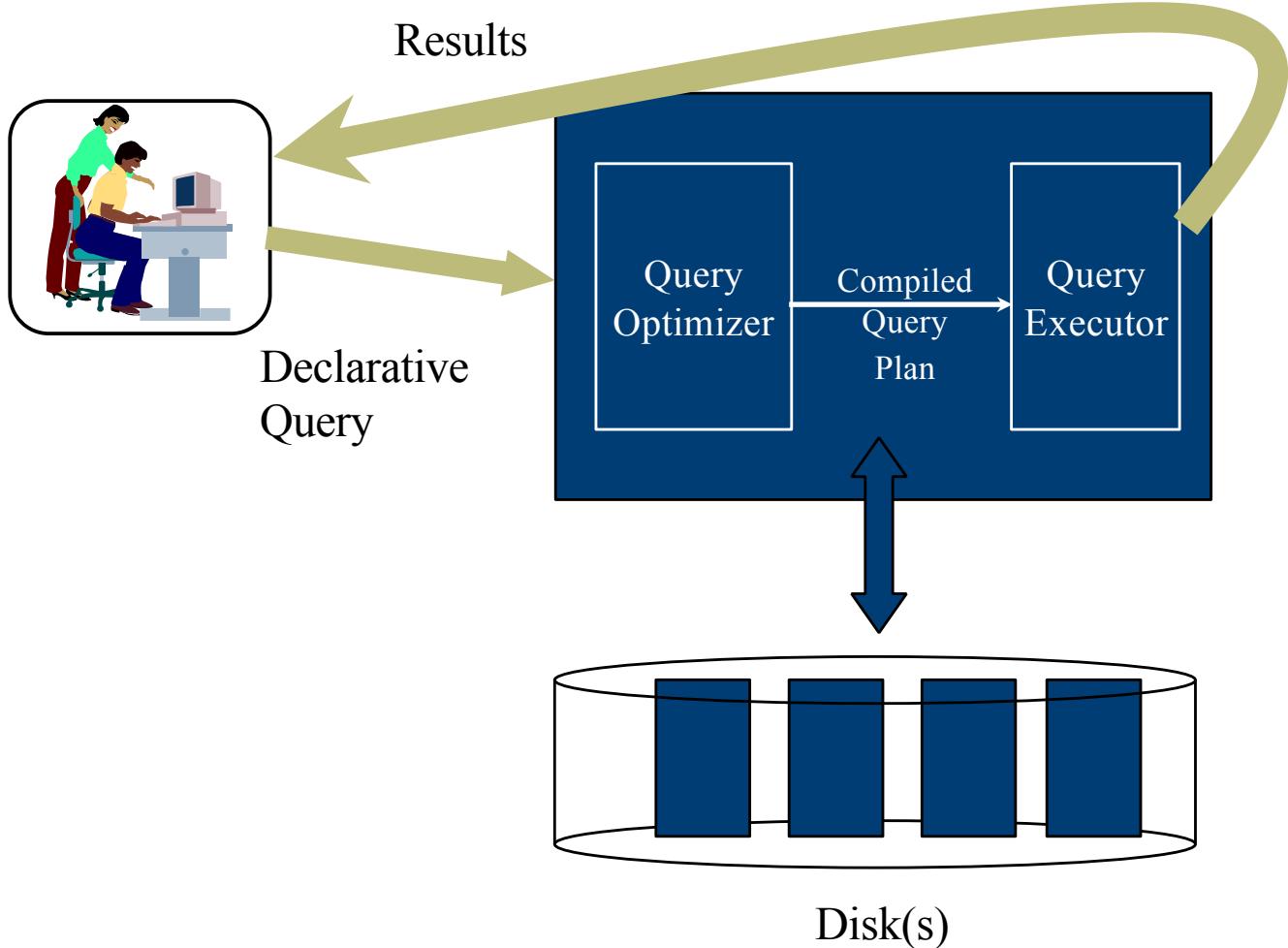
| Department Table | | |
|------------------|-------|----------|
| Dept ID | Name | Location |
| 0010 | Sales | New York |

| Dept_Empl Table | |
|-----------------|---------|
| Dept ID | Empl ID |
| 0010 | 0001 |
| 0010 | 0002 |

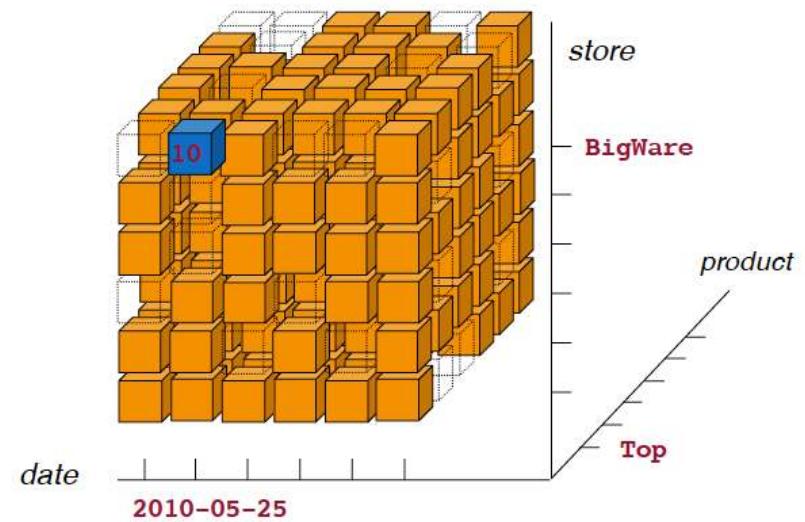
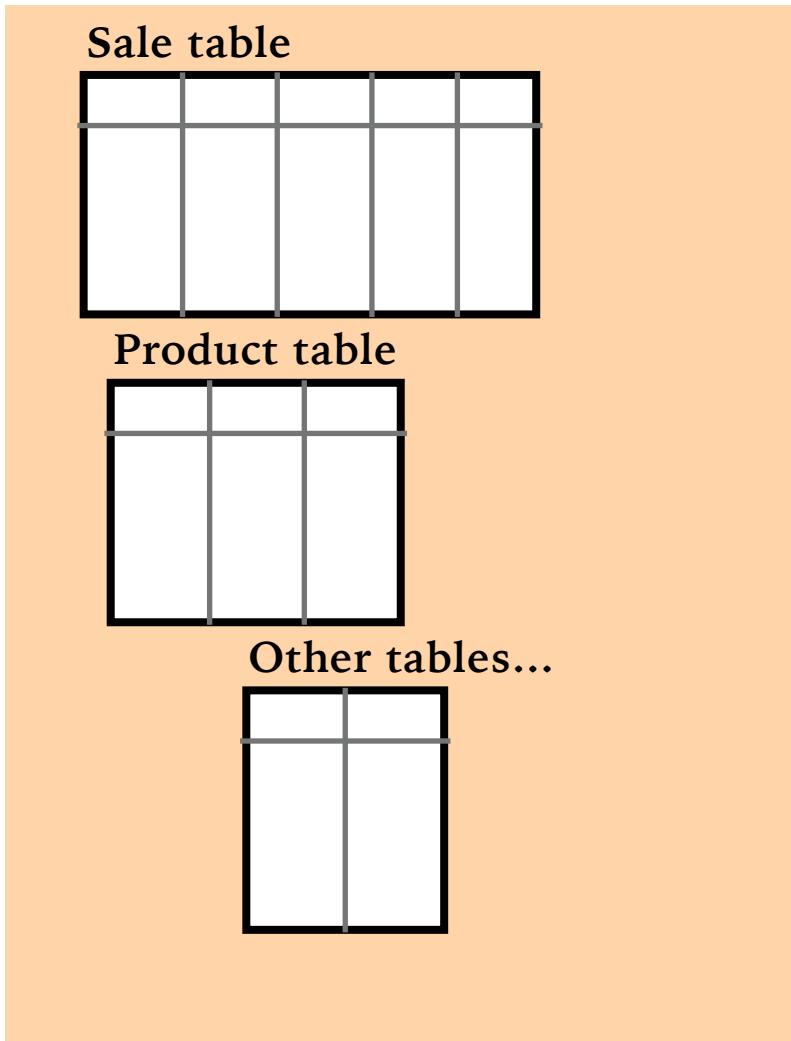
- Selection
 - `SELECT * FROM Employee WHERE salary > 3000`
- Join
 - `SELECT * FROM Employee NATURAL JOIN Dept_Empl NATURAL JOIN Department WHERE Location = 'New York'`

Data management

- efficiency
- integrity
- concurrency
- reliability
- security



Database and data warehouse



transactional vs analytical
models and systems

From data management to big data management

“Big data is a collection of data sets so **large** and **complex** that it becomes **difficult to process using on-hand database management **tools**** or traditional data processing **applications**”



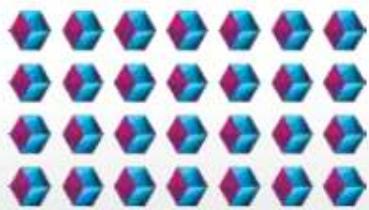
When data become “big”?



IOPS: Input/Output Operations Per Second

The 4 V's of big data

Volume



Data at Scale

Terabytes to petabytes of data

Variety



Data in Many Forms

Structured, unstructured, text, multimedia

Velocity



Data in Motion

Analysis of streaming data to enable decisions within fractions of a second.

Veracity



Data Uncertainty

Managing the reliability and predictability of inherently imprecise data types.

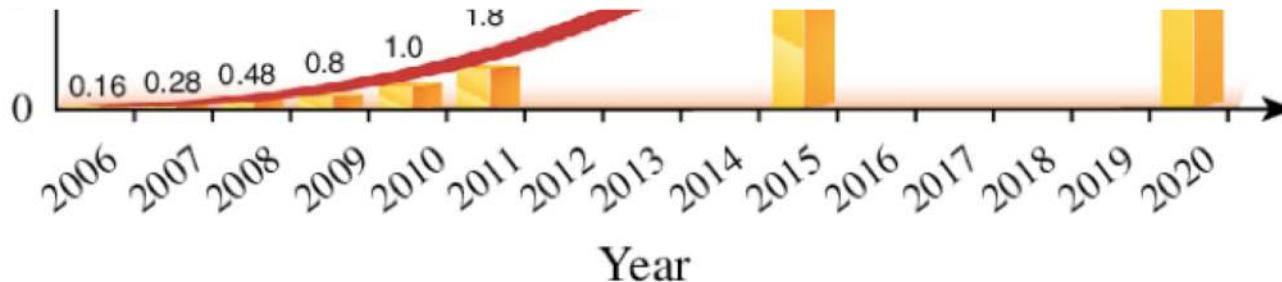
Volume

40
↑

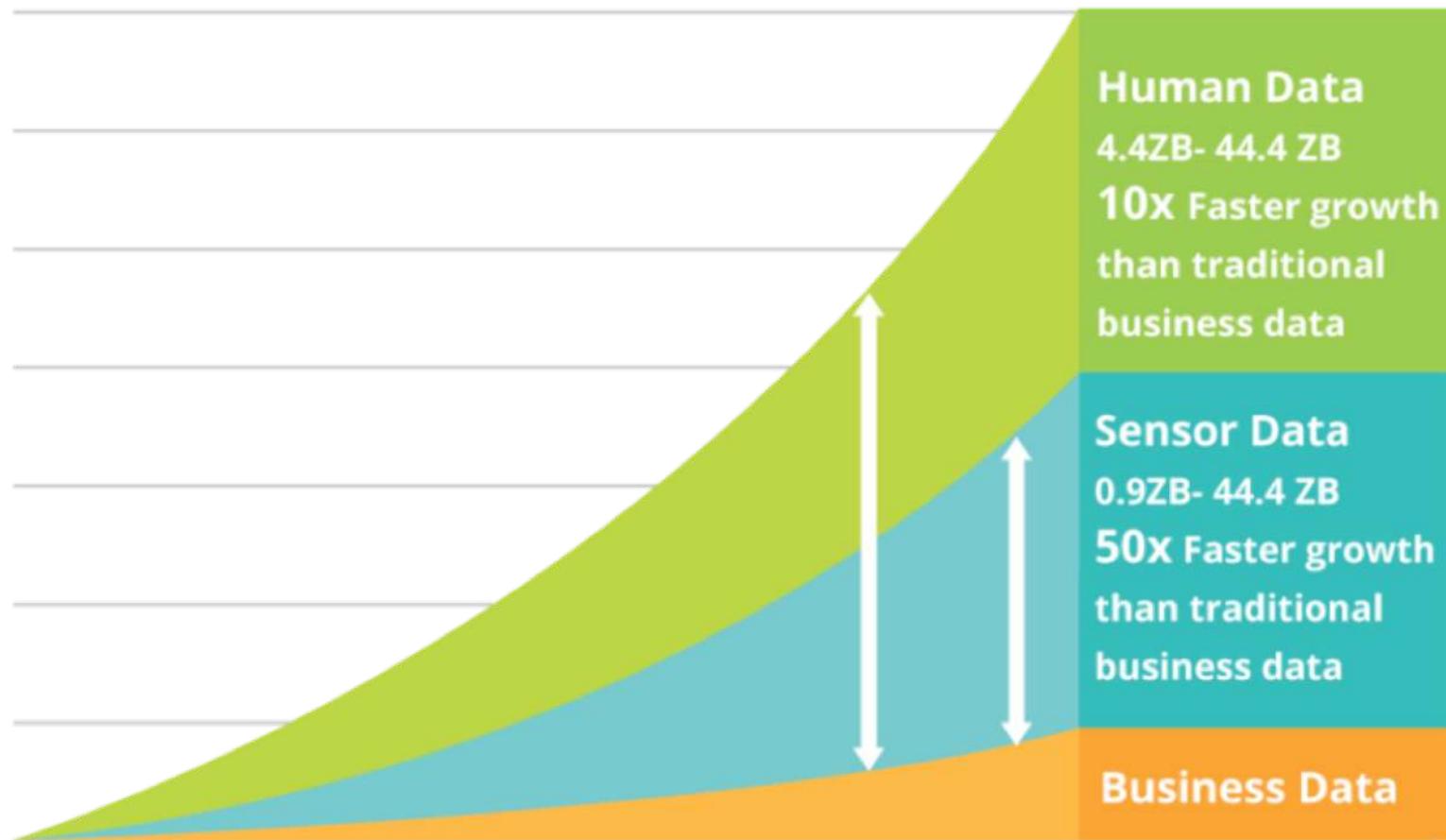


MULTIPLI DEL BYTE

| Nome | Simbolo | Multiplo | byte |
|-----------|---------|-----------|-----------------------------------|
| Kilobyte | kB | 10^3 | 1.000 |
| Megabyte | MB | 10^6 | 1.000.000 |
| Gigabyte | GB | 10^9 | 1.000.000.000 |
| Terabyte | TB | 10^{12} | 1.000.000.000.000 |
| Petabyte | PB | 10^{15} | 1.000.000.000.000.000 |
| Exabyte | EB | 10^{18} | 1.000.000.000.000.000.000 |
| Zettabyte | ZB | 10^{21} | 1.000.000.000.000.000.000.000 |
| Yottabyte | YB | 10^{24} | 1.000.000.000.000.000.000.000.000 |

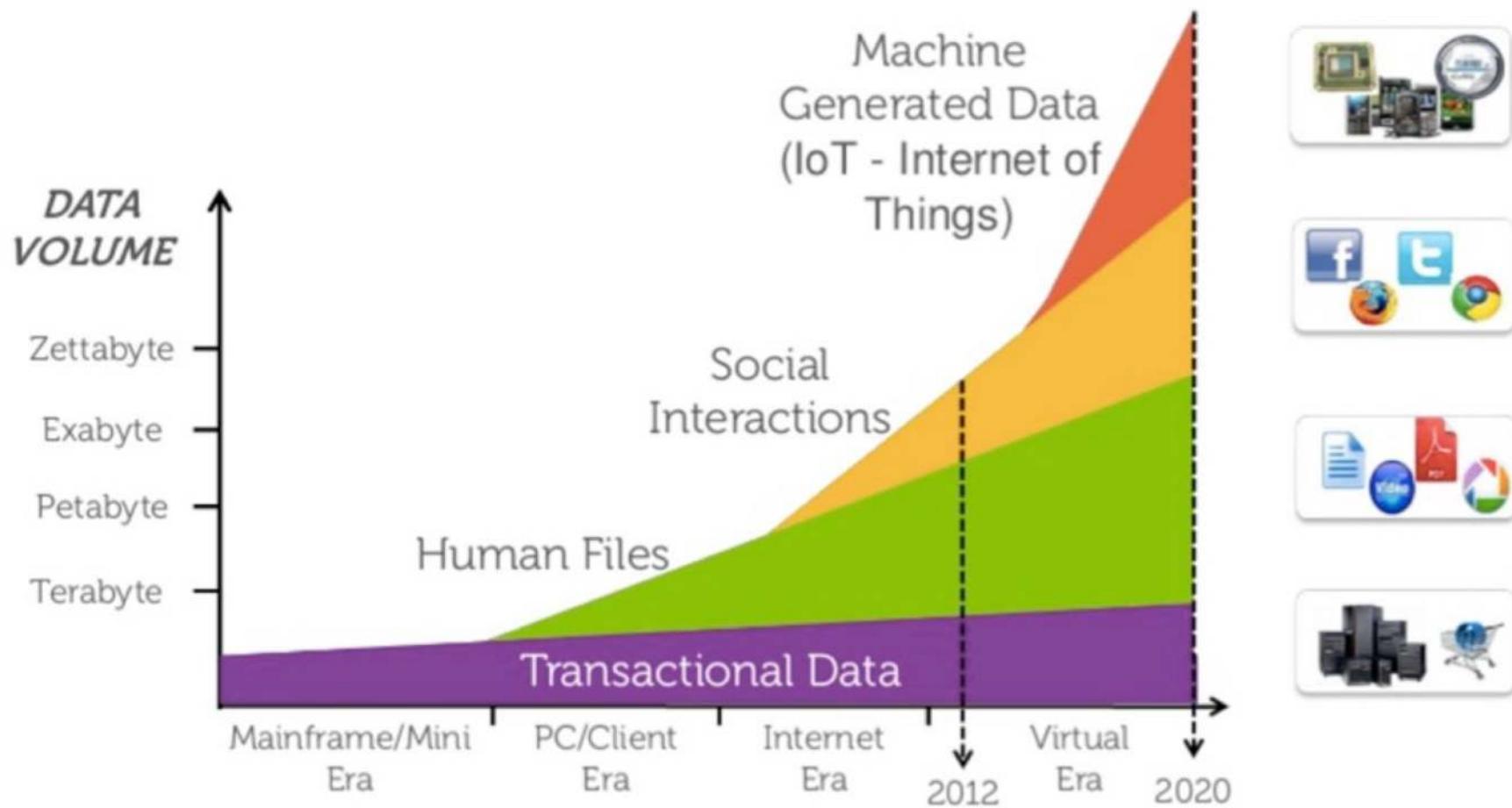


Volume



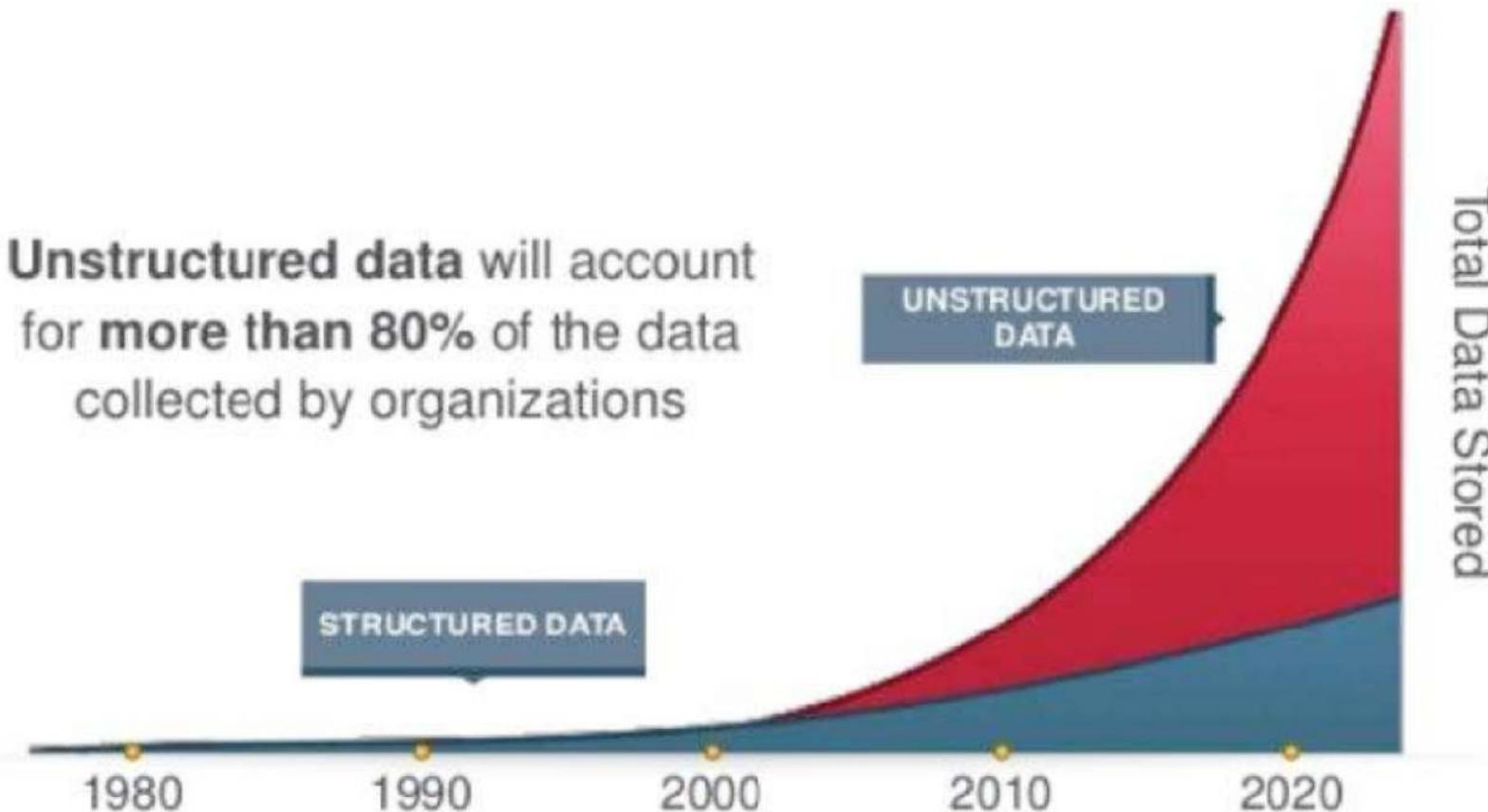
Source: Inside big data

Variety



Variety

Unstructured data will account for more than 80% of the data collected by organizations



Velocity

www.internetlivestats.com/



3,415,083,744

Internet Users in the world



1,054,631,097

Total number of Websites



143,829,458,192

Emails sent **today**



3,050,391,477

Google searches **today**



2,834,430

Blog posts written **today**



401,530,759

Tweets sent **today**



7,008,753,855

Videos viewed **today**
on YouTube



40,387,506

Photos uploaded **today**
on Instagram



62,813,292

Tumblr posts **today**



1,697,484,297

Facebook active users



456,283,065

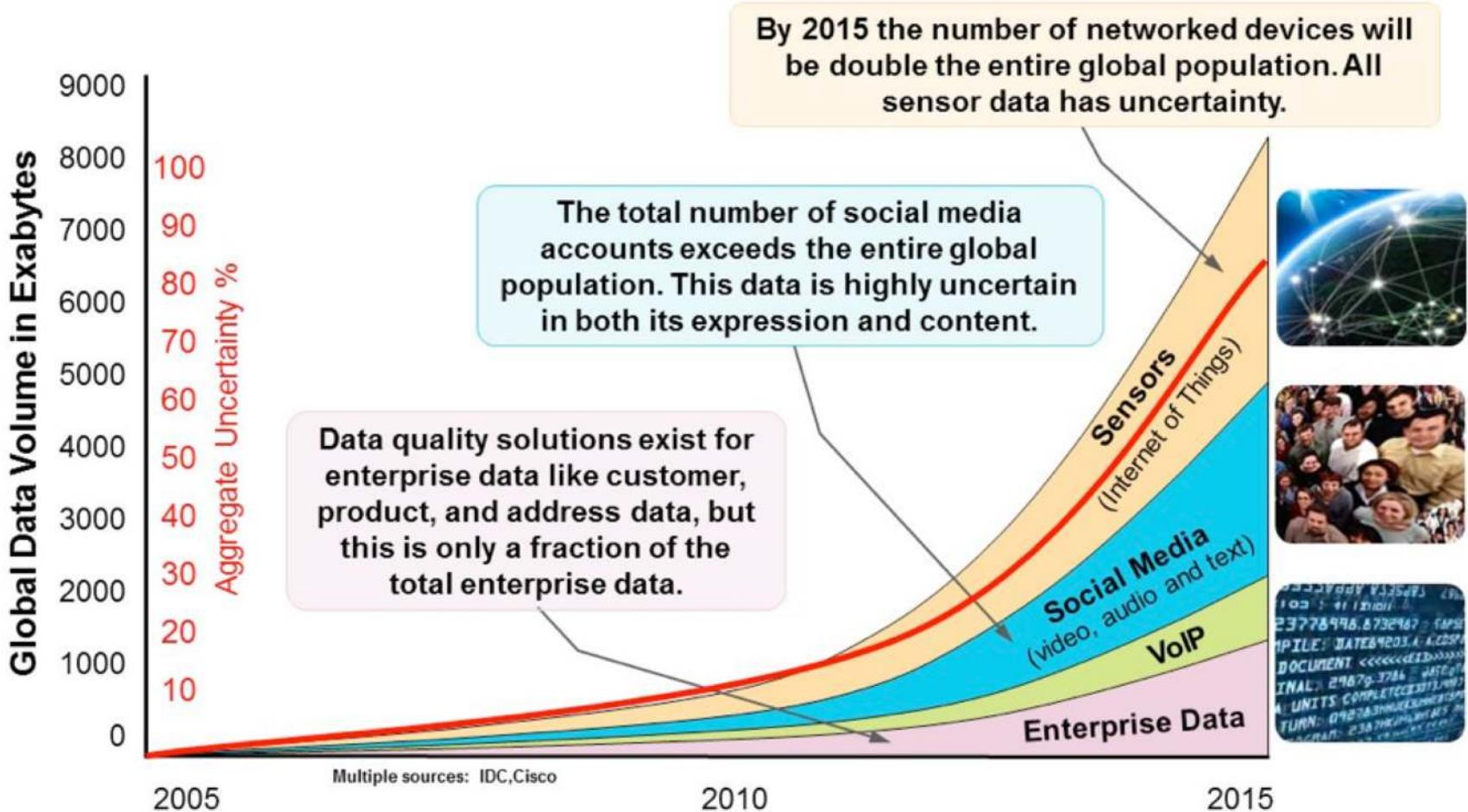
Google+ active users



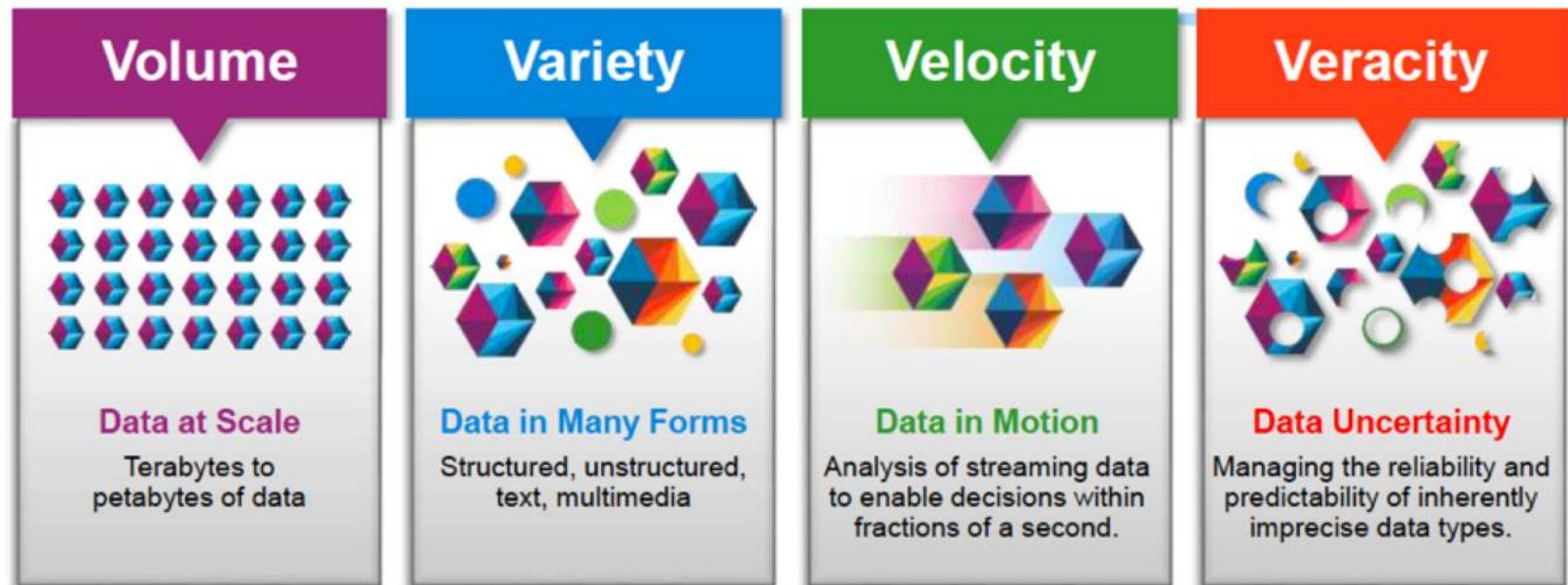
305,767,222

Twitter active users

Veracity



From 4 to 5 V's



Data Management → *Data Analysis*

Value

The real need

... need special hardware,
algorithms, tools to work at
this scale

... need **distribution** at this
scale

Distributed systems in short

- A **distributed system** is an application that coordinates the actions of several computers to achieve a specific task
- This coordination is achieved by exchanging **messages** which are pieces of data that convey some information
- The system relies on a network that connects the computers and handles the routing of messages

Distributed systems in short

- We distinguish between
 - **server node**: it provides a service of the distributed system
 - **client node**: it consumes this service
- Nothing prevents a client node to run on the same computer than a server node (this is typically the case in P2P networks)
- Both Client (nodes) and Server (nodes) are communicating software components: we assimilate them with the machines they run on

Distributed data system

Any distributed system that provide commonly needed functionalities for storing, accessing, and processing data, *by distributing **data, load** and, sometimes, **control** over the network*

Distributed data systems are standard building blocks for developing data-intensive applications

Compute- vs Data-intensive computing/applications

- **Compute-intensive**
 - computing applications which devote most of their execution time to computational requirements
 - CPU cycles are the bottleneck
- **Data-intensive**
 - computing applications which require large volumes of data and devote most of their processing time to I/O and data manipulation
 - The quantity and the complexity of data as well as the speed at which they are changing are the bottleneck

Data intensive application

- The development of data-intensive applications rely on distributed data systems
- Such distributed data systems are large-scale
 - High volume of data
 - High number of nodes

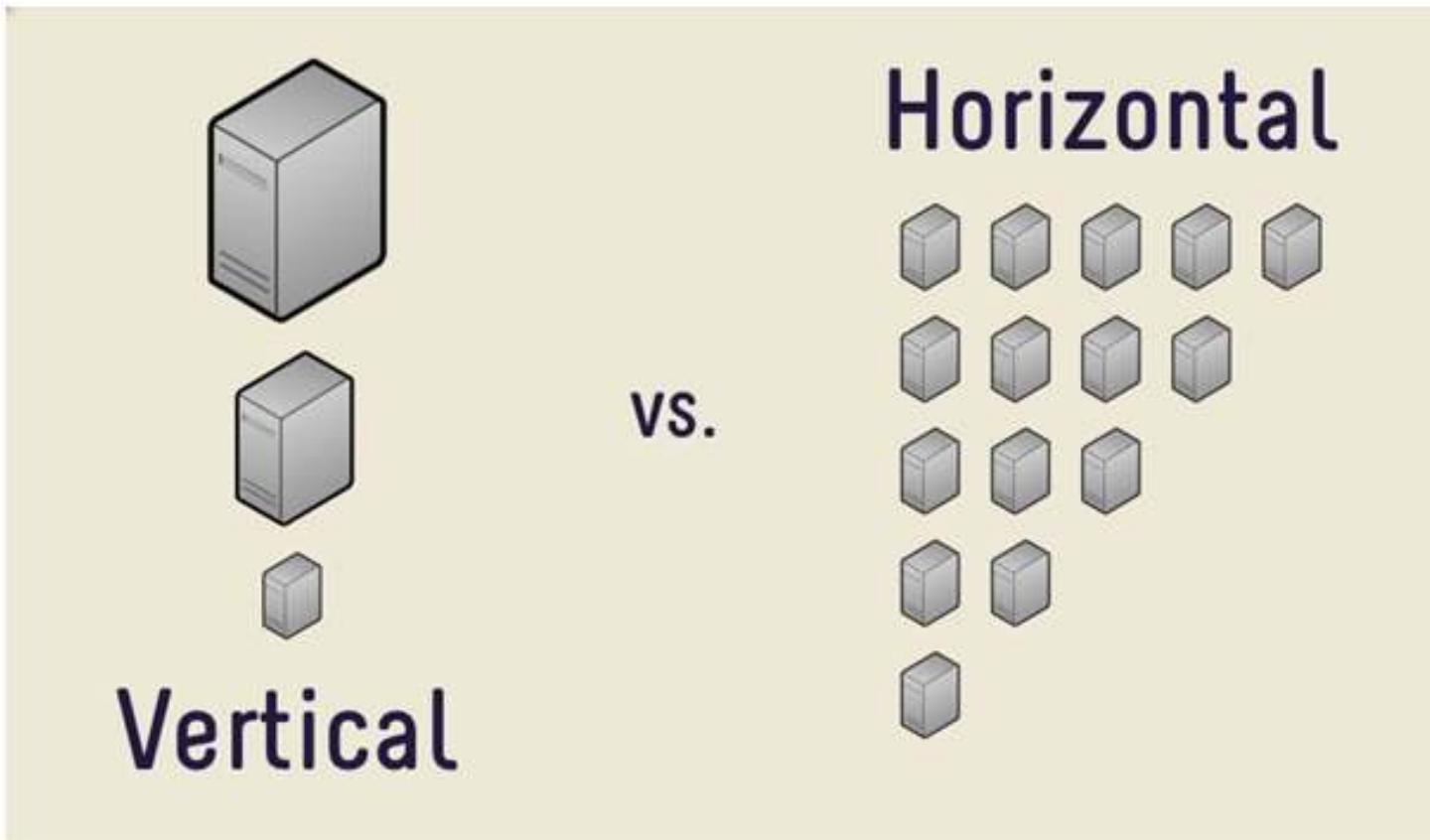
Large Scale Architectures

For Computer Science – Data Science & Engineering students:
this part has been taken from the material of the «Large Scale Computing» course

Computing at scale

- Big data applications require huge amounts of processing and data
 - Measured in petabytes, millions of users, billions of objects
 - Need special hardware, algorithms, tools to work at this scale
- **Scaling** is an issue

Vertical scaling vs horizontal scaling



What if one computer is not enough?

Buy a bigger (server-class) computer –

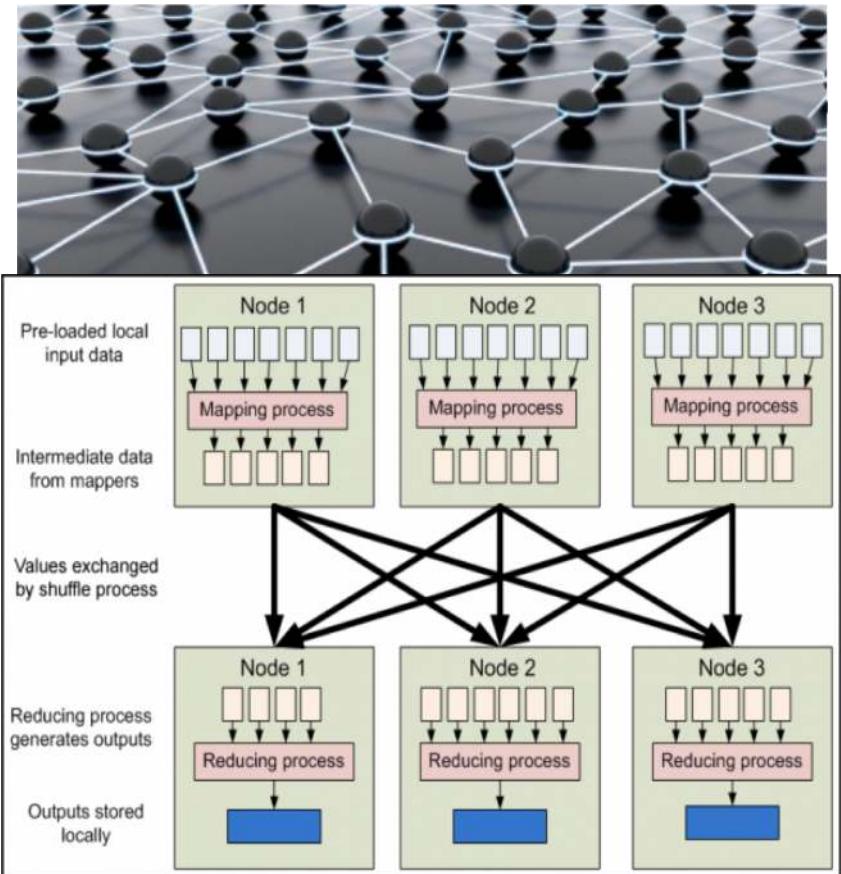
Vertical scaling

What if the biggest computer is not enough?

Buy many computers (cluster) –
horizontal scaling

Key ingredients for computing at scale

- *Networking infrastructure*
 - Clusters of computers that work together to a common goal can provide the needed resources
- *Distributed data*
 - For parallel processing
- *Distributed processing*
 - Shared-nothing model
 - New parallel programming paradigms

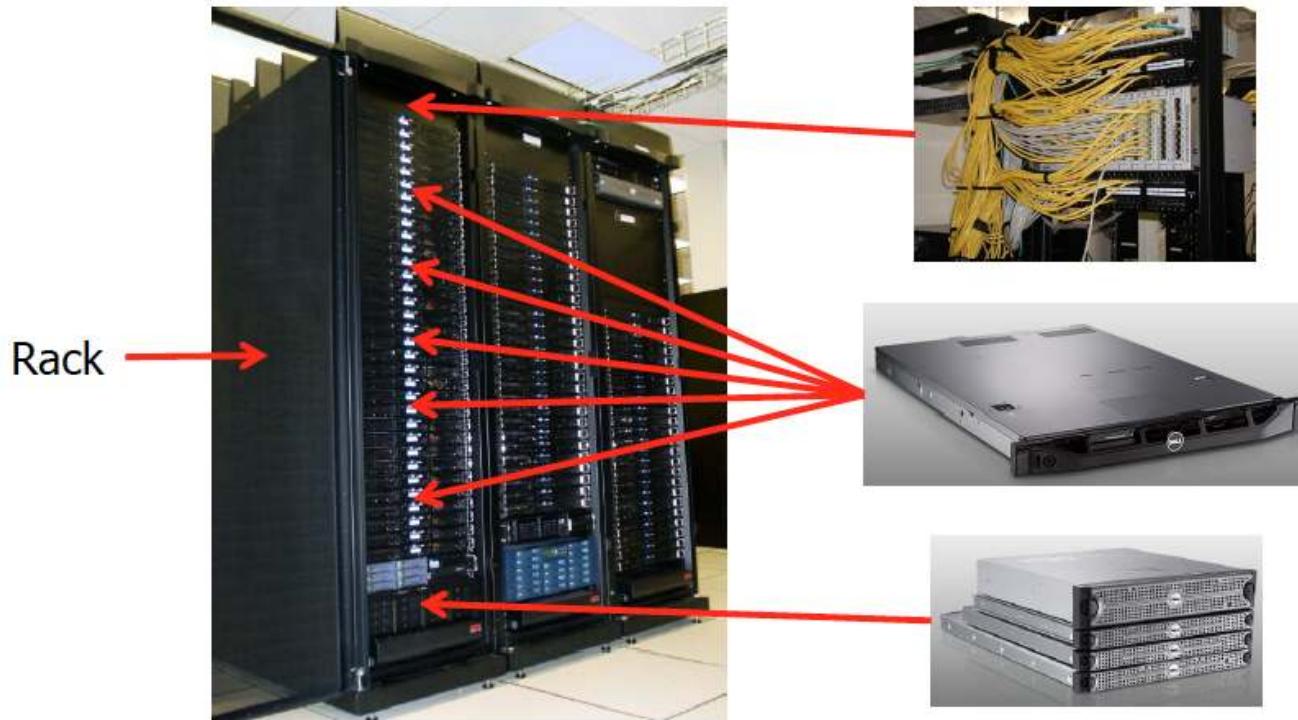


Computer clusters

Set of loosely or tightly connected computers that work together so that, in many aspects, they can be viewed as a single system

- Rely on **shared-nothing architecture** designed for data intensive computing
- Similar, very simple, basic and cheap components, available in large numbers
- Close interconnection (same room?)
- Each node set to perform the same tasks
- A cluster can be owned and used by a single organization or be available on the cloud

Cluster architecture



Computer nodes (servers) are stored on racks
8–64 compute nodes on a rack

Network **switch**
(connects nodes with each other and with other racks)

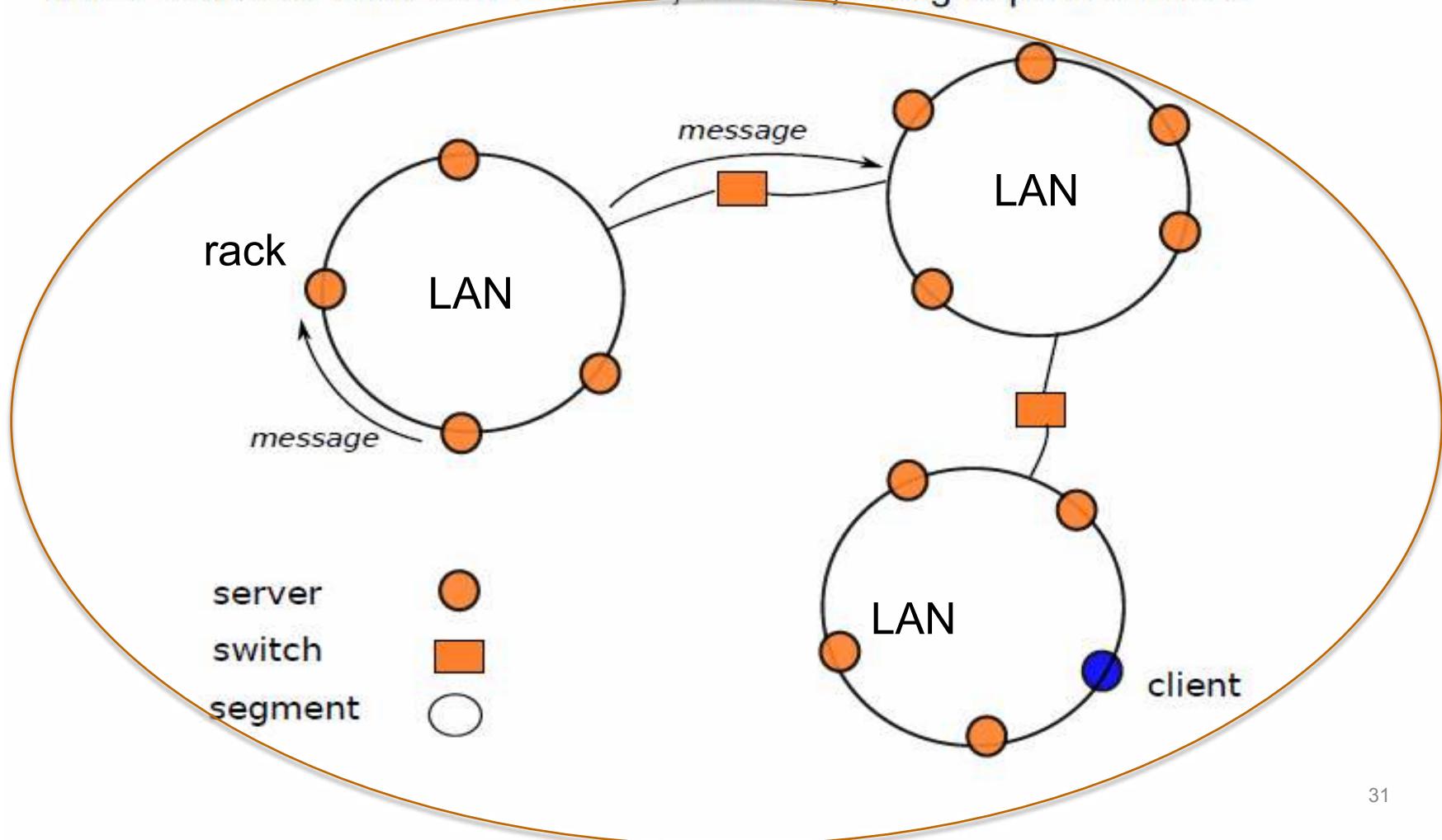
Many **nodes/blades**
(often identical)

Storage device(s)

Cluster architecture

Capacity – bandwidth: the amount of information that can be transmitted over the channel in a given time unit

Three communication levels: “racks”, clusters, and groups of clusters.

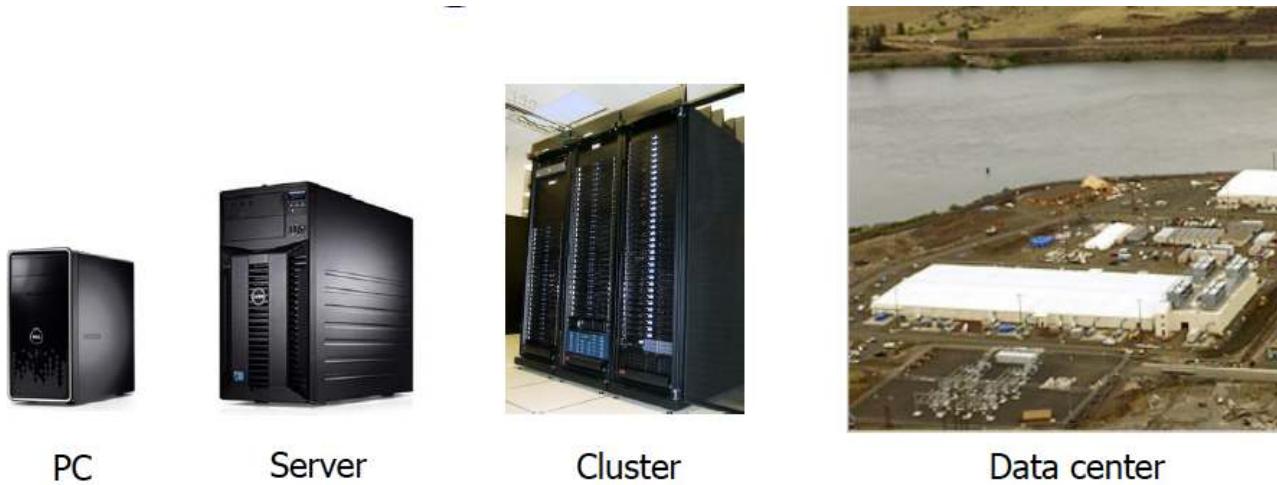


Power and cooling

- Clusters need lots of power
 - Example: 140 Watts per server
 - Rack with 32 servers: 4.5kW (needs special power supply!)
 - Most of this power is converted into heat
- Large clusters need massive cooling



Further scaling out (1)



- What if your cluster is too big (hot, power hungry) to fit into your office building?
 - Build a separate building for the cluster
 - Building can have lots of cooling and power
 - Result: **data center** (= a huge cluster, with additional special hardware)
- Hundreds or thousands of racks
- It can be owned by a single organization and used by several organizations (often according to **cloud computing principles**)

Further scaling out (2)



PC



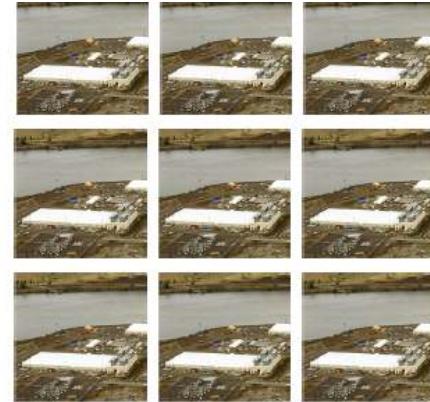
Server



Cluster



Data center



Network of data centers

- What if even a data center is not big enough?
 - Build additional data centers
- Data centers are often globally distributed

Example: google data centers

Typical setting of a Google data center.

- ① \approx 40 servers per rack;
- ② \approx 150 racks per data center (cluster);
- ③ \approx 6,000 servers per data center;
- ④ how many clusters? Google's secret, and constantly evolving ...

Rough estimate: 150-200 data centers? 1,000,000 servers?



Back to scaling

VERTICAL SCALING



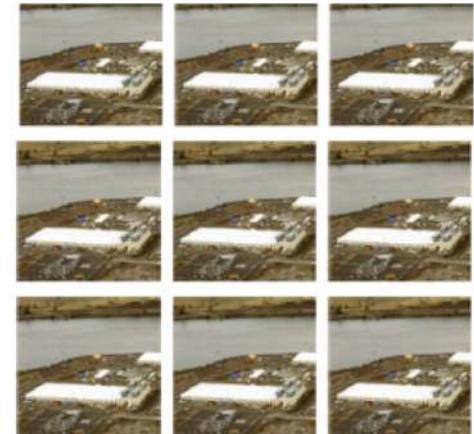
Server



Cluster



Data center



Network of data centers

HORIZONTAL SCALING



PC

ALL SHARED NOTHING ARCHITECTURES

Architectural models

What is the best architecture for developing data-intensive applications?

Some numbers

Exchanging data through the Internet is slow and unreliable with respect to LAN's

Latency: time to initiate an operation

Bandwidth: amount of information transmitted over a channel in a given time unit

| Type | Latency | Bandwidth |
|----------|---|--|
| Disk | $\approx 5 \times 10^{-3}$ s (5 millisec.); | At best 100 MB/s |
| LAN | $\approx 1 - 2 \times 10^{-3}$ s (1-2 millisec.); | $\approx 1\text{GB/s}$ (single rack); $\approx 100\text{MB/s}$ (switched); |
| Internet | Highly variable. Typ. 10-100 ms.; | Highly variable. Typ. a few MBs.; |

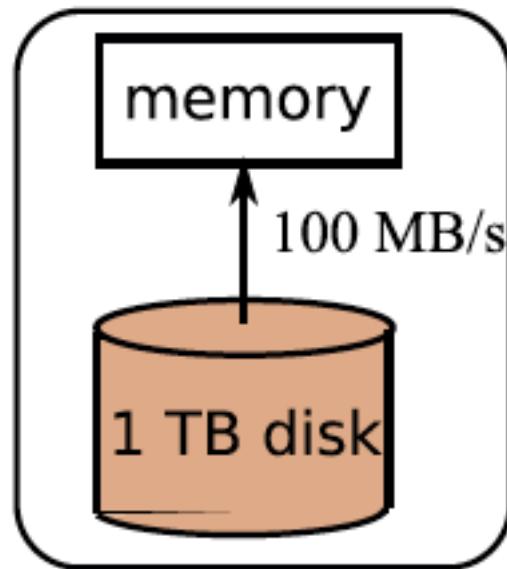
1. It is approx. one order of magnitude faster to exchange in-memory data between 2 computers machines in the same rack than for a single computer to read the same data written on the disk
2. But bandwidth is a shared resource...
3. Similar situation for latency, but less impressive

Reference scenario – analytical application

- Sequentially read a large dataset from disks
 - **batch operations** on the whole collection
 - particularly relevant for applications where most files are written once (by appending new content), then read many times
- Under this assumption:
 - the seek time (time to position the head on the appropriate disk track) is negligible regarding the transfer time
 - the advantages of **data distribution (partitioning)** and **task parallelization** become clearer

Why data distribution?

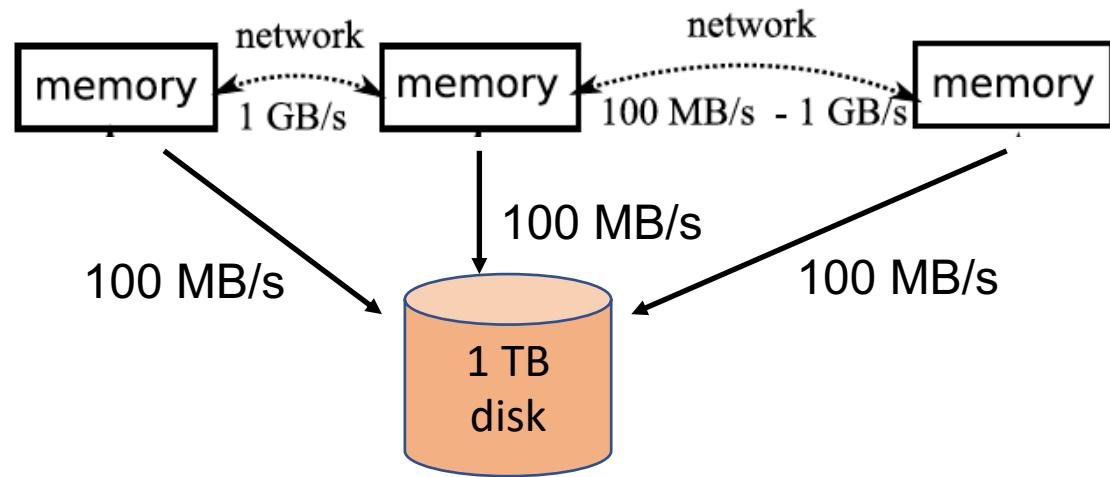
- Centralized database
- No data distribution, sequential access
 - It takes 166 minutes (more than 2 hours and a half) to read 1 TB from disk



a. Single CPU, single disk

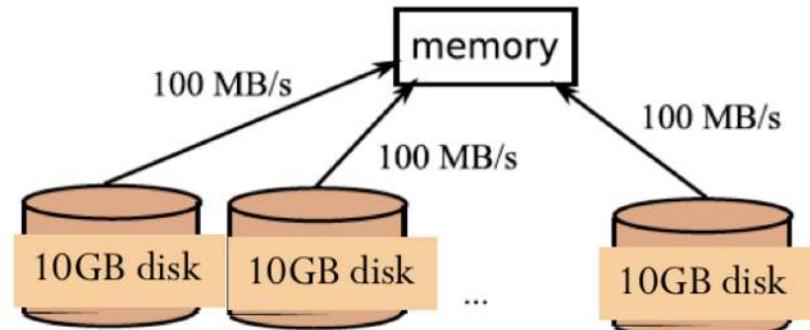
Why data distribution?

- Shared disk architecture
- No data distribution, distributed access and processing
- With 100 computers but a shared disk, this works as long as the task is CPU intensive (as in High Performance Computing), but becomes unsuited if large data exchanges are involved



Why data distribution?

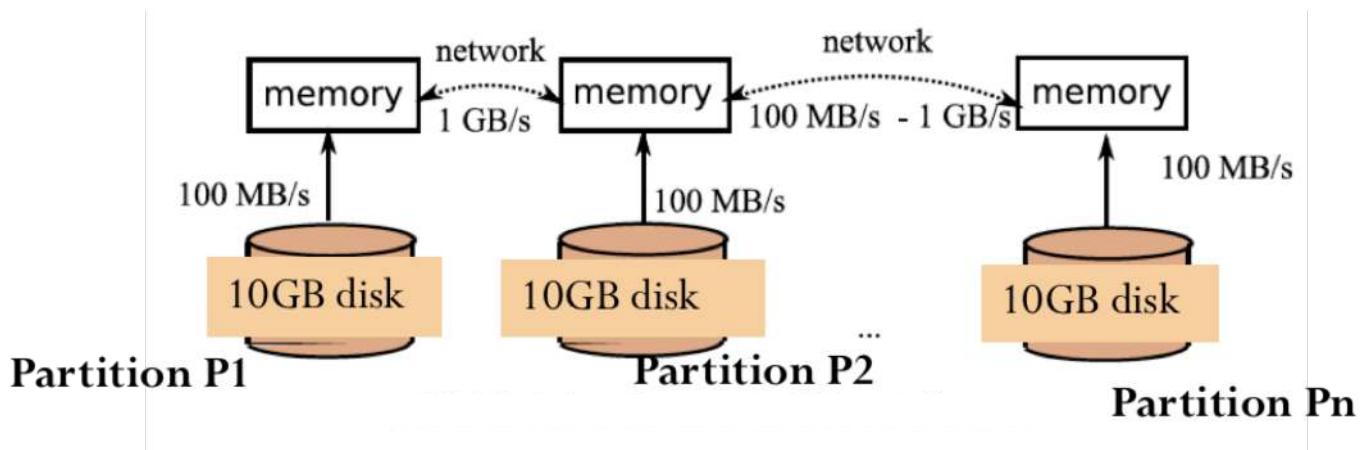
- Shared memory and processor architecture
- Data distribution, shared processor, parallel access
 - With 100 disks, assuming the disks work in parallel and sequentially (about 10 GBs from each disk): about 1mn 30s
 - Data partitioning
 - Splitting a big database into smaller subsets called partitions so that different partitions can be assigned to different nodes (also known as *sharding*)
 - When the size of the data set increases, the CPU of the computer is typically overwhelmed at some point by the data flow and it is slowed down



b. Parallel read: single CPU, many disks

Why data distribution?

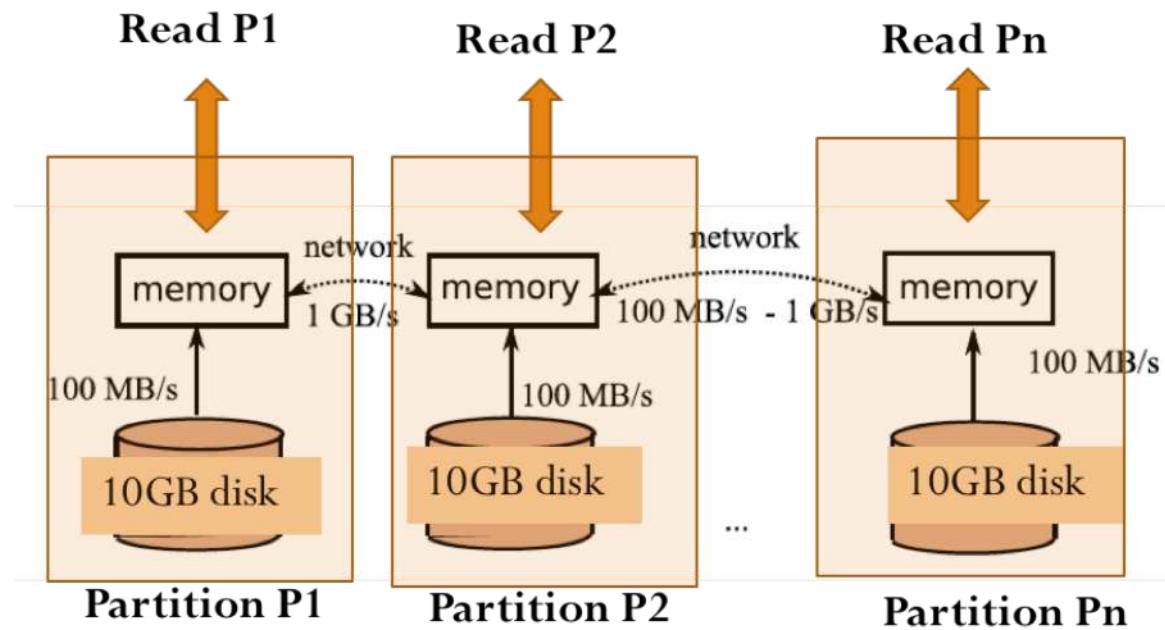
- Shared nothing architecture
- Distributed data, distributed access
 - With a cluster of 100 computers, each disposing of its own local disk: each CPU processes its own dataset
- Data partitioning



This solution is scalable, by adding new computing resources

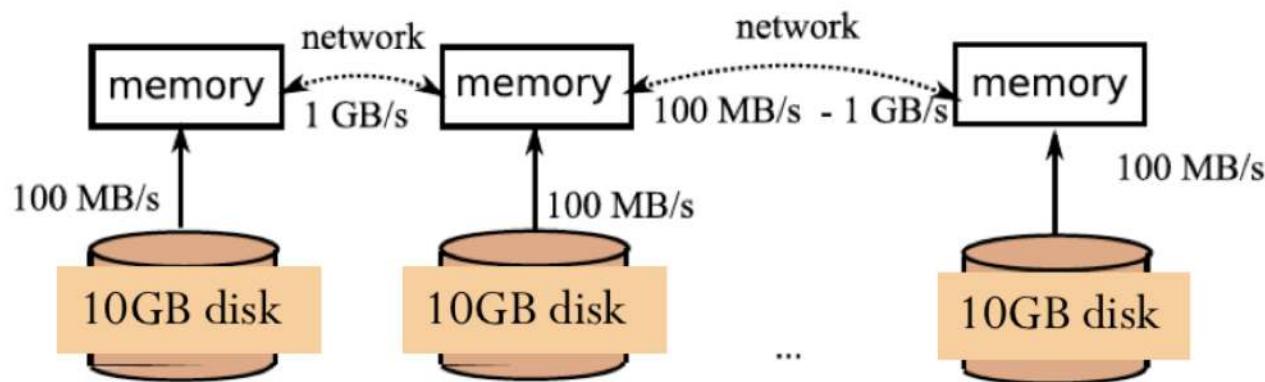
Side effect of data partitioning: task parallelization

- Shared nothing architecture
- Distributed data, distributed access: pushing the program near to the data
- Task parallelization (if the task can be parallelized!): the same operation executed over distinct partitions, by different nodes



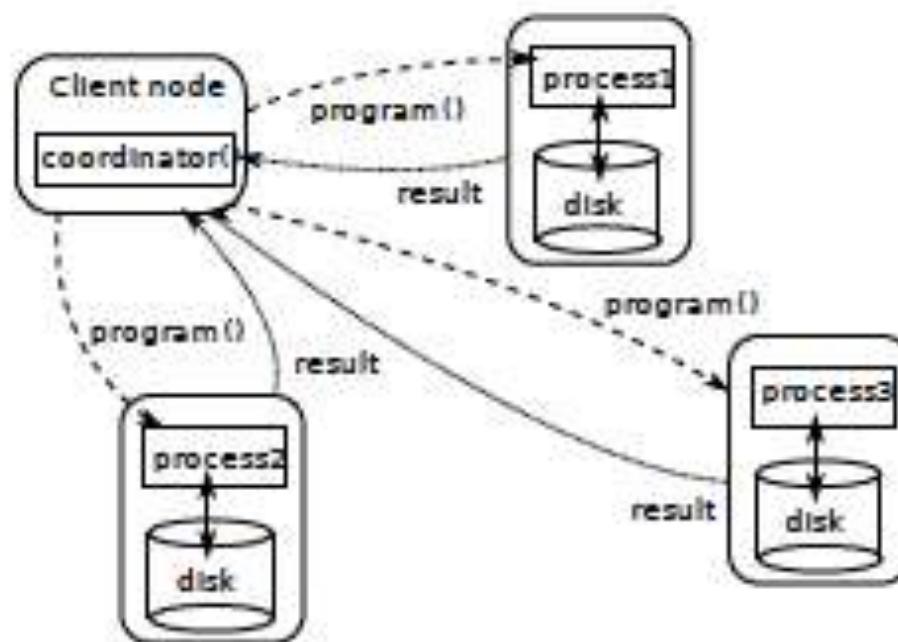
What about control distribution?

- P2P architecture
- Control is distributed



What about control distribution?

- One Master - Many Slaves
- Centralized control

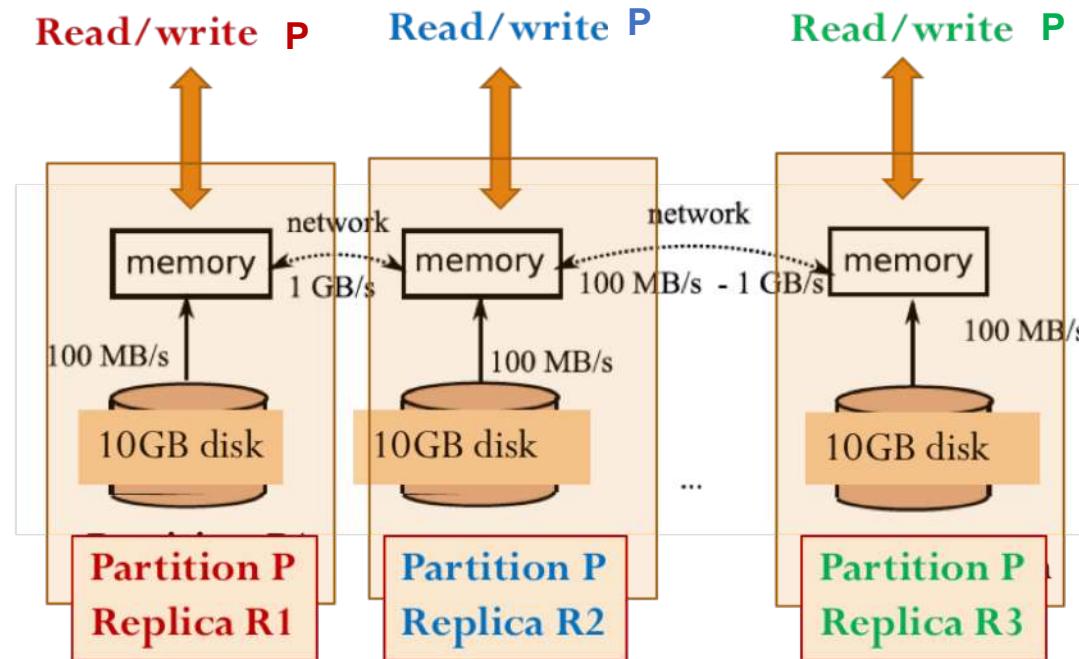


An alternative scenario – transactional application

- Workload consisting of lots of **reads** and **writes** operations, each one randomly accessing *a small piece of data in a large collection*
- More in the spirit of a database operations
- Random access to large files, seek time cannot be ignored
- Shared-nothing architectures still useful
- Distribute the load (read/write requests) through **replication**
 - Keeping a copy of the same data on several different nodes, potentially in different locations
 - Replication provides redundancy: if some nodes are unavailable, the data can still be served from the remaining nodes

Why load distribution?

- Shared nothing architecture
- Distributed data, distributed load
- Data replication
- Different operations executed over the same partition by different nodes



Principle #1

- Disk transfer rate is a bottleneck for batch processing of large scale data sets
 - typical of analytical processing
 - *data partitioning and task parallelization on many machines are a mean to eliminate this bottleneck*

Possible with shared nothing architectures

Principle #2

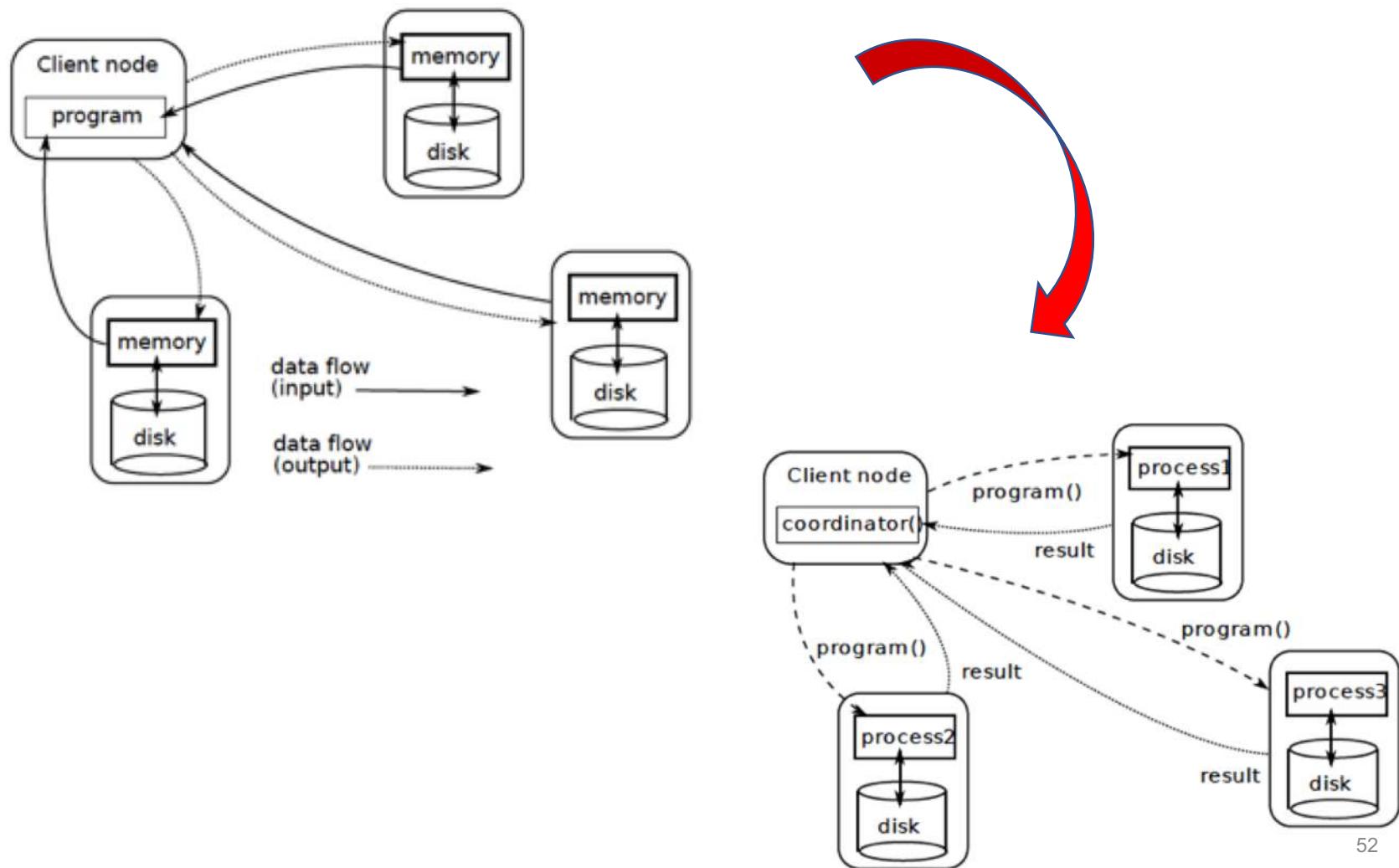
- Disk seek time is a bottleneck for transactional applications that submit a high rate of random accesses
 - typical of transaction processing
 - *replication, distribution of writes and distribution/parallelization of reads are the technical means to make such applications scalable*

Possible with shared nothing architectures

Principle #3

- **Data locality**
 - bandwidth is a scarce resource, and program should be “pushed” near the data they must access to
 - in this way, we rely on more fast communications (lower costs)

Principle #3



Additional motivations for data distribution – Fault tolerance/Reliability

- Reliability denotes the ability of a distributed system to deliver its services even when one or several of its software or hardware components fail
- **Faults** - system components deviating from their specification: program bugs, human errors, hardware or network problems
- Systems that anticipate faults and can cope with them are called **fault-tolerant**: at some time, they will produce the correct result
- Clusters proportionally increase the rate of hardware faults
- *Fault tolerance becomes a fundamental issue in distributed architectures*
- Fault tolerance is based on the assumption that a participating machine affected by a failure can always be replaced by another one, and not prevent the completion of a requested task
- *Reliability relies on redundancy/replication of both the software components and data*

Additional motivations for data distribution – Performance/Availability

- **Availability** is the capacity of a system to limit as much as possible its **latency**
- Involves several aspects:
- Failure detection
 - Periodically monitor the participating nodes to detect failures as early as possible
 - design quick restart protocols
- **Quick** recovery procedure

Said in other terms: node failures do not prevent survivors from continuing to operate

Additional motivations for data distribution - Scalability

- **Scalability** is the ability of a system to cope and continuously evolve in order to support increased load
- **Load parameters** depend on the specific system/application:
 - data volume
 - number of works (e.g., number of transactions)
 - ...
- Some systems are **elastic**
 - They can automatically add computing resources when they detect a load increase
 - Useful if the load is unpredictable
- Some other systems are **manually scaled**
 - A human analyzes the capacity and decides to add more machines to the system

Architectural approaches for large scale data management

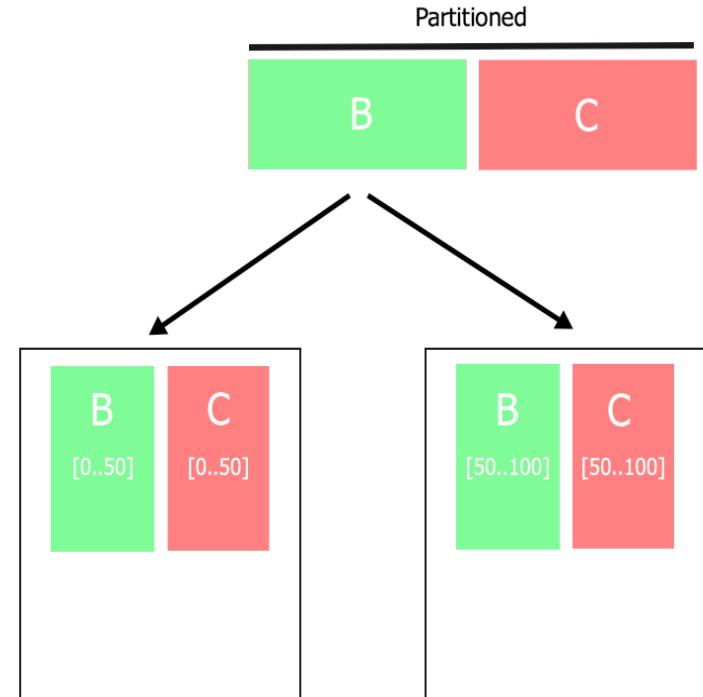
Properties of distributed data systems

- Data distribution
 - Partitioning
 - Replication
- Fault tolerance/Reliability
- Performance/ Availability
- Scalability

Partitioning

Partitioning

Splitting a big database into smaller subsets called **partitions** so that different partitions can be assigned to different nodes (also known as *sharding*)



At the basis of intra-operation parallelism

- different instances of the same operation are processed in parallel, by executing them in parallel over different nodes
- commonly referred to as *data parallelism*
- scalable approach
- at the basis of cluster computing

Assumptions

- Shared nothing architecture
- Scalable distributed data system: spread the data and the query load evenly across nodes
- *Data locality principle*: data placement is a critical performance issue
- *Data* as a set of n records R with attributes (K, A, B, C) [denoted by $R(K, A, B, C)$]
- *Load*: set of frequent read/write accesses with respect to attribute K
- K can be chosen as **partition key**

How to partition

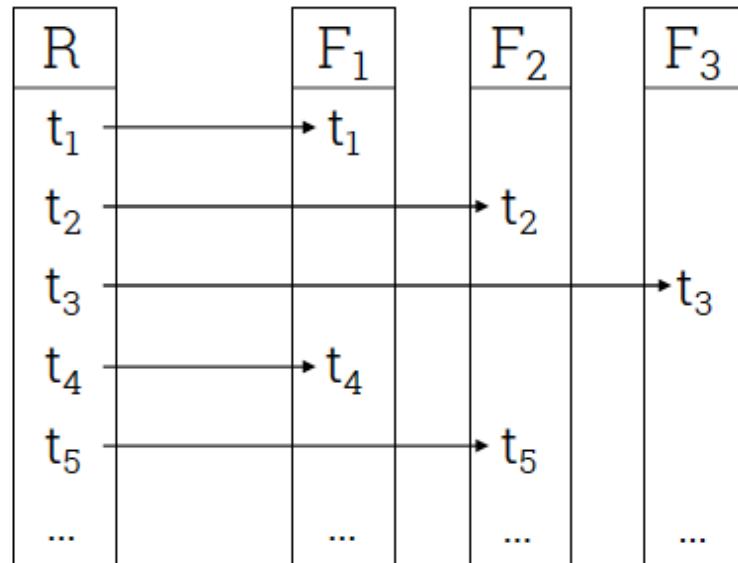
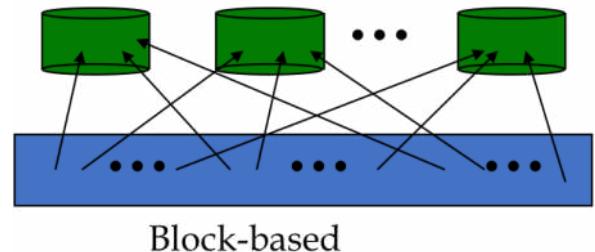
- Each dataset is divided in p partitions where p depends on dataset size and access frequency
- Favour balanced partitioning
 - Skewed partitions non scalable (the load is not evenly distributed)
- Avoid hot spots: partitions with disproportionately high load (larger partition, higher number of requests)

What kind of requests

- Batch query
 - read all data items
 - typical of analytical processing
- Point query
 - selection of a single record
 - typical of transactional processing
- Multipoint/Range query
 - selection of all the records satisfying a given condition (e.g., $A > 3$)
 - typical of transactional processing

Block-based partitioning

- Arbitrarily partition the data such that the same amount of data n/p is placed at each node
- Use a Round-Robin approach or place the first n/p into the first node, the second n/p into the second node and so on
- No hot spots

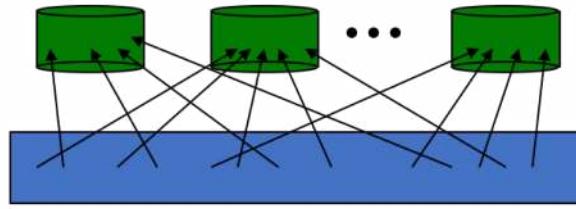


Distributes data evenly
Good for scanning full relation
Not good for point or range queries

(all nodes have to be accessed and all nodes contains data of interest)

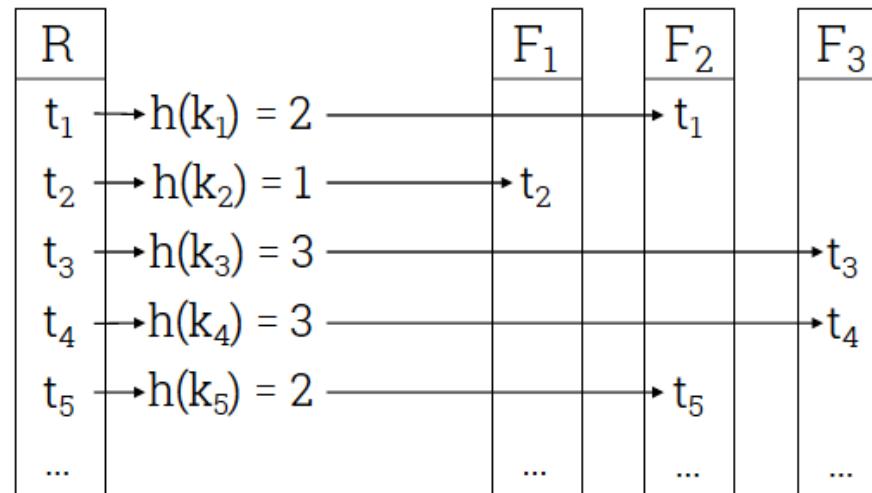
(all nodes have to be accessed but only few nodes might contain data of interest)

Hash-based partitioning



Hash-based

- Applies a hash function to some attribute that yields the partition number in $\{1, \dots, p\}$

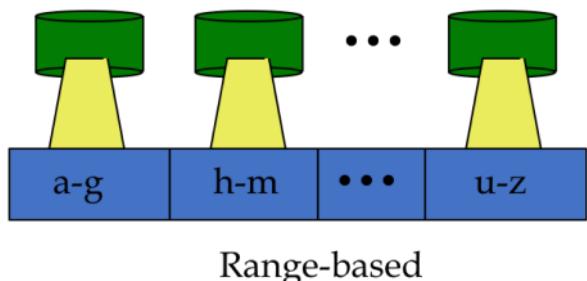


(just one node has to be accessed and the node contains all data of interest)

Distributes data evenly if hash function is good
Good for point queries on key and joins
Not good for range queries and point queries not on key

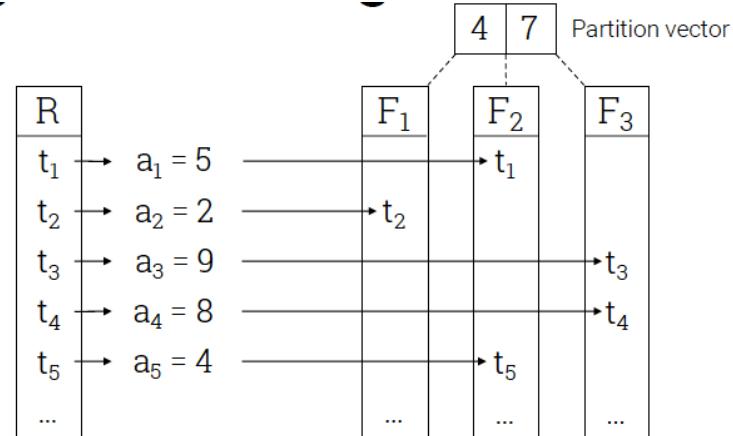
(all nodes have to be accessed but only few nodes contain data of interest)

Range-based partitioning



- Partition the data according to a specific range of an attribute (such that close values are in the same or nearby servers)
 - find separating points k_1, \dots, k_p
 - send to the first node the tuples such that $-\infty \leq t.K \leq k_1$
 - Send to the second node the tuples such that $k_1 < t.K \leq k_2$
 - Send to the n -th node the tuples such that $k_p \leq t.K \leq +\infty$
- Partition boundaries might be manually chosen by an administrator or automatically chosen by the system
- Might generate hot spots

(few nodes, those containing data of interest, have to be accessed)



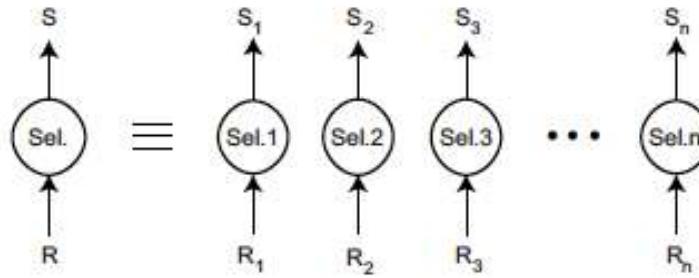
Good for some range queries on partition attribute
Need to select good vector to avoid data/execution skew

Partitioning

- Each distributed data system supports a specific partitioning scheme
- The partition key can be selected for any dataset
- The selection of the partition key might favour some requests and make some others very inefficient
- Depends on the reference workload
- Choice to be taken at design time

From partitioning to intra-operator parallelism

- **Intra-operator parallelism:** the (read) operator can be decomposed in many sub-operators, each of them executed on a given partition, in an independent way



- The execution of the operator is limited to nodes containing relevant data (i.e., data contributing to the result)

 Operator  Instance i of operator n = degree of parallelism

Partitioning can make the execution of operator supporting intra-operator parallelism more efficient

Partitioning: hot spot

- Hash-based and range-based partitioning help in determining the partition containing a given key and can reduce hot spot when the load is evenly distributed
- When all reads and writes are for the same key, you still end up with all requests being routed to the same partition
- *Partitioning does not help in solving this problem*
- *Replication helps in this case*

Time for exercizes

Partitioning: secondary index

- Partitions work as a kind of global **partition key index**
- What happens if our queries want to access data with respect to properties that do not correspond to the partition key?
- Partitions cannot help ...
- Need for a **secondary index**

Partitioning: secondary index

- each partition is completely separate
- each partition maintains its own secondary indexes, covering only the documents in that partition
- local index

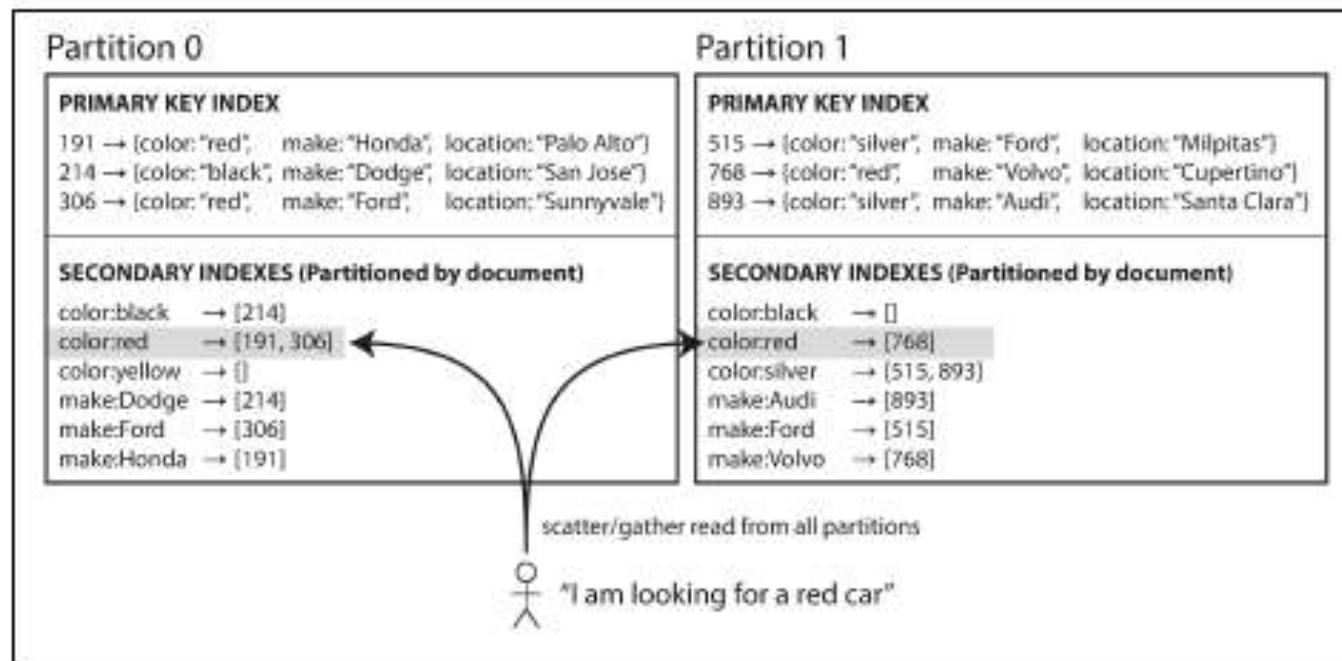


Figure 6-4. Partitioning secondary indexes by document.

- easy write
- read quite expensive
- widely used

Partitioning: secondary index

- a **global index** covers data in all partitions
- a global index is data as well and must be partitioned, possibly differently from the primary key index

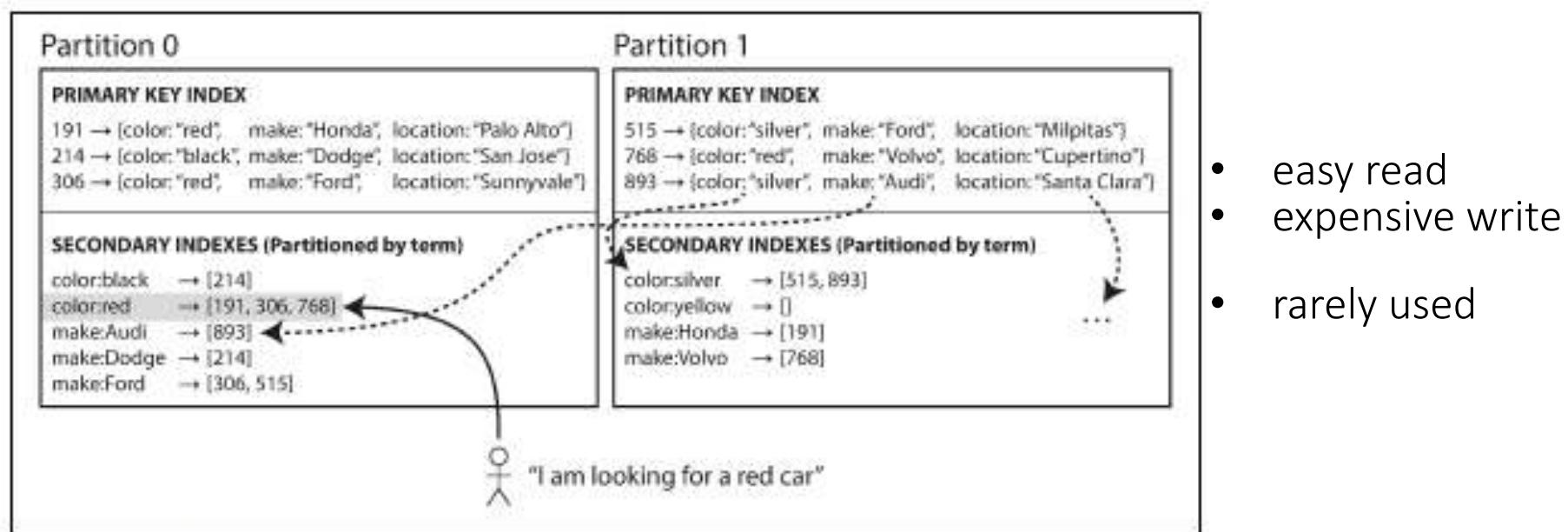


Figure 6-5. Partitioning secondary indexes by term.

Rebalancing

- The load may change
 - The number of requests or the dataset size increases, so you want to add more nodes to handle the increased load
- A machine fails, and other machines need to take over the failed machine's responsibilities

Rebalancing: process of moving load
from one node in the cluster to another

Example

- Suppose a hash-based partitioning is applied:
 - $H(v) = v \bmod p$, where p is the number of nodes
- If the number of nodes changes, most of the data must be moved
- *Before* $p = 10$
 - $H(10) = 0$
 - $H(11) = 1$
 - $H(12) = 2$
- *After* $p = 11$
 - $H(10) = 10$
 - $H(11) = 0$
 - $H(12) = 1$

Rebalancing: automatic or manual

- Rebalancing is an expensive operation, because it requires moving a large amount of data from one node to another
- **Fully automatic rebalancing:** the system decides automatically when to move partitions from one node to another, without any administrator interaction
 - less operational work to do for normal maintenance
 - it can be unpredictable
- **Fully manual rebalancing:** the assignment of partitions to nodes is explicitly configured by an administrator, and only changes when the administrator explicitly reconfigures it
 - It's slower than a fully automatic process, but it can help preventing operational surprises
 - *Typically applied in distributed data systems*

Request routing

When a client wants to make a request, how does it know which node to connect to?

- As partitions are rebalanced, the assignment of partitions to nodes changes
- Somebody needs to stay on top of those changes in order to answer the question

Request routing

- Three main approaches
 - Allow clients to contact any node. If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client
 - Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer
 - Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node without any intermediary

Request routing

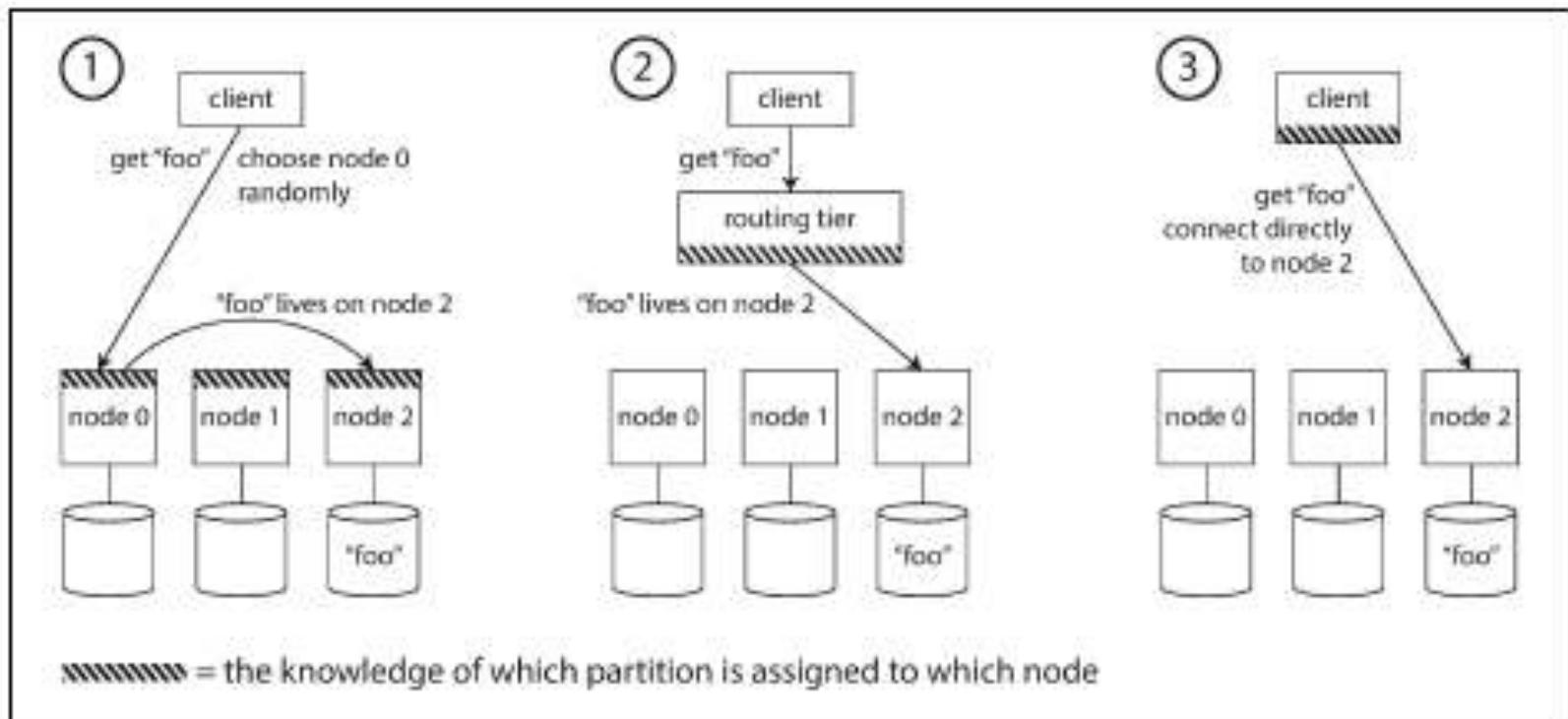


Figure 6-7. Three different ways of routing a request to the right node.

Request routing

How does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?

Request routing – coordination service

1. Protocols for achieving consensus in a distributed system, but they are hard to implement correctly
2. Rely on a **separate coordination service** (e.g., ZooKeeper) to keep track of this cluster metadata

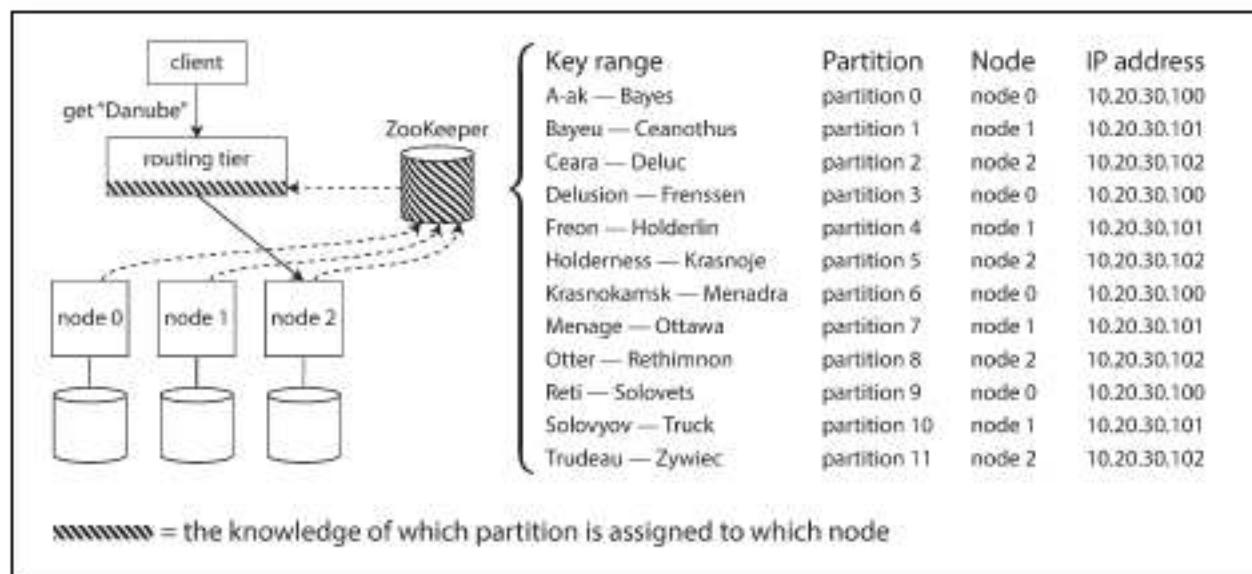
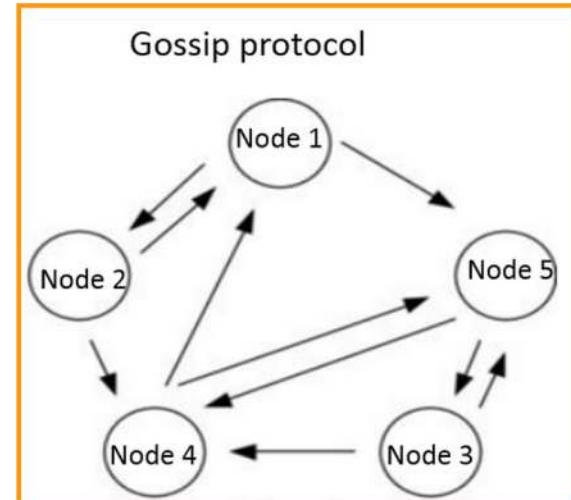


Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

Request routing – gossip protocol

- Some systems rely on a **gossip protocol** among the nodes to disseminate any changes in cluster state
- Requests can be sent to any node, and that node forwards them to the appropriate node for the requested partition
- More complexity in the database nodes but no dependency on an external coordination service



Take away

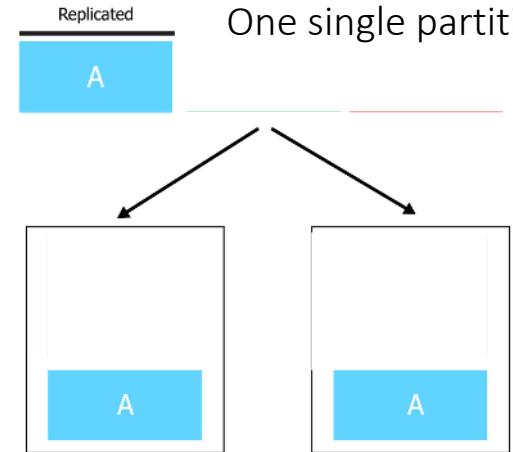
- Data partitioning is the key mechanism for scalability
- Three main approaches: block-based, range-based, and key-based
- Searches against non partitioning columns require specialized management (secondary indexes)
- Rebalancing (either automatic or manual) is an issue for guaranteeing scaling
- Request routing has to be carefully managed
- Request routing relies on either a coordination service or on some gossip protocol

Replication

Replication

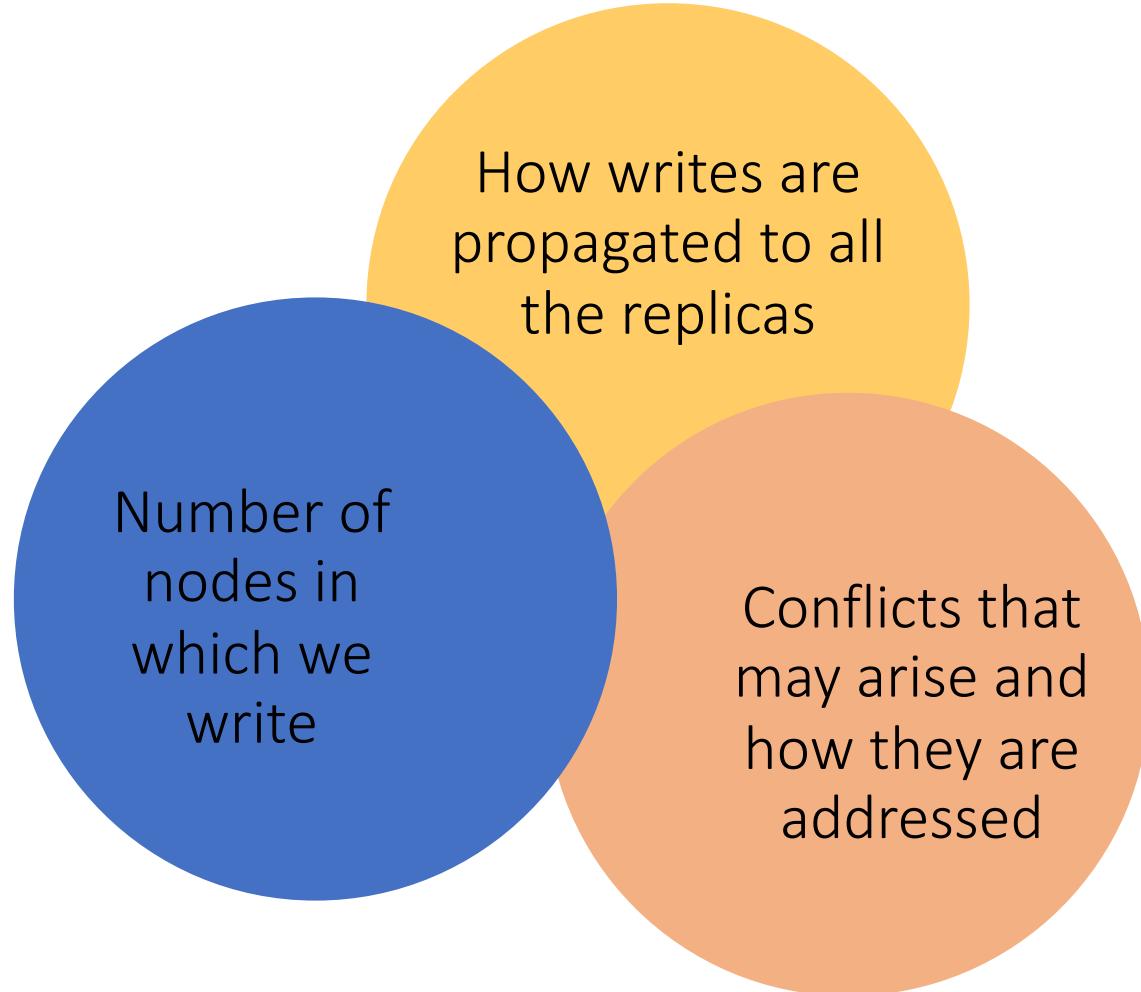
Replication means keeping a copy of the same data on multiple machines that are connected via a network

Assumption:
One single partition

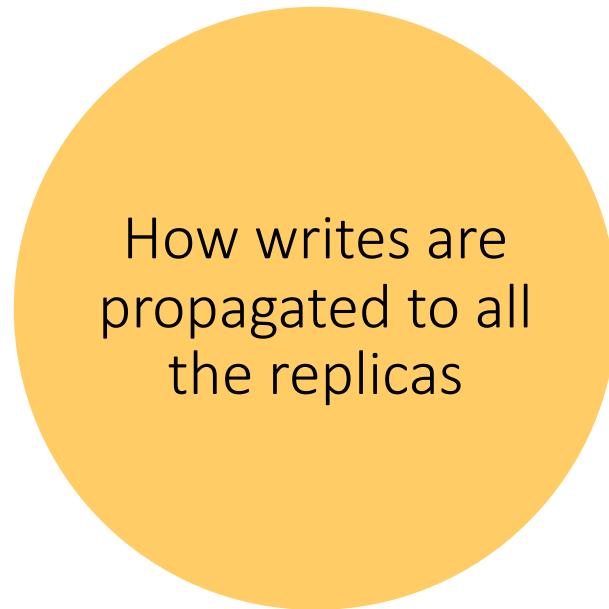


- Each node that stores a copy of the database is called a **replica**
- Most of the properties required by a distributed system depend on the replication of data
 - To keep data geographically close to your users (and thus reduce **latency**)
 - To allow the system to continue working even if some of its parts have failed (and thus increase **availability**)
 - To scale out the number of machines that can serve read/write queries (on different items) (and thus increase read **throughput, scalability**)
- Cons
 - **Performance.** Writing several copies of an item takes more time, which may affect the throughput of the system
 - **Consistency.** how do we ensure that all the data ends up on all the replicas?

Replication protocols



Replication protocols

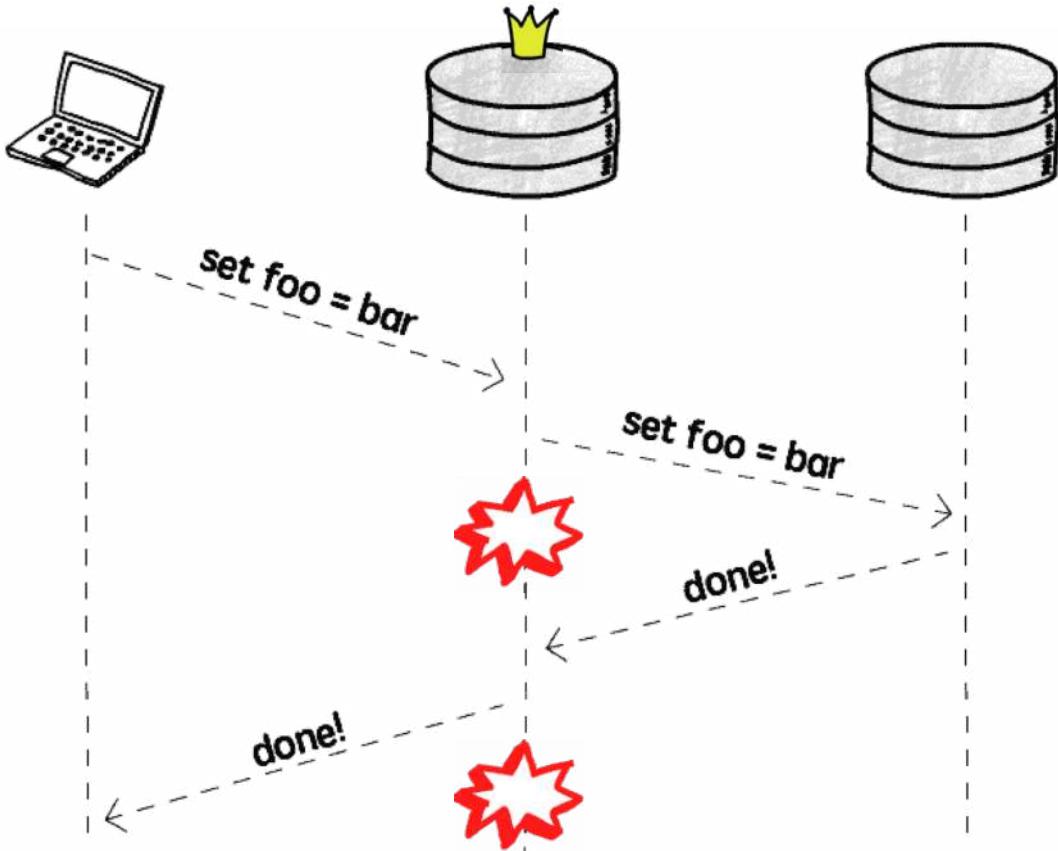


- Synchronous
- Asynchronous

Leader: nodes receiving first the write request

Follower: any other node storing a replica

Synchronous replication



Pros:

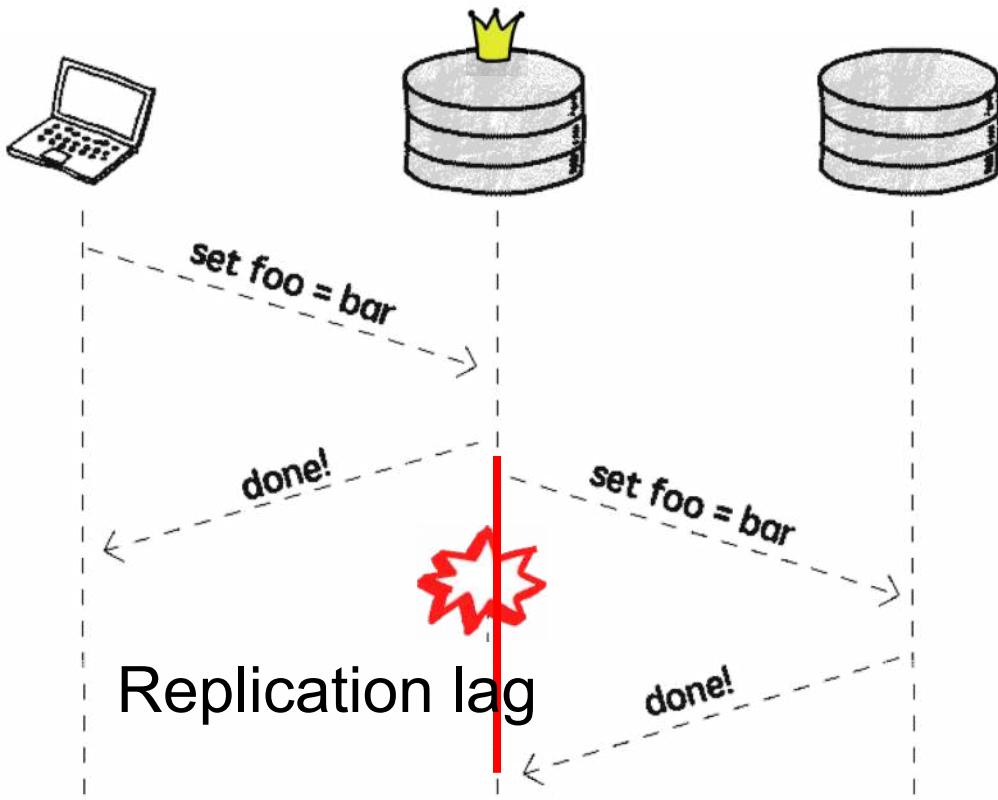
- the follower is guaranteed to have an up-to-date copy of the data (**strong consistency**)
- if the leader fails, we can be sure that the data is still available on the follower
- write requests are sequentially executed by the leader: no concurrent writes but **lower throughput and latency**

Cons:

- if the synchronous follower doesn't respond, the write cannot be processed (**limited availability**)
- applications have to wait for the completion of other clients' requests

the leader waits until the follower has confirmed that it received the write before reporting success to the user, making the write visible to other clients

Asynchronous replication



- Pros:

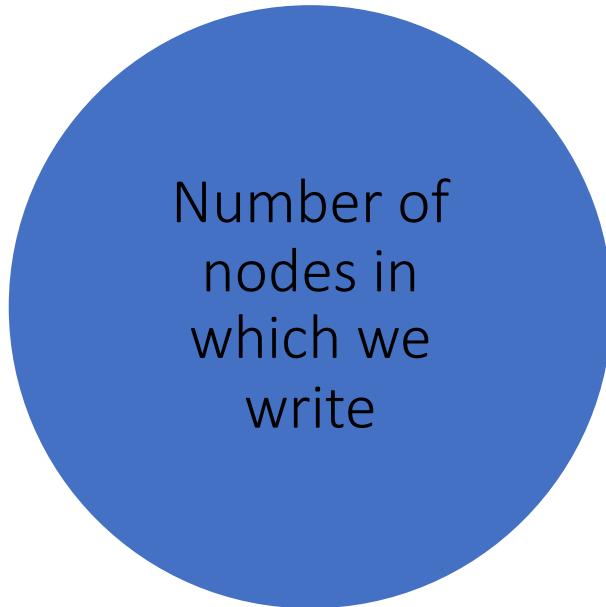
- the leader can continue processing writes, even if all of its followers have fallen behind (**high throughput, low latency**)
- at some point the replicas will become consistent (**eventual consistency**)
- **replication lag**: delay between a write happening on the leader and being reflected on a follower

- Cons:

- some of the replicas may be out of date (**data** at some point are **inconsistent**)
- if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost

the leader sends the message, but doesn't wait for a response from the followers

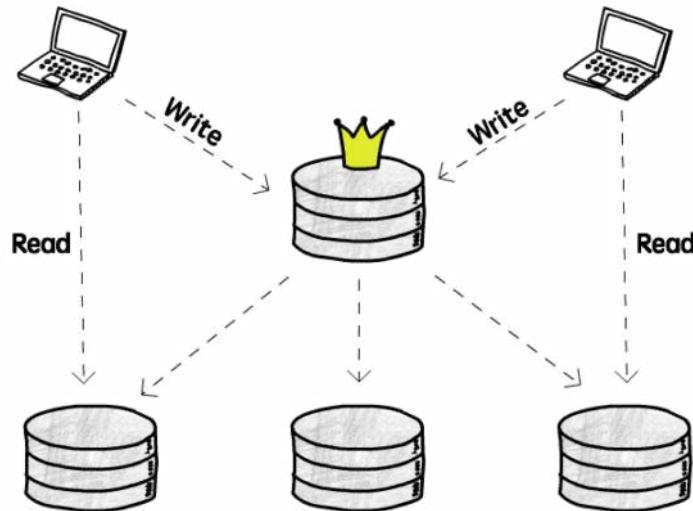
Replication protocols



- Single-leader replication
(master-slave replication)
- Multi-leader replication
(master-master replication)
- Leaderless replication

Single leader replication

- Just one **leader**, storing the primary copy of data



READ

- Clients can read from either the leader or any of the followers
- Followers are read-only from the client's point of view

WRITE

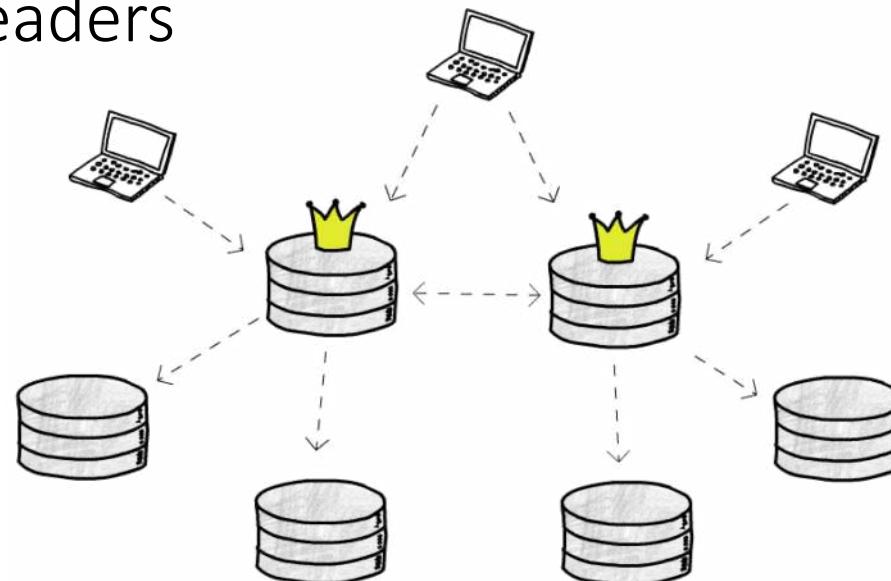
- Clients must send their write requests to the leader
 - it first writes the new data to its local storage
 - it sends the data change to all of its followers
- Each follower takes the log from the leader and updates its local copy of the database accordingly

Single leader replication

- Pros
 - Simple implementation
 - No concurrent writes (no **write conflicts** can arise)
- Cons
 - **Limited throughput:** all write operations are sequentially executed
 - **Limited availability:** single point of failure
 - Under an asynchronous protocol, **read conflicts** (reading old versions of a data item) may arise (see later)

Multi-leader (master-master) replication

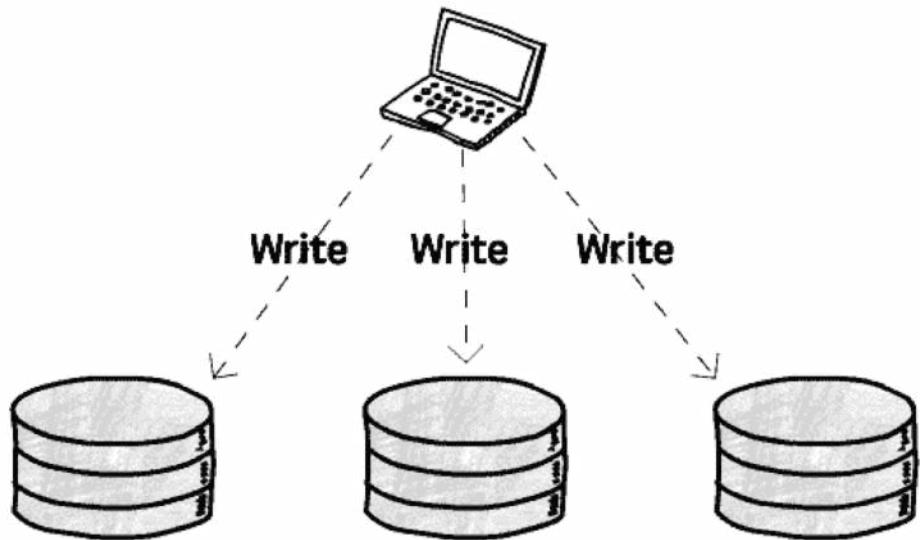
- More than one node can accept writes (**more than one leaders**, often one for each data center)
- Replication still happens in the same way:
 - each leader node that processes a write must forward that data change to all the other nodes
- Typically asynchronous in order not to lose benefits of multiple leaders



Multi-leader replication

- Pros:
 - increased write throughput
 - increased availability: in case of server failure, writes can be sent to other leaders
- Cons:
 - Under an asynchronous protocol **read conflicts**
 - the same data may be concurrently modified through different leaders, and those **write conflicts** must be resolved (see later)
- Usually **one leader in each datacenter**
 - within each datacenter, regular single leader replication is used
 - between datacenters, each leader (asynchronously) replicates its changes to the leaders in other datacenters

Leader-less replication



Every replica can accept writes

- Pros

- No leader = no failover
- High throughput and availability

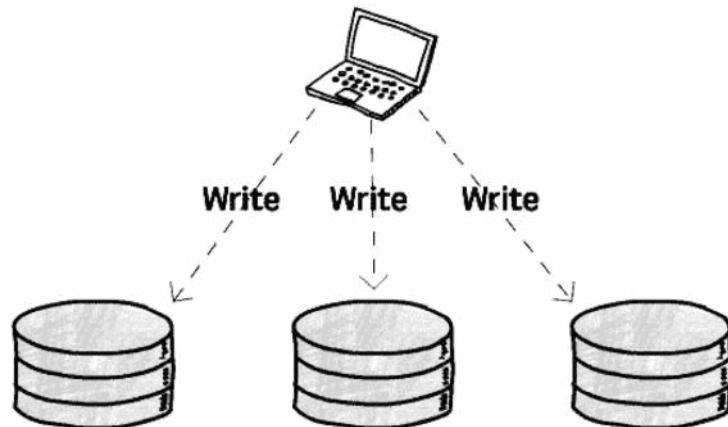
- Cons

- Read conflicts
- Write conflicts

The client sends this write request concurrently to several replicas, and as soon as it gets a confirmation from *some* of them it can consider that write a success and move on.

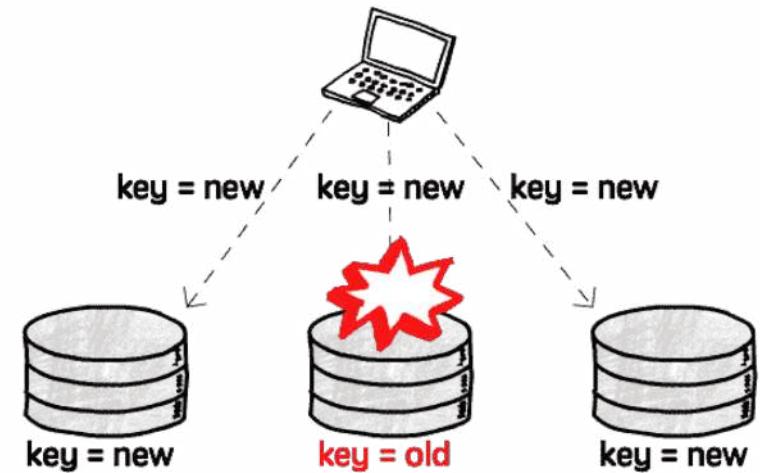
Typical of P2P systems

Leader-less replication



Write on multiple nodes

Read from multiple nodes



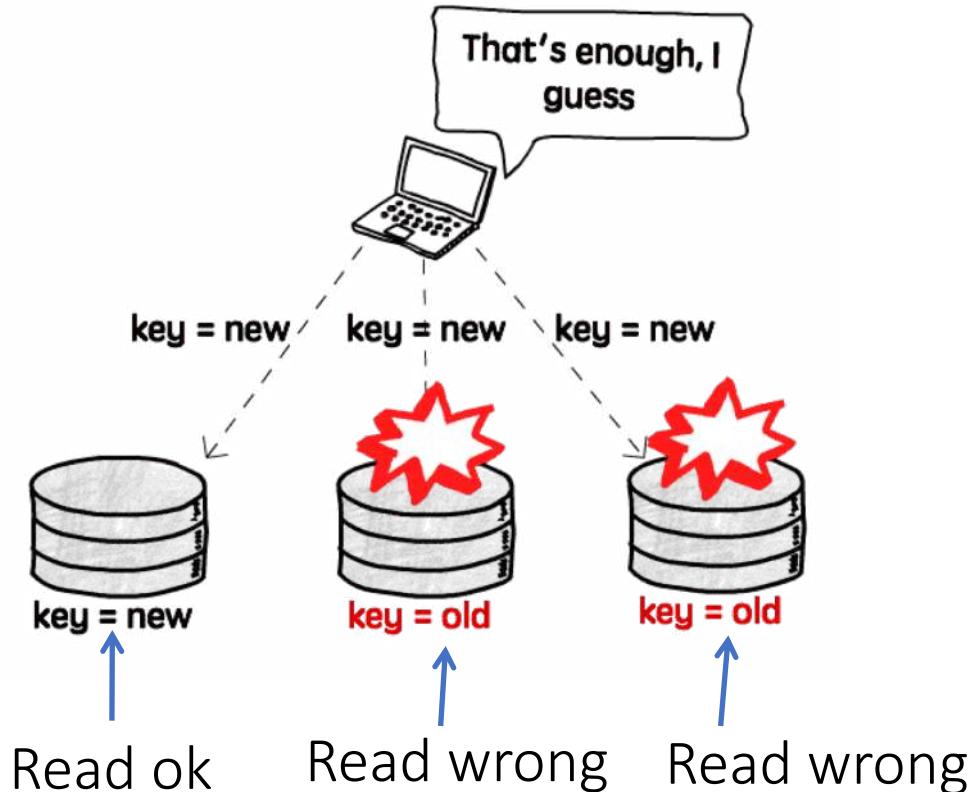
How to be sure that at least one value is up-to-date?

How to be sure that at least one value is up-to-date?

n : number of replicas

w : number of successfull writes – **write quorum**

read requests sent to r nodes in parallel – **read quorum**



$$n = 3$$

$$w = 1$$

$$r = 1$$

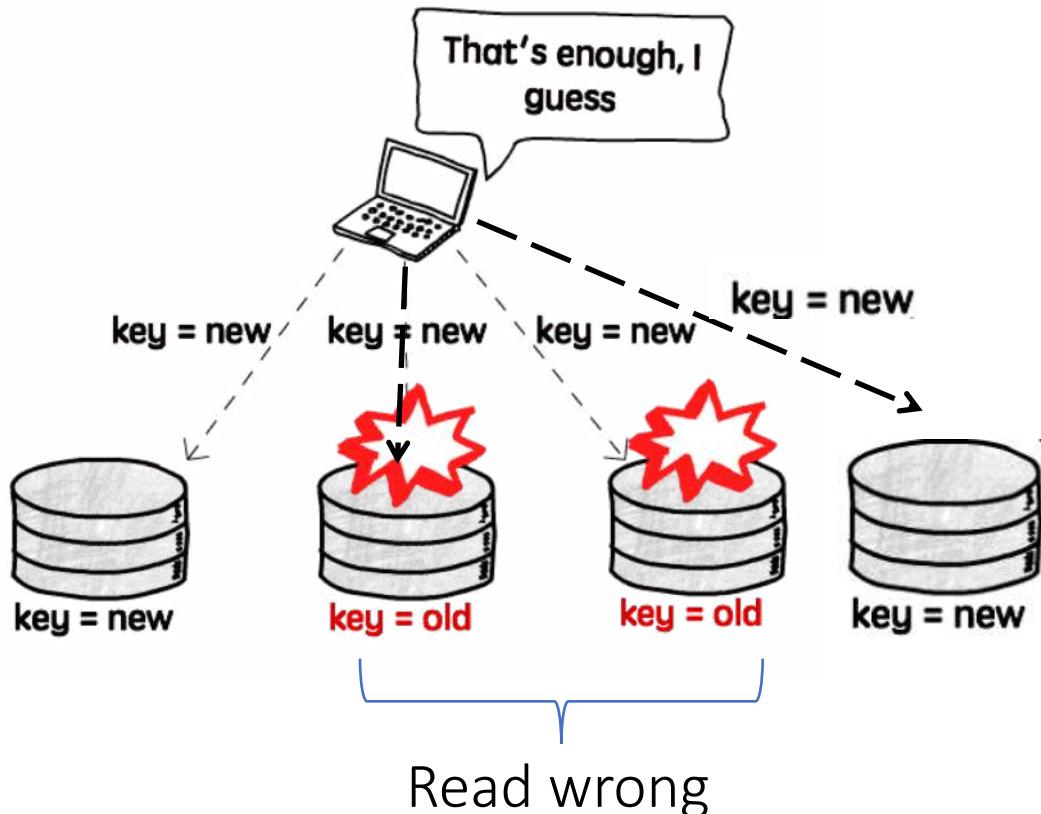
No guarantee of reading
the correct value

How to be sure that at least one value is up-to-date?

n : number of replicas

w : number of successfull writes – **write quorum**

read requests are sent to r nodes in parallel – **read quorum**



$$n = 4$$

$$w = 2$$

$$r = 2$$

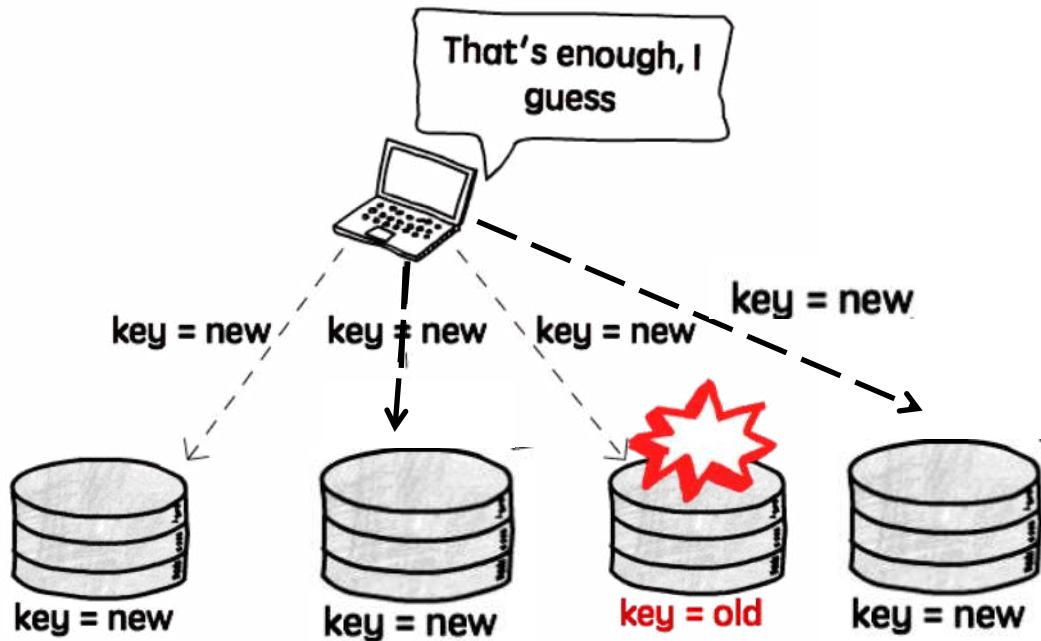
No guarantee of reading
the correct value

How to be sure that at least one value is up-to-date?

n : number of replicas

w : number of successfull writes – **write quorum**

read requests are sent to r nodes in parallel – **read quorum**



$n = 4$

$w = 3$

$r = 2$

Any pair of reads returns at least once the correct value

How to be sure that at least one value is up-to-date?

n : number of replicas

w : number of successfull writes – **write quorum**

read requests are sent to r nodes in parallel – **read quorum**

$$w+r > n$$

At least one read value is up-to-date

The power of quorums

- w and r are usually configurable
- $w < n$: we can still process writes if a node is unavailable
- $r < n$: we can still process reads if a node is unavailable
- $n = 3, w = 2, r = 2$: we can tolerate one unavailable node
- $n = 5, w = 3, r = 3$: we can tolerate two unavailable nodes
- $w = n, r = 1$: fast read but just one failed node causes all database writes to fail

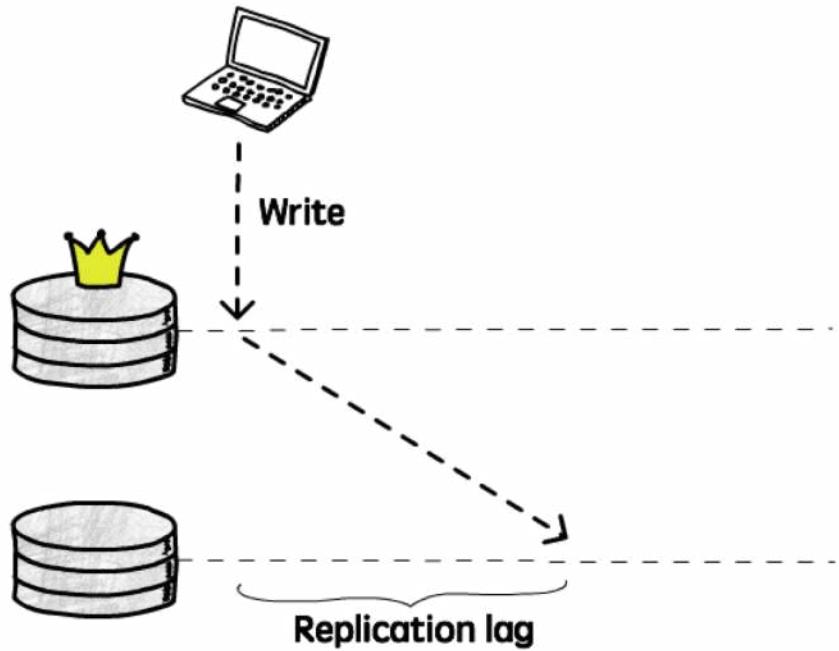
Replication protocols



Conflicts that may arise and how they are addressed

- Read conflicts
 - receiving wrong data for a read request, after the execution of some write
- Write conflicts
 - conflicts due to writes that are concurrently executed

Read conflict – replication lag



- If a client reads from a replica during the replication lag, it will receive outdated information, because the latest update(s) were not applied yet
- In **normal operations**, the replication lag may be only **a fraction of a second**, and not noticeable in practice
- if the system is operating near capacity or if there is a problem in the network, **the lag can easily increase to several seconds or even minutes**
- a real problem for applications

Typical of asynchronous single-leader and multi-leader replication protocols

Read conflict – replication lag

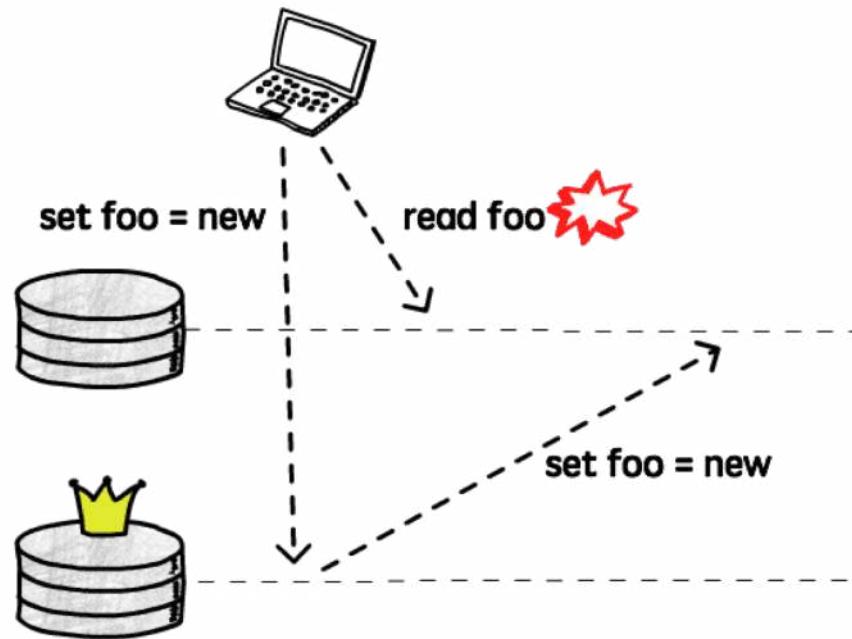
- During the replication lag, data are not consistent, they will become consistent later
- **Eventual consistency**
- Eventual consistency is not always a problem – it depends on the replication lag and the reference application
- Example
 - You post a new picture on Facebook
 - Your friend is able to see it 30 seconds after you posted it
 - Is it a problem? Probably not

Read conflict

- There are cases where the replication lag and related read conflicts are a real problem and generate specific types of conflicts
- Relevant types of conflicts:
 - Read your write conflict
 - Monotonic read conflict
 - ...
- For each type of conflict, a related consistency level is defined if the system is able to avoid it

Read your write conflict

- If the user views the data shortly after making a write, the new data may not yet have reached the replica



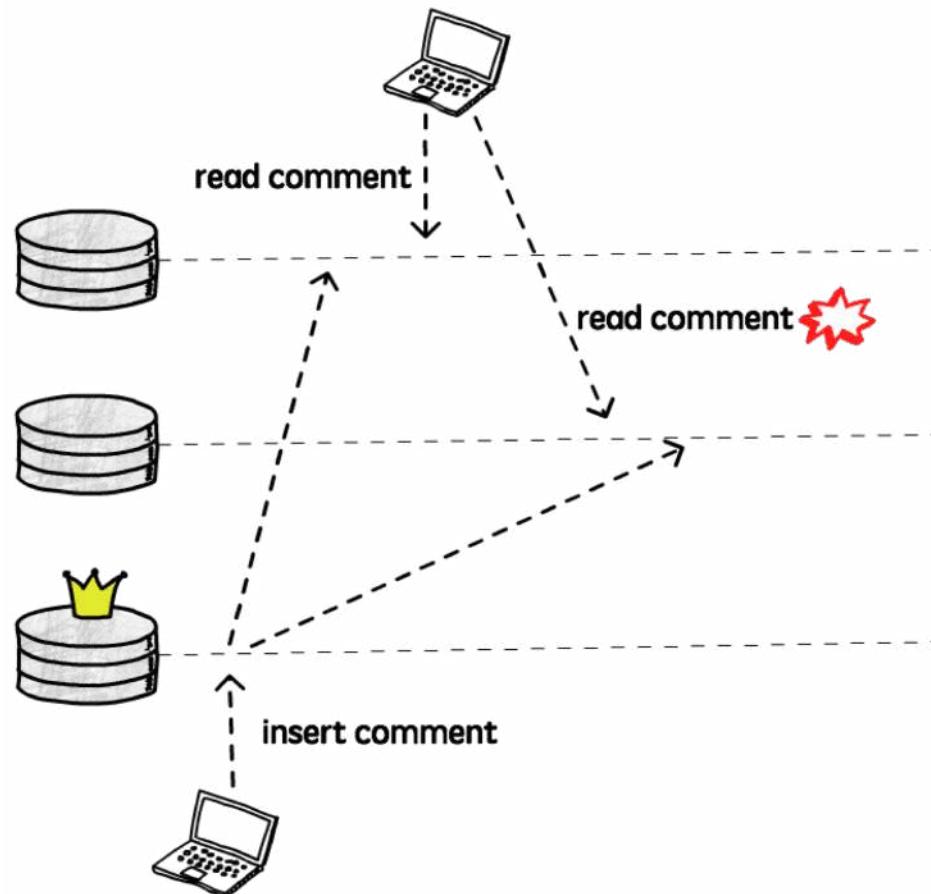
Read your write consistency

A client never reads the database in a state it was before it performed a write

1. When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
2. Read from the leader for a certain time window after the update (but defining the right time window is not simple)
3. More sophisticated solutions based on timestamps

Monotonic read conflict

- After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time



Monotonic read consistency

If you make several reads to a given value, all the successive reads will be at least as recent as the previous one.

Time never moves backwards

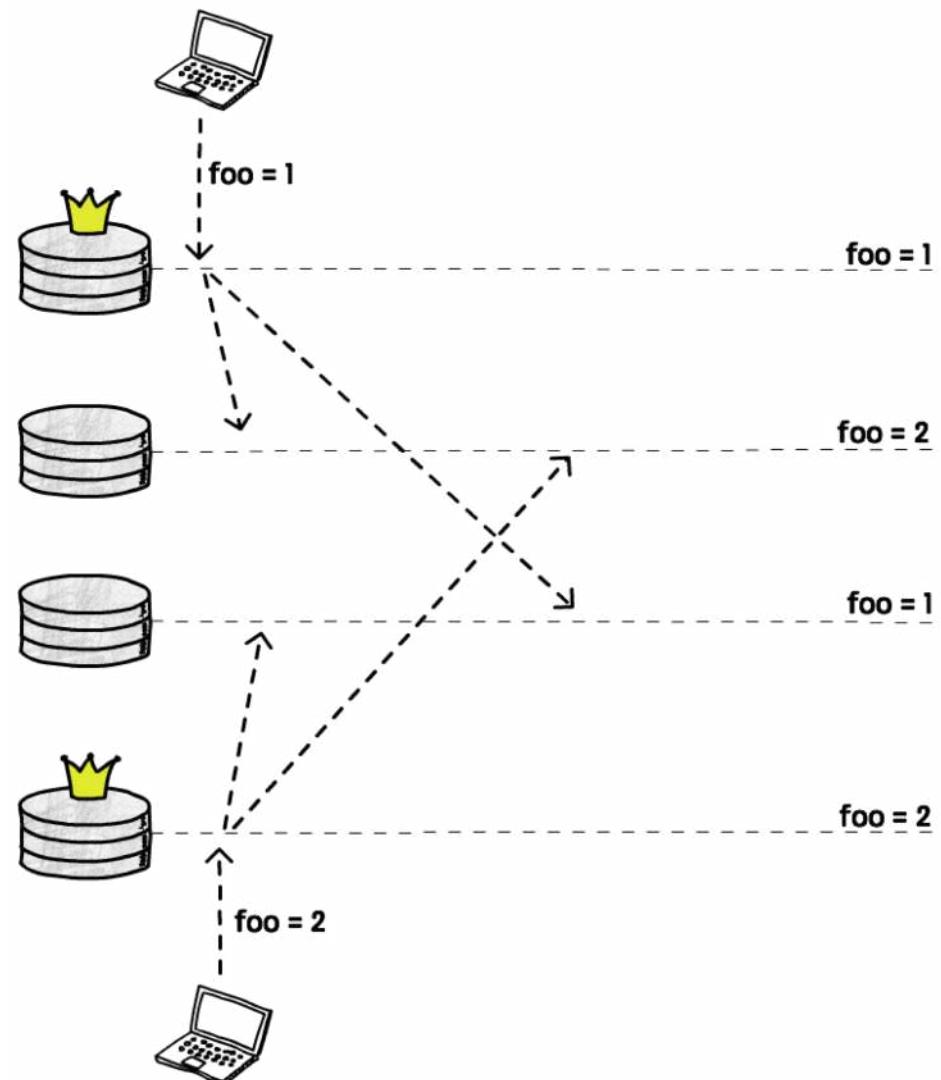
1. Each user always makes reads from the same replica
(different users can read from different replicas)
2. More sophisticated solutions based on timestamps

Read conflicts and leader-less replication

- Read conflicts might happen also under leader-less replication
- Quorums appear to guarantee that a read returns at least once the latest written value
- in practice this is not so simple
 - If a write happens concurrently with a read, the write may be reflected on only some of the replicas
 - In this case, the read may return the old or the new value
- guarantees like read your writes and monotonic reads are not always achieved

Write conflicts (concurrent writes)

- Multi-leader and leaderless replication protocols allow several clients to concurrently write to the same item through two different leaders
- *Events may arrive in a different order at different nodes, due to variable network delays and partial failures*
- Write conflicts may occur even if quorums are used

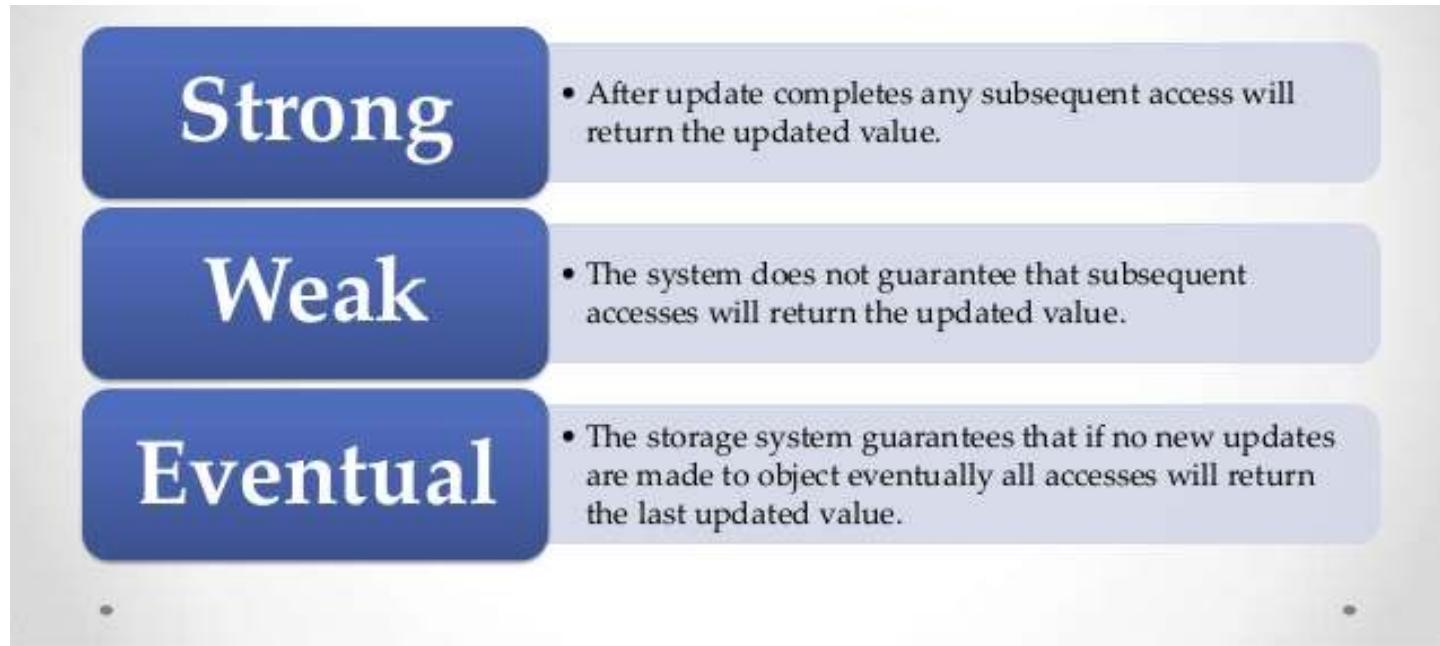


Write conflict – consistency

Avoid multiples values for the same replica, due to concurrent writes

1. No concurrent writes
2. More sophisticated solutions based on timestamps
3. In leader-less replication protocols, write some system-level custom conflict resolution code, when a write/read operation is executed, execute the code and choose one of the replicas as the correct one

Replication and consistency: recap



Read-your-write consistency

Eventual consistency but no conflict of type read-your-own-write

Monotonic read consistency

Eventual consistency but no conflict of type monotonic read

Monotonic write consistency

Eventual consistency but no write conflicts for write operations executed by the same process

...

Take away

- Synchronous vs asynchronous protocols
- Three main approaches, depending on how many leaders are available
- Read conflicts can arise with asynchronous protocols
- Write conflicts can arise when more than one node accepts writes
- Eventual consistency is the overall guarantee we want to achieve
- In general, trade-offs around replica consistency, durability, availability, and latency
- Many approaches for transferring write information
- Under read intensive scenarios, single leader could be a good option
- Under write intensive scenarios, multi-leader or leader-less protocols are better options
- Leader-less protocols often used in P2P systems

Faults and reliability: basic issues

Fault Tolerance – Centralized Systems

CENTRALIZED SYSTEMS

- if a program fails for any reason, the simple solution is to abort then restart the process
- either it works or it doesn't, **total failure**
- key concept for guaranteeing reliability: **transaction**

DISTRIBUTED SYSTEMS

- failure is a possibly frequent situation
- restarting is not always a good idea
- **partial failure**
- **transactions** still exist but ACID properties are revised

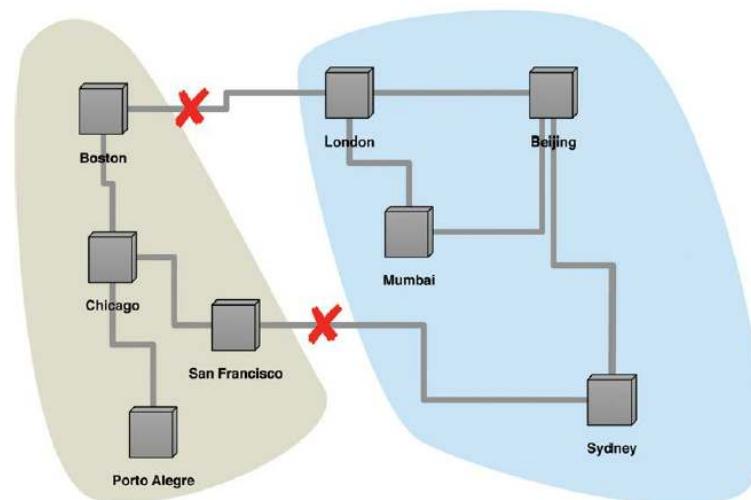
Fault Tolerance

Independence: the task handled by an individual node should be *independent* from those handled by the other components

- In case of failure, simply replace the node with another one
- Independence is best achieved in shared-nothing architectures
- Replication helps

Fault Tolerance

- Network partition/network fault: one part of the network is cut off from the rest due to a network fault
- Such faults can always occur and distributed data systems needs to be able to handle them



Issues

- How to detect that a system met a failure?
 - the client can wait for the failed node to come back (at least for a given amount of time, **timeout**)
- How to recover from the failure? (after the timeout)
 - Either abort the process and report an error (**atomic/transactional** behavior, **limited availability**) or
 - the failure is solved (through **node replacement**) and the process can be completed (**more complex** approach)
 - In both cases: **reliability and recovery**, based on **consensus protocols**
 - Systems do everything for us

Consensus protocol: example

Leader election

- In single-leader replication, **leader fault**
- A new leader has to be identified (consensus protocol)
 - predefined successor node
 - choose the node that has the most recent update to minimize data loss
- All write requests must be sent to the new leader
 - Request re-routing
- All nodes need to agree on which node is the leader
 - What happens if the old leader comes back?
 - If there were two leaders, they would both accept writes and their data would diverge, leading to inconsistency and data loss
- Sometimes a manual procedure could be better ...

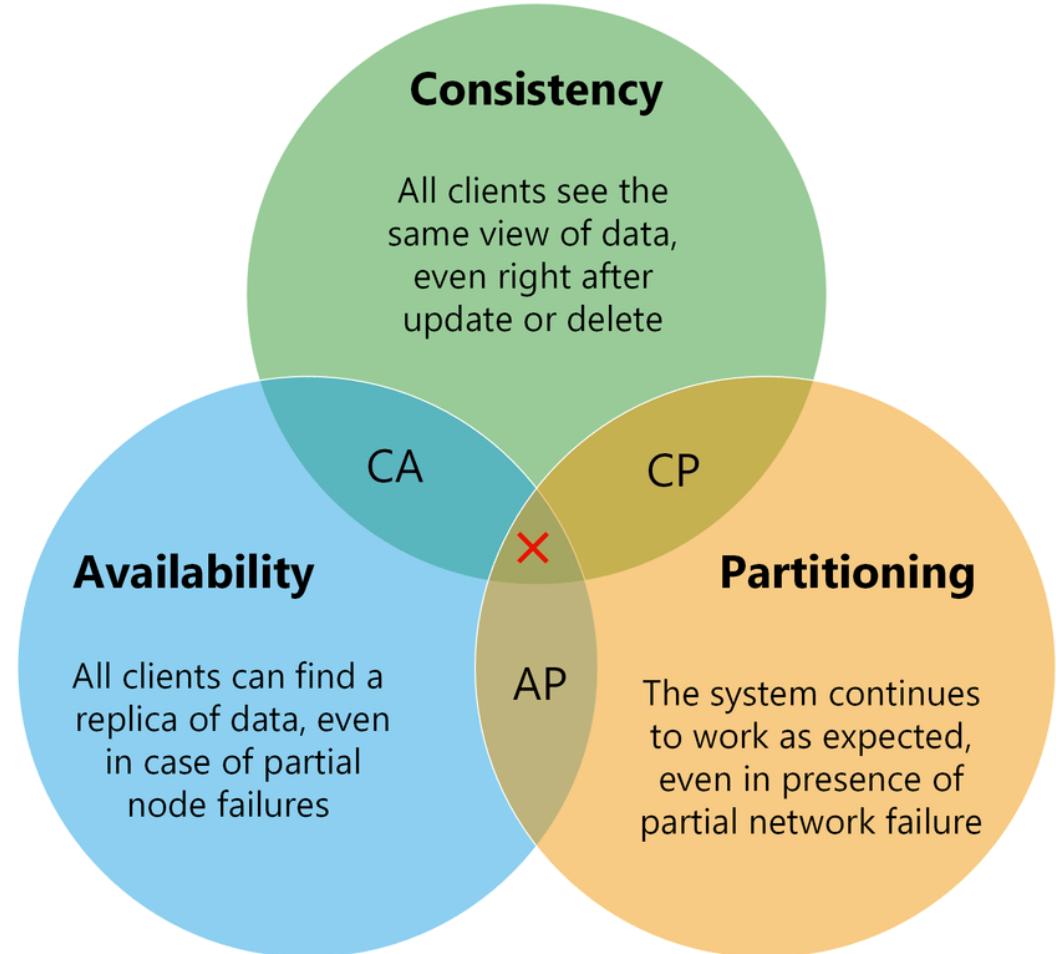
Take away

- Different kind of failures can arise
- A distributed system should be able to recover from failures
- Node independence simplifies failure management: the task handled by an individual node is independent from those handled by the other components
- Task replication is also an useful principle for failure management
- Two main issues: how to detect a fault and how to recover it
- Fault detection is often based on timeout
- Recovery often based on consensus protocols

Putting everything together: the CAP theorem

PUTTING EVERYTHING TOGETHER: THE CAP THEOREM

- Conjecture by Eric Brewer in 2000
- Proposed formal proof by Seth Gilbert and Nancy Lynch in 2002
- No distributed system can simultaneously provide consistency, availability, partition tolerance
- Since partition tolerance is mandatory, the main tradeoff is between consistency and availability



consistency

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

C

CA: available, and consistent,
unless there is a partition.

A

Availability

P

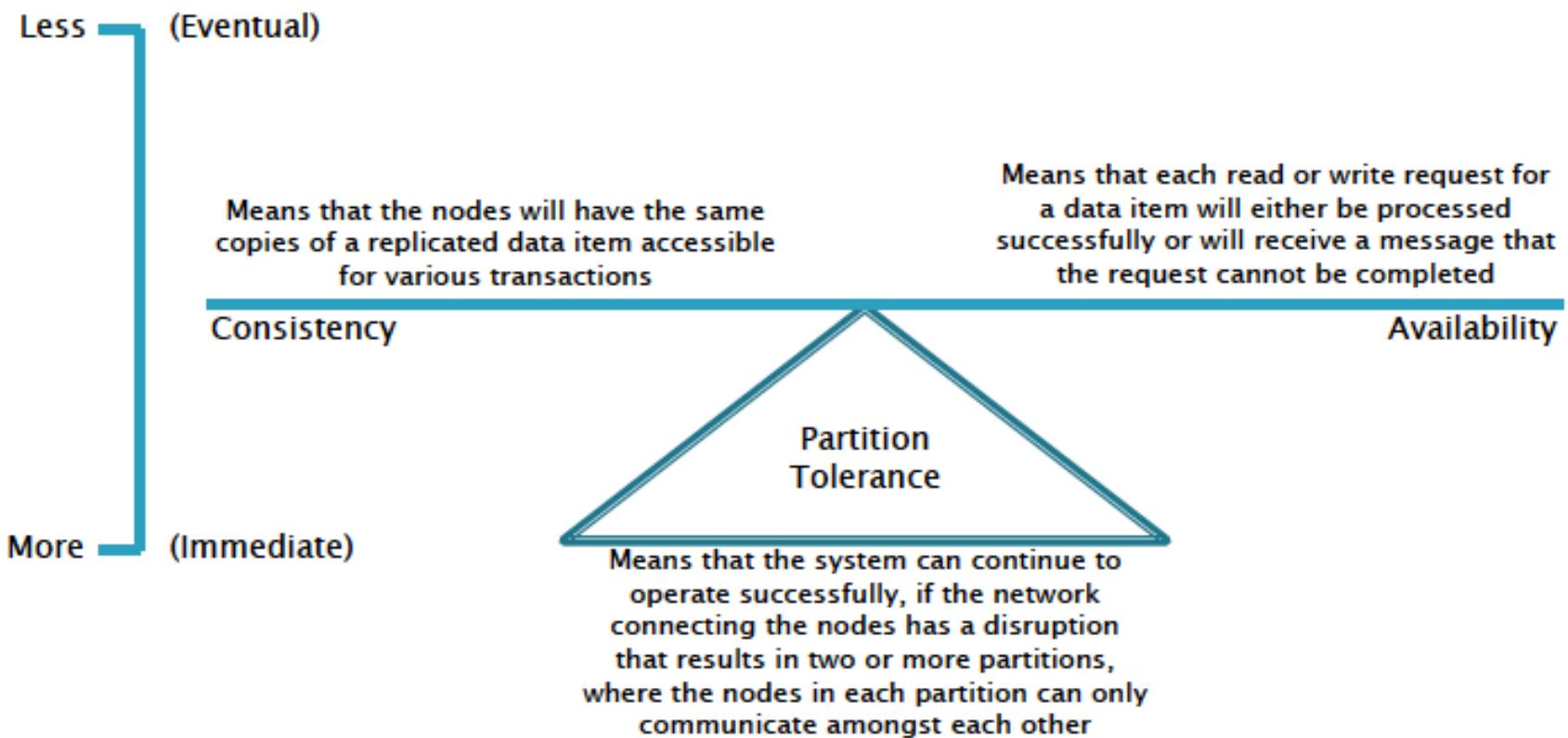
Partition-resilience

AP: a reachable replica provides
service even in a partition, but may
be inconsistent.

Claim: every distributed
system is on one side of the
triangle.

CP: always consistent, even in a
partition, but a reachable replica may
deny service without agreement of the
others (e.g., quorum).

Consistency or Availability?



Take away

- The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency
- Partition tolerance is mandatory in distributed systems, leading to either CP or AP systems
- Different levels of consistency and availability are however possible

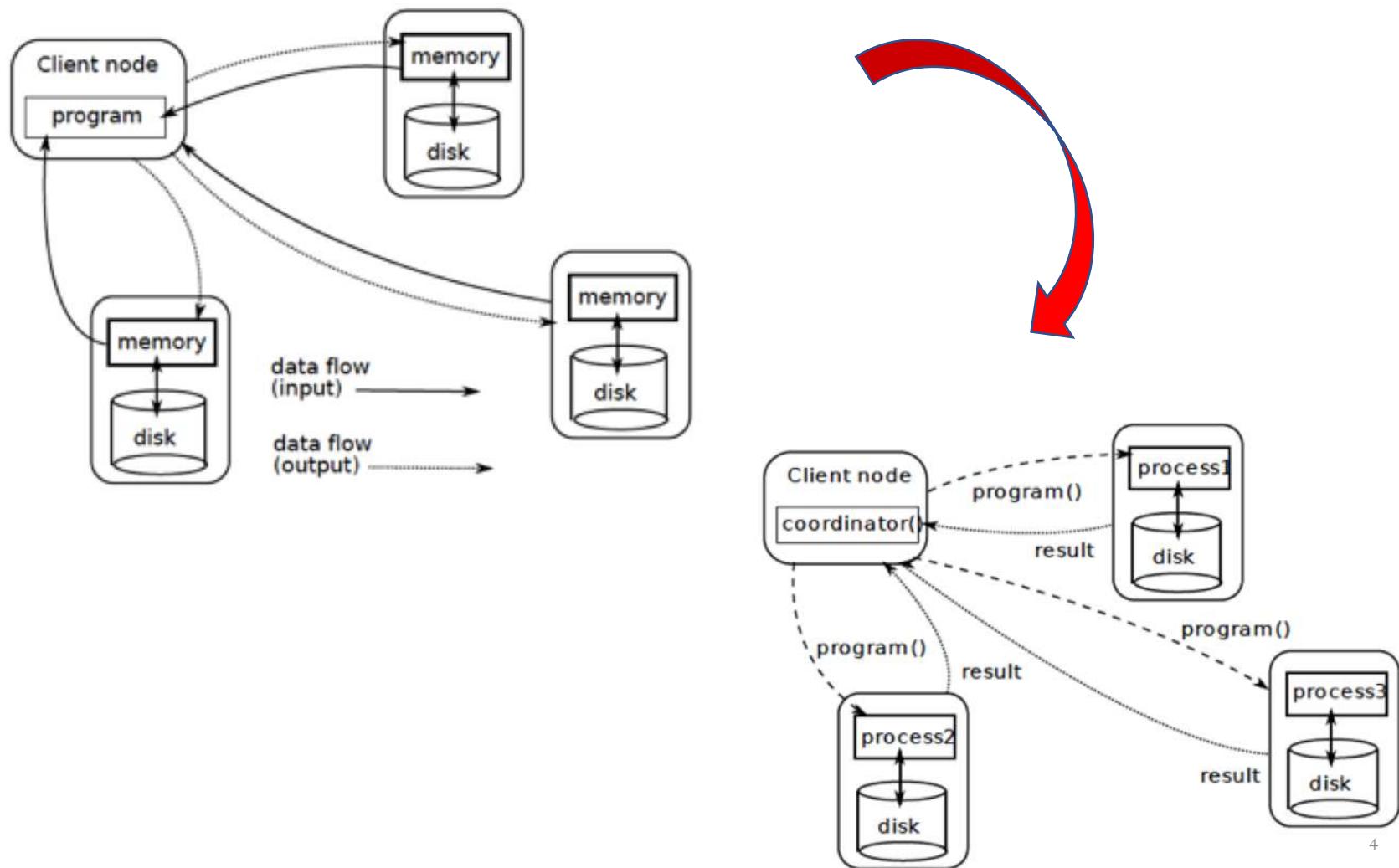
Frameworks for large scale data processing

Introduction

Large-scale distributed (cluster) computing

- Provide a framework that
 - hides the complexity of distribution to developers (load balancing, fault tolerance)
 - provides data storage capabilities
 - focuses on the fundamental nuggets of the computation, supporting new computation models, for processing data in parallel
- Principles
 - Scale “out”, not “up”
 - Move processing to data (**data locality**)
 - No focus on random access but on **batch** access
 - Tailored to **analytical processing**

Data locality



Large-scale data processing frameworks

| | Hadoop Map Reduce | Spark |
|--------------------------|----------------------|-----------------------------------|
| Storage | Disk only | In-memory or on disk |
| Operations | Map and Reduce | Map, Reduce, Join, Sample, etc... |
| Execution model | Batch | Batch, interactive, streaming |
| Programming environments | Java | Scala, Java, R, and Python |

- **MapReduce:** A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks

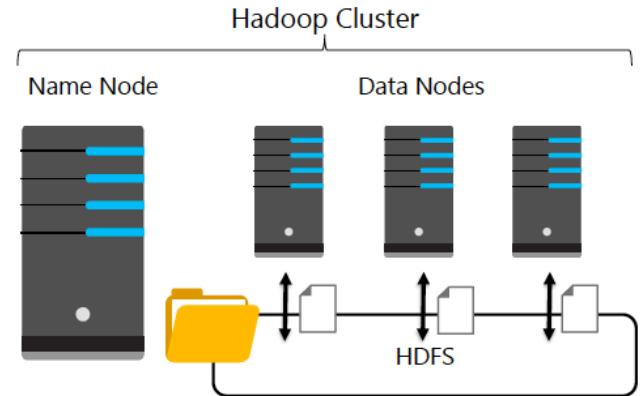
Hadoop

MapReduce implementations

- Google MapReduce
 - Proprietary system
- Hadoop
 - Open-source MapReduce framework (Apache project)
 - Originally developed by Yahoo (but originates from Google MapReduce)

The core of Hadoop

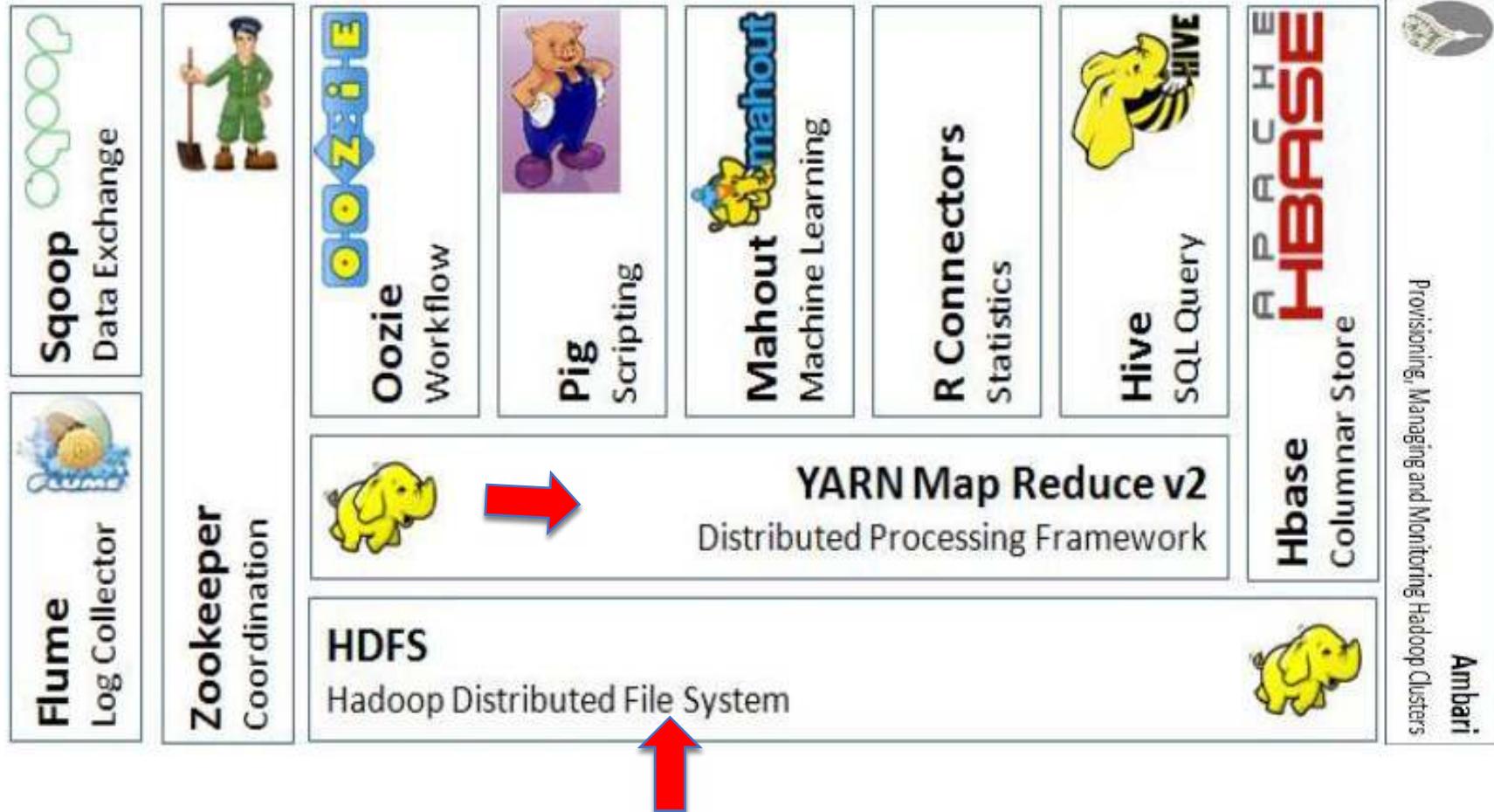
- Master-slave architecture
- **HDFS** (Hadoop Distributed File System)
 - A distributed file system
 - Fault-tolerant
 - Data is replicated with redundancy across the cluster
 - CP: consistent and partition tolerant
- Large-scale data processing infrastructure based on the **MapReduce** programming paradigm
 - Provides a high level abstraction view (programmers do not need to take care of task scheduling and synchronization)
 - Fault-tolerant (node and task failures are automatically managed by the Hadoop system)



Large-scale data processing infrastructure in Hadoop

- Separates the **what** from the **how**
- MapReduce abstracts away the “distributed” part of the problem (scheduling, synchronization,etc.)
 - *Programmers focus on what*
- The distributed part (scheduling, synchronization,etc.) of the problem is handled by the framework
 - *The Hadoop infrastructure focuses on how*

Hadoop ecosystem



Introduction to HDFS

HDFS

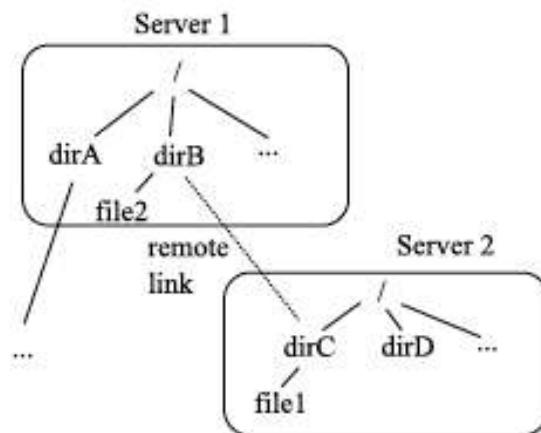
- Large-scale distributed file system
- Simplest approach for large-scale distributed data storage

Reference scenario

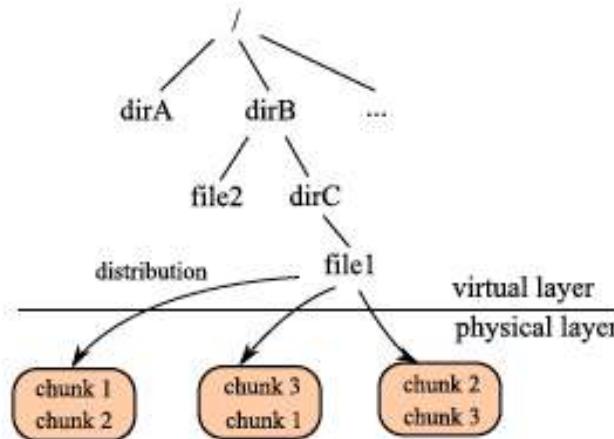
- Batch processing
 - large amount of input data: modest number of huge (multi-gigabytes) files
 - runs a (possibly long lasting) job to process it
 - produces some output data
- Read-intensive scenarios
 - files are written-once, mostly appended to (perhaps concurrently)
 - high number of read requests of the whole file (limited random accesses)

The problem

- Standard Network File System (left part) does not meet scalability requirements (what if file1 gets really big?)



A traditional network file system



A large scale distributed file system

- Distributed File System storage, based on (i) a virtual file namespace, and (ii) **partitioning** of files in (possibly replicated) “chunks”

Data distribution

- **Partitioning**: the architecture works best for very large files (e.g., several Gigabytes), partitioned in large (64-128 MBs) chunks
 - this limits the metadata information to be managed
 - compare with 4-32 KB DBMS pages
 - *block-based approach*, chunks defined based on the record position in the file
- **Replication**: usually, each partition is replicated 3 times
 - nodes holding copies of one chunk are located on different racks
 - *synchronous multi-leader replication protocol*
- Chunk size and the degree of replication can be set by the user

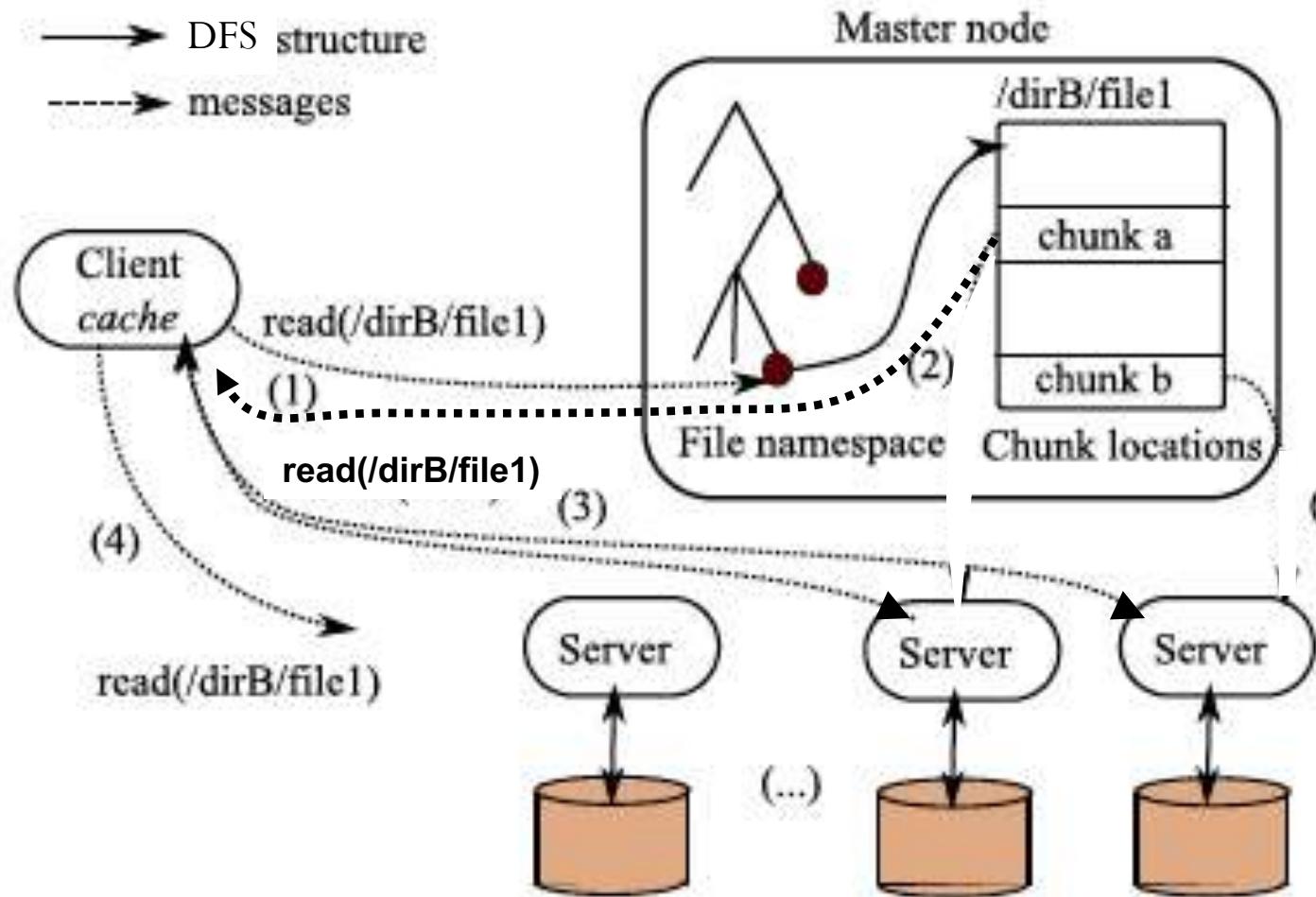
System architecture

- Master/slave architecture
- One **Master node (NameNode)**
 - performs administrative tasks: replication and balancing; garbage collection
 - manages the file system namespace
 - regulates access to files by clients for reads and writes
 - communications with the Master only involve transfer of metadata (limited data transfer)
 - might be replicated for fault tolerance
- Multiple **Slave servers (DataNodes)**
 - store chunks/partitions
 - store and retrieve the blocks when they are told to (by clients or the Master)
- Blocks are themselves stored on standard single-machine file systems
 - DFS lies on top of the standard Operating System stack

Client applications

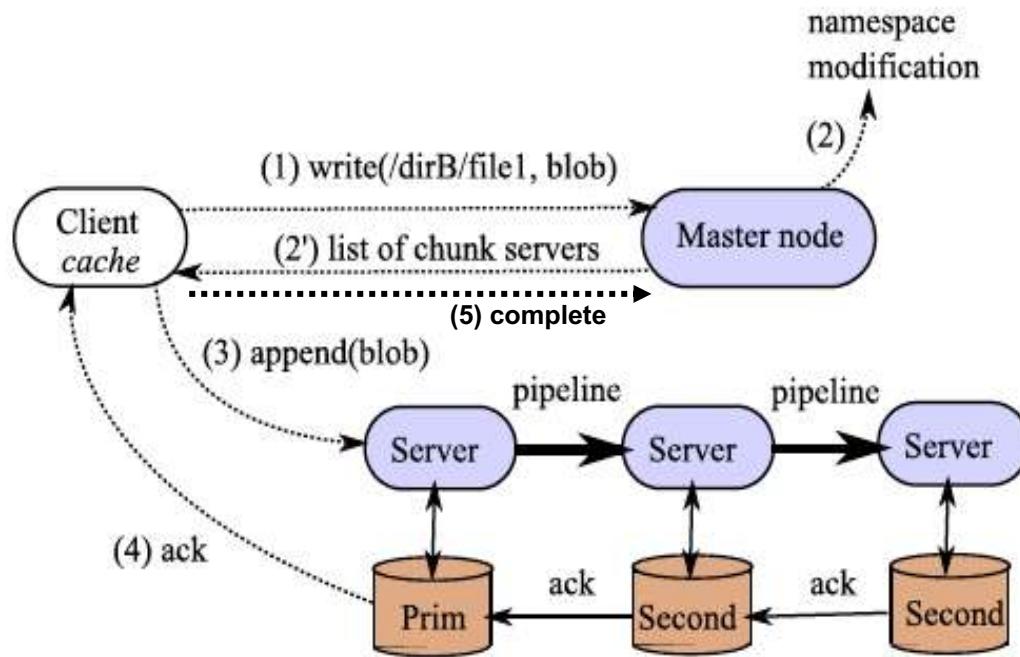
1. File access through specific APIs
2. Talk to the master node to find data/chunk servers associated with the file of interest
3. Connect to the selected chunk servers to access data
4. The Client keeps in its *cache* the addresses of the nodes accessed in the past, this knowledge can be used for subsequent accesses
 - improves scalability

File read



File write (append)

- non-concurrent append() operation



- Multi leader, synchronous (one write on a given chunk at the time)
- The master returns the servers addresses ordered with respect to their distance from the client

Recovery management

- Logging at both Master and Server sites
- The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs (for *availability*)
- The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace
- If the NameNode machine fails, manual intervention is usually required

Summary

| Feature | In HDFS Data Stores |
|---------------------|--|
| Reference scenarios | Analytical |
| Architecture | Master-slave |
| Partitioning | Block-based |
| Replication | Multi-leader, synchronous |
| Consistency | Strong |
| Availability | Limited |
| Fault tolerance | Master-slave architecture, replication of the master node |
| CAP theorem | CP |

Introduction to MapReduce

Typical Large-scale Problem

- Iterate over a large number of records in parallel
- Extract something of interest from each iteration
- Shuffle and sort intermediate results of different concurrent iterations
- Aggregate intermediate results
- Generate final output

MAP

REDUCE

Key idea: provide a functional abstraction for these two operations

MapReduce framework

An example: Word Count

- Input
 - A large textual file of words
- Problem
 - Count the number of times each distinct word appears in the file
- Output
 - A list of pairs <word, number of occurrences in the input file>

An example: Word Count

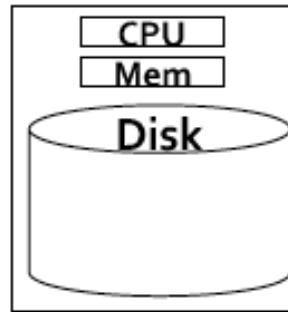
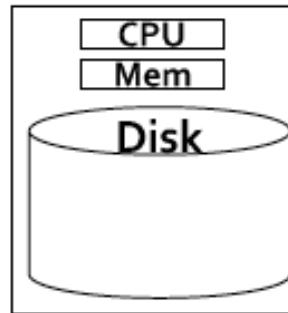
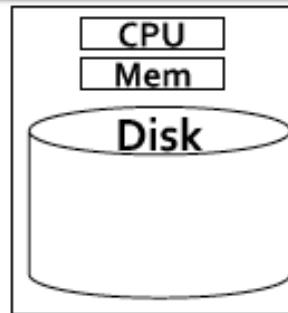
- Case 1: Entire file fits in main memory
 - A traditional single node approach is probably the most efficient solution in this case
 - The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed

An example: Word Count

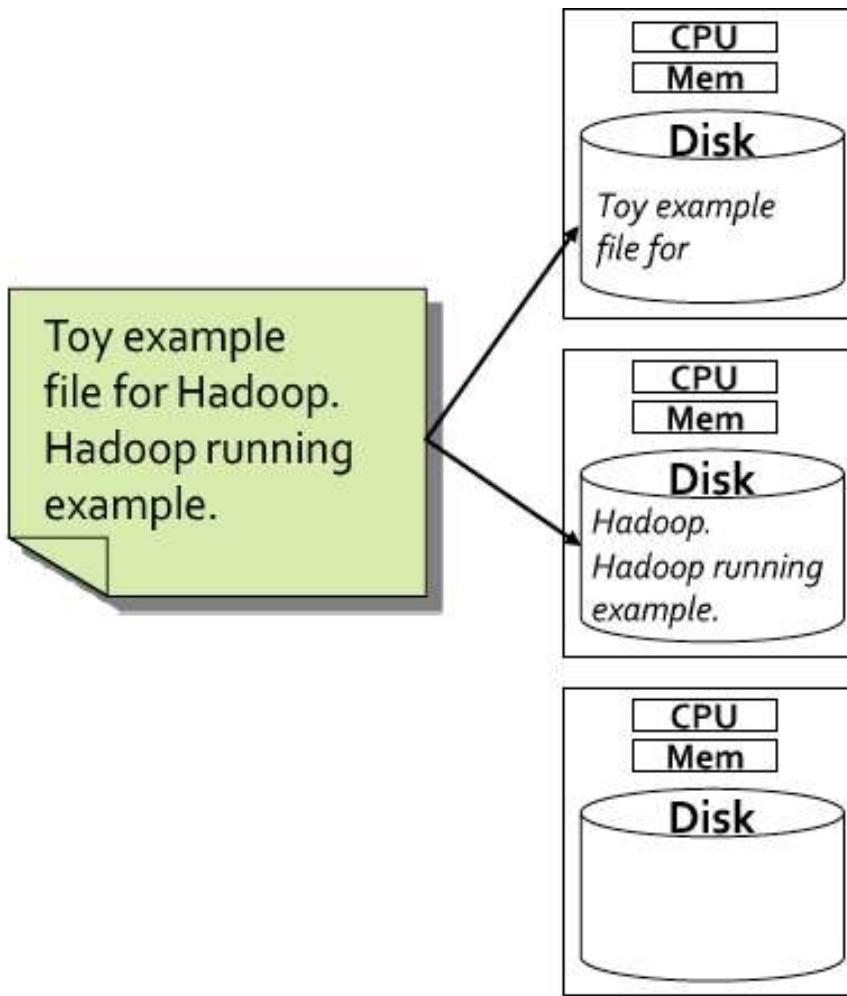
- Case 2: File too large to fit in main memory
 - Rely on a cluster
 - Distributed file system (e.g., HDFS) useful in this case: file partitioning and replication
 - How can we split this problem in a set of (almost) independent sub-tasks and execute it on the cluster?
- Suppose that
 - The cluster has 3 nodes
 - The content of the input file is
 - "Toy example file for Hadoop. Hadoop running example."
 - The input file is split in two chunks (number of replicas =1)

An example: Word Count – case 2

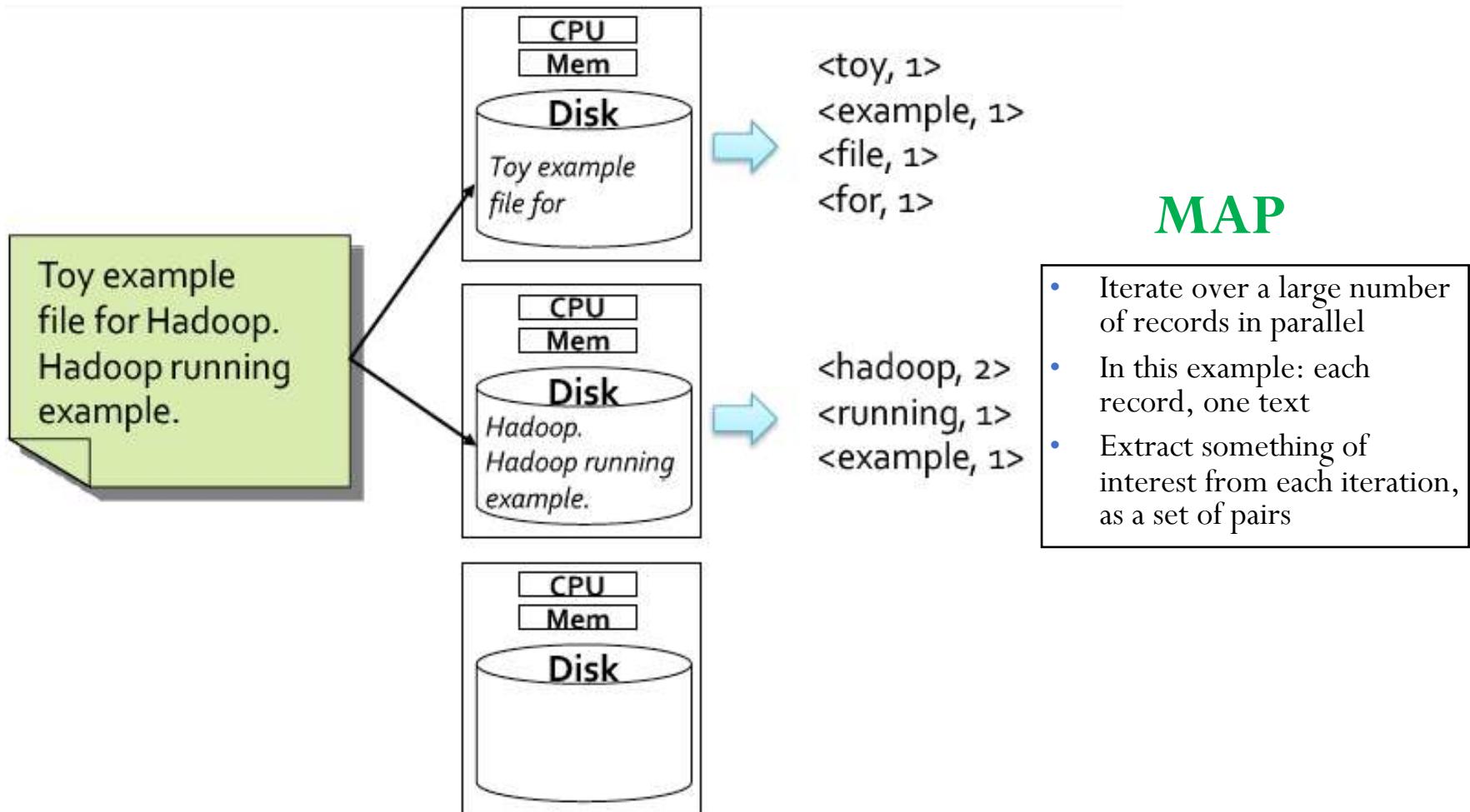
Toy example
file for Hadoop.
Hadoop running
example.



An example: Word Count – case 2



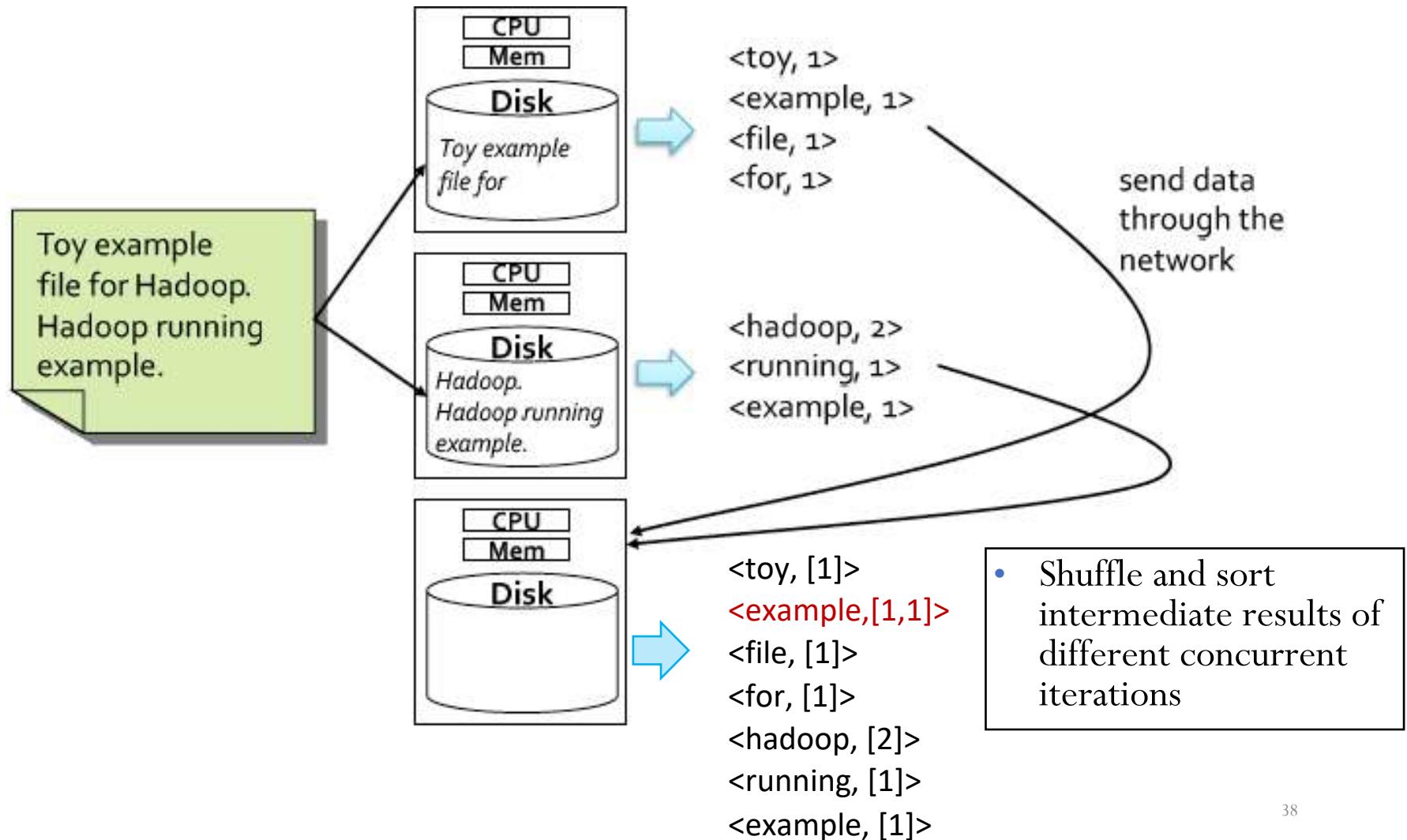
An example: Word Count – case 2



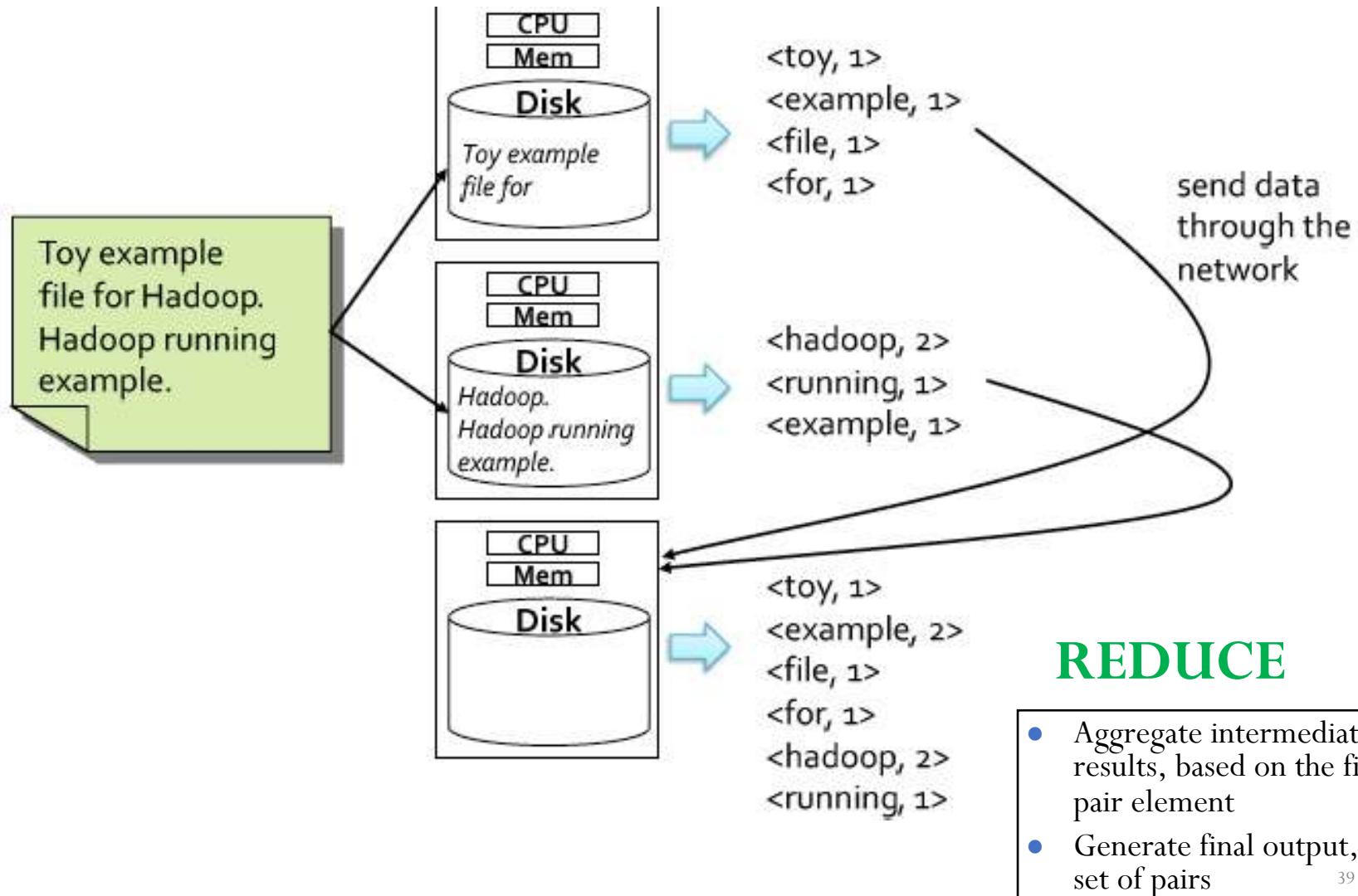
An example: Word Count – case 2

- The problem can be easily parallelized
 - Each server processes its chunk of data and counts the number of times each word appears in its chunk
 - Each server can perform it independently
- Synchronization is not needed in this phase

An example: Word Count – case 2



An example: Word Count – case 2



An example: Word Count – case 2

- Each server sends its local (partial) list of pairs <word, number of occurrences in its chunk> to a server that is in charge of aggregating local results and computing the global list/global result
- The server in charge of computing the global result needs to receive all the local (partial) results to compute and emit the final list
- A simple synchronization operation is needed in this phase

An example: Word Count – a more realistic case - complexity

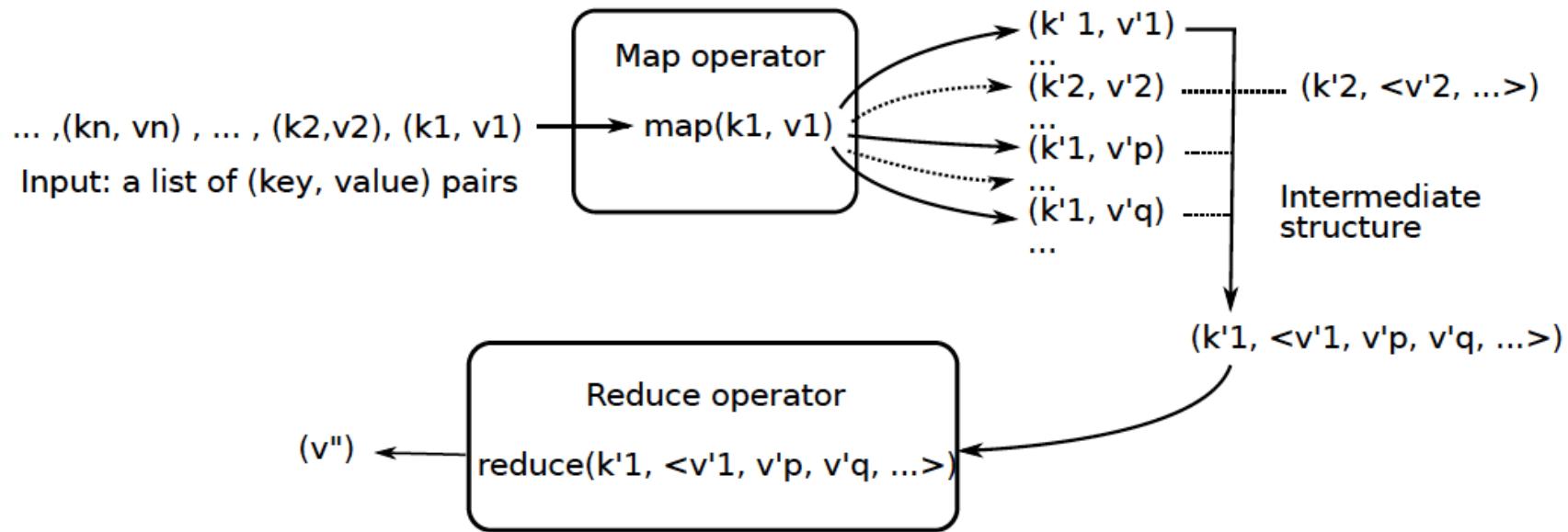
- Now suppose that
 - The file size is 100 GB and the number of distinct words occurring in it is at most 1.000
 - The cluster has 101 nodes
 - The file is optimally spread across 100 nodes and each of these servers contains one or many chunks (partitions) of the input file (1 GB, corresponding to 1/100 of the original file), stored in a distributed file system
- Each server reads 1GB of data from its local hard drive
 - Few seconds
- Each local list is composed of at most 1.000 pairs (because the number of distinct words is 1.000)
 - Few MBs
- The maximum amount of data sent on the network is $100 \times$ size of local list (number of servers \times local list size)
 - Some MBs

MapReduce programming paradigm in brief

- The MapReduce programming paradigm is based on the basic concepts of functional programming
- The programmer defines the program logic as two functions:
 - **Map**: do something to everything in a list
 - **Reduce**: combine/aggregate results of a list in some way
- The MapReduce environment takes in charge distribution aspects
- A complex program can be decomposed as a chain of Map and Reduce tasks

MapReduce programming paradigm in brief

Important: each pair, at each phase, is processed **independently** from the other pairs.



Network and distribution are transparently managed by the MapReduce environment.

MapReduce programming paradigm in brief

- The **Map phase** can be viewed as a transformation over each element of a data set
 - This transformation is a function m defined by the designer
 - Each application of m happens in **isolation**, so it can be parallelized
- The **Reduce phase** can be viewed as an aggregate operation
 - The aggregate function is a function r defined by the designer
 - Also the reduce phase can be performed in parallel since each group of key-value pairs with the same key can be processed by a distinct node
- The **shuffle and sort** phase is always the same
 - i.e., group the output of the map phase by key
 - it does not need to be defined by the designer

Back to the example – word count

- Input
 - A textual file (i.e., a list of words)
- Problem
 - Count the number of times each distinct word appears in the file
- Output
 - A list of pairs
 - <word, number of occurrences in the input file>
- *Preliminary issue:*
 - The content of each partition has to be interpreted as a set of (key-value) pairs

Back to the example – word count

Each partition = one (key, value) pair

Value: a list of words

Key: identifier of the list of words (e.g., the position of the text in the file)

```
map(key, value)
    //key: identifier, value: a list of words
    for each distinct word w in value
        emit(w, count(w,value))
```

```
reduce(key, values):
    // key: a word; value: a list of integers
    occurrences = 0
    for each c in values:
        occurrences = occurrences + c

    emit(key, occurrences)
```

Back to the example – word count

Each partition = one (key, value) pair

Value: a list of words

Key: identifier of the list of words (e.g., the position of the text in the file)

```
map(key, value)
```

```
//key: identifier, value: a list of words
```

```
for each word w in value
```

```
emit(w,1)
```

```
reduce(key, values):
```

```
// key: a word; value: a list of integers
```

```
occurrences = 0
```

```
for each c in values:
```

```
    occurrences = occurrences + c
```

```
emit(key, occurrences)
```

LESS
EFFICIENT

...WHY?

Back to the example – word count

Each partition = many (key, value) pairs

Value: one word

Key: identifier of the word (e.g., the position of the word in the partition)

```
map(key, value)
    //key: identifier, value: word
    emit(value,1)
```

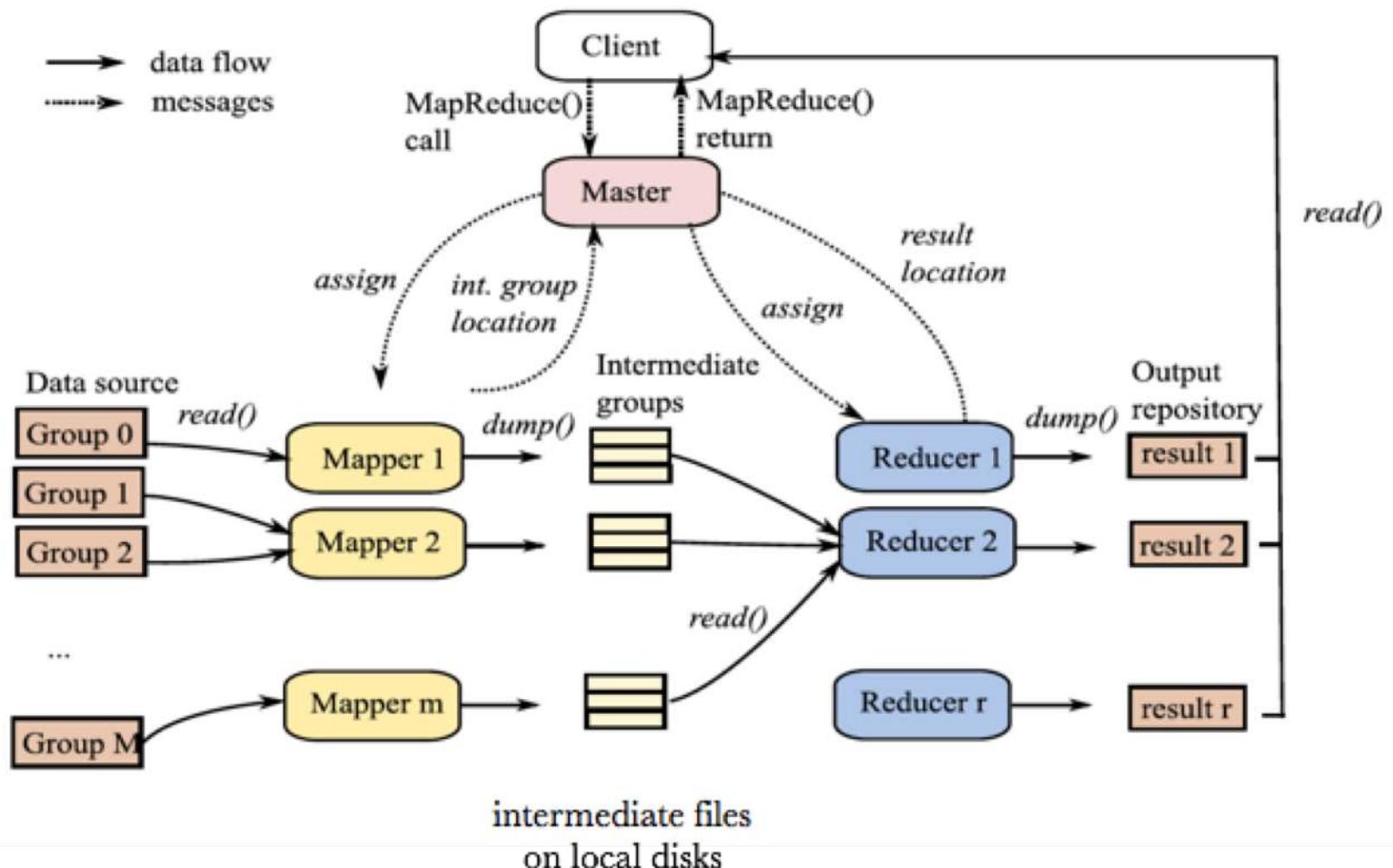
```
reduce(key, values):
    // key: a word; value: a list of integers
    occurrences = 0
    for each c in values:
        occurrences = occurrences + c

    emit(key, occurrences)
```

MapReduce data structures

- Key-value pair is the basic data structure in MapReduce
 - keys and values can be: integers, float, strings, ...
 - they can also be (almost) arbitrary data structures defined by the designer
- Both input and output of a MapReduce program are lists of key-value pairs
- *The design of a MapReduce program requires to impose the key-value structure on the input and output data sets*
 - for records in a file, input keys may correspond to their position in the file

Processing map reduce job



Spark

MapReduce limitations: programming overhead

- MapReduce is great for large-data processing
- MapReduce is a low level programming approach
- Very powerful but not easy from a programming point of view
 - writing Java programs (as in Hadoop) for everything is complex, verbose and slow
 - not everyone wants to (or can) write Java code

Example: MapReduce code

```

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<Text, Text, LongWritable, Text, Text> {
        public void map(Text k, Text val,
                       OutputCollector<Text, Text> oc,
                       Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(0, firstComma);
            String key = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }

    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {
        public void map(LongWritable k, Text val,
                       OutputCollector<Text, Text> oc,
                       Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }

    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {
        public void reduce(Text key,
                           Iterator<Text> iter,
                           OutputCollector<Text, Text> oc,
                           Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
            store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
                    first.addValue(value.substring(1));
                else second.addValue(value.substring(1));
            }
        }
    }

    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, LongWritable> {
        public void map(
            Text k,
            Text val,
            OutputCollector<Text, LongWritable> oc,
            Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String url = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore.
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }

    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
        Writable> {
        public void reduce(
            Text key,
            Iterator<LongWritable> iter,
            OutputCollector<WritableComparable, Writable> oc,
            Reporter reporter) throws IOException {
            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }
            oc.collect(key, new LongWritable(sum));
        }
    }

    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
        Text> {
        public void map(
            WritableComparable key,
            Writable val,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }

    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {
        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {
            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }

    public static void main(String[] args) throws IOException {
        JobConf ip = new JobConf(MRExample.class);
        ip.setJobName("Load Pages");
        ip.setInputFormat(TextInputFormat.class);

```

18 to 25
ip.addJob(loadPages);
ip.addJob(loadUsers);
je.addJob(joinJob);
je.addJob(groupJob);
je.addJob(limit);
je.run();

```

        reporter.setStatus("OK");
    }

    // Do the cross product and collect the values
    for (String s1 : first) {
        for (String s2 : second) {
            String outval = key + "," + s1 + "," + s2;
            oc.collect(null, new Text(outval));
            reporter.setStatus("OK");
        }
    }
}

public static class LoadJoined extends MapReduceBase
implements Mapper<Text, Text, LongWritable> {
    public void map(
        Text k,
        Text val,
        OutputCollector<Text, LongWritable> oc,
        Reporter reporter) throws IOException {
        // Do the cross product and collect the values
        for (String s1 : first) {
            for (String s2 : second) {
                String outval = key + "," + s1 + "," + s2;
                oc.collect(null, new Text(outval));
                reporter.setStatus("OK");
            }
        }
    }
}

public static class LoadAndFilterUsers extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, Text> {
    public void map(
        LongWritable k, Text val,
        OutputCollector<Text, Text> oc,
        Reporter reporter) throws IOException {
        // Pull the key out
        String line = val.toString();
        int firstComma = line.indexOf(',');
        String value = line.substring(firstComma);
        int age = Integer.parseInt(value);
        if (age < 18 || age > 25) return;
        String key = line.substring(0, firstComma);
        Text outKey = new Text(key);
        // Prepend an index to the value so we know which file
        // it came from.
        Text outVal = new Text("2" + value);
        oc.collect(outKey, outVal);
    }
}

public static class Join extends MapReduceBase
    implements Reducer<Text, Text, Text, Text> {
    public void reduce(Text key,
                      Iterator<Text> iter,
                      OutputCollector<Text, Text> oc,
                      Reporter reporter) throws IOException {
        // For each value, figure out which file it's from and
        store it
        // accordingly.
        List<String> first = new ArrayList<String>();
        List<String> second = new ArrayList<String>();

        while (iter.hasNext()) {
            Text t = iter.next();
            String value = t.toString();
            if (value.charAt(0) == '1')
                first.addValue(value.substring(1));
            else second.addValue(value.substring(1));
        }
    }
}

public static class LoadJoined extends MapReduceBase
    implements Mapper<Text, Text, LongWritable> {
    public void map(
        Text k,
        Text val,
        OutputCollector<Text, LongWritable> oc,
        Reporter reporter) throws IOException {
        // Do the cross product and collect the values
        for (String s1 : first) {
            for (String s2 : second) {
                String outval = key + "," + s1 + "," + s2;
                oc.collect(null, new Text(outval));
                reporter.setStatus("OK");
            }
        }
    }
}

public static class ReduceUrls extends MapReduceBase
    implements Reducer<Text, LongWritable, WritableComparable,
    Writable> {
    public void reduce(
        Text key,
        Iterator<LongWritable> iter,
        OutputCollector<WritableComparable, Writable> oc,
        Reporter reporter) throws IOException {
        long sum = 0;
        while (iter.hasNext()) {
            sum += iter.next().get();
            reporter.setStatus("OK");
        }
        oc.collect(key, new LongWritable(sum));
    }
}

public static class LoadClicks extends MapReduceBase
    implements Mapper<WritableComparable, Writable, LongWritable,
    Text> {
    public void map(
        WritableComparable key,
        Writable val,
        OutputCollector<LongWritable, Text> oc,
        Reporter reporter) throws IOException {
        oc.collect((LongWritable)val, (Text)key);
    }
}

public static class LimitClicks extends MapReduceBase
    implements Reducer<LongWritable, Text, LongWritable, Text> {
    int count = 0;
    public void reduce(
        LongWritable key,
        Iterator<Text> iter,
        OutputCollector<LongWritable, Text> oc,
        Reporter reporter) throws IOException {
        // Only output the first 100 records
        while (count < 100 && iter.hasNext()) {
            oc.collect(key, iter.next());
            count++;
        }
    }
}

public static void main(String[] args) throws IOException {
    JobConf ip = new JobConf(MRExample.class);
    ip.setJobName("Load Pages");
    ip.setInputFormat(TextInputFormat.class);
    Path("/user/gates/pages");
    FileOutputFormat.setOutputPath(ip,
        new Path("/user/gates/tmp/indexed_pages"));
    ip.setNumReduceTasks(0);
    Job loadPages = new Job(ip);

    JobConf lfu = new JobConf(MRExample.class);
    lfu.setJobName("Load and Filter Users");
    lfu.setInputFormat(TextInputFormat.class);
    lfu.setOutputFormat(TextOutputFormat.class);
    lfu.setOutputValueClass(Text.class);
    lfu.setMapperClass(LoadAndFilterUsers.class);
    FileInputFormat.addInputPath(lfu, new
    Path("/user/gates/users"));
    FileOutputFormat.setOutputPath(lfu,
        new Path("/user/gates/tmp/filterd_users"));
    lfu.setNumReduceTasks(0);
    Job loadUsers = new Job(lfu);

    JobConf join = new JobConf(MRExample.class);
    join.setJobName("Join Users and Pages");
    join.setInputFormat(TextInputFormat.class);
    join.setOutputValueClass(Text.class);
    join.setMapperClass(IdentityMapper.class);
    join.setReducerClass(Join.class);
    FileInputFormat.addInputPath(join, new
    Path("/user/gates/tmp/filterd_users"));
    FileInputFormat.addInputPath(join, new
    Path("/user/gates/tmp/indexed_pages"));
    FileOutputFormat.setOutputPath(join, new
    Path("/user/gates/tmp/joined"));
    join.setNumReduceTasks(0);
    Job joinJob = new Job(join);
    joinJob.addDependingJob(loadPages);
    joinJob.addDependingJob(loadUsers);

    JobConf group = new JobConf(MRExample.class);
    group.setInputFormat(KeyValueTextInputFormat.class);
    group.setOutputValueClass(Text.class);
    group.setMapperClass(LoadJoined.class);
    group.setReducerClass(ReduceUrls.class);
    FileInputFormat.addInputPath(group, new
    Path("/user/gates/tmp/joined"));
    FileOutputFormat.setOutputPath(group, new
    Path("/user/gates/tmp/grouped"));
    group.setNumReduceTasks(0);
    Job groupJob = new Job(group);
    groupJob.addDependingJob(joinJob);

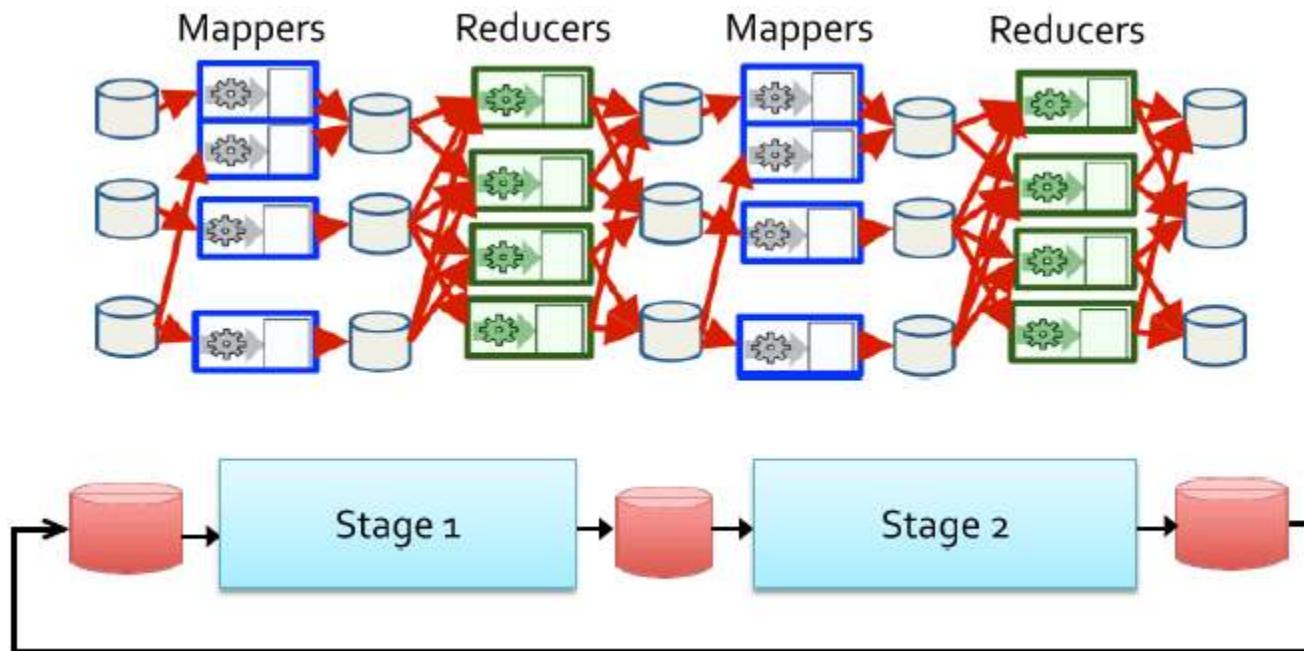
    JobConf top100 = new JobConf(MRExample.class);
    top100.setJobName("Top 100 sites");
    top100.setInputFormat(TextInputFormat.class);
    top100.setOutputKeyClass(LongWritable.class);
    top100.setOutputValueClass(Text.class);
    top100.setMapperClass(LimitClicks.class);
    top100.setReducerClass(LimitClicks.class);
    FileInputFormat.addInputPath(top100, new
    Path("/user/gates/tmp/grouped"));
    FileOutputFormat.setOutputPath(top100, new
    Path("/user/gates/top100.txtfor18to25"));
    top100.setNumReduceTasks(1);
    Job limit = new Job(top100);
    limit.addDependingJob(groupJob);

    JobControl jc = new JobControl("Find top 100 sites for use
    18 to 25");
    jc.addJob(loadPages);
    jc.addJob(loadUsers);
    jc.addJob(joinJob);
    jc.addJob(groupJob);
    jc.addJob(limit);
    jc.run();
}
}

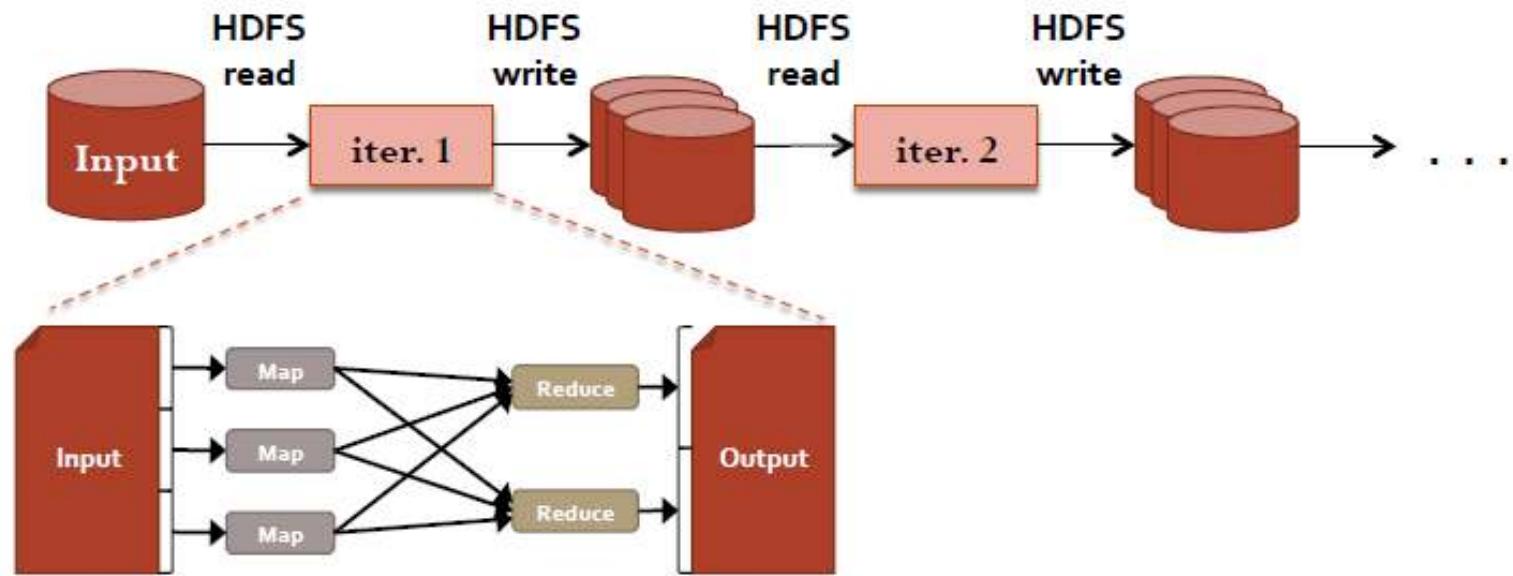
```

MapReduce limitations: iterative jobs

- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage

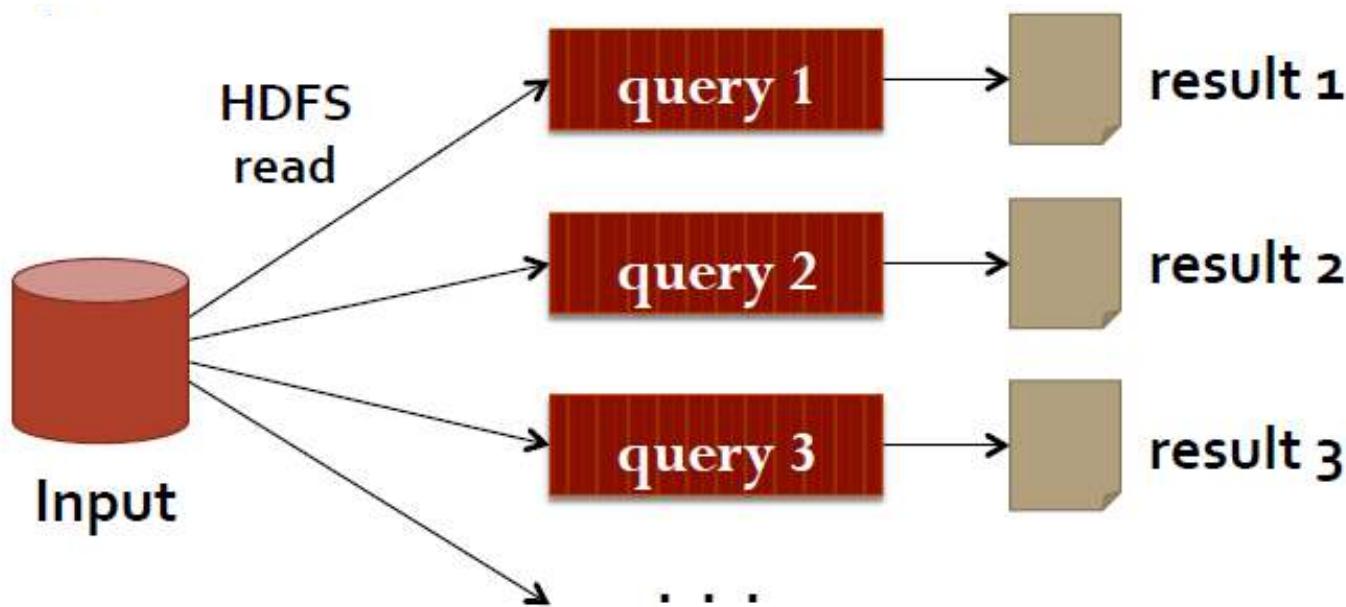


MapReduce limitations: iterative jobs



Iterative jobs are slow due to disk I/O, high communication, and serialization

MapReduce limitations: multiple reads of the same dataset

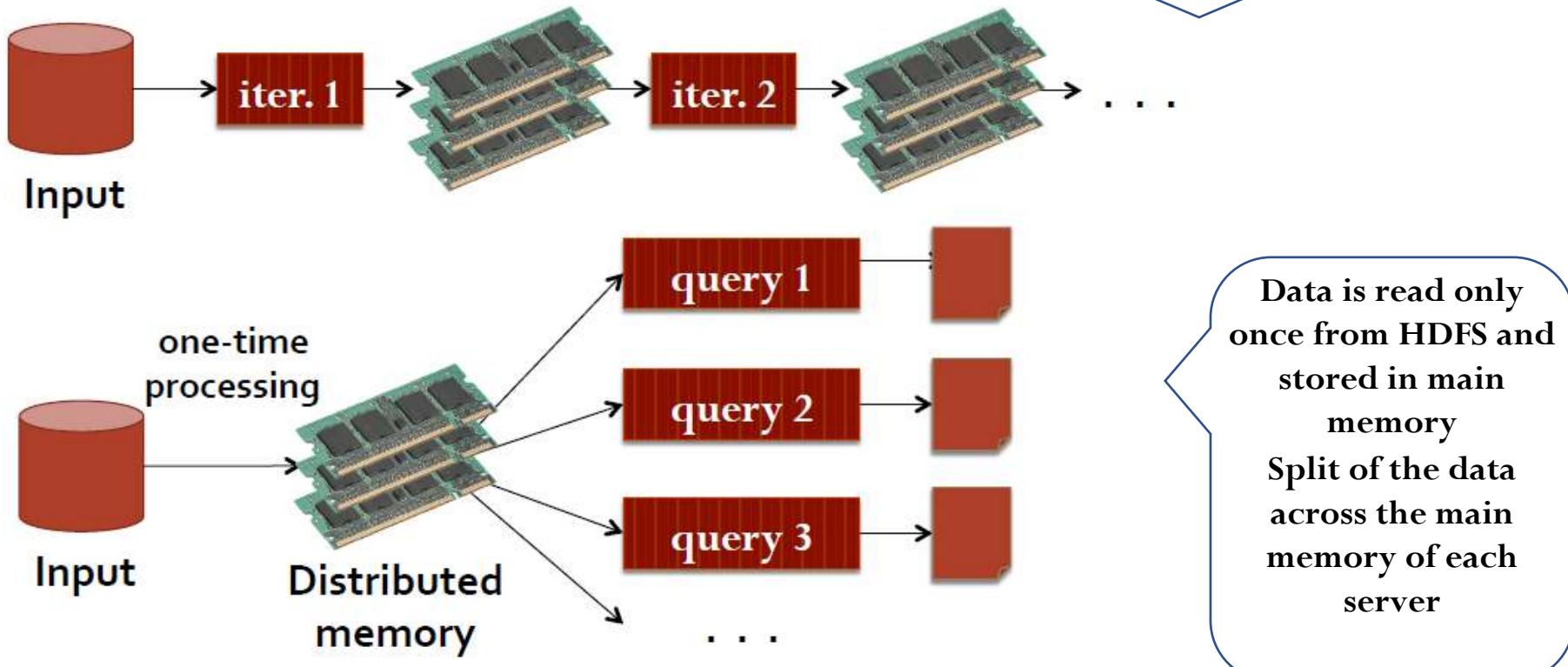


Why Spark?

- Motivation
 - Using MapReduce for complex iterative jobs or multiple jobs on the same data involves lots pf disk I/O
- Opportunity
 - The cost of main memory decreased
 - Hence, large main memories are available in each server
- Solution
 - Keep more data in main memory
 - Enabling data sharing in main memory as a resource of the cluster

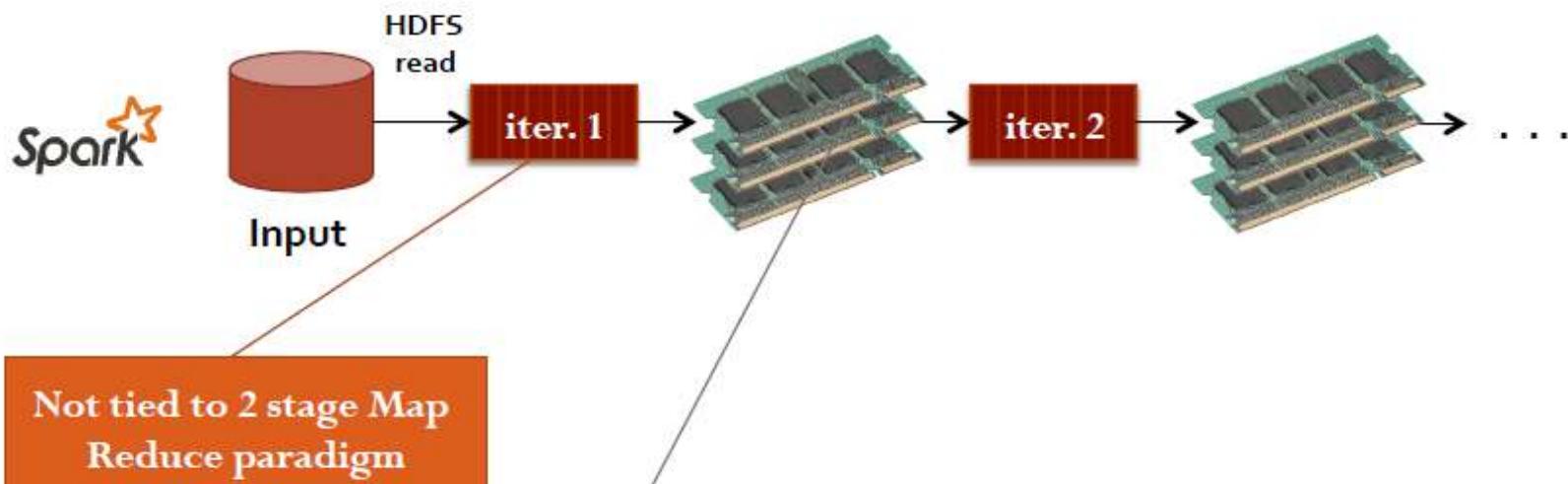


Data sharing in Spark



10-100x faster than network and disk

Spark data flow



1. Extract a working set
2. Cache it
3. Query it repeatedly

Spark data flow

- Everything you can do in Hadoop, you can also do in Spark
- Spark's computation paradigm is not "just" a single MapReduce job, but a multi-stage, in-memory dataflow graph based on **Resilient Distributed Datasets (RDDs)**
- Data are represented as RDDs
 - Partitioned/distributed collections of objects spread across the nodes of a cluster
 - Stored in main memory (when it is possible) or on local disk
- Spark programs are written in terms of operations on RDDs

Spark computing framework

- Provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs
- Manages job scheduling and synchronization
- Manages the split of RDDs in partition and allocates RDDs' partitions in the nodes of the cluster
- Hides complexities of fault-tolerance and slow machines
 - RDDs are automatically rebuilt in case of machine failure (lazy evaluation)

Systems for large-scale data management

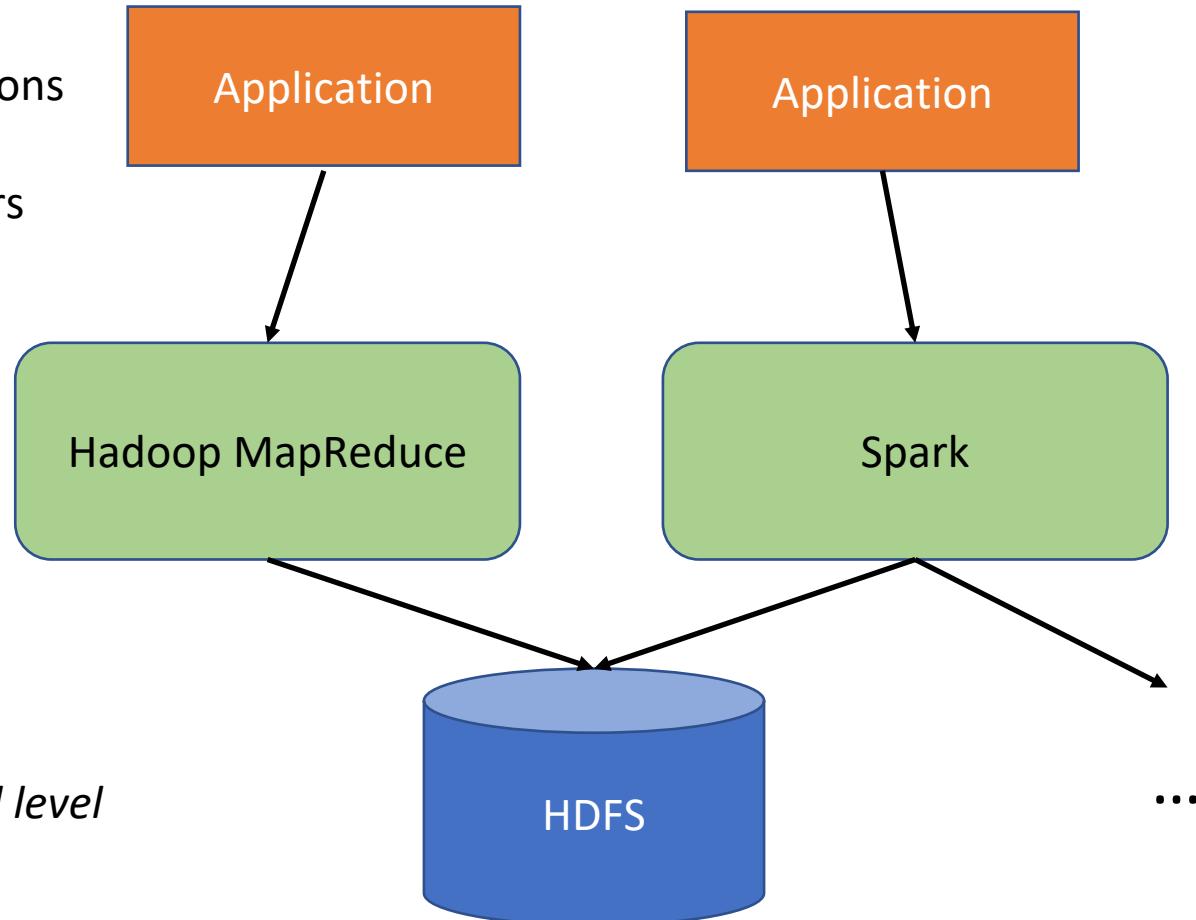
Introduction

Large-scale distributed (cluster) computing

- Work on clusters
- Provide a framework that
 - hides the complexity of distribution to developers (load balancing, fault tolerance)
 - provides proprietary data storage capabilities, in terms of **distributed file systems**
 - focuses on the fundamental nuggets of the computation, supporting **new computation models, for processing data in parallel**
- Principles
 - Scale “out”, not “up”
 - Move processing to data (data locality)
 - Focus on **(read) batch access**
 - Tailored to ***analytical processing***

Large-scale (cluster) computing

logical level
inside applications
in terms of
(key-value) pairs



Is this enough?

- In 70's, relational database management systems have been introduced with the aim of overcoming the limitations of file systems for data management

Can the (relational) data management system technology be a reference technology also for large-scale data management?

Relational DBMSs (RDBMSs)

Mostly standard

Clear separation between logical (relational model) and physical levels (files)

Declarative languages at the logical level (SQL)

Well established technology

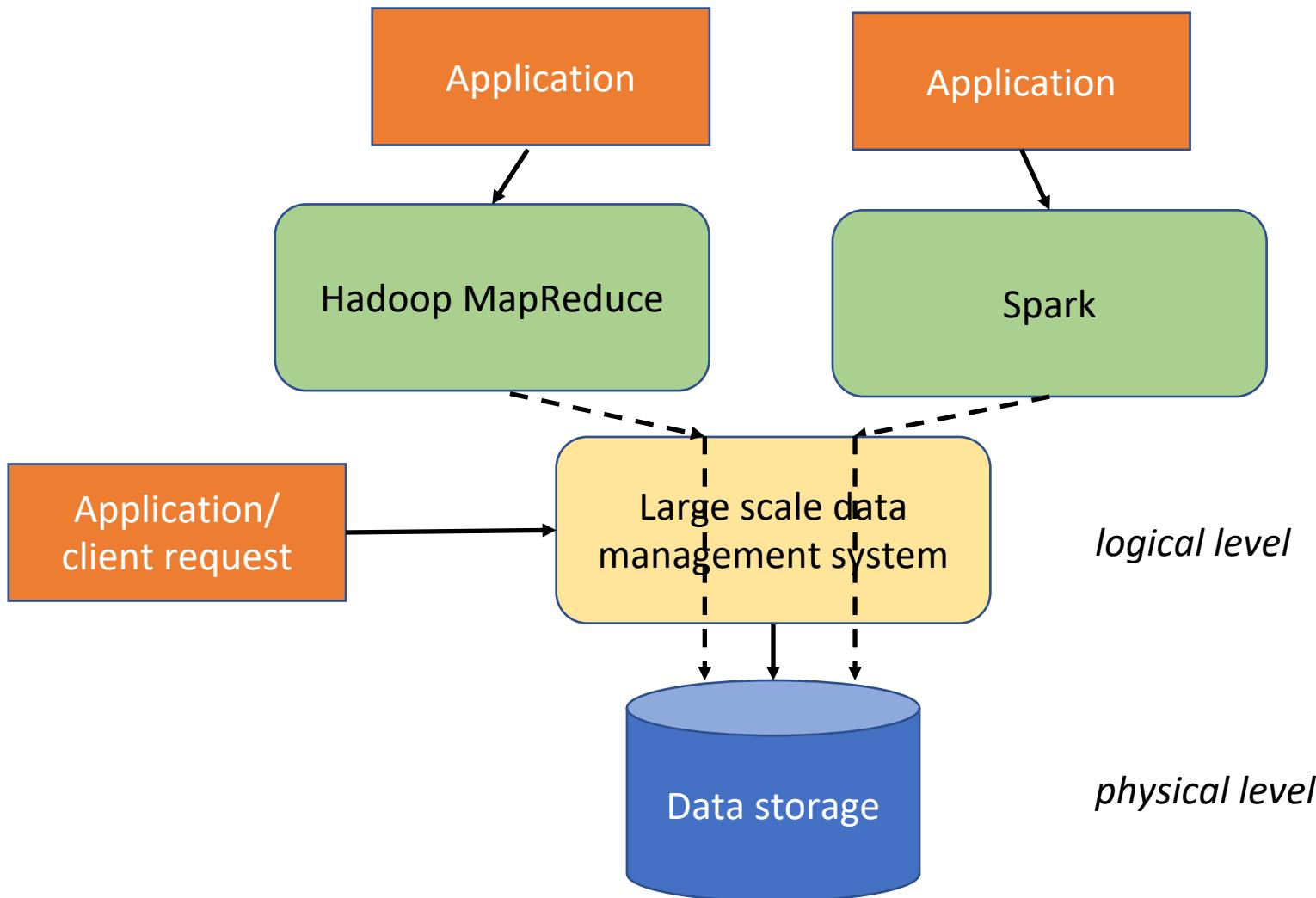
Are RDBMSs enough in a large-scale environment?

- No!
- Traditional RDBMSs can be made to run over clusters, but weren't really designed for it
- Typical DBMS functionalities can be extended to distributed architectures
- When considering large-scale distributed architectures, traditional algorithms and protocols have to be extended
 - **Very complex** algorithms and protocols
 - New issues to take into account (e.g., replica consistency) – **very costly**

Large-scale data management

- Work on clusters
- Provide a framework that
 - hides the complexity of distribution to developers (load balancing, fault tolerance)
 - provides **data storage capabilities (physical level)**, that can be used by large-scale data processing frameworks as well
 - provides a **logical level, partially independent from the physical one**
 - focuses on **data representation and querying**
- Principles
 - Scale “out”, not “up”
 - Move processing to data (data locality)
 - Focus also on **read and write random access**
 - Tailored to **transactional (and analytical) processing**

Large-scale data management



Main differences

Relational data management

1. Integration database
2. Rigid schema, structured data
3. Flat data
4. Tailored to normalization and join
5. Fully featured declarative language (SQL)

Large-scale data management

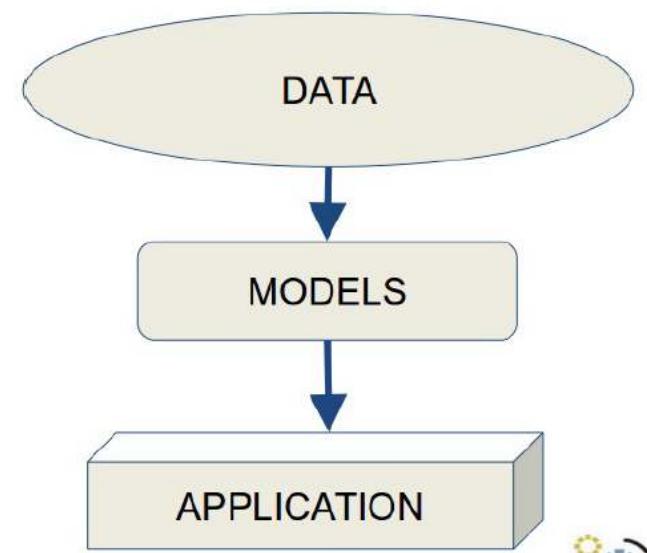
1. Application database
2. Flexible schema, possibly unstructured data
3. More complex data
4. Joins are an issue, normalization is no more a reference principle
5. Mainly procedural code

Main differences:

(1) integration vs application databases

- **Integration databases**

- One database may serve many applications, all access the same relational schema (logical level)
- The schema is usually more complex than any single application needs
- Different teams access different views of data
- Complex integrity constraint checking and access control at the DBMS level

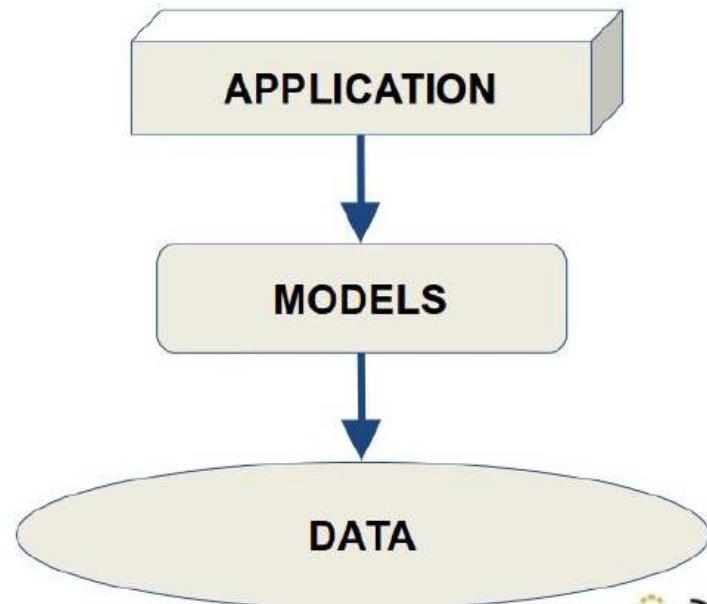


Workflow in integration DB

Main differences:

(1) integration vs application databases

- Application databases
 - Directly accessed by a **single application**
 - Data exchange between applications based on specific format (e.g., XML or JSON)
 - Rather than using RDBMS for everything, developers can choose the right DB for the right application
 - Only the team using the application needs to know about the database structure, which makes it much easier to maintain and evolve the schema
 - Since the application team controls both the database and the application code, **many features** typically supported by DBMSs (integrity checking, access control) **can be put in the application code**



Workflow in application DB

Main differences:

(2) rigid vs flexible schema

- In a relational database, all the tuples inside one relation share the same schema
- Trend accelerated by the decentralization and individualization of content generation (**crowdsourced data**), leads to the need for **relaxing the notion of schema**
- Different records of the same collection might contain different fields
- A field might hold several values
- **Semi-structured data representation**

Example

- In the relational model, the only way to cope with flexibility is to
 - Identify **in advance** all possible properties - not always possible
 - Rely on **null values** – not always convenient

Product: iPhone 12

Price: 1.100

Camera: Fine

Screen: Very good

Accessories: {Headphone, Case,}

Product: iPhone 12

Price: 1.000

Camera: Excellent

Screen: Poor in sunshine

Operating system: Easy to use

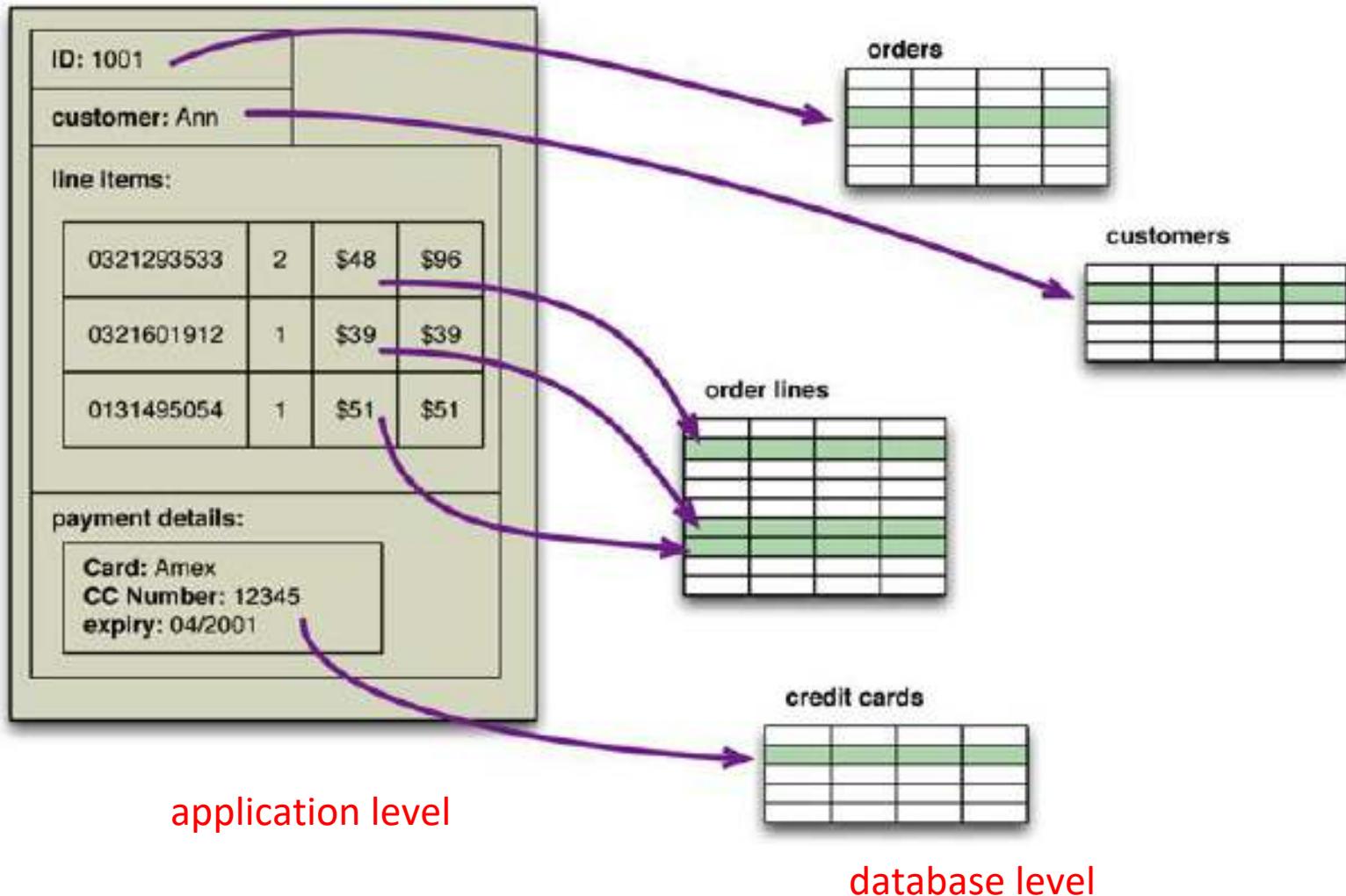
| Product | Price | Camera | Screen | Accessories | Operating System |
|-----------|-------|-----------|------------------|----------------------|------------------|
| iPhone 12 | 1.100 | Fine | Very good | Headphone, Case, ... | NULL |
| iPhone 12 | 1.000 | Excellent | Poor in sunshine | NULL | Easy to use |

Main differences:

(3) flat vs complex data

- In the relational model, each attribute value belongs to an atomic domain: number, string, date,..
- No sets, no lists
- But programming languages usually rely on more complex models (e.g., object-based models)
- Impedence mismatch between the persistent data model of the database (e.g., relational) and the in-memory data structures used by programs (e.g., object-based)
- In application databases, the relation between application and database is tighter, thus the database model should be closer to the application model

Example

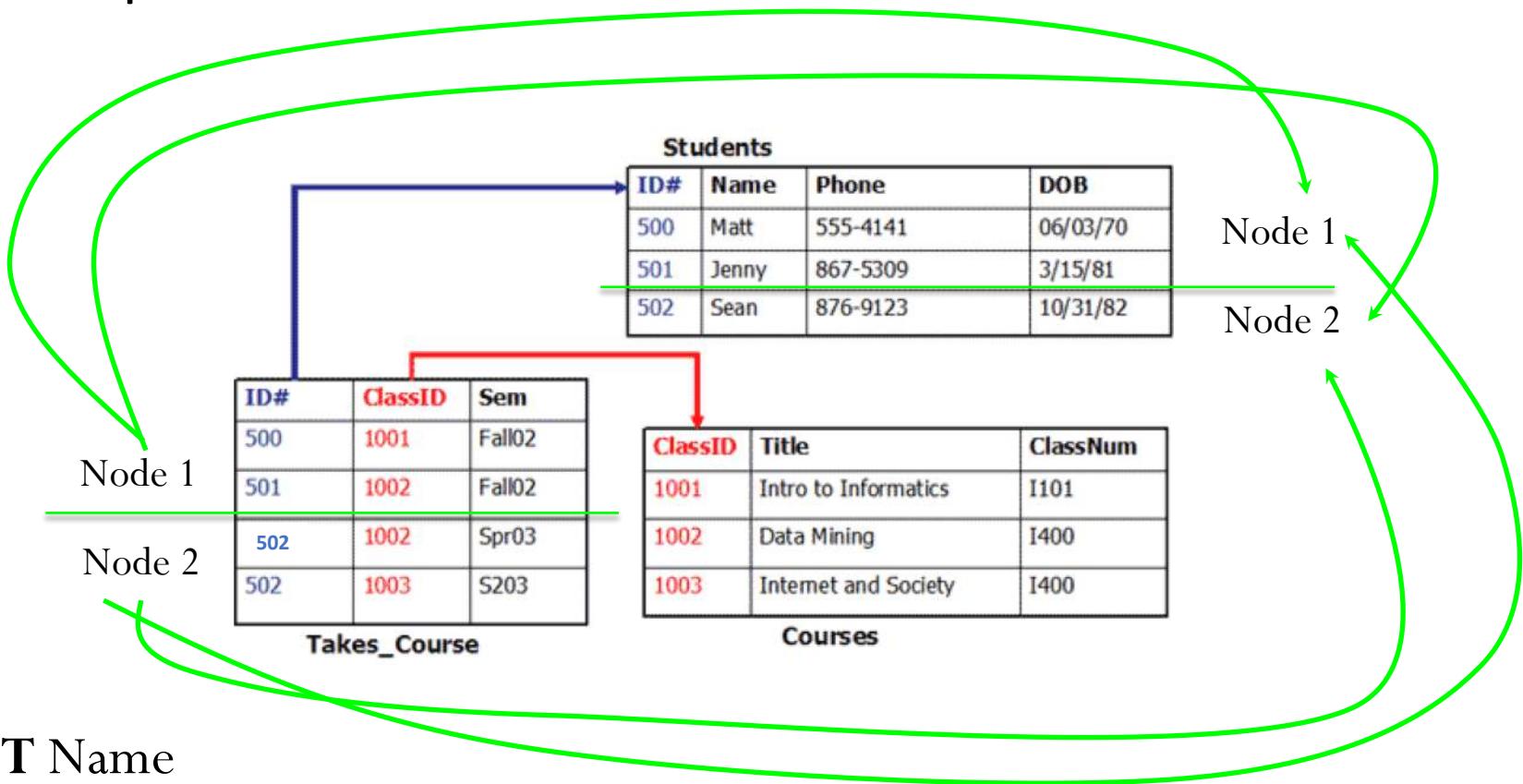


Main differences:

(4) normalization and joins

- In RDBMs, data are split into tables, according to some normal form (often 3NF)
- Queries can merge data contained in different tables using joins
- But joins are inefficient for large volumes of data
- Solution: take into account joins at design time
- **Query-based design**

Example



SELECT Name

FROM Students **NATURAL JOIN** Takes_Course

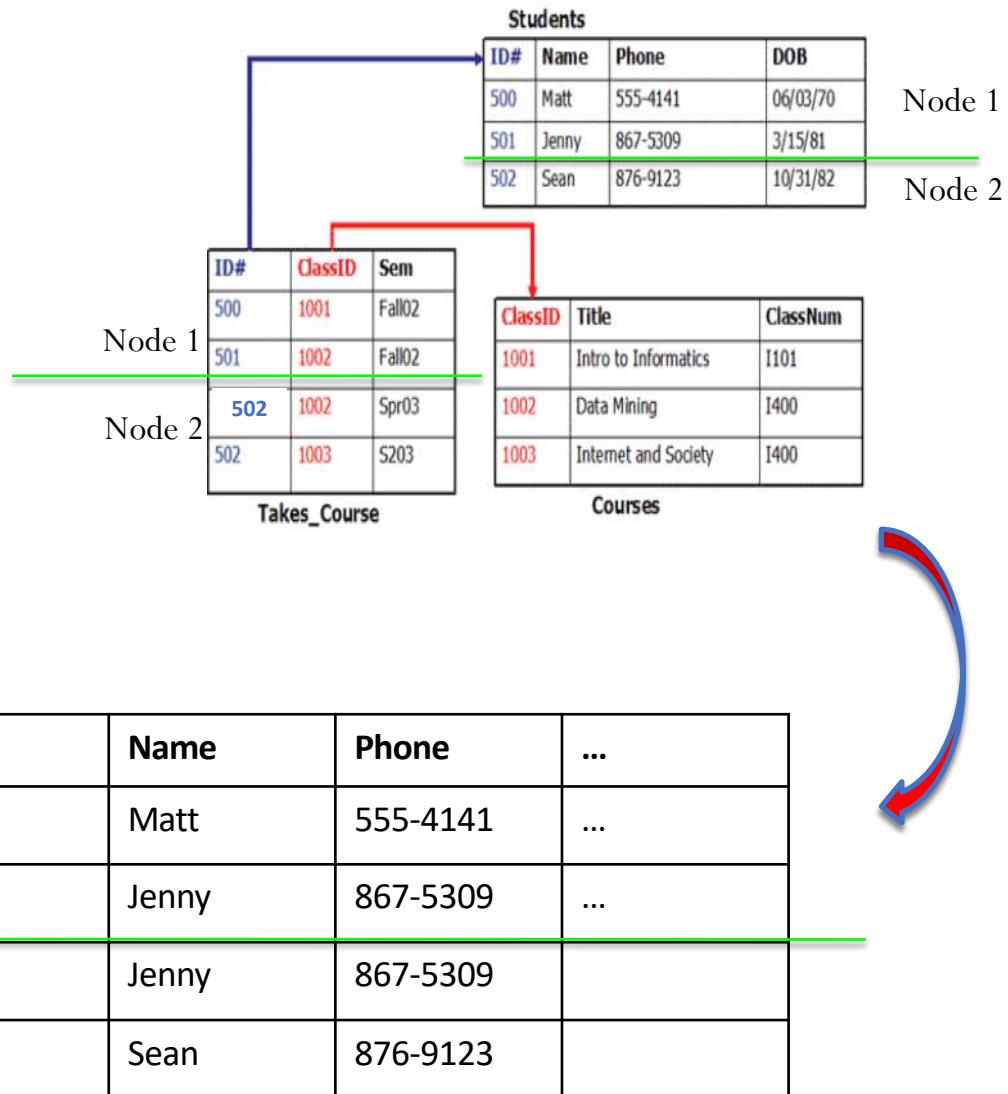
WHERE ClassID = 1001

High data communication across the network

Example

Query-based design

```
SELECT Name
FROM Students_Take_Courses
WHERE ClassID = 1001
```



Main differences:

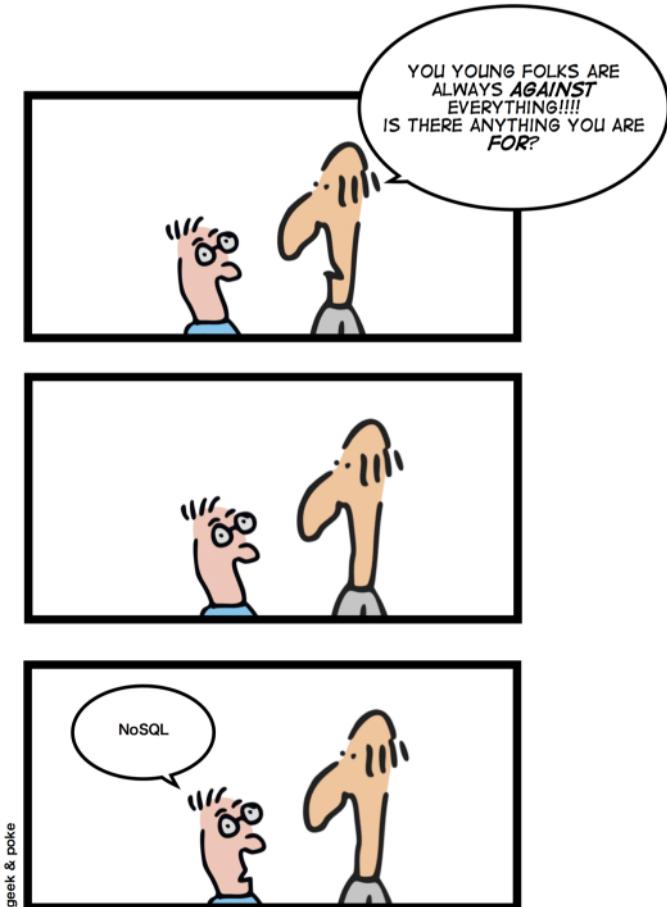
(5) interaction languages

- **Moving the complexity at the application layer**
 - In application databases, many DBMS functionalities can be moved at the application level
 - There is no more the need for the DBMS to provide a fully featured declarative language
 - Most of the code is procedural and executed at the application side (e.g., using MapReduce)
- **Interfaces with the database become simpler**
 - read and writes of one or a small number of records
 - avoid operations that require a large data communication (no complex queries or joins)
 - SQL-like languages are still provided for continuity with the past:
similar syntax, very different underlying model

Introduction to NoSQL systems

NoSQL systems (or data stores)

- New data management systems tailored to large-scale data management requirements
- **(N)ot (O)nly SQL** not necessarily “Anti SQL”
- **Movement** more than “one” technology



NoSQL data stores

SYSTEM ARCHITECTURE

- Designed to run on large clusters
- Data distribution (partitioning, replication)
- Strong or eventual consistency
- Often non-ACID transactions
- Application database oriented
- Optimized for specific scenarios (analytical/read intensive, transactional/read-write intensive)

DATA MODEL

- No relational model
- Flexible data model
- Two main modeling approaches: aggregate-oriented and graph-oriented
- Limited impedance mismatch
- Query-based design

STYLE OF INTERACTION

- Simple API and powerful integration with parallel programming frameworks
- Sometimes, SQL-like query languages

Just another temporary trend?

- There have been **other trends** here before
 - **object** databases, XML databases, etc.
- **But NoSQL data stores:**
 - are answer to **real practical problems** big companies have
 - are often developed by the **biggest players**
 - outside academia but based on **solid theoretical results** (e.g., well established results on distributed processing)
 - widely used
 - typically open-source

Who is NoSQL?



(Some) disadvantages of NoSQL

- New and sometimes buggy
- Data is generally replicated, potential for inconsistency
- No standardized data model
- No standardised language for data access
- Data integrity often enforced at the application level

... but

- More and more companies accept the weak points and choose NoSQL databases for their strengths
- NoSQL technologies are also often used as secondary databases for specific data processing applications (in line with the application database concept)

The end of relational databases?

- Relational databases are not going away
 - are ideal for a lot of structured data, reliable, mature, etc.
- RDBMS became one option for data storage
- Polyglot persistence – using different data stores in different circumstances

NoSQL logical data models

- Two main *groups* of models
 - aggregate-oriented
 - graph-oriented
- For each model: no standard

Take away

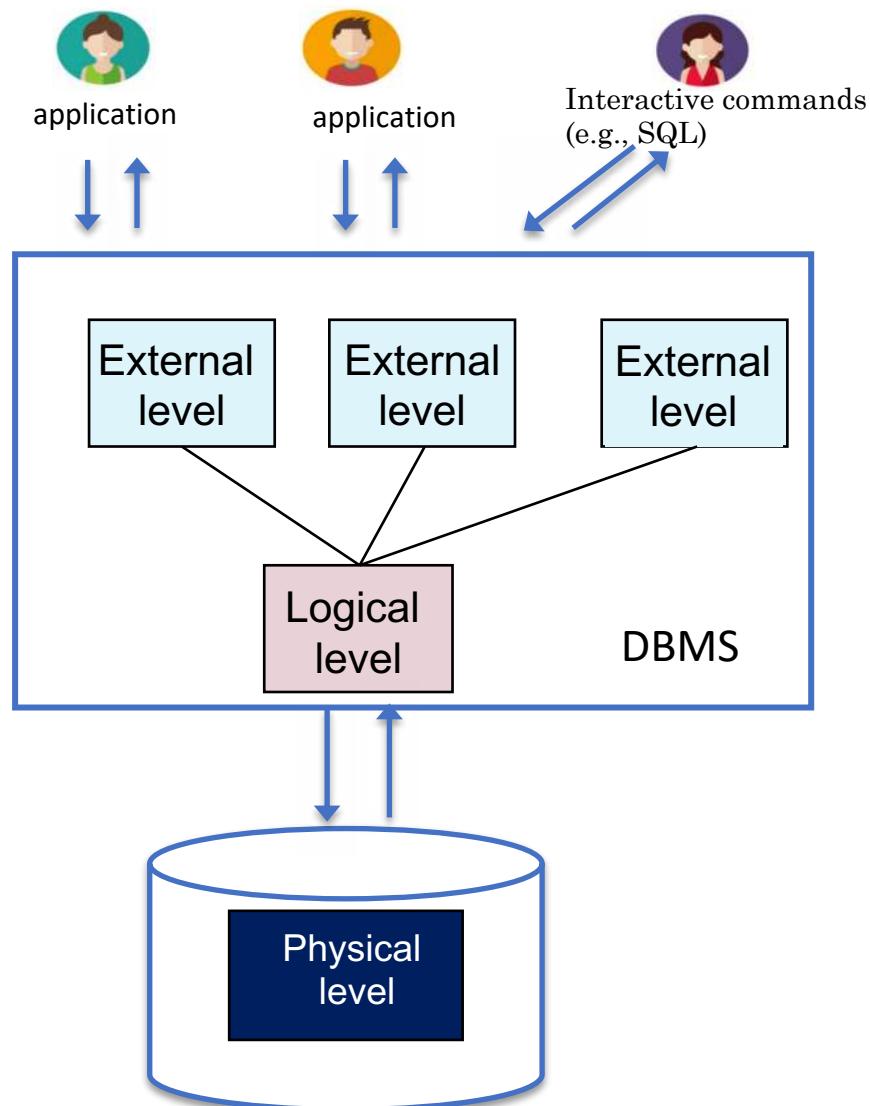
- Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism
- Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures
- There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services
- The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters
- NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics
 - Not using the relational model, schedules
 - Various interaction protocols
 - Horizontal scaling
 - Replication and sharding
 - Relaxing consistency, no more ACID transactions
 - Open-source

Aggregate-oriented NoSQL data stores

Introduction

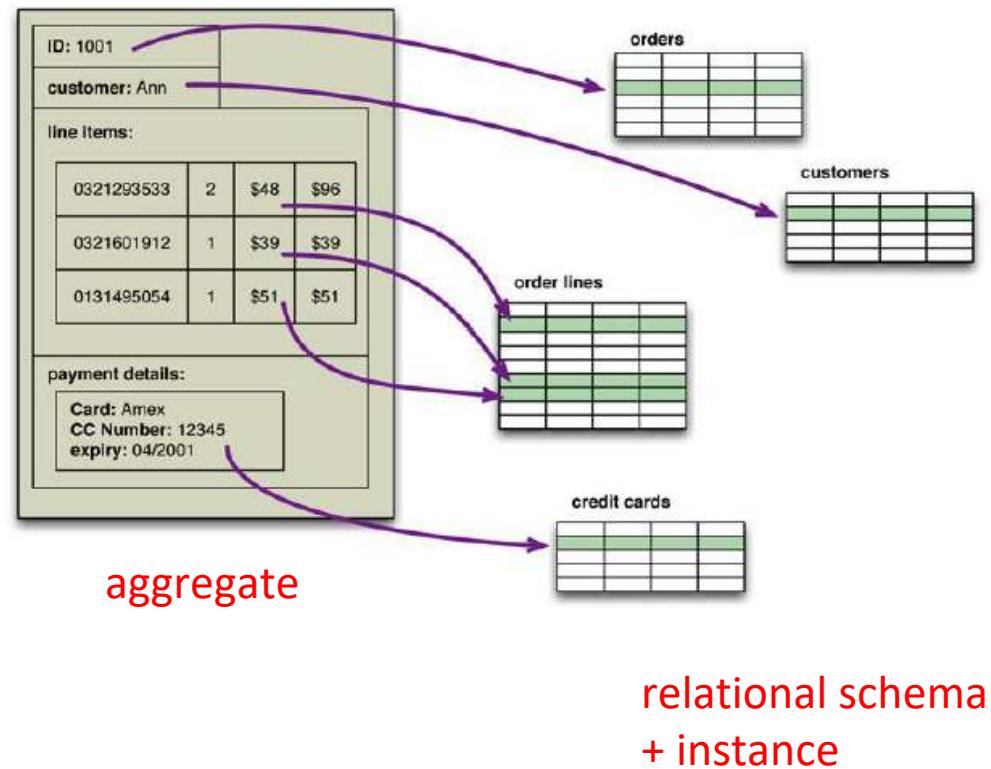
Data models

- A **data model** is a set of constructs for representing a certain reference domain
- **Logical model**: how entities and their associations are represented inside the system
 - Relational model: tables, columns and rows
- **Storage model**: how the DBMS stores and manipulates the data at the physical level
- A logical model is usually independent of the storage model
- In NoSQL systems, this is no more true



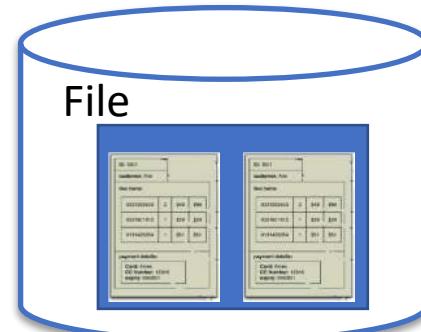
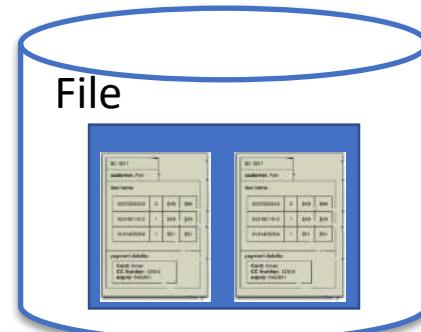
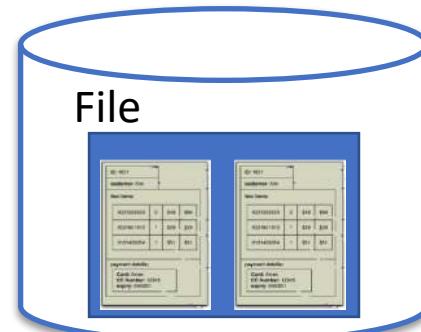
Aggregate: definition

- **Logical level:** an aggregate is a data unit with a **complex structure**
- Not simply a tuple (a table row) like in RDBMS
 - Example: complex record with: simple fields, arrays, records nested inside
- For data manipulation and management of consistency



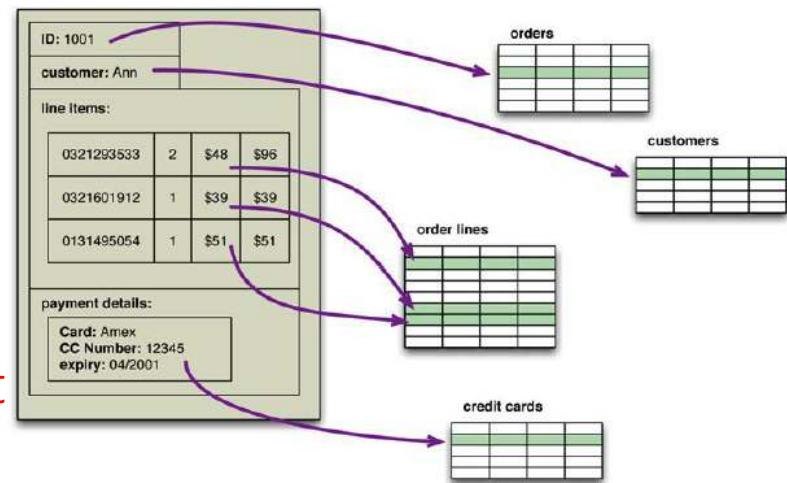
Aggregate: definition

- **Physical level:** an aggregate is the unit of interaction with the data store
- All the data about a unit of interest (an aggregate) are **kept together** on the same node
- Partitioning separates **different units** (different aggregates) on different nodes



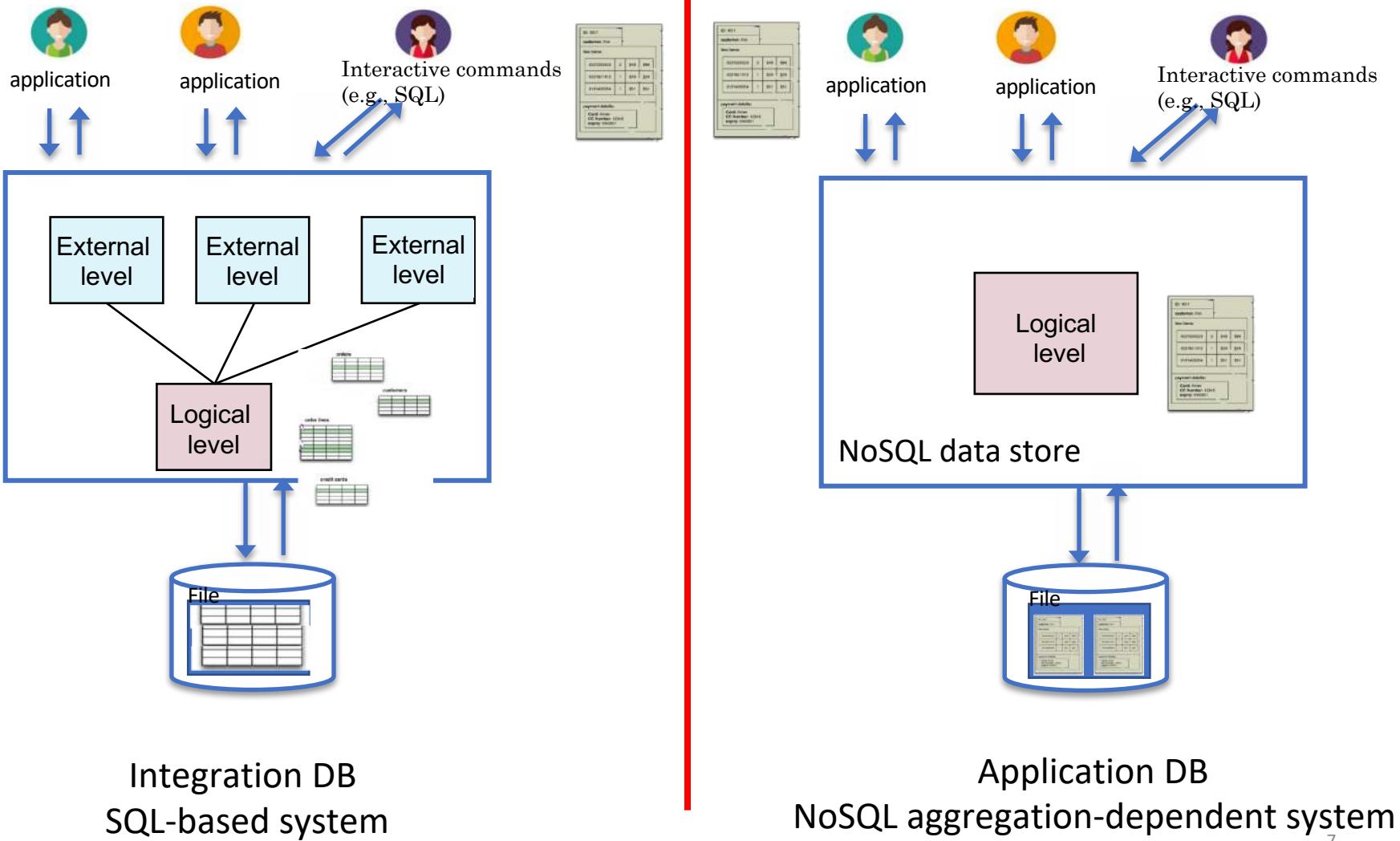
Aggregates

- Relational model is **aggregate-ignorant**
 - It is not a bad thing, it is a **feature**
 - Allows to easily **look** at the data in **different ways**
 - Best choice for **integration database** (no primary structure for data manipulation)
- **Advantages** of aggregates:
 - good option for **application database**
 - easier for application programmers to work with
 - easier for database systems to handle operating on a cluster
 - **limit impedance mismatch:** strict relationship with **JSON**, a lightweight format for data exchange



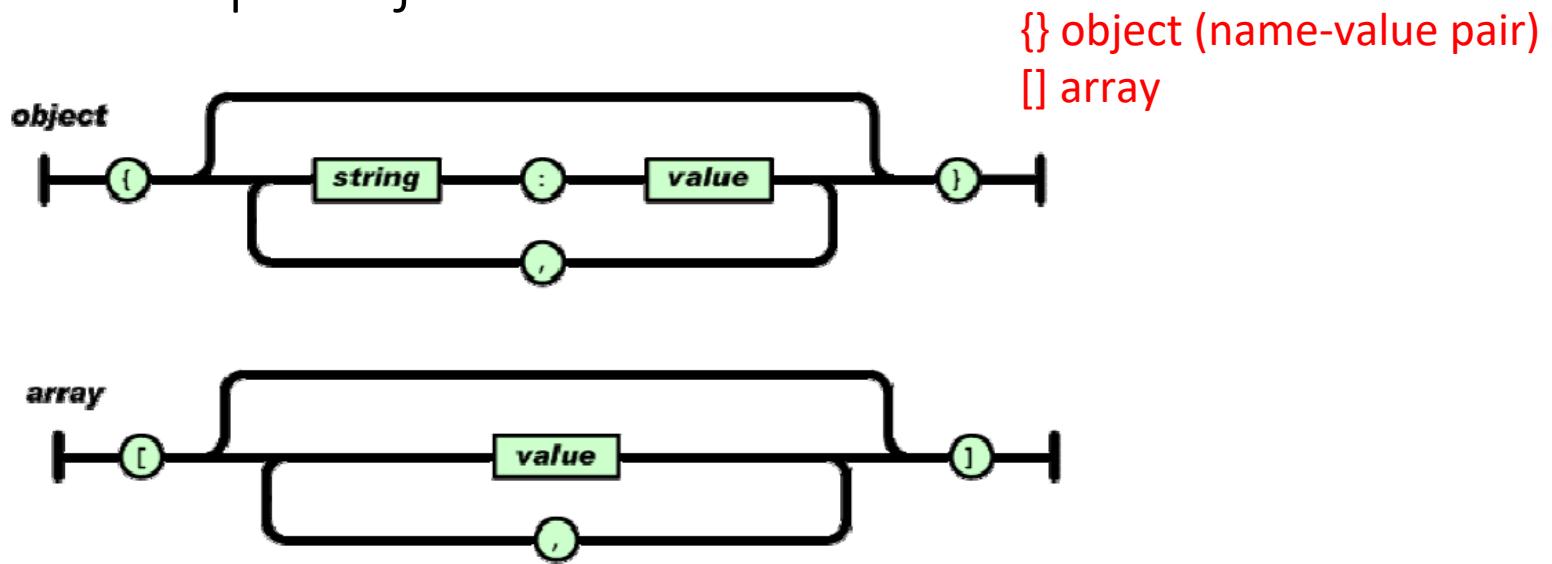
JSON: see short introduction on Aulaweb: Resources

Aggregates



JSON

- In case you haven't seen it, here is a quick introduction to JavaScript Object Notation



- JSON schema can be used to specify the structure of a set of JSON objects
 - Semistructured schema

Example

- Imagine we decide to build a product review database, collecting together all the reviews of a given product:

Product: iPhone 12

Reviewer: 458743

Date: 12.4.2021

Price: 1.100

Camera: Fine

Screen: Very good

Accessories: {Headphone, Case,}

Product: iPhone 12

Reviewer: 636534

Date: 30.5.2021

Price: 1.000

Camera: Excellent

Screen: Poor in sunshine

Operating system: Easy to use

- Different products have different fields
- Some fields have several values
- Products are related to other products as accessories
- A new product may have new qualities (fields) not yet in the database
- The same product may appear many times with different values and fields

Example JSON

{ object (name-value pair)
[] array

Product: iPhone 12

Reviewer: 458743

Date: 12.4.2021

Price: 1.100

Camera: Fine

Screen: Very good

Accessories: {Headphone, Case,}

Product: iPhone 12

Reviewer: 636534

Date: 30.5.2021

Price: 1.000

Camera: Excellent

Screen: Poor in sunshine

Operating system: Easy to use

```
{  
  "Product": "iPhone 12",  
  "Review":  
    {"reviewer": "458743",  
     "date": "12.4.2021",  
     "camera": "Fine",  
     "screen": "Very good",  
     "price": 1.100,  
     "accessories": ["Headphone", "Case"]}  
}
```

```
{  
  "Product": "iPhone 12",  
  "Review":  
    {"reviewer": "636534",  
     "date": "30.5.2021",  
     "camera": "Excellent",  
     "screen": "Poor in sunshine",  
     "price": 1.000,  
     "operatingSystems": "Easy to use"}  
}
```

Example JSON schema

```
{  
  "type": "object",  
  "title": "Review",  
  "description": "A review for a certain product",  
  "properties": {  
    "Product": { "type": "string", "description": "The product for which the review is given" },  
    "Review": {  
      "type": "object",  
      "properties": {  
        "reviewer": { "type": "string", "description": "" },  
        "date": { "type": "string", "description": "" },  
        "reviewer": { "price": "string", "description": "" },  
        "camera": { "type": "string", "description": "" },  
        "screen": { "type": "string", "description": "" }  
      }  
    }  
  },  
  "required": ["Product"]  
}
```

Aggregates and JSON

- We rely on JSON and a simplified JSON schema for describing aggregates at a meta-logical level
- Close but not coinciding with logical models provided by aggregate-oriented NoSQL data stores
- Useful for understanding aggregate modeling before implementation

Properties achieved

1. Application database
2. Flexible schema, possibly unstructured data
3. More complex data
4. Joins are an issue, normalization is no more a reference principle
5. Mainly procedural code

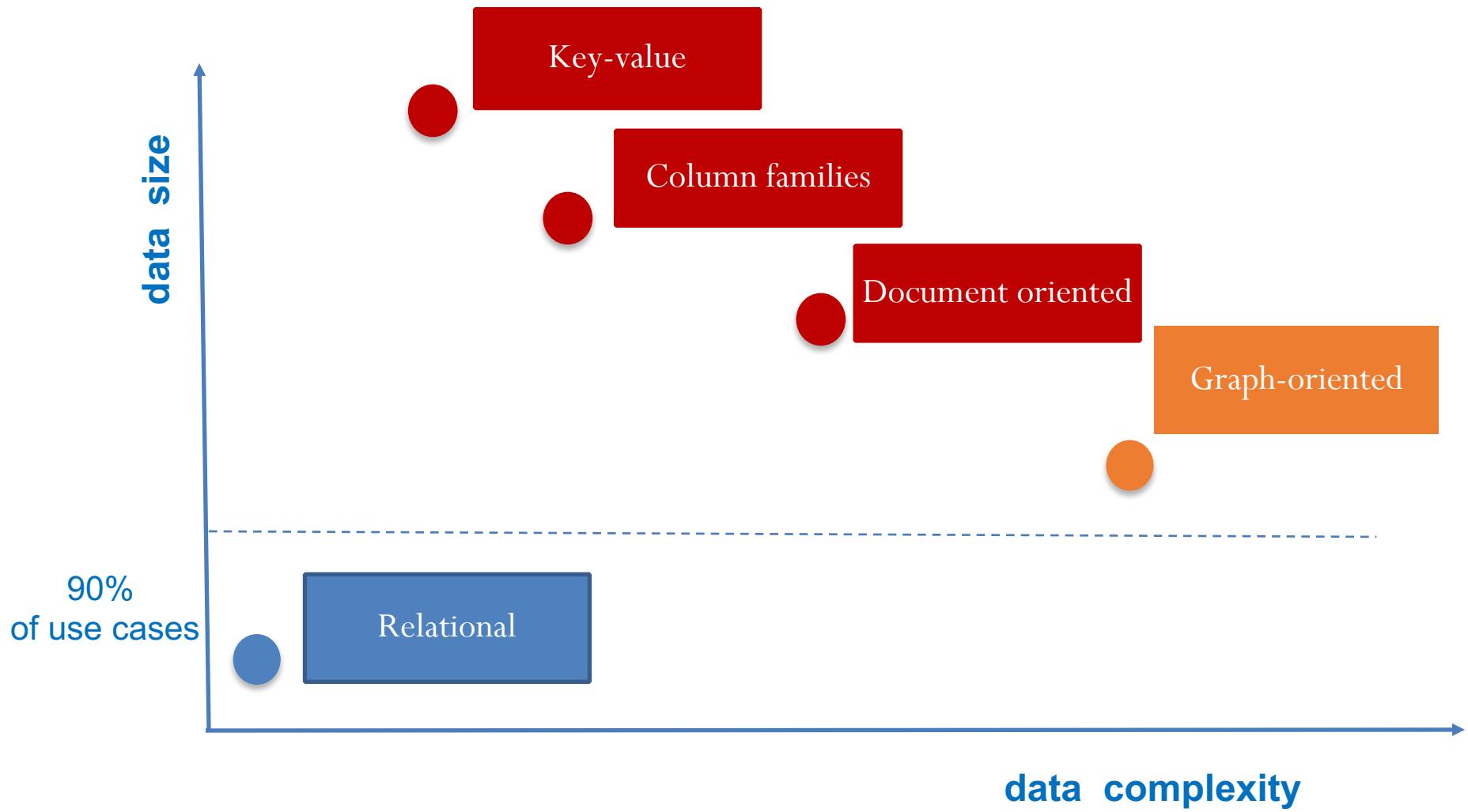
Aggregate properties

- Aggregates give the database information about
 - which portions of data will be manipulated together (logical view)
 - what should be stored on the same node (physical view)
 - *strict relationship between logical and physical levels*
- Minimize the number of nodes accessed during a search (*if the aggregate is modeled taking into account the workload*)
- Impact on concurrency control and consistency
 - NoSQL databases typically support atomic manipulation of a single aggregate at a time
 - Update that affects multiple aggregates leaves open a time slot during which clients could perform an inconsistent read
 - Part of the consideration for deciding how to aggregate data

Aggregate-oriented logical data models

- Aggregate-oriented NoSQL databases based on aggregates are categorized according to the **characteristics of aggregates** themselves [Rick Cattell, 2010]
 - Key value
 - Document-oriented
 - Column family
- Entities are represented as pairs **(key, value)**:
 - **key** is an identifier (not necessarily unique among a collection) of an entity
 - **value** describes the entity structure and corresponds to an aggregate, following a JSON style
- Different **models differ for the degree of structure associated with aggregates** and, as a consequence, the types of manipulations to be applied over aggregates
- At the physical level, the **key is the partitioning value**: different aggregates associated with the same key are stored in the same node

Aggregate-oriented logical data models



Aggregate-oriented logical design

Relational database design

- Conceptual design
 - From the requirements to a conceptual schema
 - *Conceptual model:* entity-relationship model
- Logical design
 - From a conceptual schema to a logical schema
 - *Logical model:* relational model
 - The logical schema can then be directly created in an RDBMS
- Physical design
 - Optimizing the storage of the database on disk
 - Indexing, clustering,...

Aggregate-oriented design

- Conceptual design
 - From the requirements to a conceptual schema + workload
 - *Conceptual model*: entity-relationship model
- Logical design
 - From a conceptual schema + workload to a logical schema
 - *Logical model*: one specific aggregate-oriented model
 - Often useful to start by designing a meta-logical schema, based on a meta-notation corresponding to JSON-like schema
 - Given a NoSQL system, the meta-logical schema has to be translated into the aggregate-oriented model provided by the system
- Physical design
 - Optimizing the storage of the database on disk
 - Partitioning (always), indexes (sometimes)

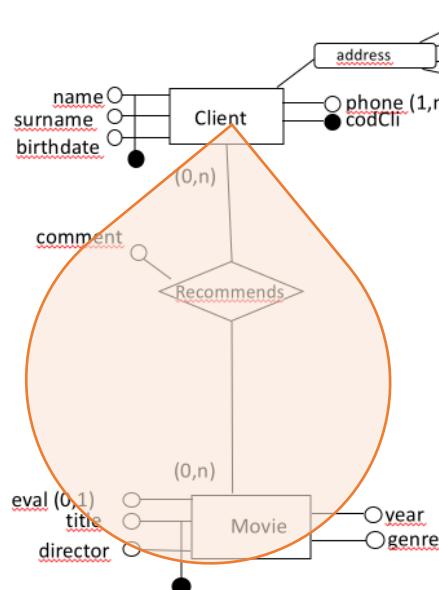
Aggregate-oriented logical design: principles

1. Minimize the number of joins by minimizing the number of collections
2. Reduce redundancy by using complex attributes (lists, sets)
3. Selections over simple attributes are more efficient than selections over complex attributes (sets, lists)
 - In some systems, selections over complex attributes are not possible

Aggregate

- Taking a reference workload into account, we extend the attributes of an entity E with the attributes of other entities associated with E or of associations involving E
- We «encapsulate» in E properties related to other entities or associations to facilitate their retrieval, based on the operations contained in the workload

ER schema



{ } object (name-value pair)
[] array

JSON (instance level)

```
{
    "codcli": 375657,
    "name": "John",
    "surname": "Black",
    "birthdate": "15/10/2000",
    "address": {"city": "Genoa", "street": "Via XX Settembre", "streetNumber": 15, "postalCode": 16100},
    "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction", "director": "quentin tarantino"}}, {"movie": {"comment": "very nice", "title": "pulp fiction", "director": "quentin tarantino"}}]
}
```

JSON – tree view

```

    ▼ object {6}
        codcli : 375657
        name : John
        surname : Black
        birthdate : 15/10/2000
    ▼ address {4}
        city : Genoa
        street : Via XX Settembre
        streetNumber : 15
        postalCode : 16100
    ▼ movies [2]
        ▼ 0 {1}
            ▼ movie {3}
                comment : very nice
                title : pulp fiction
                director : quentin tarantino
        ▼ 1 {1}
            ▼ movie {3}
                comment : very nice
                title : pulp fiction
                director : quentin tarantino
```

Aggregate schema

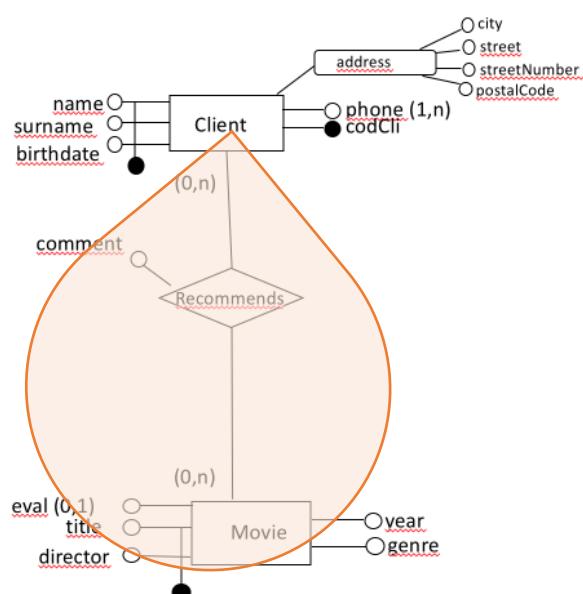
{ } object (name-value pair)
[] array

- We rely on a JSON schema meta-notation

JSON (instance level)

```
{  
    "codcli": 375657,  
    "name": "John",  
    "surname": "Black",  
    "birthdate": "15/10/2000",  
    "address": {"city": "Genoa", "street": "Via XX Settembre",  
               "streetNumber": 15, "postalCode": 16100},  
    "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
                      "director": "quentin tarantino"}},  
              {"movie": {"comment": "very nice", "title": "pulp fiction",  
                      "director": "quentin tarantino"}}]  
}
```

ER schema



JSON schema meta-notation (semistructured schema level)

```
client: { codCli, name, surname, birthdate,  
          address: {city, street, streetNumber, postalCode},  
          movies: [{movie: {comment, title, director,...}}]  
        }
```

We underline all sets of attributes that represent identifiers

JSON schema

```
{  
  "type": "object",  
  "title": "Client",  
  "description": "",  
  "properties": {  
    "codCli": { "type": "string", "description": "" },  
    "name": { "type": "string", "description": "" },  
    "surname": { "type": "string", "description": "" },  
    "birthdate": { "type": "string", "description": "" },  
    "address": {  
      "type": "object",  
      "properties": {  
        "city": { "type": "string", "description": "" },  
        "street": { "type": "string", "description": "" },  
        "streetNumber": { "type": "string", "description": "" },  
        "postalCode": { "type": "string", "description": "" }  
      },  
      "required": ["city", "street", "streetNumber", "postalCode"]  
    },  
    "movies": {  
      "type": "array",  
      "description": "",  
      "items": {  
        "type": "object",  
        "title": "movie",  
        "description": "",  
        "properties": {  
          "title": { "type": "string", "description": "" },  
          "director": { "type": "string", "description": "" },  
          "comment": { "type": "string", "description": "" }  
        },  
        "required": ["title", "director", "comment"]  
      },  
      "minItems": 0  
    },  
    "required": ["codCli", "name", "surname", "birthdate", "address"],  
    "id": ["codCli", ["name", "surname", "birthdate"]]  
  }  
}
```

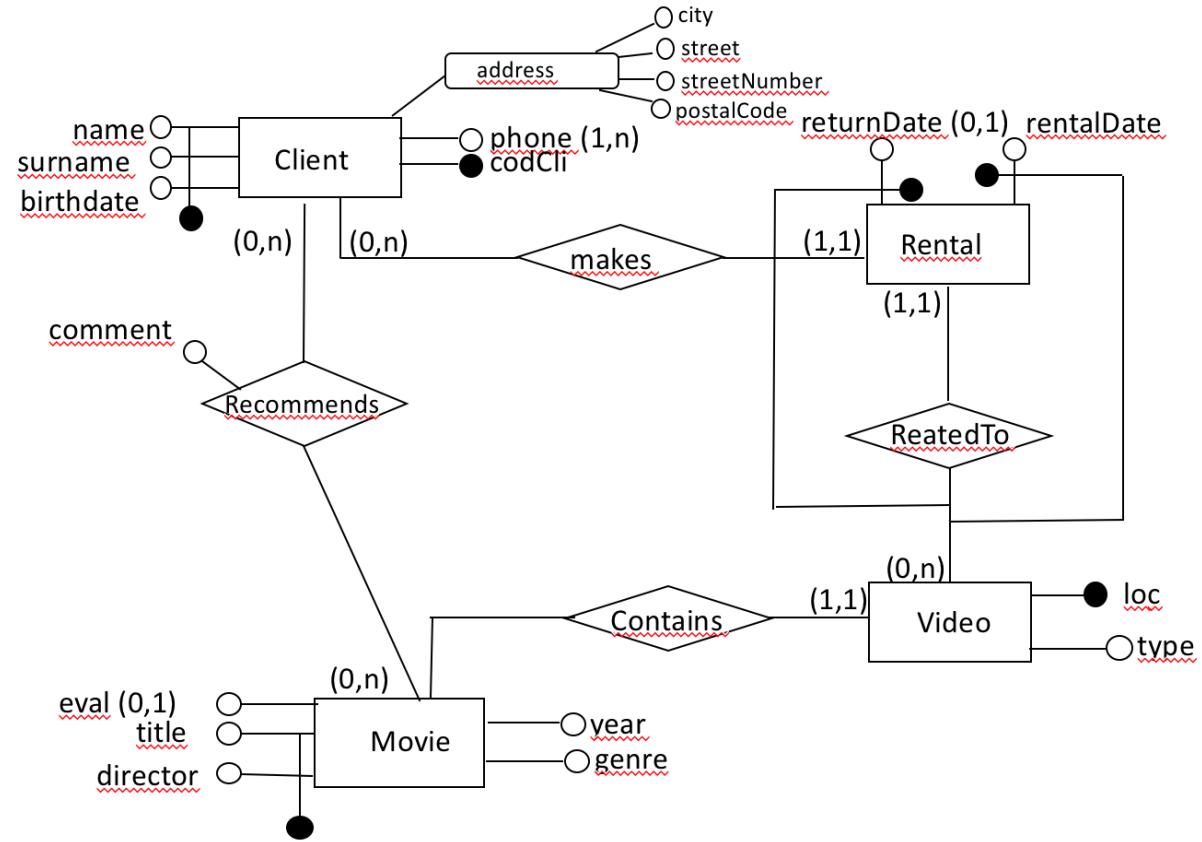
JSON schema meta-notation

```
client: {  
  codCli, name, surname, birthdate,  
  address: {city, street, streetNumber, postalCode},  
  movies: [{movie: {comment, title, director,...}}]  
}
```

Methodology

- Input
 - ER schema
 - Workload
- Step 1
 - Each query in the workload is modeled in a formal and non ambiguous way
- Step 2
 - The ER schema is annotated with query information
- Step 3
 - The aggregate-oriented logical schema is generated starting from the annotated ER schema
- Output
 - The aggregate-oriented logical schema

Workload



- Q1. Average age of clients
- Q2. Name and surname of clients and related recommended movies
- Q3. Genre and year of the movies and their related recommendations, together with the name and the surname of the client who made them
- Q4. Name and surname of clients who recommended the movie 'pulp fiction' by 'quentin tarantino'
- Q5. Given a movie, all information of videos that contain it
- Q6. Videos of type 'DVD', rented from a certain date
- Q7. The videos of type 'VHS' containing the movie 'pulp fiction' by 'quentin tarantino' and the clients that rented them

Step 1: query modeling

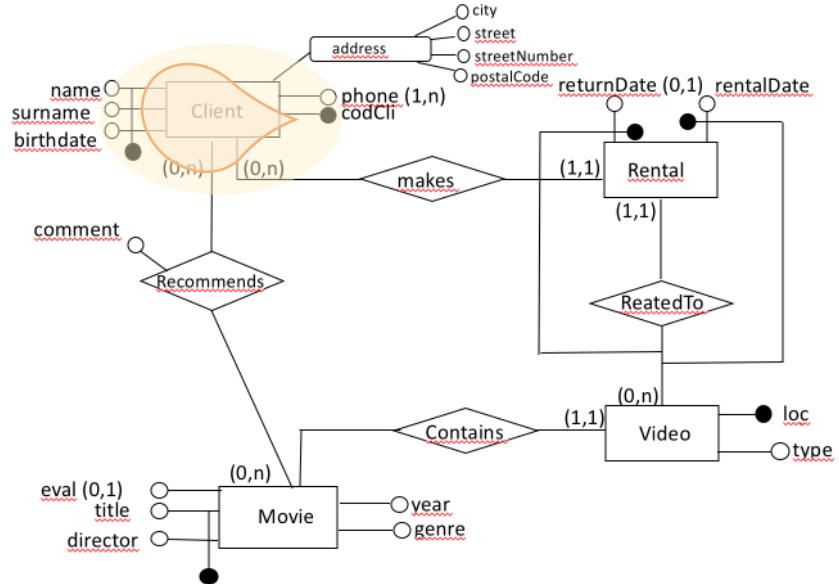
- For each query Q in the workload, determine
 - The entity E to be used as aggregate
 - The set of entities LS used for selecting data to be returned as query result (= FROM and WHERE clauses of an SQL query), the attributes used in the selections, and the path connecting each of them to E
 - If LS is not empty, E must be selected from LS entities
 - The set of entities LP used for projections (= SELECT clause of an SQL query), the attributes used in the projections, and the path connecting each of them to E
 - If you can choose, better if E appears at the $(0,1)$ or $(1,1)$ side of an association connecting selection/projection entities [as we will see, this limits selection conditions over nested attributes]
- $Q \rightarrow Q(E, LS, LP)$

Case 1: no selection entity, one projection entity (projection over some entity attributes)

- Q1. Average age of clients

- E = Client
- LS = []
- LP = [Client (birthdate) _!]

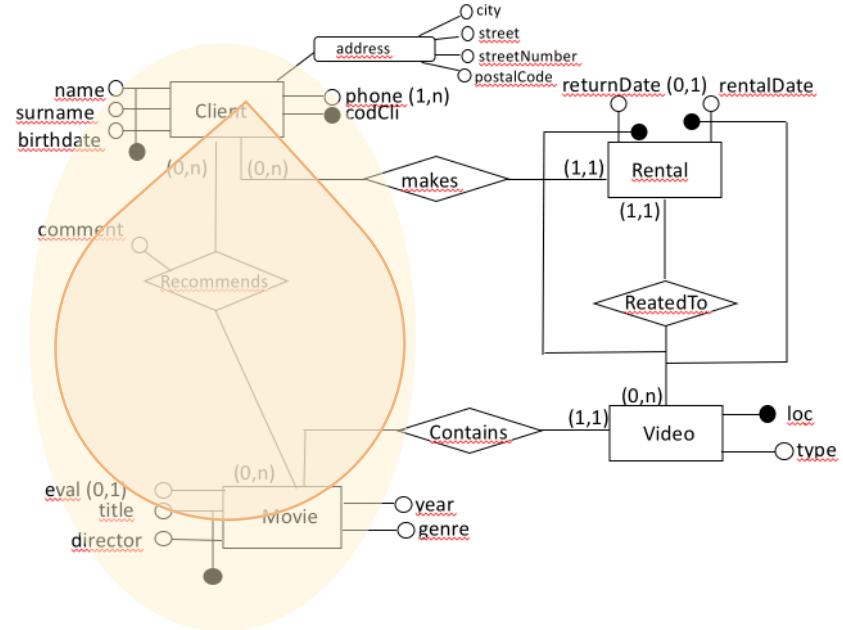
- Q1 → Q1(Client,
[],
[Client (birthdate) _!])



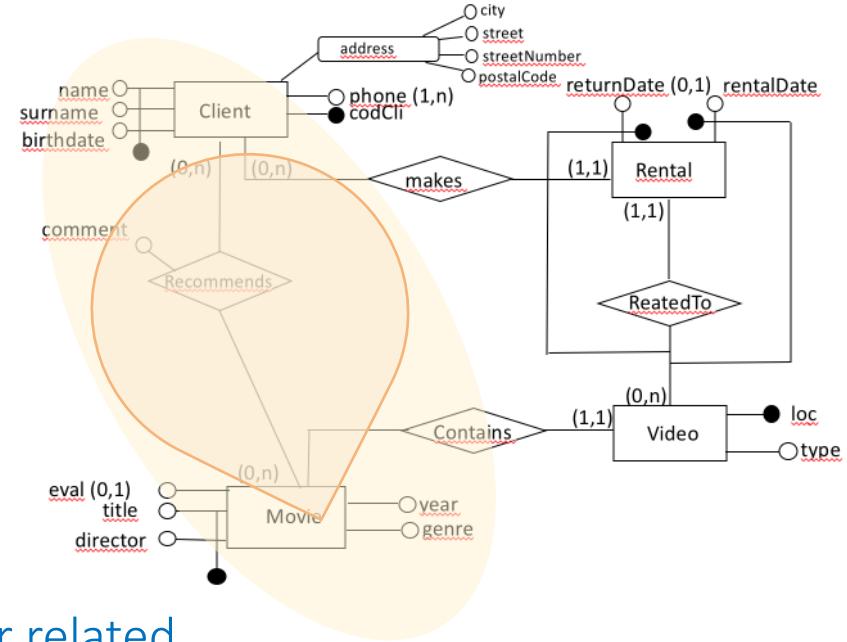
Case 2: no selection entity, two projection entities (projection over some entity attributes)

- Q2. Name and surname of clients and related recommended movies

- E = Client
- LS = []
- LP = [Client(name, surname)_!, Movie(title, director)_R]
- Q2 (Client,
[],
[Client(name, surname)_!, Movie(title, director)_R])

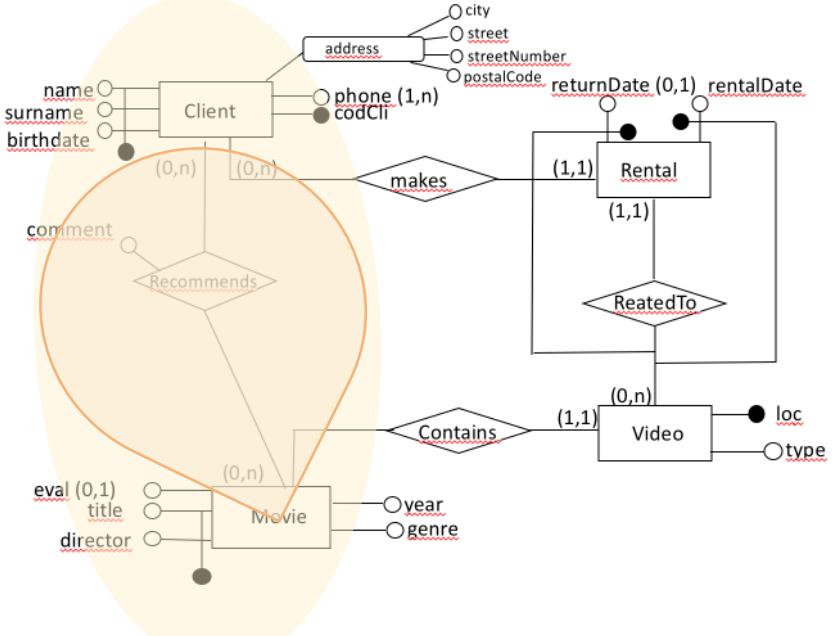


Case 3: no selection entities, two projection entities (projection over entity and association attributes)



- Q3. Genre and year of the movies and their related recommendations, together with the name and the surname of the client who made them
- E = **Movie**
- LS = **[]**
- LP = **[Movie(genre, year)_!, Client(name, surname)_R(comment)]**
- Q3 (**Movie**,
[],
[Movie(genre, year)_!, Client(name, surname)_R(comment)])

Case 4: one selection entity, one projection entity (projection over some entity attributes)

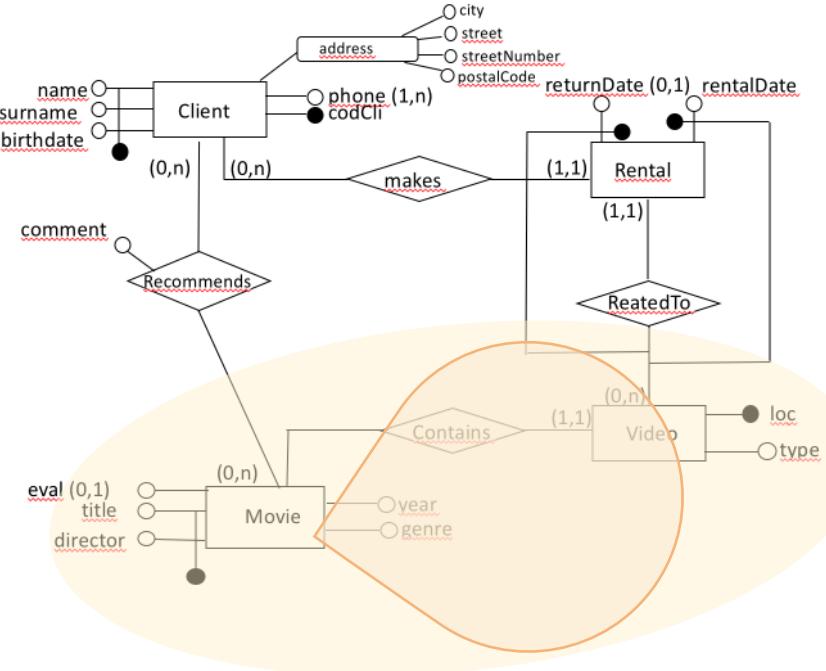


- Q4. Name and surname of clients that recommended the movie 'pulp fiction' of 'quentin tarantino'
- E = Movie
- LS = [Movie(title, director)_!]
- LP = [Client(name, surname)_R]
- Q4 (Movie,
[Movie(title, director)_!],
[Client(Name, Surname)_R])

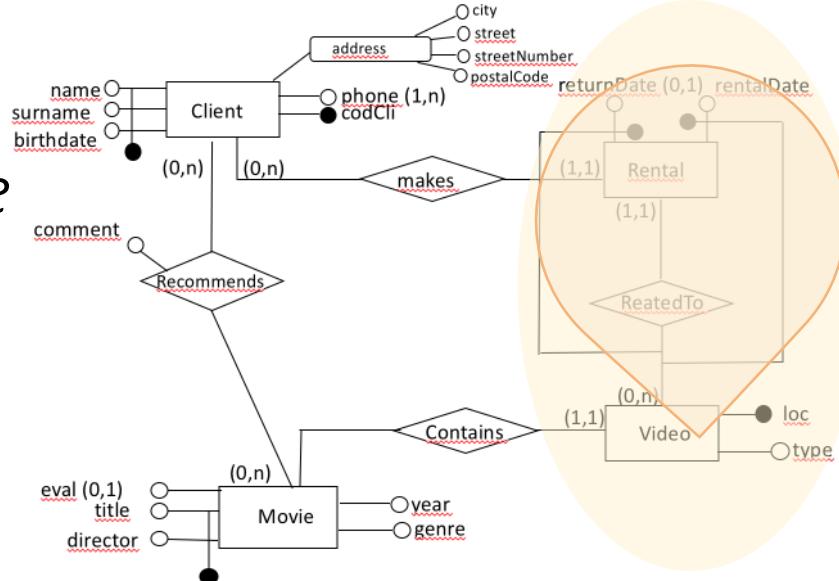
Case 5: one selection entity, one projection entity (on *all the attributes*)

- Q5. Given a movie, all information of videos that contain it

- E = Movie
- LS = [Movie(title, director)_!]
- LP = [Video_C]
- Q5 (Movie,
[Movie(title, director)_!],
[Video_C])

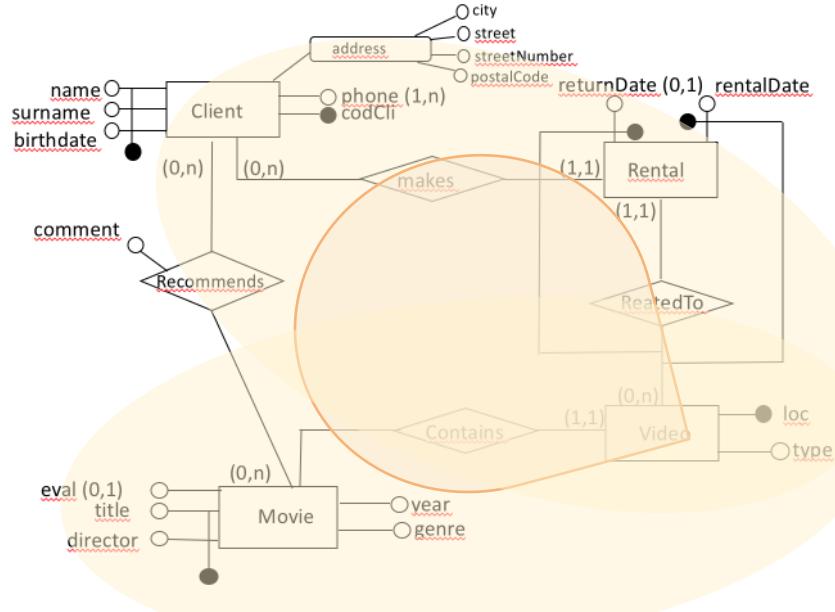


Case 6: two selection entities, one projection entity (over some entity attributes)



- Q6. The videos of type 'dvd', rented from a given date
- E = **Video**
- LS = **[Video(type)_!, Rental(rentalDate)_Rt]**
- LP = **[Video(loc)_!]**
- Q6 (**Video**,
[Video(type)_!, Rental(rentalDate)_Rt],
[Video(loc)_!])

Case 7: two selection entities, two projection entities



- Q7. The videos of type 'vhs' containing the movie 'pulp fiction' of 'quentin tarantino' and the clients that rented them
- E = **Video**
- LS = **[Video(type)_!, Movie(title, director)_C]**
- LP = **[Video(loc)_!, Client(codCli)_MRT]**
- Q7 (**Video**,
[Video(type)_!, Movie(title, director)_C]
[Video(loc)_!, Client(CodCli)_MRt])

Step 2: annotation of the ER schema

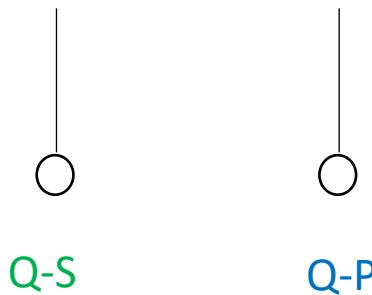
- The ER schema is annotated with information from each query Q (**E**, **LS**, **LP**)



Aggregation entity **E**

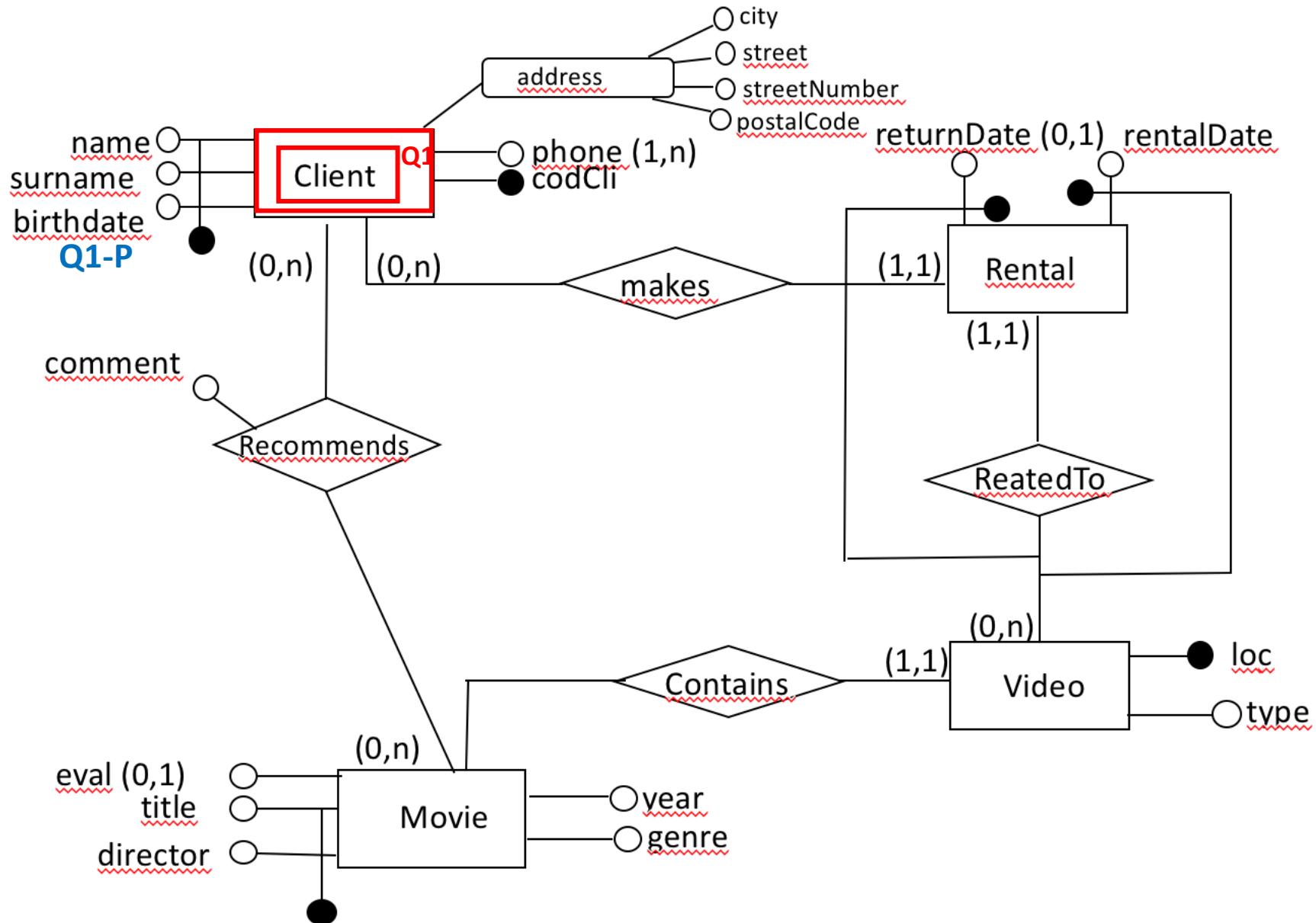


Navigation arrows on association edges,
described in **LS** and **LP**, towards the aggregation entity



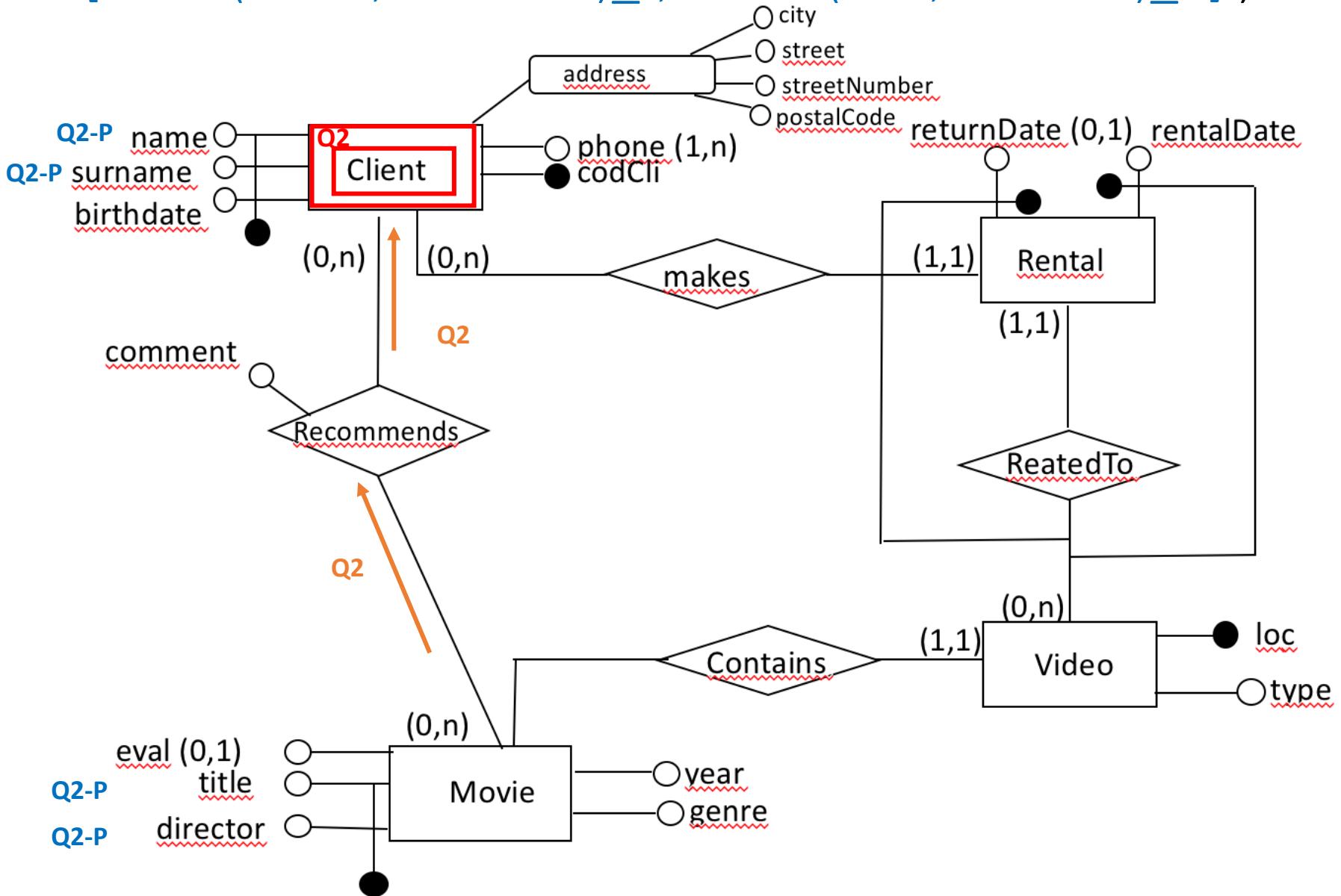
Reference to query Q on each attribute involved in
selection **LS**: **Q-S**
projection **LP**: **Q-P**

Q1(Client, [], [Client(birthdate)_!])

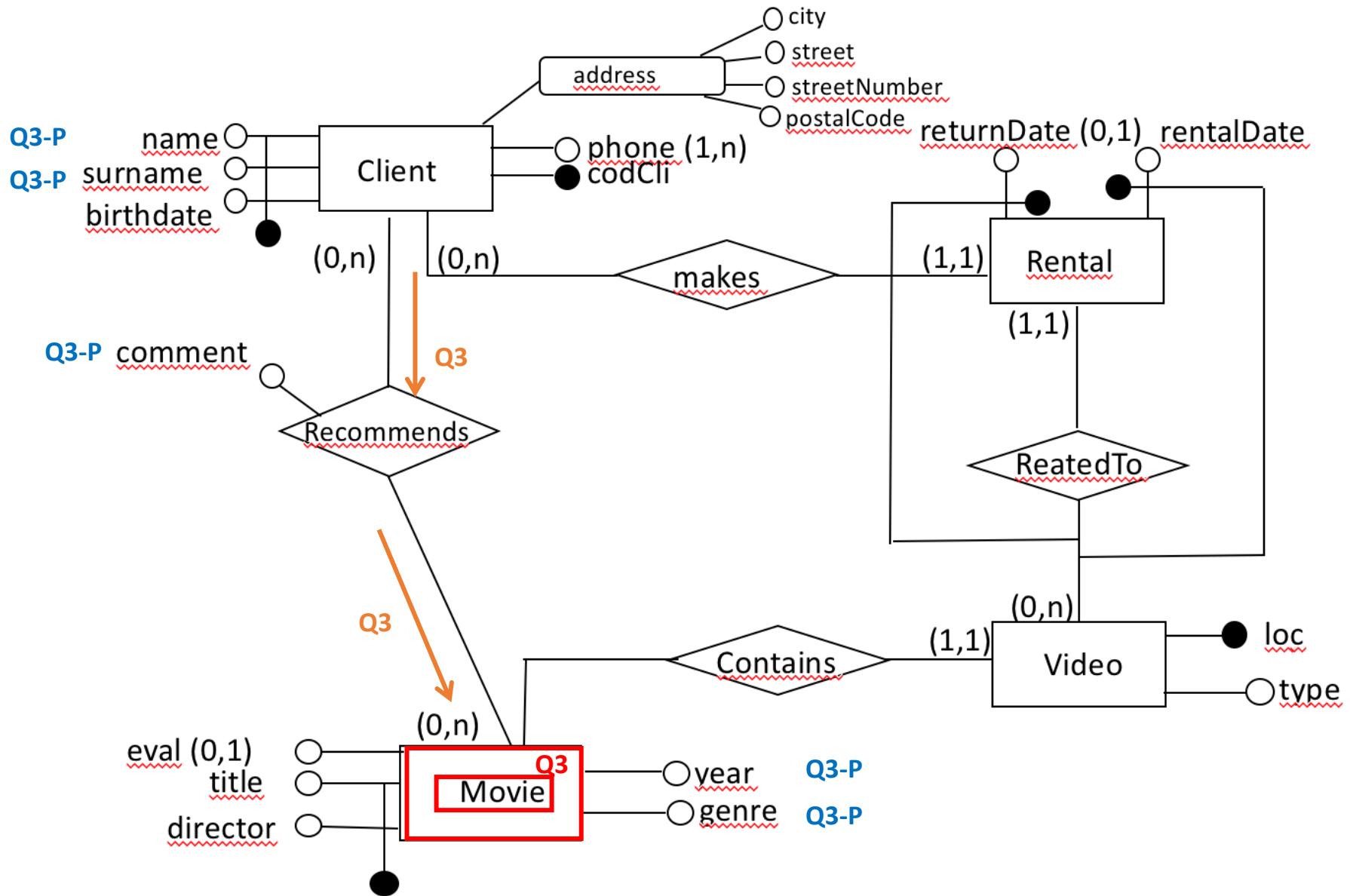


Q2 (Client, [],

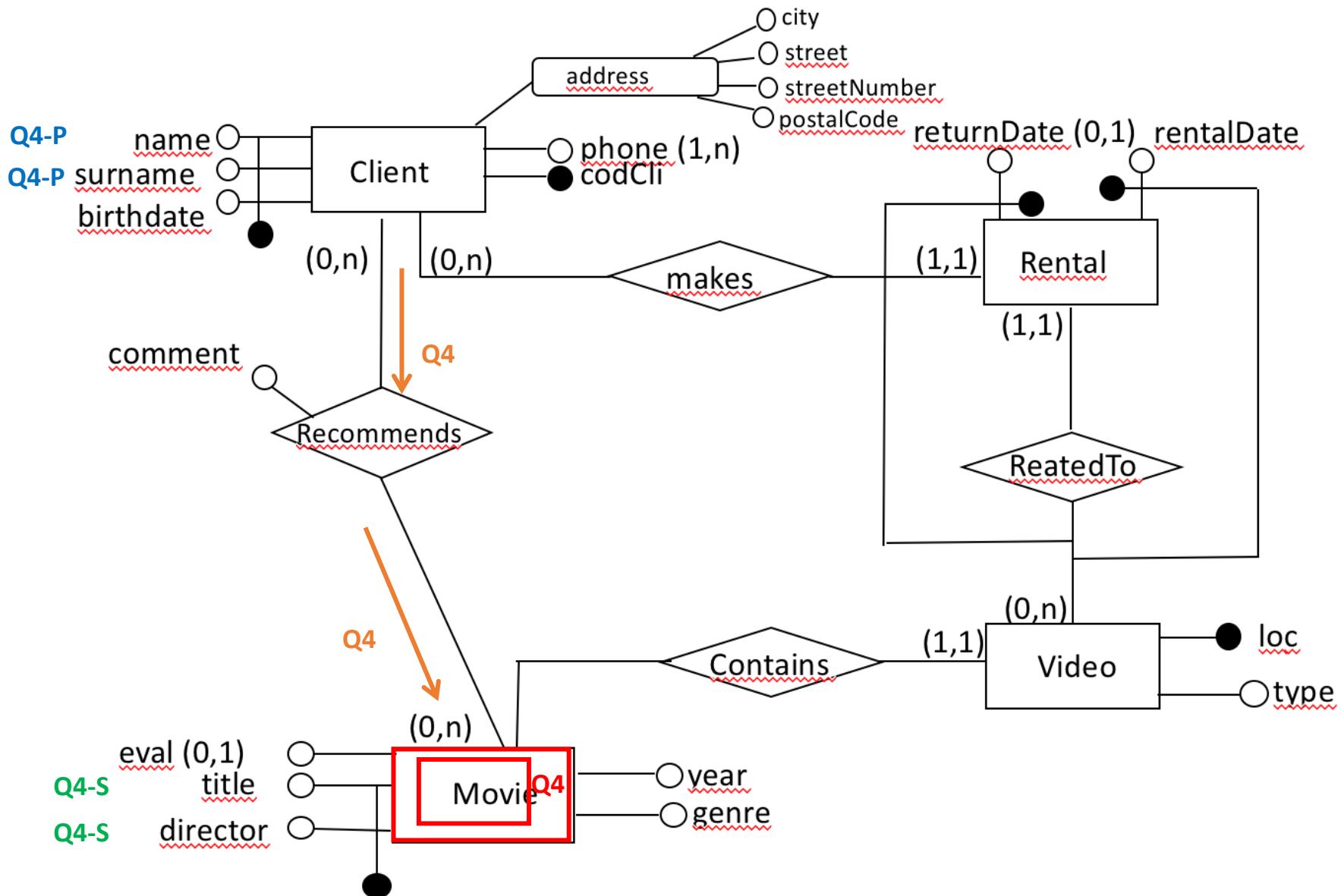
[Client(name, surname)_!, Movie(title, director)_R])



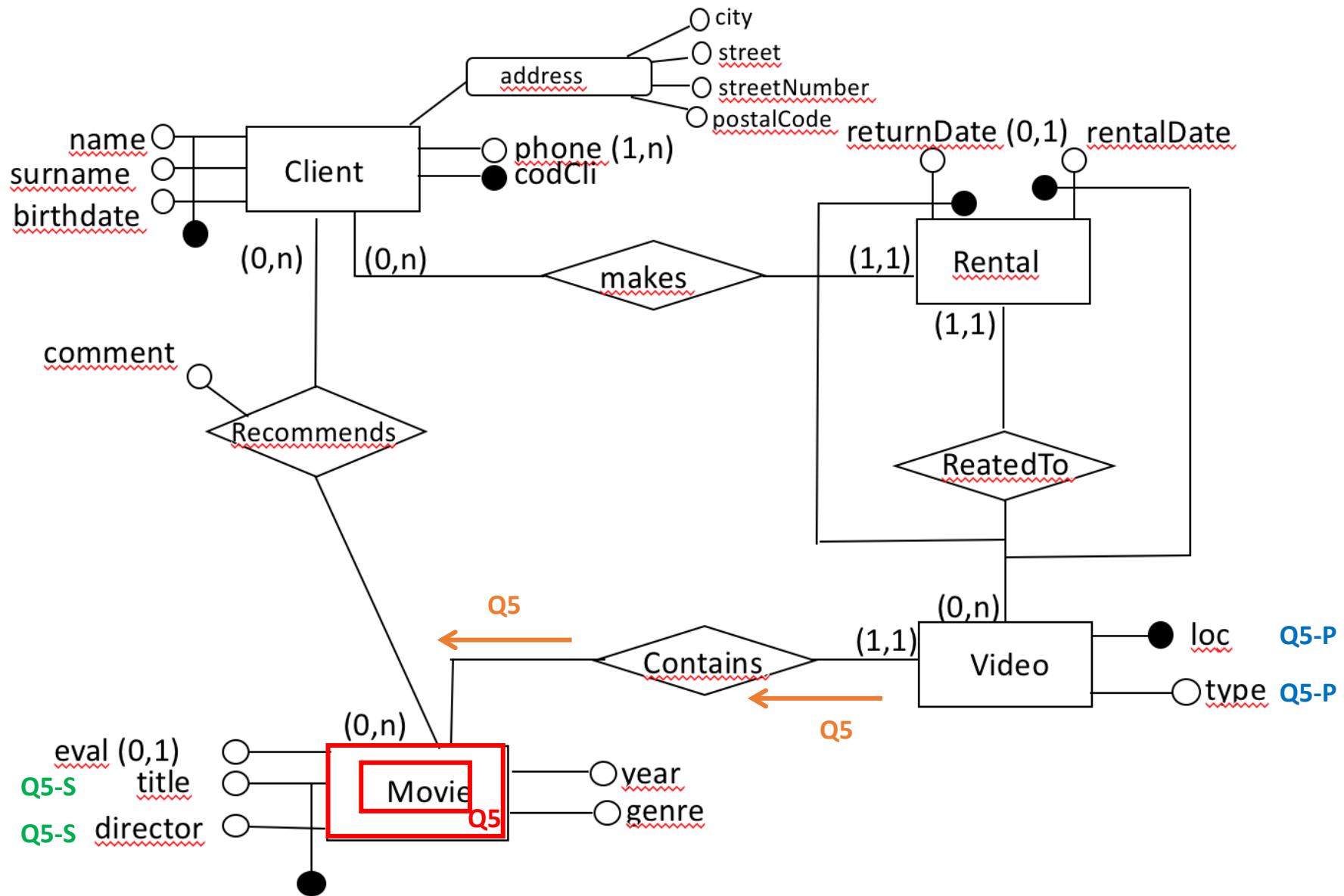
Q3 (Movie, [],
 [Movie(genre, year)_!, Client(name, surname)_R(comment)])



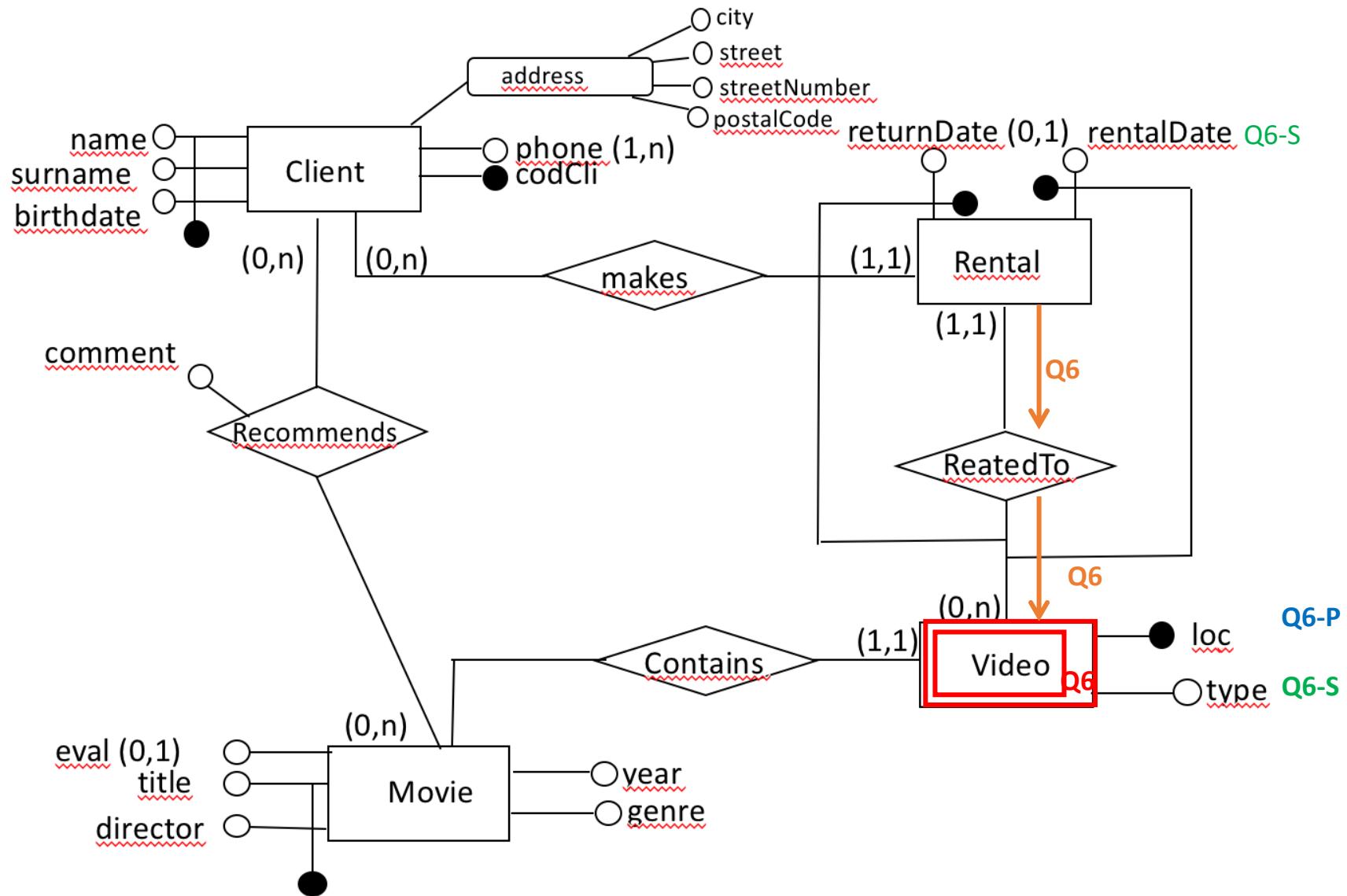
Q4 (Movie, [Movie(title, director)_!], [Client(name, surname)_R])



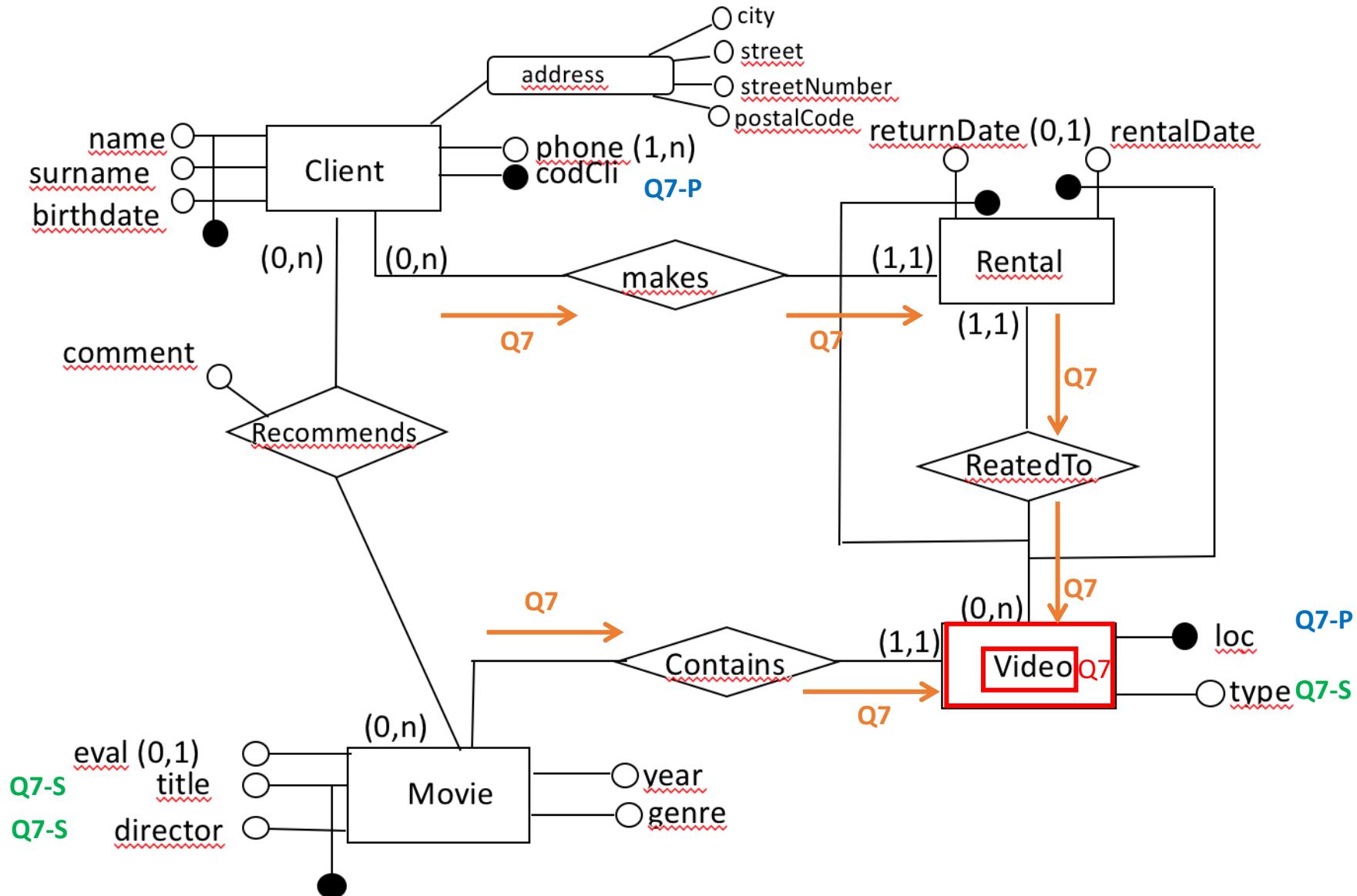
Q5 (Movie, [Movie(title, director)_!], [Video_C])



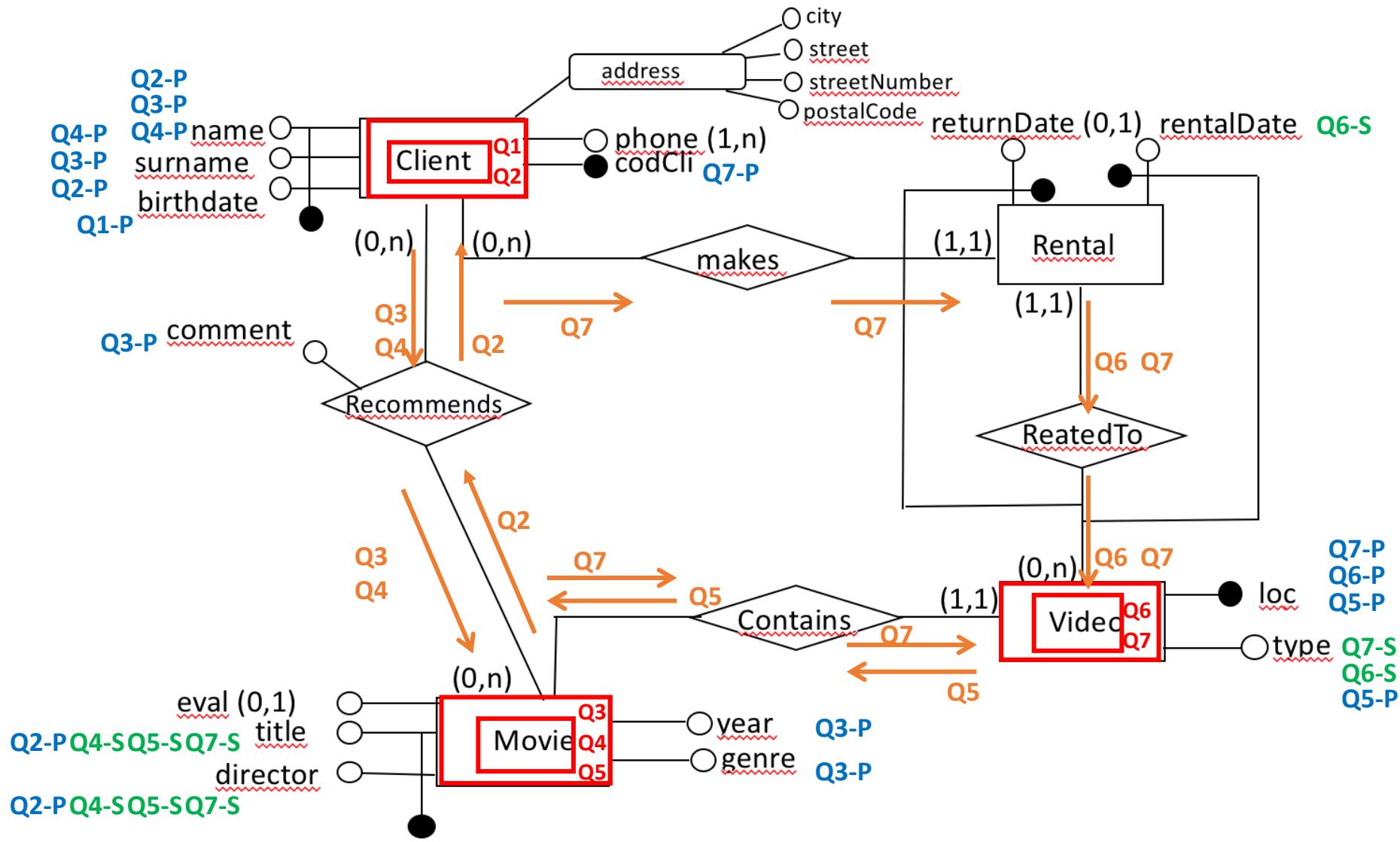
Q6 (Video, [Video(type)_!, Rental(rentalDate)_Rt], [Video(loc)_!])



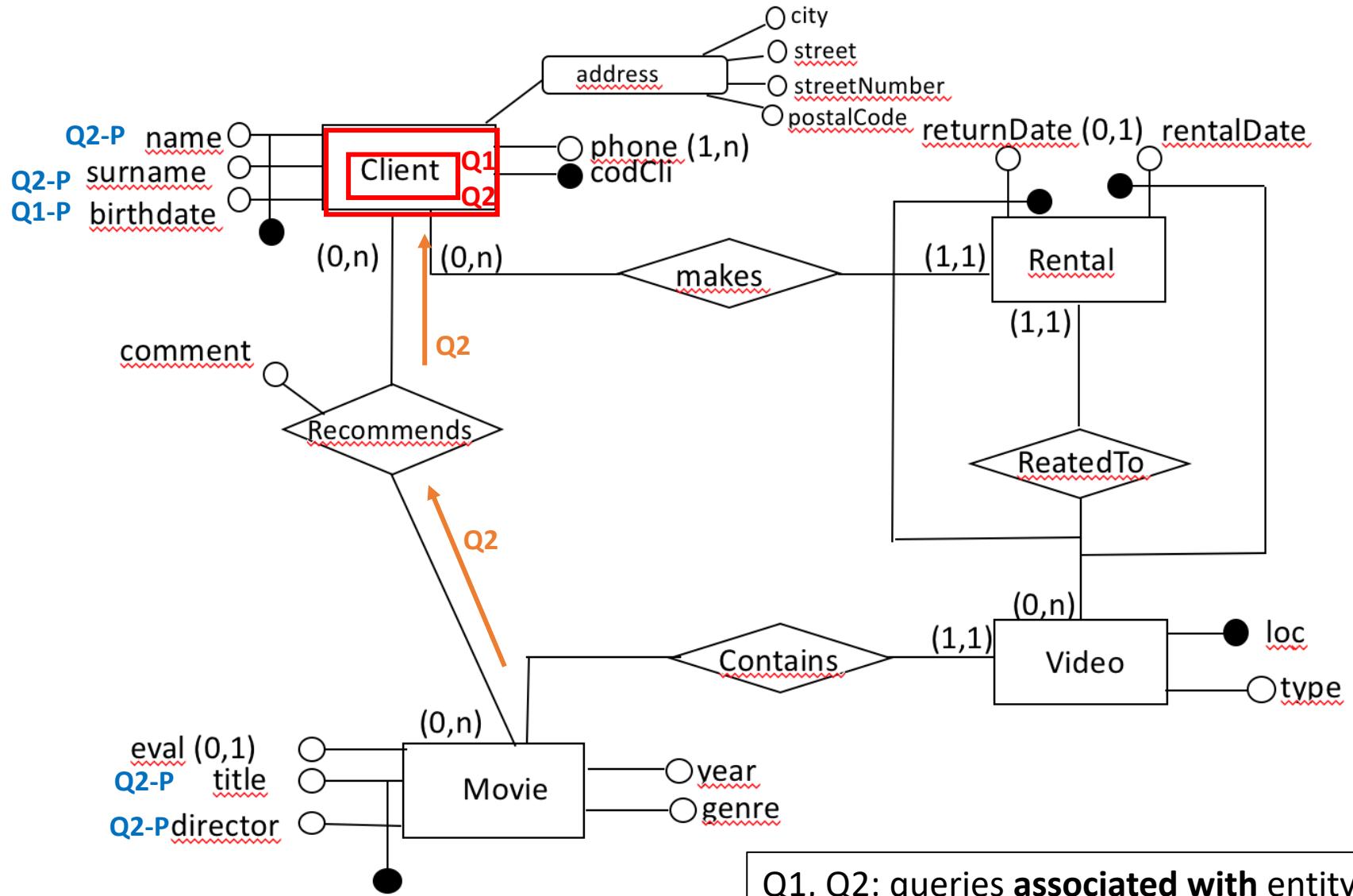
Q7 (Video, [Video(type)_!, Movie(title, director)_C]
[Video(loc)_!, Client(codCli)_M_{Rt}])



Final annotated ER schema

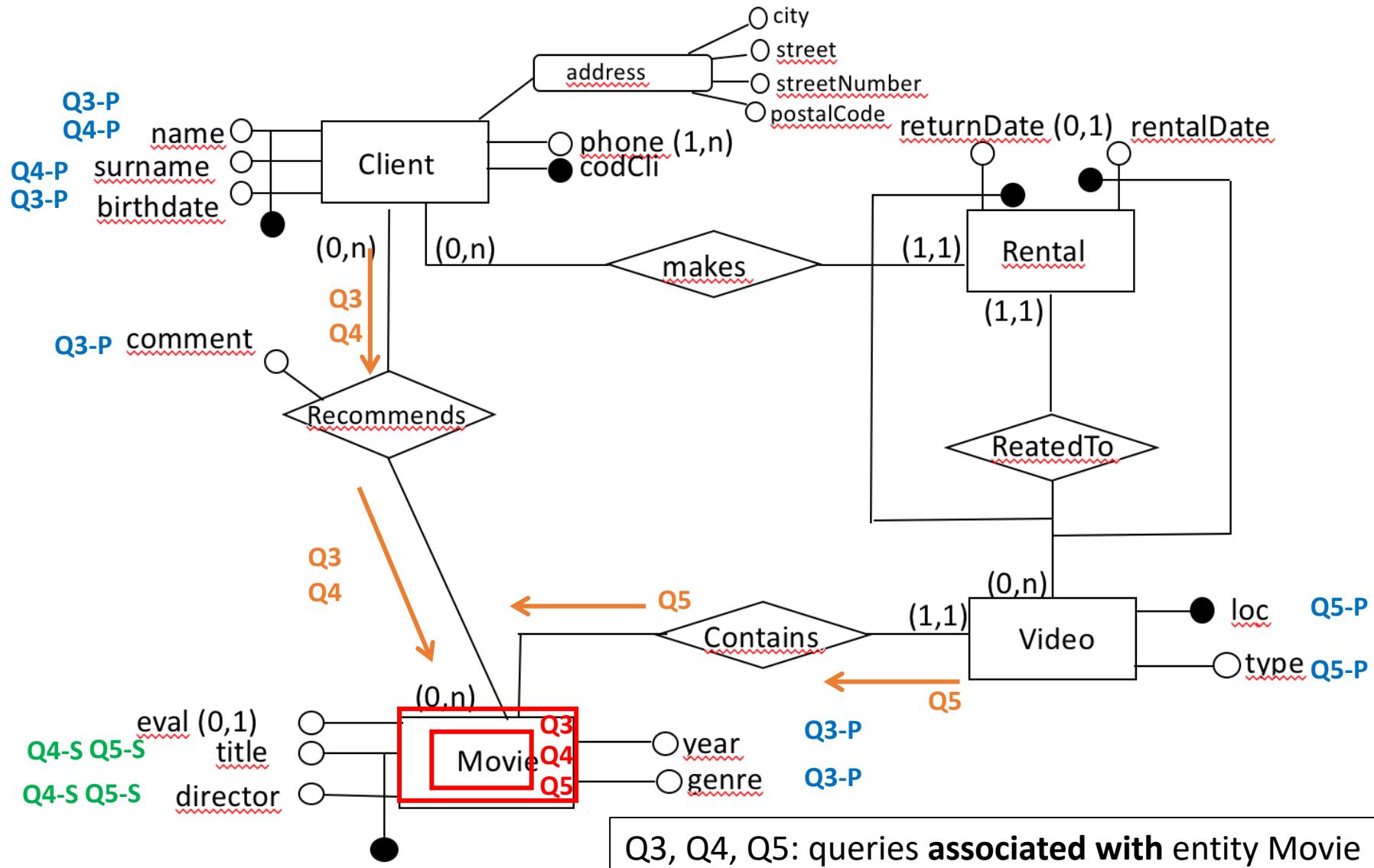


Final annotated ER schema

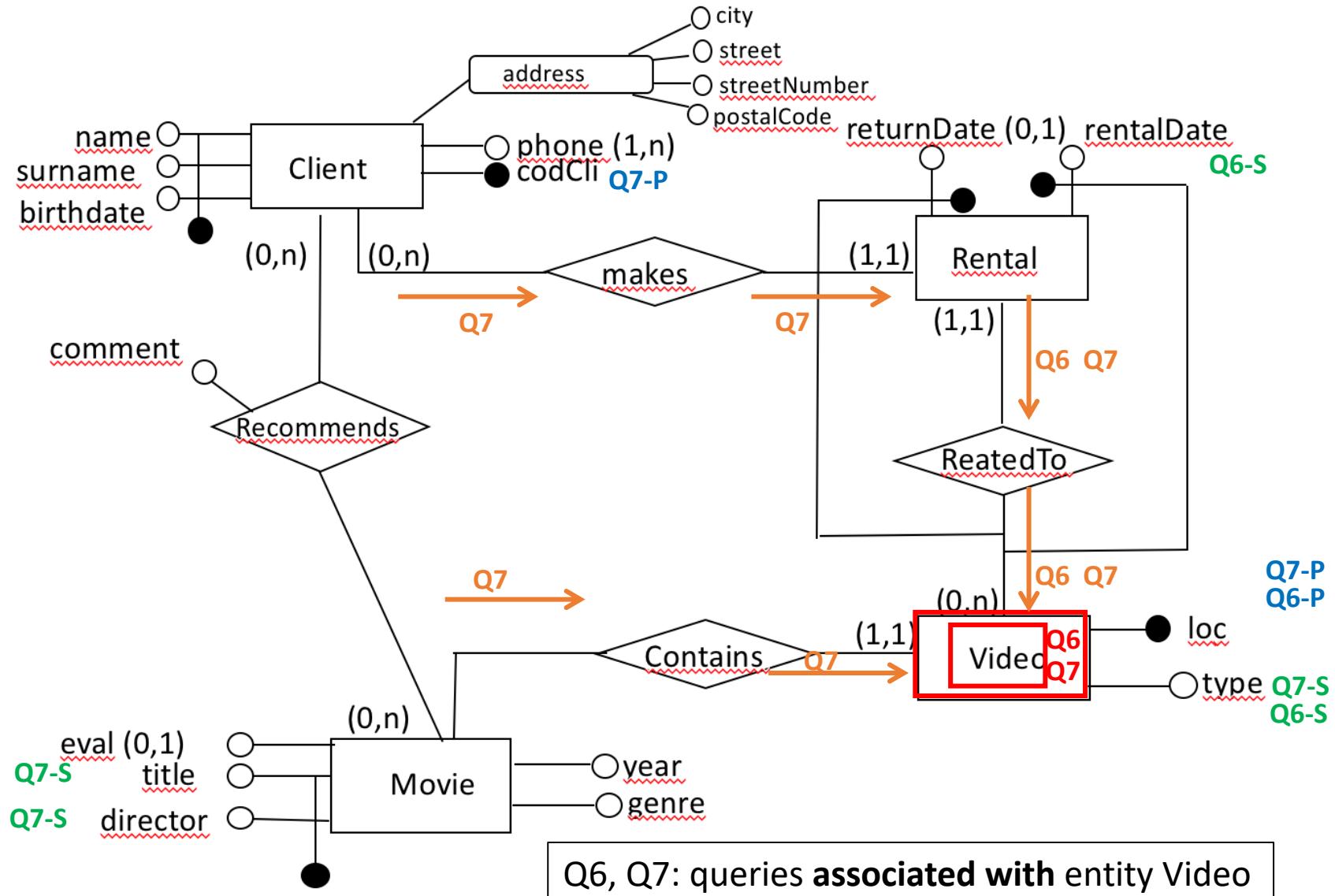


Q1, Q2: queries associated with entity Client

Final annotated ER schema



Final annotated ER schema



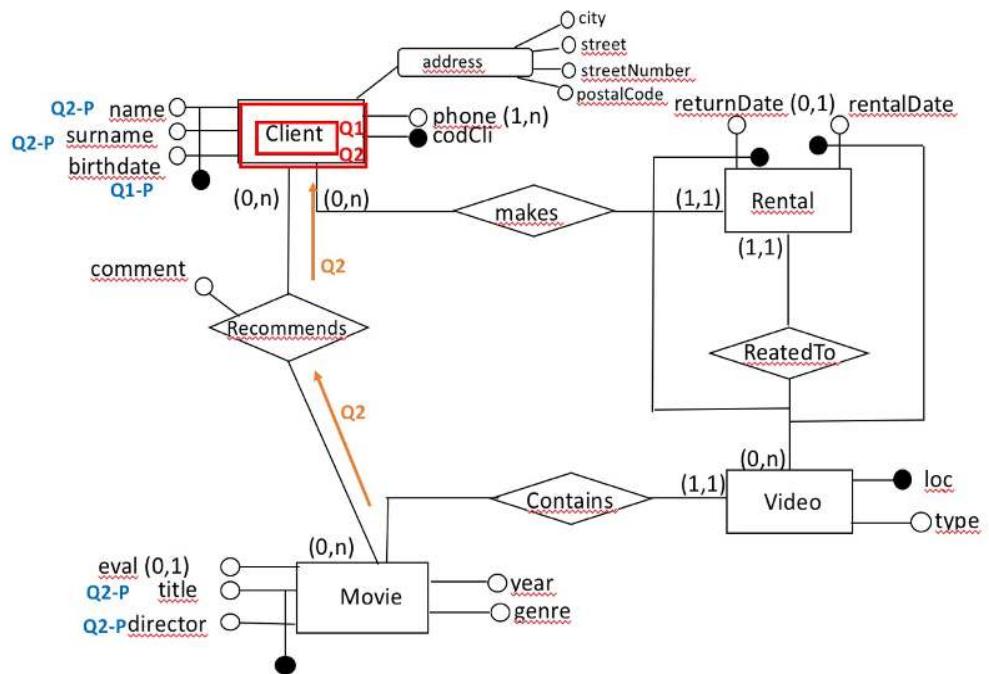
Step 3: translation into a JSON meta-notation

- Create one collection E for each aggregation entity E
- Aggregates inside collection E are characterized by
 - All attributes of entity E annotated with queries associated with E (= their identifier is located inside E)
 - Represented as simple attributes
 - Further attributes corresponding to entities and associations located on paths annotated with one query associated with E
 - Represented as simple or complex attributes depending on the cardinality of associations included in the considered path
 - An id for the aggregate, corresponding to one of the identifiers for E in the ER schema

Aggregation entity Client

- Create collection **Client**
- Queries associated with **Client**
 - Q1, Q2
- Add all the attributes of **Client** annotated by Q1 or Q2
 - name, surname, birthdate, ...
- Now Consider Q1
 - No path entering in Client is annotated by Q1 → no further attributed added

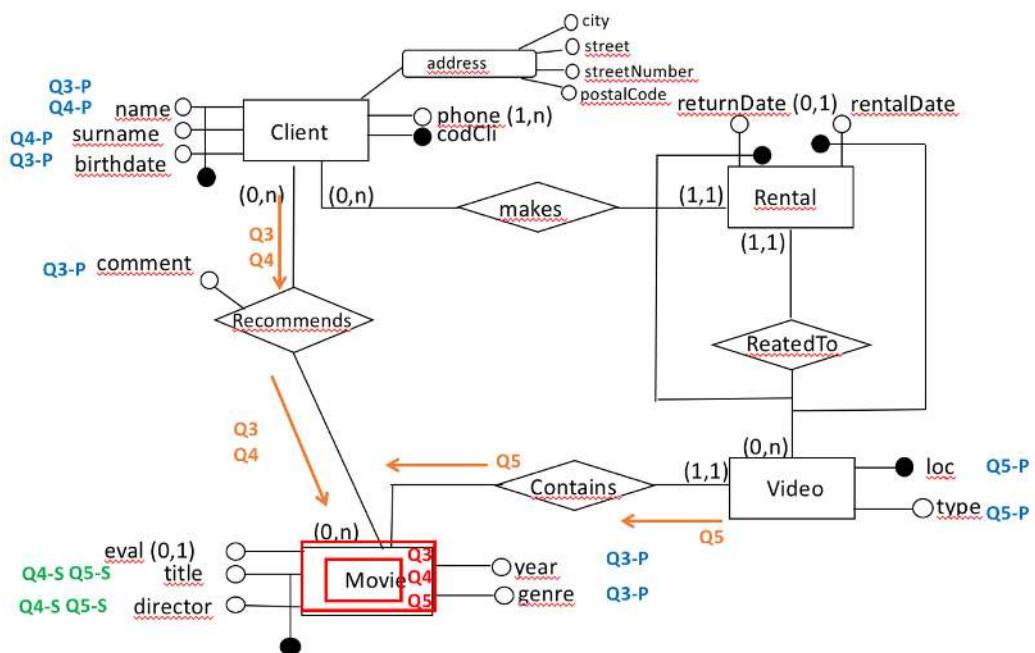
```
client:  
{name, surname, birthdate,  
recommends: [{movie: {title, director}}]  
}
```



- Now consider Q2
 - One single path of length 1 (= one association) annotated with Q2 : **Client – Recommends – Movie**
 - Cardinality constraint for **Client** in association **Recommend**: **(0,n)**
 - Add to **Client** a **set-based attribute** containing tuples composed of attributes of **Recommends** and **Movie** annotated with Q2
 - Client (name, surname, birthdate, recommends: [{movie: {title, director}}])

Aggregation entity Movie

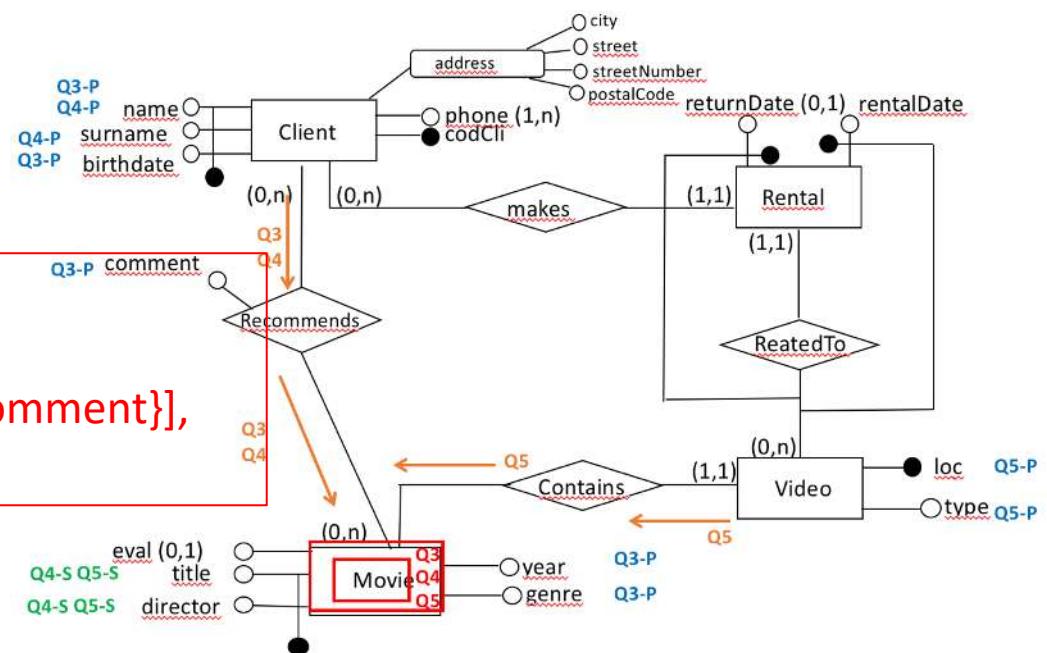
- Create collection **Movie**
- Queries associated **Movie**
 - Q3, Q4, Q5
- Add all the attributes of **Movie** annotated by Q3, Q4, o Q5
 - {title, director, year, genre, ...}
- Now consider Q3
 - One single path of length 1 (= one association) annotated with Q3: **Client – Recommends - Movie**
 - Cardinality constraint for **Movie** in association **Recommend**: **(0,n)**
 - Add to **Movie** a **set-based attribute** containing tuples composed of attributes of **Recommends** and **Client** annotated with Q3
 - movie:
 - {title, director, year, genre,
 - recommended_by**: [{name, surname, comment}],...



Aggregation entity Movie

movie:

{title, director, year, genre,
 recommended_by: [{name, surname, comment}],
 contained_in: [{video: {loc, type}}]}



- Now consider Q4

- One single path of length 1 annotated with Q4: Client – Recommends - Movie
- All the attributes annotated with Q4 are also annotated with Q3, thus they have already been inserted in the schema
- Nothing change

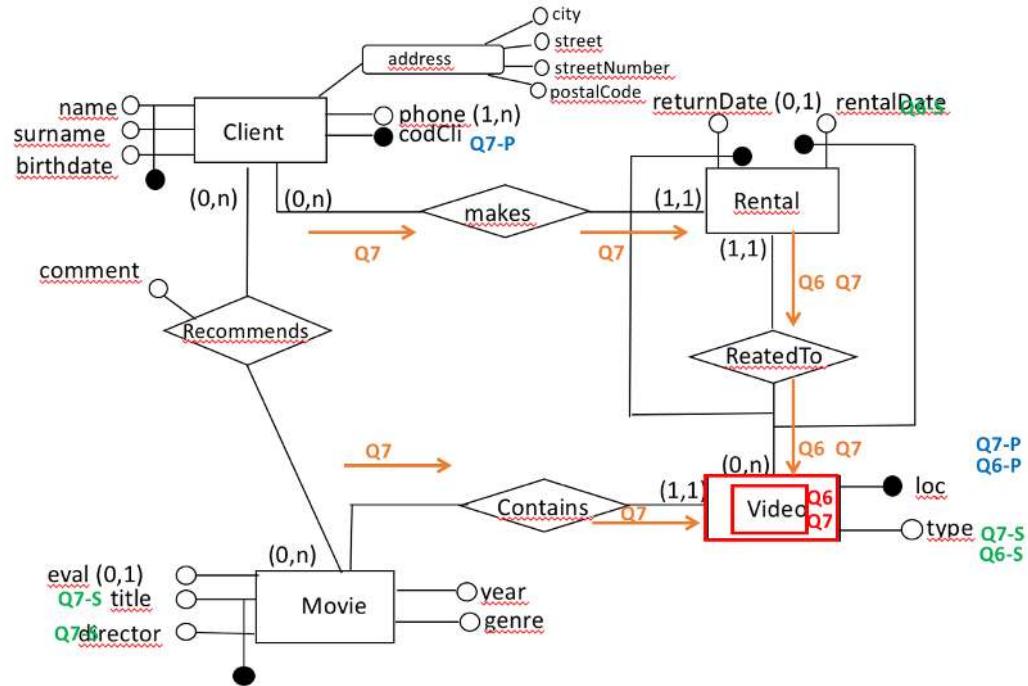
- Now consider Q5

- One single path of length 1 annotated with Q5: Video – Contains – Movie
- Cardinality constraint for Movie in association Contains: (0,n)
- Add to Movie one **set-based attribute** containing tuples composed of attributes of Contains and Video annotated with Q5
- movie:

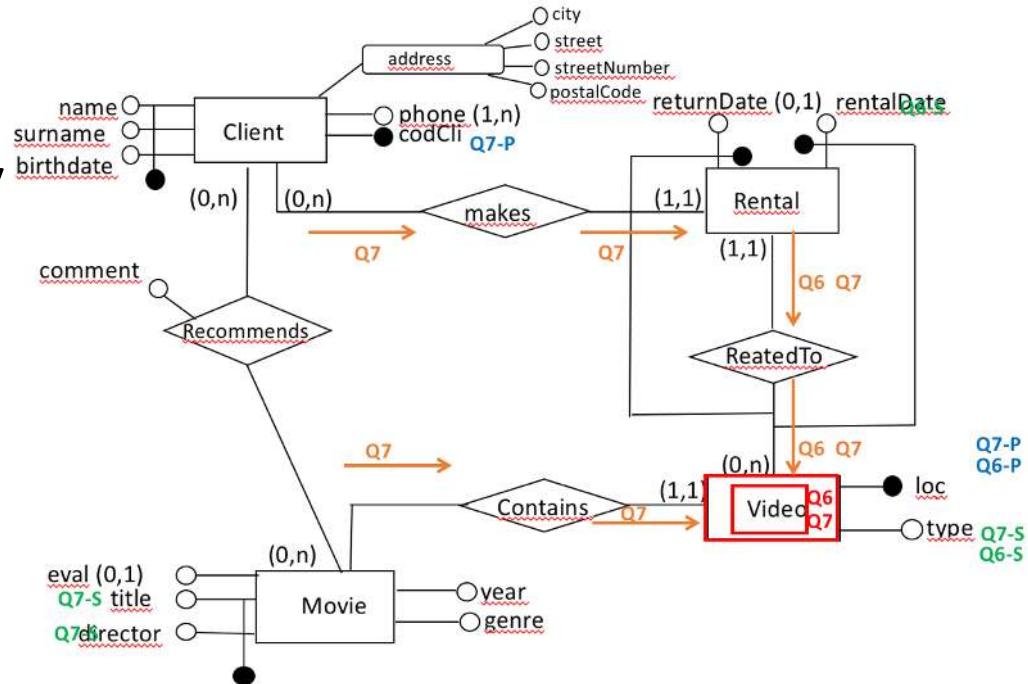
{title, director, year, genre,
 recommended_by: [{name, surname, comment}],
 contained_in: [{video: {loc, type}}]}

Aggregation entity Video

- Create collection **Video**
- Query associated with **Video**
 - Q6, Q7
- Add all the attributes of **Video** annotated by Q6, Q7
 - {loc, type, ...}
- Now consider Q6
 - One single path of length 1 annotated with Q3: **Rental** – **RelatedTo** – **Video**
 - Cardinality constraint for **Video** in association **RelatedTo**: $(0,n)$
 - Add to **Video** one **set-based attribute** containing tuples composed of attributes of **Rental** and **RelatedTo** annotated with Q5
 - **video:**
 $\{\text{loc, type, rentals: [\{rental: \{rentalDate\}\}] , ...}$

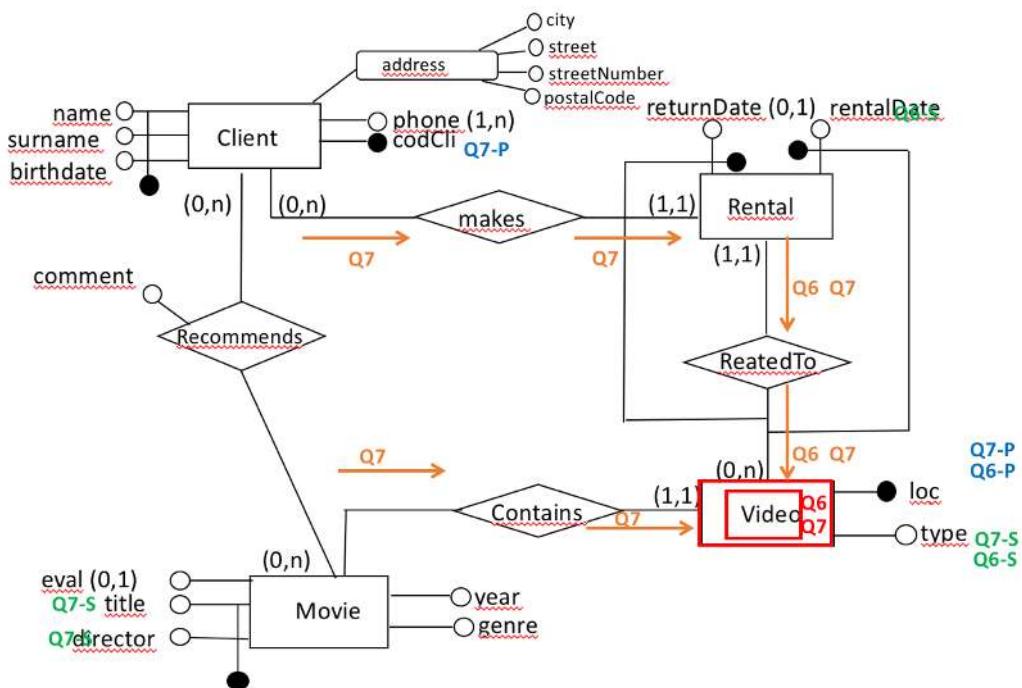


Aggregation entity Video



- Now consider Q7, path **Client –Makes-Rental-RelatedTo-Video**
 - Path of length 2: **Client –Makes-Rental** + **Rental-RelatedTo-Video**
 - Now we analyze each subpath of length 1, starting from **Video**
 - Rental-RelatedTo-Video**: already considered in Q6
 - No attribute in **Rental** are annotated with Q7 → nothing to do
- Client-Makes-Rental**: cardinality constraint **(1,1)** from the **Rental** side
- Add the attributes annotated with Q7 in **Client** and **Makes** (only codCli) to the set-based attribute «**rentals**» in **Video**
- video:{loc, type, rentals: [{rental: {rentalDate, codcli}}]}, ...

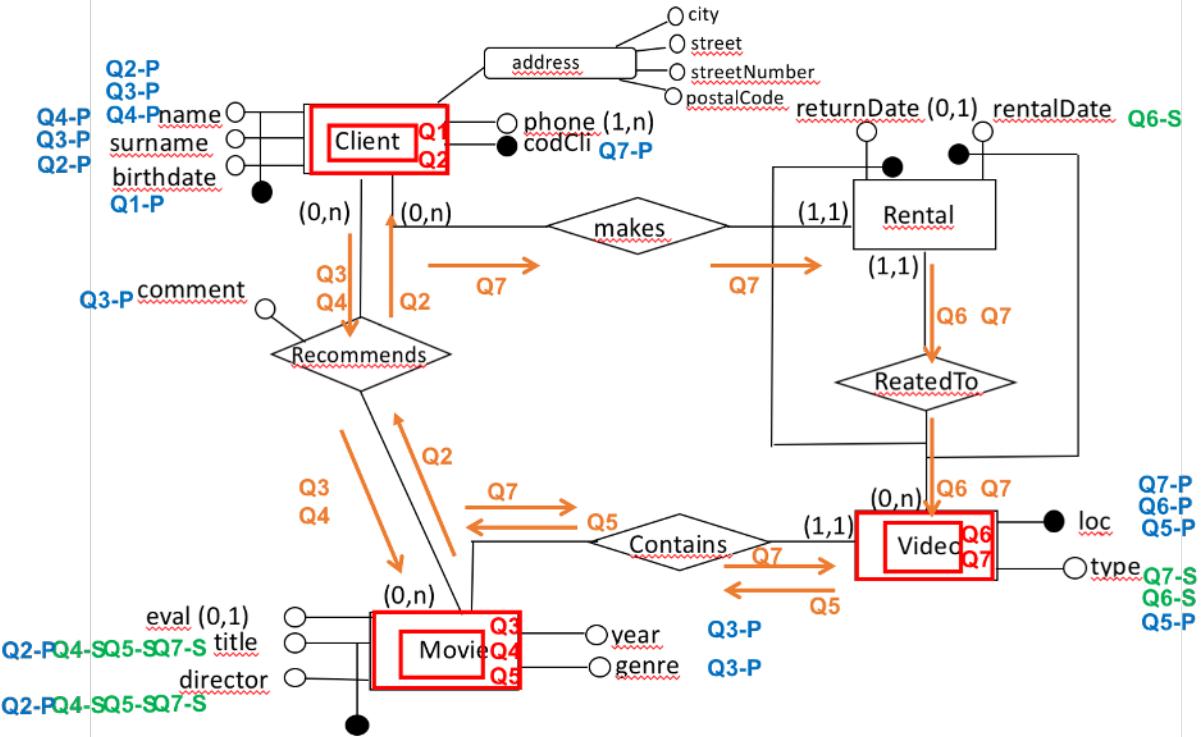
Aggregation entity Video



- Now consider Q7, path Movie-Contains-Video
 - Path with length 1
 - Cardinality constraint for **Video** in association **Contains**: **(1,1)**
 - Add to **Video** the **attributes of Movie** and **Contains** annotated with Q7
 - video: {loc, type, rentals: [{rental: {rentalDate, codcli}}]}, **title, director**

```
video: {loc, type, rentals: [{rental: {rentalDate, codcli}}]}, title, director}
```

Final schema



- client: {name, surname, birthdate, recommends: [{movie: {title, director}]}]}
 - Q1, Q2
- movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}]}
 - Q3, Q4, Q5
- video: {loc, type, rentals: [{rental: {rentalDate, codCli}}]}, title, director}
 - Q6, Q7

Final schema – remarks (1)

- client: {name, surname, birthdate, recommends: [{movie: {title, director}}]}
 - movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{video: {loc, type}}]}
 - video: {loc, type, rentals: [{rental: {rentalDate, codCli}}]}, title, director}
-
- These elements for items in a collection are optional, they can be added or not
 - Possible alternative schema (no movie, video, and rental element, client element added):
-
- client: {name, surname, birthdate, recommends: [{title, director}]}
 - movie: {title, director, year, genre, recommended_by: [{client: {name, surname, comment}}], contained_in: [{loc, type}]}
 - video: {loc, type, rentals: [{rentalDate, codCli}]), title, director}

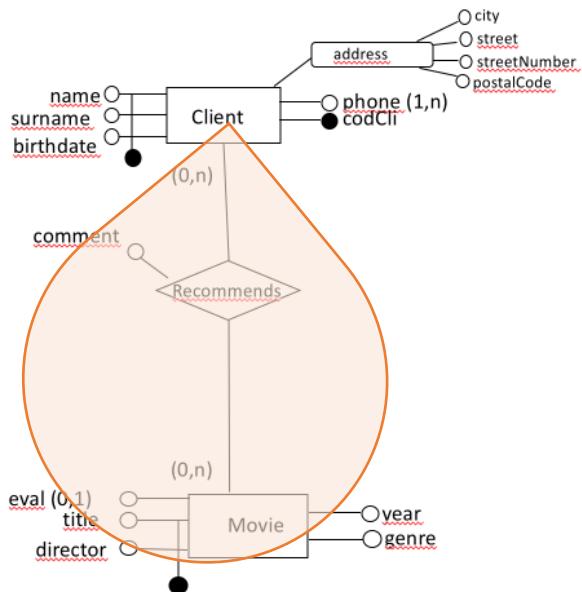
An app for aggregate-oriented logical design

- <https://amazing-benz-6643f3.netlify.app/>
- Example files:
https://www.dropbox.com/sh/b64lbe20a0hm1he/ADp_XJfVakMvjqfdUDujDeea?dl=0

Architecture of aggregate-oriented NoSQL systems

Back to aggregates (logical view)

Aggregate-oriented **schema** (using the meta-notation, for schema representation in aggregate-oriented NoSQL systems – simplified **JSON schema information**)



client: { codCli, name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [movie: {comment, title, director,...}]
}

JSON: **instance level** (for data representation and exchange)

```
{ "codcli": 375657,  
  "name": "John",  
  "surname": "Black",  
  "birthdate": "15/10/2000",  
  "address": {"city": "Genoa", "street": "Via XX Settembre",  
             "streetNumber": 15, "postalCode": 16100},  
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
                    "director": "quentin tarantino"}},  
            {"movie": {"comment": "very nice", "title": "pulp fiction",  
                    "director": "quentin tarantino"}}]}
```

Back to aggregates (logical view)

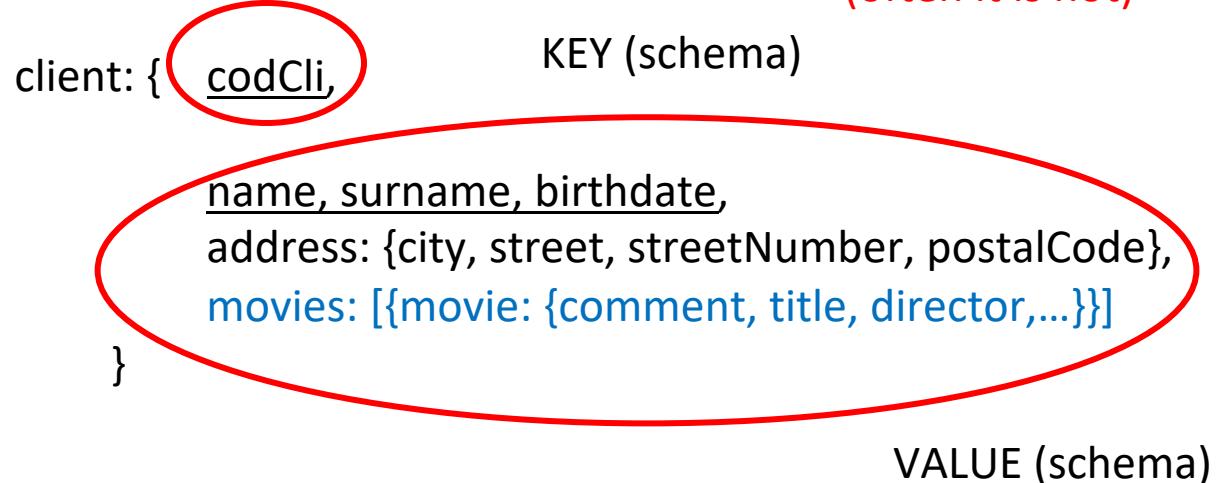
An aggregate is a **data unit** with a **complex structure**

Each aggregate = (at least) one **(key, value)** pair

key = partitioning key

JSON: **instance level**

Aggregate-oriented **schema**



KEY (instance)

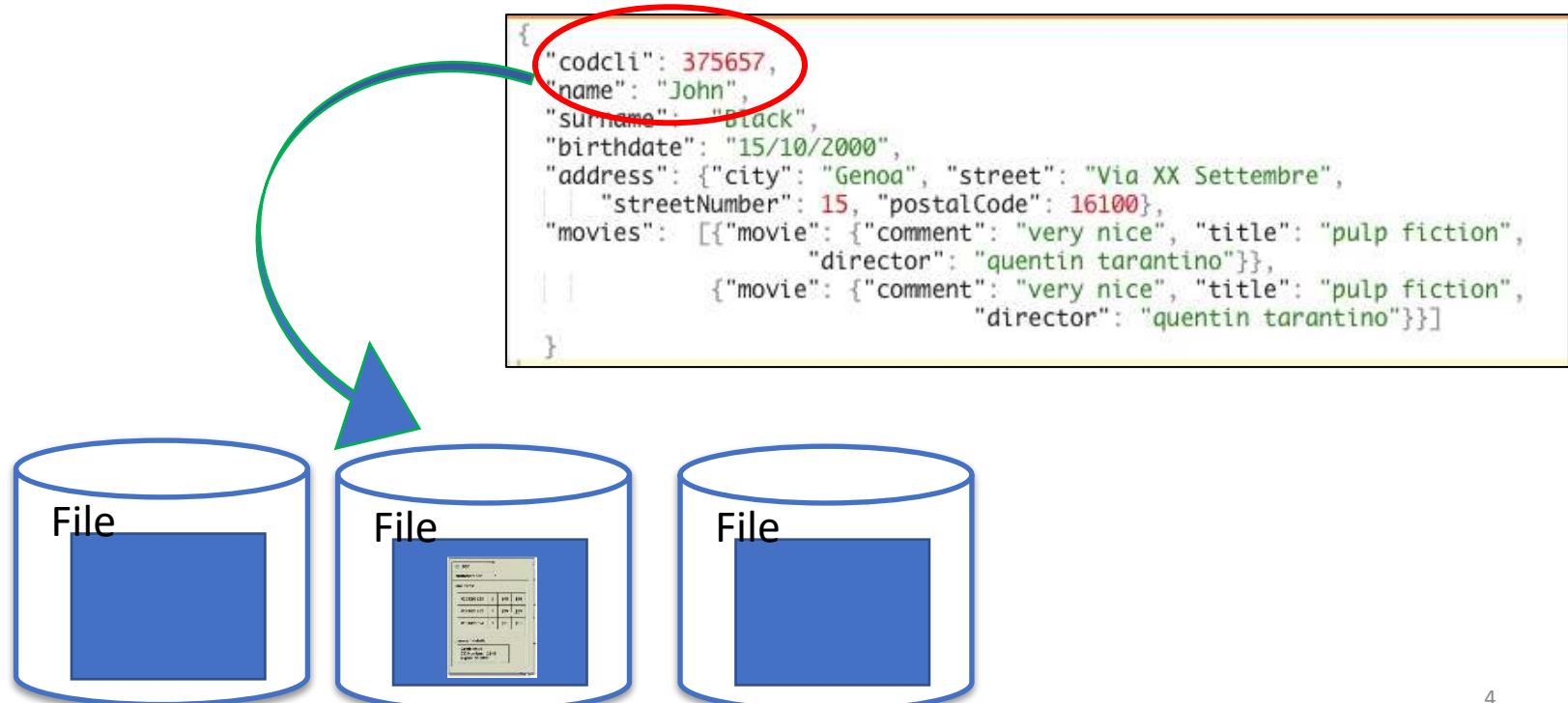
```
{"codcli": 375657,  
  "name": "John",  
  "surname": "Black",  
  "birthdate": "15/10/2000",  
  "address": {"city": "Genoa", "street": "Via XX Settembre",  
             "streetNumber": 15, "postalCode": 16100},  
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
                     "director": "quentin tarantino"}},  
            {"movie": {"comment": "very nice", "title": "pulp fiction",  
                     "director": "quentin tarantino"}}]}
```

VALUE (instance)

The key might be an identifier but it is not mandatory (often it is not)

Aggregates (physical view)

- All the data about a unit of interest (an aggregate) are **kept together** on the same node
- Partitioning separates **different units** (different aggregates) on different nodes
- The aggregate key used as partitioning key



Summary

- **Later:**
 - presentation of specific aggregate-oriented NoSQL systems
 - customization of the aggregate-oriented design methodology to represent aggregates inside each system
- **Now:**
 - reference architectural properties of aggregate-oriented NoSQL systems

Aggregate-oriented data stores: architectural features in short

| Feature | In aggregate-oriented data stores |
|------------------------|---|
| Reference architecture | Often P2P |
| Reference scenarios | Transactional (read/write intensive) |
| Partitioning | Often hash-based or range-based, efficient usage of RAM and indexes |
| Replication | Often leader-less |
| Consistency | Often eventual consistency |
| Availability | Often high, in general tunable |
| Fault tolerance | High (often no master, P2P ring) |
| Transactions | Usually , no ACID transactions |
| CAP theorem | AP or CP |

Distributed architecture

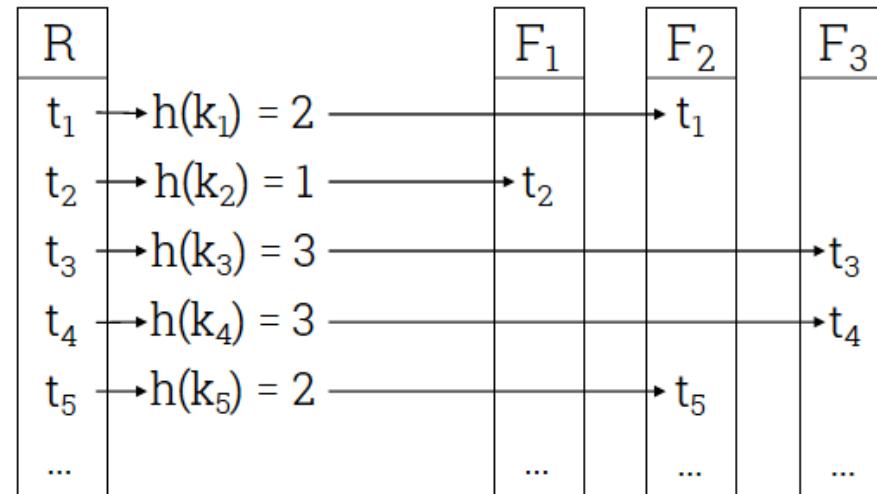
- Often (but not always) there is **no master** and **every node is a peer**
- All the nodes in a cluster play the same role
- *Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster*
- When a node goes down, read/write requests can be served from other nodes in the network
- You can add nodes to the cluster to improve the capacity of the cluster (scalability)
- Gossip protocols for intra-ring communication

(Back to) Hash-based partitioning

- Many (but not all) aggregate-oriented systems implement a distributed storage based on hash-based partitioning

(Back to) Hash-based partitioning

- Applies a hash function to some attribute that yields the partition number in $\{1, \dots, p\}$



(just one node has to be accessed
and the node contains all data of
interest)

Distributes data evenly if hash function is good
Good for point queries on key and joins
Not good for range queries and point queries not on key

(all nodes have to be accessed but only few nodes
contain data of interest)

Hash-based partitioning: rebalancing

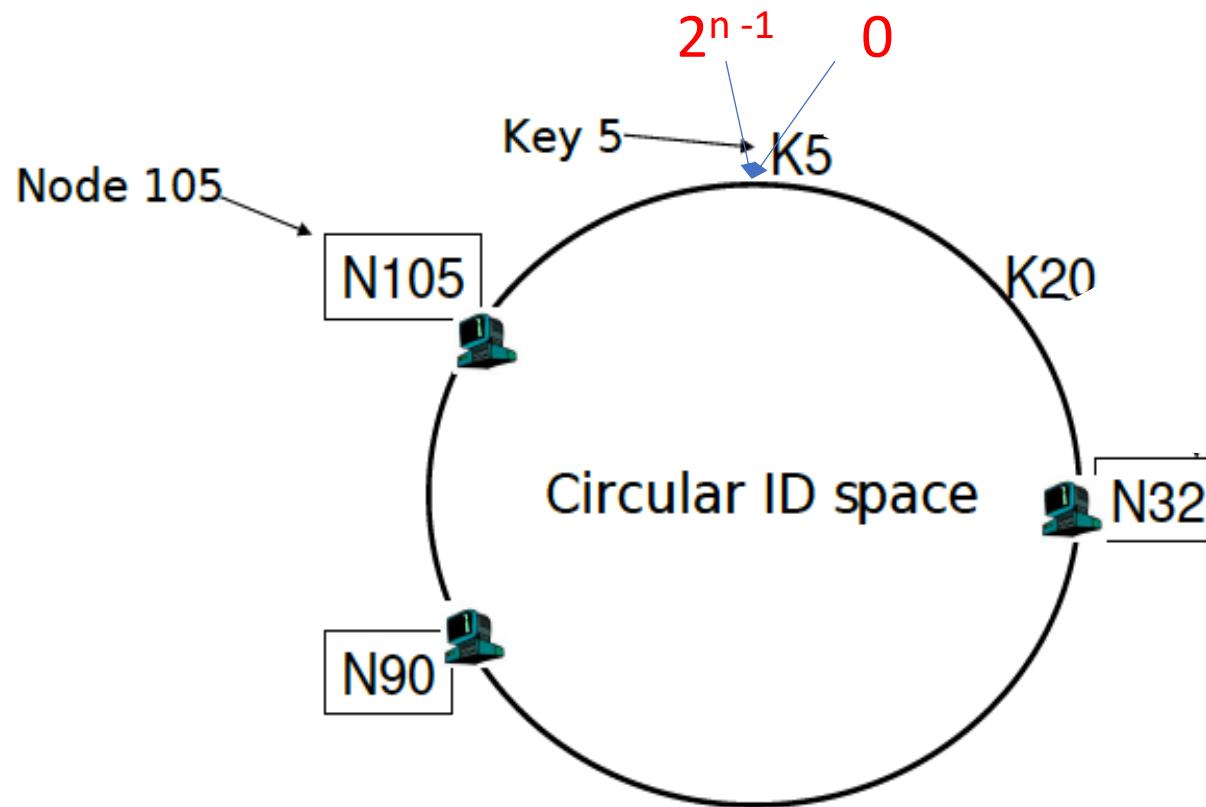
- Suppose a hash-based partitioning is applied:
 - $H(v) = v \bmod p$, where p is the number of nodes
- If the number of nodes changes, most of the data must be moved
- *Before $p = 10$*
 - $H(10) = 0$
 - $H(11) = 1$
 - $H(12) = 2$
- *After $p = 11$*
 - $H(10) = 10$
 - $H(11) = 0$
 - $H(12) = 1$
- Typical situation in a large scale distributed data system
- New approaches have to be adopted for better scaling

Partitioning with consistent hashing

- Initially proposed in the context of distributed caching systems
- Use a simple, non-mutable hash function h that maps both the server address and the aggregate keys to the same large address space A
- Assuming to use n bits for representing hash values, hash values correspond to the interval $[0, 2^n-1]$
- The hash function can be implemented in different ways
- Example: $H(k) = k \bmod 2^n$

Consistent hashing: ring

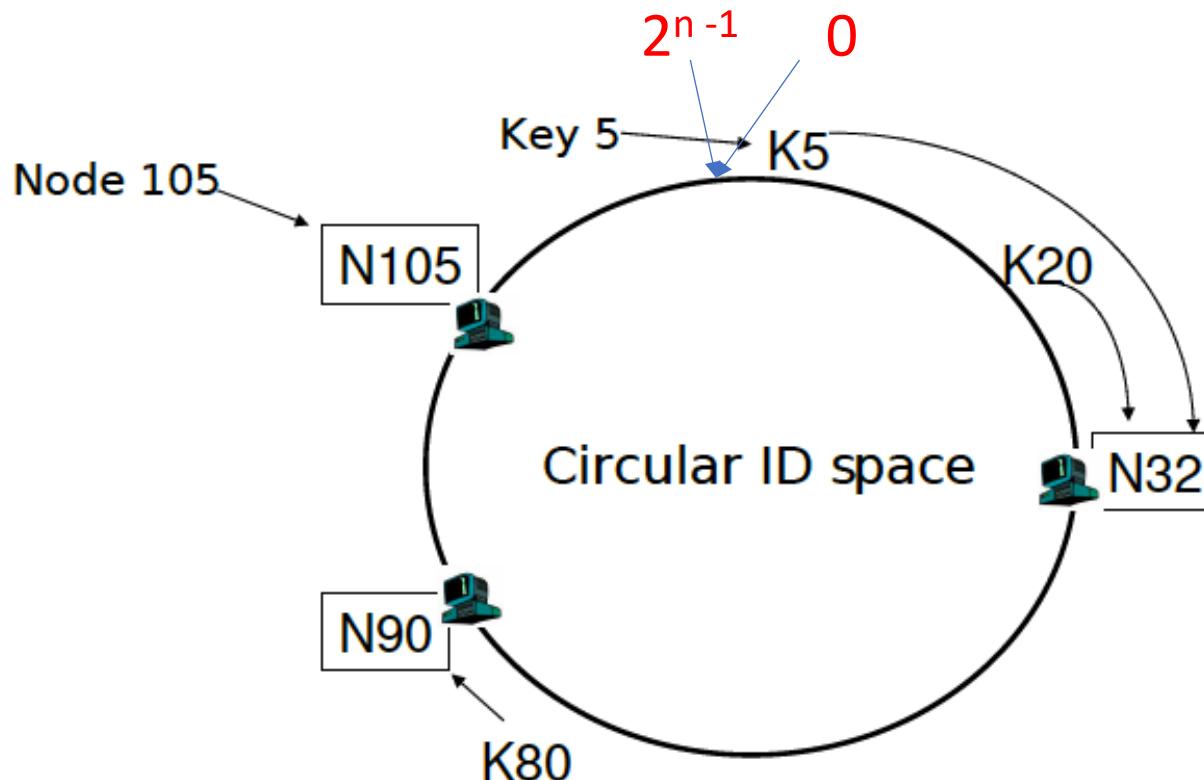
- Organize A as a ring, scanned in clockwise order
- Each element has a successor, the successor of $2^n - 1$ being 0



Consistent hashing: storage rule

- Each node is associated with ad id, used for hashing (no details on that)
- Aggregates with key value hashed into k are assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space (successor node of k)
- If S and S' are two adjacent nodes (clock-wise) in the ring, all the keys in range $(h(S), h(S')] are mapped to $S'$$

Consistent hashing: example

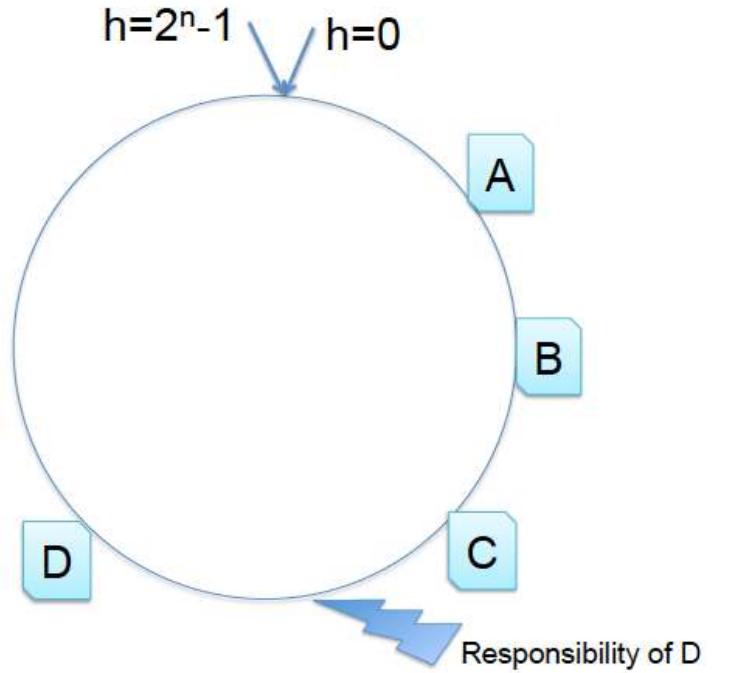


A key is stored at its successor: node with next higher ID

Consistent hashing: scalability

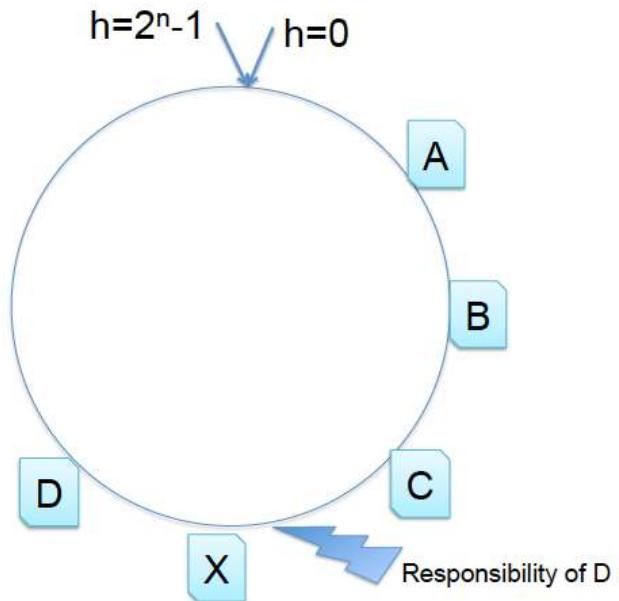
- When a new server is added, we do not need to rehash the whole data set
- Instead, the new server takes place at a position determined by the hash value on the ring, and part of the objects stored on its predecessor must be moved
- The **reorganization is local**, as all the other nodes remain unaffected

Consistent hashing: scalability



Redistribute the load at D

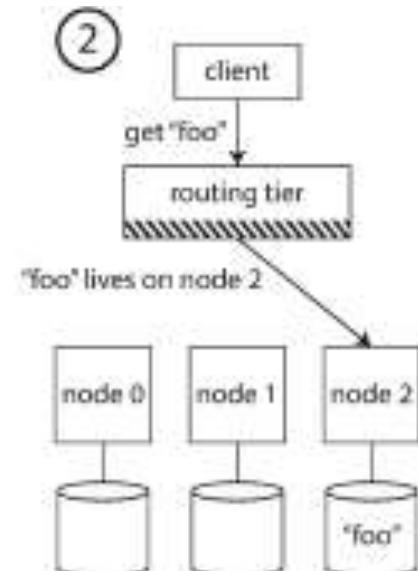
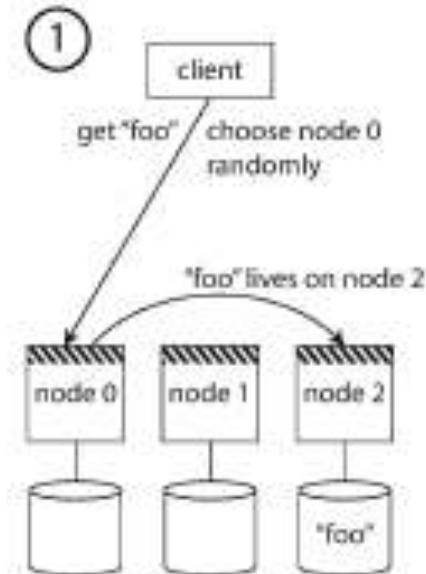
When X joins:
compute the hash value of X on
the ring $h(X)$



Consistent hashing: routing

- Main question: [where is the hash directory](#) (servers locations)?

- Each node locally records (some) information about the hash directory
 - Typical approach in P2P architectures
- On a specific ("Master") node, acting as a load balancer
 - raises scalability issues
 - typical approach in Master-slave architectures (do you remember HDFS?)



Consistent hashing: routing (under option 1)

- Suppose a client wants to read items with partition key equal to k
- Assume a client knows only one node S but it does not know how to access $h(k)$
- Remember that in a P2P network, requests can be sent to any node
- How is it possible to locate the node storing k , starting from S ?

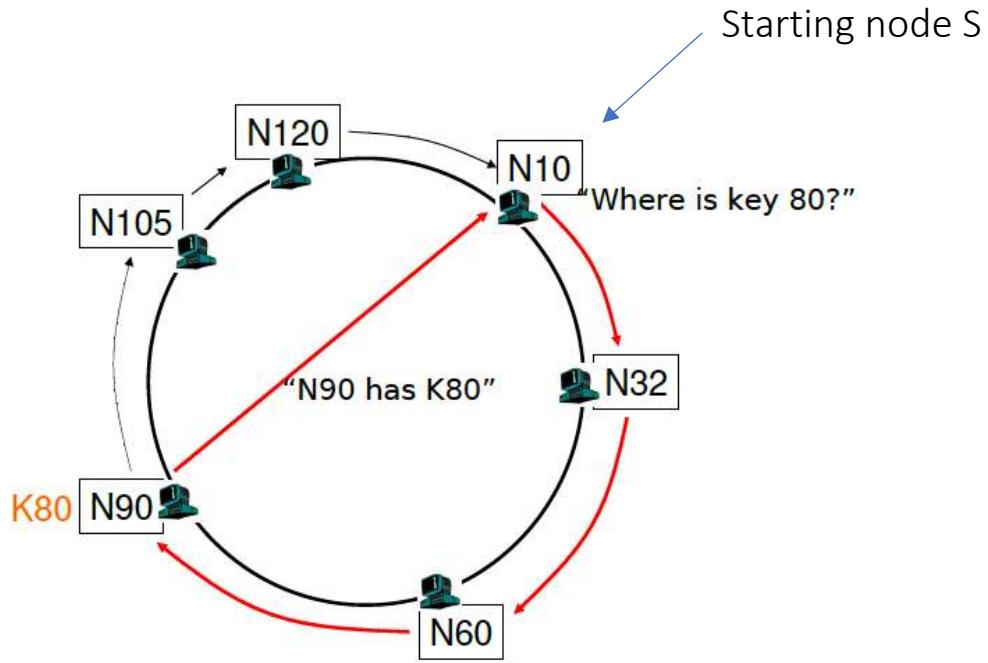
Consistent hashing: routing (under option 1)

- a) Each node records its successor on the ring
- b) Each node records $\log(n)$ carefully chosen other nodes
- c) Full duplication of the hash directory at each node

Consistent hashing: routing (under option 1.a)

Each node records its successor on the ring

- The starting node S sends a message to its successor
- Hop-by-hop from there
- This is $O(N)$, where N is the number of nodes, no good
- Low maintenance protocol (smallest amount of metadata at each node – address of the success)

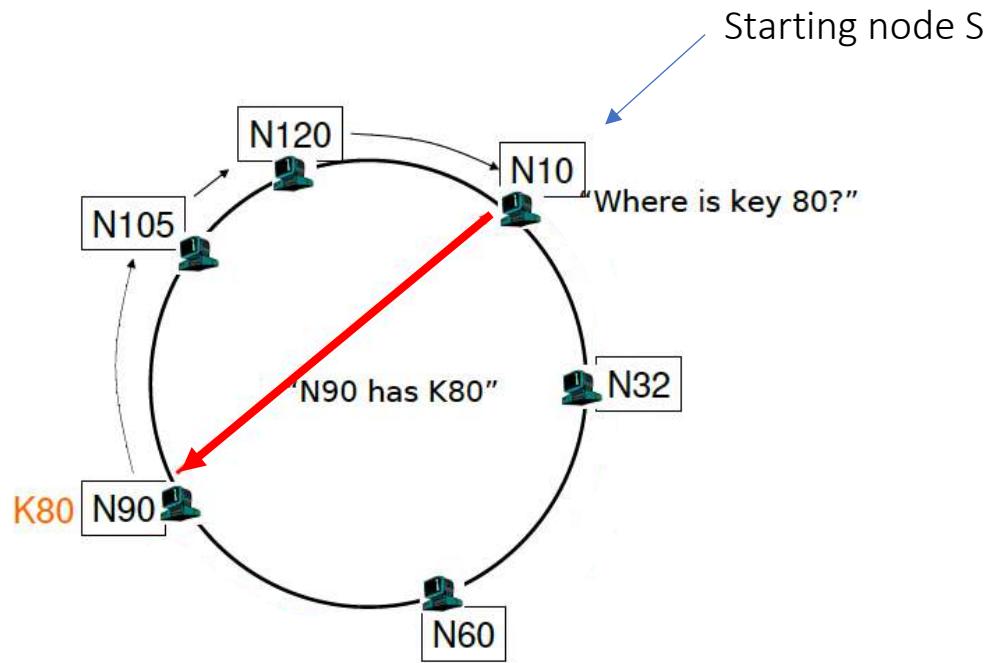


```
// ask node n to find the successor of id
n.find_successor(id)
if (id ∈ (n, successor])
    return successor;
else
    // forward the query around the circle
    return successor.find_successor(id);
```

Consistent hashing: routing (under option 1.c)

Full duplication of the hash directory at each node

- Ensures 1 message for routing
- Heavy maintenance protocol which can be achieved through **gossiping** (broadcast of any event affecting the network topology)

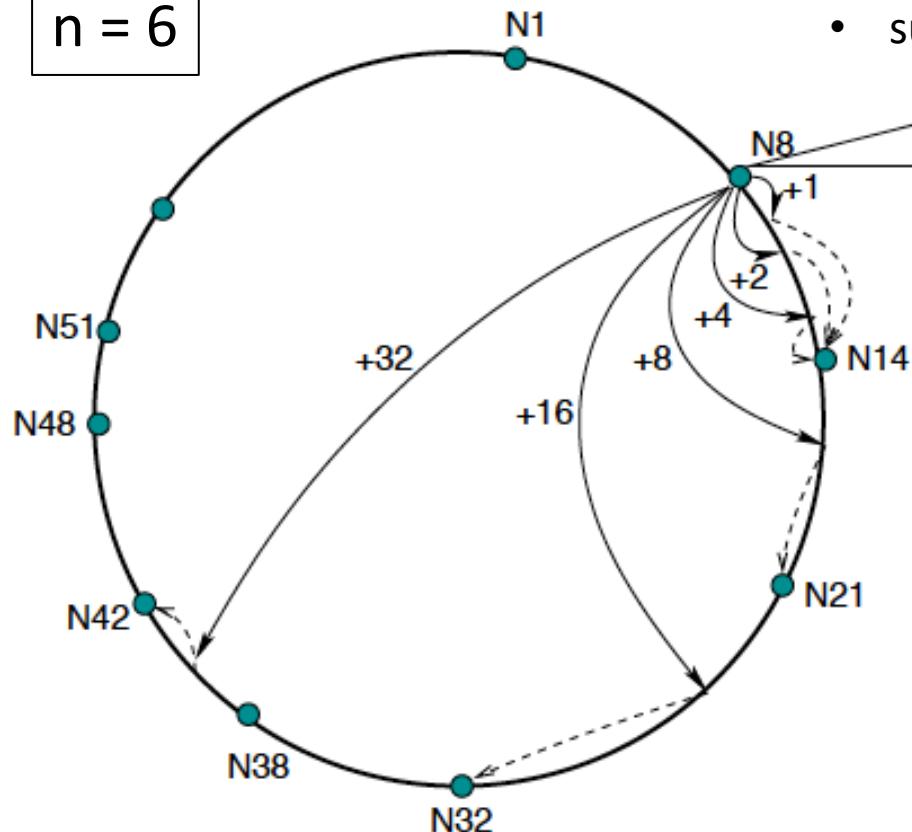


Consistent hashing: routing (under option 1.b)

- Each node records $\log(n)$ carefully chosen other nodes
- Ensures $O(\log(N))$ messages for routing queries, where N is the number of nodes
- Convenient trade-off for highly dynamic networks
- Based on the CHORD algorithm

CHORD algorithm

$n = 6$



i-th friend of a node p:

- node that follows p by at least 2^{i-1} , $i = 1, \dots, n$
- the first node with hash value equal or following value $h(p) + 2^{i-1}$, clockwise responsible for key $h(p) + 2^{i-1}$
- $\text{successor}(h(p) + 2^{i-1}) \rightarrow p.\text{finger}(i)$

Finger table

| | |
|-----------|-------|
| $N8 + 1$ | $N14$ |
| $N8 + 2$ | $N14$ |
| $N8 + 4$ | $N14$ |
| $N8 + 8$ | $N21$ |
| $N8 + 16$ | $N32$ |
| $N8 + 32$ | $N42$ |

$N8 + 2^0$
 $N8 + 2^1$
 $N8 + 2^2$
 $N8 + 2^3$
 $N8 + 2^4$
 $N8 + 2^5$

Memorize locations of other (friend) nodes (at most n in a ring with 2^n nodes) in a **finger table** associated with each node in the ring

- Low number of friends, low space for storing the finger table
- Each node knows more about nodes closely following it on the circle than about nodes farther away

CHORD algorithm: routing

- A node p cannot (in general) find directly the node p' responsible for a key k ($\text{successor}(k)$, following k on the ring) but p can find a friend which holds a more accurate information about k

CHORD algorithm: routing

- If the starting node S is responsible for the searched key k , it replies to the client
- If k falls between S and its successor, node S returns its successor
- Otherwise, S searches its finger table for the node S' whose ID most immediately precedes k , and then apply again the procedure at node S
- The reason behind this choice of S' is that the closer S' is to k , the more it will know about the identifier circle in the region of k
- It can be proved that with high probability the search converges in $O(\log(N))$ hops

CHORD algorithm: routing

```
// ask node n to find the successor of id
n.find_successor(id)
  if ( $id \in (n, \text{successor}]$ )
    return successor;
  else
     $n' = \text{closest\_preceding\_node}(id);$ 
    return  $n'.\text{find successor}(id);$ 

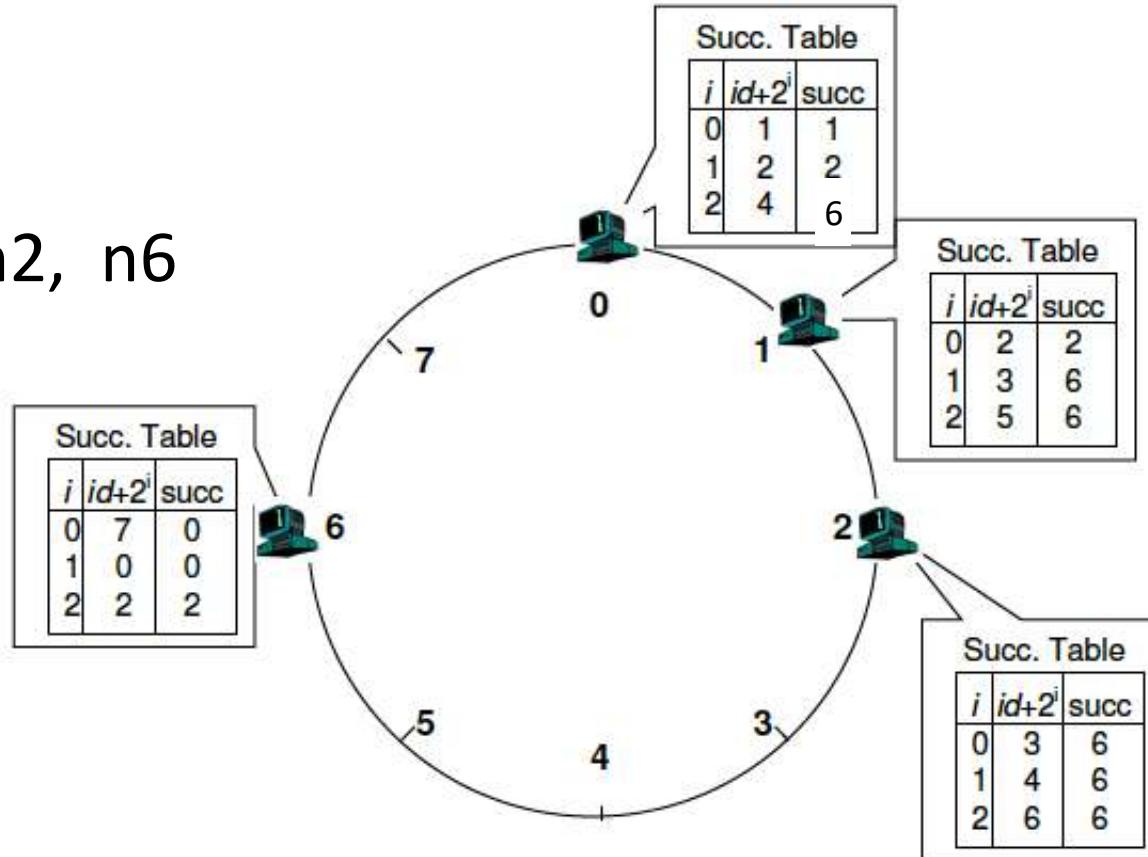
// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for  $i = n$  downto 1
    if ( $\text{finger}[i] \in (n, id)$ )
      return  $\text{finger}[i];$ 
  return  $n;$ 
```

Example

Nodes:

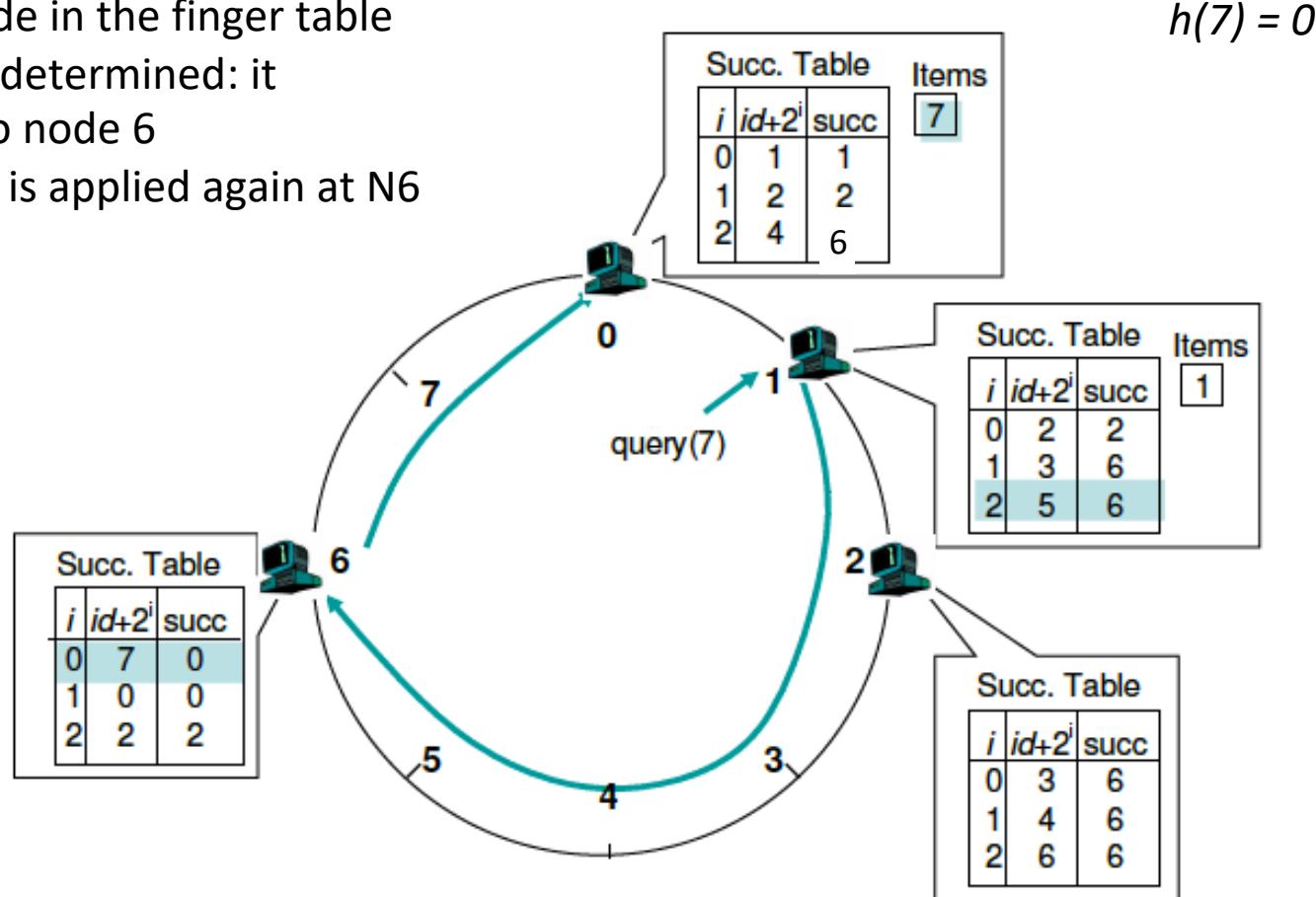
n_0, n_1, n_2, n_6

$$H(n_i) = i$$



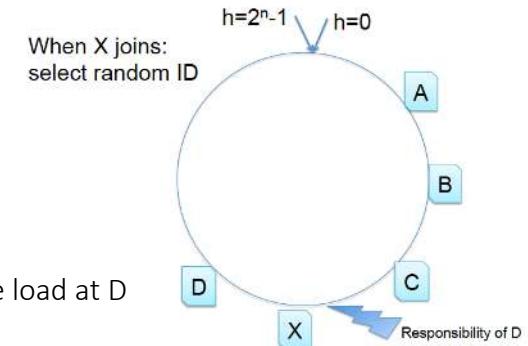
1. N1 does not store 7: $h(7) = 0$ and N1 can only stores values hash values in $(0,1]$ (from the figure, we notice that N1 only stores 1)
2. the closest node in the finger table preceding 0 is determined: it corresponds to node 6
3. the procedure is applied again at N6

Example



Request: a client send to node 1 the requests for reading/writing aggregates with key value equal to 7

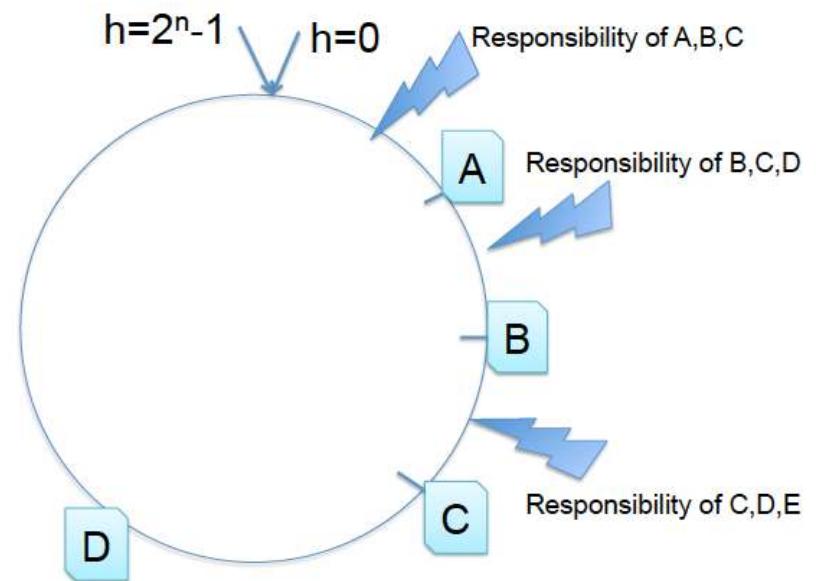
CHORD algorithm: scalability



- When a node p wants to join, it uses a contact node p' which carries out three tasks
 1. p must initialize its own routing table
 $\Rightarrow p'$ uses its routing table to locate p friends
 2. the routing table of the existing nodes must be updated to reflect the addition of p
 \Rightarrow more tricky
 3. finally p takes from its successor all the items k such that $h(k) \leq h(p)$
- Step 2 is performed by using a “stabilization” protocol that each node runs periodically in the background and which updates Chord’s finger tables and successor pointers
- No details (see additional references on AulaWeb)

Consistent hashing: replication

- Often leader-less, quorums
- Let m =degree of replication
- Assign key k to m nodes:
*successor($h(k)$),
successor(successor($h(k)$))
successor(...successor($h(k)$) ...)
 $m - 1$ successors forward*

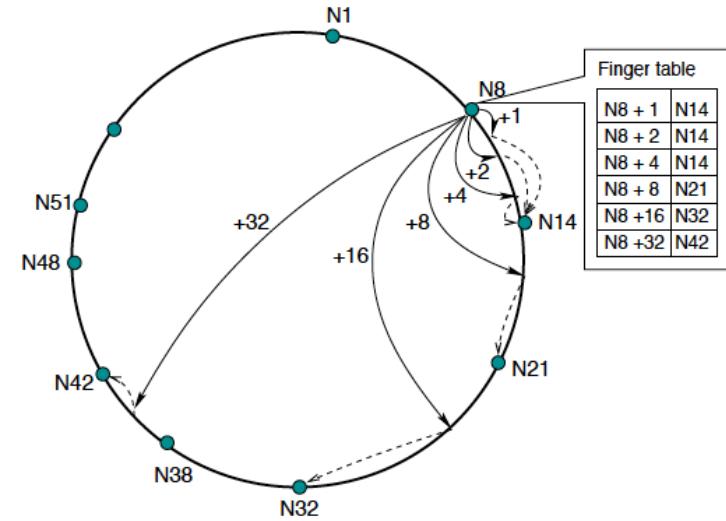


Consistent hashing: fault tolerance

- Through replication for being able to access data after a fault
- Specific approaches for taking care of the (possibly shared) knowledge about the hash directory (i.e., the partitioning)
 - If the hash directory is stored in one Master node, the node becomes a single point of failure and such data must be replicated as well
 - If the hash directory is shared and locally stored on each node (as in Chord), other approaches are possible

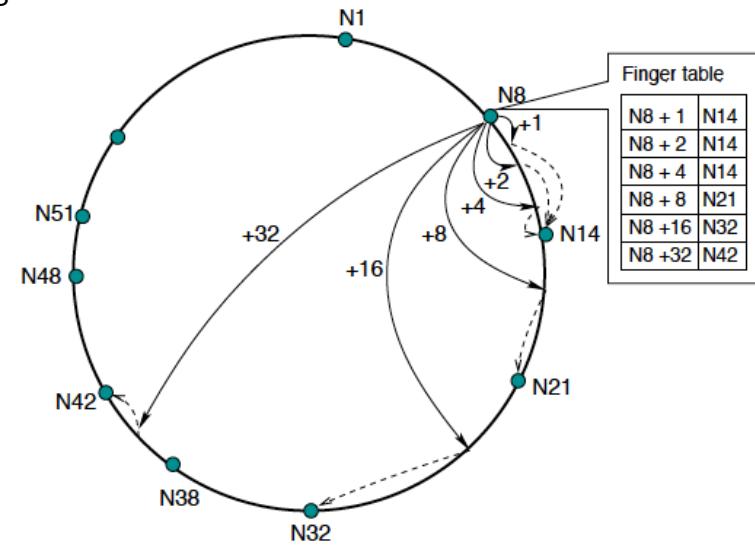
Consistent hashing: fault tolerance (in CHORD)

- The correctness of the Chord protocol relies on the fact that each node knows its successor
- When considering fault tolerance, the successor of a node p can be considered as the first live node following p on the ring
- If nodes 14, 21, and 32 fail simultaneously, the successor of node 8 is node 38 but, based on the finger table, the successor is set to node 42 (no finger points to 38)



Consistent hashing: fault tolerance (in CHORD)

- Incorrect successor will lead to incorrect lookups
 - Consider a query for key 30 initiated by node 8
 - Node 8 will return node 42 (the first node it knows about from its finger table), instead of the correct successor, node 38
- To increase robustness, each node maintains a successor list of size r , containing the node's first r successors
- If a node's immediate successor does not respond, the node can substitute the second entry in its successor list
- All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r
- In the previous example, setting $r = 4$, solve the problem



Consistency and transactions

- Traditional database systems satisfy ACID properties
- ACID properties favour Consistency wrt Availability and performance
- In distributed data systems: consistency means replica consistency
- But scalability is a problem
- Give up Consistency and Isolation in exchange for high availability and high performance
- No strong consistency but eventual consistency
- Relaxed properties: from ACID to BASE

Consistency and transactions: from ACID to BASE

- **Atomicity**
 - The transaction corresponds to a single indivisible action
- **Consistency**
 - The transaction cannot leave the database in an inconsistent state
- **Isolation**
 - Transactions cannot interfere with each other and see intermediate results
- **Durability**
 - Once a transaction commits, the results are made persistent

Consistency and transactions: from ACID to BASE

- Basically Available
 - The database appears to work most of the time
- Soft-state
 - Stores do not have to be write-consistent, nor do different replicas have to be mutually consistent all the time
- Eventual Consistency
 - Stores exhibit consistency at some point in the figure

Soft state and durability

- Durability in ACID transactions:
 - When Write is committed, the change is permanent
 - Strict durability is costly
 - In some cases, strict durability is not essential and it can be traded for scalability (write performance)
- A simple way to **relax durability**:
 - **Store data in memory and flush to disk regularly**
 - delay write on disks
 - if the system shuts down, we loose updates in memory (unless we use logging mechanisms)

Visual Guide to NoSQL Systems



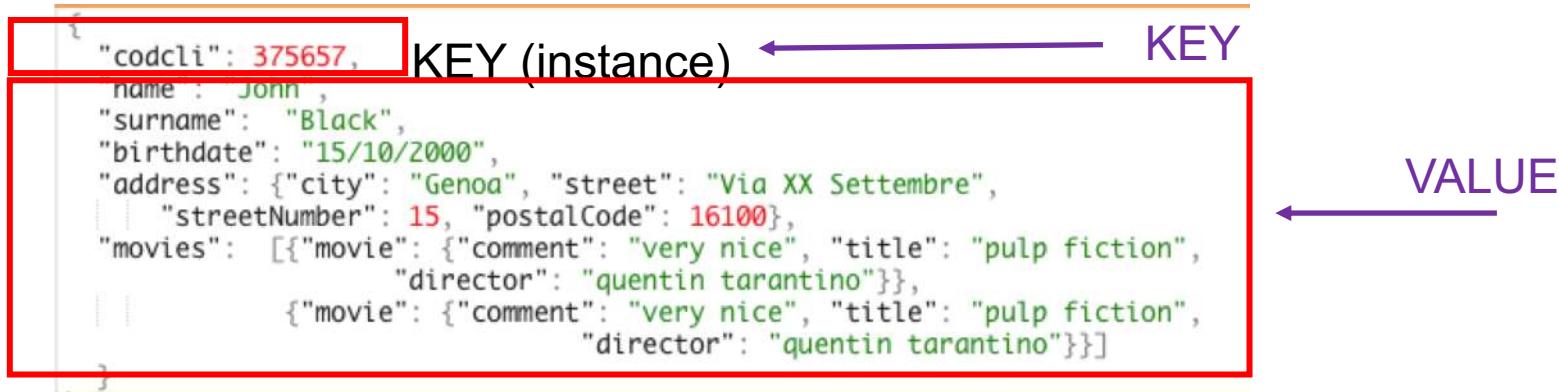
Document-oriented NoSQL data stores

Document-oriented data model at a glance

- Each data instance is represented in the form **(key, value)**
 - key is an identifier
 - value is the aggregate, corresponding to **a document** that, in turns, can be interpreted as a set of **<name, nested-document>** pairs
- Structure of the aggregate is **visible at the logical and application level**
 - All nested document components can be accessed during read and write operations
- Also in this case, data instances can be grouped into logical collections

JSON: **instance level**

Logical & Application level



Documents

- Can be represented using a **semi-structured** language like XML, JSON, BSON, and so on
- Are self-describing, **hierarchical tree data structures**
- Can consist of scalar values, collections, and maps
- Are **structurally similar** but **not identical** to each other
- **Semi-structured** data model

Example - document

```
{  
  "codcli": 375657,  
  "name": "John",  
  "surname": "Black",  
  "birthdate": "15/10/2000",  
  "address": {"city": "Genoa", "street": "Via XX Settembre",  
    "streetNumber": 15, "postalCode": 16100},  
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
    "director": "quentin tarantino"}},  
    {"movie": {"comment": "very nice", "title": "pulp fiction",  
    "director": "quentin tarantino"}]}  
}
```

```
▼ object {6}  
  codcli : 375657  
  name : John  
  surname : █ Black  
  birthdate : 15/10/2001  
▼ address {4}  
  city : Genoa  
  street : Via XX Settembre  
  streetNumber : 15  
  postalCode : 16100  
▼ movies [2]  
  ▼ 0 {1}  
    ▼ movie {3}  
      comment : very nice  
      title : pulp fiction  
      director : quentin tarantino  
  ▼ 1 {1}  
    ▼ movie {3}  
      comment : very nice  
      title : pulp fiction  
      director : quentin tarantino
```

Document structure

- No predefined schema, no DDL
- The schema of the data can differ across documents
- In documents, there are no empty attributes
 - if a given attribute is not found, it was not set or not relevant to the document
- New attributes can be created without the need to define them or to change the existing documents

Example – similar documents

```
{ "codcli": 375657,
  "name": "John",
  "surname": "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
    "streetNumber": 15, "postalCode": 16100},
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",
      "director": "quentin tarantino"}},
    {"movie": {"comment": "very nice", "title": "pulp fiction",
      "director": "quentin tarantino"}]}
}

▼ object {6}
  codcli : 123456
  name : Jane
  surname : □ White
  phonenbr : 3406701353
  email : jane.white@gmail.com
  ▼ movies [1]
    ▼ 0 {1}
      ▼ movie {3}
        comment : best movie ever
        title : pulp fiction
        director : quentin tarantino

  ▼ object {6}
    codcli : 375657
    name : John
    surname : □ Black
    birthdate : 15/10/2001
    ▼ address {4}
      city : Genoa
      street : Via XX Settembre
      streetNumber : 15
      postalCode : 16100
    ▼ movies [2]
      ▼ 0 {1}
        ▼ movie {3}
          comment : very nice
          title : pulp fiction
          director : quentin tarantino
      ▼ 1 {1}
        ▼ movie {3}
          comment : very nice
          title : pulp fiction
          director : quentin tarantino

  ▼ object {6}
    codcli : 123456
    name : Jane
    surname : "White"
    phonenbr : "3406701353"
    email : "jane.white@gmail.com"
    "movies": [
      {
        "movie": {
          "comment": "best movie ever",
          "title": "pulp fiction",
          "director": "quentin tarantino"
        }
      }
    ]
}
```

Collections

- Pairs can be grouped into logical **collections**: sets of aggregates, i.e., sets of (key, value) pairs
- A collection is a grouping of documents

Collections and schema information

- (nested) names in pairs representing the document
- collections do not enforce specific data structure
 - documents within a collection can have different fields (**flexibility**)
 - in practice, all documents in a collection are similar and have related purpose

Actually, in mongoDB we may require data validation against a JSON schema

Example - collection

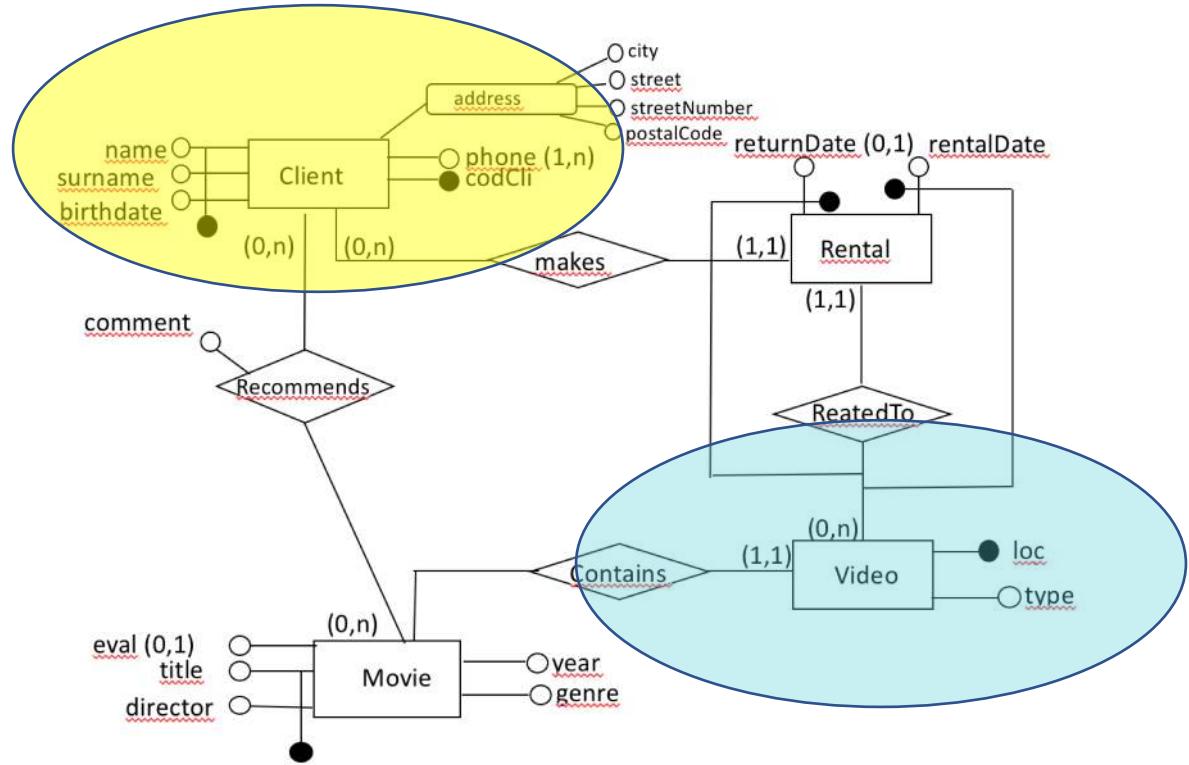
```
{  
    "codcli": 375657,  
    "name": "John",  
    "surname": "Black",  
    "birthdate": "15/10/2000",  
    "address": {"city": "Genoa", "street": "Via XX Settembre",  
    .... "streetNumber": 15, "postalCode": 16100},  
    "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
        .... "director": "quentin tarantino"}},  
    .... {"movie": {"comment": "very nice", "title": "pulp fiction",  
        .... "director": "quentin tarantino"}}]  
}  
  
{  
    "codcli": 123456,  
    "name": "Jane",  
    "surname": "White",  
    "phonenbr": "3406701353",  
    "email": "jane.white@gmail.com",  
    "movies": [  
        ....  
        {"  
            "movie": {  
                .... "comment": "best movie ever",  
                .... "title": "pulp fiction",  
                .... "director": "quentin tarantino"  
            }  
        }  
    ]  
}
```

Keys

- (**key**, value) pair
- At the logical level: identification + data retrieval
 - the key **needs** to assume **unique values** in the collection (i.e., it is an **identifier**)
 - mandatory
- At the physical level, the key could tell the system **how to partition the data** and where to store the data
 - **partition key**
- Partition key and identifier could to be different
- Aggregates can be directly retrieved **by specifying values for attributes in the key or for nested attributes**
- **The structure of the aggregate is exposed**
 - All nested sub-documents can be accessed

Keys

- Usually, one single attribute
- The type and the content identified starting from the designed aggregates



Video: {loc,...}

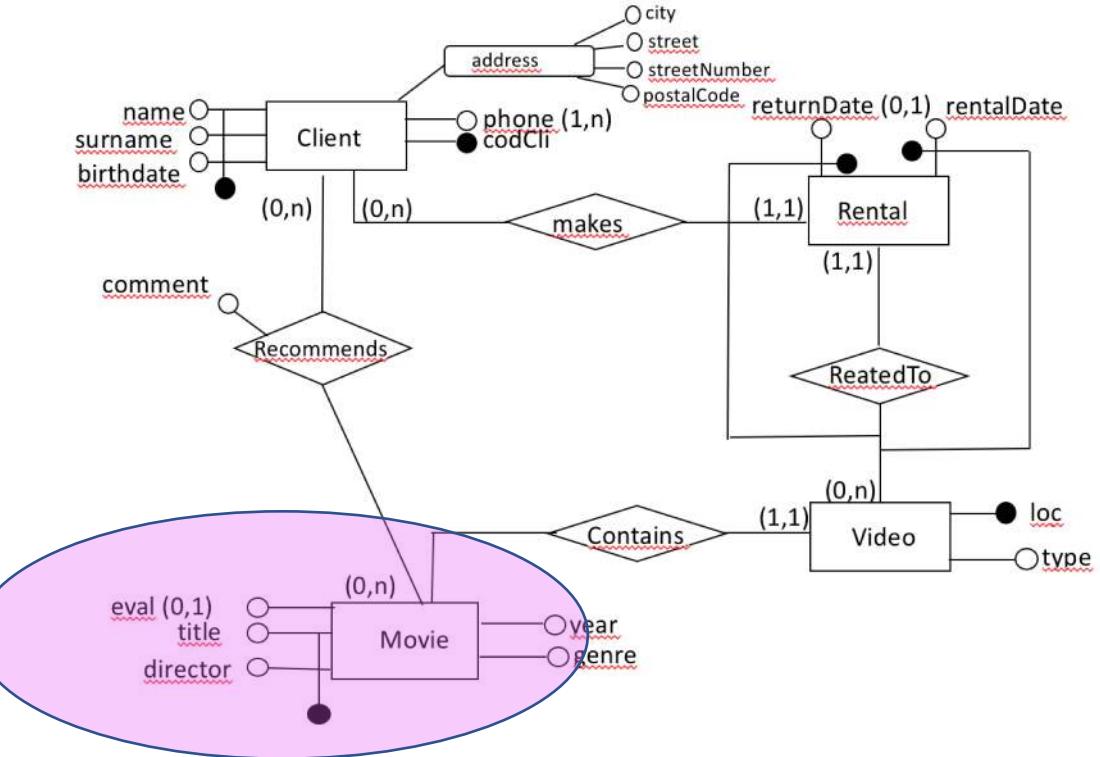
key is loc

Client: {codCli, ...}

key is codCli

_id in MongoDB

Keys



Movie : {title, director,...}

key could be (title, director)

or we can use a new attribute **movId** as the key (in this case values will be provided by the system)

Identifier vs partition/sharding key

- Ids mainly refer the way data can be [referenced](#)
 - In the Videos collection, videos are identified by loc
 - In the Clients collection, clients are identified by codCli
 - In the Movies collection, clients are identified by title&director (or movieId)
- In document-oriented data stores, the choice of an identifier does not impact how data can be retrieved
- Partition keys impact the way data are stored

Video Rental Example

Logical & application level

```
client: { codCli,
```

```
    name, surname, birthdate,  
    address: {city, street, streetNumber, postalCode},  
    movies: [ {movie: {comment, id:{title, director},...}} ]
```

```
]
```

```
}
```

REFERENCE

```
movie: { id: {title,director},
```

```
    year, genre, recommended_by: [ {name, surname, comment} ],  
    contained_in: [ {video: {loc, type}} ] }
```

```
video: {loc,
```

```
    type, rentals: [ {rental: {rentalDate, codCli}} ], title, director}}
```



REFERENCE



Query features

- Document data stores provide different query features
- In general:
 - you can query the data inside the document without having to retrieve the whole document by its key and then introspect the document

Interaction

- Basic **lookup** based on the **key** or **field names** in the aggregates
 - Document get(key)
 - Document get([name:value])
 - Void put(key, document)
 - Void set(key, [name:value])
 - Void remove(key)
- We can query inside (nested) documents, at any level
- Additional search constraints depending on the specific document representation (e.g, range constraints on values)
- We can retrieve part of the aggregate rather than the whole one (projection operations)
- In case collections are supported, the collection name has to be specified in each operation

Example 1 – Find the title of the movie in a certain video

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

get(Video, 1234, [title])

Retrieve the video by the key (location number)
and project on the title

Example 2 – Find the videos containing a certain movie

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

get(Video, [title: 'pulp fiction'])

Retrieve the video by some filtering conditions on (nested) attributes
and we return the whole videos

Example 3 – Find the videos rented by a certain client

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

get(Video, '375657', [rentals.rental.codCli])

Retrieve the video by the key
and we return some nested attributes (dot notation)

Example 4 – Find the videos containing a certain movie - with the movies collection

```
movie: { id: {title, director}, year, genre,  
        recommended_by: [ {name, surname, comment} ],  
        contained_in: [ {video: {loc, type}} ] }
```

- `get(Movie, {'pulp fiction', 'quentin tarantino'}, [contained_in.video])`
- Retrieve all the information about the videos containing a certain movie

REMARK: the relationship is part of the aggregate, a single data access to retrieve all the relevant information (contained in a single data node)

Example 5 – Find the videos containing a certain movie - with the movies collection, Movieid

```
movie: { Movieid,  
         title, director, year, genre,  
         recommended_by: [ {name, surname, comment} ],  
         contained_in: [ {video: {loc, type}} ] }
```

- `get(Movies,[title: 'pulp fiction', director: 'quentin tarantino'])`
- At the data store level, retrieve the information about the videos containing a certain movie

REMARK: the relationship is part of the aggregate, a single data access to retrieve all the relevant information (contained in a single data node)

Example 4 - Relationships

```
{"loc": 1234,
```

```
  "type": "dvd",
  "rentals": [{"rental": {"rentalDate": "15/10/2021",
    "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

Find the age(s) of customer(s) that rented a given video

```
{
  "codcli": 375657,
  "name": "John",
  "surname": "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
    "streetNumber": 15, "postalCode": 16100},
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}},
    {"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}]}
}
```

get(Videos, 1234)

- At the data store level, find Video 1234 in collection Videos, no detailed information the clients that rented it
- at the *application level*, we go inside the aggregate value and we discover that it was rented by Client 375657
- we can execute get(Clients, 375657) to retrieve information about a/the customer that rented video 1234 (thus navigating the customer reference stored inside the video)

Advanced interaction

- Some document oriented data stores support more expressive interaction
- Specifically, management of explicit references among documents (for relationships) and a sort of **join** operation traversing them
- MongoDB, e.g., supports the aggregation framework and the `$lookup` construct that allows to navigate references (join)
- Embed vs reference
- More expressive query languages (e.g., group by)

Example 4 – Relationships – In MongoDB

```
{"loc": 1234,
```

```
  "type": "dvd",
  "rentals": [{"rental": {"rentalDate": "15/10/2021",
    "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

Find the age(s) of customer(s) that rented a given video

D REMARKS:
via

1. System support for join, but we need to explicitly tell the system what to look for and what to match
2. Since the relationship is not part of the aggregate, navigating it requires two distinct data accesses, at two possibly distinct data nodes (queries on embedded relationships are more efficient)

... and we can join clients that rented a video

```
{ "codcli": "375657", "name": "John Doe", "age": 30 }
```

Popular document data stores



Each product has some features that may not be found in others

Popular document data stores

- MongoDB [<http://www.mongodb.org/>],
- CouchDB [<http://couchdb.apache.org/>],
- Terrastore [<https://code.google.com/p/terrastore/>],
- OrientDB [<http://www.orientechnologies.com/>],
- RavenDB [<http://ravendb.net/>],

MongoDB – A document data store



MongoDB in short

| Feature | MongoDB |
|----------------------------------|--|
| Model | Document-based |
| Query language | Supported, aggregation framework |
| Reference scenarios | Transactional (read/write intensive) & analytical (read intensive) |
| Partitioning (Sharding) | Hash-based & range-based (not all the collections are sharded) |
| Indexes | Primary, secondary, multiattr, fulltext |
| Replication | Master-slave, replica set |
| Consistency | Strong, eventual at replicas |
| Availability | Can be mediated with consistency, through r/w concerns |
| Fault tolerance | By replica set, system remains operational on failing nodes |
| Transactions | ACID transactions (multidocument since 4.0) (read concern) |
| CAP theorem | CP |
| Distributed by | MongoDB Inc. |

Mongo DB



- History:
 - Development started by 10gen in 2007
 - Open sourced in 2009
 - In 2013, 10gen became MongoDB Inc.
- Uses BSON format
 - Based on JSON –B stands for Binary
- Written in C++, C, Javascript
- Supports APIs (drivers) in many languages
 - JavaScript, Python, Ruby, Perl, Java, JavaScala, C#, C++, Haskell, Erlang, ...

Who uses mongoDB?



Uber



Lyft



LaunchDarkl
y



Stack



Delivery Hero



ViaVarejo



Bepro
Company



Accenture



CircleCI

MongoDB

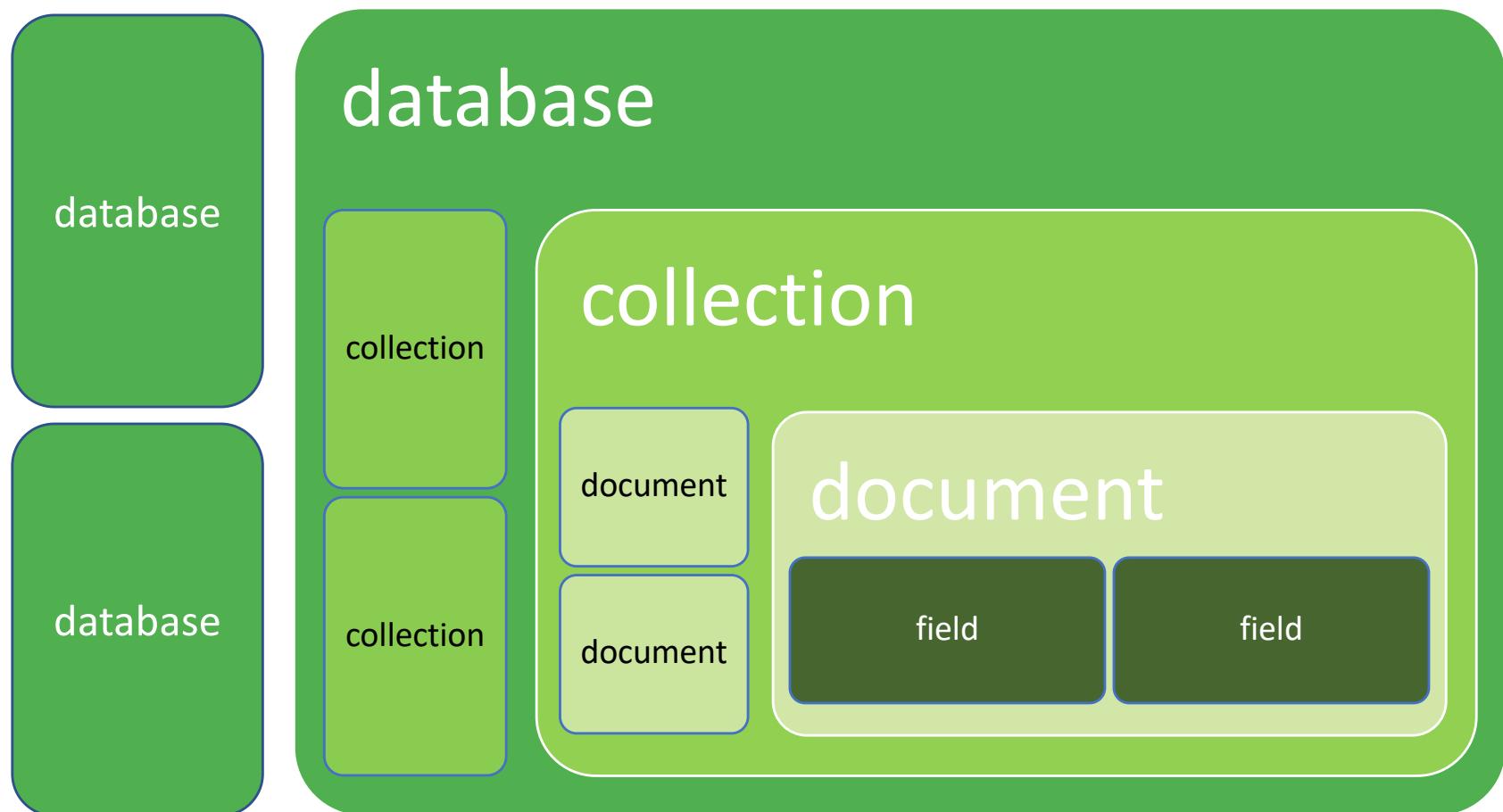
- MongoDB **Compass** (GUI to explore and manipulate documents)
- MongoDB **Atlas** (on cloud)
- Supported in Azure Cosmos DB



Data Model and Interaction

MongoDB: Basics

- Mongodb instance



MongoDB: Basics

| Oracle | MongoDB |
|-------------------|------------------------------|
| database instance | MongoDB instance |
| schema | database |
| table | collection |
| row | document |
| rowid | _id |
| join | embedded documents, \$lookup |

MongoDB: Basics

- Each MongoDB instance has multiple databases
 - Similar to a database schema in a RDBMS
- Each database can have multiple collections
 - Similar to a table in a RDBMS
- When we store a document, we have to choose which database and collection this document belongs

`db.collection.insertOne(document)`

- Document identifier **`_id`** will be created for each document, field name reserved by system

Schema validation

A collection may be associated with a schema in JSON schema

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name", "year", "major", "address" ],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        year: {
          bsonType: "int",
          minimum: 2017,
          maximum: 3017,
          description: "must be an integer in [ 2017, 3017 ] and is required"
        },
        major: {
          enum: [ "Math", "English", "Computer Science", "History", null ],
          description: "can only be one of the enum values and is required"
        },
        gpa: {
          bsonType: [ "double" ],
          description: "must be a double and is required"
        }
      }
    }
  }
})
```

MongoDB query language

- `db.collection.find(<query>, <projection>)`
- Provides functionality similar to the SELECT command
 - `<query>` where condition
 - `<projection>` fields in result set

`<field>: <1 or true>`

Specifies the inclusion of a field.

`<field>: <0 or false>`

Specifies the exclusion of a field.

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find()
```

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find({ "loc": 1234 }, { title: 1 })
```

where
(filtering -
selection)

Select
(projection)

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find( {}, {title:1} )
```

Select
(projection)

Projection only, `_id` is included in the result
if not explicitly excluded (i.e., `{_id: 0}`)

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```



```
db.videos.find({ "title": "pulp fiction" })
```

where
(filtering -
selection)

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find({ "rentals.rental.codCli": "375657" })
```

path expression (dot notation)

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                    "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find({ "type": { $in, ["dvd", "vhs"] } })
```

MongoDB query language

```
// in movies
```

```
{"id": 12345678,  
 "title": "pulp fiction",  
 "director": "quentin tarantino",  
 "year": 1994,  
 "genre": ["drama", "crime"],  
 "recommended_by": [],  
 "contained_in": [{"video": {"loc": 1234,  
 "type": "dvd"}  
 }]  
 }
```

```
db.movies.find({ "year" : { $geq: 1990 } })
```

MongoDB – Query Operators

| Name | Description |
|-------------|--|
| \$eq | Matches value that are equal to a specified value |
| \$gt, \$gte | Matches values that are greater than (or equal to) a specified value |
| \$lt, \$lte | Matches values less than or (equal to) a specified value |
| \$ne | Matches values that are not equal to a specified value |
| \$in | Matches any of the values specified in an array |
| \$nin | Matches none of the values specified in an array |
| \$or | Joins query clauses with a logical OR returns all |
| \$and | Join query clauses with a logical AND |
| \$not | Inverts the effect of a query expression |
| \$nor | Join query clauses with a logical NOR |
| \$exists | Matches documents that have a specified field |

<https://docs.mongodb.org/manual/reference/operator/query/>

MongoDB query language

```
// in videos
{"loc": 1234,
 "type": "dvd",
 "rentals": [{"rental": {"rentalDate": "15/10/2021",
                      "codCli": 375657}}],
 "title": "pulp fiction",
 "director": "quentin tarantino"}
```

```
db.videos.find({"title": "pulp fiction",
 "director" : "gabriele salvatores" })
```

```
db.videos.find({$or: [{"title": "pulp fiction",
 "director" : "gabriele salvatores" }] })
```

MongoDB query language

- `db.collection.find(<query>, <projection>)`
- Provides functionality similar to the SELECT command
 - `<query>` where condition
 - `<projection>` fields in result set

`<field>: <1 or true>`

Specifies the inclusion of a field.

`<field>: <0 or false>`

Specifies the exclusion of a field.

- Return a `cursor` to handle a result set
- Can modify the query to impose limits, skips, and sort orders
- Can specify to return the ‘top’ number of records from the result set

- `db.collection.findOne(<query>, <projection>)`

Sort

- `db.movies.find({"year" : {$geq: 1990}}).sort ({"title": 1})`
- `db.movies.find({"director" : "quentin tarantino"}).sort ({"year": -1})`

Count

- `count()` or `find().count()`
- Can have the same arguments as `find`
- `db.movies.count({ "year" : { $geq: 1990 } })`

Lookup

```
{  
  $lookup:  
    {  
      from: <collection to join>,  
      localField: <field from the input documents>,  
      foreignField: <field from the documents of the "from" collection>,  
      as: <output array field>  
    }  
}  
  
      SELECT *, <output array field>  
      FROM collection  
      WHERE <output array field> IN (  
        SELECT *  
        FROM <collection to join>  
        WHERE <foreignField> = <collection.localField>  
      );
```

Lookup

```
db.orders.insert([
  { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
  { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
  { "_id" : 3 }
])

db.inventory.insert([
  { "_id" : 1, "sku" : "almonds", "description": "product 1", "instock" : 120 },
  { "_id" : 2, "sku" : "bread", "description": "product 2", "instock" : 80 },
  { "_id" : 3, "sku" : "cashews", "description": "product 3", "instock" : 60 },
  { "_id" : 4, "sku" : "pecans", "description": "product 4", "instock" : 70 },
  { "_id" : 5, "sku": null, "description": "Incomplete" },
  { "_id" : 6 }
]

db.orders.aggregate([
  {
    $lookup:
      {
        from: "inventory",
        localField: "item",
        foreignField: "sku",
        as: "inventory_docs"
      }
  }
])
```

Lookup

```
db.orders.insert([
  { "_id" : 1, "item" : "almonds", "price" : 12 },
  { "_id" : 2, "item" : "pecans", "price" : 20 },
  { "_id" : 3, "item" : "cashews", "price" : 15 }
])

db.inventory.insert([
  { "_id" : 1, "sku" : "almonds", "description" : "product 1", "instock" : 120 },
  { "_id" : 2, "sku" : "bread", "description" : "product 2", "instock" : 50 },
  { "_id" : 3, "sku" : "cashews", "description" : "product 3", "instock" : 80 },
  { "_id" : 4, "sku" : "pecans", "description" : "product 4", "instock" : 70 },
  { "_id" : 5, "sku": null, "description": "placeholder" },
  { "_id" : 6 }
])

db.orders.aggregate([
  {
    $lookup:
      {
        from: "inventory",
        localField: "item",
        foreignField: "sku",
        as: "inventory_docs"
      }
  }
])
```

The screenshot shows a MongoDB shell command being run. The command uses the `$lookup` pipeline stage to join the `orders` collection with the `inventory` collection based on the `item` field in `orders` and the `sku` field in `inventory`. The joined documents are stored in the `inventory_docs` array of each `orders` document.

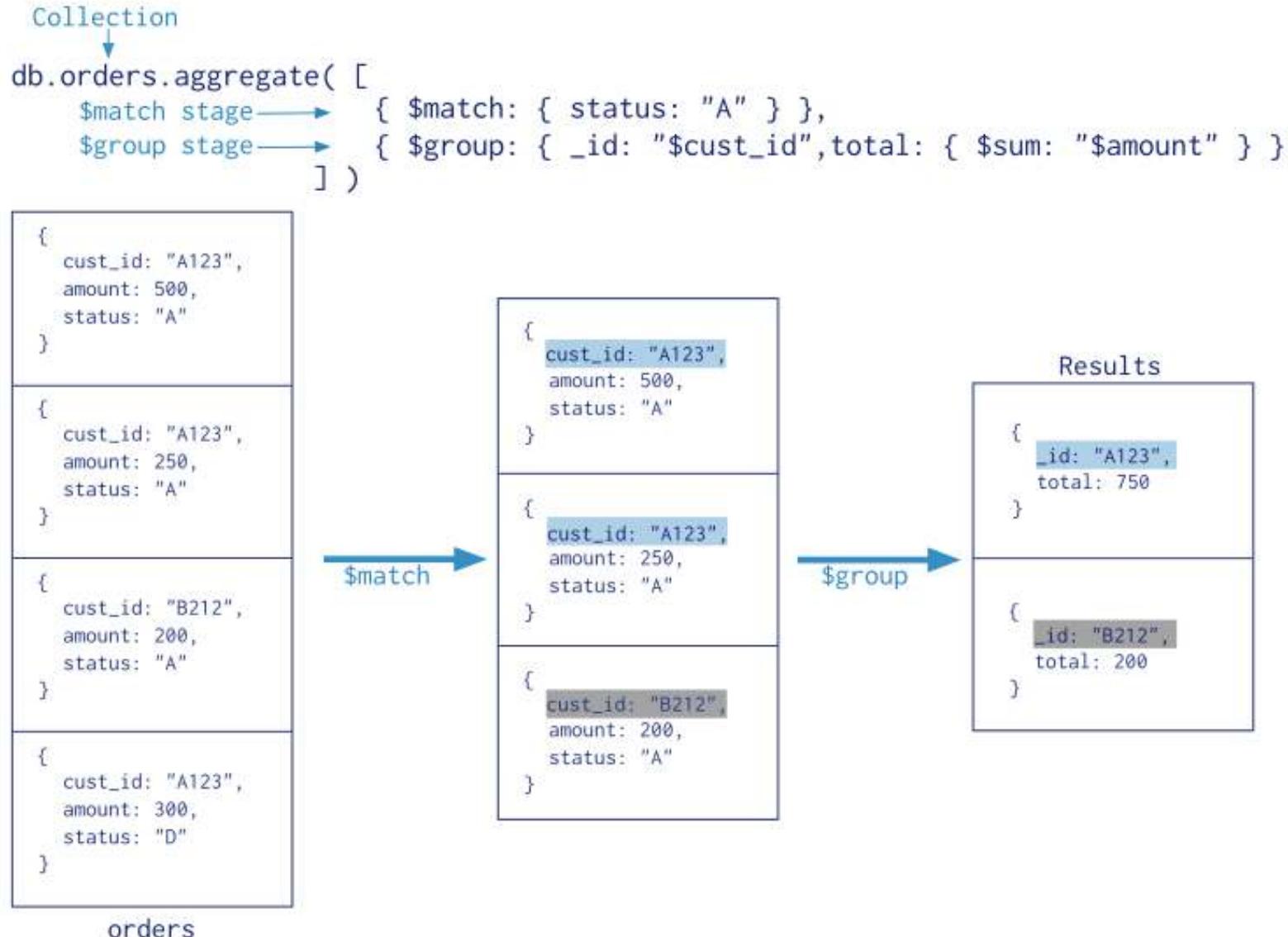
On the right side of the image, there is a visual representation of the resulting SQL query. It consists of two parts:

```
SELECT *, inventory_docs
FROM orders
WHERE inventory_docs IN (
  SELECT *
  FROM inventory
  WHERE sku = orders.item
);
```

Aggregates

- Aggregation framework provides SQL-like aggregation functionality
- Documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through
- Expressions produce output documents based on calculations performed on input documents
- Example
 - `db.orders.aggregate({$group : {_id: type,
totalquantity: { $sum: quantity} } })`

Aggregates



Map Reduce

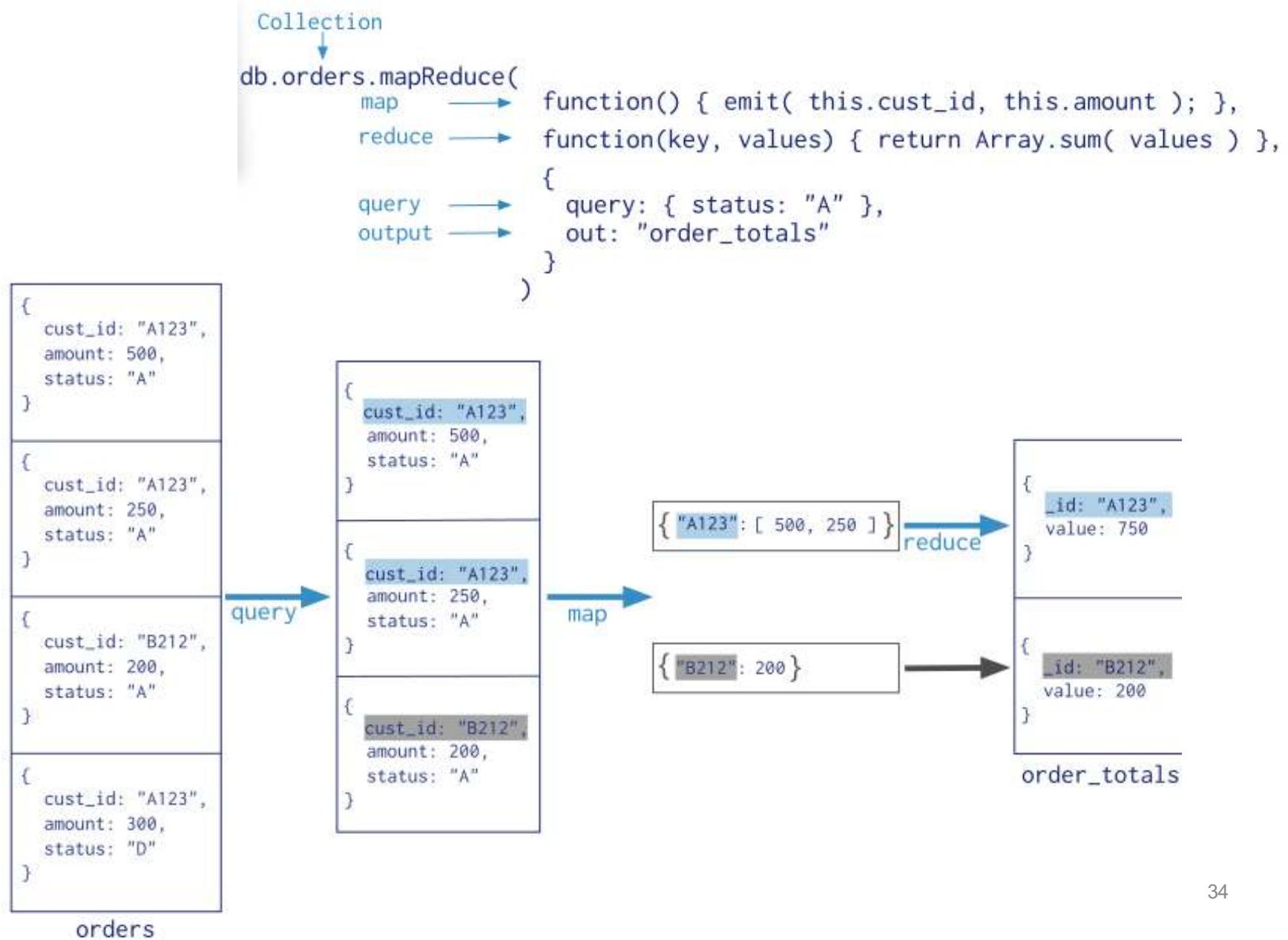
REMARK: partition* is called shard* in mongodb terminology

- Performs complex aggregator functions given a collection of keys, value pairs
- Must provide at least a map function, reduction function and a name of the result set

```
db.collection.mapReduce( <mapfunction>,
                         <reducefunction>,
                         { out: <collection>,
                           query: <document>,
                           sort: <document>,
                           limit: <number> } )
```

- If the input is a partitioned collection, mongos will automatically dispatch the map-reduce job to each partition in parallel
- if the out field for mapReduce has the partitioning value, the output collection is partitioned using the `_id` field as the partition key

Map Reduce



CRUD Operations

All write operations in MongoDB are **atomic** on the level of a single **document**.

- Create

```
db.collection.insertOne( <document> )  
db.collection.insertMany( [<document>,<document>, ...] )
```

- Update

```
db.collection.update( <query>, <update>, <options> )  
... updateOne and updateMany
```

- Delete

```
db.collection.deleteMany( <query> )  
db.collection.deleteOne( <query> )
```

- Read

```
db.collection.find( <query>, <projection> )
```

Indexing

MongoDB - Indexes

```
> db.clients.find(firstname: "alice") .explain()
{
  "cursor" : "BasicCursor",
  "nscanned" : 1000000,
  "nscannedObjects" : 1000000,
  "n" : 1,
  "millis" : 721,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {}
}
```

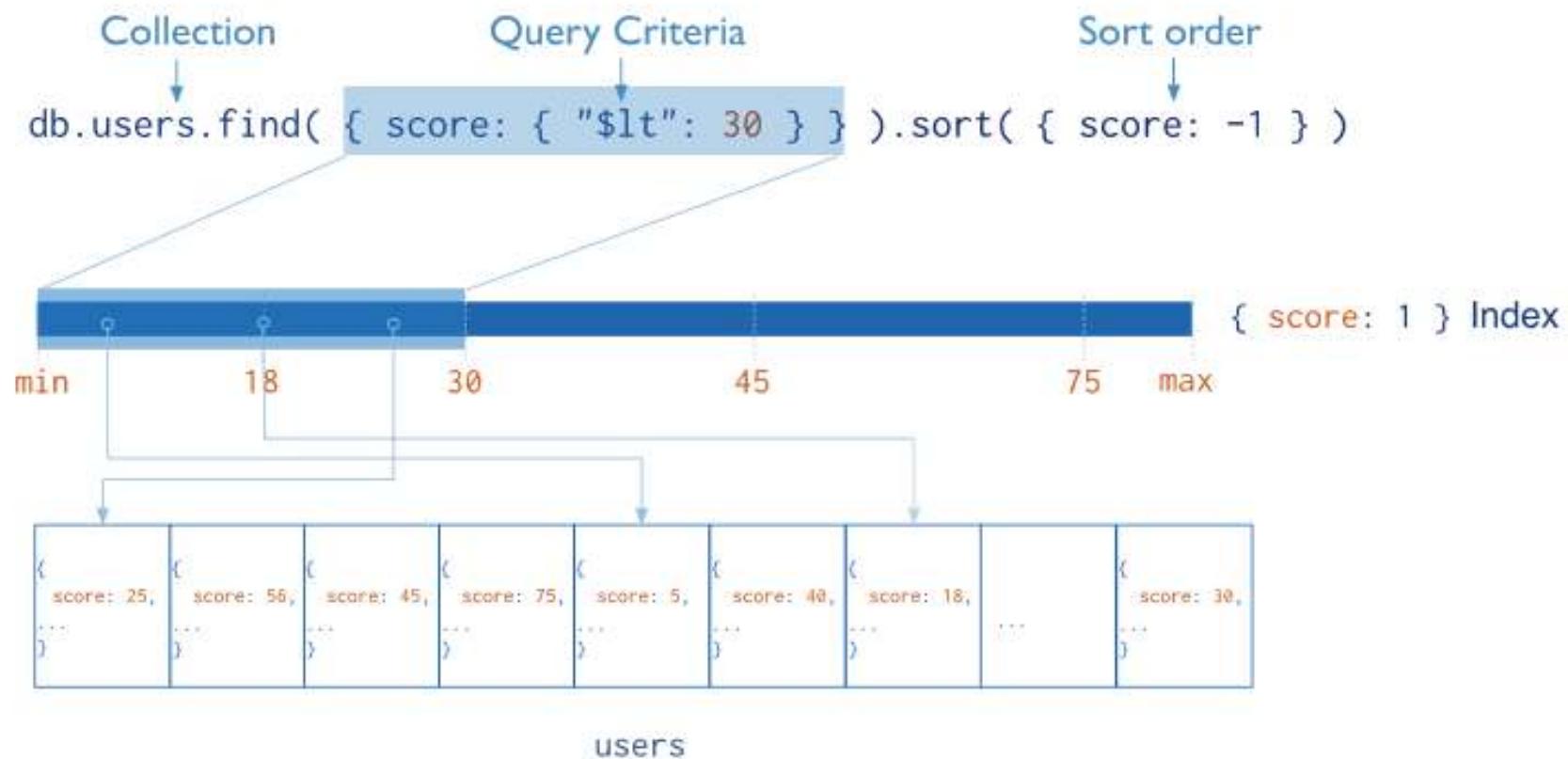
MongoDB - Indexes

```
>db.users.ensureIndex({"firstname" : 1})
```

```
> db.clients.find(firstname: "alice")).explain()
```

```
{
  "cursor" : "BtreeCursor username_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 3,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "isMultiKey" : false,
  "indexOnly" : false,
  "indexBounds" : {"firstname" : [ ["alice", "alice"] ] }
}
```

Index functionalities



Index functionalities

- An index is automatically created on the `_id` field (the primary key)
- Users can create other indexes
 - to improve query performance (filter conditions, sorting on the field) or
 - to enforce unique values for a particular field (`unique`)

```
db.users.ensureIndex( { "username" : 1 } , { "unique" : true } )
```

Index functionalities

- Supports single field index as well as compound index
`db.users.ensureIndex({ "age" : 1, "username" : 1 })`
 - Like SQL: order of the fields in a compound index matters
 - If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array
 - Also supports hash indexes
-
- <https://docs.mongodb.com/manual/indexes/>

Full-text Indexes and Search

- MongoDB has a special type of index for searching for text within documents
 - built-in support for multi-language stemming and stop words
- Heavyweight, be cautious
- `db.stores.createIndex({ name: "text", description: "text" })`
- Text search
 - `db.stores.find({ $text: { $search: "java coffee shop" } })`
- It is possible to exclude words, and to sort on relevance ranking `textScore`

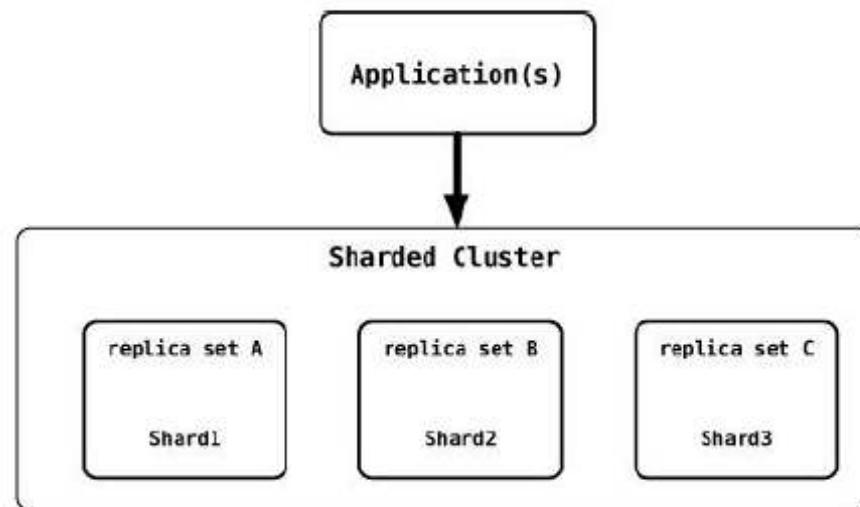
Architecture

MongoDB in short

| Feature | MongoDB |
|----------------------------------|--|
| Model | Document-based |
| Query language | Supported, aggregation framework |
| Reference scenarios | Transactional (read/write intensive) & analytical (read intensive) |
| Partitioning (Sharding) | Hash-based & range-based (not all the collections are sharded) |
| Indexes | Primary, secondary, multiattr, fulltext |
| Replication | Master-slave , replica set |
| Consistency | Strong, eventual at replicas |
| Availability | Can be mediated with consistency, through r/w concerns |
| Fault tolerance | By replica set, system remains operational on failing nodes |
| Transactions | ACID transactions (multidocument since 4.0) (read concern) |
| CAP theorem | CP |
| Distributed by | MongoDB Inc. |

Collection sharding/data partitioning

- By sharding/partitioning the data is split by certain field and moved to different nodes

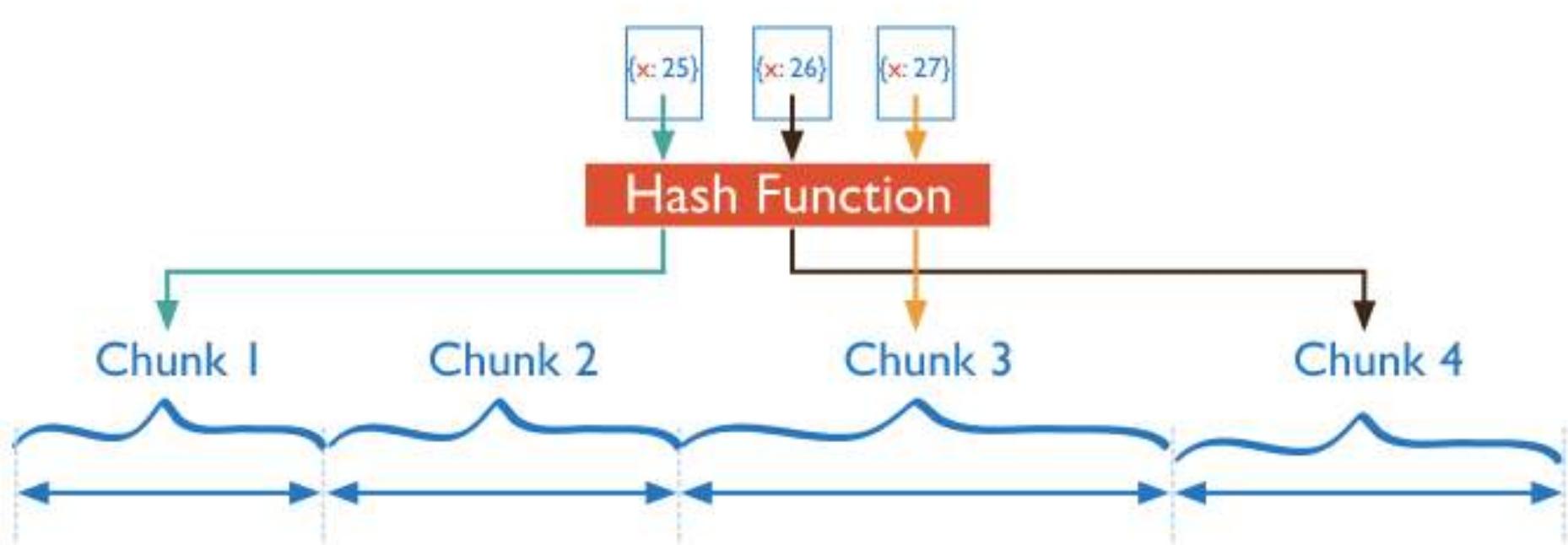


- Partitioning on the first name of the customer:

```
db.runCommand({shardcollection:"videorental.clients",
key:{firstname:1}})
```

Partitioning / Sharding

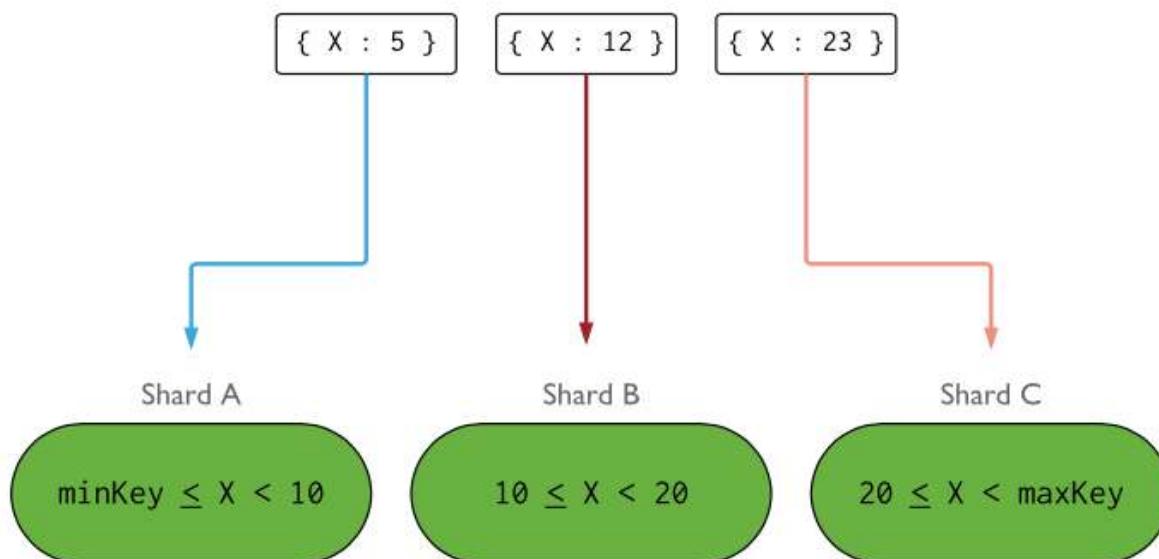
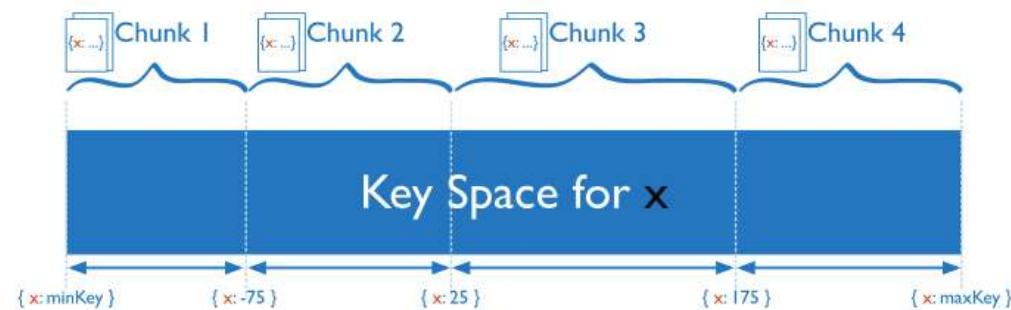
- Hash-based sharding



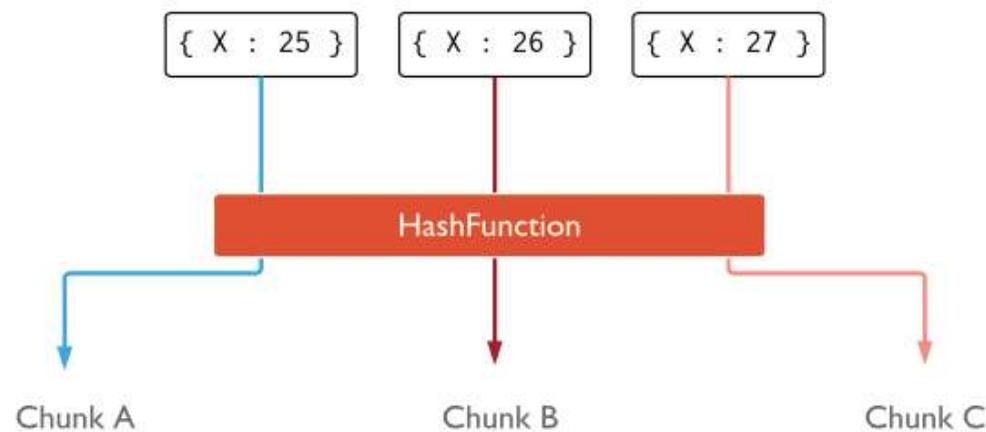
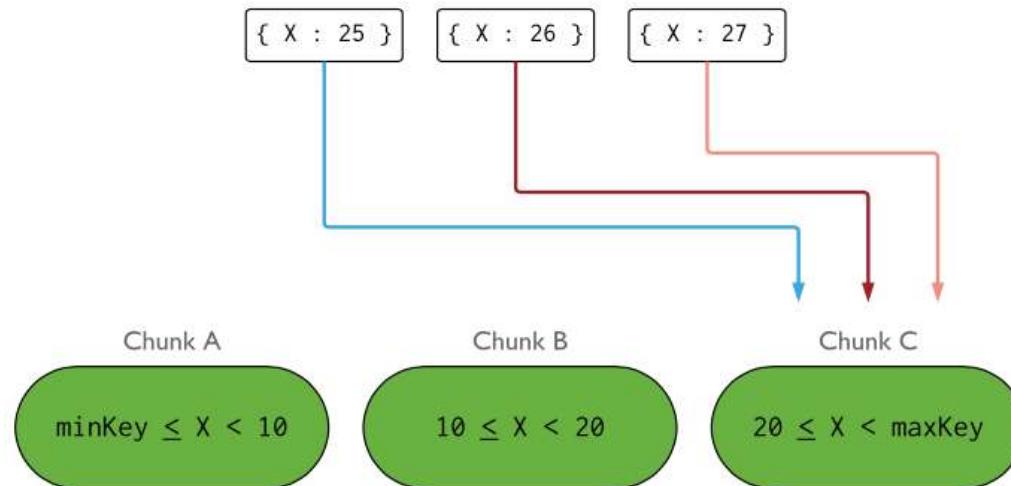
Partitioning / Sharding

- Ranged sharding

Ranged sharding is most efficient when the shard key is
Large Cardinality
Low Frequency
Non-Monotonically Changing

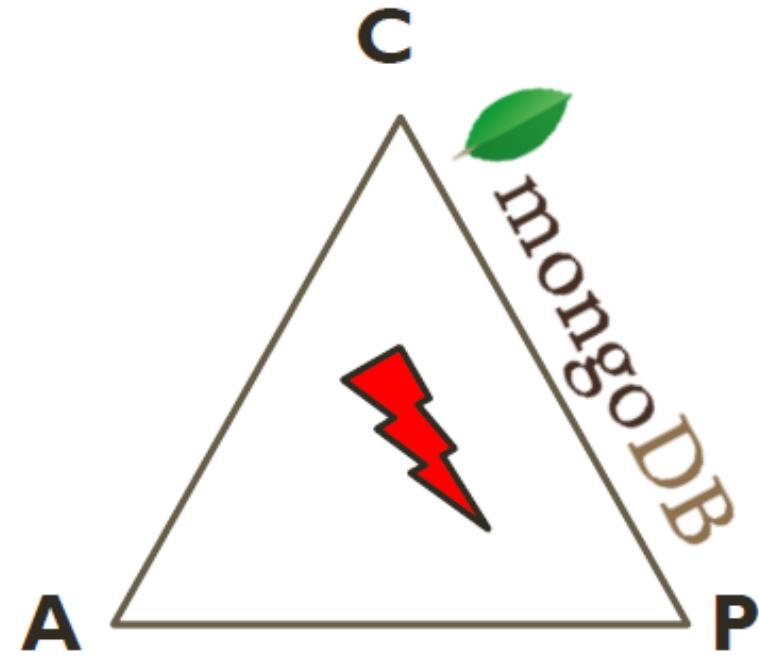


Partitioning with monotonically increasing keys



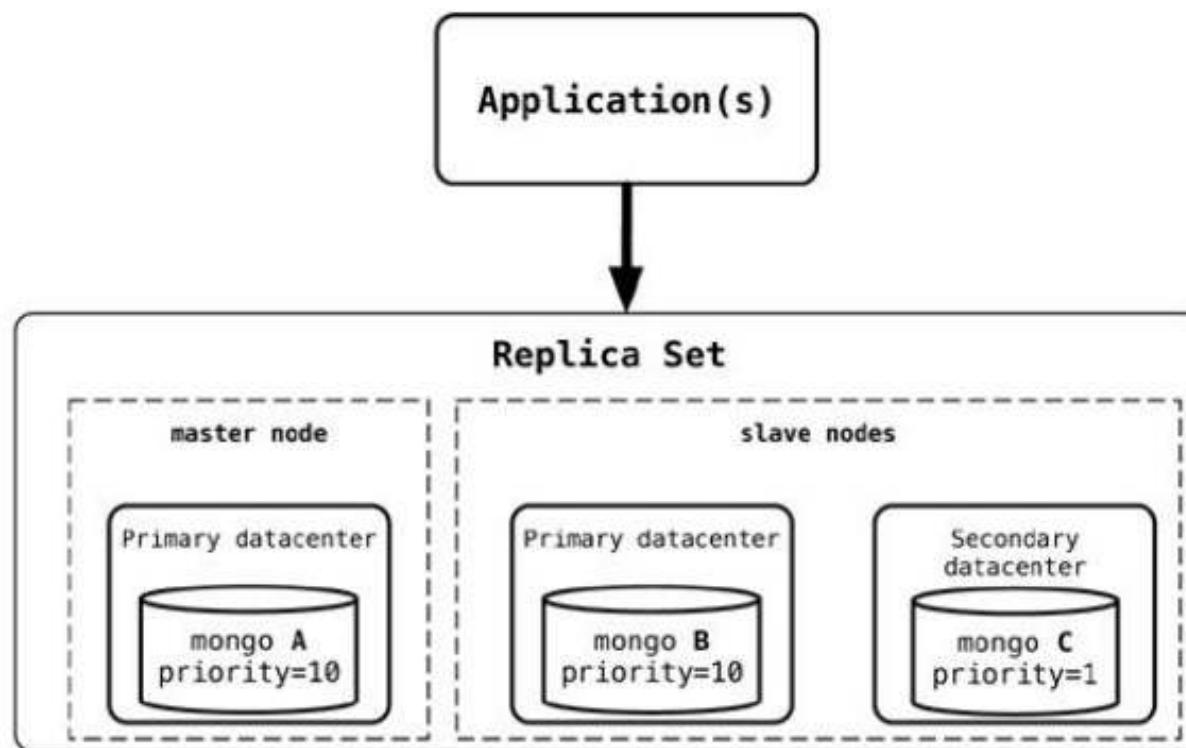
CAP in MongoDB

- Focus on **Consistency** and **Partition tolerance**
- Consistency
 - all replicas contain the same version of the data
- Partition tolerance
 - multiple entry points
 - system remains operational on system split
- Availability
 - system remains operational on failing nodes
 - traded off with consistency



Replication

- A MongoDB database makes use of **replica sets** for consistency and availability following a **master-slave** approach

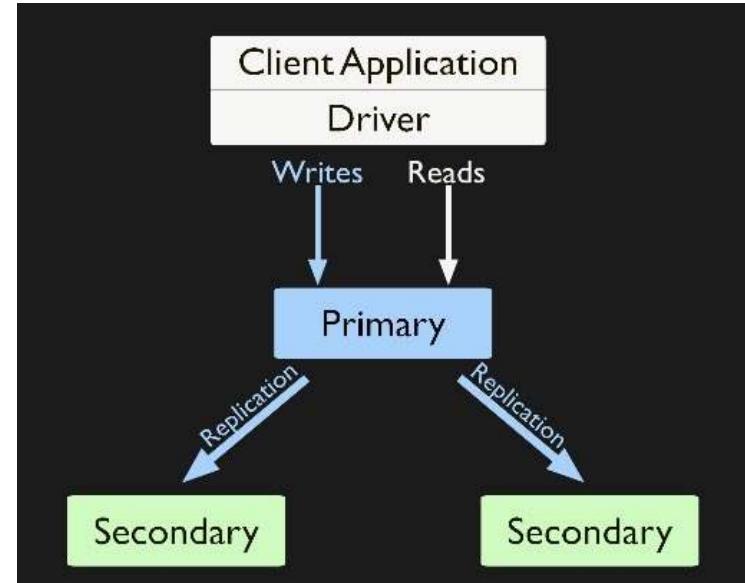


Replica sets

- The replica-set nodes elect the master, or primary, among themselves
 - The one closer to the other servers or having more RAM
 - Users can affect this by assigning a priority to a node
- All requests go to the master node
- Data is replicated to the slave nodes and the clients can get to the data even when the primary node is down
- If the master node goes down, the remaining nodes in the replica set vote among themselves to elect a new master

Replication & fault tolerance

- Primary accepts **all read and write** operations
- Secondaries **only accepts read operations** not write



- Secondaries replicate the primary's oplog and apply the operations to their data sets **asynchronously**
 - With a **replication lag**
- By having the secondaries' data sets reflect the primary's data set, the replica set can continue to function despite the failure of one or more members

Consistency vs availability

- Through the effective use of write concerns and read concerns, the level of consistency and availability, can be adjusted such as
 - waiting for stronger consistency guarantees, or
 - loosening consistency requirements to provide higher availability

Transactions

- All write operations in MongoDB are **atomic** on the level of a single **document**
- When a single write operation modifies multiple documents (e.g. `db.collection.updateMany()`), the modification of each document is atomic, but the operation as a whole is not atomic.
- For situations that require atomicity of reads and writes to multiple documents (in a single or multiple collections), MongoDB supports multi-document transactions

Transactions

- Atomic transactions are possible at the multi-document level since version 4.0 (2018)
- All transactions that contain read operations must use read preference **primary**
 - All operations in a given transaction must route to the same member
- Until a transaction commits, the data changes made in the transaction are not visible outside the transaction

Use cases

Suitable use cases



- **Event Logging**
 - Storing logs of events, acting as a central data store for event storage
 - Events can be sharded by
 - the application that generated the event
 - the type of the event (e.g., order_processed, customer_logged)
- **Content Management Systems, Blogging Platforms**
 - content management systems or applications for publishing websites, managing user comments, user registrations, profiles, web-facing documents.
- **Web Analytics or Real-Time Analytics**
 - store data for real-time analytics; since parts of the document can be updated, it's very easy to store page views or unique visitors, and new metrics can be easily added without schema changes
- **E-Commerce Applications**
 - E-commerce applications often need to have flexible schema for products and orders, as well as the ability to evolve their data structure without expensive database refactoring or data migration



When not to use

- **Complex Transactions Spanning Different Documents**
 - Document data stores are not suited for atomic cross-document operations.
- **Queries against Varying Aggregate Structure**
 - In document databases data is saved as an aggregate in the form of application entities.
 - If the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity.

References & Credits

- References:
- Kristina Chodorow, MongoDB – The definitive guide,
3rd Ed., O'Reilly, 2019
- <https://docs.mongodb.com/>
- Credits:
- Riccardo Torlone, Big Data, Università di Roma Tre
- Kathleen Durant, CS 3200, Northeastern University

From the aggregate-oriented
logical schema to MongoDB
logical/physical schema

Input

- The aggregate-oriented logical schema in metanotation
- The annotated ER diagram
- One NoSQL system S (for today, MongoDB)

Output

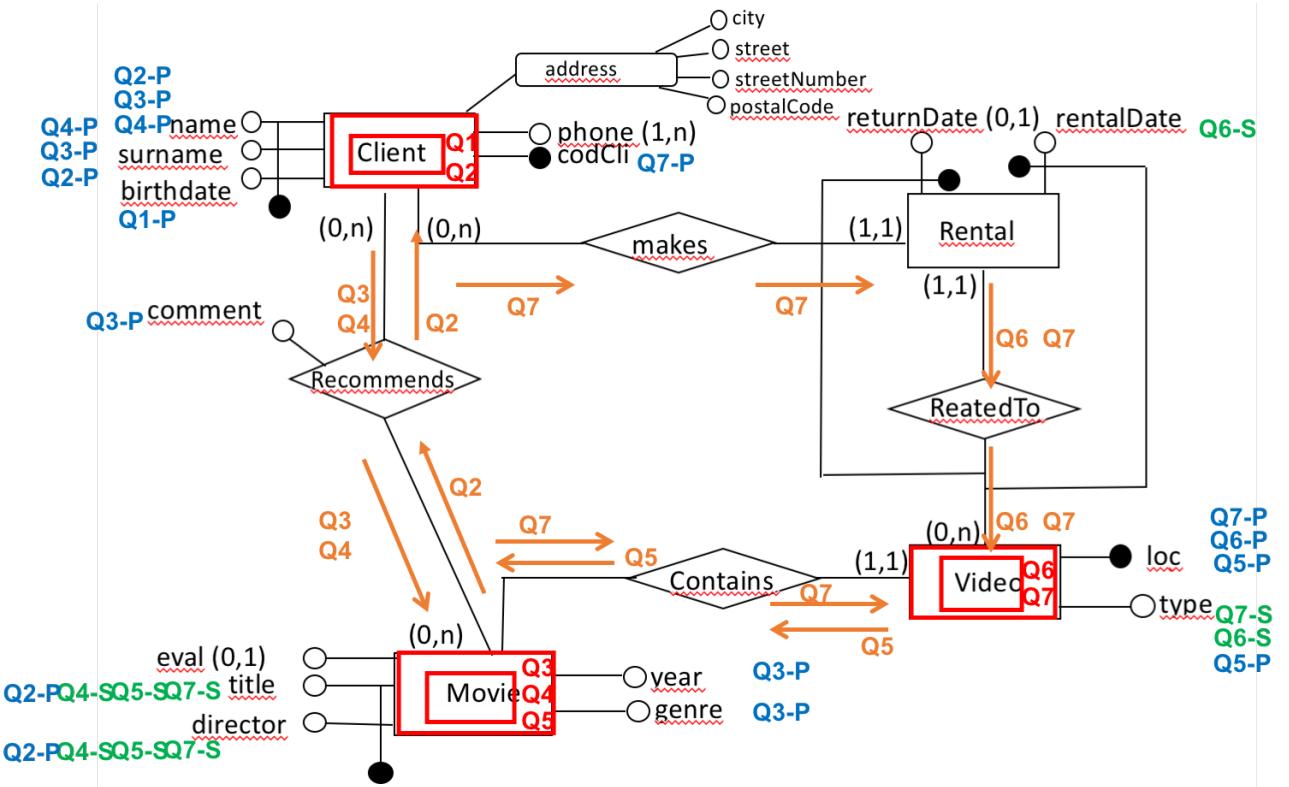
- A schema for S (in metanotation + + partition keys + indexes)

ASSUMPTION: all selection conditions are equality based

Issues

- From the aggregate-oriented logical schemas in meta-notation, the corresponding aggregation entity and the set of associated queries to a set of collections for S
- Each collection allows one subset of the queries to be executed
- From the set of selection attributes and the identifiers of the aggregation entity to the partition key and indexes

Input



- client: {name, surname, birthdate, recommends: [{title, director}]}
 - Q1, Q2
- movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}
 - Q3, Q4, Q5
- video: {loc, type, rentals: [{rentalDate, codCli}]}, title, director
 - Q6, Q7

Design in MongoDB

Remarks

- The identifier in Mongodb always is the `_id` field (ObjectId type)
 - Automatically assigned
 - Monotonically increasing
 - Can be assigned at document insertion, but cannot be updated later
- The `_id` field is automatically indexed
- Partition/shard keys must be indexed
(in what follows, we use the term partition key for uniformity with aggregates, in Mongodb read shard key)

Aggregation entity Client

client: {name, surname, birthdate, recommends: [{title, director}]}

Queries associated with Client: Q1, Q2

Selection attributes for Q1: {}

Selection attributes for Q2: {}

Projection attributes for Q1: {birthdate}

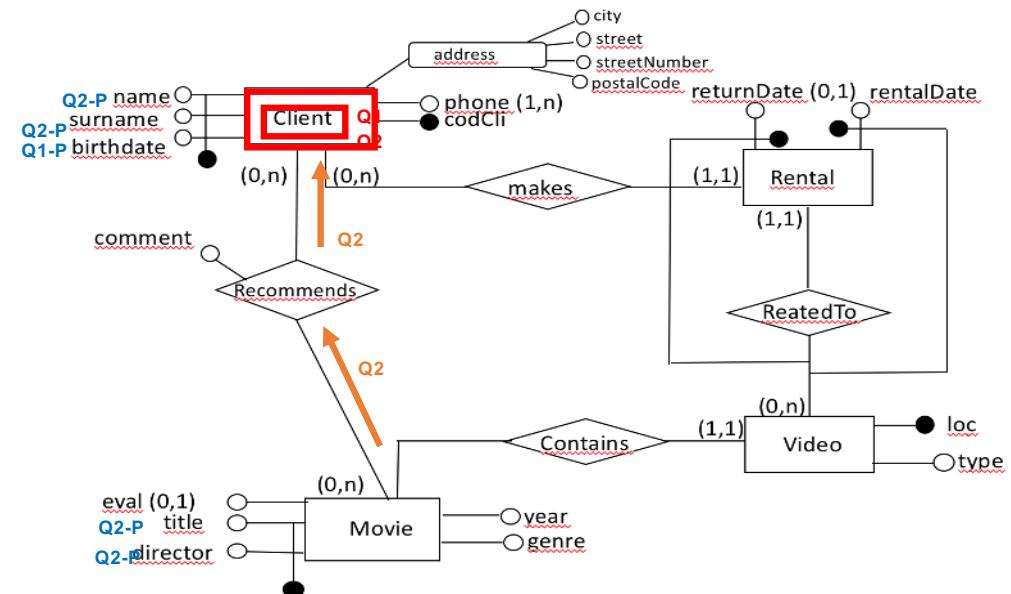
Projection attributes for Q2: {name, surname, title, director}

No selection attribute → no
need for a specific partition key

_id field automatically assigned
could be partition key as well

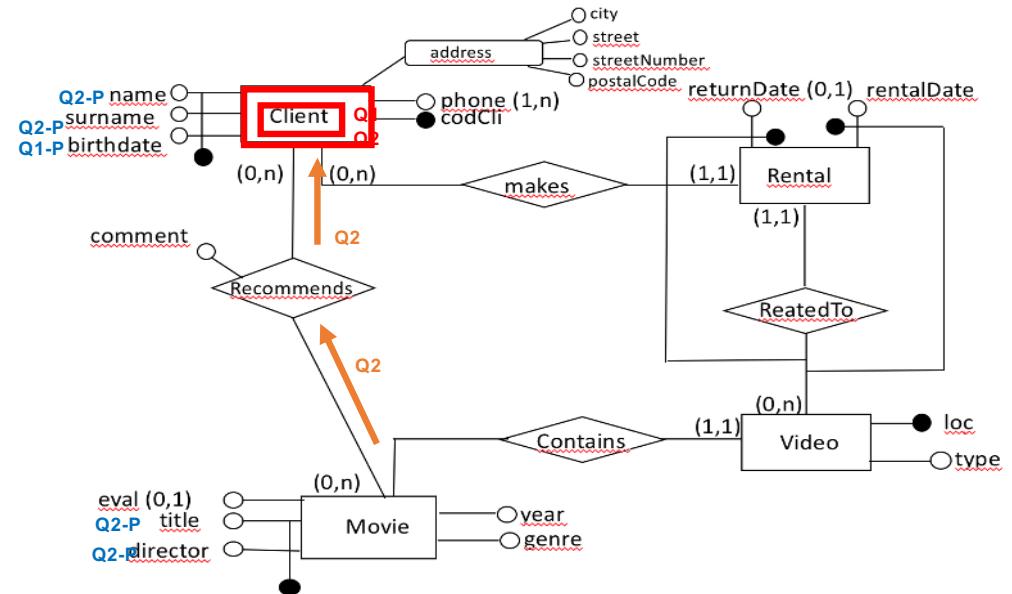
Other partition keys: whatever
you want (name)

Q1. Average age of clients
Q2. Name and surname of clients
and related recommended movies



Aggregation entity Client

client: {_id, name, surname, birthdate, recommends: [{title, director}]}
Q1. Average age of clients
Q2. Name and surname of clients and related recommended movies



```
db.clients.createIndex( {"name": 1, "surname":1, "birthdate":1}, {unique: true})
```

```
db.clients.createIndex( {"name": 1}, {unique: false})
db.adminCommand( { shardCollection: "db.clients", key: { name: 1 },
field: "hashed" } )
```

(non unique index on "name" automatically created if clients is empty)

Aggregation entity Movie

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}

Queries associated with Movie: Q3, Q4, Q5

Selection attributes for Q3: {}

Selection attributes for Q4: { title, director }

Selection attributes for Q5: { title, director }

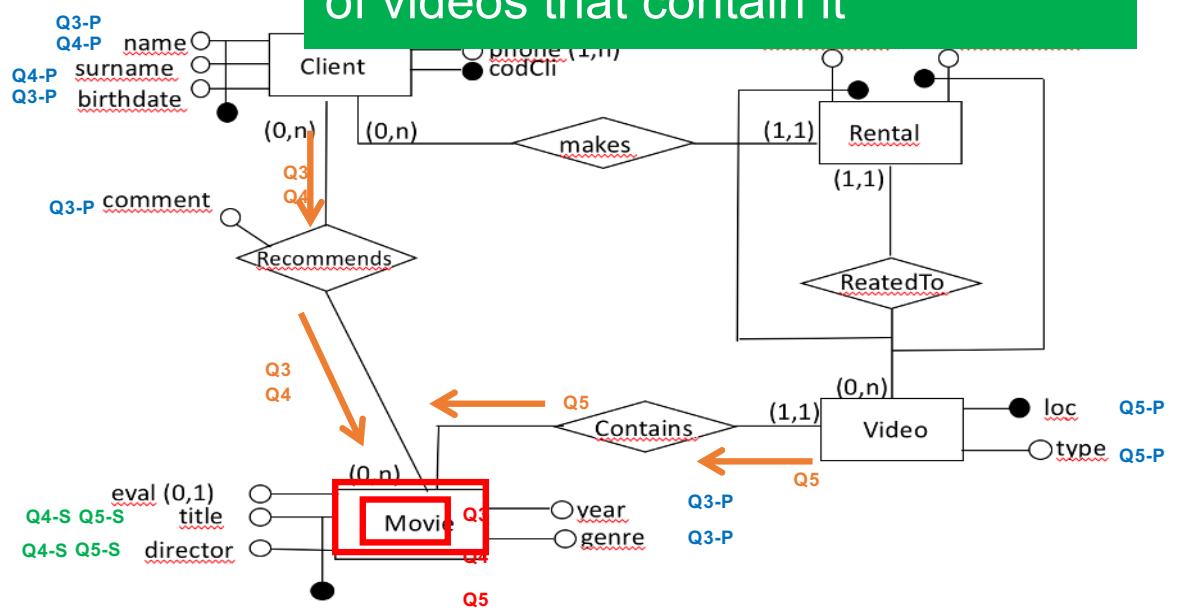
Q3. Genre and year of the movies and their related recommendations, together with the name and the surname of the client who made them

Q4. Name and surname of clients who recommended the movie 'pulp fiction' by 'quentin tarantino'

Q5. Given a movie, all information of videos that contain it

Partition key { title, director }
(with unique index)
hashed

_id field automatically assigned
could be partition key as well



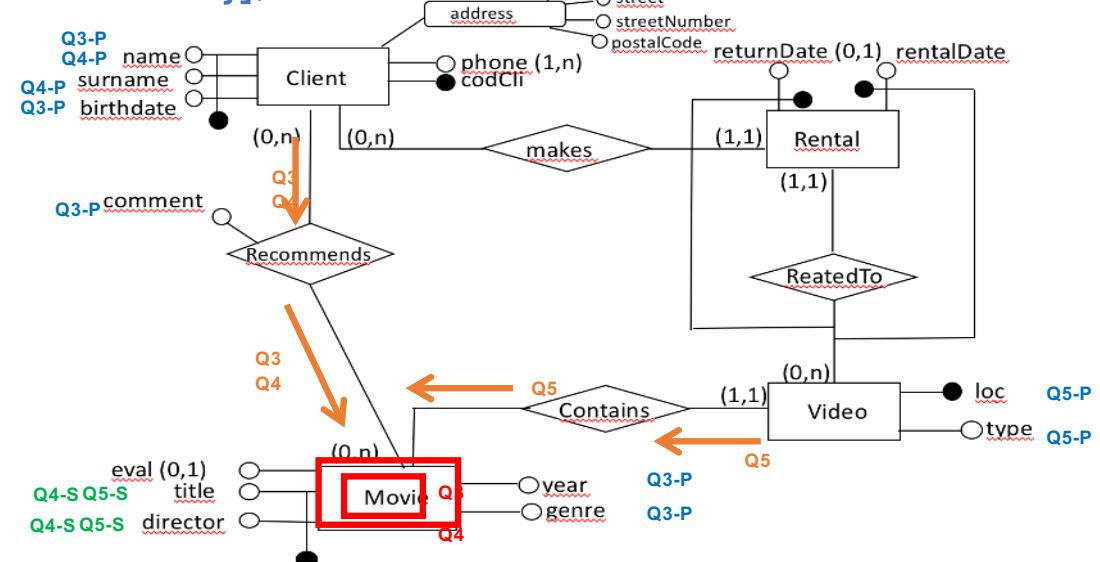
Aggregation entity Movie

movie: {[_id](#), [title](#), [director](#), year, genre,
recommended_by: [{name, surname, comment}],
contained_in: [{loc, type}]}

Q3. Genre and year of the movies and their related recommendations, together with the name and the surname of the client who made them

Q4. Name and surname of clients who recommended the movie 'pulp fiction' by 'quentin tarantino'

Q5. Given a movie, all information of videos that contain it



```
db.movies.createIndex( {"title": 1, "director":1}, {unique: true})
```

```
db.adminCommand( { shardCollection: "db.movies",
                    key: {"title": 1, "director":1},
                    field: "hashed" } )
```

(non unique index on {title, director} automatically created if movies is empty)

Aggregation entity Video

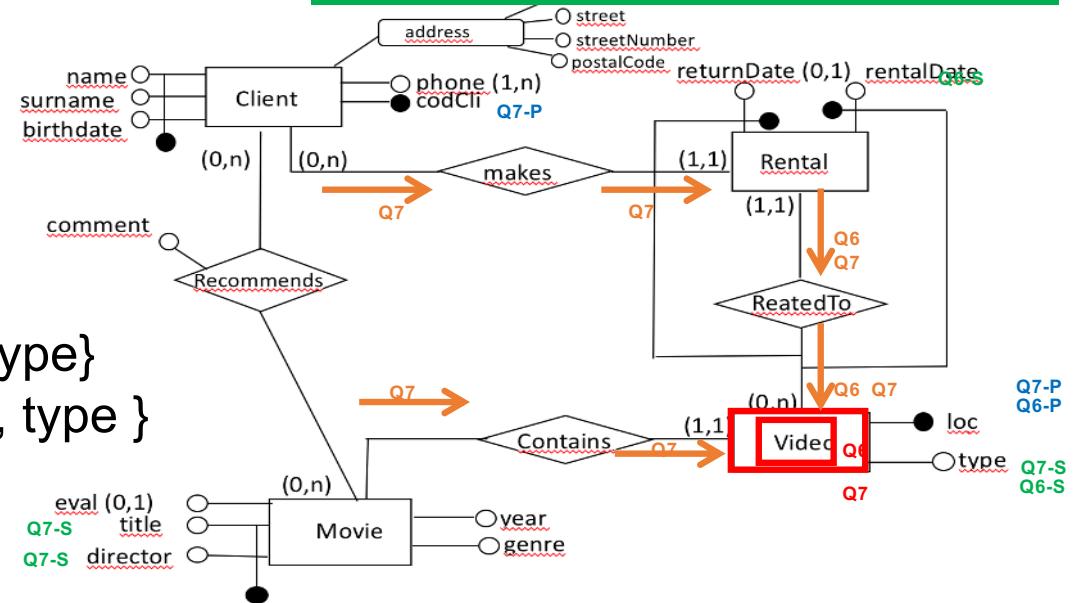
video: {loc, type, rentals: [{rentalDate, codCli}],
title, director}

Q6. Videos of type 'DVD', rented from a certain date
Q7. The videos of type 'VHS' containing the movie 'pulp fiction' by 'quentin tarantino' and the clients that rented them

Queries associated with Video: Q6, Q7

Selection attributes for Q6: { rentalDate, type}

Selection attributes for Q7: { title, director, type }



type appears in both → it becomes the partition key (an equality is always specified)

Partition key = { type }

A non-unique index has to be created on the partition key

Aggregation entity Video

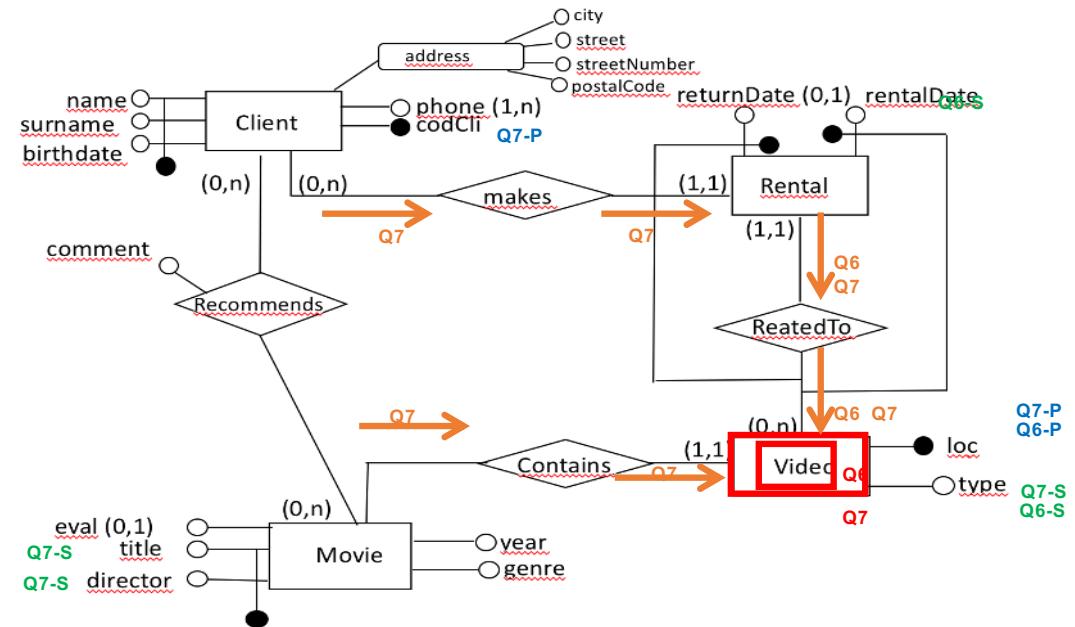
video: {_id, loc, type, rentals: [{rentalDate, codCli}], title, director}

What about the unique index on loc?

MongoDB does not support unique indexes across shards, except when the unique index contains the full shard key as a prefix of the index.

If you don't have an index on the shard key, you should at least have a compound index that starts with the shard key.

Unique index on type, loc



Index rules

- A unique index on `_id` is always created by the system
- All sharded collections must have an index that supports the shard key. The index can be:
 - an index on the shard key or
 - a compound index where the shard key is a prefix of the index
- If sharding is executed on an empty collection, a non-unique index is automatically created on the shard key
- For a ranged sharded collection, only the following indexes can be unique:
 - the index on the shard key
 - a compound index where the shard key is a prefix
 - the default `_id` index; however, the `_id` index only enforces the uniqueness constraint per shard if the `_id` field is not the shard key or the prefix of the shard key.

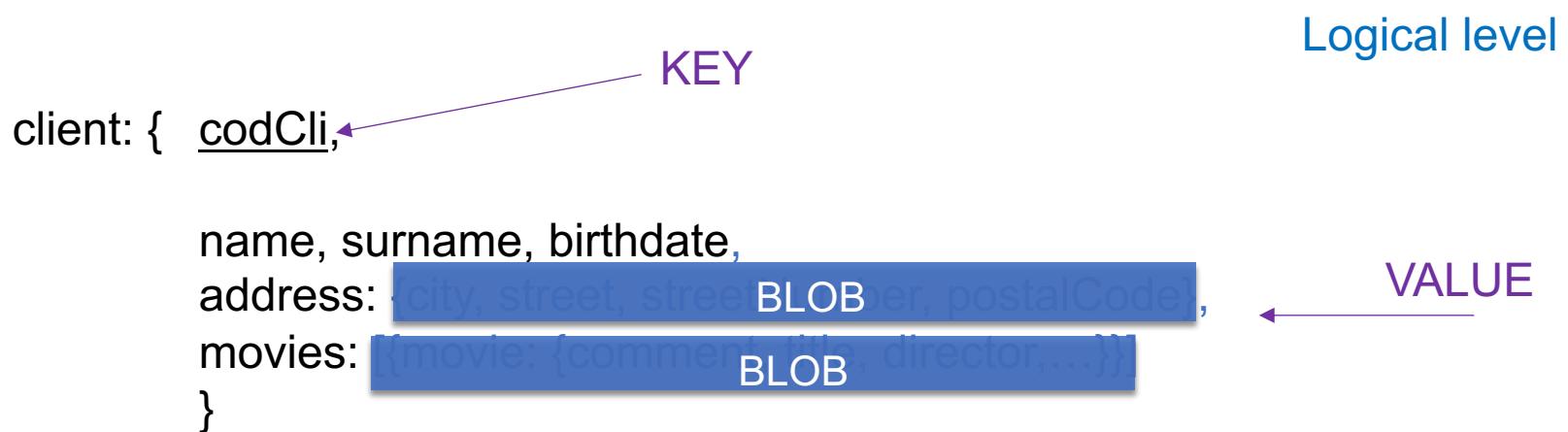
Index rules

- Although you can have a unique compound index where the shard key is a prefix, if using unique parameter, the collection must have a unique index that is on the shard key.
- You cannot specify a unique constraint on a hashed index
- The decision on whether an index must be unique or non-unique depends on the identifiers detected during the aggregate modeling

Column-family NoSQL data stores

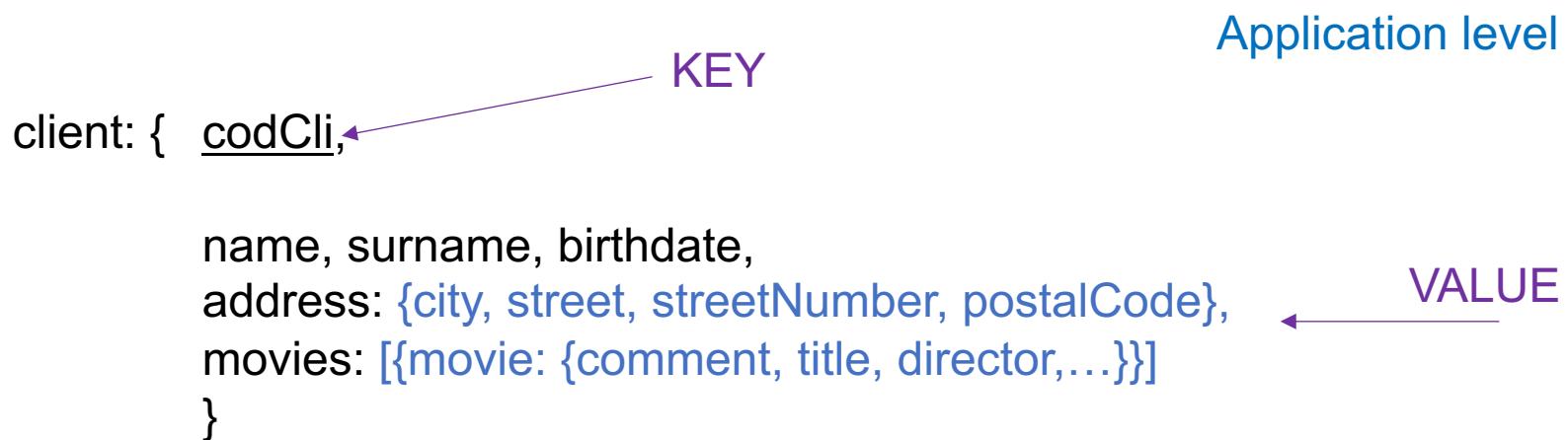
Column-family data model at a glance

- Each data instance is represented in the form (**key, value**)
 - key is an identifier
 - value is the aggregate, corresponding to a set of pairs (**column-key, column-value**)
- Column-value can be either an atomic value or a complex one
- In this second case, **the value structure is visible only at the application level, it is opaque** (a blob) at the logical value
- Data instances can be grouped into logical collections



Column-family data model at a glance

- Each data instance is represented in the form (**key, value**)
 - key is an identifier
 - value is the aggregate, corresponding to a set of pairs (**column-key, column-value**)
- Column-value can be either an atomic value or a complex one
- In this second case, **the value structure is visible only at the application level, it is opaque** (a blob) at the logical value
- Data instances can be grouped into logical collections



Column-family data model at a glance

- Columns can be organised into **families**: sets of related data (columns) that are often accessed together

Logical level

Client

Family: info: {codCli,

name, surname, birthdate,

address: {city, street, streetNumber, postalCode},

}

Column family associated with client personal information

Client

Family:movies: {codCli,

movies: [{movie: {comment, title, director, ...}}]

}

Column family associated with client movie recommendations

Column-family data model at a glance

- Columns can be organised into **families**: sets of related data (columns) that are often accessed together

Application level

Client

Family: info: {codCli,

name, surname, birthdate,

address: {city, street, streetNumber, postalCode},

}

Column family associated with client personal information

Client

Family: movies: {codCli,

movies: [{movie: {comment, title, director,...}}]

}

Column family associated with client movie recommendations

Instance structure

- Limited schema information: column-family names + column-names inside column-families
- Nested values are not visible at the NoSQL system level but only at the application level

Collections

- Pairs can be grouped into logical **collections/namespaces**: sets of aggregates, i.e., sets of (key, value) pairs sharing the same column-family
- Each collection becomes a sort of table

Client

Family: info: {codCli,

```
    name, surname, birthdate,  
    address: {city, street, str BLOB number, postalCode},  
}
```

Client
Family: info

| name | surname | birthdate | address |
|------|---------|-----------|---------|
| | | | |
| | | | |
| | | | |

Collections and schema information

- Schema information can be provided either when inserting the data instances (DML) or before (DDL)
- Inside the same collection, the chunk of data corresponding to a column family **might** correspond (depending on the specific system) to different sets of column-names for each row
- **Flexibility**
- Columns **must** however belong to the same column family

Client
Family: info: {codCli,

name, surname, birthdate,
address: {city, street, streetNumber, postalCode},

}

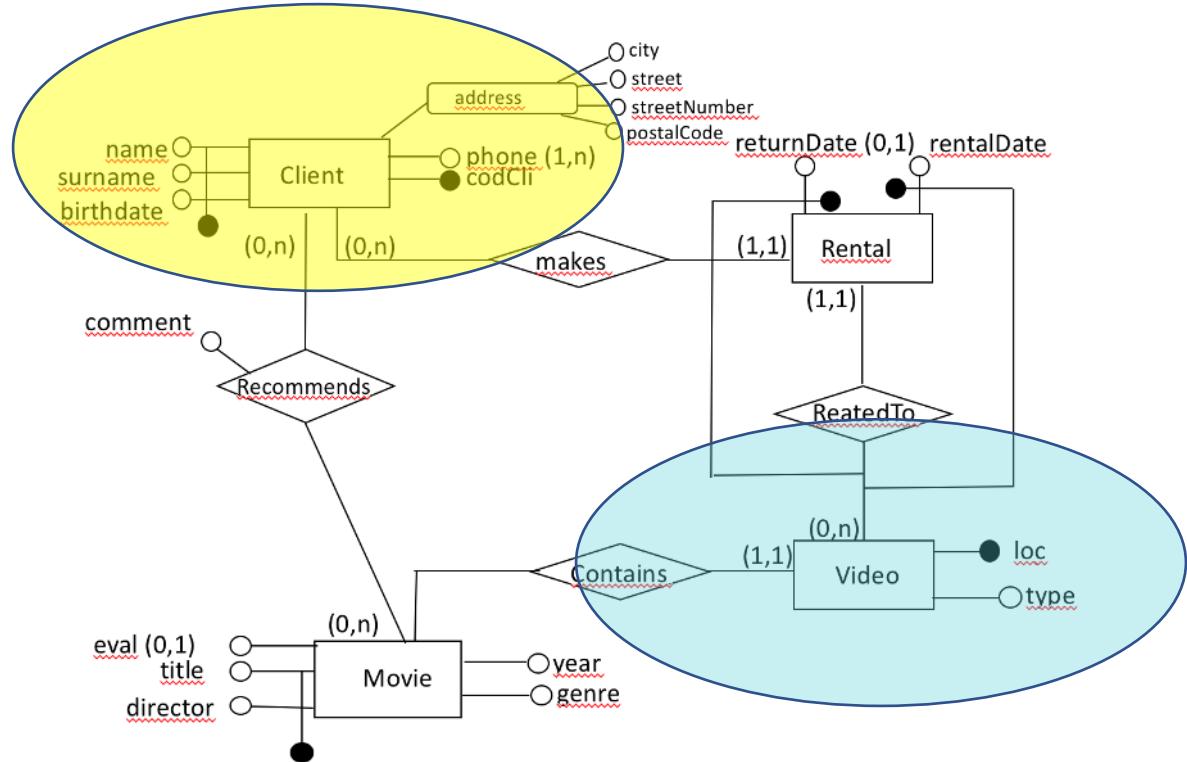


Keys

- (**key**, value) pair
- At the logical level: identification + data retrieval
 - the key **needs** to assume **unique values** in the collection (i.e., it is an **identifier**)
- At the physical level, the key tells the system **how to partition the data** and where to store the data
 - **partition key**
- Usually, the partition key is a subset of or is equal to the primary key
- Aggregates can be directly retrieved **only by specifying values for attributes in the partition key**
 - primary key attributes can be used for retrieval together with the partition key, under some restrictions

Keys

- Usually, more than one attribute
- *Partition key subset of (or equal to) primary key (in bold in the examples)*
- The type and the content of the key is identified starting from the designed aggregates, taking into account the features of the system at hand



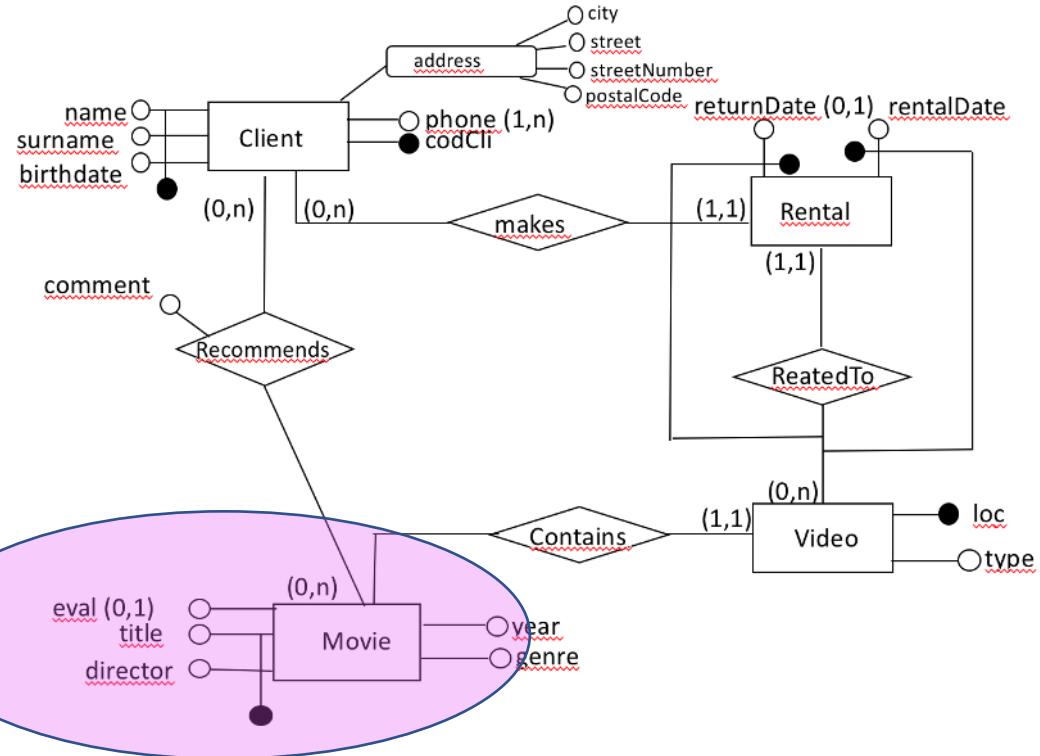
Client: {codCli, ...}

Video: {loc, ...}

primary key is loc
(partition) key is loc

primary key is codCli
(partition) key is codeCli

Keys



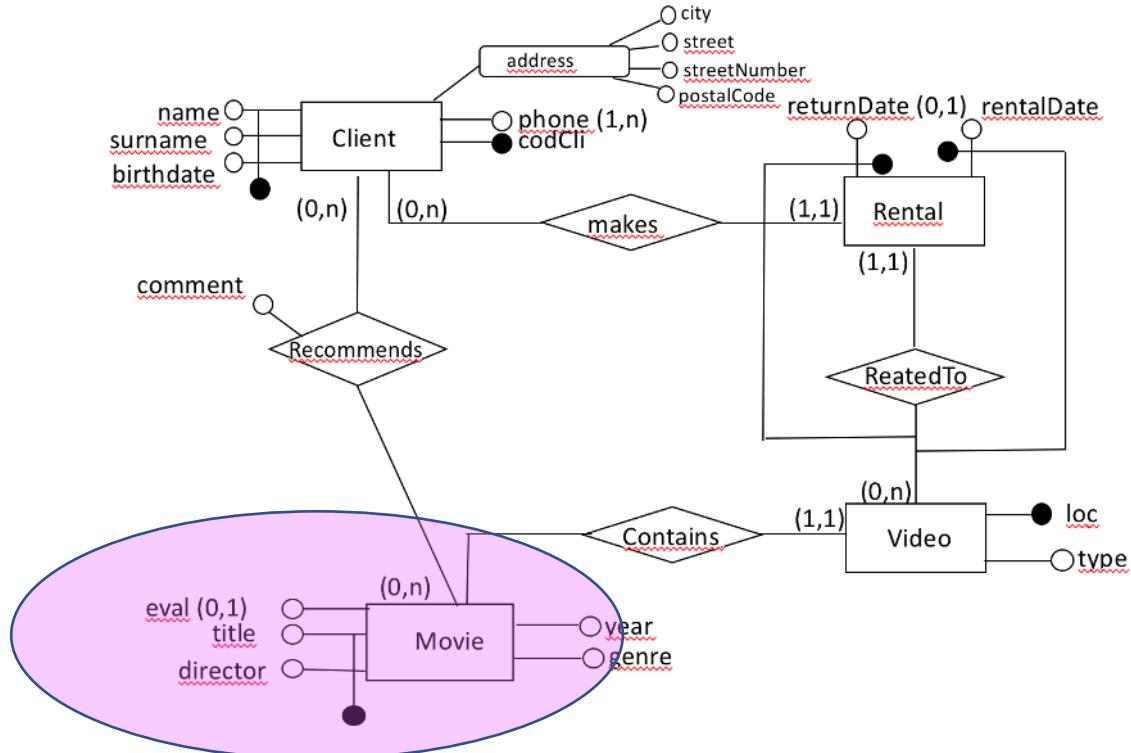
Movie : **{title, director, ...}**

(partition) key could be either
 (title, director)
 title
 director
 → we select (title, director) (in bold)

Keys

Movie : **{title, director}**, ...}

(partition) key could be either
 (title, director)
 title
 director
 → we selected (title, director) (in bold)



Movie aggregate from our modeling example:

movie: { **title, director**,
 year, genre, recommended_by: [{name, surname, comment}],
 contained_in: [{video: {loc, type}}] }

Identifier vs partition/sharding key

- In the Videos collection, videos are partitioned by loc
- In the Clients collection, clients are partitioned by codCli
- In the Movies collection, movies are partitioned by (title, director)
- This impacts the way data are stored
- This impacts the way data can be retrieved

Video Rental Example

client: { codCli,

 name, surname, birthdate,

 address: {city, street, streetNumber, postalCode},
 BLOB

 movies: [{movie: {comment_id: {title, director}, ...}}]
 BLOB

 }
}

REFERENCE

Logical level
(references
exists but they
are opaque in
the system)

movie: { title, director,

 year, genre, recommended_by: [{name, surname, comment}],
 BLOB

 contained_in: [{video: {loc, ...}}] }
 BLOB

video: {loc,

 type, rentals: [{rental: {rentDate, codCli}}, ...] }
 BLOB
 title, director}}

Video Rental Example

Application level
(references can be used at this level, by accessing the structure of opaque components)

```
client: { codCli,
```

```
    name, surname, birthdate,  
    address: {city, street, streetNumber, postalCode},  
    movies: [ {movie: {comment, id:[title, director},...}} ]
```

REFERENCE



```
movie: { id: {title,director},
```

```
    year, genre, recommended_by: [ {name, surname, comment} ],  
    contained_in: [ {video: {loc, type}} ] }
```

REFERENCE

```
video: {loc,
```

```
    type, rentals: [ {rental: {rentalDate, codCli}} ], title, director}}
```

Interaction

- Basic **lookup** based on the **key** and **column name** (of a given column family)
 - `void define(family)`
 - `void insert(key, family, columns)`
 - `columns get(key, family)`
 - `value get(key, family, column)`
 - `value get(key, family, column, value)`
 - `void put(key, family, column, value)`
- It is not possible to arbitrarily **join** data contained in different column-families (always possible **at the application level**)
- It is not possible to navigate the nested structure of the column-values
- In case collections are supported, collection name is also specified

Example 1 – Find the title of the movie in a certain video

Column family: All

video: {loc, type, rentals: [{rentalDate, codCli}],
title, director}

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

value get(collection, key, family, column)

get(Videos, '1234', 'all', [title])

- loc is the partition key, we can retrieve data starting from it
- the title of the movie contained in the video can be directly retrieved from the data store

Example 2 – Find the videos containing a certain movie

Collection family: All

video: {loc, type, rentals: [{rentalDate, codCli}],
title, director}

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

value get(collection, key, column)

- Even if the title and director values are «visible» at the data store level, we have to retrieve data starting from the key
- We can only get all the videos, and
- At the application level, filter them by title and director

Example 3 – Find the videos rented by a certain client

Collection family: All

video: {loc, type, rentals: [{rentalDate, codCli}],
title, director}

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

value get(collection, key, family, column)

- Client data is «not visible» at the data store level
- At the data store level, we can only get all the videos, and
- At the application level, go inside video rentals and filter them by codCli

Example 2 – Find the videos containing a certain movie - with the movies collection

Collection family: All

movie: { title, director,

year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{video: {loc, type}}] }

value get(collection, key, family, column)

- `get(Movies, 'pulp fiction:quentin tarantino', 'all', contained_in)`
- At the data store level, retrieve the information about the videos containing a certain movie (the whole blob)

REMARK: the relationship is part of the aggregate, a single data access to retrieve all the relevant information (contained in a single data node)

Example 4 - Relationships

Find the age(s) of customer(s) that rented a given video

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [{"rental": {"rentalDate": "15/10/2021",  
 "codCli": 375657}}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

```
{  
 "codcli": 375657,  
 "name": "John",  
 "surname": "Black",  
 "birthdate": "15/10/2000",  
 "address": {"city": "Genoa", "street": "Via XX Settembre",  
 "streetNumber": 15, "postalCode": 16100},  
 "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",  
 "director": "quentin tarantino"}},  
 {"movie": {"comment": "very nice", "title": "pulp fiction",  
 "director": "quentin tarantino"}}]  
}
```

get(Videos, 1234, 'all')

- At the data store level, find Video 1234 in collection Videos and retrieve all the information
- at the *application level*, we go inside the aggregate value and we discover that it was rented by Client 375657
- at the *application level*, we can execute get(Clients, 375657, 'all', birthdate) to retrieve information about a/the customer birthdate that rented video 1234 (thus navigating the customer reference stored inside the video), and then the age is computed

Example 4 - Relationships

```
{"loc": 1234,  
 "type": "dvd",  
 "rentals": [ {"rental": {"rentalDate": "15/10/2018",  
 "codCli": 375657},  
 {"rental": {"rentalDate": "15/10/2018",  
 "codCli": 375657},  
 {"rental": {"rentalDate": "15/10/2018",  
 "codCli": 375657}],  
 "title": "pulp fiction",  
 "director": "quentin tarantino"}
```

Find the age(s) of clients (represented by the codCli field) that rented a given video?

- REMARKS:
- 1. Who knows that inside **rentals** there is the code of the client that rented the video?
And where to look for the client information and how to match?
 - The application, the data store is completely unaware of that!
 - 2. Since the relationship is not part of the aggregate, navigating it requires a sort of join, no system support for it
 - at join, no system support for it
 - 3. Since the relationship is not part of the aggregate, navigating it (at application level) requires two distinct data accesses, at two possibly distinct data nodes
- aggregate value and we discover the customer birthdate that rented video 1234 (thus navigating the customer reference stored inside the video), and then the age is computed

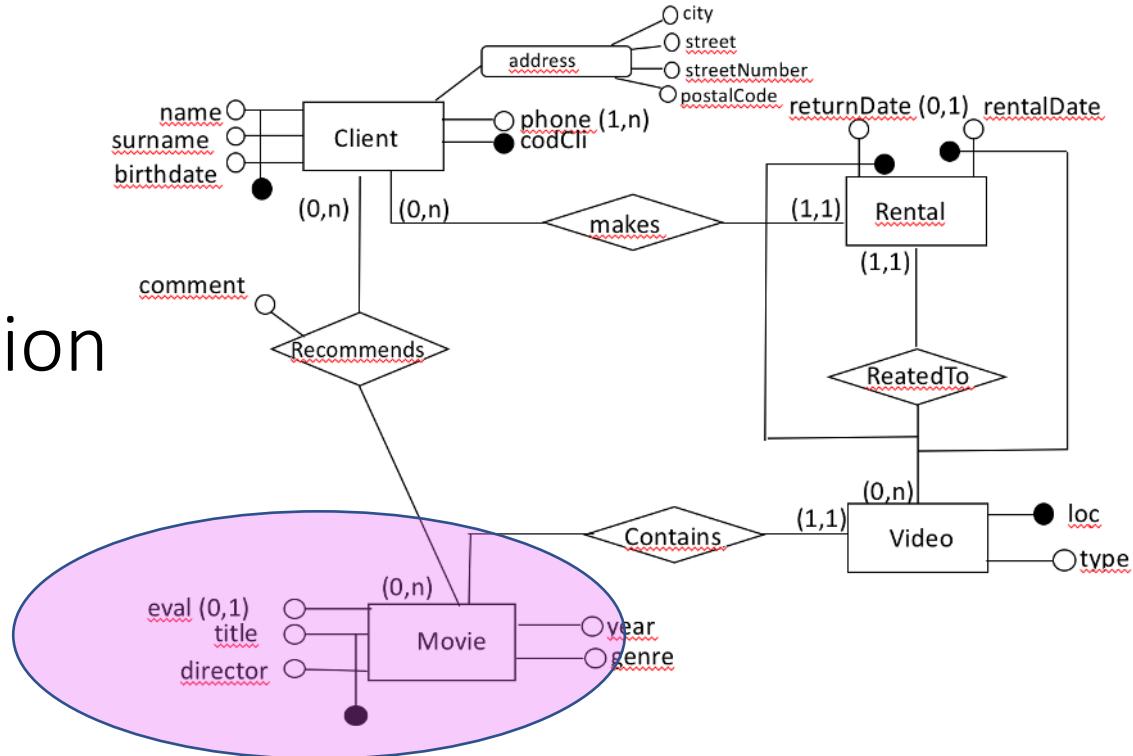
Advanced interaction

- Some column-family systems support SQL-like languages
- Joins are never supported
- *Only queries for which the set of nodes containing relevant data can be determined by the system through the partition key in advance are executed*
 - A value for the partition key has always to be specified
- The execution of other queries (but not joins) can however be forced or admitted through the creation of indexes

Keys physical representation

Movie : {**title**, director ...}

primary key is (title, director)
(partition) key is title



movie: { **title**,

director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{video: {loc, type}}] }

Keys physical representation

movie: { title,

director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{video: {loc, type}}] }

| <u>title</u> | <u>director</u> | year | genre | Recommended_by | Contained_in |
|--------------|-----------------|------|--------|----------------|--------------|
| A | B | 1975 | comedy | ... | ... |
| A | C | 2010 | comedy | ... | ... |



| <u>title</u> | B:year | B:genre | ... | C:year | C:genre | C:... |
|--------------|--------|---------|-----|--------|---------|-------|
| A | 1975 | comedy | | 2010 | comedy | |

Example – Find the videos containing a certain movie - using the movies collection

movie: { title,

director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{video: {loc, type}}] }

(partition) key is title
primary key is (title, director)

```
SELECT contained_in
FROM Movies
WHERE title = 'pulp fiction' AND director = 'quentin tarantino'
```

- An equality condition with respect to the partition key has been specified, the query is admitted
- Single operation at the system level retrieves information about the videos containing a certain movie

Example – Find the videos containing a movie of a certain director - using the movies collection

```
movie: { title,  
         director, year, genre, recommended_by: [ {name, surname, comment} ],  
         contained_in: [ {video: {loc, type}} ] }
```

(partition) key is title
primary key is (title, director)

```
SELECT contained_in  
FROM Movies  
WHERE director = 'quentin tarantino'
```

- This query is not admitted because no partition key value (title) is specified

Popular column-family data stores





A column-family data store

Cassandra

- Used by more than 1500 companies
- Developed at Facebook
 - Initial release: 2008 (stable release: 2013)
 - now, Apache Software License
- Written in: Java
- Operating System: cross-platform
- Java, Scala, Ruby, C#, Python, Perl, PHP, C++
Cassandra clients

Who uses Cassandra?



- Some of the largest production deployments:
 - **Apple**: over 75,000 nodes storing over 10 PB of data
 - **Netflix**: 500 nodes, 420 TB, over 1 trillion requests per day
 - **eBay**: over 100 nodes, 250 TB

Who else uses Cassandra?



Data model and interaction

Cassandra data model

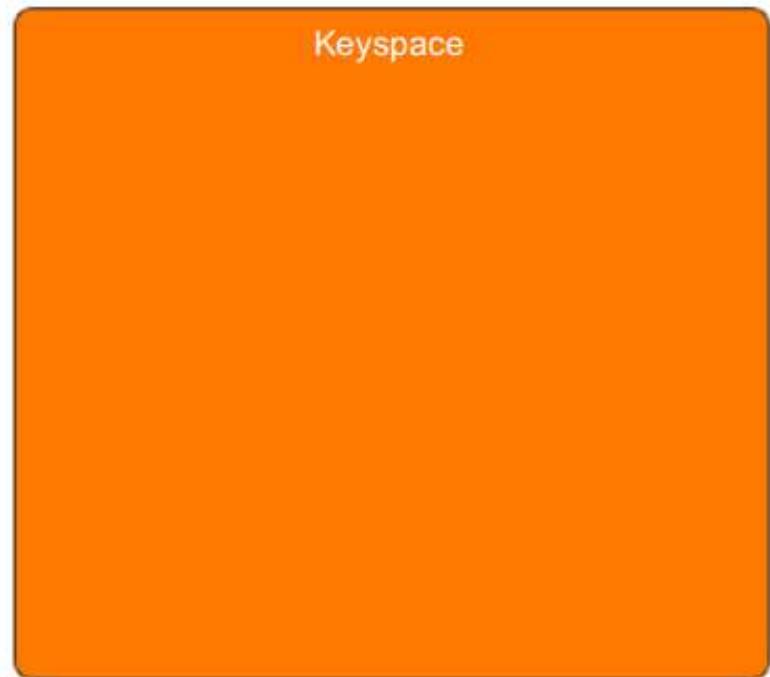
- The original Cassandra data model is similar to that discussed for generic column-family data stores, with a basic interaction protocol - Thrift
- More recently, a new high level data model (with primary keys) and related (SQL-like) language has been proposed
- We will discuss only the high level data model and language since the interaction protocol for the original data model is now deprecated

Data model and interaction

- Concepts closer to a traditional relational database with tables, columns and rows
- Cassandra Query Language (CQL) – similar to SQL
 - Still the same data underneath, just abstracted away
 - reintroduction of schema so that you don't have to read code to understand the data model
 - CQL creates a common language so that details of your data can be easily communicated (but it is not a standard)
 - best-practice Cassandra interface that hides the details of the original data model
- CQL is one of the strongest reasons why Cassandra has become so widely used today

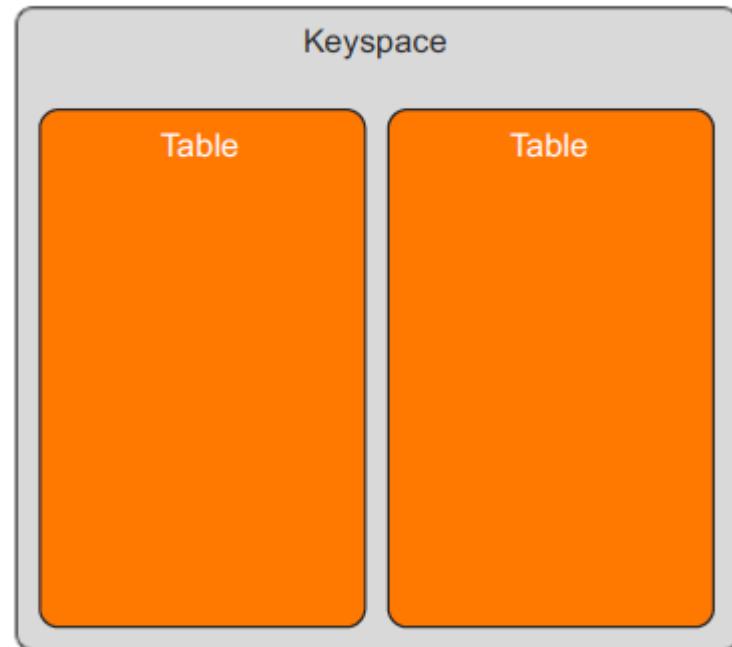
Data model

- Keyspace – Database in an RDBMS



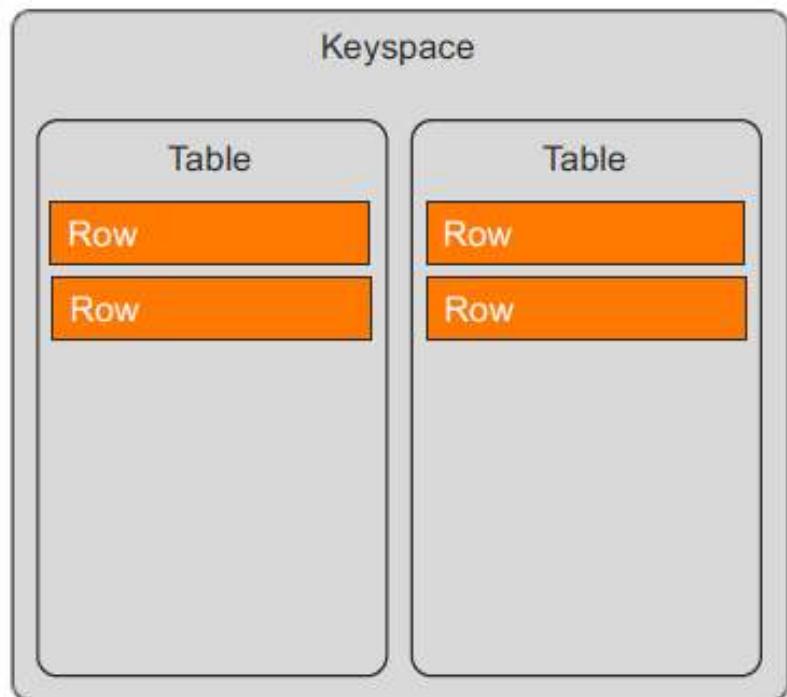
Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS



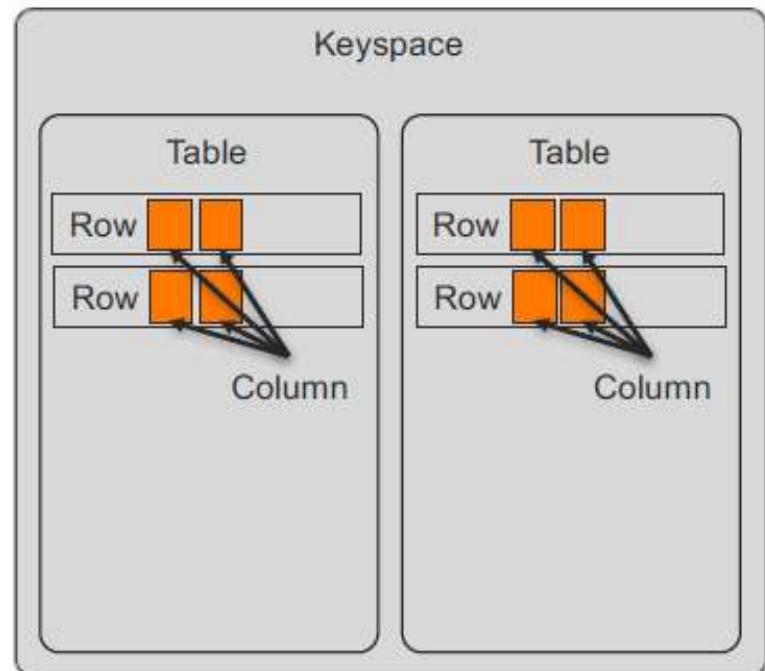
Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS



Data model

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS
- Column – Column in an RDBMS



CQL at the glance

```
CREATE TABLE clients (  
    codCli int PRIMARY KEY,  
    name text,  
    surname text,  
    birthdate date );
```

It looks like SQL but the original Cassandra model is still there, even if hidden by the CQL data model

```
INSERT INTO clients (codCli, name, surname, birthdate)  
VALUES (3, 'john', 'black', '15/10/2000');
```

```
SELECT * FROM users;
```

| <i>codCli</i> | <i>name</i> | <i>surname</i> | <i>birthdate</i> |
|---------------|-------------|----------------|-------------------|
| 3 | <i>john</i> | <i>black</i> | <i>15/10/2000</i> |

CQL atomic data types

- Strings
 - Ascii, varchar, text
- Numbers
 - Bigint, decimal, double, float, int,...
- Time
 - Date, time, timestamp
- ...

CQL – CREATE TABLE and partition key

- Similar to SQL CREATE TABLE

Partition key codCli
Primary key codCli

```
CREATE TABLE clients (
    codCli int PRIMARY KEY,
    name text,
    surname text,
    birthdate date );
```

- By default, the **partition key** is the first column of a composite key (thus, partition key is not unique but do not forget the match with the old Cassandra model)

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director));
```

Partition key (title, director)
Primary key (title, director)

Partition key title
Primary key (title, director)

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((title, director)));
```

CQL – CREATE TABLE and partition key

Partition key =

first attribute of the primary key

or

a prefix of the primary key, specified inside ()

CQL – CREATE TABLE

- No foreign keys

```
CREATE TABLE movie (
    title text,
    director text,
    Year int,
    genre text,
    PRIMARY KEY ((title, director));
```

```
CREATE TABLE video (
    loc int PRIMARY KEY,
    type text,
    title text,
    director text);
```

There is no way to specify the foreign key

Side effect:

Table video can refer movies not included in table movie

CQL - INSERT

- Syntax similar to SQL

```
INSERT INTO t2 (a, b, c, d) VALUES (0,0,0,0);
```

- **No primary key violation:** if you insert a row with the same primary key of another row in the table, the first will be simply updated
 - INSERT becomes an UPDATE!

CQL – CREATE TABLE and partition key

- All the attributes that follow the partition key attributes in a primary key definition are called **clustering columns**
- In a node, all rows in a collection with the same partition key value are **locally sorted with respect to clustering columns** (default: ascending order)
- Consequences on how to query data

Example

```
CREATE TABLE T
(
    A int,
    B int,
    C int,
    D int,
    PRIMARY KEY (A, B, C)
);
```

```
INSERT INTO T (A, B, C, D) VALUES (0,0,0,0);
INSERT INTO T (A, B, C, D) VALUES (0,1,2,2);
INSERT INTO T (A, B, C, D) VALUES (0,0,1,1);
INSERT INTO T (A, B, C, D) VALUES (0,1,3,3);
INSERT INTO T (A, B, C, D) VALUES (1,1,4,4);
```

Two partitions
(physical level)

| | A | B | C | D |
|----|---|---|---|--------------|
| P1 | | | | (1, 1, 4, 4) |
| P2 | | | | (0, 0, 0, 0) |
| | | | | (0, 0, 1, 1) |
| | | | | (0, 1, 2, 2) |
| | | | | (0, 1, 3, 3) |

Example

Two partitions
(physical level)

| | A | B | C | D |
|----|--------------|---|---|---|
| P1 | (1, 1, 4, 4) | | | |
| P2 | (0, 0, 0, 0) | | | |
| | (0, 0, 1, 1) | | | |
| | (0, 1, 2, 2) | | | |
| | (0, 1, 3, 3) | | | |

If the partitions are physically stored on two nodes N1 and N2...

Node N1

A 1:4:D

1 4

Node N2

A 0:0:D 0:1:D 1:2:D. 1:3:D

0 0 1 2 3

If the partitions are physically stored on a single node N...

Node N

A 1:4:D

1 4

A 0:0:D 0:1:D 1:2:D. 1:3:D

0 0 1 2 3

CQL - UPDATE

```
CREATE TABLE T
(
    a int,
    b int,
    c int,
    d int,
    PRIMARY KEY (a, b, c)
);

INSERT INTO T (a, b, c, d) VALUES (0,0,0,0);
```

- UPDATE is similar to INSERT!

```
UPDATE T SET a = 0 AND b = 0 AND c = 0 AND d = 1;
```

- is equivalent to

```
INSERT INTO T (a, b, c, d) VALUES (0,0,0,1);
```

- It can contain a WHERE clause following the same rules of the WHERE clause in SELECT (see later)

CQL - DELETE

- Marks data for removal from a table
- Data is removed later through a process called **compaction**
- Unlike SQL it MUST contain a WHERE clause, following the same rules than the WHERE clause in SELECT (see later)

```
DELETE FROM T  
WHERE a = 0 AND b = 0 AND c = 0;
```

CQL - SELECT

- Similar to SELECT statement in SQL
- Retrieve all the rows from a table

```
SELECT * FROM T;
```

- Project all the rows from a table

```
SELECT a FROM T;
```

- Select some rows from a table, with many limitations

```
SELECT * FROM T WHERE a = 0 AND b = 0 AND c = 1;
```

CQL – SELECT, restrictions

- Many restrictions on what you can do in the WHERE clause
 - no join
 - restrictions on selection conditions
- Some queries cannot be executed
- Solutions
 - Query-based design
 - Indexes
 - Forcing the execution of non-admitted queries

CQL – SELECT, restrictions: no join

- Queries must refer a single table
- Joins are not supported
 - As we know, joining (portions of) tables stored in different nodes leads to a high data communication
 - Impact on performance
- Two options to overcome the problem
 1. Go back to the logical design and restructure your data (query-based design)
 2. Execute joins by writing specific applications, accessing data stored in Cassandra
 - Store data in Cassandra and use MapReduce frameworks or Spark

CQL – SELECT, restrictions: no join

- If you design your aggregates using the methodology, all workload queries can be executed over your data without the need of join execution
- However, join execution might be needed for executing queries that do not belong to the workload

Example

```
CREATE TABLE movie (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((title, director));
```

```
CREATE TABLE video (
    loc int PRIMARY KEY,
    type text,
    title text,
    director text);
```

```
SELECT year
FROM movie M, video V
WHERE M.title = V.title AND
M.director = V.director AND loc
= 1234;
```

Non admitted

Example – approach 1

```
CREATE TABLE movie (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((title, director));
```

```
CREATE TABLE video (
    loc int PRIMARY KEY,
    type text,
    title text,
    director text);
```

```
CREATE TABLE movie_video (
    title text,
    director text,
    year int,
    genre text,
    loc int PRIMARY KEY,
    type text);
```

```
SELECT year
FROM movie_video
WHERE loc = 1234;
```

Admitted

Query-based design: go back to the design

Example – approach 2

- Write a program that do the following:

1. executes

```
SELECT title, director  
FROM videos  
WHERE loc = 1234;
```

2. Let **t** represent the returned tuple

3. Execute

```
SELECT year  
FROM movies  
WHERE title = t.title AND director = t.director;
```

CQL – SELECT, restrictions: selection conditions

- The WHERE clause **must** contain an equality-based selection condition on each attribute of the partition key
 - The system is able to identify the nodes contributing to the result
 - IN clause is also allowed (a set of nodes is identified)
- The WHERE clause **can** contain selection conditions on clustering columns, following the ordering provided in the primary key, with some additional restrictions
 - On the identified nodes, rows to be returned must be sequentially stored

Example

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director);
```

Non admitted

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      year = 2000
```

```
SELECT*
FROM movies
WHERE title = 'pulp fiction'
```

Admitted

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      director= 'quentin tarantino'
```

Admitted

```
SELECT*
FROM movies
WHERE director= 'quentin tarantino'
```

Non admitted

CQL – SELECT, restrictions: selection conditions

- Data to be retrieved from a partition has to be sequentially stored
- Clustering columns support `=`, `IN`, `>`, `>=`, `<`, `<=`, `CONTAINS` operators
 - `CONTAINS` only for collection types, see later
- The WHERE clause can contain conditions over any prefix of the clustering column list, as defined in the primary key
- If more than one clustering column has been defined, range restrictions are allowed only on the last clustering column being restricted in the WHERE clause

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director, year);
```

Example

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      director= 'quentin tarantino' AND
      year > 2000
```

Admitted

Non admitted

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      year > 2000
```

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, year, director);
```

Example

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      director= 'quentin tarantino' AND
      year > 2000
```

Non admitted

Admitted

```
SELECT*
FROM movies
WHERE title = 'pulp fiction' AND
      year > 2000
```

Indexes

- Exceptions to the previous rules are possible if indexes are created
- An index can be created on any column
 - B+-Tree indexes
 - Local at any node
- WHERE conditions on any attributes upon which an index has been created are admitted

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (    title,
                      director,
                      year);
```

```
CREATE INDEX ON movies(year);
```

```
SELECT*
FROM movies
WHERE year > 2000;
```

Admitted

Enforcing query execution

- Executing a non admitted query, you will get the following message

Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING.

- The system tells you that the execution of your request **might be** inefficient
- You can force the query execution by using clause
ALLOW FILTERING
enforcing a full scan of data (quite inefficient)

Example

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title,
                  director,
                  year);
```

```
CREATE INDEX ON movies(year);
```

```
SELECT*
FROM movies
WHERE year > 2000;
```

```
SELECT*
FROM movies
WHERE year > 2000
ALLOW FILTERING ;
```

Admitted

Admitted

Other clauses

- Aggregate functions: COUNT, SUM, MAX, MIN,...
- GROUP BY
 - Over primary key columns in defined order as arguments
- ORDER BY
 - Depending on the clustering order

CQL – Collection types

- Extension of basic features of a column-family data mode with nested atomic values: **sets, lists, maps**
- Sets hold list of unique elements **that are visible to the system**
- Lists hold ordered, possibly repeating elements
- **Maps hold a list of key-value pairs**
- Collection types cannot be nested
- Collection field cannot be used in primary keys

```
CREATE TABLE mytable(
    row text,
    Y text,
    myset set<text>,
    mylist list<int>,
    mymap map<text, text>,
    PRIMARY KEY (row)
);
```

CQL – Collection types - INSERT

- Inserting

```
INSERT INTO mytable (row, myset)
VALUES (123, { 'apple', 'banana'});
```

```
INSERT INTO mytable (row, mylist)
VALUES (123, ['apple','banana','apple']);
```

```
INSERT INTO mytable (row, mymap)
VALUES (123, {1:'apple',2:'banana'})
```

```
CREATE TABLE mytable(
    row text,
    Y text,
    myset set<text>,
    mylist list<text>,
    mymap map<int, text>,
    PRIMARY KEY (row)
);
```

CQL – Collection types - UPDATE

- Updating

```
UPDATE mytable SET myset = myset + {'apple','banana'}  
WHERE row = 123;
```

```
UPDATE mytable SET myset = myset - { 'apple' }  
WHERE row = 123;
```

```
UPDATE mytable SET mylist = mylist + ['apple','banana']  
WHERE row = 123;
```

```
UPDATE mytable SET mylist = ['banana'] + mylist  
WHERE row = 123;
```

```
UPDATE mytable SET mymap['fruit'] = 'apple'  
WHERE row = 123
```

```
UPDATE mytable SET mymap = mymap + { 'fruit':'apple' }  
WHERE row = 123
```

```
CREATE TABLE mytable(  
    row text,  
    Y text,  
    myset set<text>,  
    mylist list<text>,  
    mymap map<text, text>,  
    PRIMARY KEY (row)
```

Example – collection types

```
CREATE TABLE clients (
codCli int PRIMARY KEY,
name text,
surname text,
birthdate date,
emails set<text>,
phones map<text, text>,
hobbies list<text>
);
```

```
CREATE INDEX ON clients (name);
CREATE INDEX ON clients (emails);
CREATE INDEX ON clients (hobbies);
CREATE INDEX ON clients (keys(phones));
CREATE INDEX ON clients (values(phones));
CREATE INDEX ON clients (entries(phones));
```

```
SELECT * FROM clients
WHERE name = 'Benjamin';

SELECT * FROM clients
WHERE emails CONTAINS
      'Benjamin@oops.com';

SELECT * FROM clients
WHERE hobbies CONTAINS tennis';
```

```
SELECT * FROM clients
WHERE phones CONTAINS KEY 'office';
```

```
SELECT * FROM clients
WHERE phones CONTAINS '0108567586';
```

```
SELECT * FROM clients
WHERE phones['office'] = '0108567586';42
```

CQL - User Defined Types

- User defined types allow a compact usage of complex data
- Nesting
- An alternative approach for dealing with associations and joins

Example

```
CREATE TYPE address_t(  
    city text,  
    street text,  
    streetNumber int,  
    postalCode int);
```

```
client: {  
    codCli,  
    name, surname, birthdate,  
    address: {city, street, streetNumber, postalCode},  
    movies: [ {comment, title, director,...}]  
}
```

```
CREATE TYPE movie_t(  
    comment text,  
    title text,  
    director int,  
    ...);
```

```
CREATE TABLE clients (  
    codCli int PRIMARY KEY,  
    name text,  
    surname text,  
    birthdate date,  
    address frozen<address_t>,  
    movies set<frozen<movie_t>>  
) ;
```

Values for `address_t` treated as a blob (no way to select or update components, you should do that at the application level)

Values for `movie_t` treated as a blob (no way to select components, you should do that at the application level)

Frozen types

- Frozen <> makes a component **opaque** for the system
- Frozen<> can be used with both user defined types and collection types, **it is mandatory in most situations**
- With frozen <>, we can overcome most limitations related to collection types and UDTs
- Collection types and UDTs nesting **is allowed by using frozen types**
- Frozen types can be used inside either frozen or non frozen collection types
- Collection fields with frozen types can be used as primary key

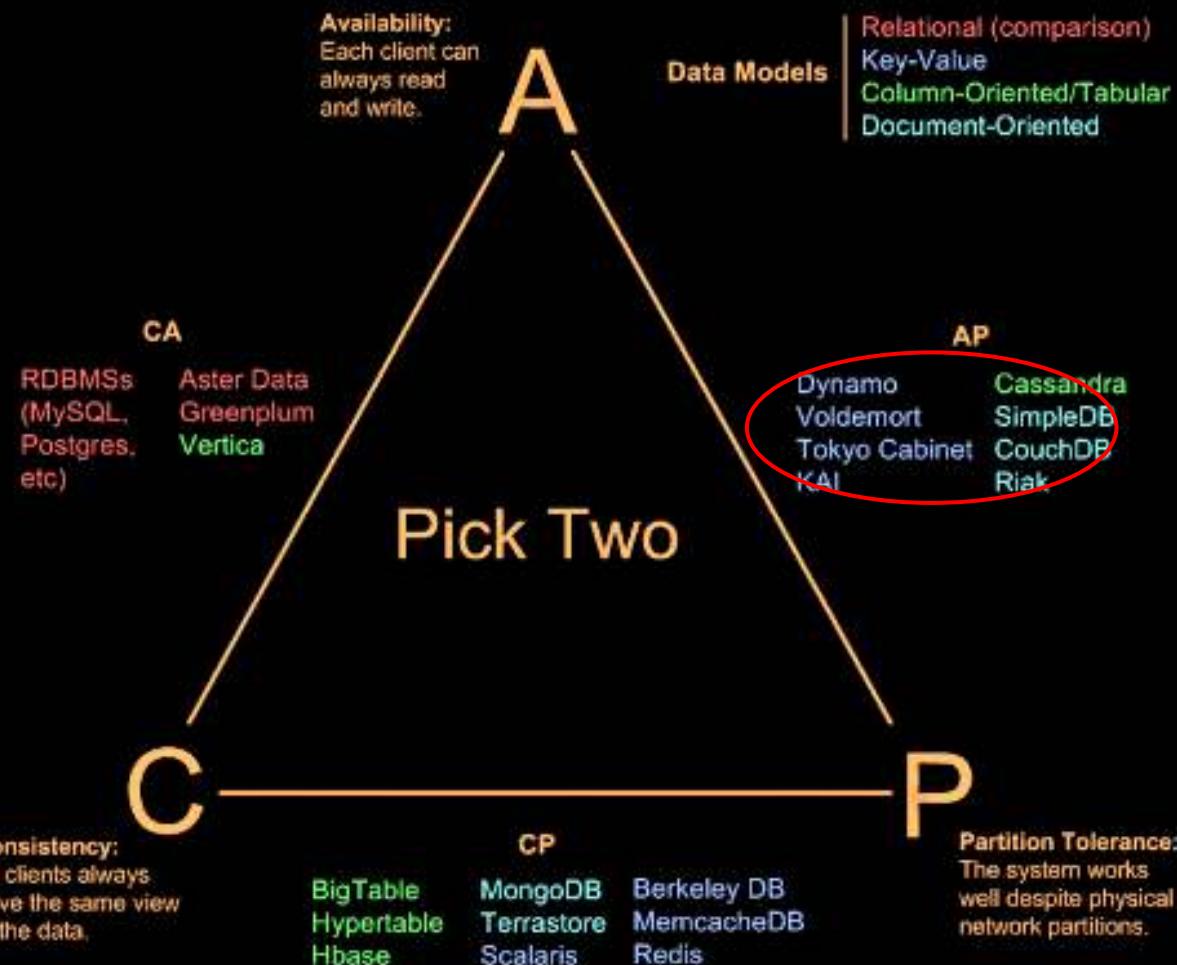
Architecture

Cassandra in short

| Feature | In Cassandra |
|---------------------|--------------------------------------|
| Model | Column-family |
| Architecture | P2P |
| Query language | Supported (CQL) |
| Reference scenarios | Transactional (read/write intensive) |
| Partitioning | Consistent hashing |
| Indexes | Local secondary index |
| Replication | Leader-less |
| Consistency | Eventual consistency |
| Availability | High, tunable |
| Fault tolerance | High (no master, P2P ring) |
| Transactions | No ACID transactions |
| CAP theorem | AP |
| Distributed by | Apache third-parties |

CAP theorem

Visual Guide to NoSQL Systems



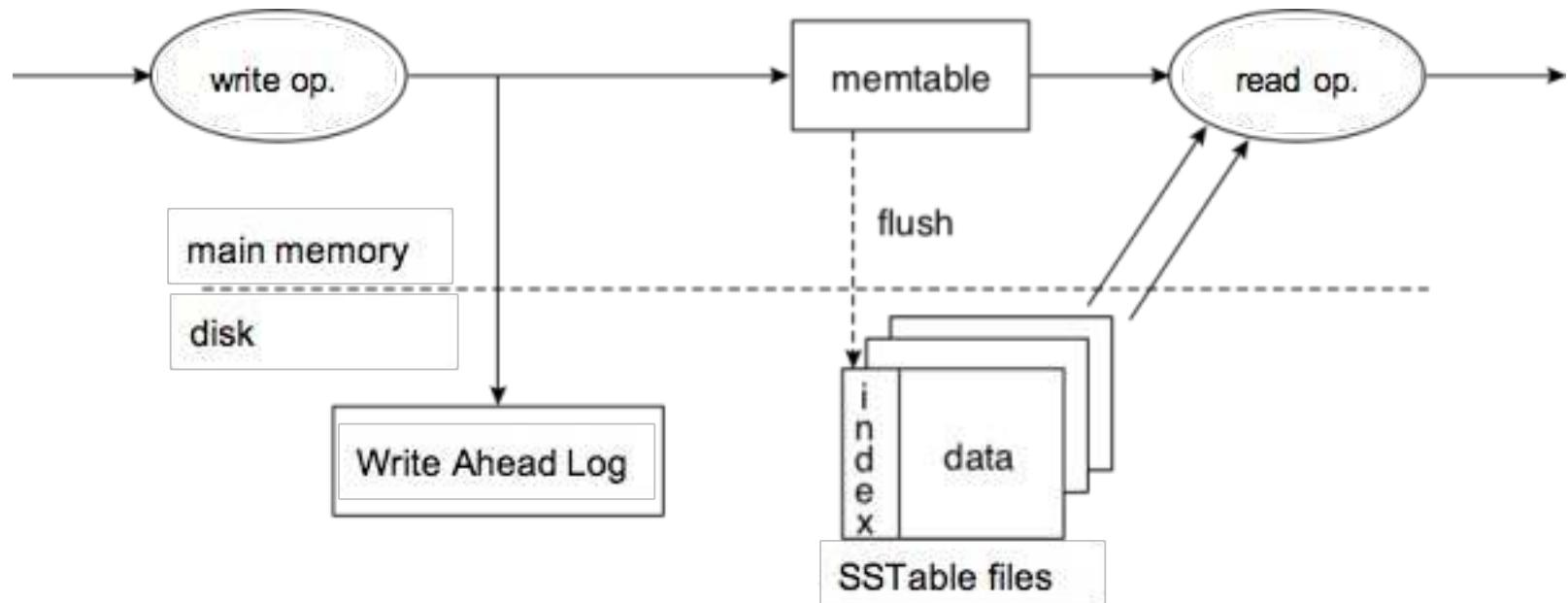
Cassandra

AP: Highly available system

Local Persistence

- Organization of local data store at nodes relies on **local file system**
- Write operations happens in three steps
 - Write to **commit log** in local disk of the node
 - Update **in-memory data structure** (write-back cache of data rows that can be looked up by key)
 - When the in-memory structure is full, it is written (flushed) on disk as an **SSTable**
- Read operation
 - Looks up in-memory data structures first before looking up files on disk

Local persistence



Persistency + durability
but also high throughput of write operations

Configurable read/write consistency

- A **consistency level** can be specified for a given query session or for each read and write query
- It allows to set parameters **w** and **r** in the formula $w + r > n$ (n number of replicas)
- Common consistency levels

ALL – All nodes in the cluster must confirm (even non-local nodes)

QUORUM – A quorum of nodes (half the replication factor plus one) in the cluster must confirm

ONE / TWO / THREE – One, two or three nodes in the cluster must confirm

| WRITE | | |
|--------|--|---|
| Level | Description | Usage |
| ALL | A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition. | Provides the highest consistency and the lowest availability of any other level. |
| QUORUM | A write must be written to the commit log and memtable on a quorum of replica nodes across <i>all</i> datacenters. | Use in single or multiple datacenter clusters to maintain strong consistency across the cluster. Use if you can tolerate some level of failure. |
| ONE | A write must be written to the commit log and memtable of at least one replica node. | Satisfies the needs of most users because consistency requirements are not stringent. |
| TWO | A write must be written to the commit log and memtable of at least two replica nodes. | Similar to ONE. |
| THREE | A write must be written to the commit log and memtable of at least three replica nodes. | Similar to TWO. |

| READ | | |
|--------|--|--|
| Level | Description | Usage |
| ALL | Returns the record after all replicas have responded. The read operation will fail if a replica does not respond. | Provides the highest consistency of all levels and the lowest availability of all levels. |
| QUORUM | Returns the record after a quorum of replicas from <i>all</i> datacenters has responded. | Used in single or multiple datacenter clusters to maintain strong consistency across the cluster. Ensures strong consistency if you can tolerate some level of failure. |
| ONE | Returns a response from the closest replica, as determined by the snitch . By default, a read repair runs in the background to make the other replicas consistent. | Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write. |
| TWO | Returns the most recent data from two of the closest replicas. | Similar to ONE. |
| THREE | Returns the most recent data from three of the closest replicas. | Similar to TWO. |

Transactions

- No ACID transactions with rollback or locking mechanisms
- Cassandra offers Atomic, Isolated, and Durable (AID) transactions with eventual/tunable consistency (no classical consistency, no integrity constraints)

Column-family stores use cases

Suitable use cases



- *Applications with a high number of writes*
- Event Logging
 - Great choice to store event information
- Content Management Systems, Blogging Platforms
 - You can store blog entries with tags, categories, links, and trackbacks in different columns
 - Comments can be either stored in the same row or moved to a different keyspace
 - Similarly, blog users and the actual blogs can be put into different column families
- Counters
 - Often, in Web applications you need to count and categorize visitors of a page to calculate analytics
 - Counter can be maintained in Cassandra



When not to use

- Systems that require ACID transactions for writes and reads
- Prototypes
 - during the early stages query patterns may change and this requires to change the column family design
 - in Cassandra, the cost may be higher for query change as compared to schema change

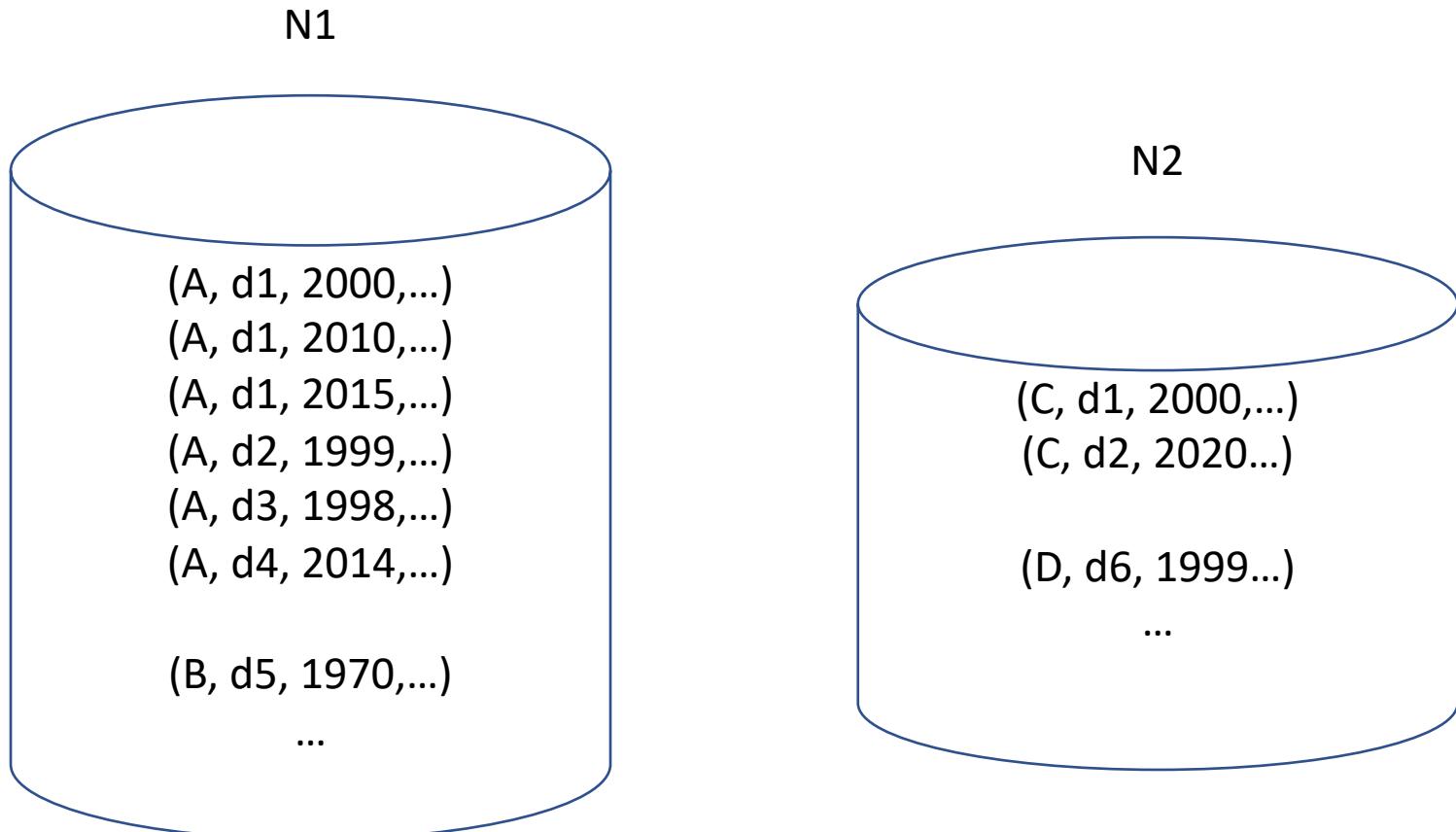
Cassandra, MapReduce, Hadoop

- Integration with Hadoop MapReduce (MapReduce I/O from/to Cassandra)
- Integration with Pig (Pig I/O from/to Cassandra)

Understanding Cassandra queries

Examples

```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director, year) ;
```



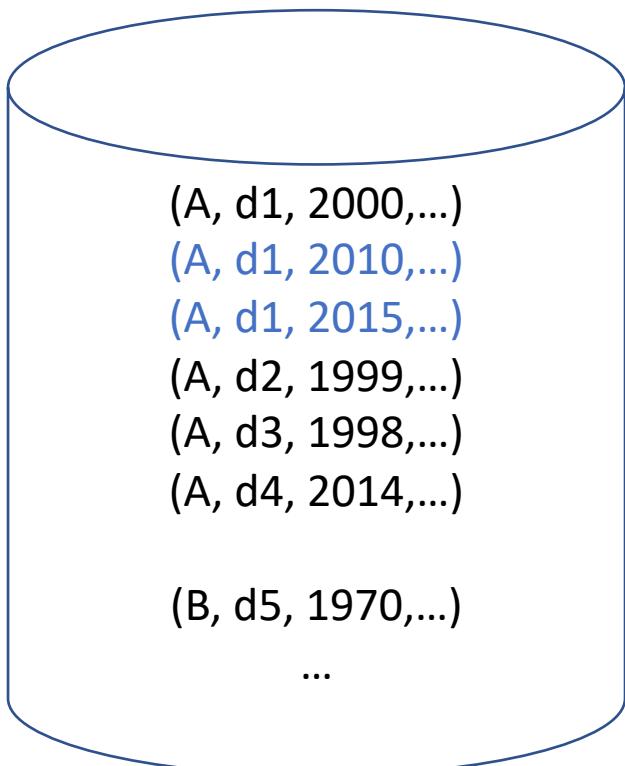
Inside each partition, ordered with respect to the partition key and the clustering column values
(order by title, director, year)

- Queries can be executed if
1. It is possible to locate the nodes storing the rows to be retrieved
 2. In each node, the rows to be retrieved are sequentially stored

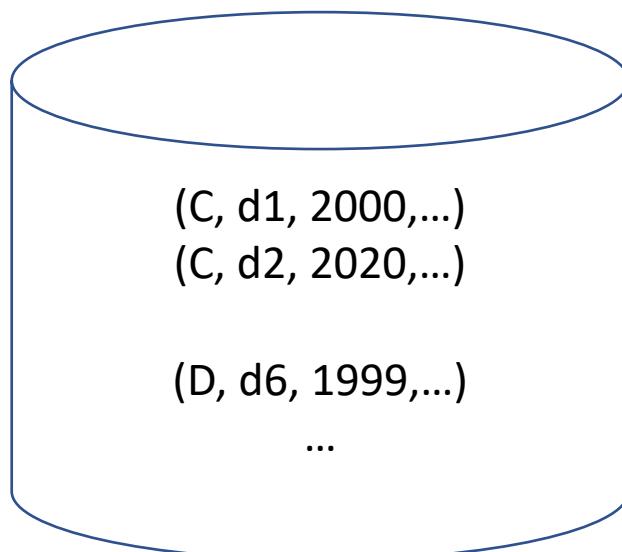
```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director, year);
```

```
SELECT*
FROM movies
WHERE title = 'A' AND
      director= 'd1' AND
      year > 2000
```

N1



N2



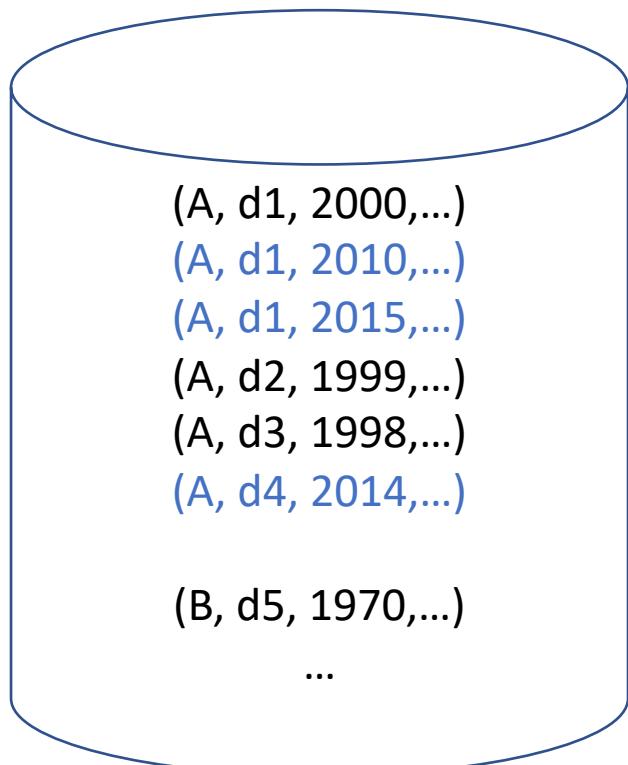
The query is **ADMITTED** because

1. It is possible to locate the nodes storing the rows to be retrieved (N1)
2. In N1, the rows to be retrieved are sequentially stored (two sequential tuples, in blue)

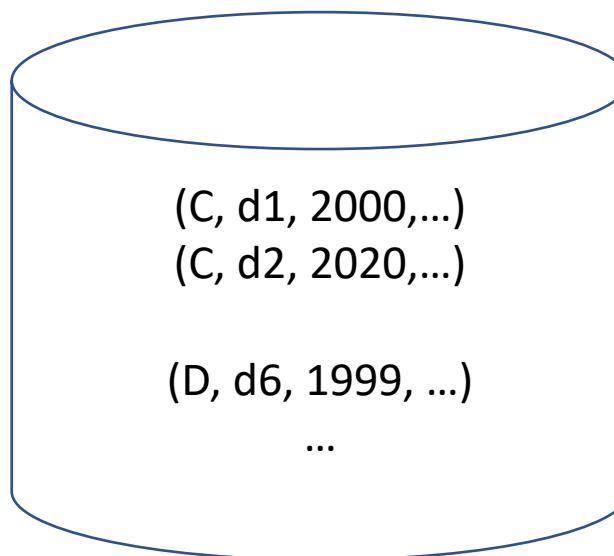
```
CREATE TABLE movies (
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY (title, director, year);
```

```
SELECT*
FROM movies
WHERE title = 'A' AND
      year > 2000
```

N1



N2



The query is **NON ADMITTED** because

1. It is possible to locate the nodes storing the rows to be retrieve (N1)
2. But, in N1 the rows to be retrieved are **NOT** sequentially stored (three tuples, in blue)

Conditions that guarantee the desired behaviour

- The WHERE clause **must** contain an equality-based selection condition on each attribute of the partition key
 - IN clause is also allowed (a set of nodes is identified)
 - The system is able to identify the nodes contributing to the result
- The WHERE clause **can** contain selection conditions over **any prefix of the clustering column list**, as defined in the primary key
 - If more than one clustering column exist, range restrictions are allowed only on the last clustering column being restricted in the WHERE clause
 - On the identified nodes, rows to be returned must be sequentially stored

From the aggregate-oriented
logical schema to Cassandra
logical/physical schema

Input

- The aggregate-oriented logical schema in meta-notation
- The annotated ER diagram
- One NoSQL system S (for today, Cassandra)

Output

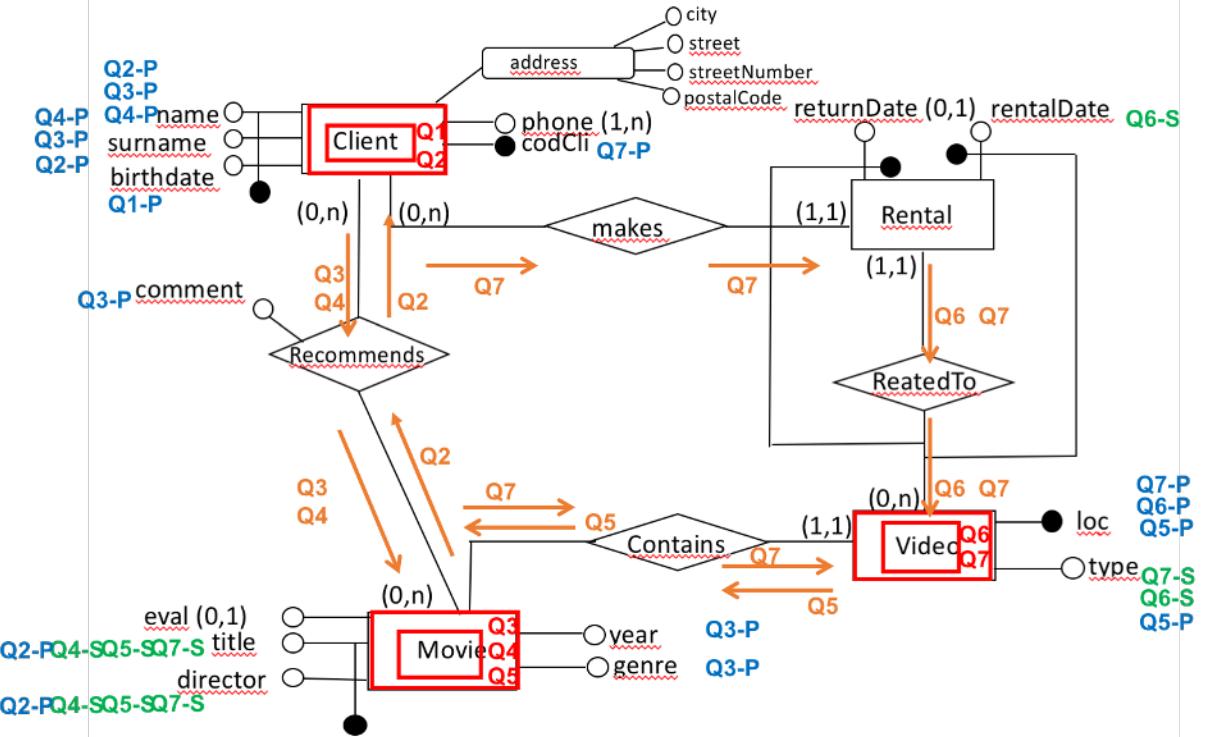
- A schema for S (in metanotation + + partition keys + indexes)

ASSUMPTION: all selection conditions are equality based

Issues

- From the aggregate-oriented logical schemas in meta-notation, the corresponding aggregation entity and the set of associated queries
to a set of collections for S
- Each collection allows one subset of the queries to be executed
- From the set of selection attributes and the identifiers of the aggregation entity to the partition key and indexes

Input



- client: {name, surname, birthdate, recommends: [{title, director}]}
 - Q1, Q2
- movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}
 - Q3, Q4, Q5
- video: {loc, type, rentals: [{rentalDate, codCli}], title, director}
 - Q6, Q7

Design in Cassandra

Aggregation entity Client

client: {name, surname, birthdate, recommends: [{title, director}]}
↳

Queries associated with Client: Q1, Q2

Selection attributes for Q1: {}

Selection attributes for Q2: {}

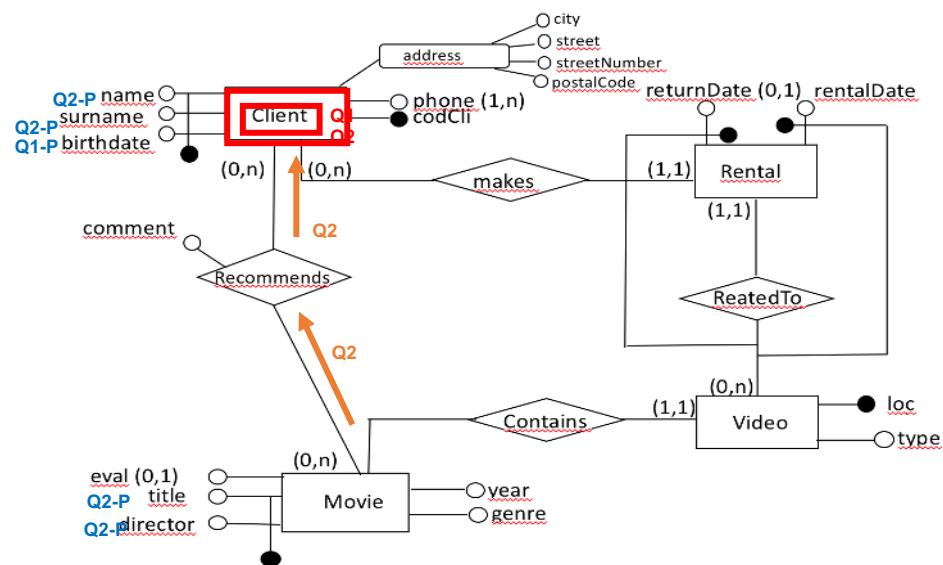
Q1. Average age of clients
Q2. Name and surname of clients and related recommended movies

No selection attribute → no need for a specific partition key

The primary key is taken from the Client identifier (name, surname, birthday)

Partition key = primary key = { name, surname, birthday }

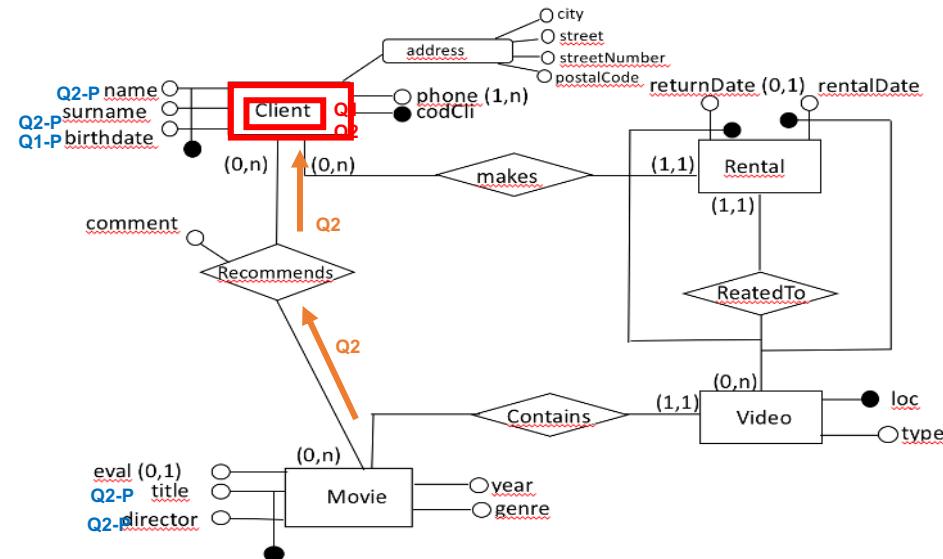
but any other alternative is fine



Aggregation entity Client – option 1 (UDT)

client: {name, surname, birthdate, recommends: [{title, director}]}
 |
 |

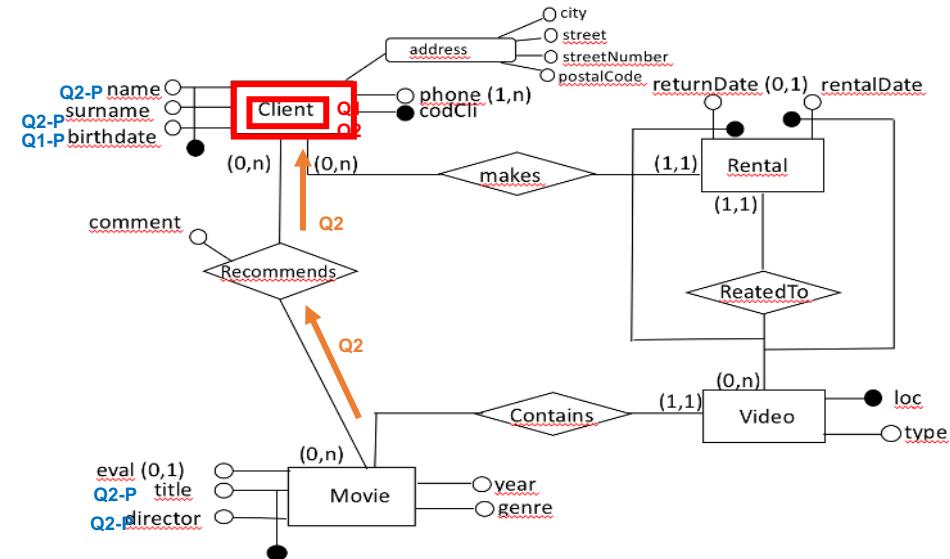
```
CREATE TYPE movie_t(
    title text,
    director text);
```



```
CREATE TABLE Clients (
    name text,
    surname text,
    birthdate date,
    recommends set<frozen<movie_t>>,
    PRIMARY KEY (name, surname, birthdate));
```

Aggregation entity Client – option 2 (collection types)

client: {name, surname, birthdate, recommends: [{title, director}]}
 |
 |



```
CREATE TABLE Clients (
    name text,
    surname text,
    birthdate date,
    recommends set<frozen<map<text, text>>>,
    PRIMARY KEY (name, surname, birthdate));
```

Cassandra design rules

Let A be the aggregate and Q_A the queries associated with A

1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - You can set, e.g., partition key = primary key = aggregate identifier

Aggregation entity Movie

movie: {title, director, year, genre,
recommended_by: [{name, surname,
comment}], contained_in: [{loc, type}]}

Queries associated with Movie: Q3, Q4, Q5

Selection attributes for Q3: {}

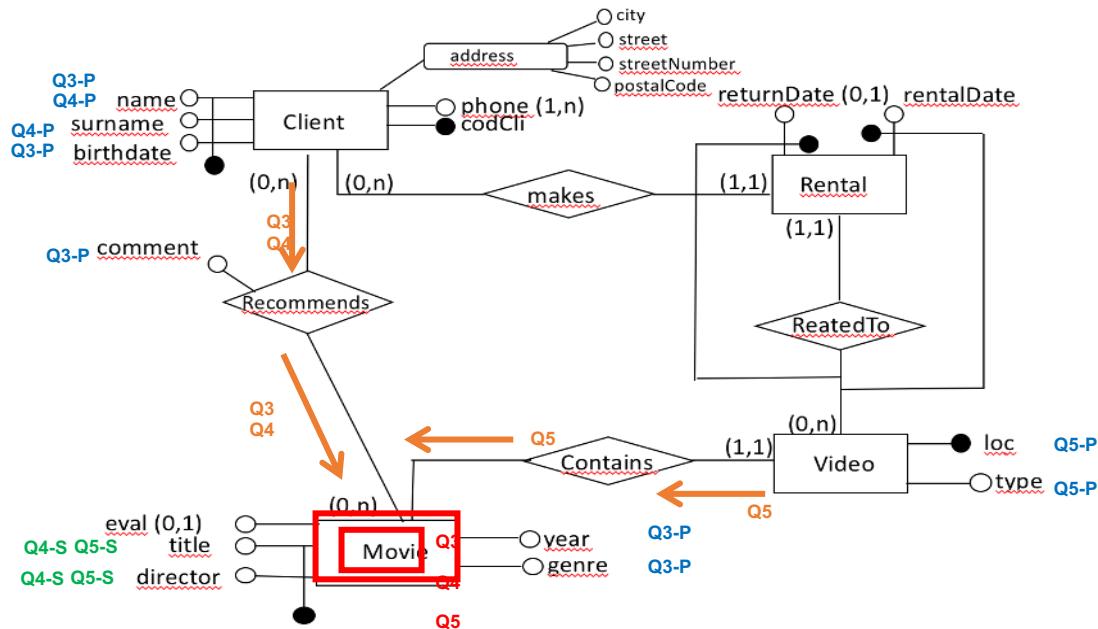
Selection attributes for Q4: { title, director }

Selection attributes for Q5: { title, director }

Title and director must appear
in the primary key to execute
Q4 and Q5
no other attribute is needed

Partition key = primary key =
{ title, director }

Q3. Genre and year of the movies
and their related
recommendations, together with
the name and the surname of the
client who made them
Q4. Name and surname of clients
who recommended the movie 'pulp
fiction' by 'quentin tarantino'
Q5. Given a movie, all information
of videos that contain it

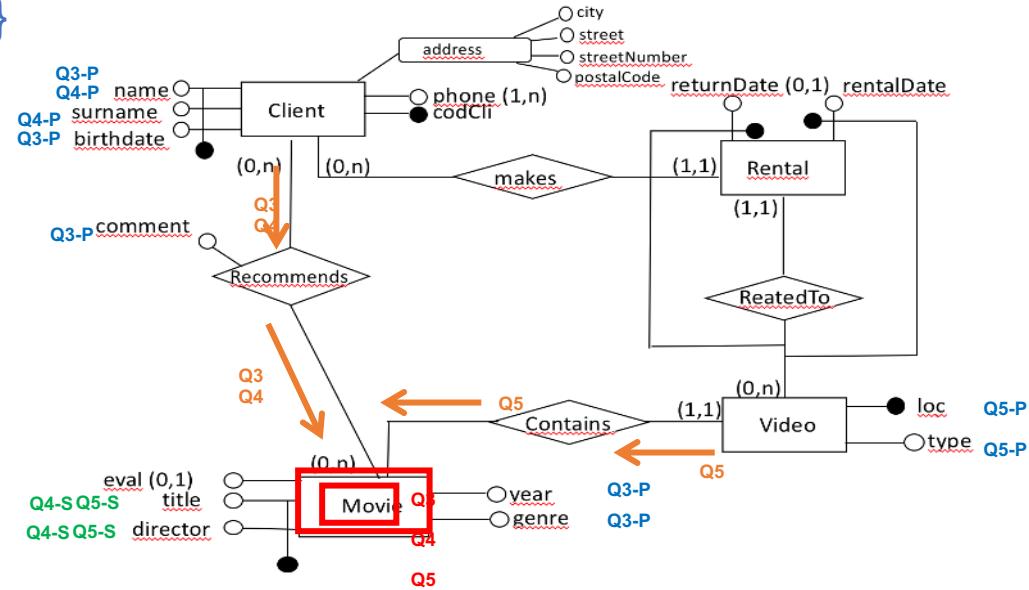


Aggregation entity Movie – option 1 (UDT)

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}

```
CREATE TYPE clientComment_t (
    name text,
    surname text,
    comment text);
```

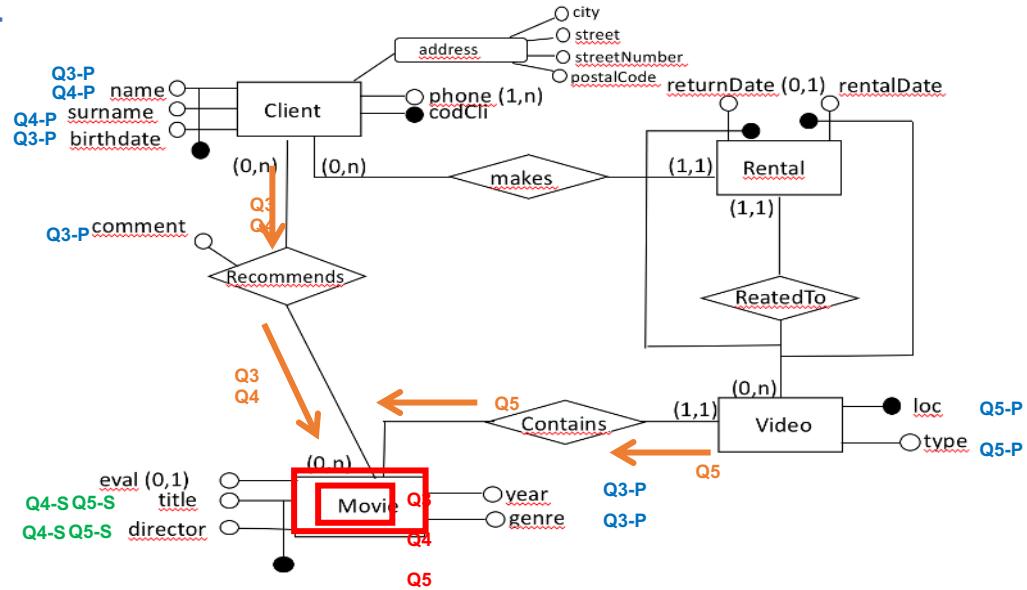
```
CREATE TYPE video_t (
    loc integer,
    type text);
```



```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by set<frozen<clientComment_t>>,
    contained_in set<frozen<video_t>>,
    PRIMARY KEY ((title, director));
```

Aggregation entity Movie – option 2 (collection types)

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}



```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by set<frozen<map<text, text>>>,
    contained_in set<frozen<map<text, text>>>,
    PRIMARY KEY ((title, director));
```

Cassandra design rules

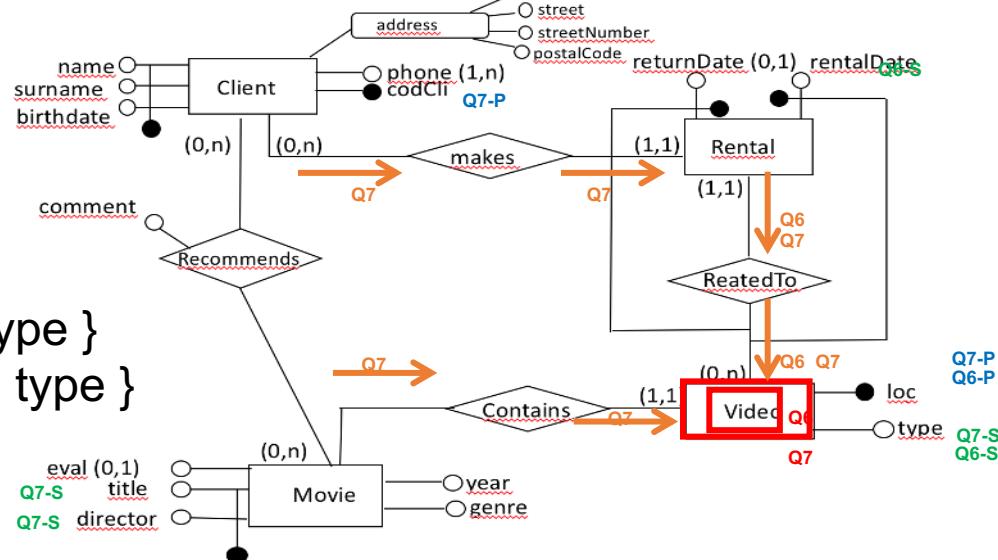
Let A be the aggregate and Q_A the queries associated with A

1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - Partition key = primary key = aggregate identifier
2. If queries in Q_A share the same set of selection attributes or the set of selection attribute is empty
 - The shared set of selection attributes become the partition key
 - If needed, add the aggregate identifier to the partition key to obtain the primary key

Aggregation entity Video

video: {loc, type, rentals: [{rentalDate, codCli}], title, director}

- Q6.** Videos of type 'DVD', rented from a certain date
Q7. The videos of type 'VHS' containing the movie 'pulp fiction' by 'quentin tarantino' and the clients that rented them



Queries associated with Video: Q6, Q7

Selection attributes for Q6: { rentalDate, type }

Selection attributes for Q7: { title, director, type }

rentalDate is a **nested attribute** → it **cannot belong** to the primary key

type appears in both → it becomes the **partition key** (an equality is always specified)

title and **director** become clustering columns

To define the primary key, we need a video identifier → **loc**

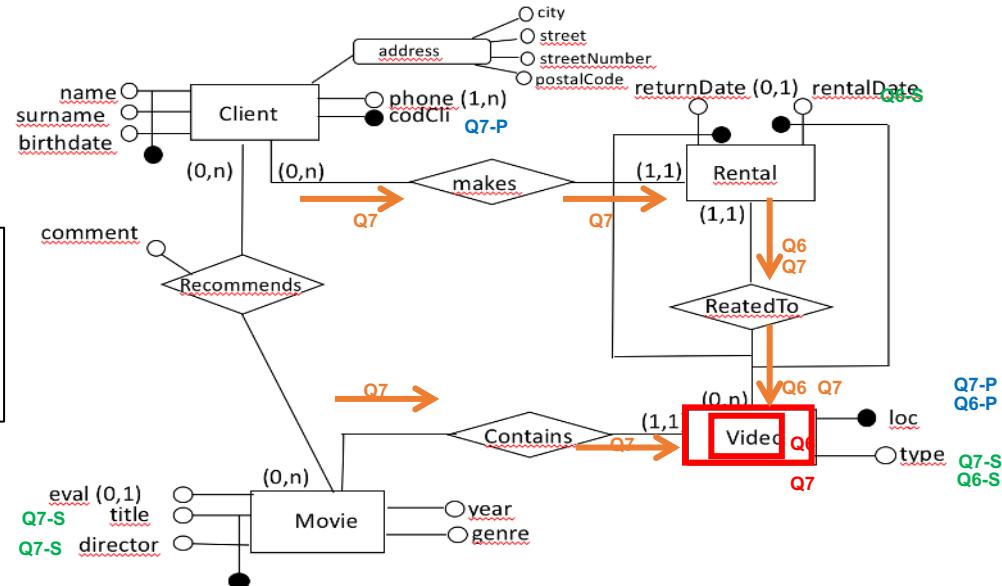
Partition key = { type }

Primary key = { type, title, director, loc }

Aggregation entity Video – option 1 (UDT)

video: {loc, type, rentals: [{rentalDate, codCli}], title, director}

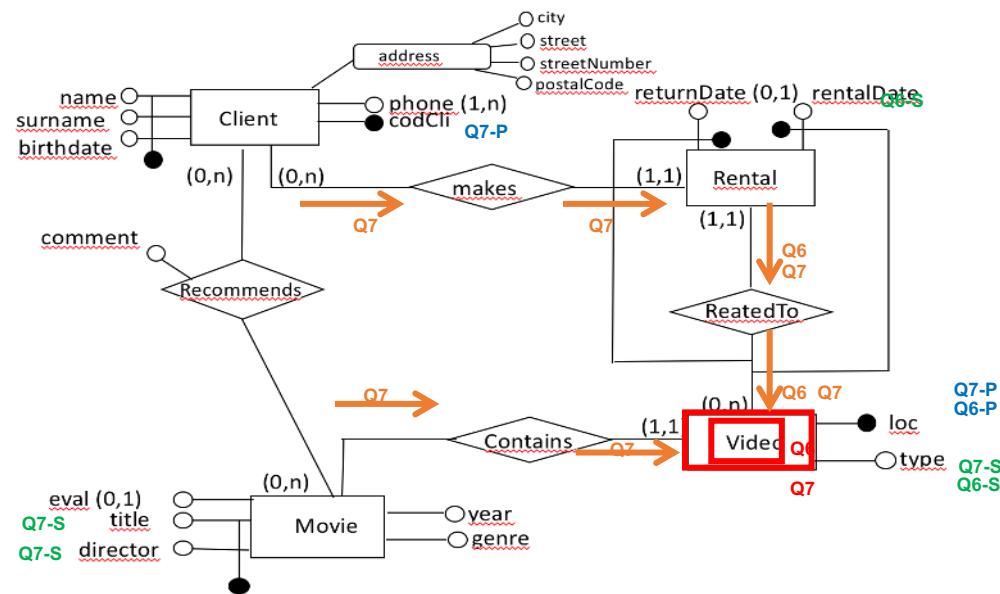
```
CREATE TYPE clientRentals_t (
    rentalDate date,
    codCli int);
```



```
CREATE TABLE Videos (
    loc int,
    type text,
    rentals set<frozen<clientRentals_t>>,
    title text,
    director text,
    PRIMARY KEY (type, title, director, loc));
```

Aggregation entity Video – option 2 (collection types)

video: {loc, type, rentals: [{rentalDate, codCli}], title, director}



```
CREATE TABLE Videos (
    loc int,
    type text,
    rentals set<frozen<map<text, text>>>,
    title text,
    director text,
    PRIMARY KEY (type, title, director, loc));
```

Aggregation entity Video: problem!

```
CREATE TYPE clientRentals_t(  
    rentalDate date,  
    codCli int);
```

In both cases,
conditions on rentalDate
cannot be specified, neither
creating some index or specifying the ALLOW FILTERING clause!

```
CREATE TABLE Videos (  
    loc int,  
    type text,  
    rentals set<frozen<clientRentals_t>>,  
    title text,  
    director text,  
    PRIMARY KEY (type, title, director, loc));
```

```
CREATE TABLE Videos (  
    loc int,  
    type text,  
    rentals set<frozen<map<text, text>>>,  
    title text,  
    director text,  
    PRIMARY KEY (type, title, director, loc));
```

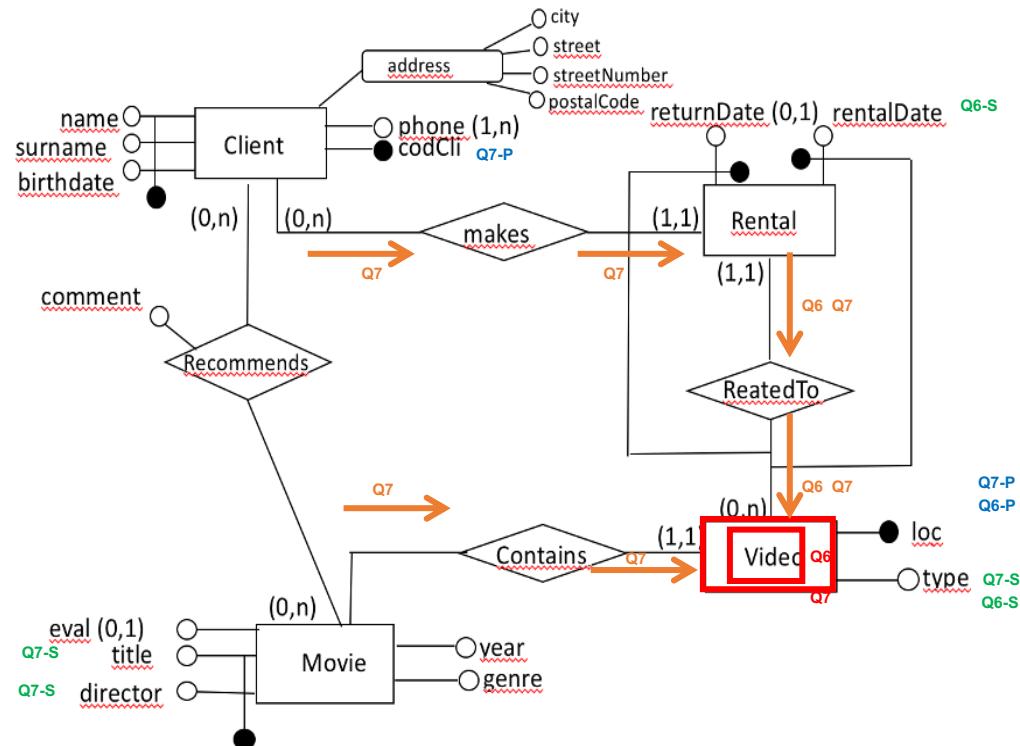
Back to the selection of the aggreg. entity

- The problem is due to the selection of the aggregation entity during the logical design phase

Q6 (**Video**, [Video(type)_!, Rental(rentalDate)_Rt], [Video(loc)_!])

Q7 (**Video**, [Video(type)_!, Movie(title, director)_C] [Video(loc)_!, Client(CodCli)_Mrt]))

- Q6: the aggregation entity is an entity from the **(0,n)** side of an association, this leads to a selection condition on a nested attribute



Back to the selection of the aggreg. entity

- Suppose we change the Q6 aggregation entity

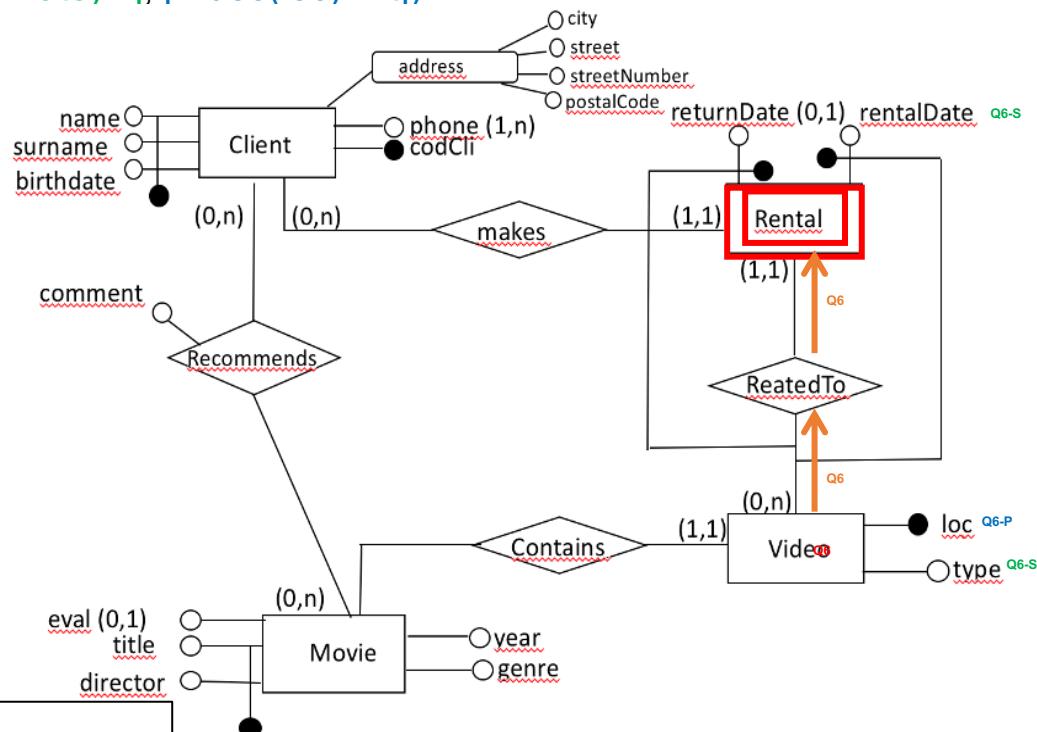
Q6 (**Video**, [Video(type)_!, Rental(rentalDate)_Rt], [Video(loc)_!])



Q6 (**Rental**, [Video(type)_Rt, Rental(rentalDate) !], [Video(loc) Rt])

- Q6: now the aggregation entity is an entity from the **(1,1)** side of an association

rental: {rentalDate, loc, type}



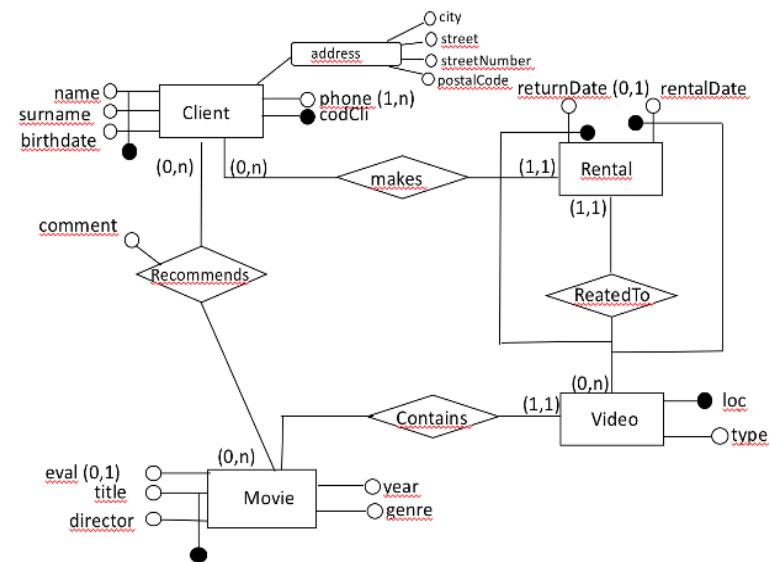
```
CREATE TABLE Rentals (
    rentalDate date,
    loc int,
    type text,
    PRIMARY KEY (rentalDate, type, loc));
```

Cassandra design rules

Let A be the aggregate and Q_A the queries associated with A

1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - Partition key = primary key = aggregate identifier
2. If queries in Q_A share the same set of selection attributes or the set of selection attribute is empty
 - The shared set of selection attributes become the partition key
 - If needed, add the aggregate identifier to the partition key to obtain the primary key
3. If one selection attribute is nested, it cannot be included in the primary key → the selection might not be allowed
 - During the aggregate design, favour aggregation entities from the one-side of one-to-many associations

Does it always work?



Q6-S

Determine the surname of clients with name «John» that recommended one film produced in 1997

Q8(**Client**, [Client(name)_!, Movie(year)_R], [Client(surname)_!])



Q6-P

Q6-S

OR

Q8(**Movie**, [Client(name)_R, Movie(year)_!], [Client(surname)_R])

Does it always work?

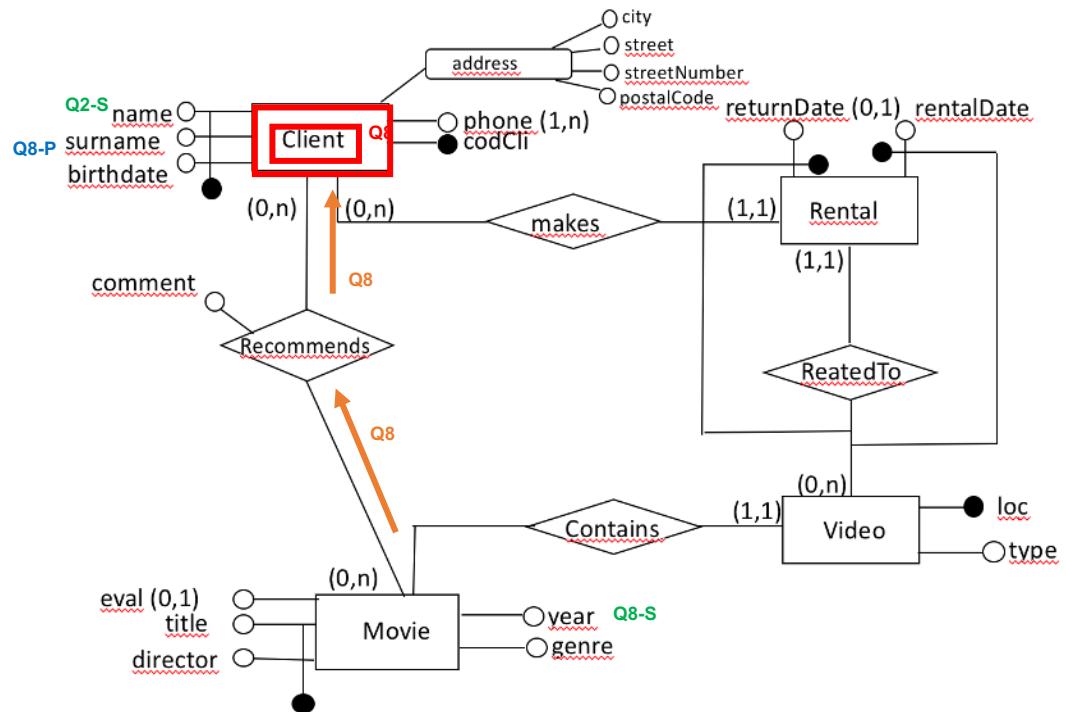
Determine the surname of clients with name «John» that recommended one film produced in 1997

Q8(**Client**, [Client(name)_!, Movie(year)_R], [Client(surname)_!])

client: {codcli, name, surname, recommends: [{year}] }

```
CREATE TABLE Clients (
codCli int,
name text,
surname text,
recommends set<text>,
PRIMARY KEY (name, codCli));
```

By creating an index on the set,
the query can be specified



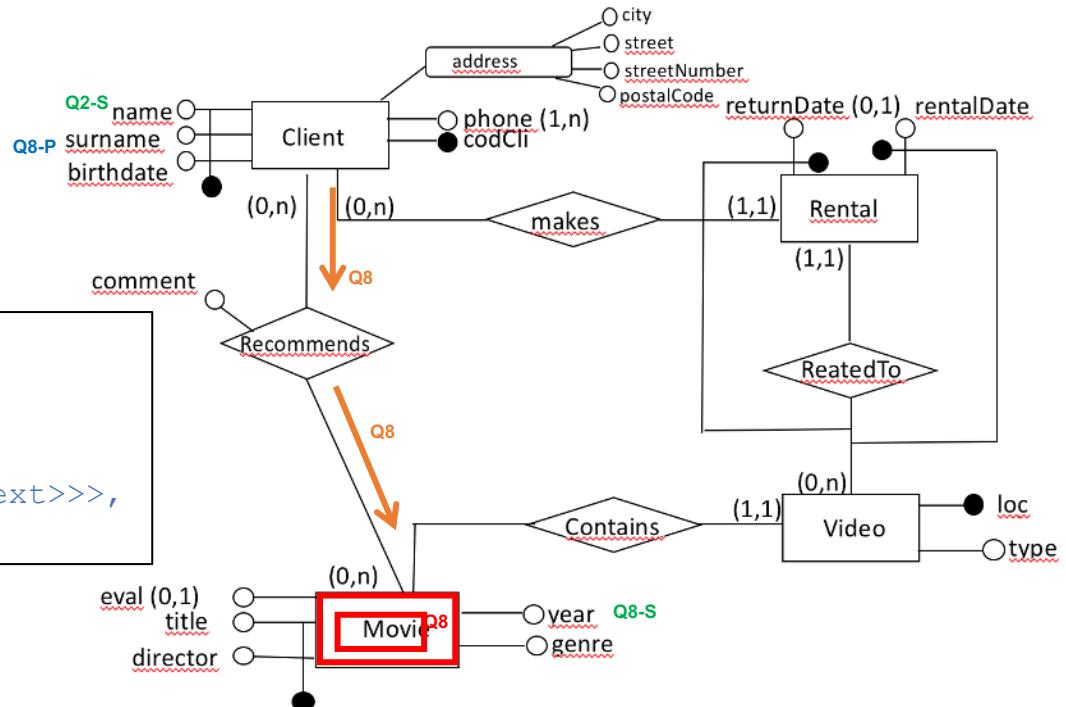
Does it always work?

Determine the surname of clients with name «John» that recommended one film produced in 1997

Q8(Movie, [Client(name)_R, Movie(year)_!], [Client(surname)_R])

movie: {title, director, year, recommended_by: [{name, surname}] }

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    recommended_by set<frozen<map<text, text>>>,
    PRIMARY KEY (year, title, director));
```



The query cannot be specified

Cassandra design rules

Let A be the aggregate and Q_A the queries associated with A

1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - Partition key = primary key = aggregate identifier
2. If queries in Q_A share the same set of selection attributes or the set of selection attribute is empty
 - The shared set of selection attributes become the partition key
 - If needed, add the aggregate identifier to the partition key to obtain the primary key
3. If one selection attribute is nested, it cannot be included in the primary key → the selection might not be allowed
 - During the aggregate design, favour aggregation entities from the one-side of one-to-many associations
4. Indexes allow the selection of atomic values inside sets/lists/maps
 - During the aggregate design, favour nesting of single attributes

Does it always work?

Determine the surname of clients with name «John» that recommended one film produced in 1997, **together with the genre of the film**

Q8(**Client**, [Client(name)_!, Movie(year)_R], [Client(surname)_!, Movie(genre)_R])

OR

Q8(**Movie**, [Client(name)_R, Movie(year)_!], [Client(surname)_R, Movie(genre)_!])

Does it always work?

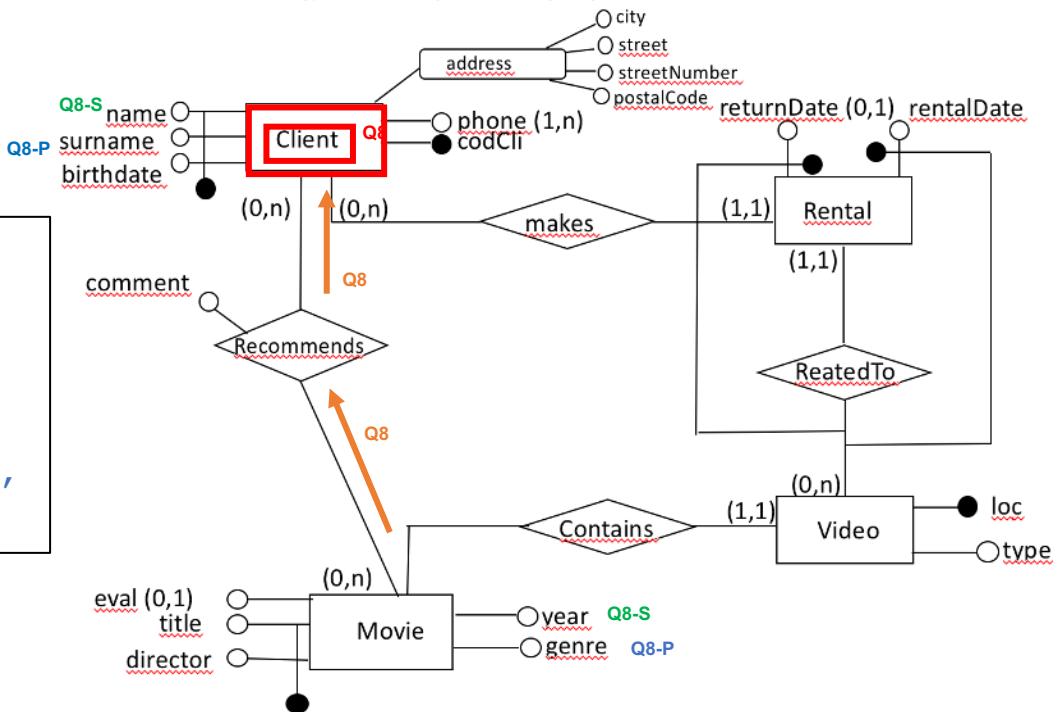
Determine the surname of clients with name «John» that recommended one film produced in 1997, together with the genre of the film

Q8(Client, [Client(name)_!, Movie(year)_R], [Client(surname)_!, Movie(genre)_R])

client: {codCli, name, surname, recommends: [{year, genre}] }

```
CREATE TABLE Clients (
    codCli int,
    name text,
    surname text,
    recommends
        set<frozen<map<text, text>>>,
    PRIMARY KEY (name, codCli));
```

The query cannot be specified
(condition on a subcomponent of
recommends, frozen)!



Does it always work?

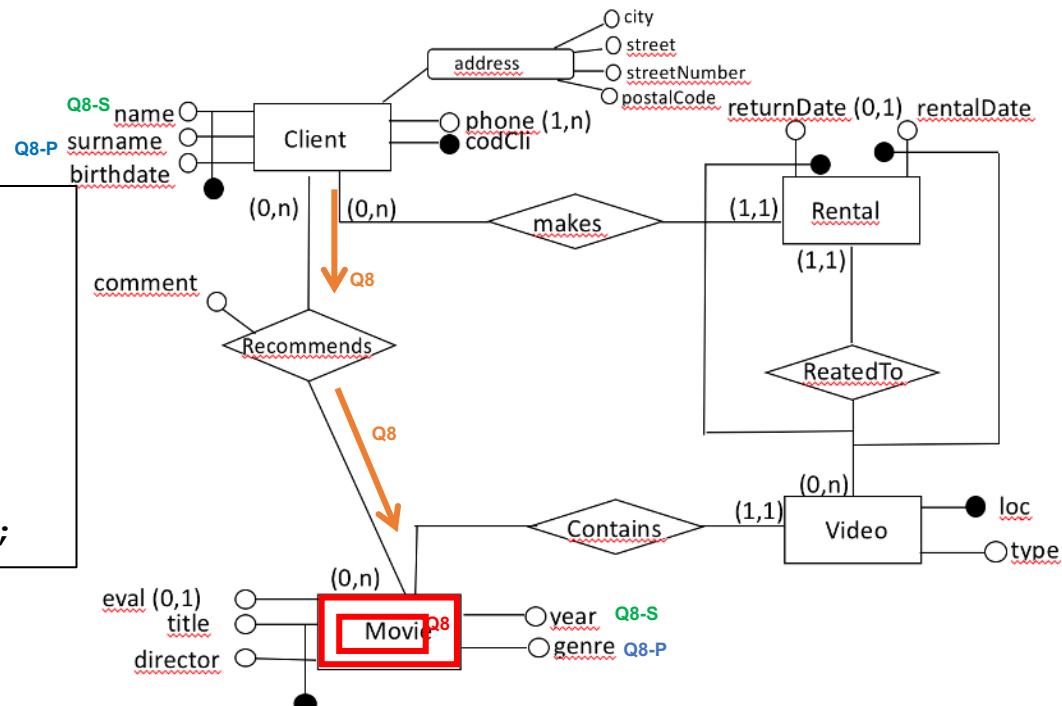
Determine the surname of clients with name «John» that recommended one film produced in 1997 together with the genre of the film

Q8(Movie, [Client(name)_R, Movie(year)_!], [Client(surname)_R])

movie: {title, director, year, recommended_by: [{name, surname}] }

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by
        set<frozen<map<text, text>>>,
    PRIMARY KEY (year, title, director) );
```

The query cannot be specified
(condition on a subcomponent of
recommended_by, frozen)!



Does it work?

- The only possible solution in this case is to change the design approach and create a table corresponding to the many-to-many association

recommendation: {codCli, title, director, name,
surname, year, genre}

```
CREATE TABLE Recommendation (
    codCli int,
    name text,
    surname text,
    title text,
    director text,
    year int,
    genre text,
    PRIMARY KEY ((name, year), codCli, title, director);
```

Cassandra design rules

Let A be the aggregate and Q_A the queries associated with A

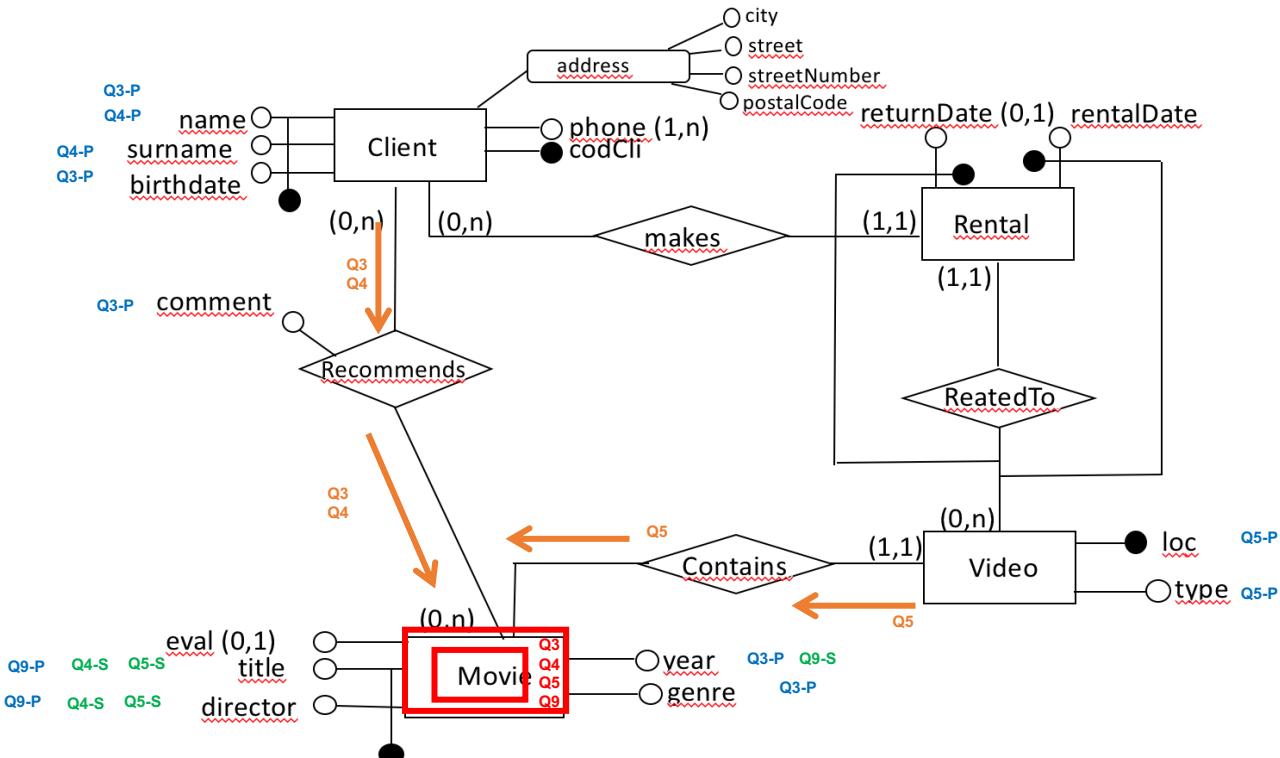
1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - Partition key = primary key = aggregate identifier
2. If queries in Q_A share the same set of selection attributes or the set of selection attribute is empty
 - The shared set of selection attributes become the partition key
 - If needed, add the aggregate identifier to the partition key to obtain the primary key
3. If one selection attribute is nested, it cannot be included in the primary key → the selection might not be allowed
 - During the aggregate design, favour aggregation entities from the one-side of one-to-many associations
4. Indexes allow the selection of atomic values inside sets/lists/maps
 - During the aggregate design, favour nesting of single attributes
5. Sometimes, in presence of many-to-many associations, during the aggregate design, new aggregates corresponding to the association and not to one entity should be taken into account to avoid selections over nested attributes

Back to Movie: new queries, more than one Cassandra table

Add a new query in the workload: determine the film produced in 1997

Q9 = (Movie, [Movie(year)_!], [Movie(title, director)_!])

The aggregate does not change with respect to the previous workload, however ...



movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}

Back to Movie: new queries, more than one Cassandra table

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}], contained_in: [{loc, type}]}

Queries associated with Movie: Q3, Q4, Q5, **Q9**

Selection attributes for Q3: {}

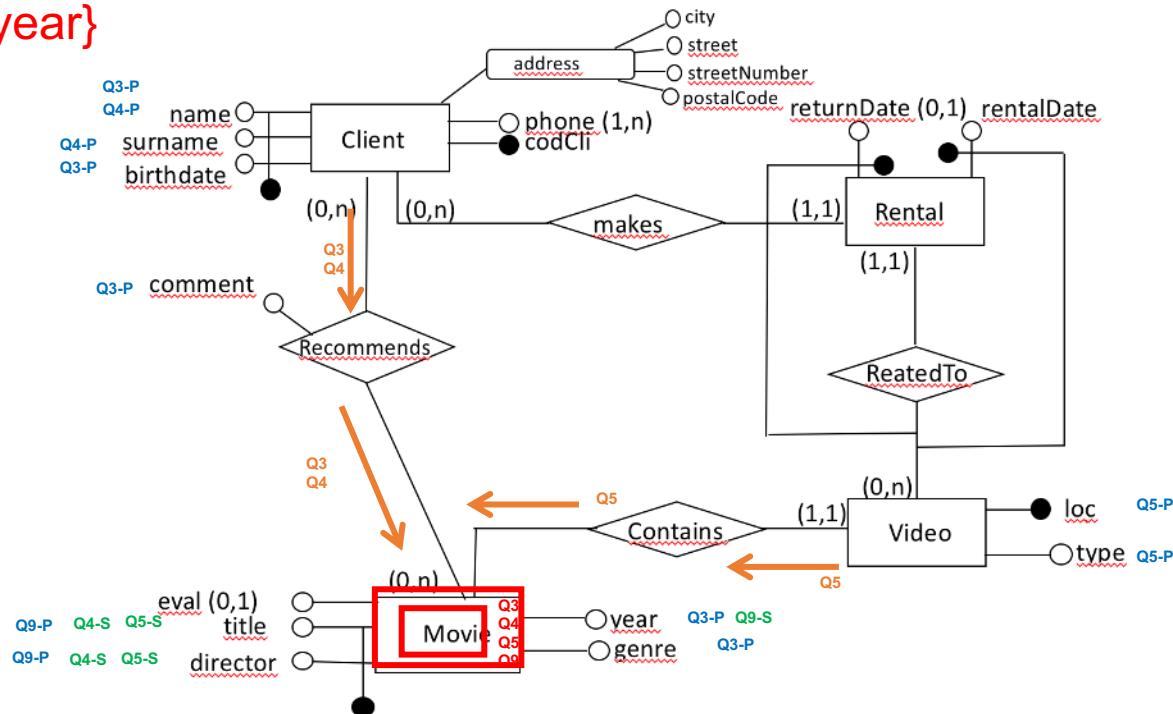
Selection attributes for Q4: { title, director }

Selection attributes for Q5: { title, director }

Selection attributes for Q9: { year }

Selection attributes of the queries are **disjoint**

How to define the partition key and the primary key?



Back to Movie: new queries, more than one Cassandra table

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{loc, type}]}

Queries associated with Movie: Q3, Q4, Q5, **Q9**

Selection attributes for Q3: {}

Selection attributes for Q4: { title, director }

Selection attributes for Q5: { title, director }

Selection attributes for Q9: { year}

PRIMARY KEY ((title, director), year) only Q3, Q4, Q5 admitted

PRIMARY KEY (year, title, director) only Q3, Q9 admitted

The only solution is to split the aggregate into two column-families

Back to Movie: new queries, more than one Cassandra table

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{loc, type}]}

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by set<frozen<map<text, text>>>,
    contained_in set<frozen<map<text, text>>>,
    PRIMARY KEY ((title, director), year);
```

For queries Q3, Q4, Q5

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by set<frozen<map<text, text>>>,
    contained_in set<frozen<map<text, text>>>,
    PRIMARY KEY (year, title, director);
```

For queries Q3, Q9

Back to Movie: new queries, more than one Cassandra table

movie: {title, director, year, genre, recommended_by: [{name, surname, comment}],
contained_in: [{loc, type}]}

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    genre text,
    recommended_by set<frozen<map<text, text>>>,
    contained_in set<frozen<map<text, text>>>,
    PRIMARY KEY ((title, director), year);
```

For queries Q3, Q4, Q5

For queries Q9

```
CREATE TABLE Movies (
    title text,
    director text,
    year int,
    PRIMARY KEY (year, title, director);
```

Cassandra design rules

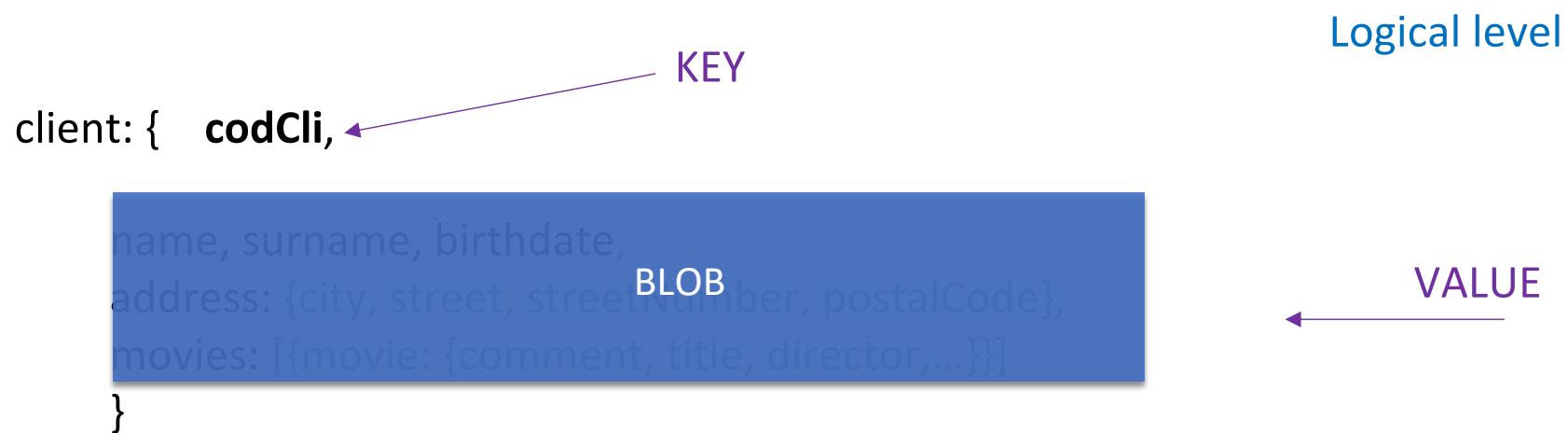
Let A be the aggregate and Q_A the queries associated with A

1. If the set of selection attributes of Q_A is empty, the partition key no matters
 - Partition key = primary key = aggregate identifier
2. If queries in Q_A share the same set of selection attributes or the set of selection attribute is empty
 - The shared set of selection attributes become the partition key
 - If needed, add the aggregate identifier to the partition key to obtain the primary key
3. If one selection attribute is nested, it cannot be included in the primary key → the selection might not be allowed
 - During the aggregate design, favour aggregation entities from the one-side of one-to-many associations
4. Indexes allow the selection of atomic values inside sets/lists/maps
 - During the aggregate design, favour nesting of single attributes
5. Sometimes, in presence of many-to-many associations, during the aggregate design, new aggregates corresponding to the association and not to one entity should be taken into account to avoid selections over nested attributes
6. If the non-empty set of selection attributes for queries in Q_A are disjoint, one aggregate has to be mapped in two or more Cassandra tables.

Key-value NoSQL data stores

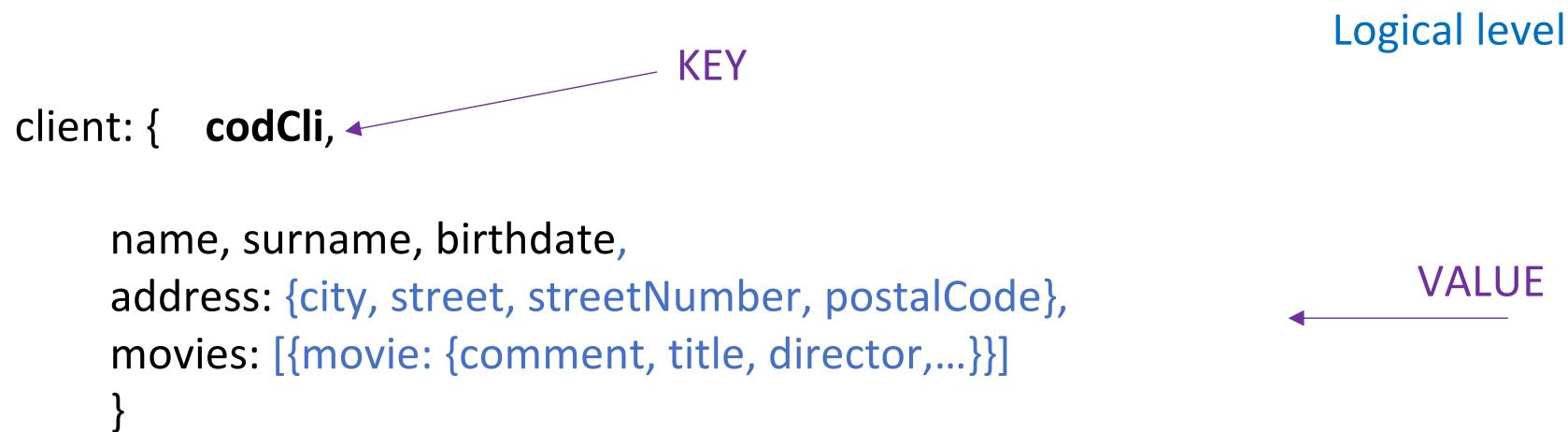
Key-value data model at a glance

- Each data instance is represented in the form **(key, value)**
 - key is an identifier
 - value is the aggregate
- The aggregate **is totally opaque at the logical level**
 - just a big blob of mostly meaningless bit, that the system just stores
 - values do not have a known structure
 - very high flexibility: arbitrary aggregate content
- The aggregate is **visible at the application level**
- Data instances can be grouped into logical collections



Key-value data model at a glance

- Each data instance is represented in the form **(key, value)**
 - key is an identifier
 - value is the aggregate
- The aggregate **is totally opaque at the logical level**
 - just a big blob of mostly meaningless bit, that the system just stores
 - values do not have a known structure
 - very high flexibility: arbitrary aggregate content
- The aggregate is **visible at the application level**
- Data instances can be grouped into logical collections



Instance structure

- No schema information
- No nested values are visible at the NoSQL system level but only at the application level

Collections

- Pairs can be grouped into logical **collections/namespaces**: sets of aggregates, i.e., sets of (key, value) pairs

| key | value |
|-----|-------|
| key | value |
| key | value |
| key | value |

In RDBMS world: A table with two columns:
ID **column** (primary key)
DATA **column** storing the value
(unstructured BLOB)

A

| | |
|---|----|
| 1 | V1 |
| 2 | V2 |
| 3 | V3 |
| 4 | V4 |

(see buckets in Riak)

```
"type": "dvd",
"rentals": [{"rental": {"rentalDate": "15/10/2021",
  "codCli": 375657}}],
"title": "pulp fiction",
"director": "quentin tarantino"}]

...
{"name": "John",
"surname": "Black",
"birthdate": "15/10/2000",
"address": {"city": "Genoa", "street": "Via XX Settembre",
  "streetNumber": 15, "postalCode": 16100},
"movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}},
  {"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}]}
```

Collections and schema information

- The only «schema» information the system is aware of is the collection
- There is no structure specification nor schema constraint in the data model
- But at application-level, we can take advantage of some regularities in data, if any

Collections

- Pairs can be grouped into logical **collections/namespaces**: sets of aggregates, i.e., sets of (key, value) pairs

| key | value |
|-----|-------|
| key | value |
| key | value |
| key | value |

Videos

| | |
|------|-------|
| 1234 | value |
| 5678 | value |
| 9999 | value |
| 0000 | value |

(see buckets in Riak)

Clients

| | |
|--------|-------|
| 375657 | value |
| 375658 | value |
| 375659 | value |
| 375660 | value |

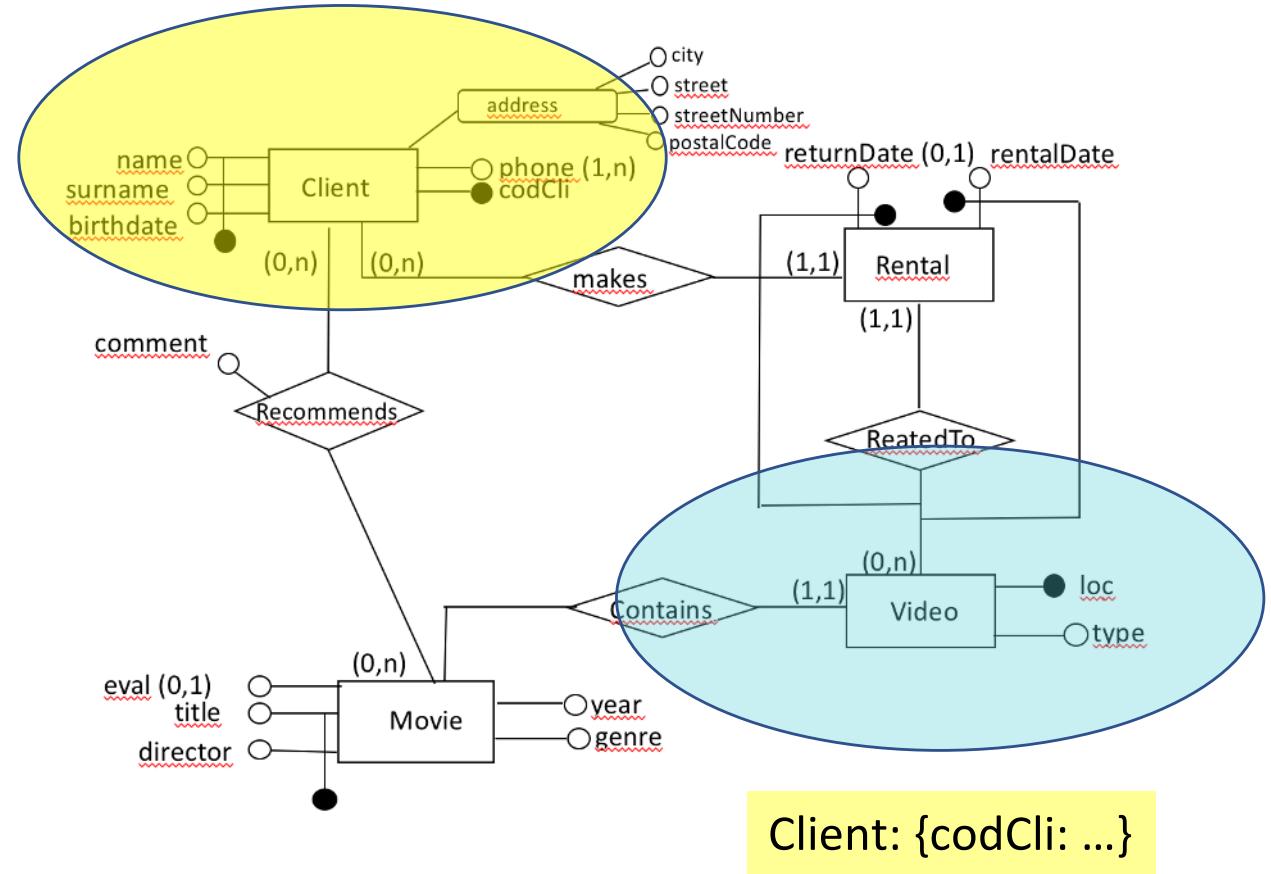
Keys

- (**key**, value) pair
- At the logical level: identification + data retrieval
 - the key **needs** to assume **unique values** in the collection (i.e., it is an **identifier**)
- At the physical level, the key tells the system **how to partition the data** and where to store the data
 - **partition key**
- The partition key and the identifier/primary key coincide
- Aggregates can be directly retrieved **only** by specifying **values** for attributes in the partition key
- *A simple **hash table** (map), primarily used when all accesses to the data are via the **key***

Keys

- How to design the key?
 - Provided by the user (natural unique key): userID, e-mail,...
 - Generated by some algorithm
 - Derived from time-stamps (or other data)
- Use cases with natural keys:
 - user profiles (user ID), ...
 - shopping cart data (**user ID**)
 - web session data (with the **session ID** as the key)
- Expiration of keys
 - After a certain time interval
 - e.g. for caches, session/shopping cart objects,...

Keys



Videos

| | |
|------|-------|
| 1234 | value |
| 5678 | value |
| 9999 | value |
| 0000 | value |

Key is loc

Clients

| | |
|--------|-------|
| 375657 | value |
| 375658 | value |
| 375659 | value |
| 375660 | value |

Key is codeCli

Keys

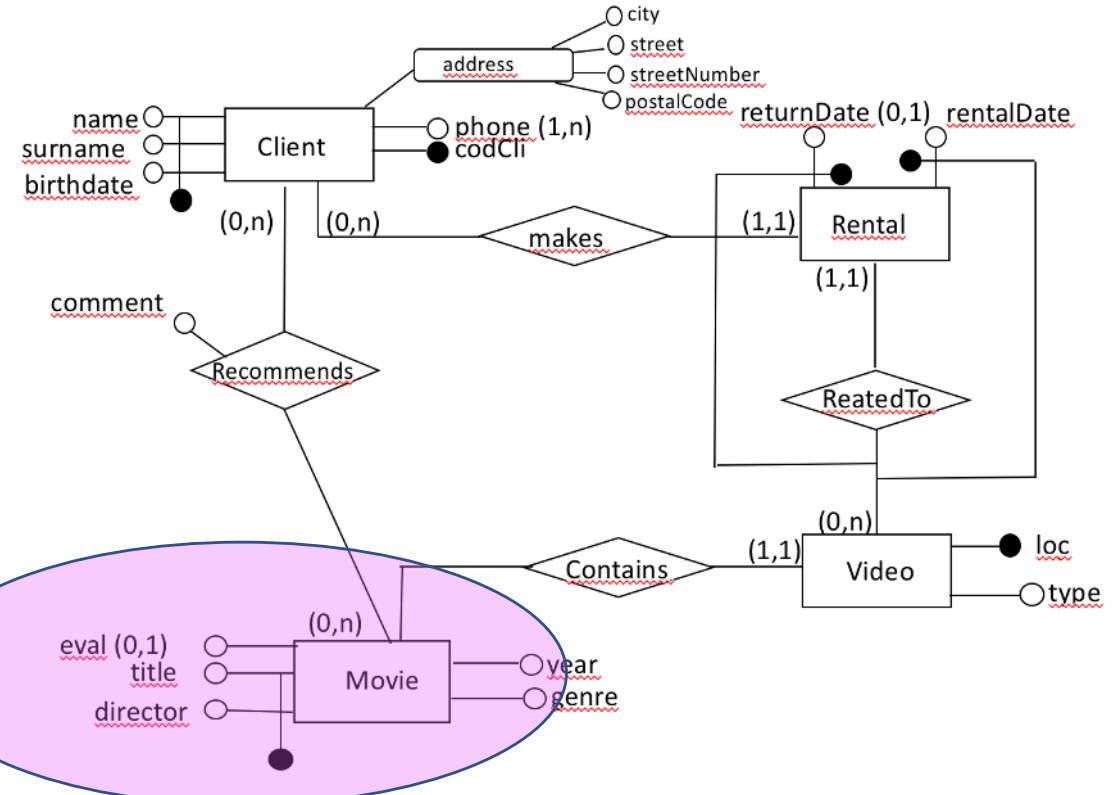
There is no single unique attribute, different options are possible

1. Id
2. (title and director are included in the value)

Movie : {id: ...}

Movies

| | |
|---|-------|
| 1 | value |
| 2 | value |
| 3 | value |
| 4 | value |

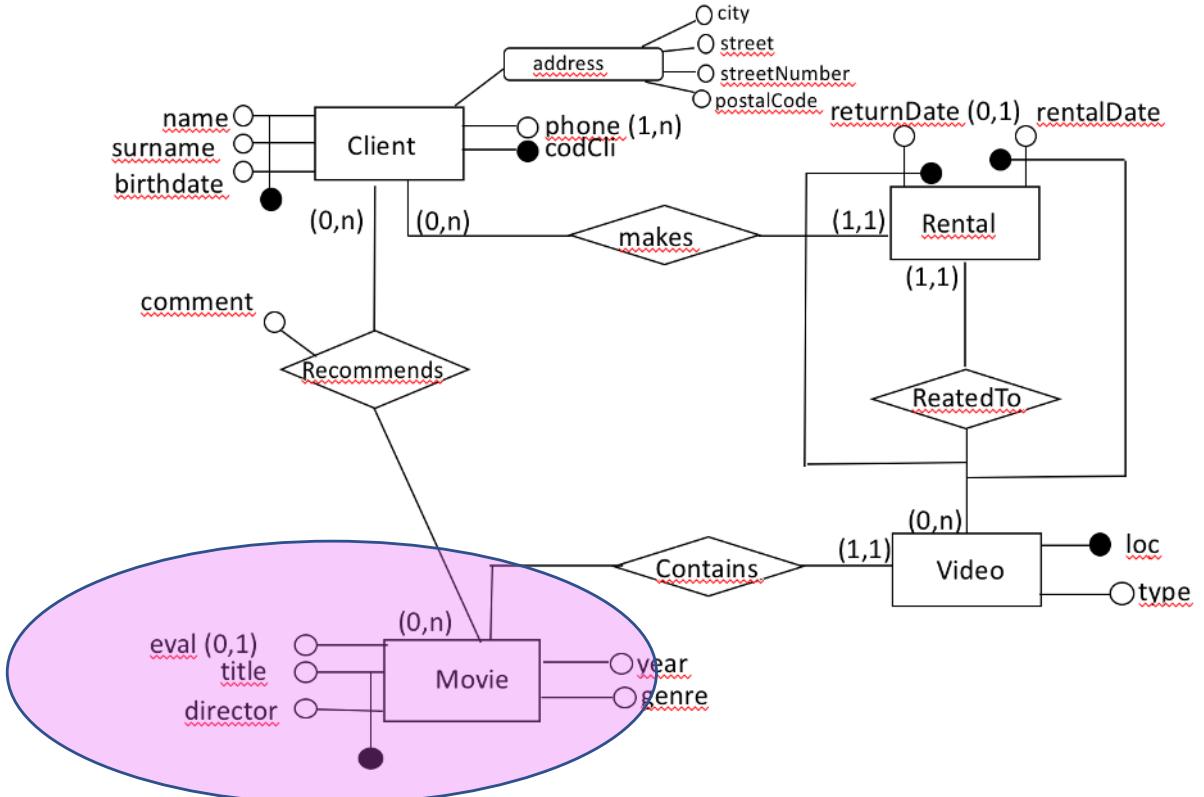


Key is id

Keys

There is no single unique attribute, different options are possible

2. title:director



Movie : {'title:director': ...}

Movies

| | |
|--------------------------------|-------|
| pulp fiction:quentin tarantino | value |
| kill bill:quentin tarantino | value |
| le iene:quentin tarantino | value |
| dumbo : gabriele salvatores | value |

Key is title:director

Identifier vs partition/sharding key

- In the Videos collection, videos are partitioned by loc
- In the Clients collection, clients are partitioned by codCli
- In the Movies collection, clients are partitioned by
 - Id
 - Title:director
- This impacts the way data are stored
- This impacts the way data can be retrieved

Video Rental Example

Logical level

```
client: { codCli,
```

```
    name, surname, birthdate,  
    address: {city, street, streetNumber, postalCode},  
    movies: [ {movie: {comment, id:{title, director},...}} ]
```

```
}
```

REFERENCE

```
movie: { id: {title,director},
```

```
    year, genre, recommended_by: [ {name, surname, comment} ],  
    video: {loc, type} ] }
```

REFERENCE

```
video: {loc,
```

```
    type, rentals: [ {rental: {BLOB alDate, codCli}} ], title, director } }
```

Video Rental Example

Application level

```
client: { codCli,
```

 name, surname, birthdate,
 address: {city, street, streetNumber, postalCode},
 movies: [{movie: {comment, id:{title, director},...}}]
}

REFERENCE

REFERENCE

```
movie: { id: {title,director},  
        year, genre, recommended_by: [ {name, surname, comment} ],  
        video: {loc, type} } }
```

contained_in: [

```
video: {loc,
```

```
        type, rentals: [ {rental: {rentalDate, codCli}} ], title, director}}
```

Interaction

- Basic operations:
 - Put a value for a key
(if the key already exists
the corresponding value
is overwritten)

`put (key, value)`
 - Get the value for the key

`value := get (key)`
 - Delete a key (and the
corresponding value)

`delete (key)`

Interaction

- **Lookup** based on the **key**
- The application can read the entire aggregate by using the key
- Queries with respect to specific aggregate fields are not supported (in general): we need to read the whole aggregate and check **query conditions at the application level**
- **Associations navigated by sequence of lookups**
- In case collections are supported, key values are preceded by the collection name in each operation

Example 1 – Find the title of the movie in a video

```
{"loc": 1234,  
  "type": "dvd",  
  "rentals": [{"rental": {"rentalDate": "15/10/2021",  
                  "codCli": 375657}}],  
  "title": "pulp fiction",  
  "director": "quentin tarantino"}
```

get(Videos,1234)

- at the data store level, find Video 1234 in namespace Videos, no detailed information about its content
- at the *application level*, we go inside the aggregate value and we discover that it contains «Pulp Fiction»

Example 2 – Find the videos containing a certain movie

```
{"loc": 1234,  
  "type": "dvd",  
  "rentals": [{"rental": {"rentalDate": "15/10/2021",  
                  "codCli": 375657}}],  
  "title": "pulp fiction",  
  "director": "quentin tarantino"}
```

get(Videos,???)

- We cannot filter on other attributes than the key
- At the data store level, can only get all the videos, and
- At the application level, filter them by titles and director

Example 2 – Find the videos containing a certain movie - we should use the movies collection instead

get(Movies,pulp fiction:quentin tarantino)

- At the data store level, get the value
- At application level, find *contained_in* and get the location

Movies collection

Key: title:director,

Value contains {year, genre,

recommended_by: [{name, surname, comment}],

contained_in: [{video: {loc, type}}]}

REMARK: the relationship is part of the aggregate, a single data access to retrieve all the relevant information (=a single data node)

Example 2 – Find the videos containing a certain movie - we should use the movies collection instead, what if we chose the other key?

Movies collection

Key:MovieID,

Value contains {title, director, year, genre,

recommended_by: [{name, surname, comment}],

contained_in: [{video: {loc, type}}]}

get(Movies,???)

- We cannot filter on other attributes than the key
- At the data store level, can only get all the movies, and
- At the application level, filter them by titles and director

REMARK: still likely more efficient than starting from Videos, since Movies are less than videos and all the videos containing a certain movie are stored inside the same aggregate

Example 3 - Relationships

Find the age(s) of customer(s) that rented a given video

```
{"loc": 1234,
```

```
  "type": "dvd",
  "rentals": [{"rental": {"rentalDate": "15/10/2021",
    "codCli": 375657}}],
  "title": "pulp fiction",
  "director": "quentin tarantino"}
```

```
{
  "codcli": 375657,
  "name": "John",
  "surname": "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
    "streetNumber": 15, "postalCode": 16100},
  "movies": [{"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}},
    {"movie": {"comment": "very nice", "title": "pulp fiction",
    "director": "quentin tarantino"}]}
}
```

get(Videos,1234)

- At the data store level, find Video 1234 in namespace Videos, no detailed information the clients that rented it
- at the *application level*, we go inside the aggregate value and we discover that it was rented by Client 375657
- at the *application level*, we can execute get(Clients, 375657) to retrieve information about a/the customer that rented video 1234 (thus navigating the customer reference stored inside the video)

Example 3 - Relationships

```
{"loc": 1234,  
  "type": "dvd",  
  "rentals": [{"rental": {"rentalDate": "15/10/2021", "codCli": 375657},  
    "title": "pulp fiction",  
    "director": "quentin tarantino"}]}
```

Find the age(s) of customer(s) () that rented a given video

REMARKS:

- 1. Who knows that inside BLOB1 there is the code of the client that rented the video?
And where to look for the client information and how to match?
- 2. Since the relationship is not part of the aggregate, navigating it requires a sort of join, no system support for it
The application, the data store is completely unaware of that!
- 3. Since the relationship is not part of the aggregate, navigating it (at application level) requires two distinct data accesses, at two possibly distinct data nodes
 - initially we can execute get(Clients, 375657) to retrieve info about the customer that rented video 1234 (thus navigating the customer reference stored inside the video)

Advanced interaction

- Some systems support additional functionality
 - Use of indexes: The data must be indexed first
 - Some kind of additional index (e.g. full text) can be used
 - Example: Riak search

Key-value stores use cases

K-V Stores: Suitable Use Cases

- **Storing Web Session Information**
 - Every web session is assigned a **unique session_id** value
 - Everything about the session can be **stored by a single PUT** request or retrieved using a **single GET**
 - **Fast**, everything is stored in a **single object**
- **User Profiles, Preferences**
 - Every user has a unique **user_id/user_name** + preferences (language, time zone, design, access rights, ...)
 - As in the previous case: Fast, single object, single GET/PUT
- **Shopping Cart Data**
 - Similar to the previous cases

K-V Stores: When Not to Use

- Relationships among Data
 - Relationships between different sets of data
 - Some key-value stores (Riak) provide link-walking features
- Multi-operation Transactions
 - Saving multiple keys
 - Failure to save any of them → revert or roll back the rest of the operations
- Query by Data
 - Search the keys based on something found in the value part
 - Additional indexes needed (some stores provide them)
- Operations by Key Sets
 - Operations are limited to one key at a time
 - No way to operate upon multiple keys at the same time

Popular key-value data stores



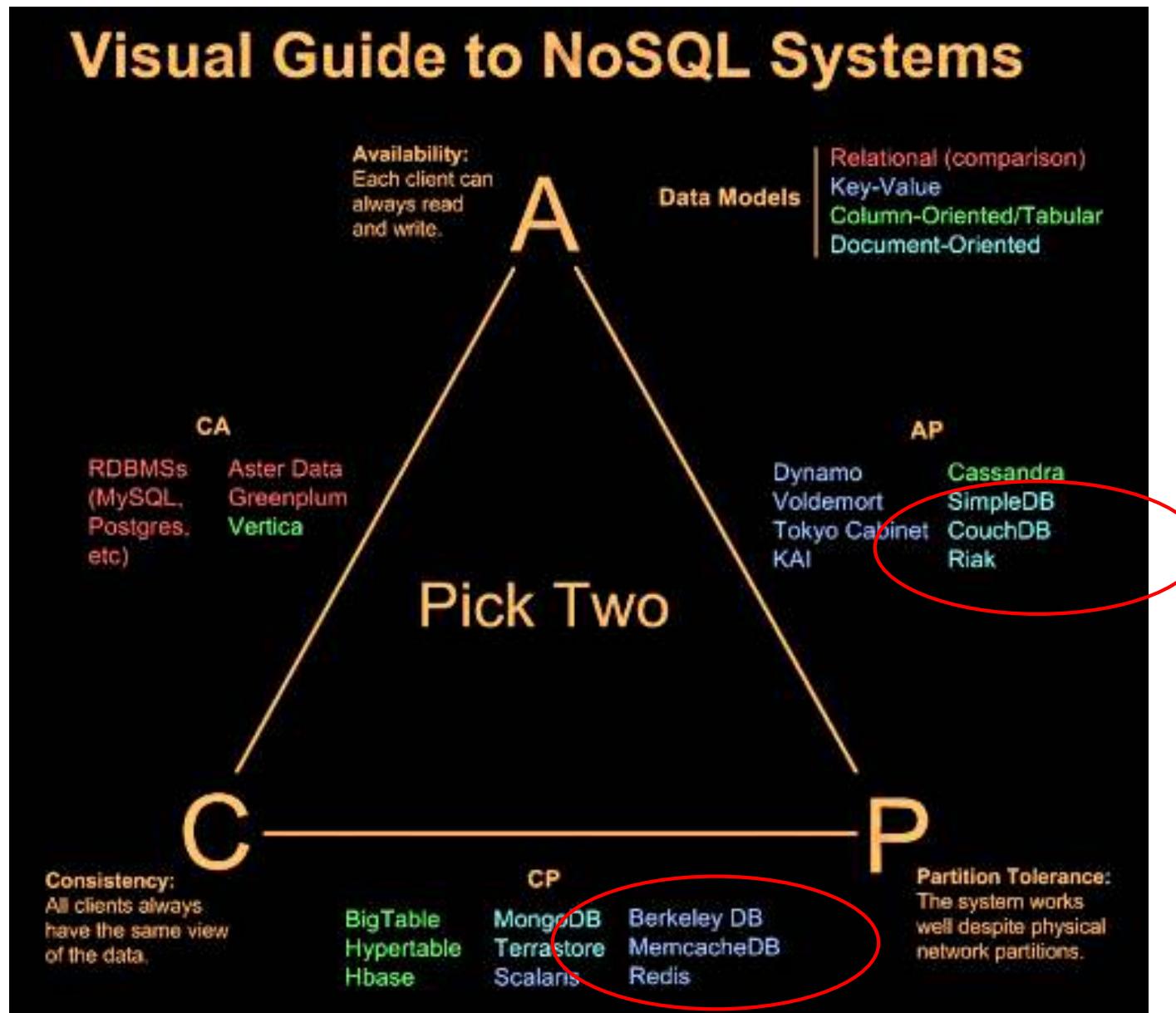
Ranked list: <http://db-engines.com/en/ranking/key-value+store>

Key-value stores are quite diverse

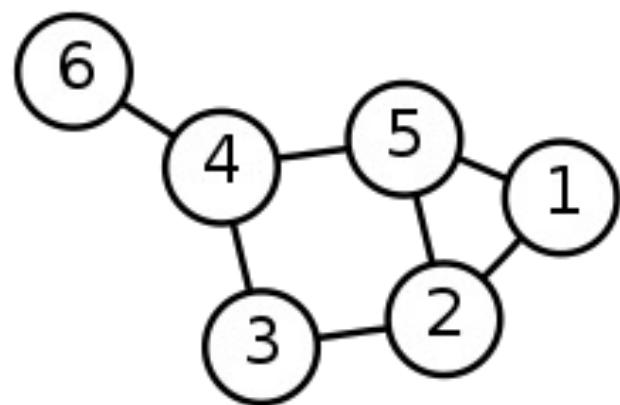
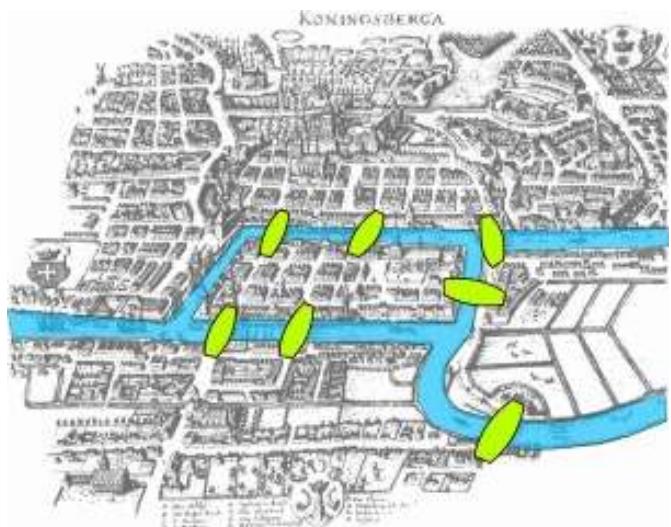
- **Dozens of key-value stores** – are all of them the same?
 - **Embedded local storages**
 - LevelDB
 - Local storage for many systems, Log-structured Merge Tree
 - **Distributed key-value Stores**
 - Riak, Infinispan
 - **Memory caches**
 - Redis, Memcached



CAP theorem: key-value stores



Graph Data Management



Graphs

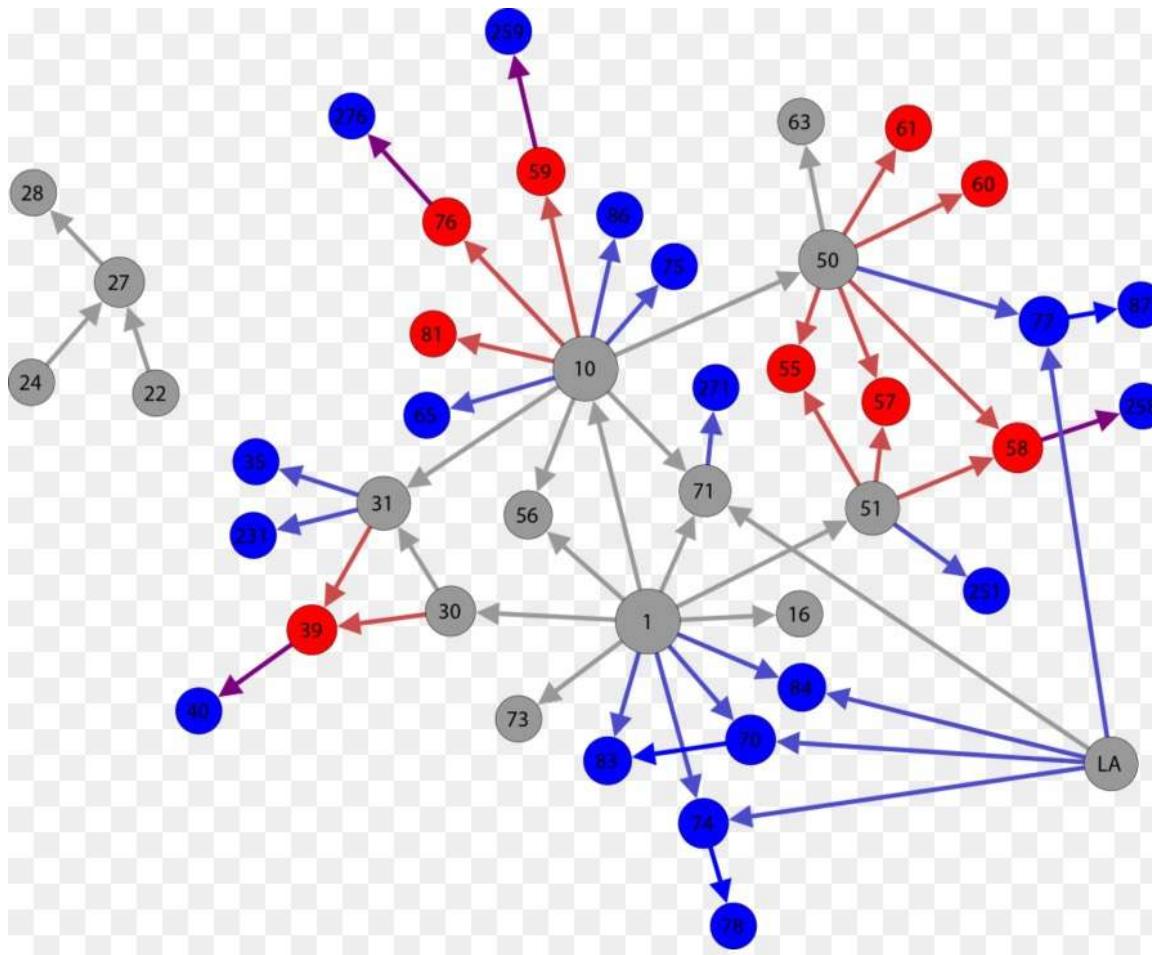
Why Graphs?

Why Graphs?

- Well-known generic data structure
 - One way to address the impedance mismatch
(objects in your Java don't match the structure of a DB)
 - Maths and algorithms are well understood
 - «blackboard friendly»
-
- Many use cases: data naturally modelled as graphs
 - Social networks
 - Web & Semantic Webs
 - Networks (roads, rails, telecommunication, ...)
 - Biological networks ...

Graphs

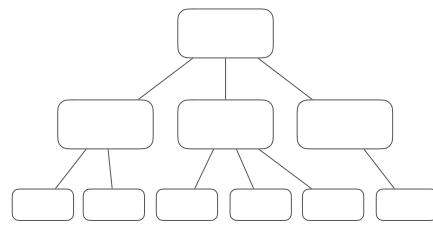
- Capture data consisting of complex relationships
- Focus on data inter-connection and topology



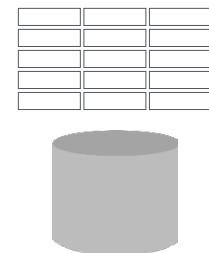
Why Graphs?

**Business
Processes**

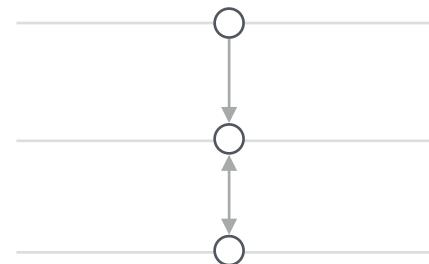
Hierarchies



**Data
Structure**



Traditional Supply Chain



Information

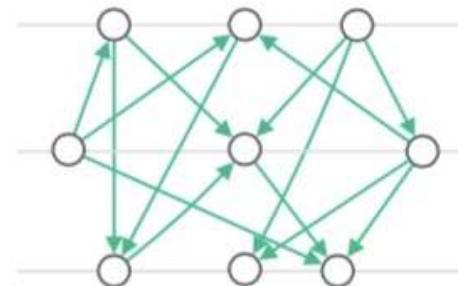


Why Graphs?

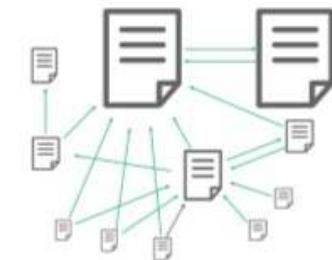
Business Processes



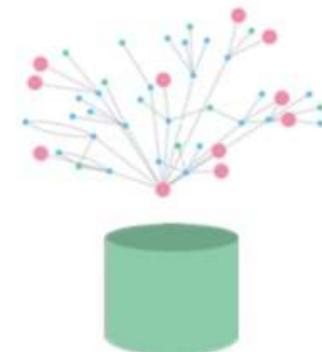
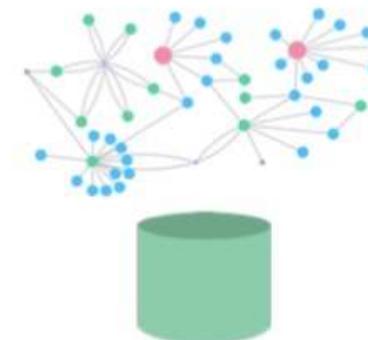
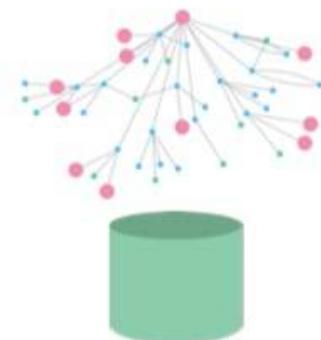
Dynamic Supply Chain



Knowledge



Data Structure



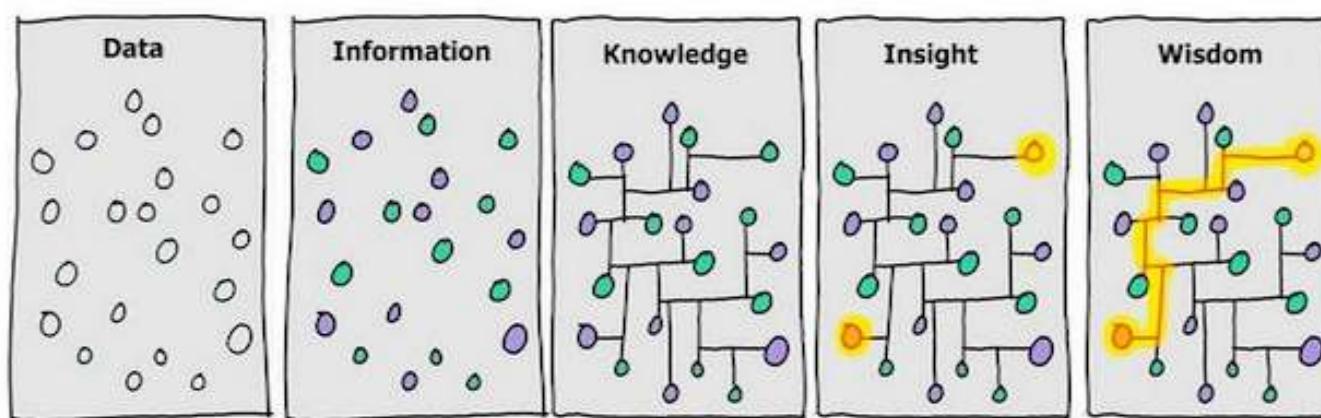
Graphs are everywhere!

Graphs provide a universal and simple blueprint for how to look at the world and make sense of it.

Everyone* uses graphs!

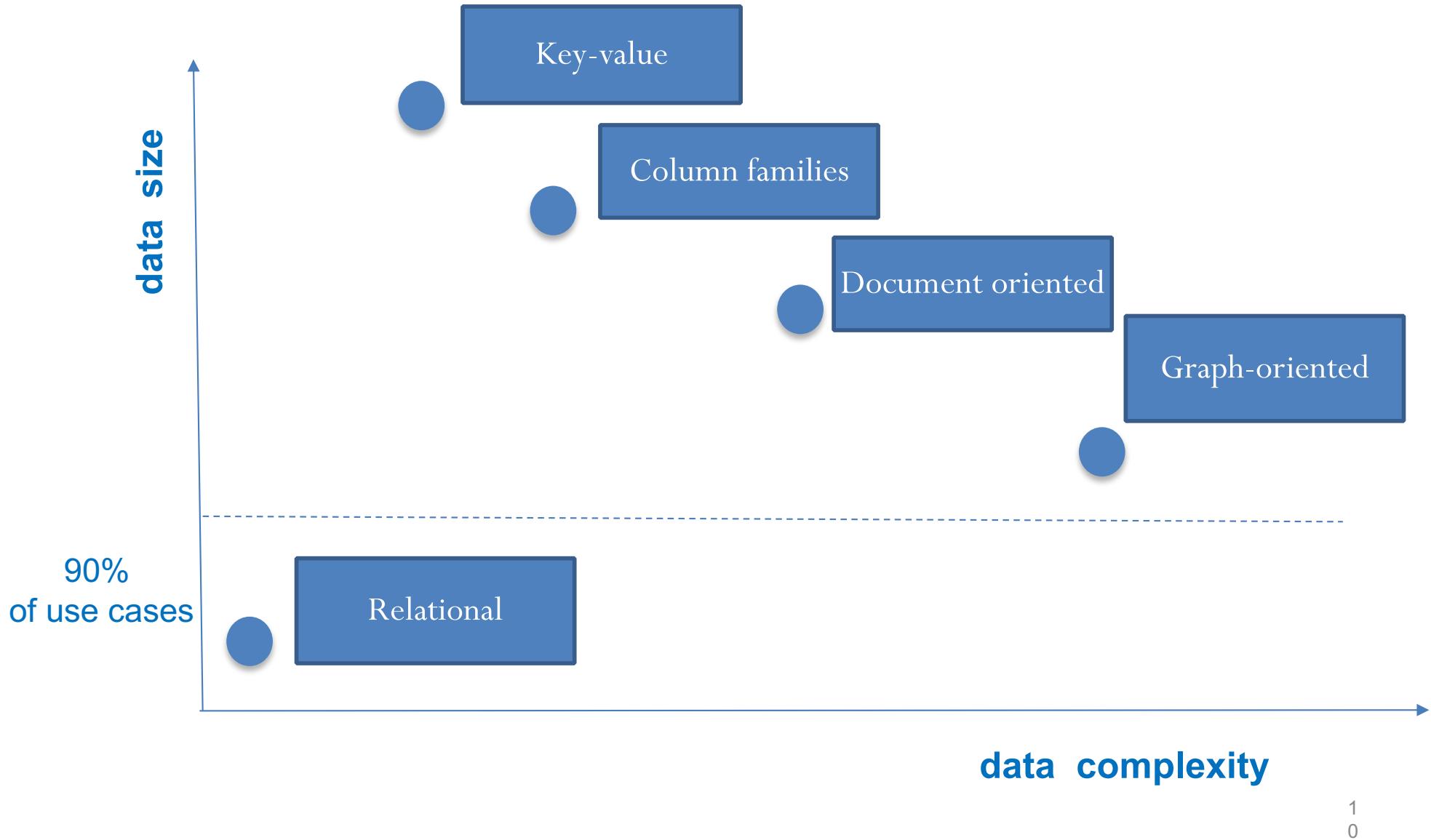
Tech-driving applications
= data science + multi-hop relationships

*not yet :-(



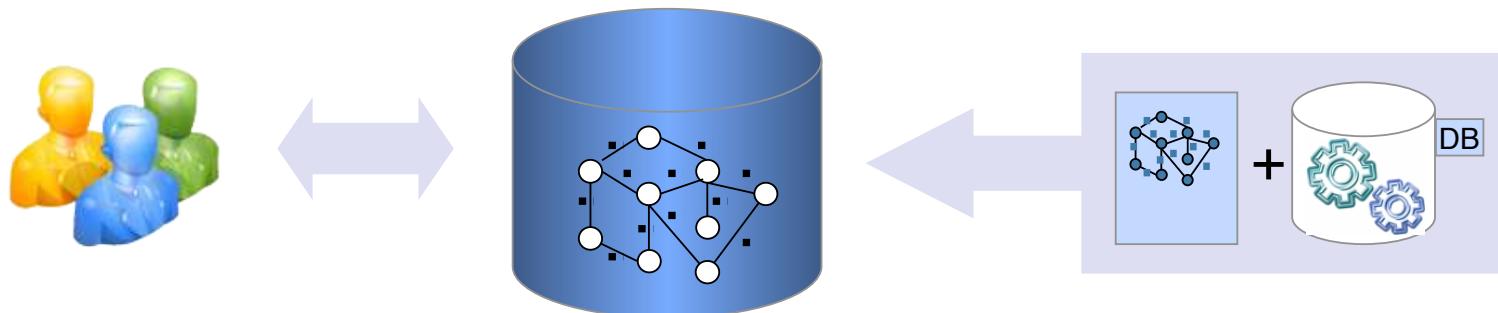
[Cartoon by David Somerville, based on a two pane version by Hugh McLeod.]

NoSQL data models

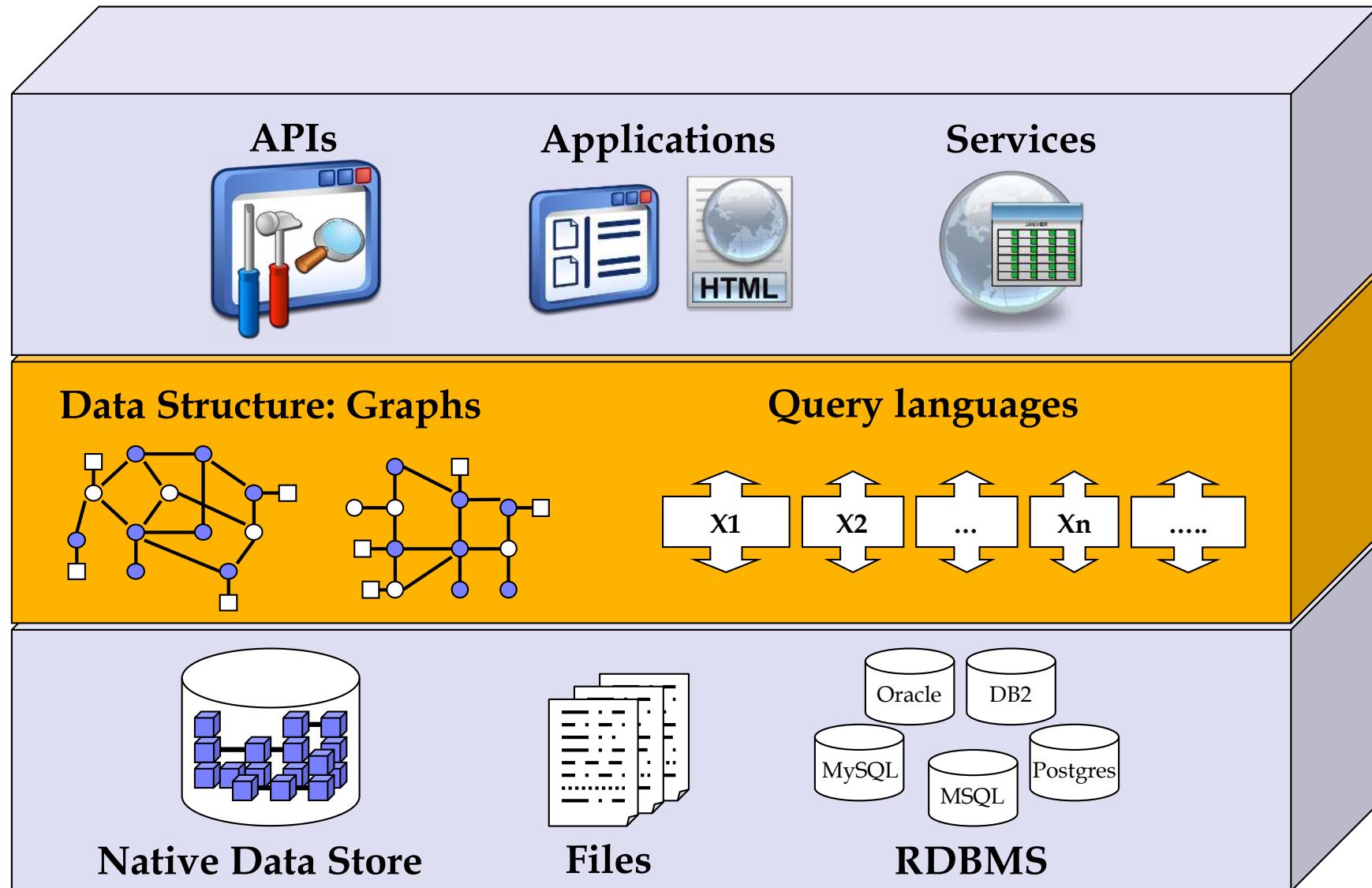


Data Management (database approach)

Manage **huge volumes** of data with **logical precision**
Separate modeling from implementation levels



The Database Modelling Level



Graph Data Model

Data structure

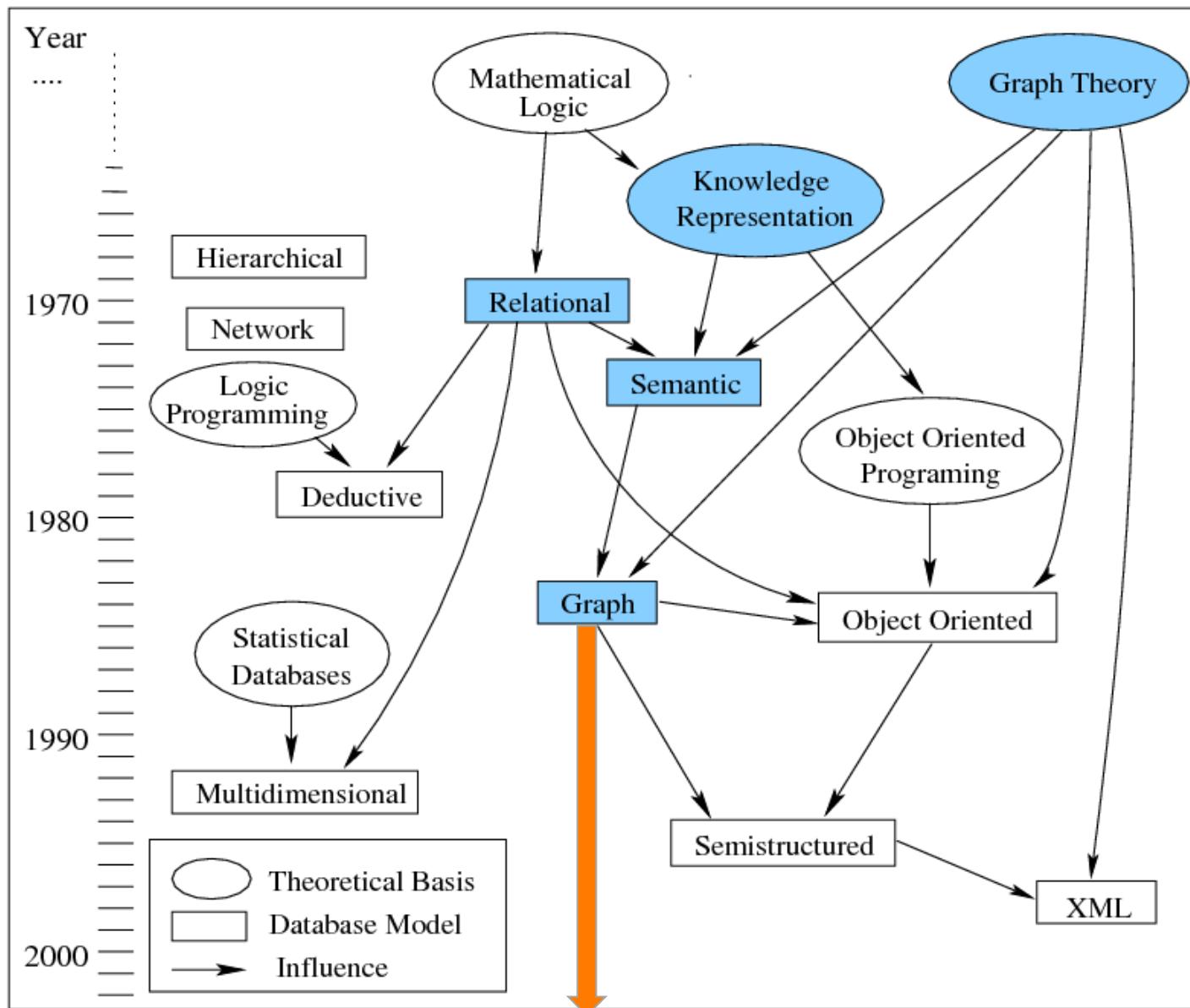
Data and/or the schema are represented by graphs, or by data structures generalizing the notion of graph (hypergraphs or hypernodes).

Graph Query Languages

Query Language

Data manipulation is expressed by graph transformations, or by operations whose main primitives are on graph features like paths, neighborhoods, subgraphs, graph patterns, connectivity, and graph statistics.

Historical Perspective – Data models



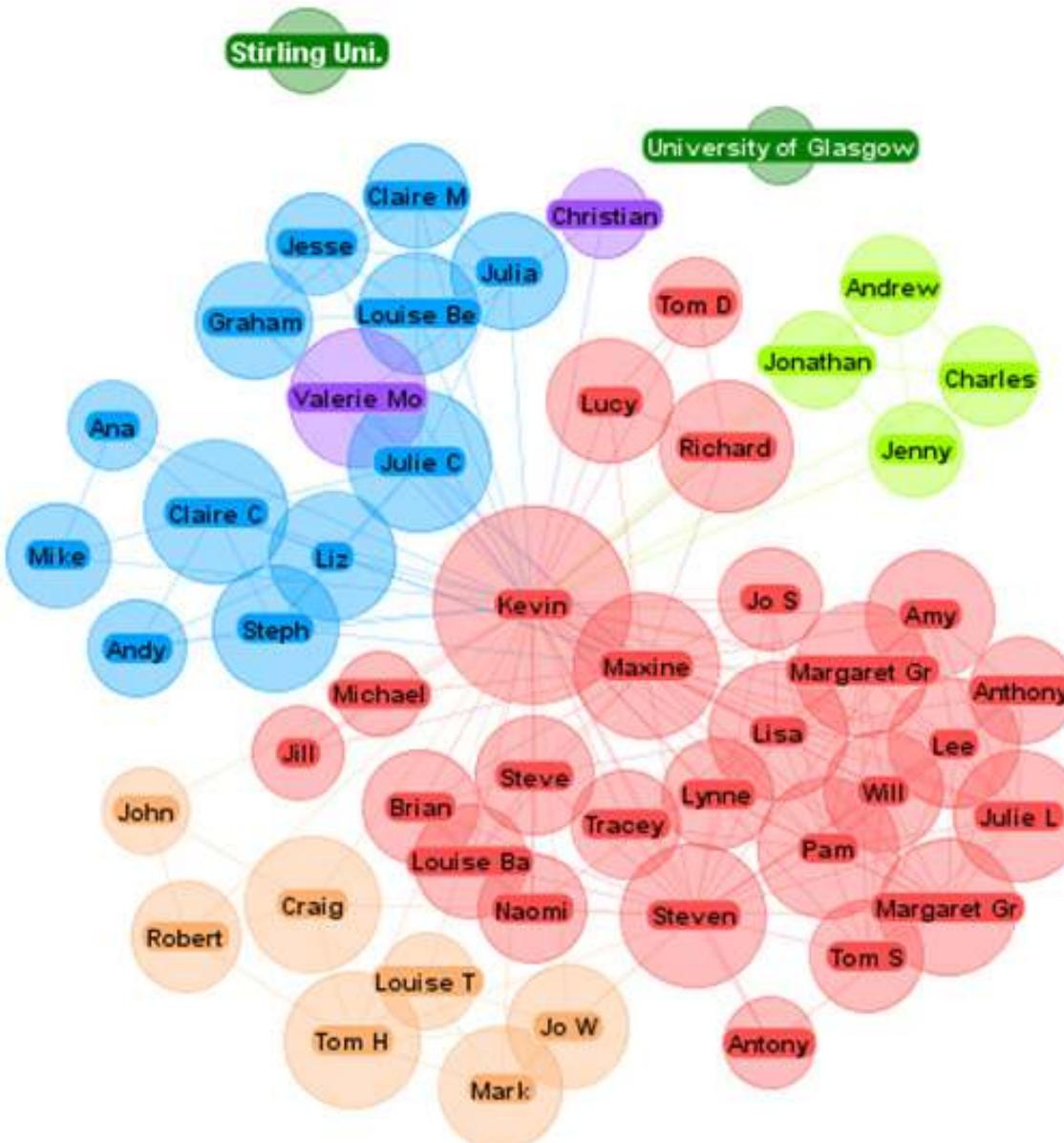
Y2k Change of Phase

Hardware able to process big graphs

As always, software behind hardware, and theory/models behind software

Use Cases

Social Network Domain

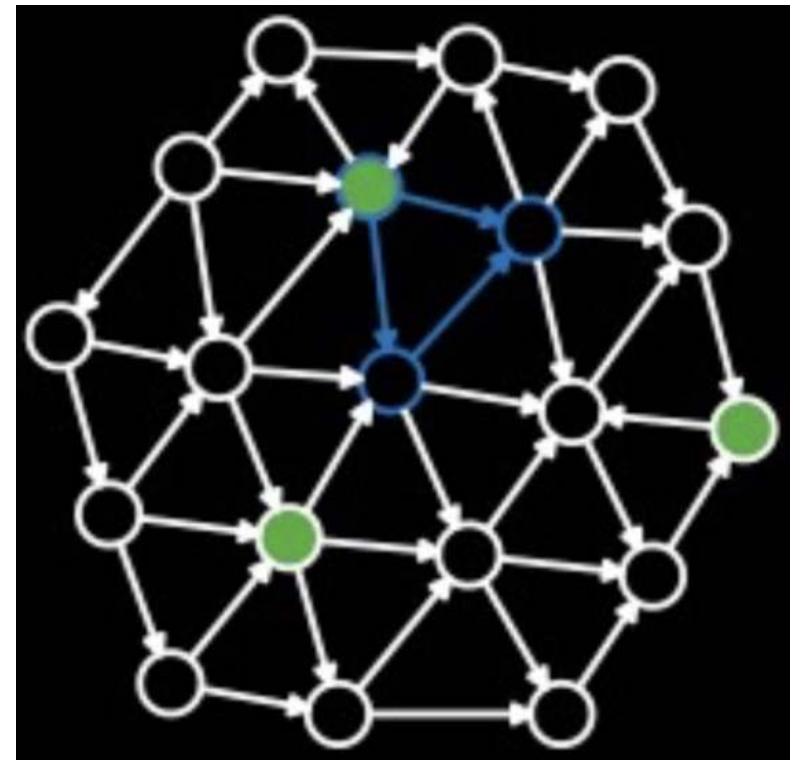


Social network domain – starting from Facebook

- Who likes me?
- Who likes something I've posted?
- Who likes something I like?
- Is any of my friend attending an event I am interested in?
- Is any of my old school mates friend any of my university colleagues?
- ...

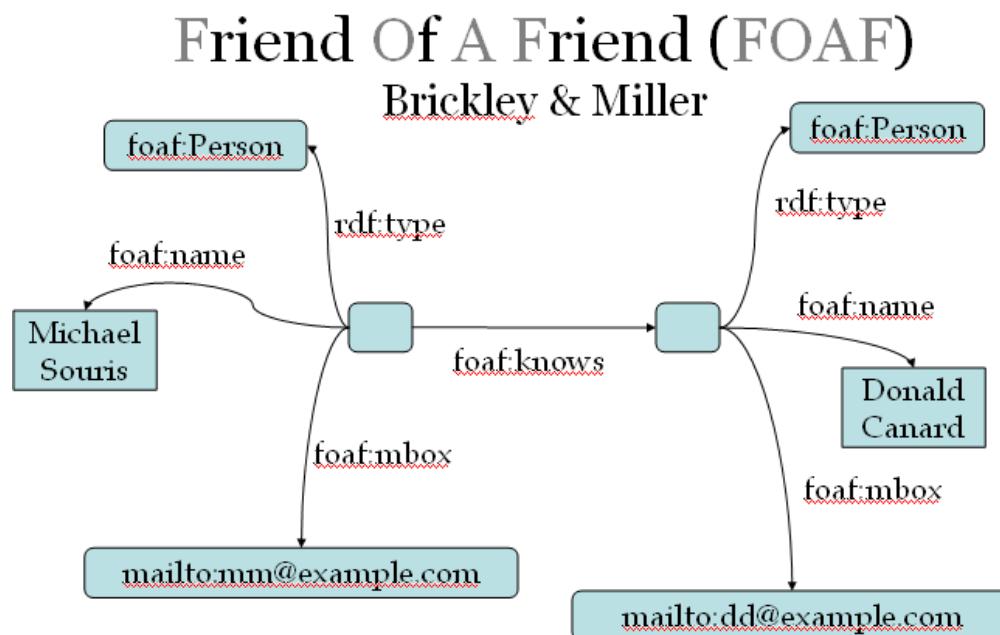
Query = Traversal

- “Who likes me”
 - traversal of depth 1 of the incoming nodes to me with the property “Like”
- “Is any of my old school mates friend of any of my university colleagues?”
 - traversal from me to all my friends, then from friend to friend



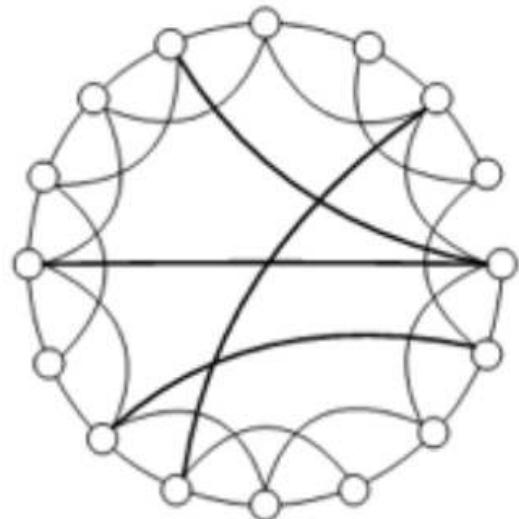
Use Cases - Web Domain

| Use Case | Graph Query |
|---|------------------|
| What is/are the most cited paper/s? | Degree of a node |
| What is the influence of article D? | Paths |
| What is the Erdős distance between authos X and author Y? | Distance |
| Are suspects A and B related? | Paths |
| All relatives of degree one of Alice | Adjacency |

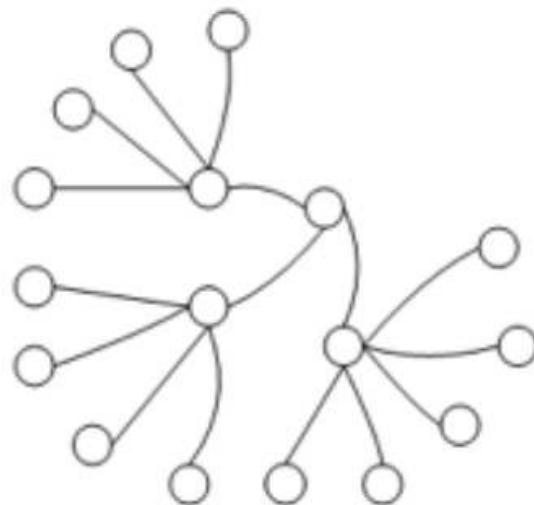


Use Cases – Social Network

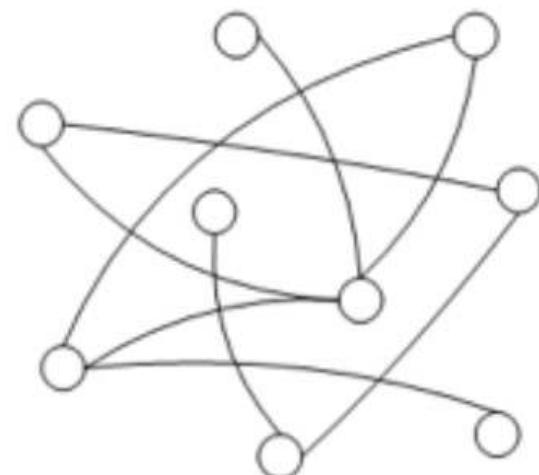
(a) Small-World Network (SWN)



(b) Scale-Free Network (SFN)

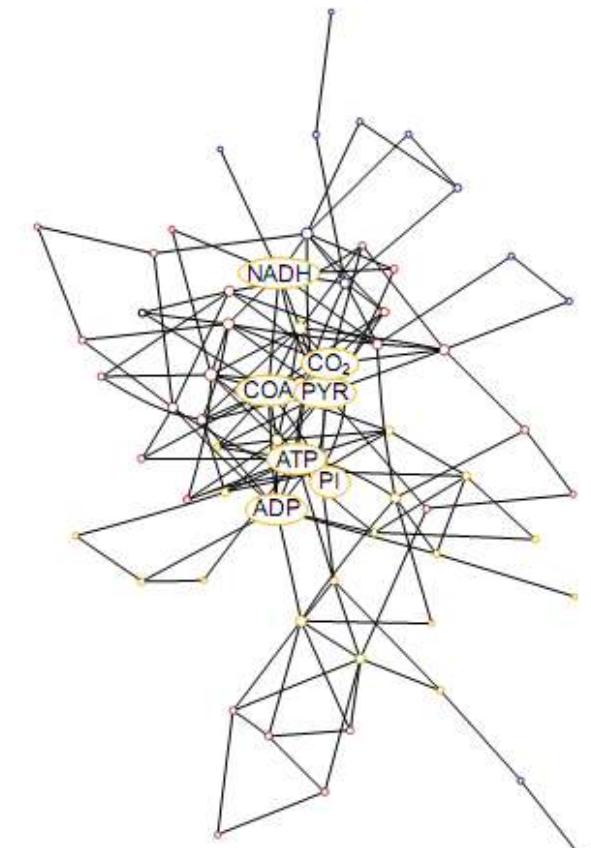


(c) Random Network (RN)



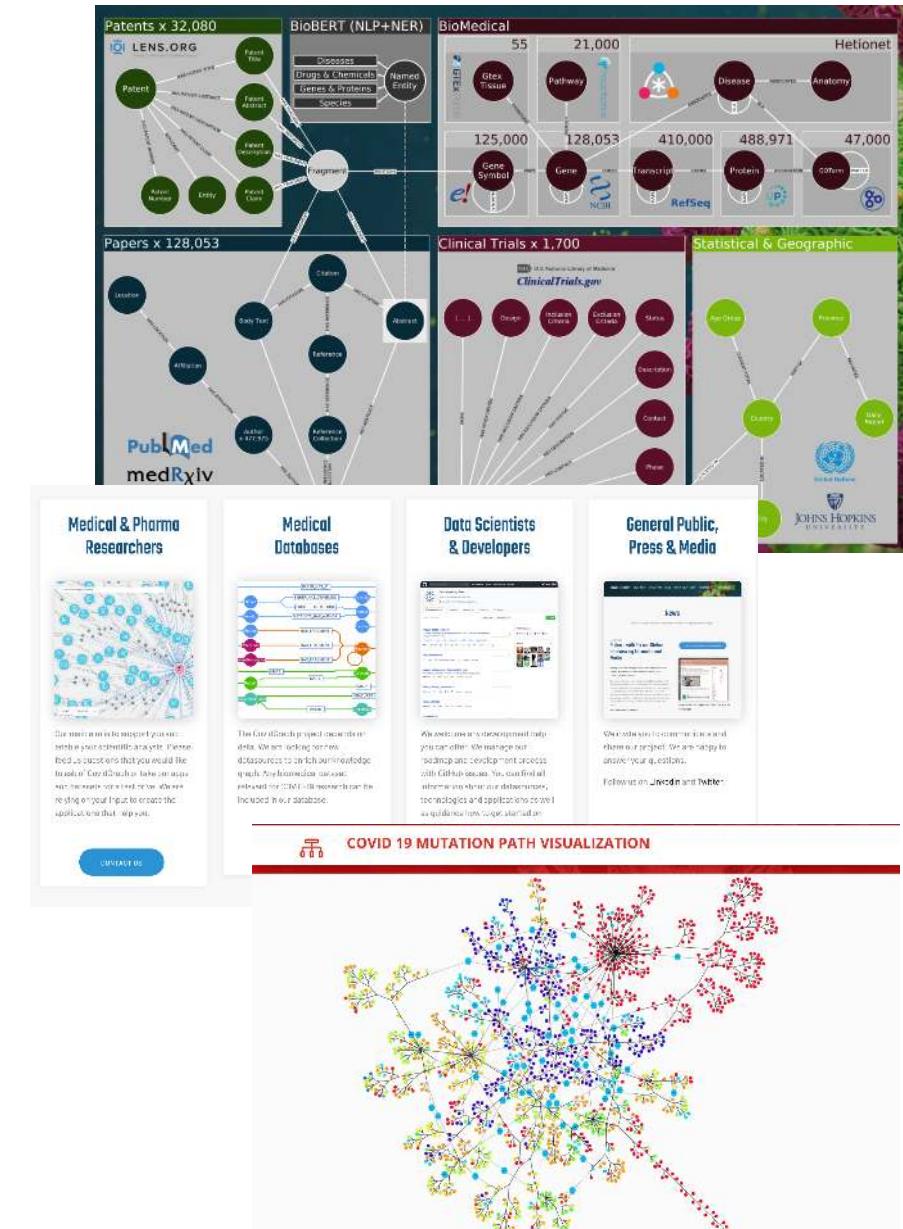
Use Cases - Biology Domain

| Use Case | Graph Query |
|---|---------------------------------------|
| Chemical structure associated with a node | Node matching |
| Find the difference in metabolisms between two microbes | Graph intersection, union, difference |
| To combine multiple protein interaction graphs | Majority graph query |
| To construct pathways from individual reactions | Graph composition |
| To connect pathways, metabolism of co-existing organisms | Graph composition |
| Identify “important” paths from nutrients to chemical outputs | Shortest path queries |
| Find all products ultimately derived from a particular reaction | Transitive Closure |
| Observe multiple products are co-regulated | Least common ancestor |
| To find biopathways graph motifs | Frequent subgraph recognition |
| Chemical info retrieval | Subgraph isomorphism |
| Kinase enzyme | Subgraph homomorphism |
| Enzyme taxonomies | Subsumption testing |
| To find biopathways graph motifs | Frequent subgraph recognition |



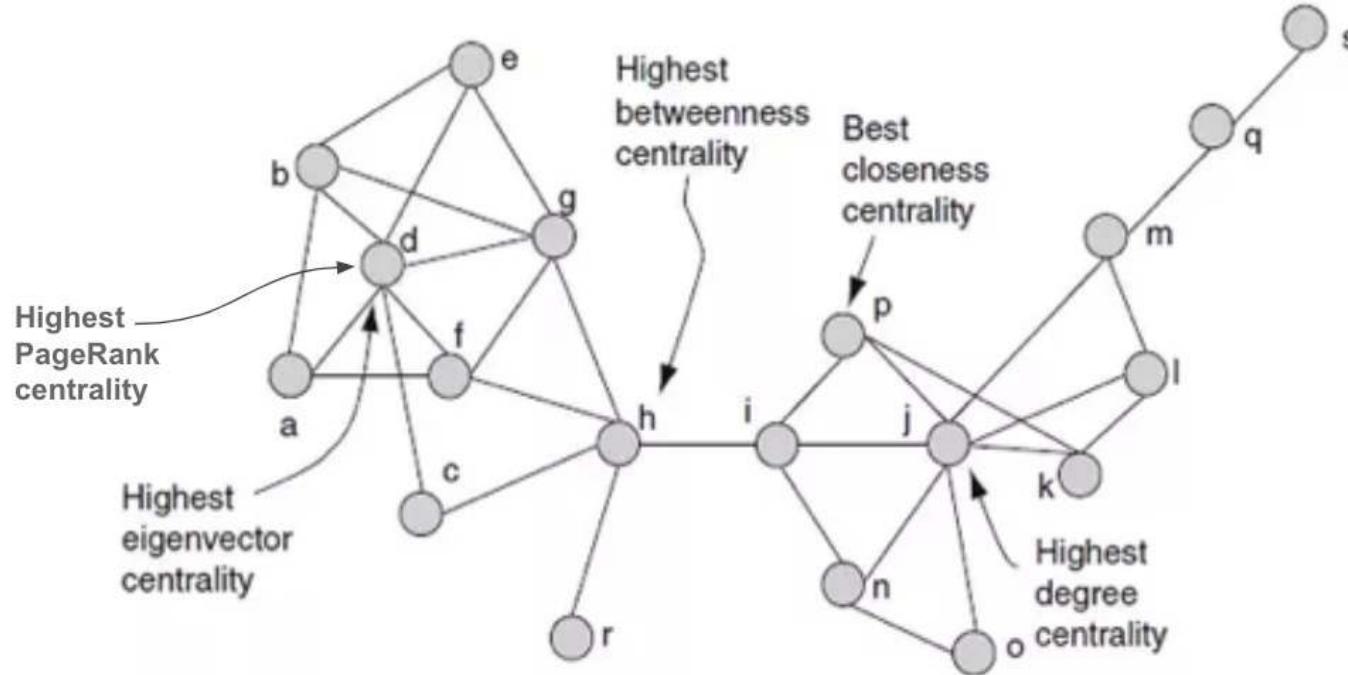
Use Cases – A plethora of domains

- Among which, the covidgraph.org initiative aiming at building the Covid19 knowledge graph:
 - Collecting patents, publications about the human coronaviruses
 - Biomedical data (genomics and omics)
 - Experimental data about clinical trials
 - Key demographic indicators



Use cases – A plethora of domains

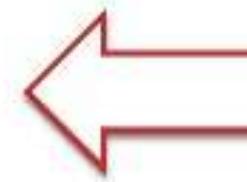
Centrality Measures



Different measures can be useful in different scenarios such web-ranking (page-rank), critical points detection (betweenness), transportation hubs (closeness)

Different types of graph data

- **large set of small graphs**
 - e.g., chemical compounds, biological pathways, ...
 - searching for graphs that match the query
- **few numbers of very large graphs**
 - or one huge (not connected) graph
 - e.g., Web graph, social networks, ...



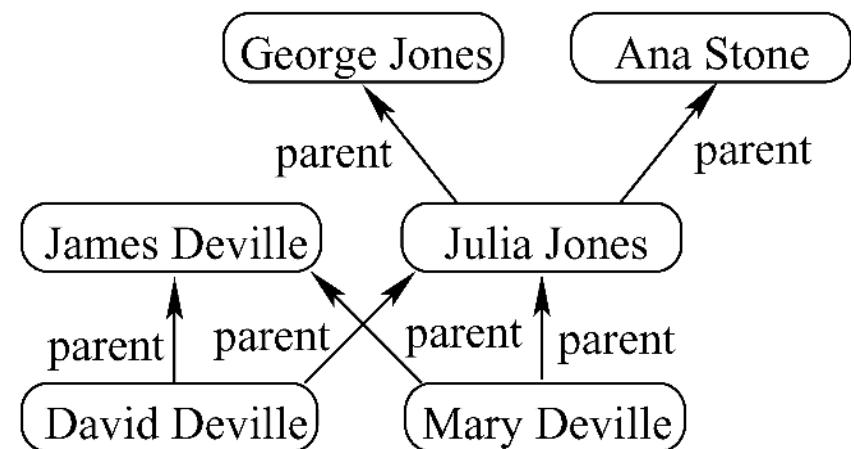
Graph Data Models

Graph data models

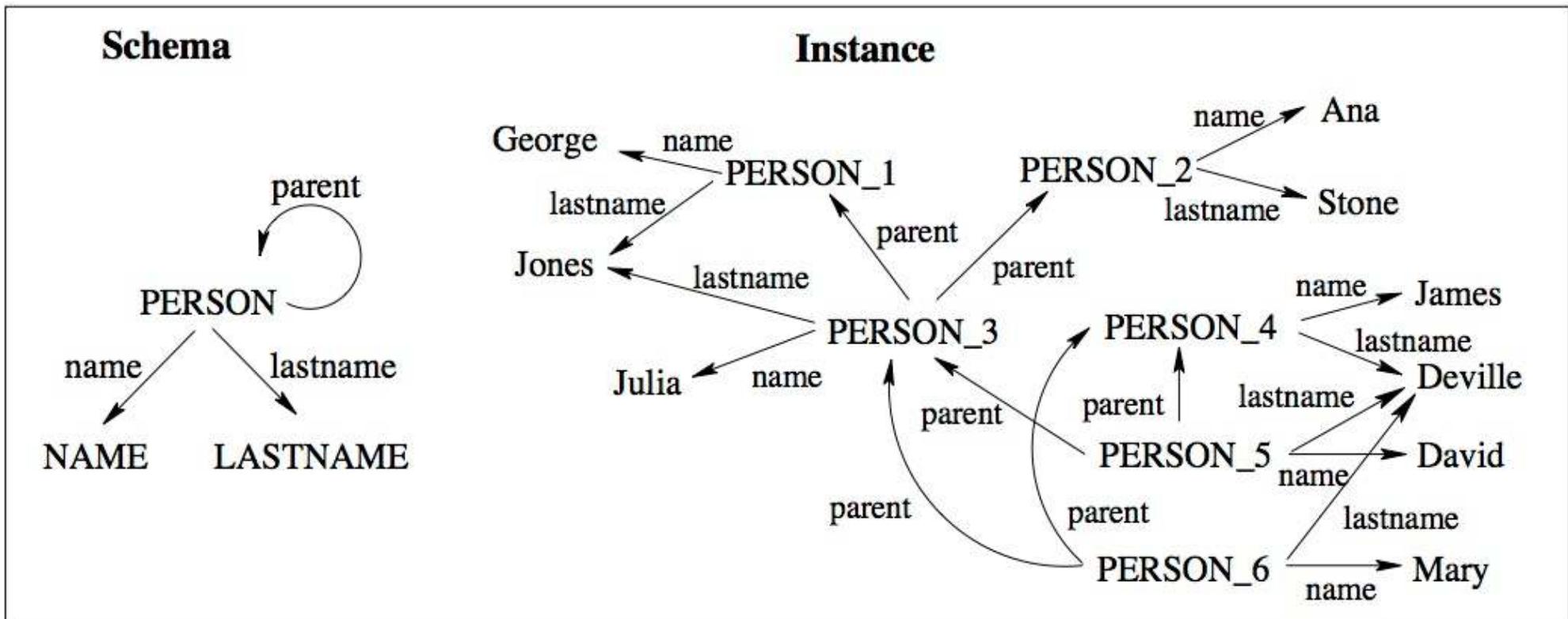
- How do humans conceptualize graphs?
- Interoperability issues (due to multiple heterogeneous data sources) are to be taken into account
- Balancing understandability and expressive power

Example: Genealogy Data and Diagram

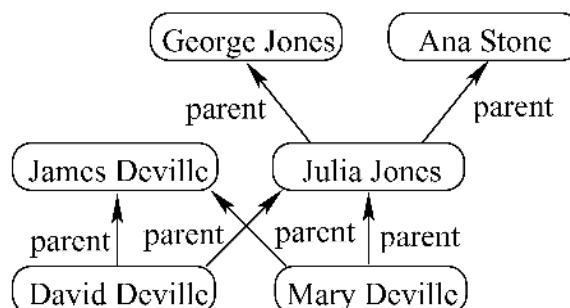
| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |



Simple Graph

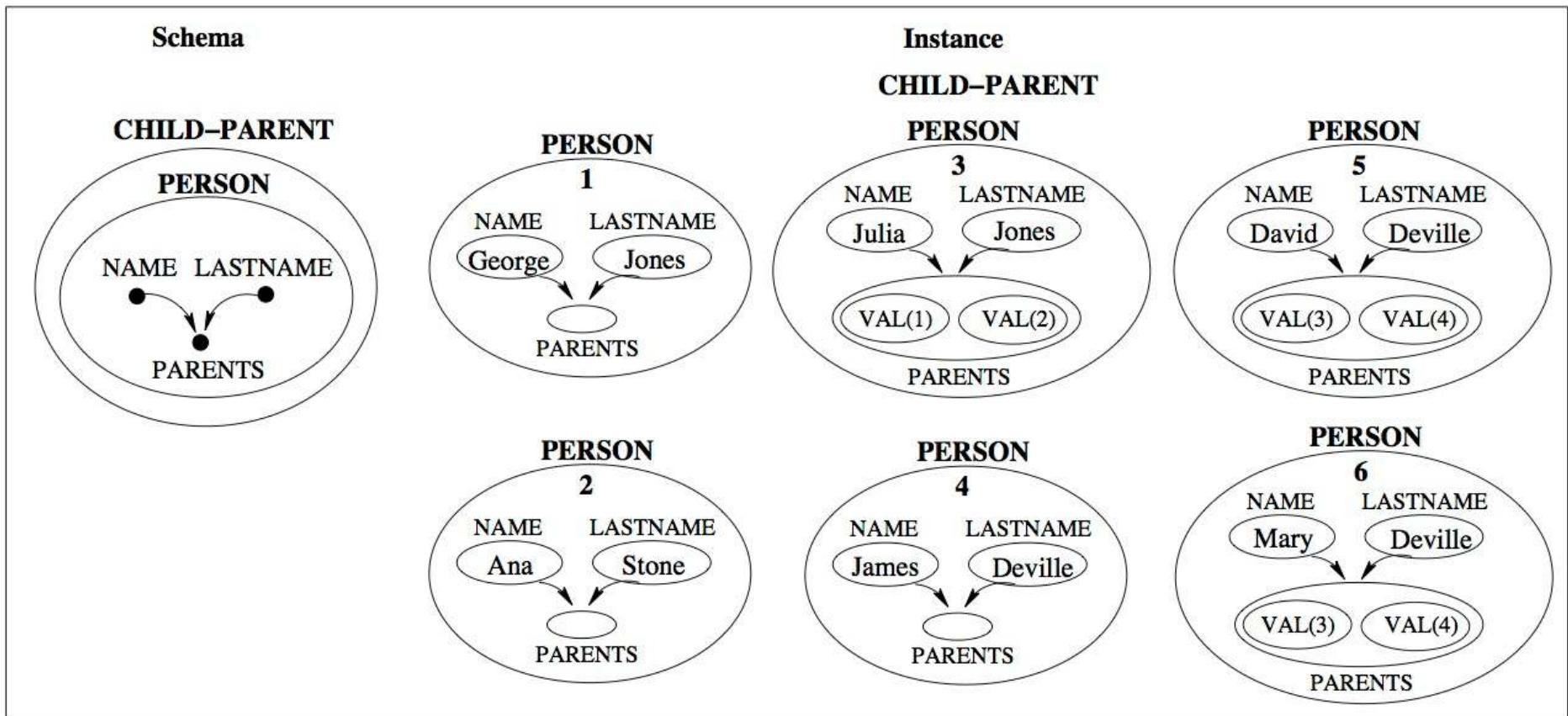


| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |

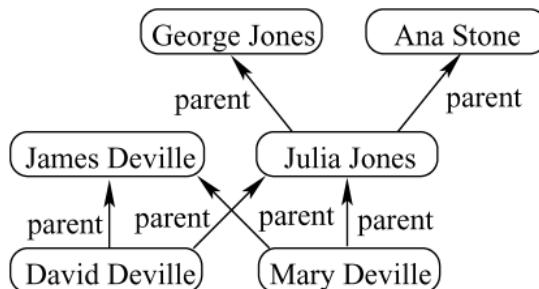


A hyperedge relates an arbitrary set of nodes

Hypergraph

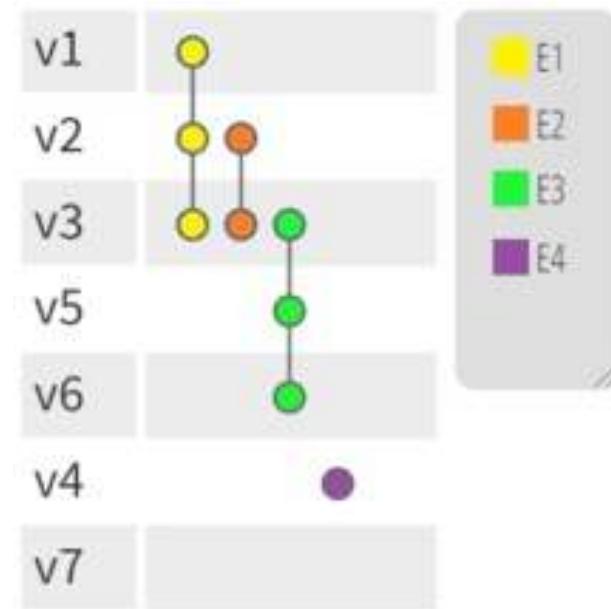
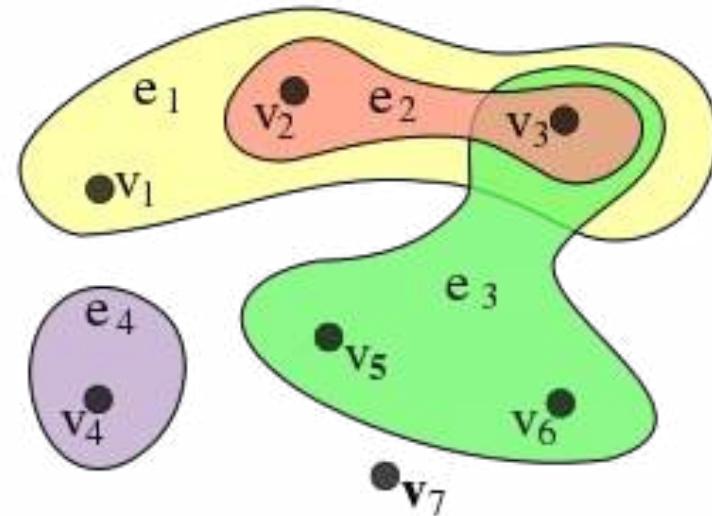


| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |



A hyperedge relates an arbitrary set of nodes

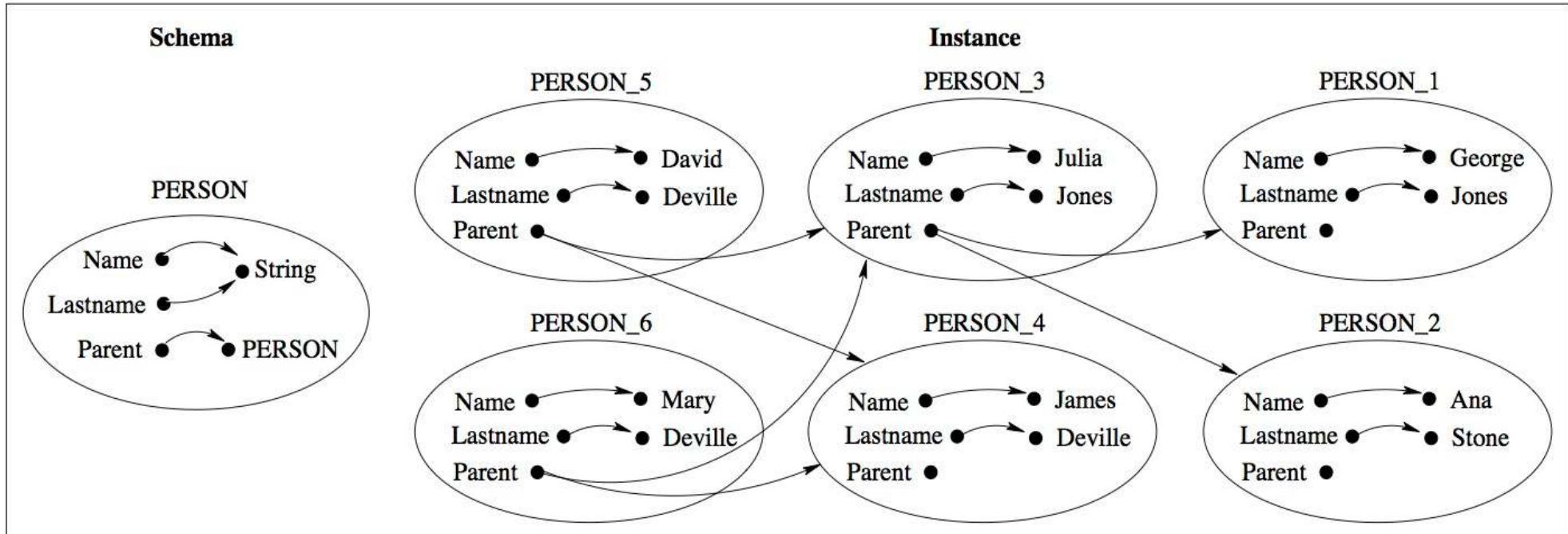
Hypergraph



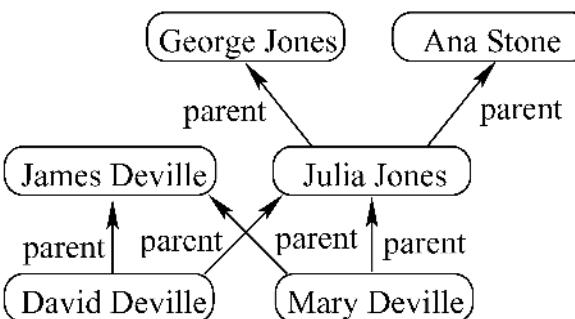
$X = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and
 $E = \{e_1, e_2, e_3, e_4\} = \{\{v_1, v_2, v_3\},$
 $\{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\}.$

Hypernode

A hypernode is a directed graph whose nodes can themselves be graphs (or hypernodes), allowing nesting of graphs.



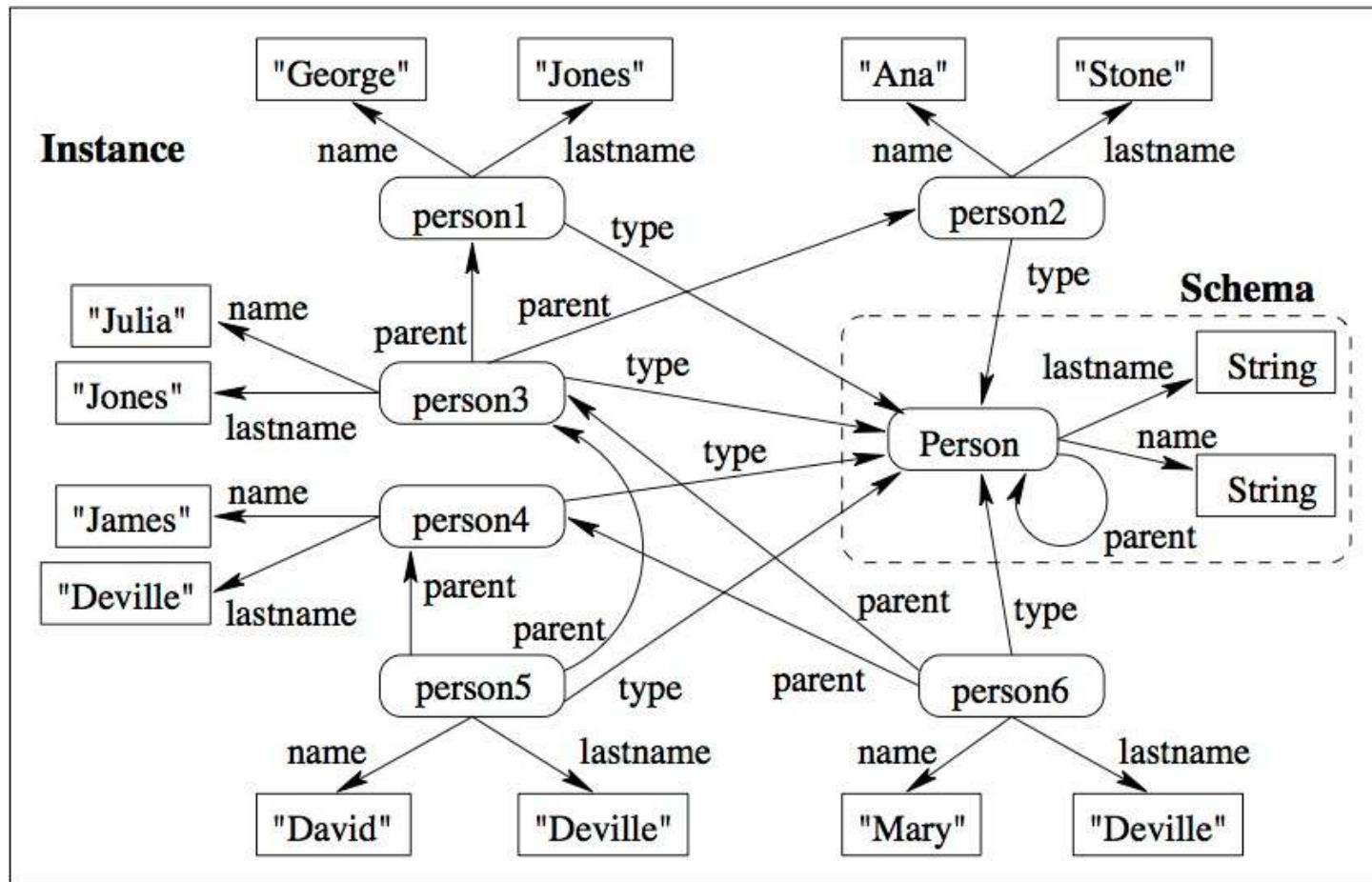
| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |



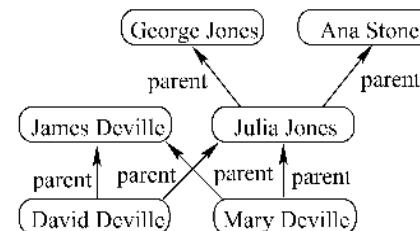
Currently, two main graph models

- RDF (triples)
- Property graphs

RDF

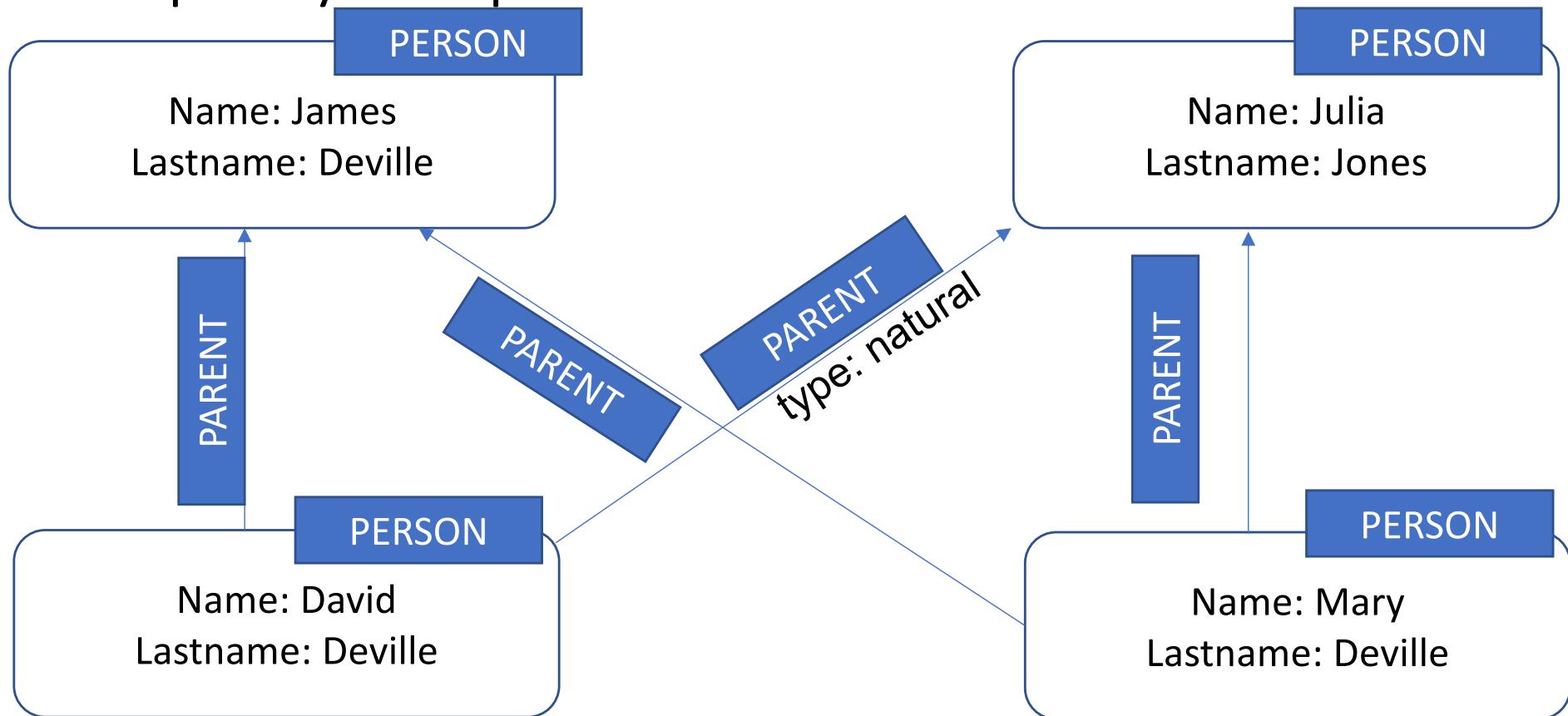


| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |

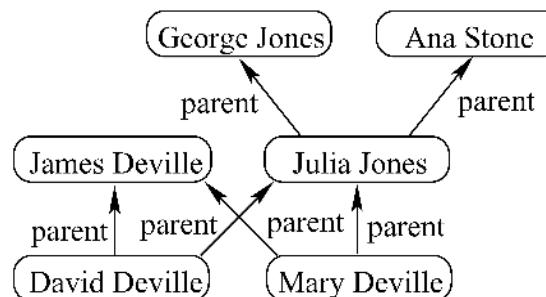


(subject, predicate, object) triples

Property Graph

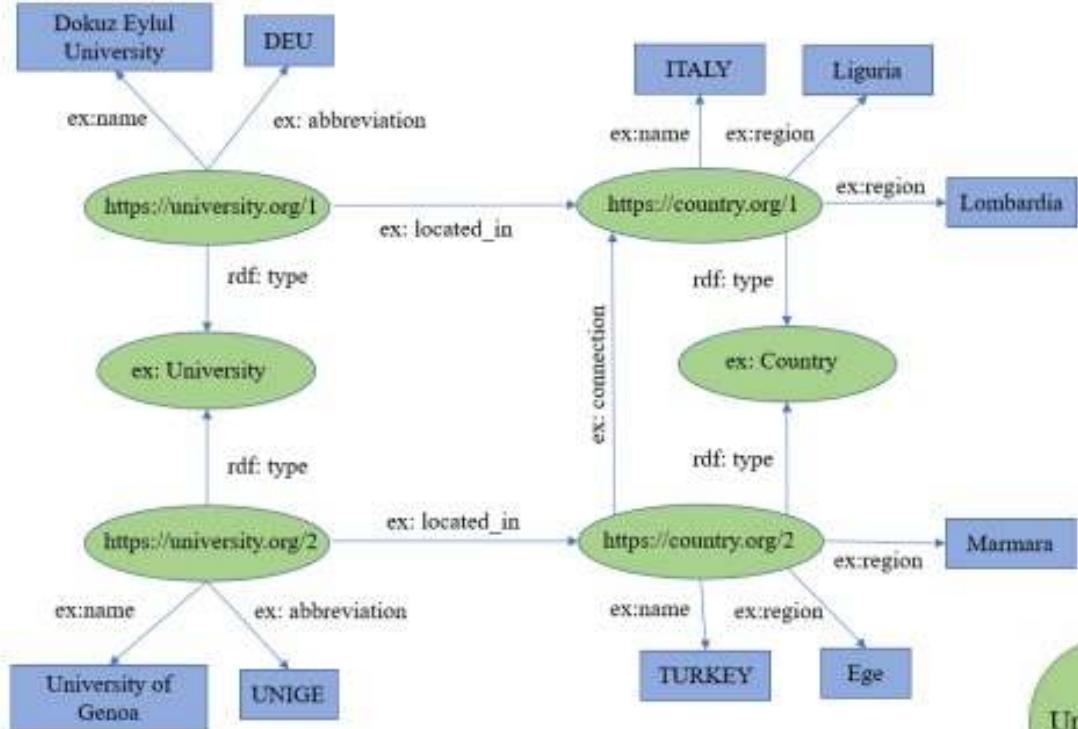


| NAME | LASTNAME | PERSON | PARENT |
|--------|----------|--------|--------|
| George | Jones | Julia | George |
| Ana | Stone | Julia | Ana |
| Julia | Jones | David | James |
| James | Deville | David | Julia |
| David | Deville | Mary | James |
| Mary | Deville | Mary | Julia |

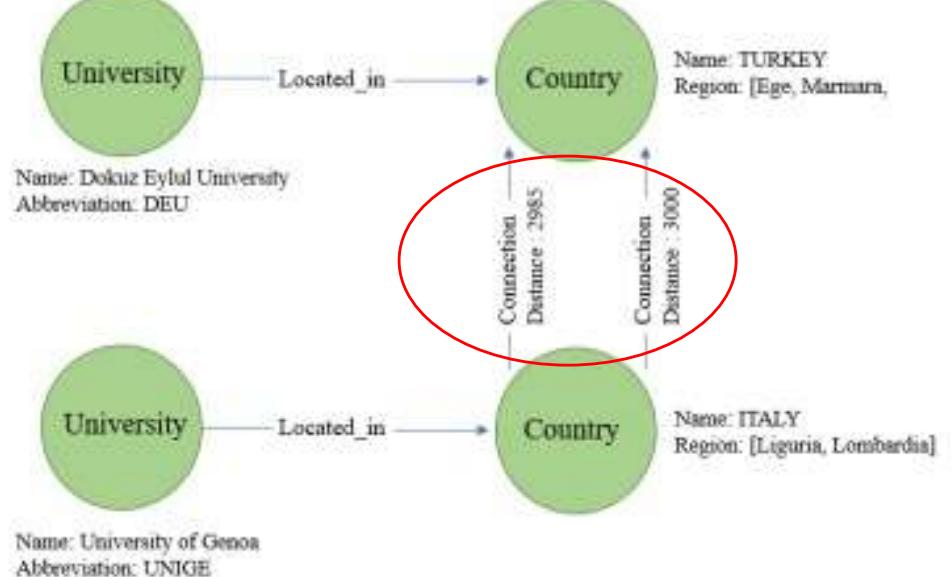


RDF & Property Graph

rdf: https://example.org/rdf-schema/
ex : https://example.org/

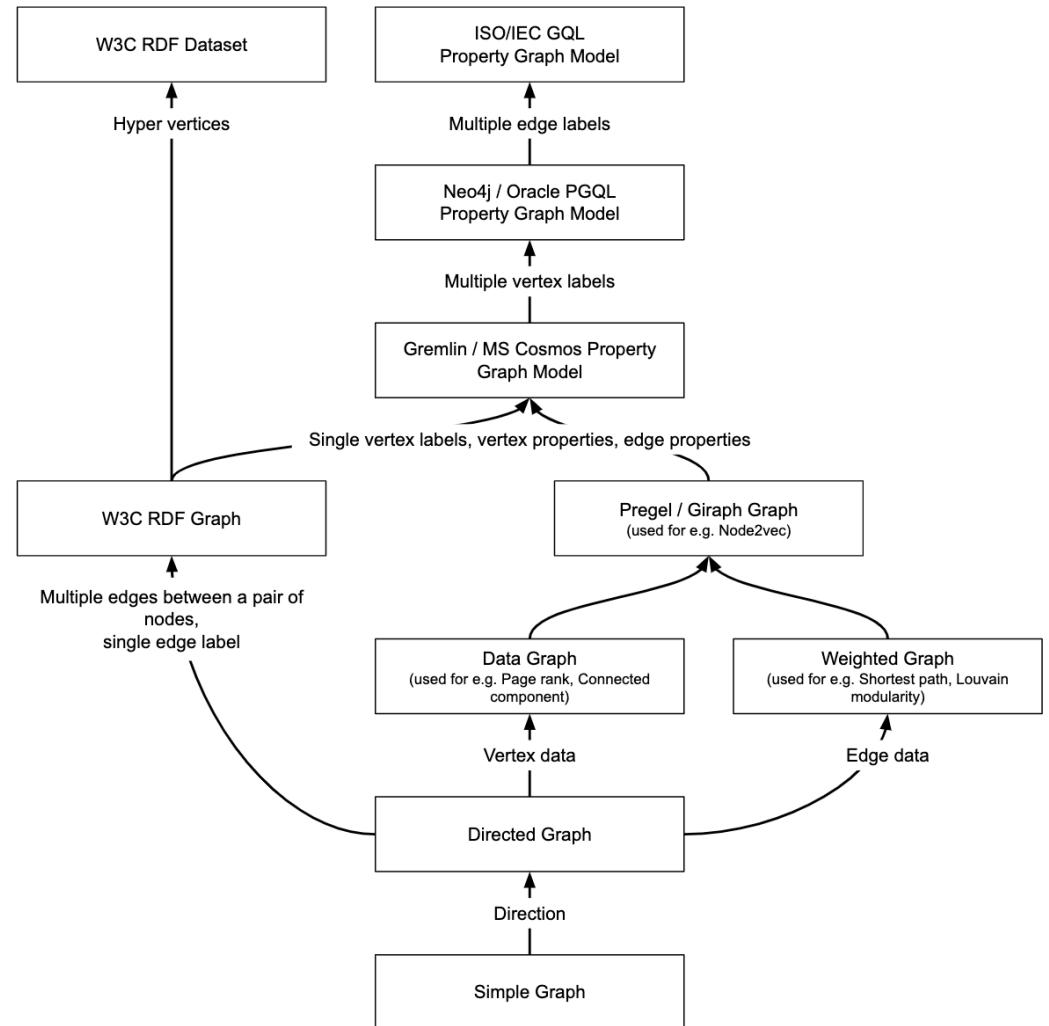


*we'll discuss transformations
in detail later
(after introducing the
constructs of the two
data models)*



A lattice of data models

- How expressive and human-friendly is a data model?
- A data model per use case
- Need of making different data models interoperable via mappings or direct translations



Property graph & data model interoperability

- Property Graph as an object model (neo4j, Apache Tinkerpop & multimodel Azure Cosmos, OrientDB, Oracle Graph)
 - But we can view it as an extended relational model (graph as an indexing and access layer over relational data e.g. SAP Hana graph, Tigergraph)
 - And as an access layer on top of RDF (BlazeGraph, StarDog)
- See later discussion about interoperability

Graph Interchange Format

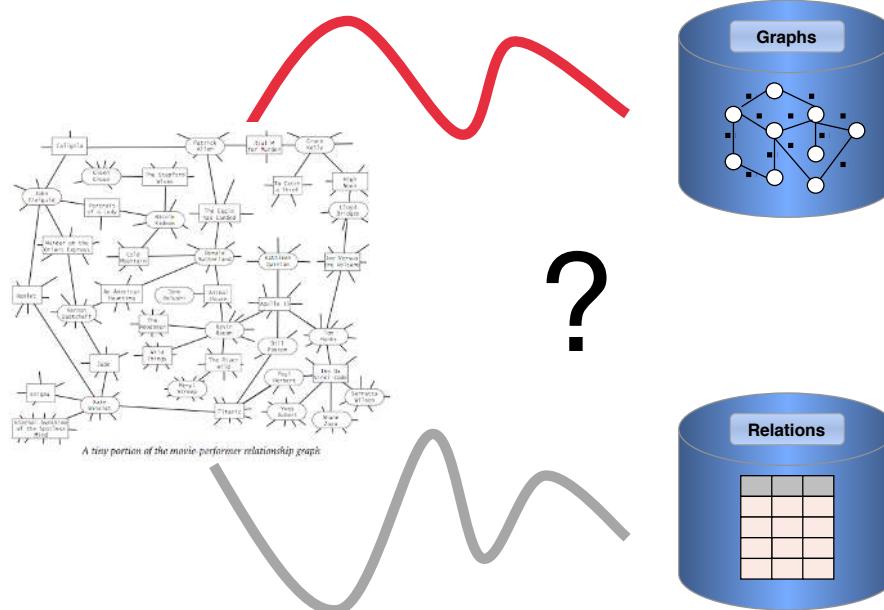
- GraphML
 - XML file containing a graph element, within which is an unordered sequence of node and edge elements
 - Each node element has a unique id
 - Each edge has source and target
 - Allows the specification of directed, undirected, hypergraphs, ...
 - Originally designed for graph visualization
 - Supported by many graph-based systems (neo4j, tinkerpop, ...)

Graph Query Languages

Graph Query Languages

| PROPERTY | Neighborhoods | Adjacent Edges | Degree of a Node | Path | Fixed-length path | Distance | Diameter |
|----------|---------------|----------------|------------------|------|-------------------|----------|----------|
|----------|---------------|----------------|------------------|------|-------------------|----------|----------|

as a graph data model?



as a relational model?

Just a list of typical requests?

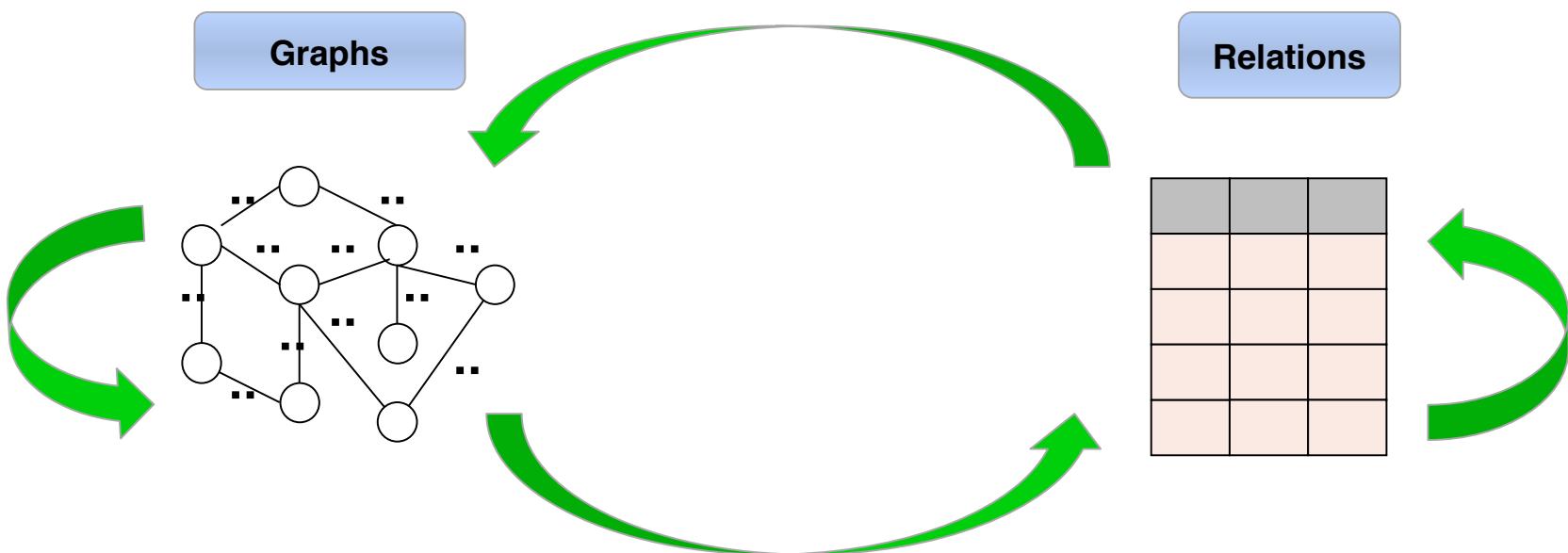
A. “Basic” Graph Queries

1. Pattern matching
2. Adjacency / neighborhood
3. Reachability / connectivity
4. Summarization
5. ...

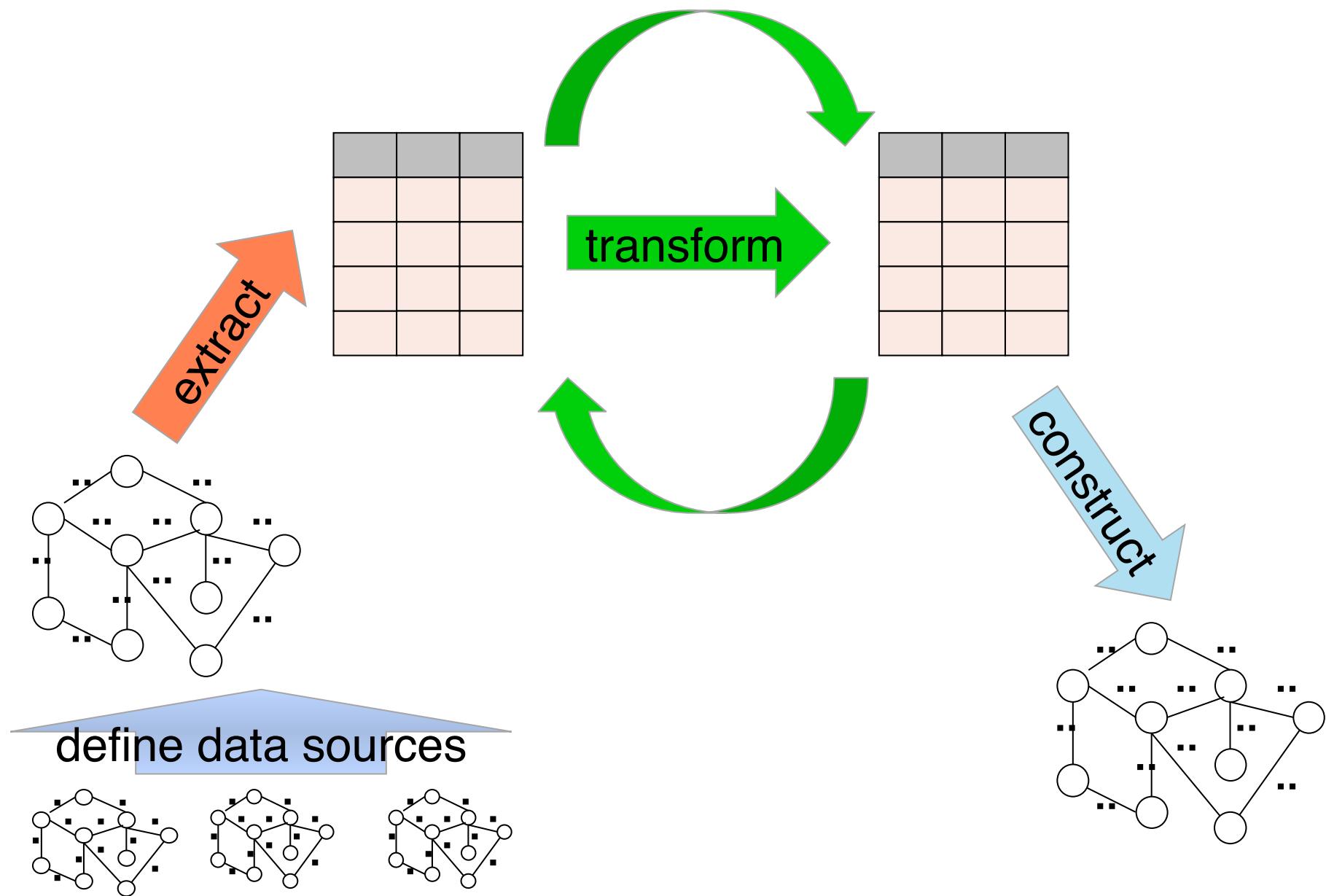
B. Analytical Queries

1. Centrality measures
2. Diameter and other global properties
4. Graph properties and parameters
5. ...

Graph Query Languages

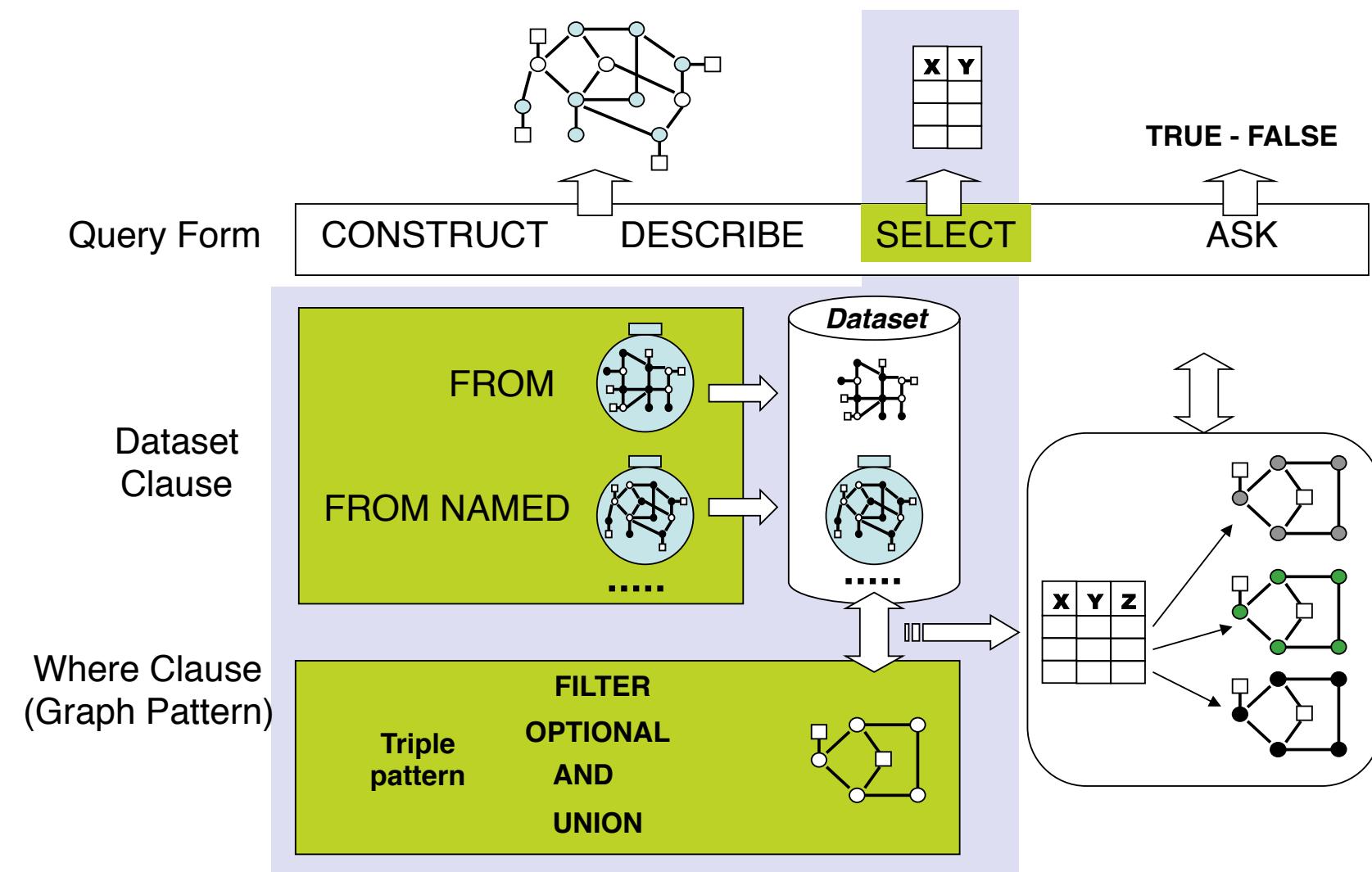


Graph Query Languages



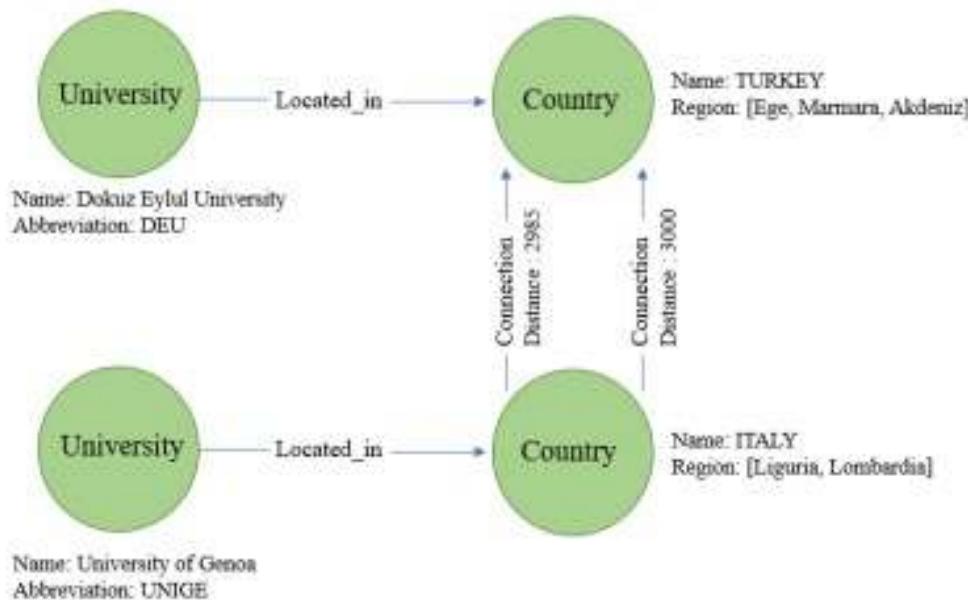
| Language | Define Source | Extract | Transform | Construct |
|-----------------|----------------------|---------------------|------------------|----------------------------|
| SQL | FROM | | WHERE | SELECT |
| SPARQL | FROM, Service | pattern matching | operators | Select, ASK, Construct, |
| Cypher | | MATCH | WHERE | RETURNS |
| | | | | |

SPARQL Query



Cypher - bindings

```
MATCH (u:University)-[:located_in]->(c:Country)  
RETURN c.name
```



| u | c |
|---|--|
| University Name: Dokuz Eylul University Abbreviation: DEU | Country Name: TURKEY Region: [Ege, Marmara, Akdeniz] |
| University Name: University of Genoa Abbreviation: UNIGE | Country Name: ITALY Region: [Liguria, Lombardia] |

GQL – Graph Query Language

<https://www.gqlstandards.org>

Ongoing ISO effort

Standalone graph query language to complement SQL

Manifesto <https://gql.today>

Since 2017 work has been proceeding on extending SQL with read-only property graph extensions based on the pattern-matching paradigm of Cypher and PGQL. SIGMOD 2017 saw the publication of the future-looking G-CORE paper on fresh directions in PG querying, matched by implementation of compositional queries and graph views in Cypher for Apache Spark. Since spring 2018 the property graph world has been coalescing around the idea of a single GQL language, drawing on all of these precedents, open to other inputs, and closely coordinated with key aspects of SQL and its ecosystem.

GQL – Towards a standard

| Cypher |
|--|
| *Create-Read-Update-Delete |
| *Regular Path Queries (RPQs)- Not fully |
| *Graph Construct/Project |
| *Composable |
| *Neo4j DB, HANA Graph, Redis Graph, Anzograph, AgensGraph, MemGraph, Cypher for Gremlin/Spark, openCypher. |

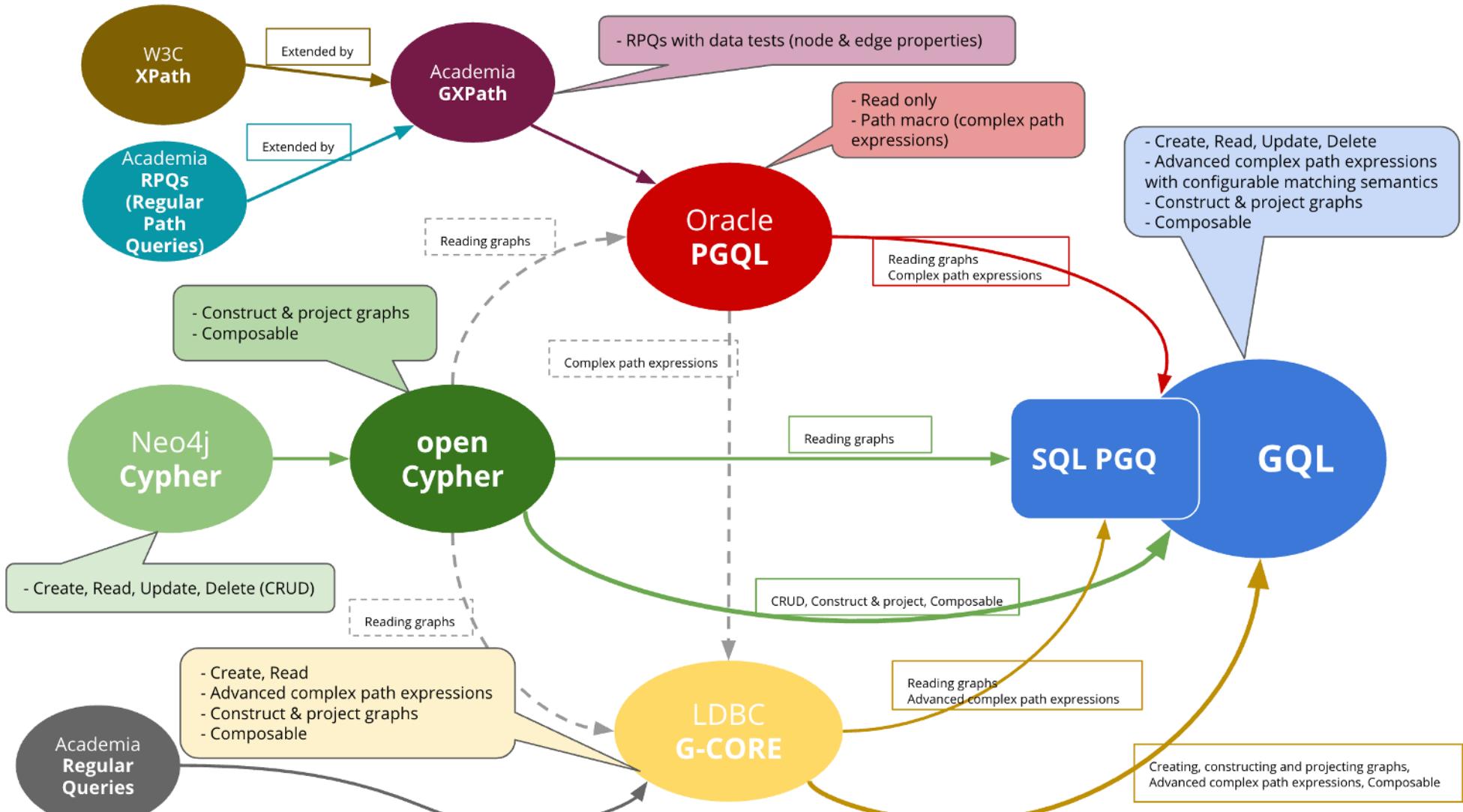
| PGQL |
|------------------------------|
| *Create-Read-Update-Delete |
| *Regular Path Queries (RPQs) |
| *Graph Construct/Project |
| *Not composable yet |
| *OraclePGX |

| G-CORE |
|------------------------------|
| *Create-Read |
| *Regular Path Queries (RPQs) |
| *Graph Construct/Project |
| *Composable |
| *G-Core interpreter on Spark |



| GQL |
|------------------------------|
| *Create-Read-Update-Delete |
| *Regular Path Queries (RPQs) |
| *Graph Construct/Project |
| *Composable |

GQL



Source: Petra Selmer

Programmatic interfaces

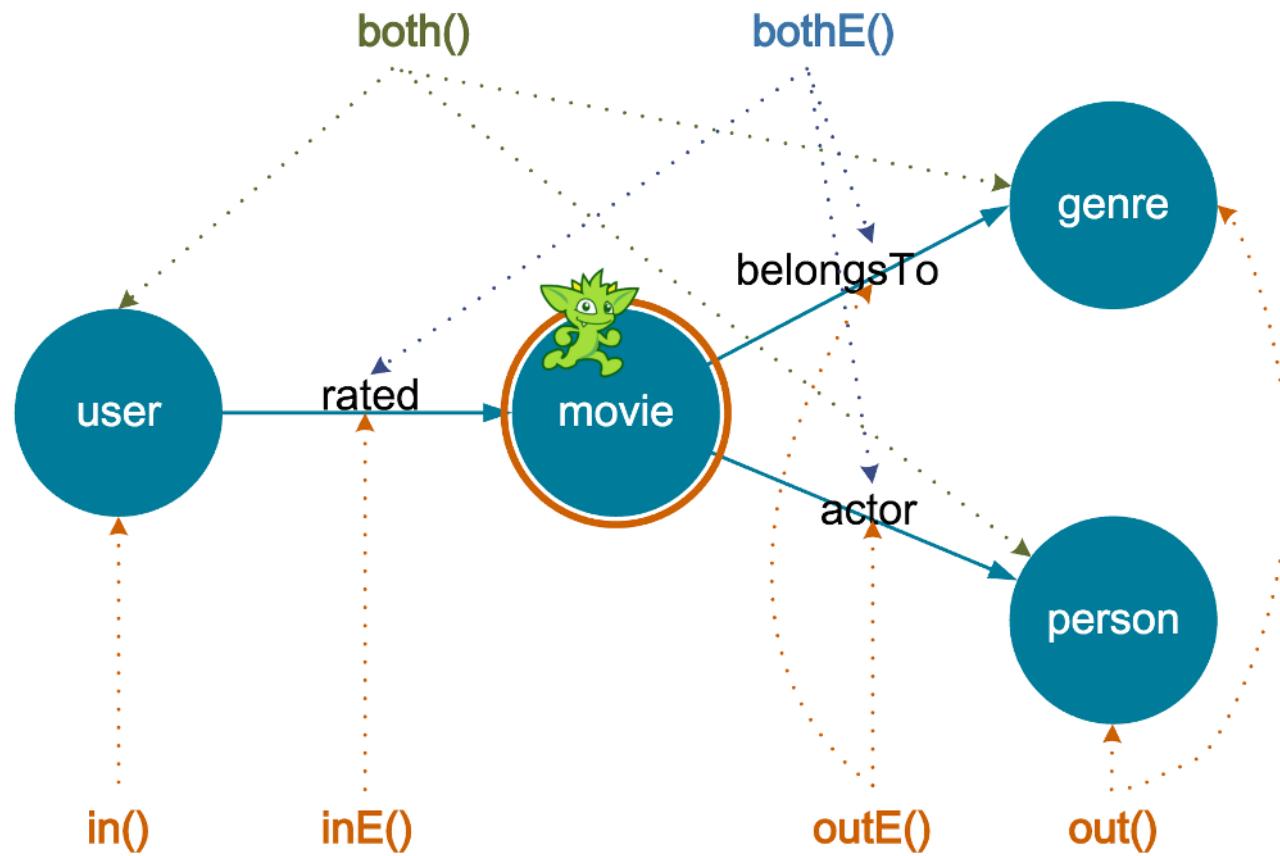
- In addition to query languages, graphs are accessed through programmatic interfaces
 - Proprietary or common APIs (mostly Tinkerpop)
 - Scripting and processing languages (Apache Gremlin, SAP GraphScript)
 - MapReduce, Spark and other processing framework

Gremlin – Queries as Graph Traversals

- Traversing means moving from one node to another along the relationship edges
 - As a node can have more than one relationship, traversal is not trivial
 - There are algorithms that try to optimise the traversal of a graph
- A graph traversal starts with a chosen node, either a specified root, or any given node
 - It can follow INCOMING or OUTGOING nodes, so go in either direction
 - Can traverse DEPTH_FIRST or BREADTH_FIRST
- Useful for asking “Who rated movie A?” or “Which movie B acted in?”

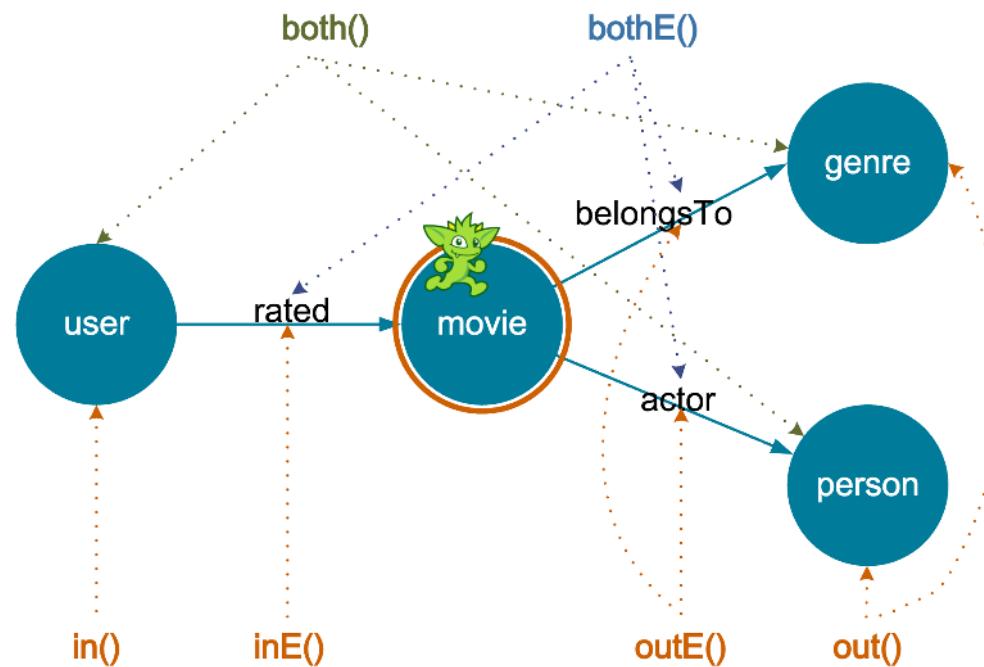
Gremlin – Simple Traversal Steps

Simple Traversal Steps

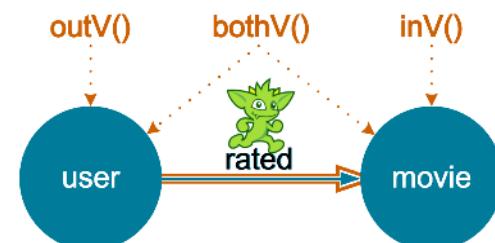


Gremlin – Simple Traversal Steps

Simple Traversal Steps

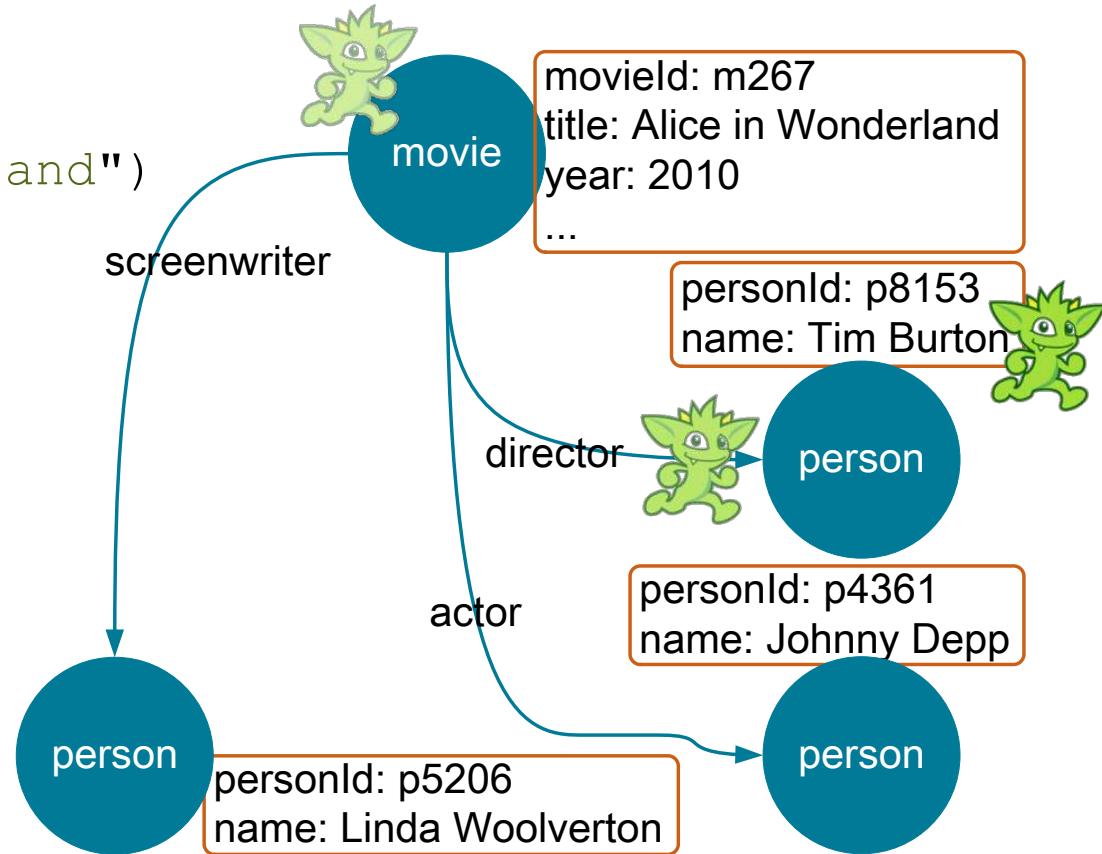


© DataStax. All rights reserved.



Gremlin – Simple Traversal Steps

```
g.V().has("title",  
          "Alice in Wonderland")  
.has("year", 2010)  
.out("director")  
.values("name")
```



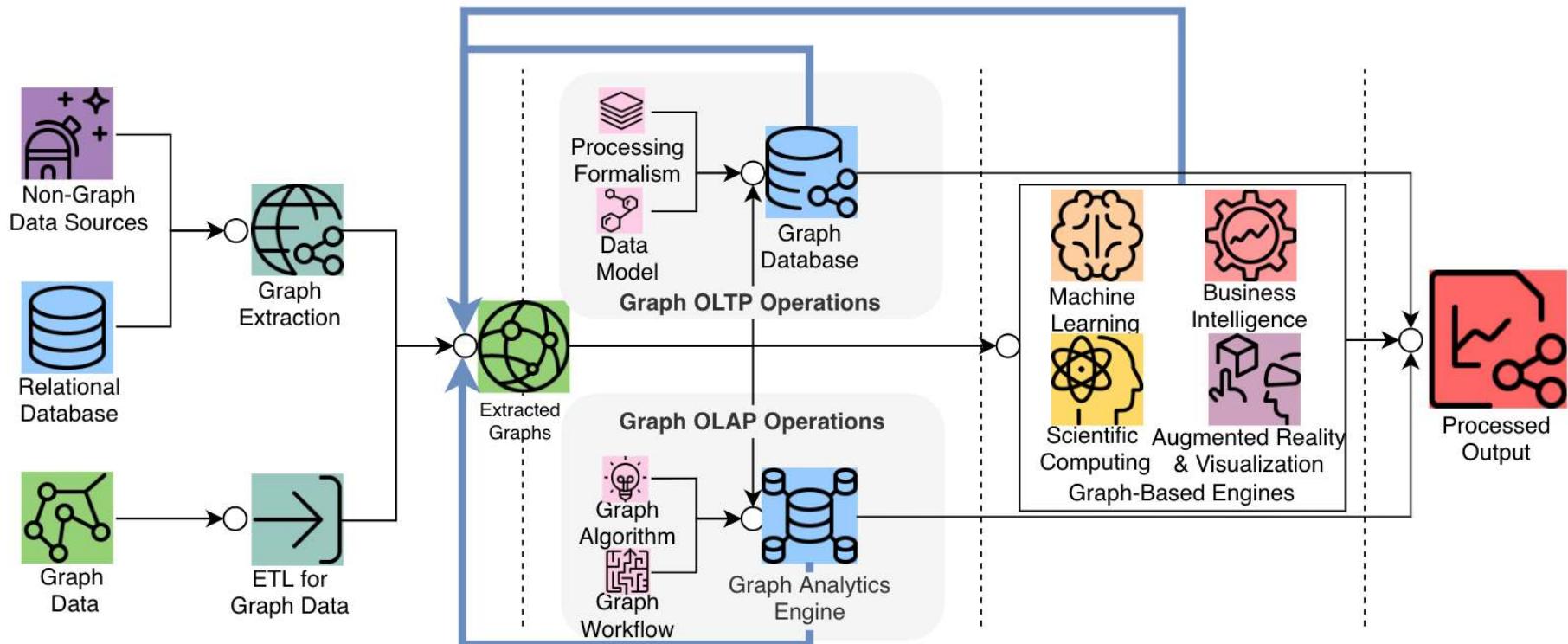
Gremlin - Graph Traversal Steps

| Lambda steps | <code>map</code> | <code>flatMap</code> | <code>filter</code> | <code>sideEffect</code> | <code>branch</code> |
|-----------------|--|--|---|--|---|
| Derived steps | <code>id, label,</code> <code>match, path,</code> <code>select, order,</code> ... | <code>coalesce, in,</code> <code>inE, inV, out,</code> ... | <code>and, coin,</code> <code>has, is, or,</code> <code>where, ...</code> | <code>aggregate,</code> <code>inject, profile,</code> <code>property,</code> <code>subgraph, ...</code> | <code>choose,</code> <code>repeat,</code> <code>union, ...</code> |
| Other steps | <code>barrier, cap, ...</code> | | | | |
| Step modulators | <code>as, by, emit, option, ...</code> | | | | |
| Predicates | <code>gt, eq, lt, neq, within, without, ...</code> | | | | |

Graph Databases and Systems

Towards a complex data ecosystem

- Several functionally different steps integrating OLTP/OLAP, from data source injection to graph database functionalities and advanced processing (ML, BI, Augmented Reality)



Operations on Graphs

- Online query processing
 - Shortest path query
 - Subgraph matching query
 - SPARQL query
- Offline graph analytics
 - PageRank
 - Community detection
- Other graph operations
 - Graph generation, visualization, interactive exploration, etc.

Interactive Workload

- Online ACID features and scalability
- System in a steady state, providing durable storage
- Updates are typically small
- Updates will conflict a small percentage of the time
- Queries typically touch a small fraction of the database

before this interval.

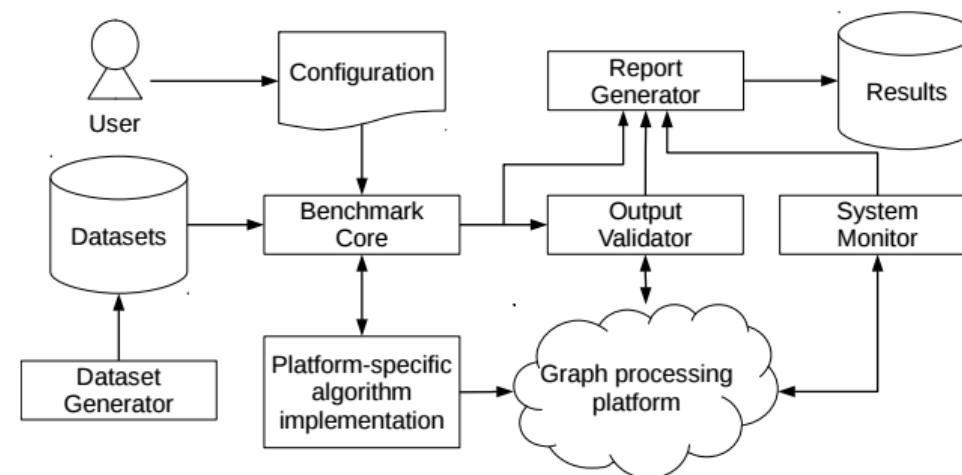
Q5. *New groups.* Given a start Person, find the top 20 Forums which that Person's friends and friends of friends became members of after a given Date. Sort results descending by the number of Posts in each Forum that were created by any of these Persons.

Business Intelligence Workload

- The workload stresses query execution and optimization
- Queries typically touch a large fraction of the data
- The queries are concurrent with trickle load
- The queries touch more data as the database grows

Graph Analytics Workload

- The analytics is done on most of the data in the graph as a single operation
- The analysis itself produces large intermediate results
- No need for isolation from possible concurrent updates



Systems

- **Graph database systems**
 - e.g. Neo4j, InfiniteGraph, DEX, Titan
- **Graph programming frameworks**
 - e.g. Giraph, Signal/Collect, Graphlab, Green Marl, Grappa
- **RDF database systems**
 - e.g. OWLIM, Virtuoso, BigData, Jena TDB, Stardog, Allegrograph
- **Relational database systems**
 - e.g. Postgres, MySQL, Oracle, DB2, SQLServer, Virtuoso, MonetDB, Vectorwise, Vertica
- **noSQL database systems**
 - e.g. HBase, REDIS, MongoDB, CouchDB, MapReduce systems like Hadoop

Workload by systems

| System | Interactive | Business Intelligence | Graph Analytics |
|------------------------------|-------------|-----------------------|--|
| Graph databases | Yes | Yes | Maybe |
| Graph programming frameworks | - | Yes | Yes |
| RDF databases | Yes | Yes | - |
| Relational databases | Maybe | Maybe | Maybe, by keeping state in temporary tables, and using the functional features of PL-SQL |
| NoSQL Key-value | Maybe | Maybe | - |

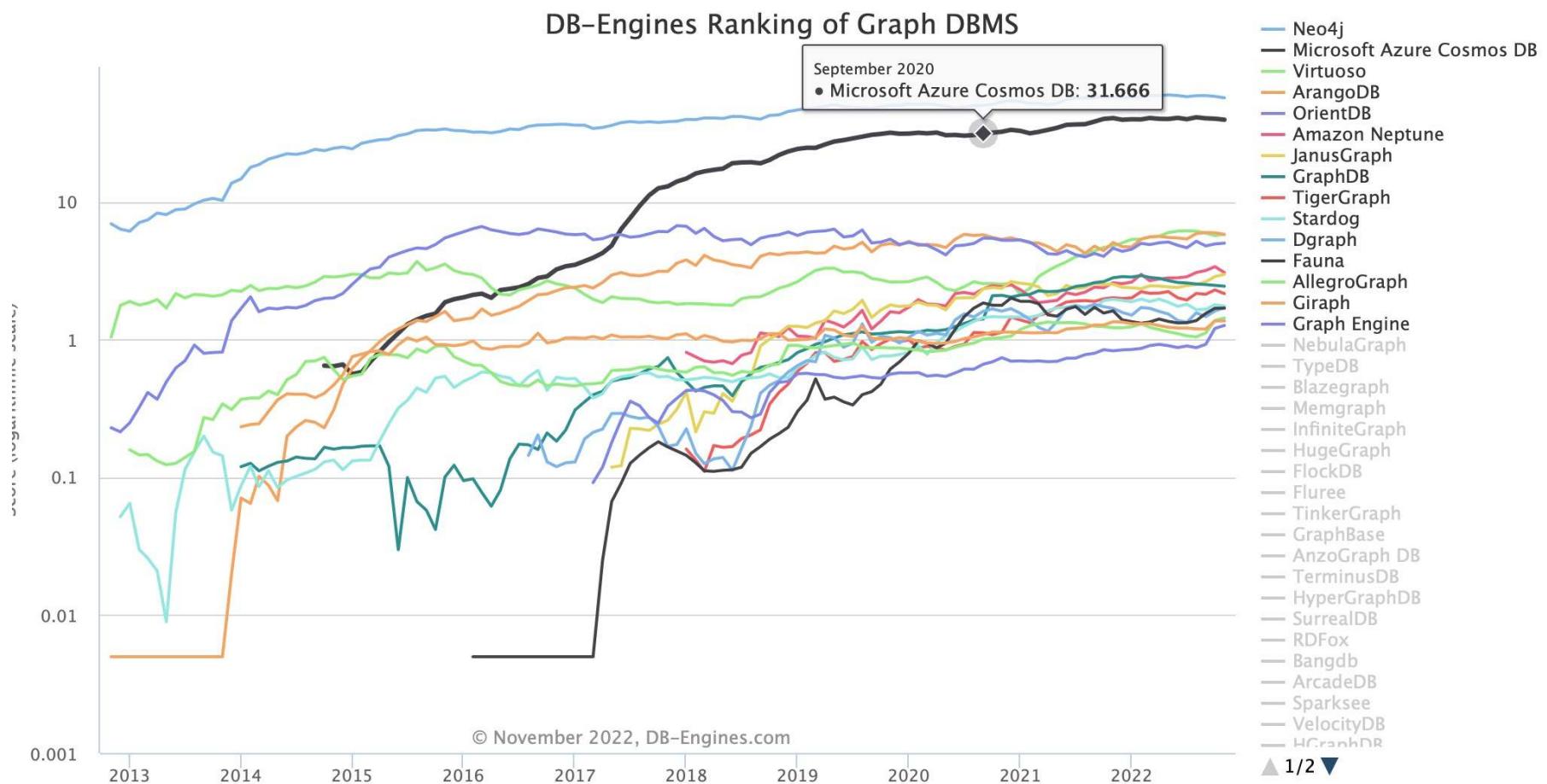
Graph Databases

1. Address need of managing graph data
2. Architecture/goals inspired by classical DBMS
3. Persistent storage of graph data
4. Transactionality
5. Closed world
6. Efficiency (over scalability)
7. () Portability (of data)
8. () Declarative query languages

| | Data model | Storage method | Query facilities | Computing model |
|-----------------------|------------------|----------------|---------------------------|-----------------|
| <i>Graph database</i> | | | | |
| AllegroGraph | ● Simple graph | ● Non-native | ● Query language ● API | Distributed |
| ArangoDB | ● Property graph | ● Native | ● ● | ● |
| Bitsy | ● Hypergraph | ● | ● | ● |
| Cayley | ● Nested graph | ● | ● | ● ● |
| FlockDB | ● | ● | ● | ● |
| GraphBase | ● | ● | ● ● | ● |
| Graphd | ● | ● | ● | ● |
| Horton | ● | ● | ● | ● |
| HyperGraphDB | ● | ● | ● ● | ● ● |
| IBM System G | ● | ● | ● | ● ● |
| imGraph | ● | ● | ● | ● |
| InfiniteGraph | ● | ● | ● ● ● | ● ● |
| InfoGrid | ● | ● | ● ● | ● ● |
| Neo4j | ● | ● | ● ● ● ● | ● |
| OrientDB | ● | ● | ● ● ● | ● |
| Sparksee/DEX | ● | ● | ● ● | ● |
| Titan | ● | ● | ● ● | ● ● |
| Trinity | ● ● | ● | ● ● ● | ● |
| TurboGraph | ● | ● | ● ● ● | ● |

Graph database engine

- The number of graph engines is growing over the years as well as their popularity



Graph Processing Frameworks

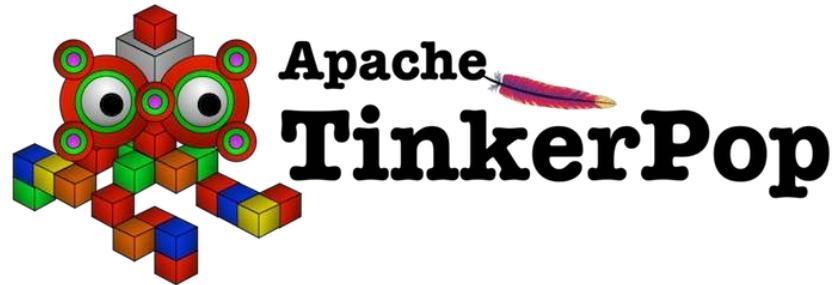
- (Offline Graph Analytical Systems)
 1. Batch processing
 2. Analysis of large graphs
 3. Facilities for graph analytical algorithms
 4. Distributed environment
 5. Multiple machines
 6. API or programming as user access

Graph Processing Frameworks

- Pregel
- ApacheGiraph
- GraphLab
- CatchtheWind
- GPS
- Mizan
- PowerGraph
- GraphX
- TurboGraph
- GraphChi

Spark Graph Fames

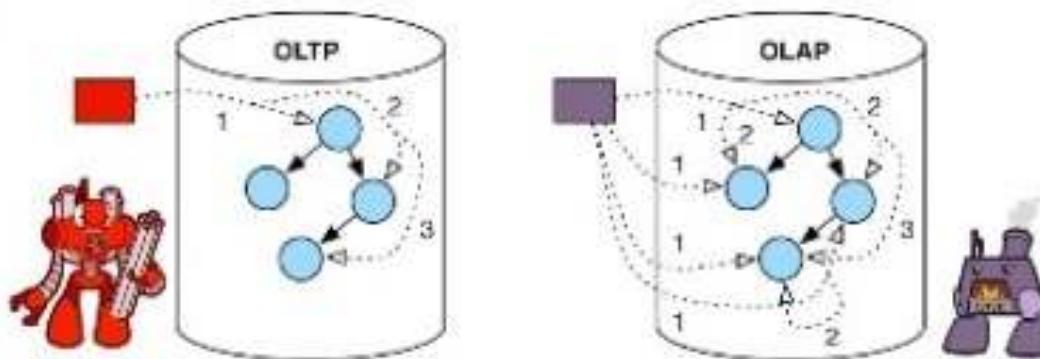
Interoperability Apache Tinkerpop



T
p
S
P

Vendor

- DataStax DSE Graph
- Azure Cosmos DB
- Neo4j
- OrientDB
- Stardog



Open Source

- Apache S2Graph
- HGraphDB
- JanusGraph
- TinkerGraph
- UniPop

Open Source (OLAP)

- Apache Giraph
- Apache Spark

To sum up ...

- Graph data management has a bright future
- One size does not fit all: Need different GDB for small, medium and web scale and for different workloads
- Very diverse use cases, standardization attempts for query languages ongoing
- What matters in a graph is its **topology** [if you do not need it, stay in the relational world]
- Need better interoperation between relational (tables) and graph data: GQL
- Interoperability among graph systems: Apache Tinkerpop

References

- Angles, Renzo, and Claudio Gutierrez. "An introduction to graph data management." *Graph Data Management*. Springer, Cham, 2018. 1-32.
- <https://arxiv.org/abs/1801.00036>

Slides Credits

- Introduction to Graph Data Management, Claudio Gutierrez, EDBT Summer School 2015
- Peter Boncz, Graph Benchmarking, EDBT Summer School 2015

THE ROLE OF SEMANTICS IN DATA MANAGEMENT

AN INTRODUCTION TO SEMANTIC WEB, ONTOLOGIES & LINKED DATA

Outline

- From the Web to the Semantic Web (a.k.a. the Web of Data)
 - An a-priori semantics for data
- Ontologies
- Semantic Web approach
- Semantic Web technology
- Linked (Open) Data

Let's start from the Web (1.0)

- The Web is a system of interlinked documents, accessed via Internet
- You can view Web pages and use hyperlinks to navigate between them
- People can easily access any of these documents
- This is the largest source of information ever
- *This is Web 1.0 or the Read-only Web*

Then, we moved to Web 2.0

- Authors started to use Ajax
- People started to share multimedia content (photos and videos)
- People started to interact on social networks
- People started to publish content in blogs
- People started to contribute to wikis
- People started to use tags and RSS
- *This is Web 2.0 or the Read-Write Web*

Web 1.0

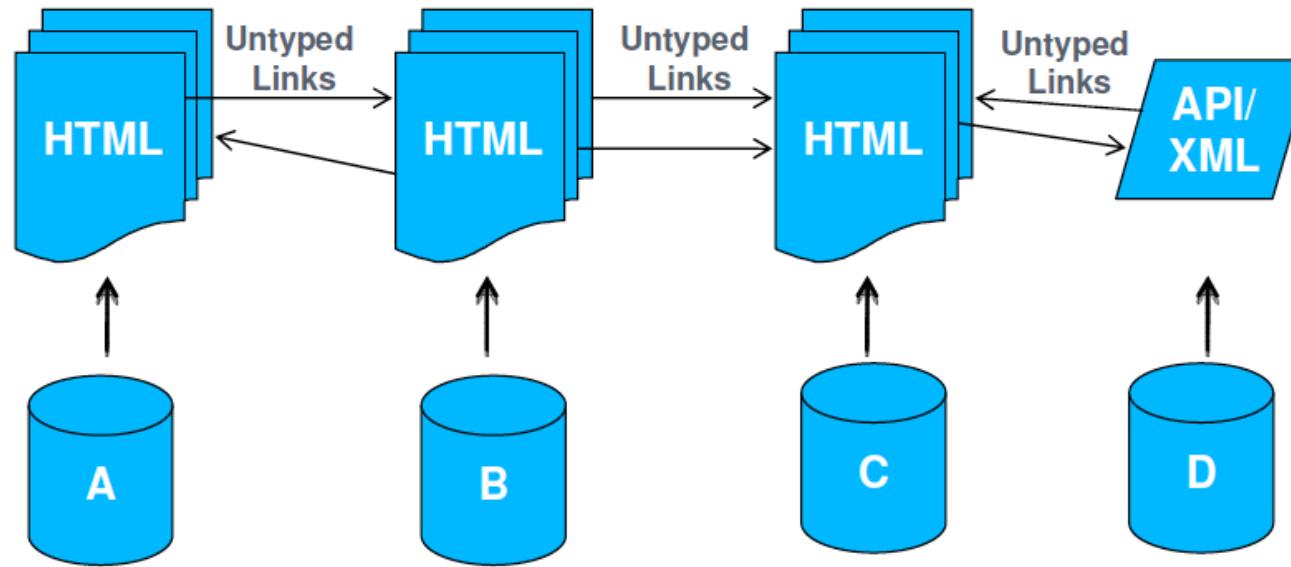
blog.aysoon.com



Web 1.0 and Web 2.0



Web of Documents



Primary objects: **documents**

Links between **documents** (or parts of them)

Degree of structure in data: fairly **low**

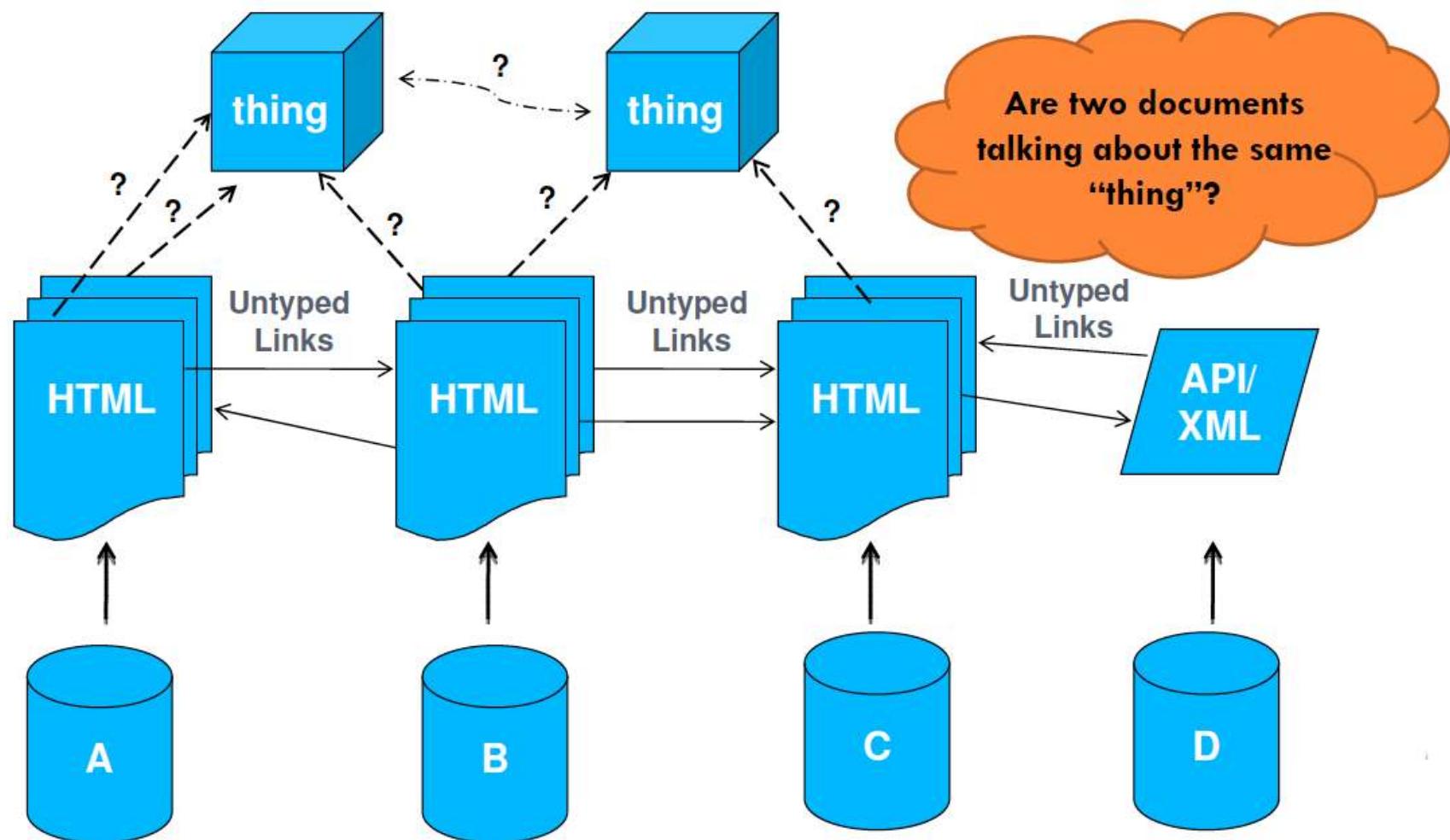
Implicit semantics of contents

Designed for: **human consumption**

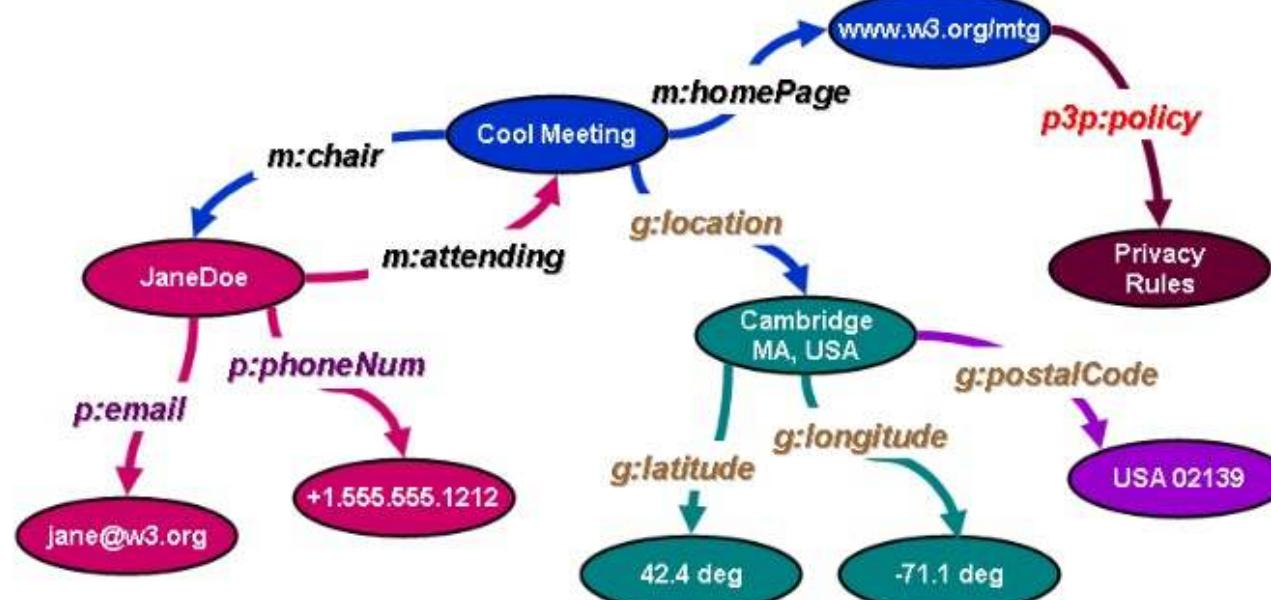
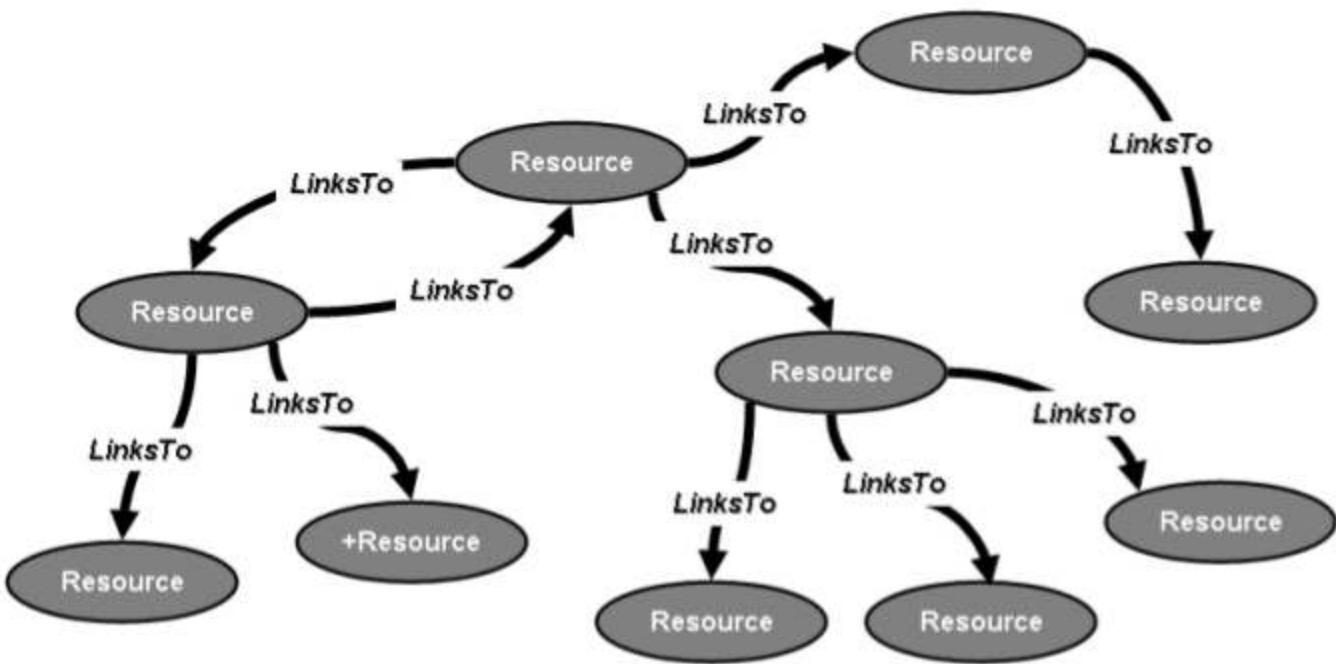
The problem

- Web pages are written in HTML
- HTML describes the *structure* of information
- HTML describes the *syntax* not the *semantics*
- if computers can understand the meaning behind information
 - they can learn what we are interested in
 - they can help us in better finding what we want
- we need to describe the *semantics* and find precise answers to complex questions
- this is what the *Semantic Web* is about
- this is Web 3.0:

The problem



the Web is about documents



the Semantic Web is about things:

- it can **recognize** entities
- it can **understand** the relationship between things

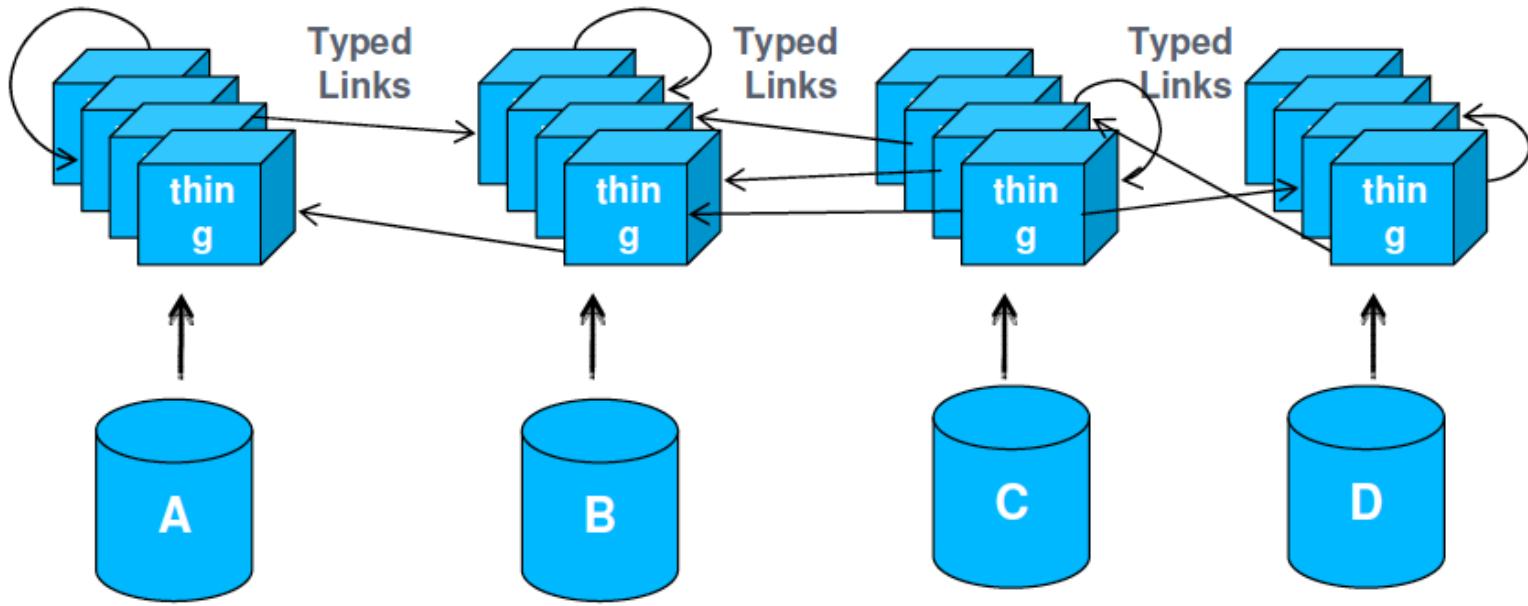
How

- Ex-post approach
 - information analysis
 - web scraping
 - natural language processing → see NLP course
- Expensive!
 - human intervention
 - hard to maintain

How

- A priori approach
 - embedding semantic annotations into data
 - we need a *data model* for describing «things» (resources) and their interrelations
 - we need some languages to
 - *specify things and their interrelations* and
 - *perform reasoning* about them

Semantic Web a.k.a. Web of Data



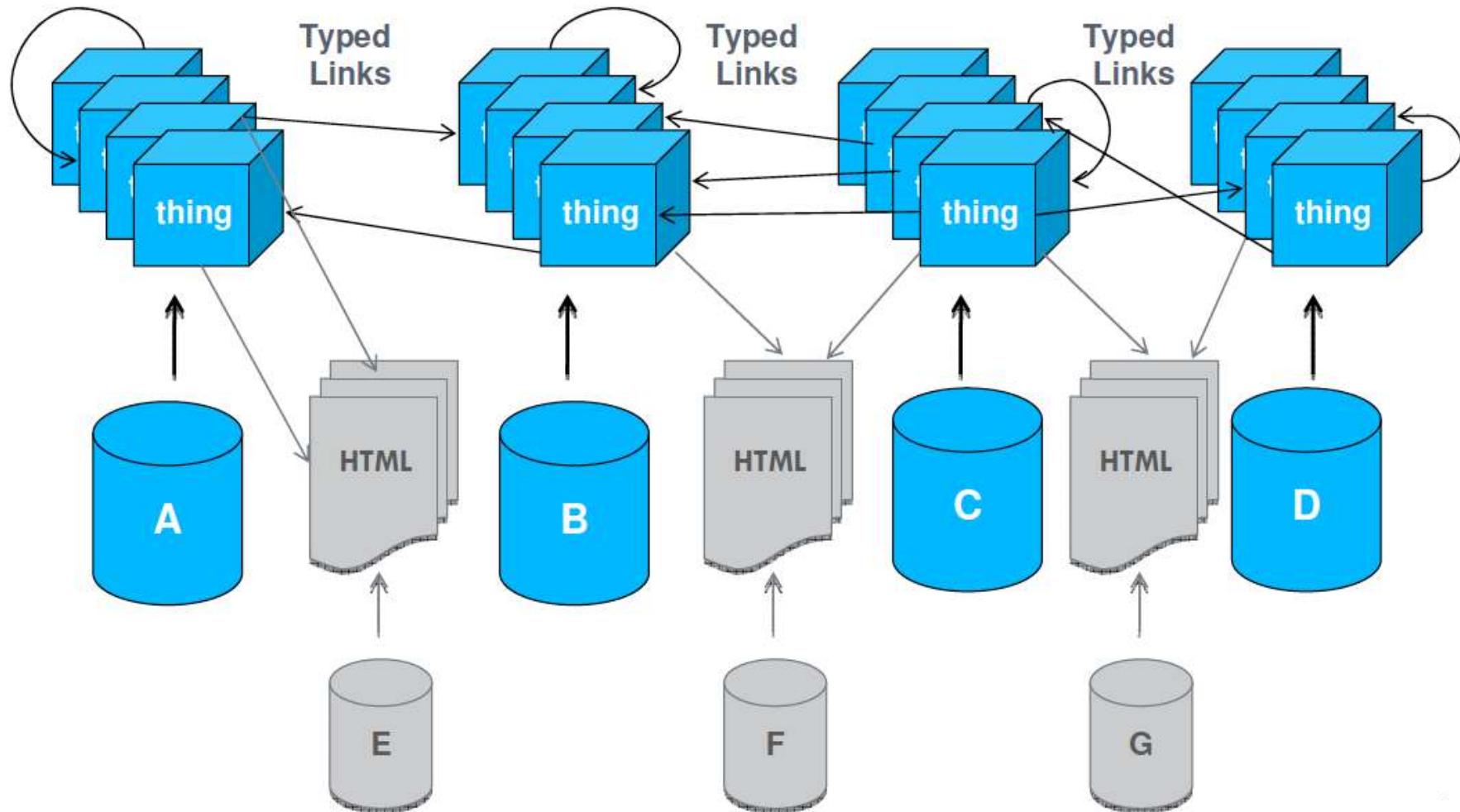
Primary objects: “**things**” (or description of things)
Links between “**things**”

Degree of Structure: **High**

Explicit semantics of contents **and** links

Designed for: Both **machines and humans**

Semantic Web a.k.a. Web of Data



An example

*Find me genes involved in signal transduction that
are related to pyramidal neurons.*

A general search

223,000 hits, 0 results

Google pyramidal neurons signal transduction [Search](#) [Advanced Search Preferences](#) [New! View and manage your Google account](#)

[Web](#) [Books](#) Results 1 - 10 of about 223,000 for [pyramidal neurons signal transduction](#) (0)

Book results for pyramidal neurons signal transduction

 [Cerebral Signal Transduction](#) - by Maarten Eduard Anton Reith - 440 pages
[Neuroprotective Signal Transduction](#) - by Mark Paul. Mattson - 347 pages
[Toxins And Signal Transduction](#) - by Yehuda Gutman, Phillip Lazarovici - 520 pages

Neurotrophin-3 and brain-derived neurotrophic factor activate ...
... and brain-derived neurotrophic factor activate multiple signal transduction events but are not survival factors for hippocampal pyramidal neurons. ...
www.ncbi.nlm.nih.gov/UniPub/IOP/pm/646092.html?pmid=8752100 - 12k -
[Cached](#) - [Similar pages](#) - [Note this](#)

K+ channel regulation of signal propagation in dendrites of ...
Pyramidal neurons receive tens of thousands of synaptic inputs on their dendrites. ...
Signal Transduction* Substances Potassium Channel Blockers ...
www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=9202119&dopt=Abstract - [Similar pages](#) - [Note this](#)

Dopamine modulates inwardly rectifying potassium currents in ...
Using outside-out patches of mPFC pyramidal neurons, which preclude involvement of ...
Signal Transduction/drug effects Signal Transduction/physiology ...
www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=PubMed&list_uids=15044547&dopt=Abstract - [Similar pages](#) - [Note this](#)
[More results from www.ncbi.nlm.nih.gov]

Loss of Hippocampal CA3 Pyramidal Neurons in Mice Lacking STAM1 ...
Loss of Hippocampal CA3 Pyramidal Neurons in Mice Lacking STAM1 ... and to be involved in the regulation of intracellular signal transduction mediated by ...
May 12, 2009 mcb.asm.org/cgi/content/abstract/21/11/3807 - [Similar pages](#) - [Note this](#)

Domain-limited search

2,580 potential results

A service of the National Library of Medicine
and the National Institutes of Health

My NCBI 
[Sign In](#) | [Register](#)

PubMed Nucleotide Protein Genome Structure OMIM PMC Journals Books

for signal transduction pyramidal neurons Go Clear Save Search

Limits Preview/Index History Clipboard Details

Display Summary Show 20 Sort By Send to

All: 2580 Review: 160 

Items 1 - 20 of 2580 Page 1 of 129 Next

1: [Naimark A, Barkai E, Mater MA, Kaplan Z, Kozlovska N, Cohen H.](#) Related Articles, Links
Upregulation of neurotrophic factors selectively in frontal cortex in response to olfactory discrimination learning.
Neural Plast. 2007;:13427.
PMID: 17710248 [PubMed - in process]

2: [Nistico R, Piccirilli S, Sebastianelli L, Nistico G, Bernardi G, Mercuri NB.](#) Related Articles, Links
The blockade of K(+)-ATP channels has neuroprotective effects in an in vitro model of brain ischemia.
Int Rev Neurobiol. 2007;82:383-95.
PMID: 17678973 [PubMed - indexed for MEDLINE]

3: [Schmidt-Hieber C, Jonas P, Bischofberger J.](#) Related Articles, Links
Subthreshold dendritic signal processing and coincidence detection in dentate gyrus granule cells.
J Neurosci. 2007 Aug 1;27(31):8430-41.
PMID: 17670990 [PubMed - indexed for MEDLINE]

4: [Alvarez VA, Ridenour DA, Sahutti BL.](#) Related Articles, Links
Distinct structural and ionotropic roles of NMDA receptors in controlling spine and synapse stability.
J Neurosci. 2007 Jul 11;27(28):7365-76.
PMID: 17626197 [PubMed - indexed for MEDLINE]

5: [Smith SS, Gong QH.](#) Related Articles, Links

Specific databases

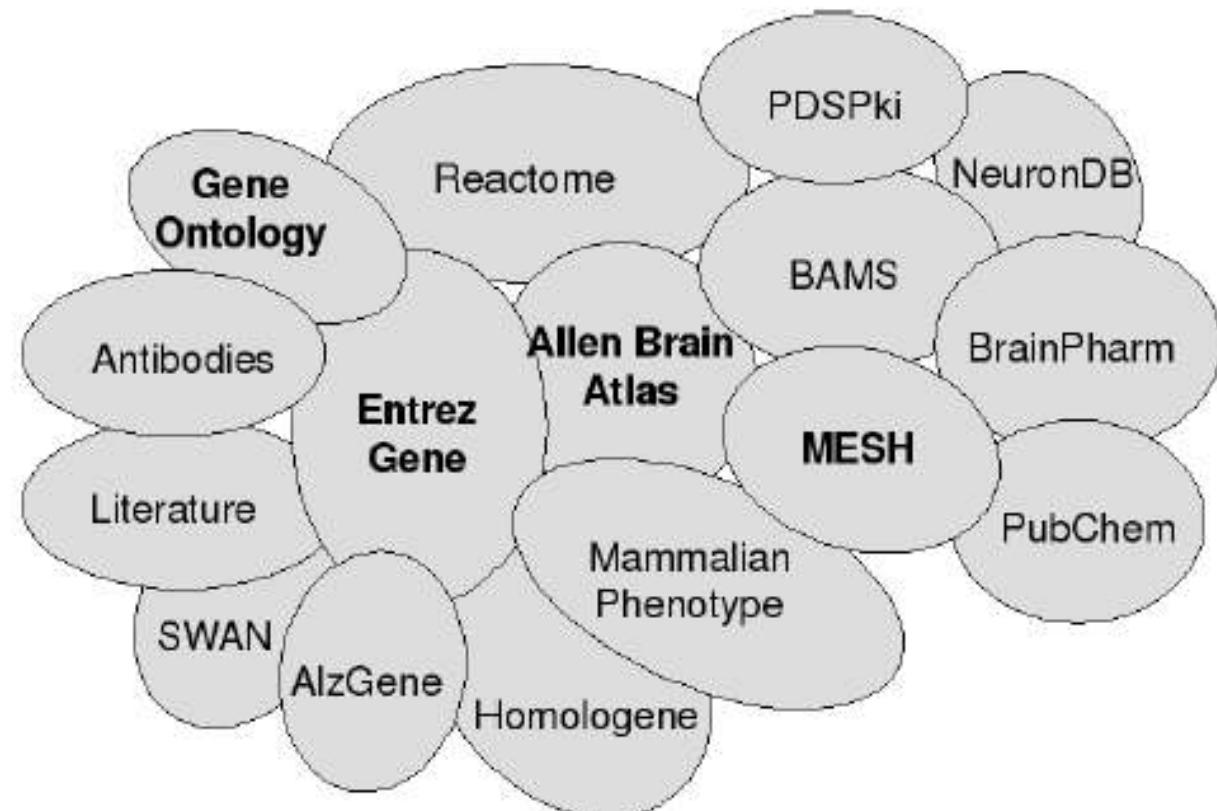
Too many silos!

| | | |
|------|---|---|
| 2580 |  PubMed: biomedical literature citations and abstracts |  |
| 959 |  PubMed Central: free, full text journal articles |  |
| none |  Site Search: NCBI web and FTP sites |  |
| 2 |  Books: online books |  |
| 4 |  OMIM: online Mendelian Inheritance in Man |  |
| none |  OMIA: online Mendelian Inheritance in Animals |  |
| 10 |  CoreNucleotide: Core subset of nucleotide sequence records |  |
| none |  EST: Expressed Sequence Tag records |  |
| 34 |  GSS: Genome Survey Sequence records |  |
| 16 |  Protein: sequence database |  |
| 5 |  Genome: whole genome sequences |  |
| none |  Structure: three-dimensional macromolecular structures |  |
| none |  Taxonomy: organisms in GenBank |  |
| none |  SNP: single nucleotide polymorphism |  |
| 35 |  Gene: gene-centered information |  |
| 10 |  HomoloGene: eukaryotic homology groups |  |
| none |  PubChem Compound: unique small molecule chemical structures |  |
| none |  PubChem Substance: deposited chemical substance records |  |
| none |  Genome Project: genome project information |  |
| 2 |  dbGaP: genotype and phenotype |  |
| none |  UniGene: gene-oriented clusters of transcript sequences |  |
| none |  CDD: conserved protein domain database |  |
| none |  3D Domains: domains from Entrez Structure |  |
| none |  UniSTS: markers and mapping data |  |
| none |  PopSet: population study data sets |  |
| none |  GEO Profiles: expression and molecular abundance profiles |  |
| none |  GEO DataSets: experimental sets of GEO data |  |
| none |  Cancer Chromosomes: cytogenetic databases |  |
| none |  PubChem BioAssay: bioactivity screens of chemical substances |  |
| none |  GENSAT: gene expression atlas of mouse central nervous system |  |
| none |  Probes: sequence-specific reagents |  |
| 1394 |  Protein Clusters: a collection of related protein sequences |  |

A Semantic Web approach

Integrate disparate databases...

- MeSH
- PubMed
- Entrez Gene
- Gene Ontology
- ...



A Semantic Web Approach

...so that *one* query...

```
prefix go: <http://purl.org/obo/owl/GO#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
prefix owl: <http://www.w3.org/2002/07/owl#>
prefix mesh: <http://purl.org/commons/record/mesh/>
prefix sc: <http://purl.org/science/owl/sciencecommons/>
prefix ro: <http://www.obisfoundry.org/ro/ro.owl#>
prefix senselab: <http://purl.org/ycmi/senselab/neuron_ontology.owl#>

SELECT ?genename ?processname ?receptor_protein_name
WHERE {
  # PubMesh includes ?gene_records mentioned in ?articles which are identified by pmid in ?pubmed_records.
  GRAPH <http://purl.org/commons/hds/pubmesh> {
    ?pubmed_record sc:has-as-minor-mesh mesh:D017966 .
    ?article sc:identified_by_pmid ?pubmed_record .
    ?gene_record sc:describes_gene_or_gene_product_mentioned_by ?article
  }
  # The Gene Ontology asserts that foreach ?protein, ?protein ro:has_function | ro:realized_as ?process .
  GRAPH <http://purl.org/commons/hds/goa> {
    ?protein rdfs:subClassOf ?restriction1 .
    ?restriction1 owl:onProperty ro:has_function .
    ?restriction1 owl:someValuesFrom ?restriction2 .
    ?restriction2 owl:onProperty ro:realized_as .
    ?restriction2 owl:someValuesFrom ?process .
  }
  # Also, foreach ?protein, ?protein has a parent class which is linked by some predicate to ?gene_record.
  ?protein rdfs:subClassOf ?protein_superclass .
  ?protein_superclass owl:equivalentClass ?restriction3 .
  ?restriction3 owl:onProperty sc:is_protein_gene_product_of_dna_described_by .
  ?restriction3 owl:hasValue ?gene_record .
  # Each ?process (that we are interested in) is a subclass of the signal transduction process.
  GRAPH <http://purl.org/commons/hds/20070416/classrelations> {
    { ?process obo:part_of go:GO_0007166 }
    UNION
    { ?process rdfs:subClassOf go:GO_0007166 }
  }
  GRAPH <http://purl.org/commons/hds/gene> {
    ?gene_record rdfs:label ?genename
  }
  GRAPH <http://purl.org/commons/hds/20070416> {
    ?process rdfs:label ?processname
  }
}
```

Mesh: Pyramidal Neurons

Pubmed: Journal Articles

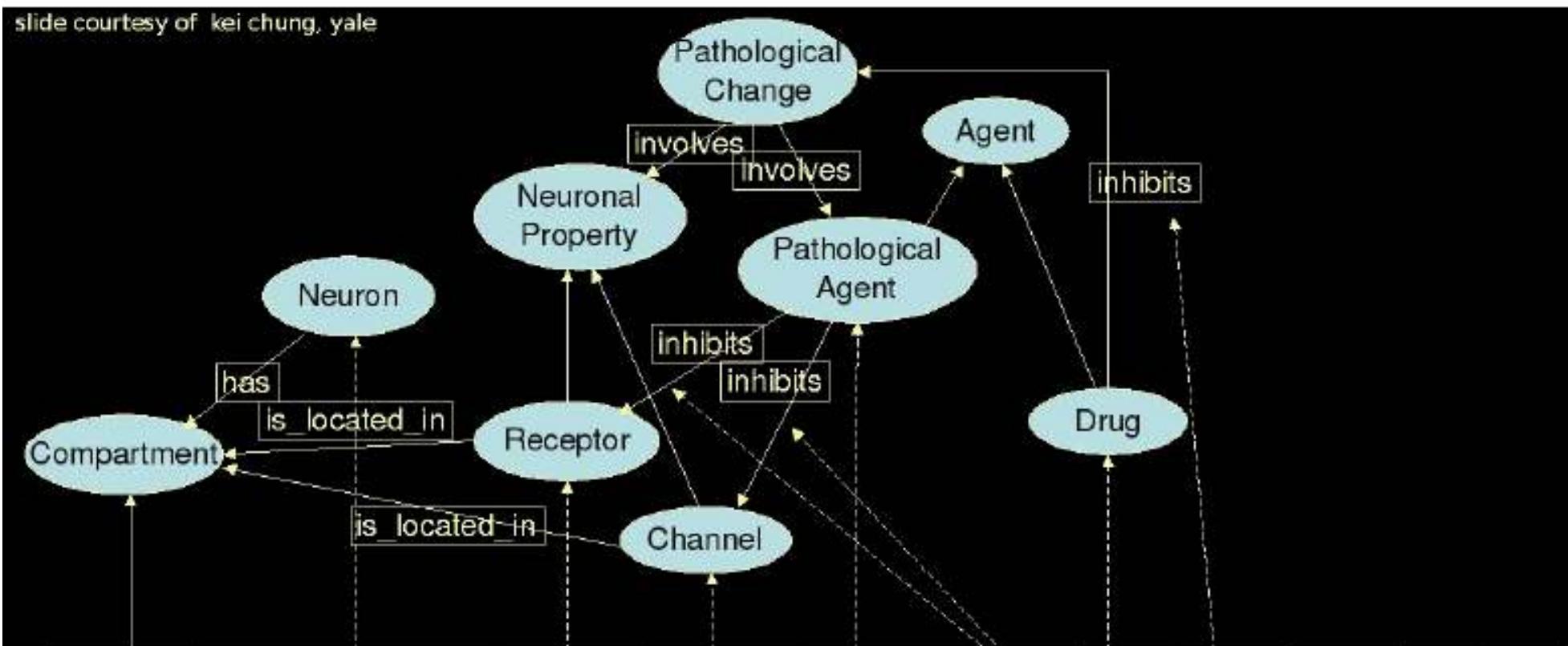
Entrez Gene: Genes

GO: Signal Transduction

A Semantic Web Approach

...(trivially) spans several databases...

slide courtesy of kei chung, yale



| Compartment | Cell: NeuronDB | Receptor | Channel | Pathological Agent (PA) | PA Action | Drug | Drug Action | Stage | Note | Detail |
|--------------|----------------------------|----------|--------------------|-------------------------|-----------|------|-------------|-------|------|--------|
| Soma | CA1 pyramidal neuron | | TA | beta Amyloid | Inhibits | | | Early | View | 66240 |
| | Olfactory bulb mitral cell | Gabaa | | | | | | Early | View | 66250 |
| Dendrite | CA1 pyramidal neuron | | TA | beta Amyloid | Inhibits | | | Early | View | 66240 |
| | Olfactory bulb mitral cell | Gabaa | | | | | | Early | View | 66250 |
| Unspecified | Oncocyte | | TA, high threshold | beta Amyloid | Inhibits | | | Early | View | 66252 |
| May 12, 2009 | CA1 pyramidal neuron | | | beta Amyloid | Inhibits | | | Early | View | 66253 |
| | CA1 pyramidal neuron | NMDA | Ca Calcium | beta Amyloid | Inhibits | | Inhibits | Early | View | 66258 |

A Semantic Web Approach

...to deliver targeted results...

| | |
|---------------|---|
| DRD1, 1812 | adenylate cyclase activation |
| ADRB2, 154 | adenylate cyclase activation |
| ADRB2, 154 | arrestin mediated desensitization of G-protein coupled receptor protein signaling pathway |
| DRD1IP, 50632 | dopamine receptor signaling pathway |
| DRD1, 1812 | dopamine receptor, adenylate cyclase activating pathway |
| DRD2, 1813 | dopamine receptor, adenylate cyclase inhibiting pathway |
| GRM7, 2917 | G-protein coupled receptor protein signaling pathway |
| GNG3, 2785 | G-protein coupled receptor protein signaling pathway |
| GNG12, 55970 | G-protein coupled receptor protein signaling pathway |
| DRD2, 1813 | G-protein coupled receptor protein signaling pathway |
| ADRB2, 154 | G-protein coupled receptor protein signaling pathway |
| CALM3, 808 | G-protein coupled receptor protein signaling pathway |
| HTR2A, 3356 | G-protein coupled receptor protein signaling pathway |
| DRD1, 1812 | G-protein signaling, coupled to cyclic nucleotide second messenger |
| SSTR5, 6755 | G-protein signaling, coupled to cyclic nucleotide second messenger |
| MTNR1A, 4543 | G-protein signaling, coupled to cyclic nucleotide second messenger |
| CNR2, 1269 | G-protein signaling, coupled to cyclic nucleotide second messenger |
| HTR6, 3362 | G-protein signaling, coupled to cyclic nucleotide second messenger |
| SRIK2, 2898 | glutamate signaling pathway |
| GRIN1, 2902 | glutamate signaling pathway |
| GRIN2A, 2903 | glutamate signaling pathway |
| GRIN2B, 2904 | glutamate signaling pathway |
| ADAM10, 102 | integrin-mediated signaling pathway |
| GRM7, 2917 | negative regulation of adenylate cyclase activity |
| LRP1, 4035 | negative regulation of Wnt receptor signaling pathway |
| ADAM10, 102 | Notch receptor processing |
| ASCL1, 429 | Notch signaling pathway |
| HTR2A, 3356 | serotonin receptor signaling pathway |
| ADRB2, 154 | transmembrane receptor protein tyrosine kinase activation (dimerization) |
| PTPRG, 5793 | transmembrane receptor protein tyrosine kinase signaling pathway |
| EPHA4, 2043 | transmembrane receptor protein tyrosine kinase signaling pathway |
| NRTN, 4902 | transmembrane receptor protein tyrosine kinase signaling pathway |

What's the trick?

- Agreement on common terms and relationships
- Incremental, flexible data structure
- Good-enough modeling
- Query interface tailored to the data model
- key concept: **ontology for the Web**

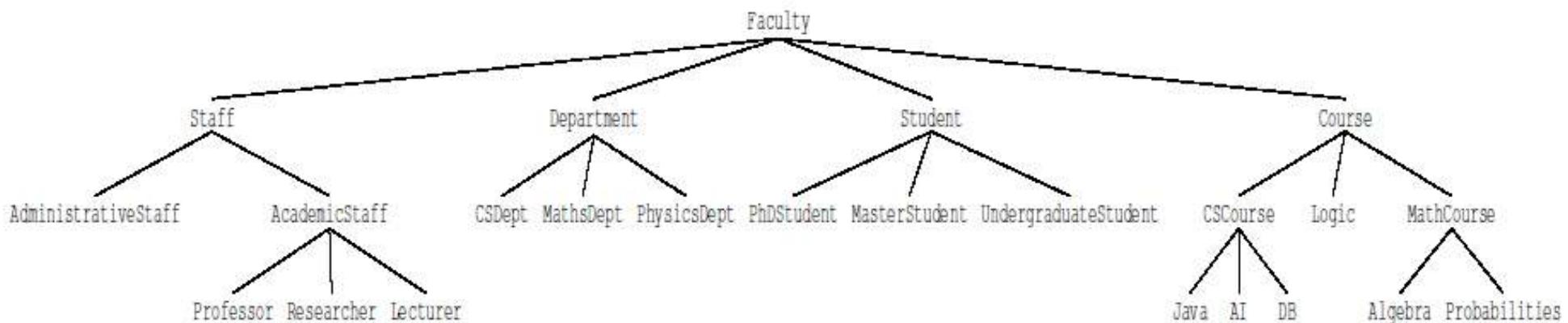
Ontologies

Ontologies

- Formal descriptions providing **human** users a shared understanding of a given domain
 - ▶ A controlled vocabulary
- Formally defined so that it can also be processed by **machines**
- **Logical semantics** that enables **reasoning**.
- Reasoning is the key for different important tasks of Web data management, in particular
 - ▶ to answer queries (over possibly distributed data)
 - ▶ to relate objects in different data sources enabling their integration
 - ▶ to detect inconsistencies or redundancies
 - ▶ to refine queries with too many answers, or to relax queries with no answer

Classes and Class Hierarchies

- Backbone of the ontology
- AcademicStaff is a **Class**
- (A class will be interpreted as a **set** of objects)
- AcademicStaff **isa** Staff
- (**isa** is interpreted as set inclusion)



Relations

- Declaration of **relations** with their **signature**
- (Relations will be interpreted as binary relations between objects)
- TeachesIn(AcademicStaff, Course)
 - ▶ if one states that “ X TeachesIn Y ”, then X belongs to AcademicStaff and Y to Course,
- TeachesTo(AcademicStaff, Student),
- Leads(Staff, Department)

Instances

- Classes have **instances**
- Dupond is an instance of the class Professor
- it corresponds to the fact: Professor(Dupond)

- Relations also have **instances**
- (Dupond,CS101) is an instance of the relation TeachesIn
- it corresponds to the fact: TeachesIn(Dupond,CS101)

- The instance statements can be seen as (and stored in) a **database**

Ontology = schema + instance

- Schema
 - ▶ The set of class and relation names
 - ▶ The **signatures** of relations and also **constraints**
 - ▶ The constraints that are used for two purposes
 - ★ checking data consistency (like dependencies in databases)
 - ★ inferring new facts
- Instance
 - ▶ The set of facts
 - ▶ The set of base facts together with the inferred facts should satisfy the constraints
- Ontology (i.e., Knowledge Base) = Schema + Instance

Reasoning

- The use of logic as ontology semantics enables reasoning
- For instance, from the following facts:

- Leads(Dupond, CS Department)
- Leads(Staff, Department)

it follows that

- Staff(Dupond)
- Department(CS Department)

The Semantic Web

- Augments the World Wide Web through the usage of specific **ontologies**
- Represents the Web's information in a machine-readable fashion
- Enables...
 - ... targeted search
 - ... data browsing
 - ... automated agents

Semantic Web technologies

- A family of technology standards that ‘play nice together’, including:
 - Flexible data model
 - Distributed query language
 - Expressive ontology language

The technologies enable us to build applications and solutions that were not possible, practical, or feasible traditionally.

The Semantic Web approach

How does a Semantic Web approach help us merge data sets, infer new relations, and integrate outside data sources?

Thanks to Ivan Herman for this example

The Semantic Web approach

- Map the various data onto an abstract data representation (**instances and schema of some ontology**)
- Make the data independent of its internal representation
- Merge the resulting representations
- Start making queries on the whole
 - Queries not possible on the individual data sets

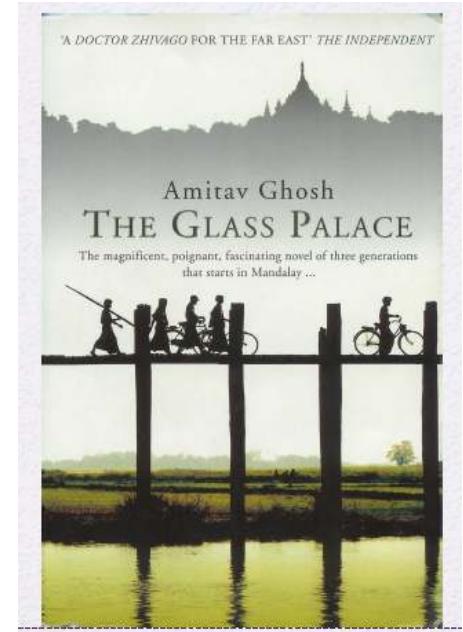
Dataset “A”: a simplified book store

Books

| ID | Author | Title | Publisher | Year |
|-------------------|--------|------------------|-----------|------|
| ISBN0-00-651409-X | id_xyz | The Glass Palace | id_qpr | 2000 |

Authors

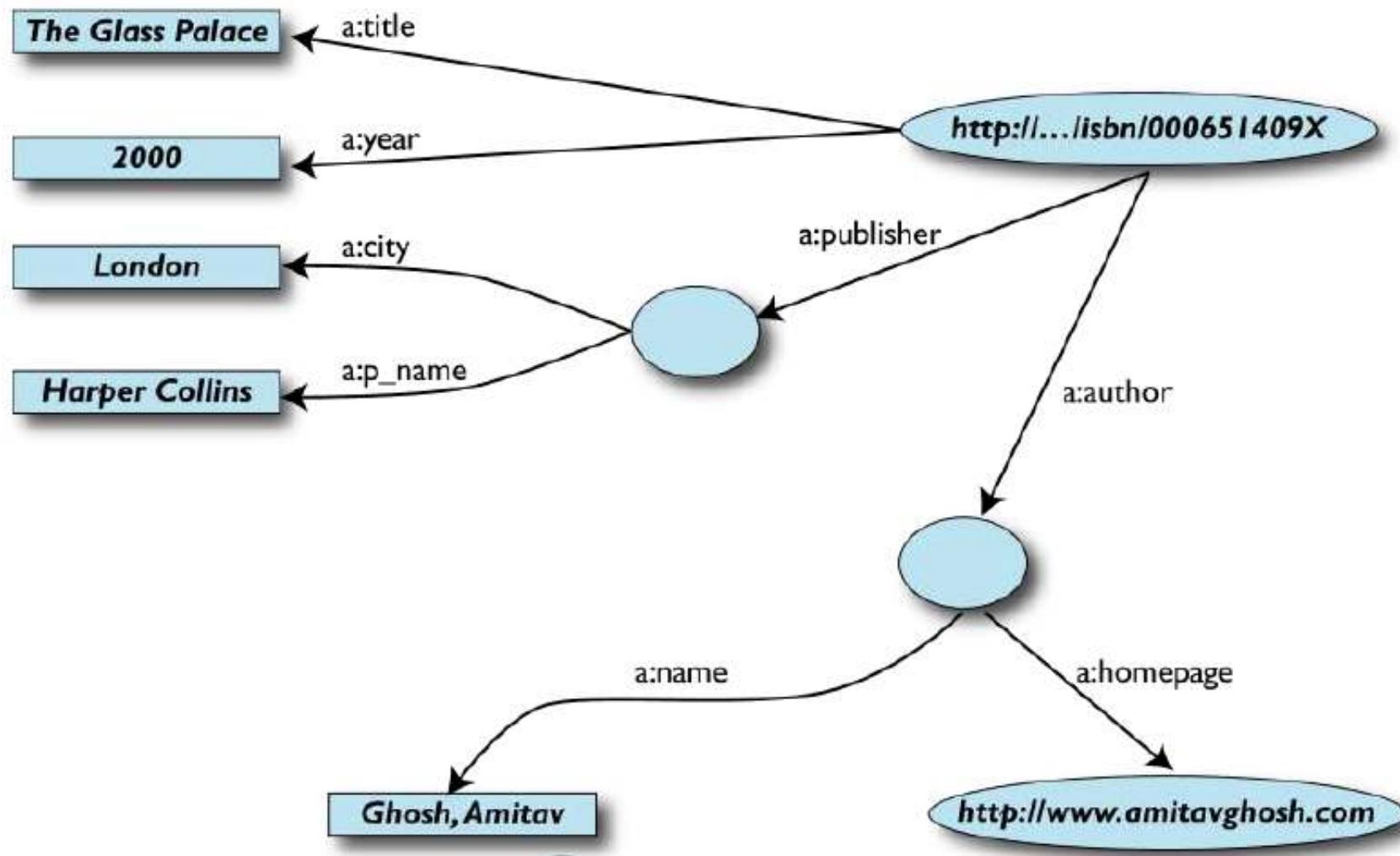
| ID | Name | Home page |
|--------|---------------|---|
| id_xyz | Ghosh, Amitav | http://www.amitavghosh.com |



Publishers

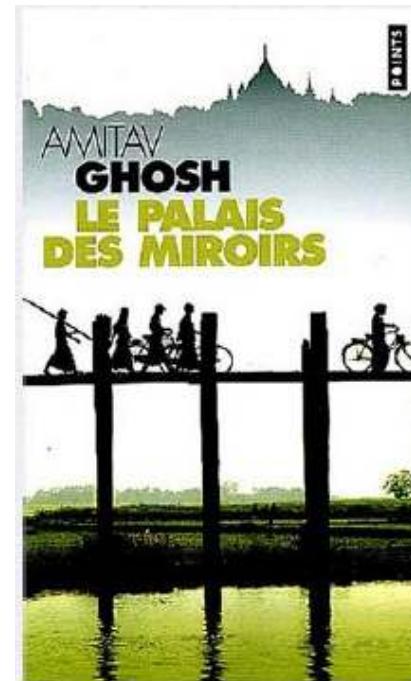
| ID | Publisher Name | City |
|--------|----------------|--------|
| id_qpr | Harper Collins | London |

1st: export your data as a set of relations among entities (ontology instance level)

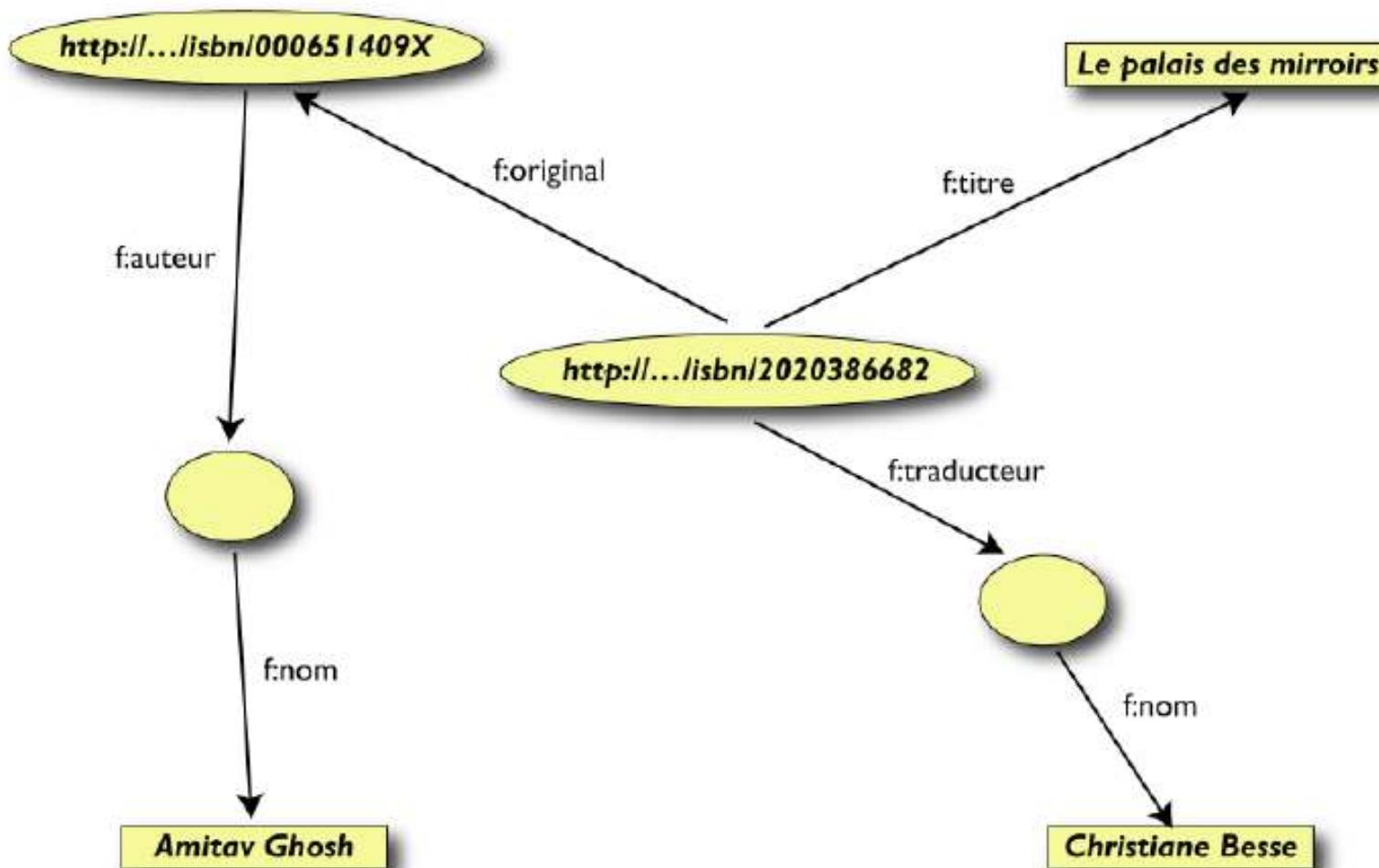


Dataset “F”: another book store

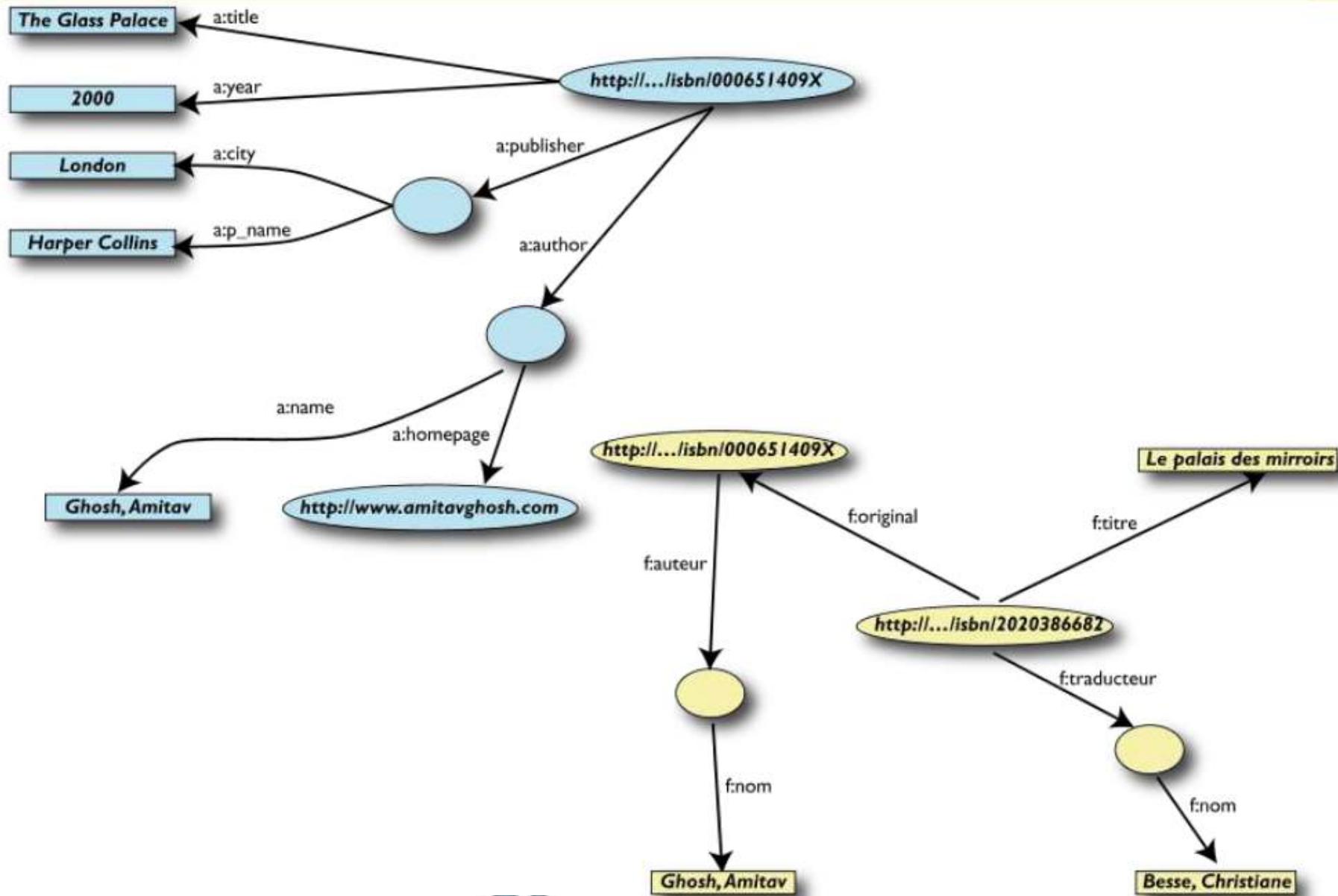
| | A | B | D | E |
|----|--------------------|-----------------------------|------------|--------------------|
| 1 | ID | Titre | Traducteur | Original |
| 2 | ISBN0 2020386682 | Le Palais des miroirs | A13 | ISBN-0-00-651409-X |
| 3 | | | | |
| 6 | ID | Auteur | | |
| 7 | ISBN-0-00-651409-X | A12 | | |
| 11 | Nom | | | |
| 12 | Ghosh, Amitav | | | |
| 13 | Besse, Christianne | | | |



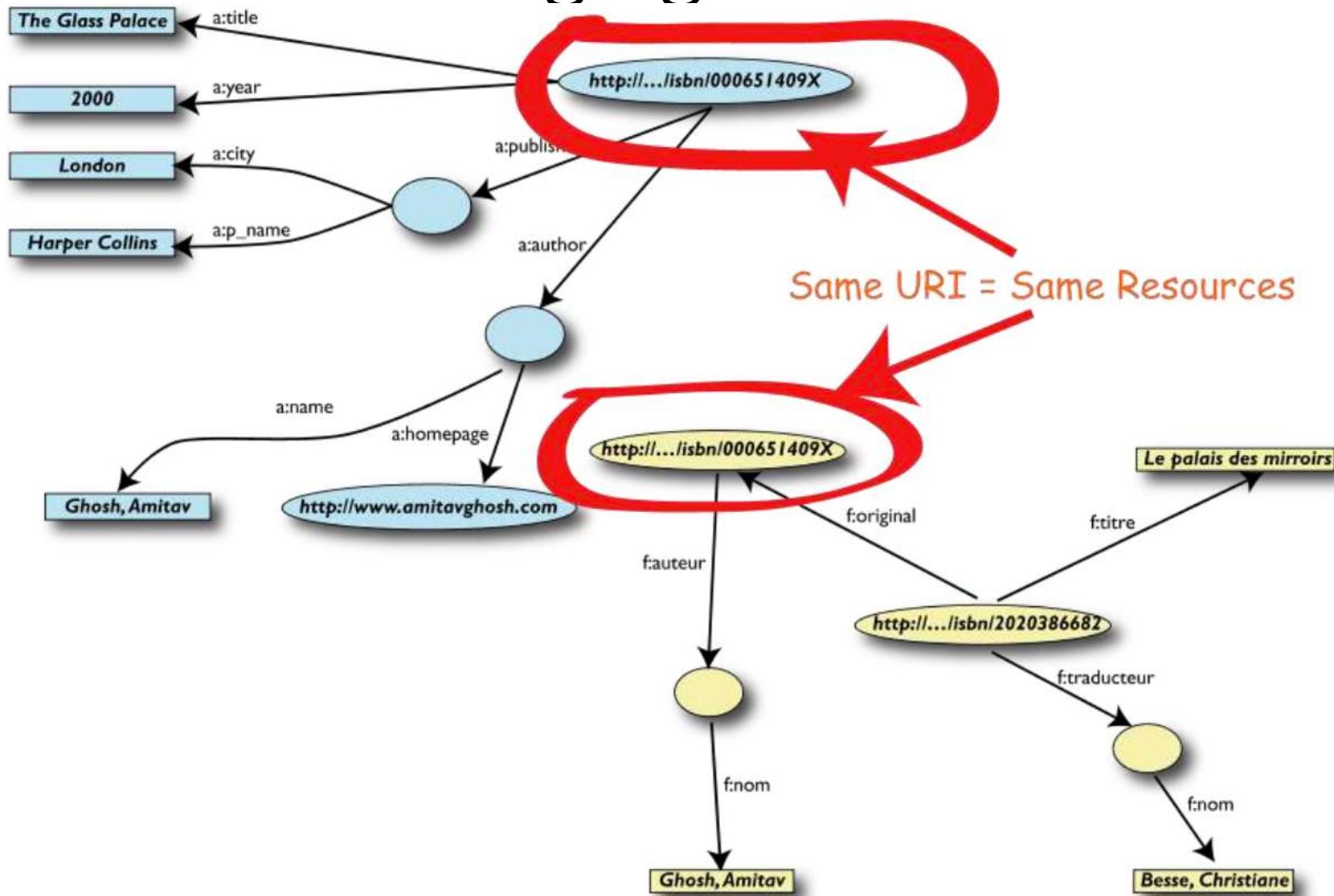
2nd: export your second set of data (ontology instance level)



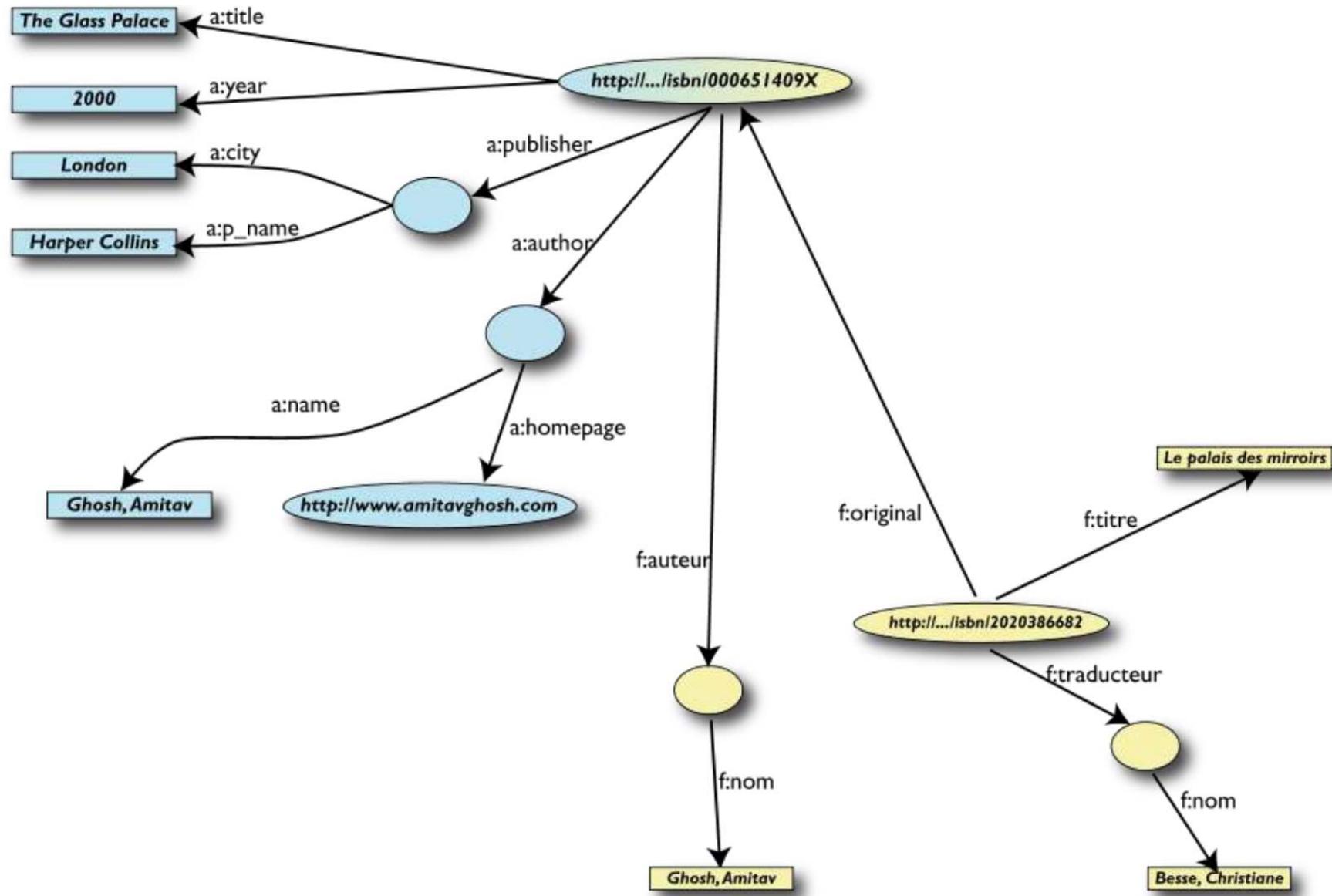
3rd: start merging data



3rd: start merging data



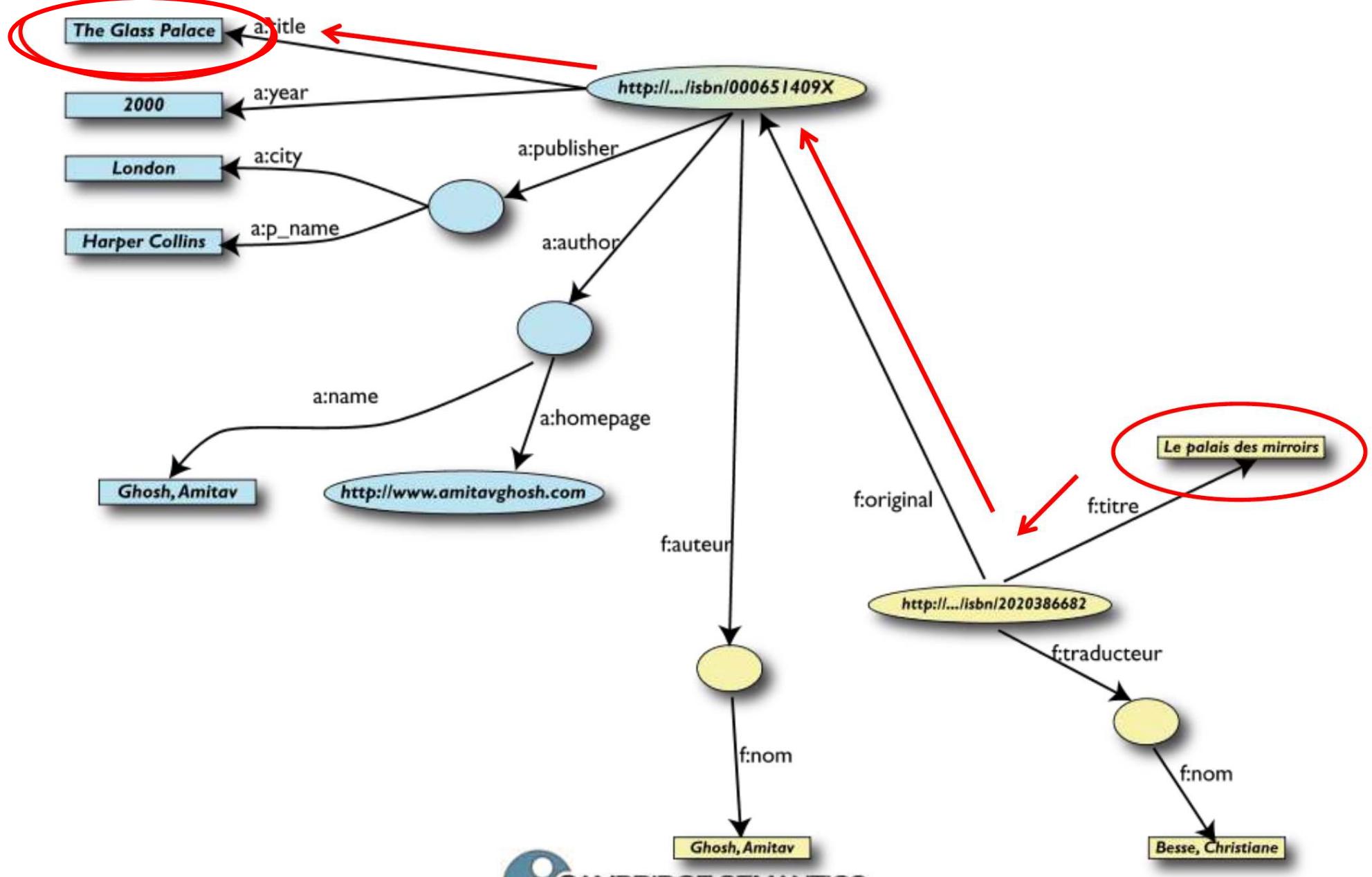
4th: merge identical resources



Start executing queries ...

- User of data set “F” can now ask queries like:
 - “What is the title of the original version of Le “Palais des miroirs?”
- This information is not in the data set “F”...
...but can be retrieved after merging with data set “A”!

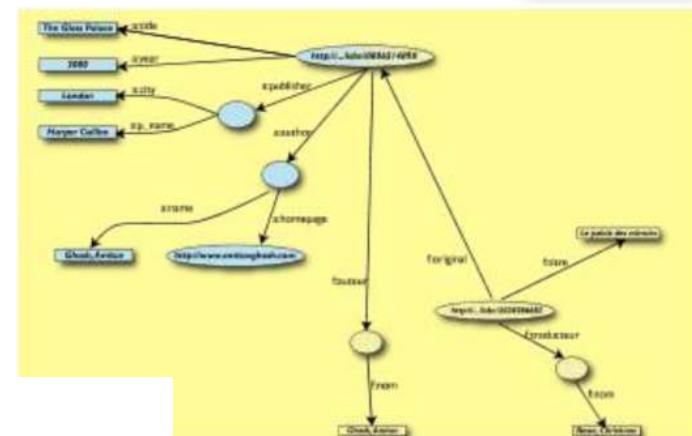
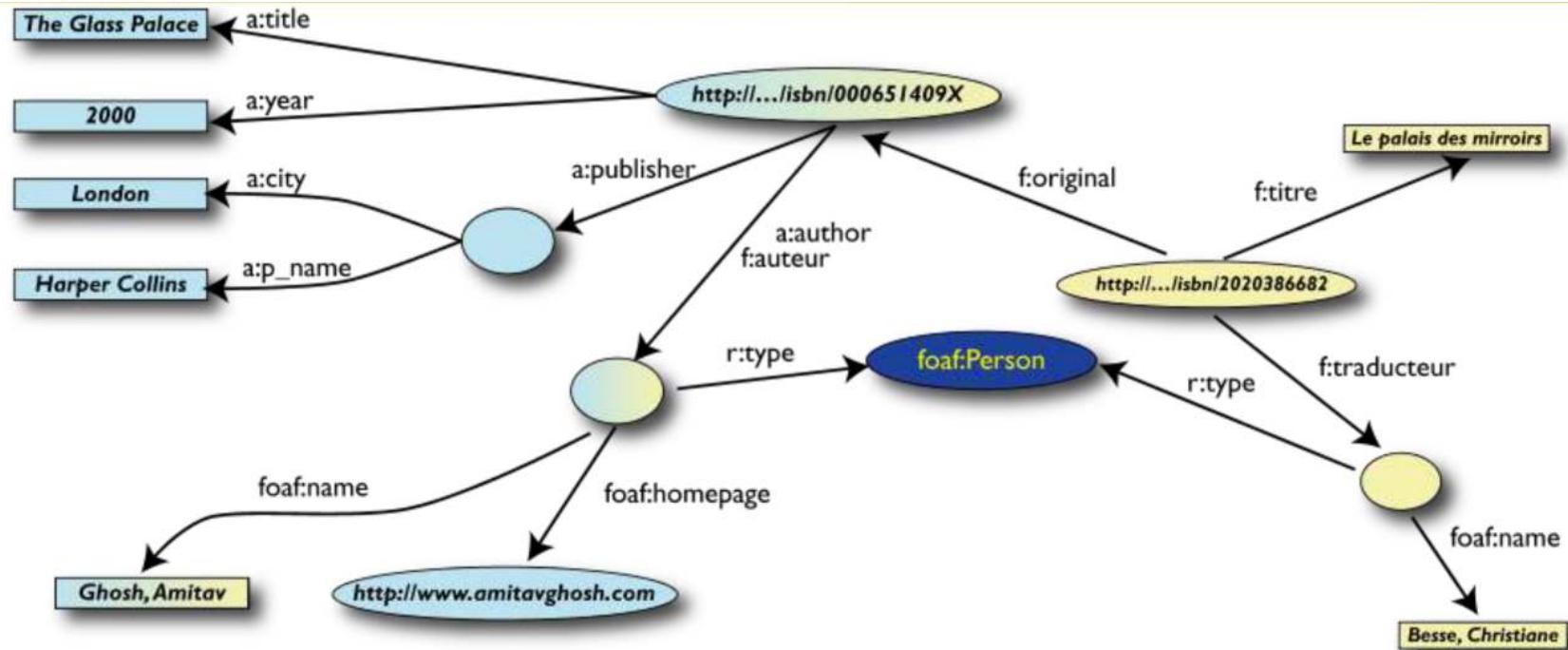
5th: query the merged dataset



More can be achieved ...

- We “know” that a:author and f:auteur are really the same
- But our automatic merge does not know that!
- Let us add some extra information to the merged data:
 - a:author is the same as f:auteur
 - Both identify a Person, a category (type) for certain resources
 - Person is an **ontology concept**

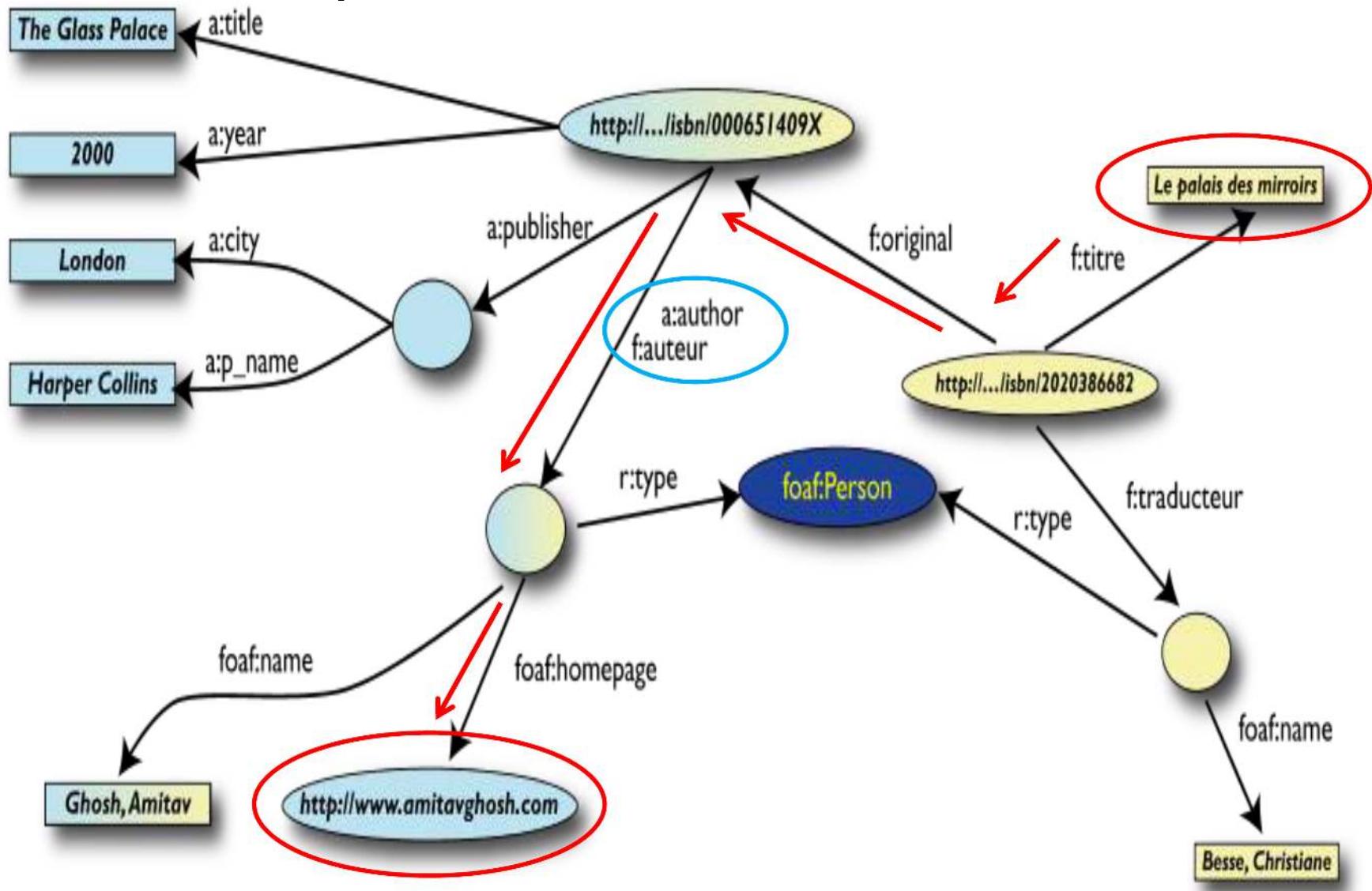
3rd revisited: use the extra knowledge (add ontology schema information)



Start making richer queries

- User of data set “F” can now query:
 - “What is the home page of Le Palais des miroirs’s ‘auteur’?”
- The information is not in data set “F” or “A”...
- ...but was made available by:
 - Merging data sets “A” and “F”
 - Adding three simple “glue” statements

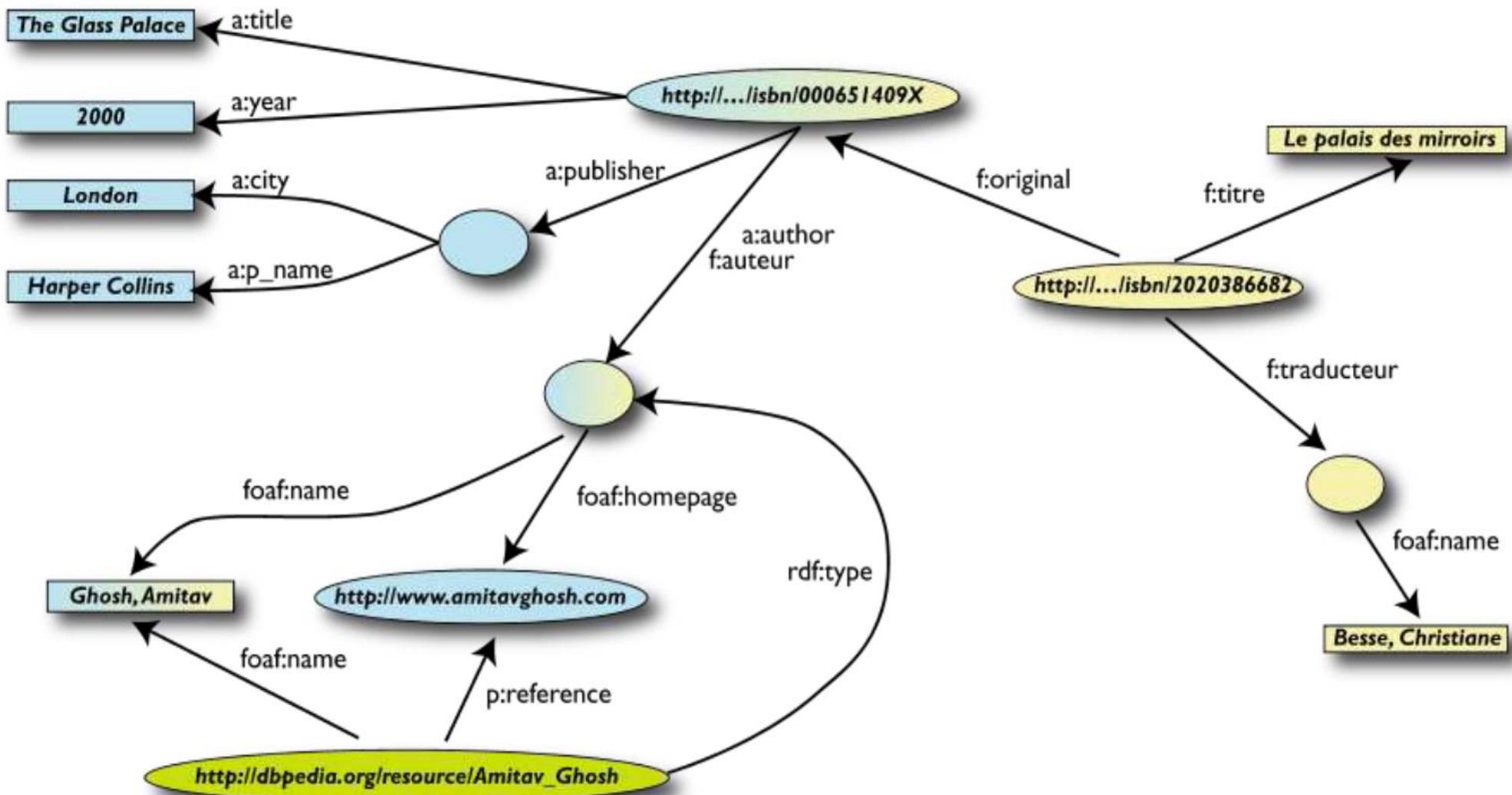
Richer queries



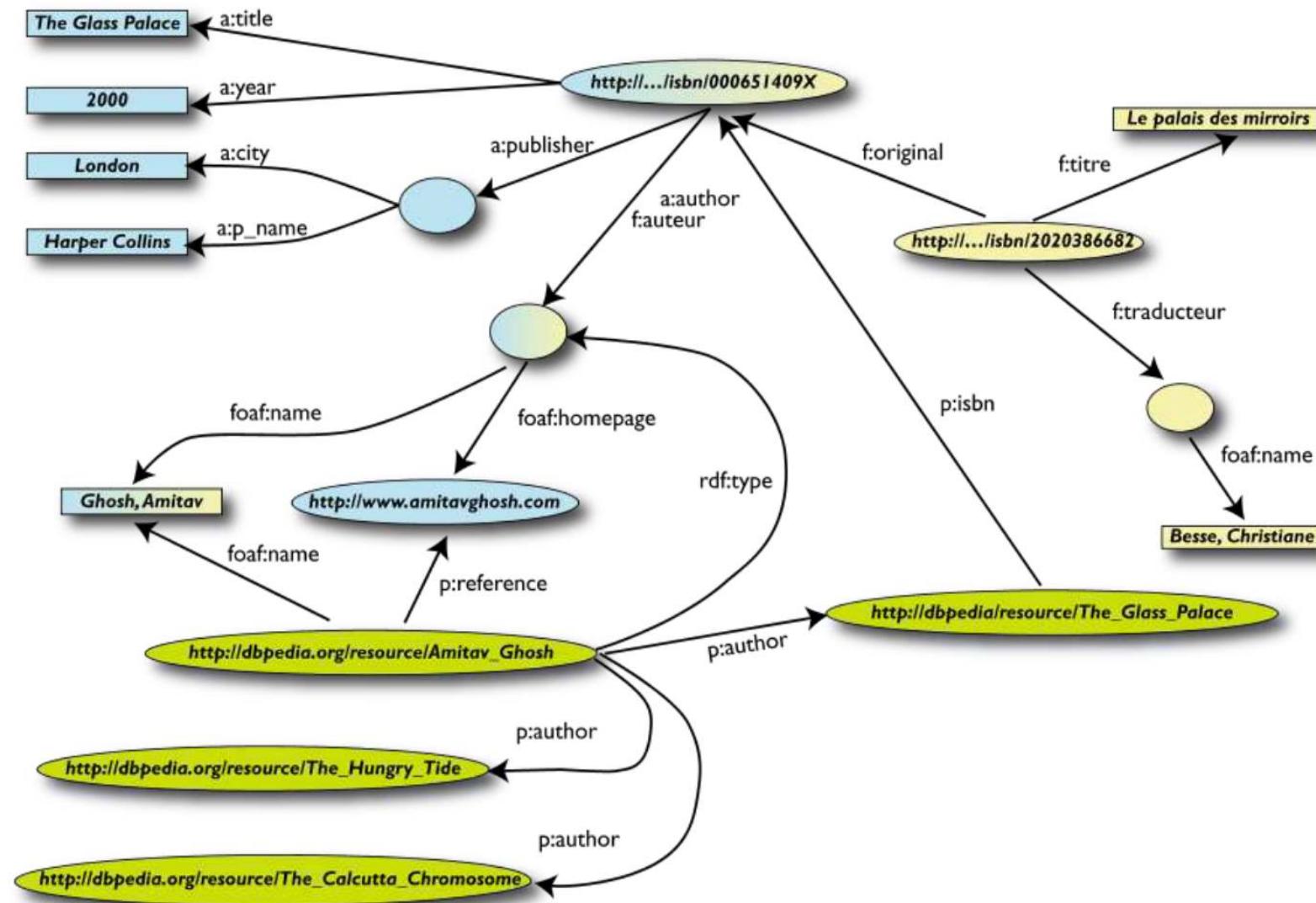
Bring in other data sources

- We can integrate new information into our merged data set from other sources
 - e.g. additional information about author Amitav Ghosh
- Perhaps the largest public source of general knowledge is Wikipedia
- Structured data can be extracted from Wikipedia using dedicated tools (leading to the dbpedia data source)

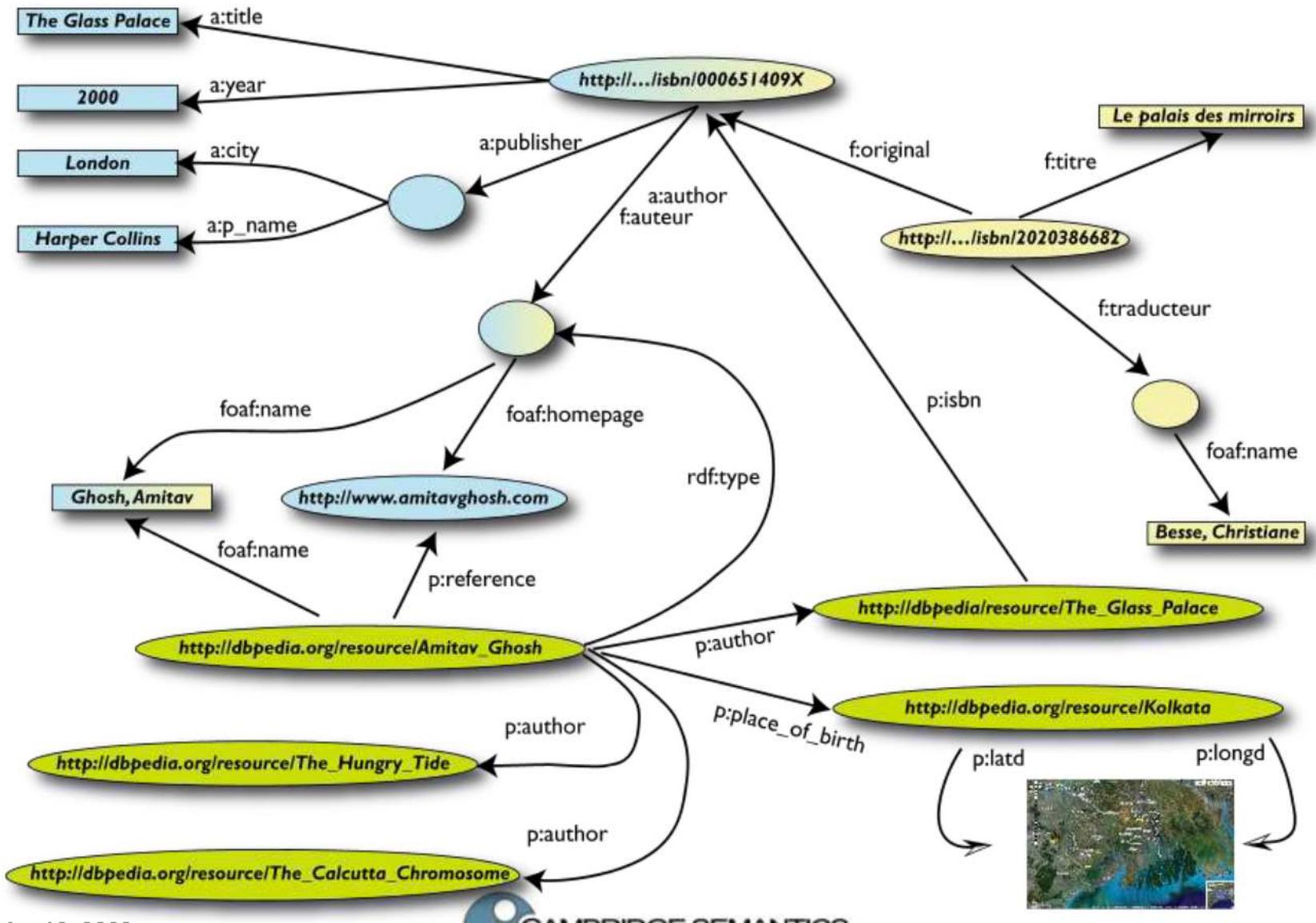
7th: merge with Wikipedia data



7th: merge with Wikipedia data



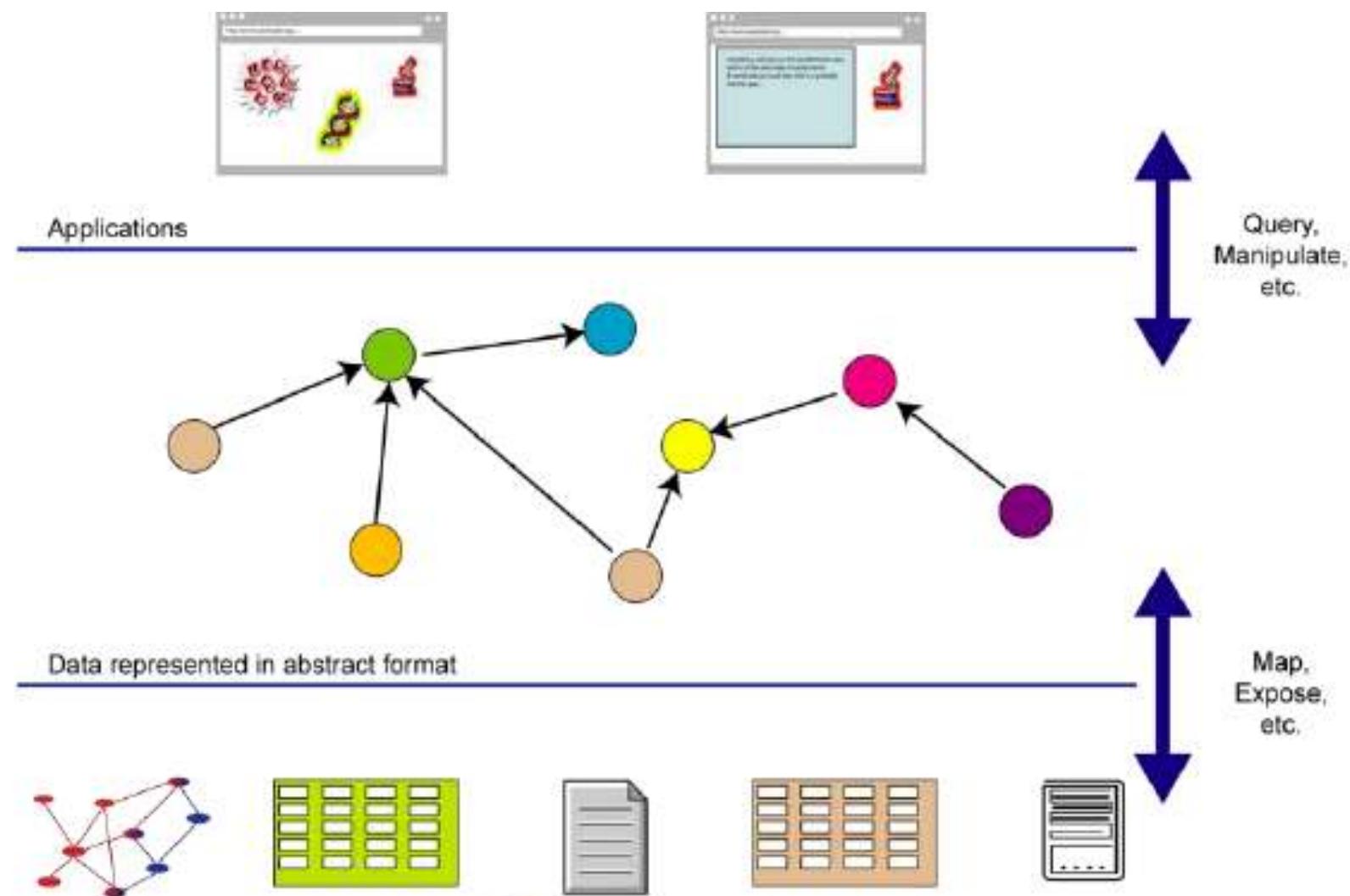
7th: merge with Wikipedia data



What did we do?

- We combined different data sets that
 - ...may be internal or somewhere on the Web
 - ...are of different formats (RDBMS, Excel spreadsheet, (X)HTML, etc)
 - ...have different names for the same relations
- We could combine the data because some identifiers were identical i.e. the ISBNs in this case (i.e., they correspond to the same ontology instance)
- We could add some simple additional information (the “glue”) to help further merge data sets (through new ontology instances, ontology relations, ontology concepts)
- The result? Answer queries that could not previously be asked

What did we do?



The abstraction pays off because ...

- ...the graph representation is independent of the details of the native structures
- ...a change in local database schemas, HTML structures, etc. do not affect the whole “schema independence”
- ...new data, new connections can be added seamlessly & incrementally

Semantic Web technologies make such integration possible

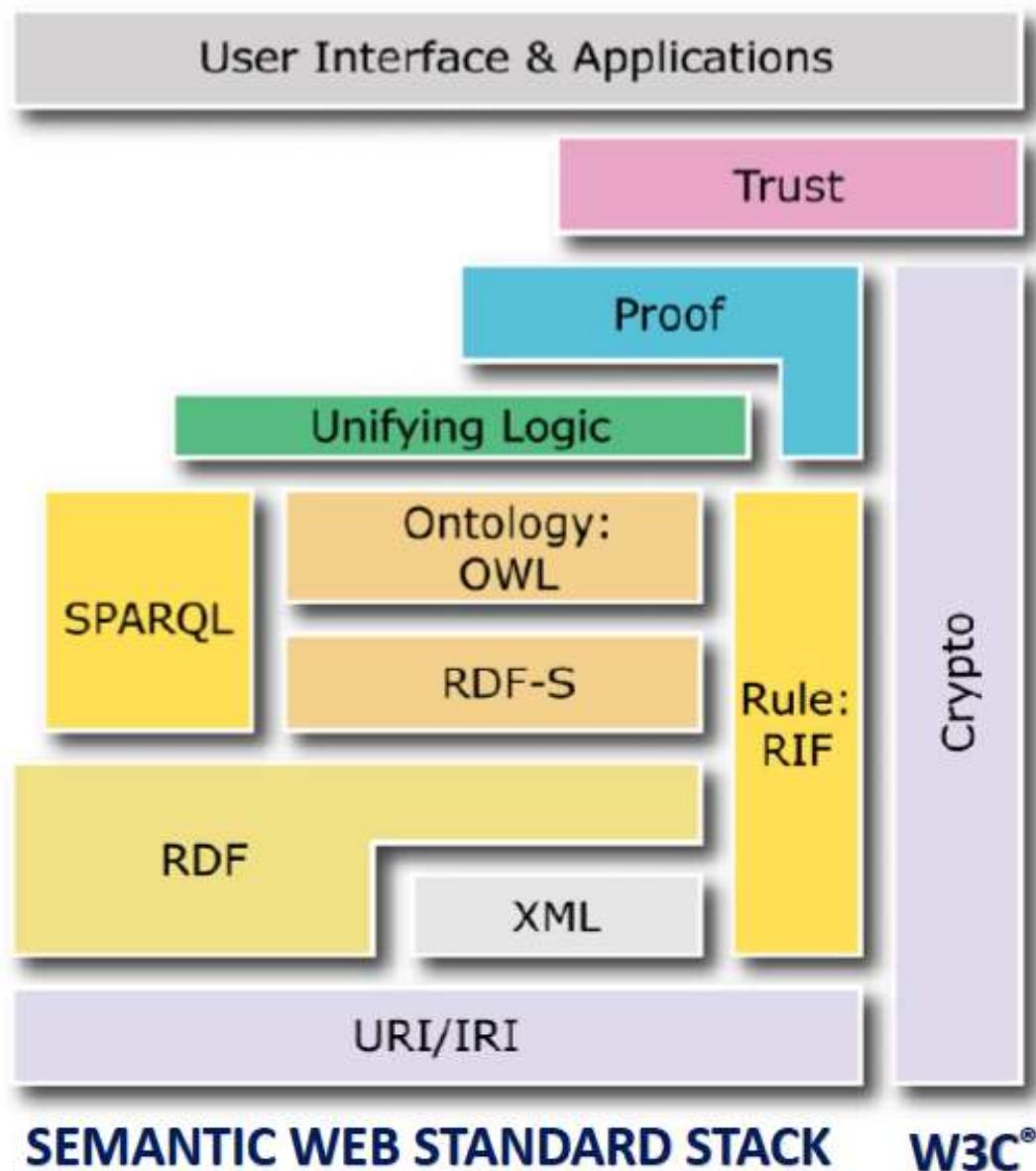
The Semantic Web

- The Semantic Web is a Web in which the **resources** (things) are semantically described, through the usage of an **ontology**
- A resource is anything that can be referred to by a **URI** (Uniform Resource Identifiers)
 - a web page, identified by an URL
 - a fragment of an XML document, identified by an element node of the document
 - a web service
 - a thing, an object, a property, etc.
- Examples
 - <http://www.example.org/file.html#home>
 - [http://www.example.org/file2.xml#xpath>//q\[@a=b\]](http://www.example.org/file2.xml#xpath>//q[@a=b])
 - <http://www.example.org/form?a=b&c=d>

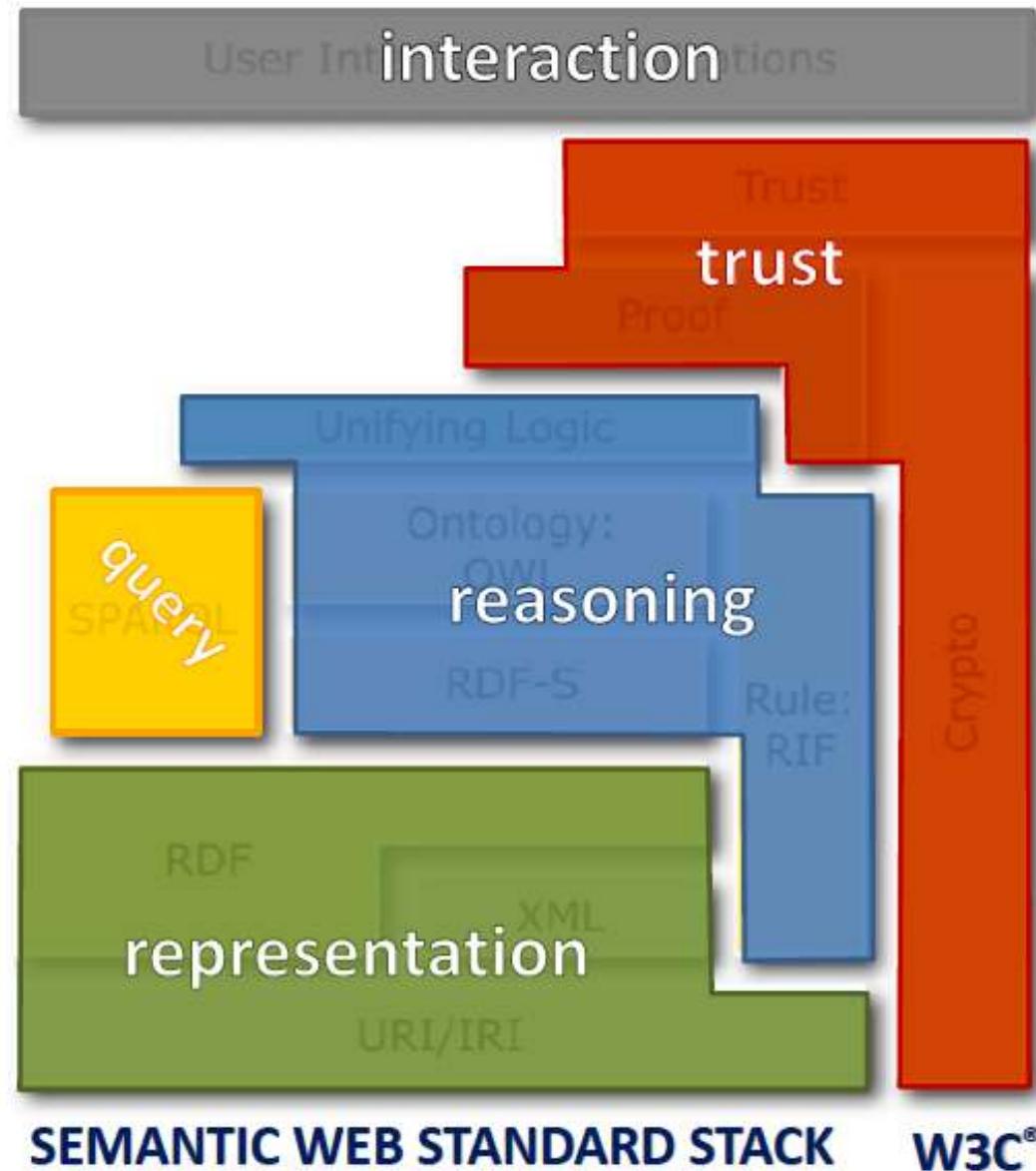


*"Now! That should clear up
a few things around here!"*

The Semantic Web Standard Stack



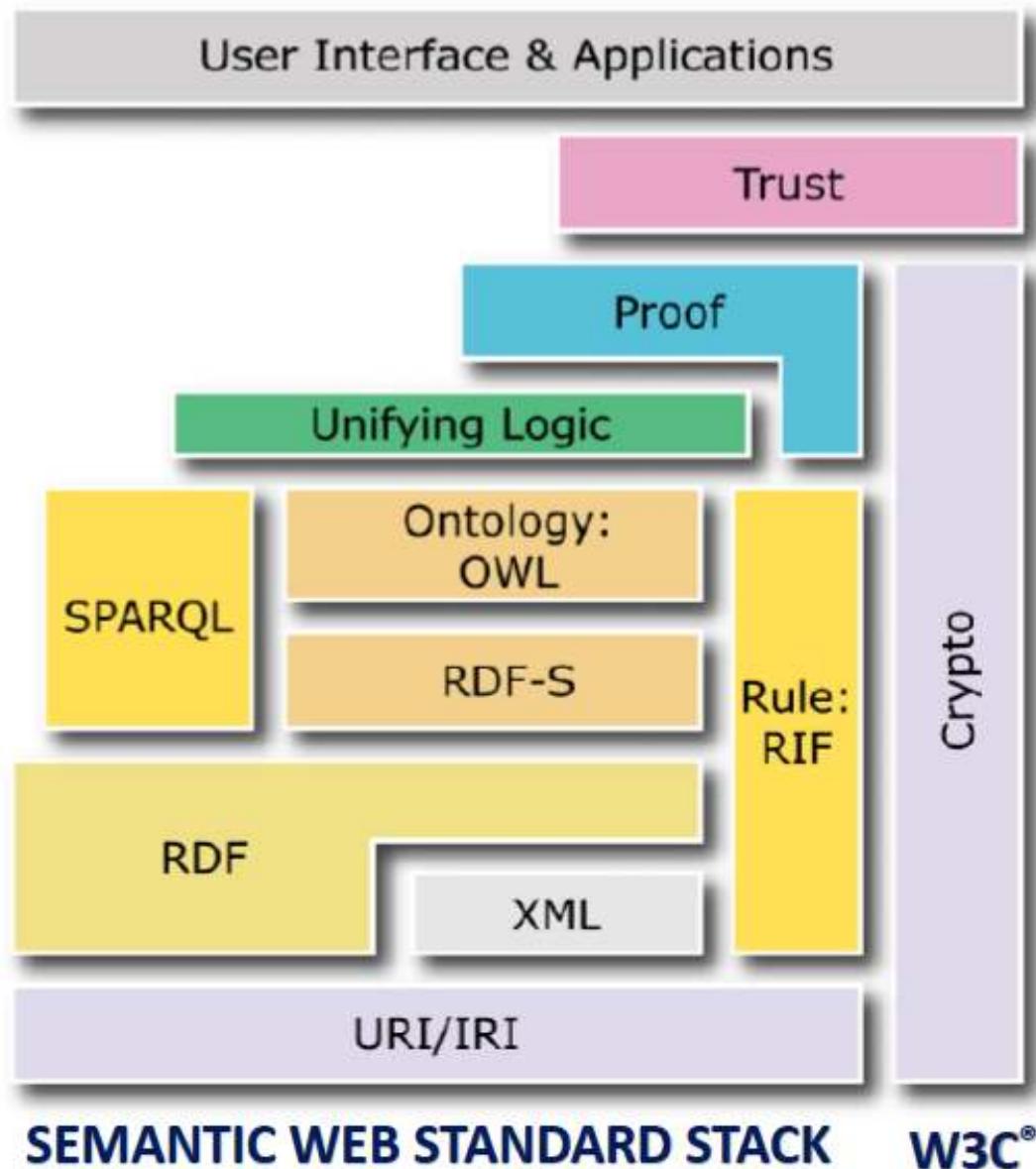
The Semantic Web Standard Stack



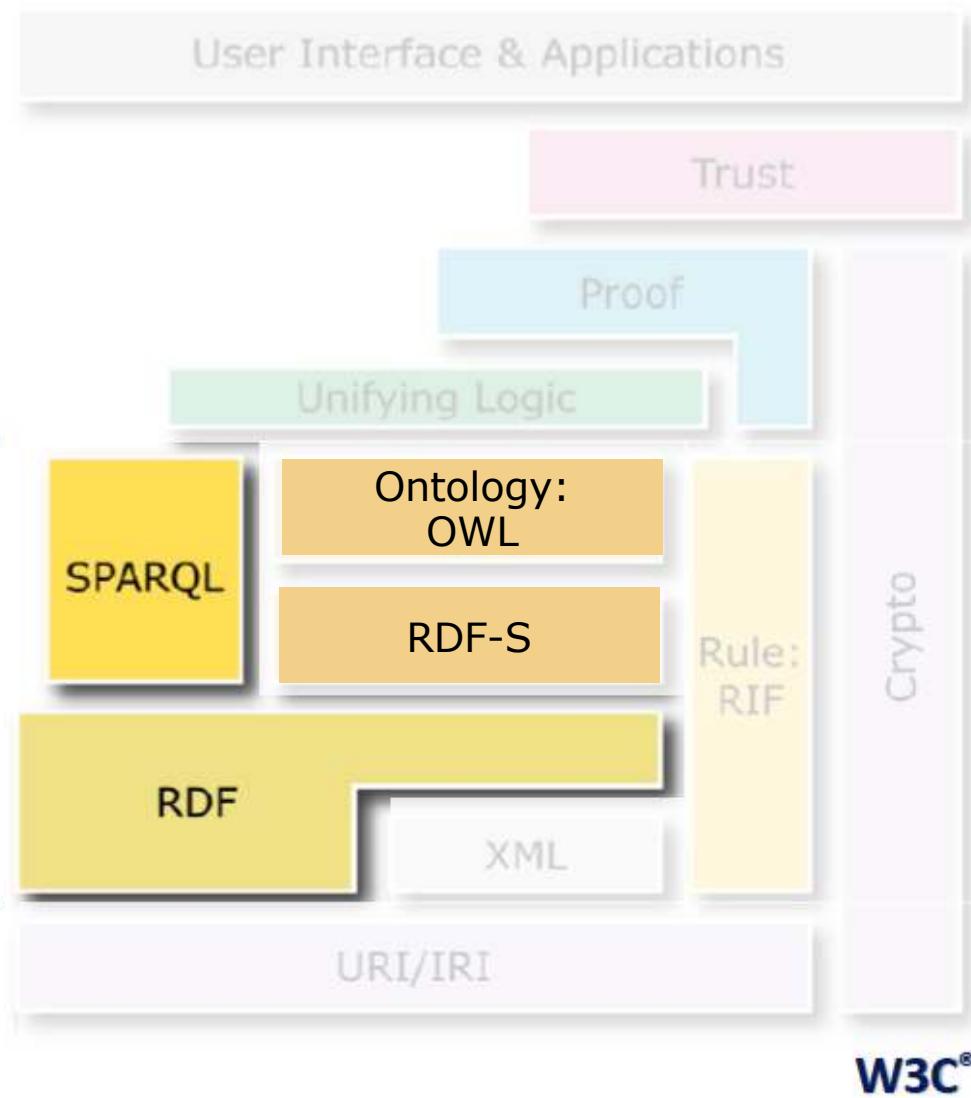
Ontology Languages for the Semantic Web

- RDF: a very simple ontology language
- RDFS: Schema for RDF
 - ▶ Can be used to define richer ontologies
- OWL: a much richer ontology language

The Semantic Web Standard Stack



The Semantic Web Standard Stack



W3C Linked Open Data Project

- Goal: “expose” open datasets in RDF on the Web
- Set RDF links among the data items from different datasets
- Set up, if possible, query endpoints

Linked Open Data

- **Linked data** is just RDF (RDFS, OWL) data, linking together resources potentially described in different datasets
- **Linked open data** is just linked data freely accessible on the Web along with any required ontologies

Properties of Linked (Open) Data

- Anyone can publish data on the Web of Linked Data
- Entities are connected by links
 - ▣ Creating a global data graph that spans data sources and enables the discovery of new data sources.
- Data is self-describing
 - ▣ If an application encounters data represented using an unfamiliar vocabulary, the application can resolve the URLs that identify vocabulary terms in order to find their RDFS or OWL definition.
- The Web of Data is open
 - ▣ Meaning that applications can discover new data sources at run-time by following links.

Example: DBpedia

- DBpedia is a community effort to
- extract structured (“infobox”) information from Wikipedia
- provide a query endpoint to the dataset
- interlink the DBpedia dataset with other datasets on the Web

Example: DBpedia

```
@prefix dbpedia <http://dbpedia.org/resource/>.  
@prefix dbterm <http://dbpedia.org/property/>.
```

```
dbpedia:Amsterdam
```

```
    dbterm:officialName "Amsterdam" ;  
    dbterm:longd "4" ;  
    dbterm:longm "53" ;  
    dbterm:longs "32" ;  
    dbterm:leaderName dbpedia:Lodewijk_Asscher ;  
    ...  
    dbterm:areaTotalKm "219" ;  
    ...
```

```
dbpedia:ABN_AMRO
```

```
    dbterm:location dbpedia:Amsterdam ;  
    ...
```

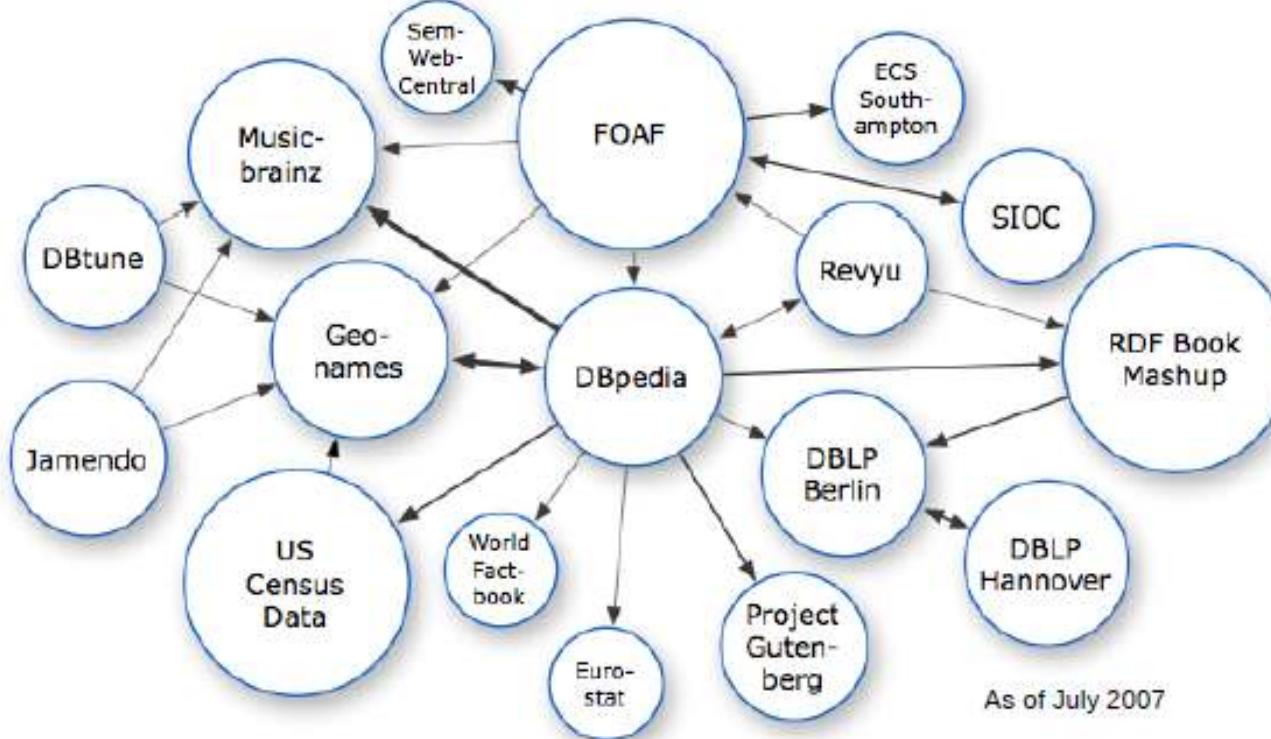


The Kaisersgracht at dusk

Location of Amsterdam
Coordinates: 52°32'31.1"N 4°19'30.2"E

| | |
|----------------|--|
| Country | Netherlands |
| Province | North Holland |
| Government | |
| - Type | Municipality |
| - Mayor | Job Cohen ^[1] (Pvda) |
| - Aldermen | Lodewijk Asscher Carolinus Gheerts Tjeerd Herrema Maarten van Poelgeest Manjik Vos |
| - Secretary | Erik Gerritsen |
| Area | |
| - City | 219 km ² (84.6 sq mi) |
| - Land | 160 km ² (64.3 sq mi) |
| - Water | 59 km ² (23.0 sq mi) |
| - Urban | 1,003 km ² (387.3 sq mi) |
| - Metro | 1,815 km ² (700.8 sq mi) |
| Elevation | 2 m (7 ft) |
| Population | (1 October 2008) ^{[2][3]} |
| - City | 760,269 |
| - Density | 4,459/km ² (11,548.8/sq mi) |
| - Urban | 1,384,422 |
| - Metro | 2,168,372 |
| - Demonym | Amsterdamer |
| Time zone | CET (UTC+1) |
| - Summer (DST) | CEST (UTC+2) |
| Postcodes | 1011 – 1109 |
| Area code(s) | 020 |
| Website | www.amsterdam.nl |

Linked Open Data Project

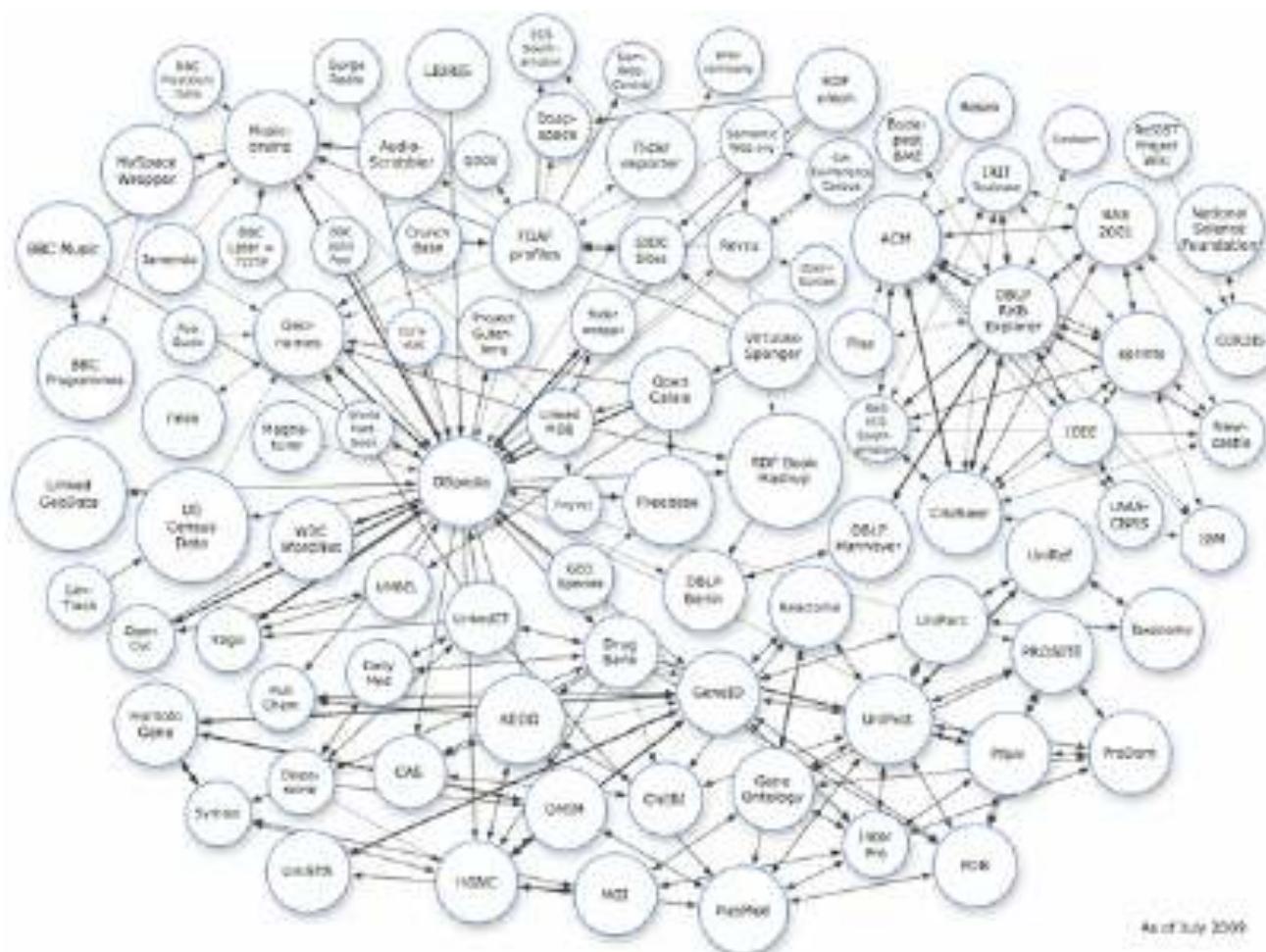


Linked Open Data Project



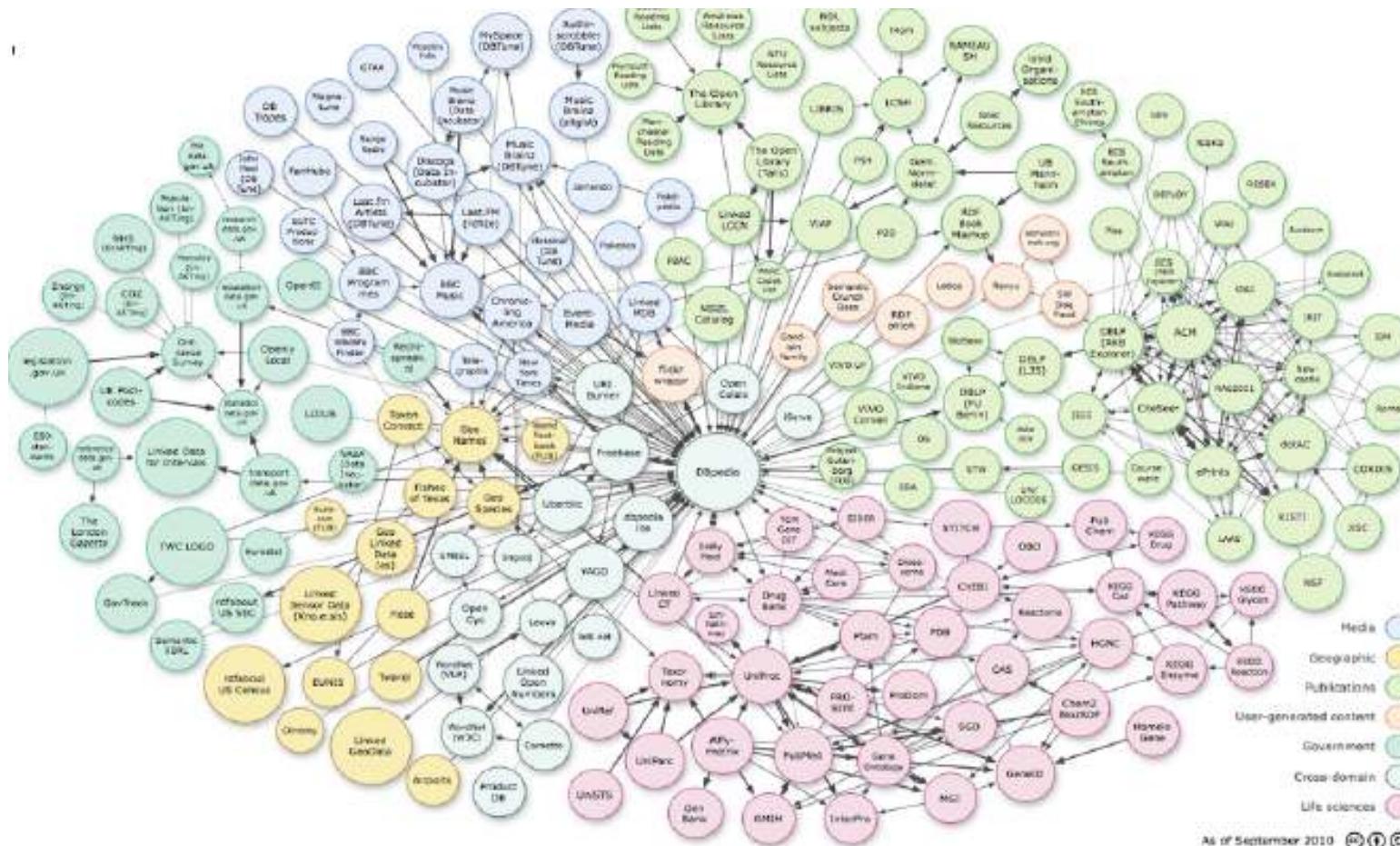
2008

Linked Open Data Project



2009

Linked Open Data Project

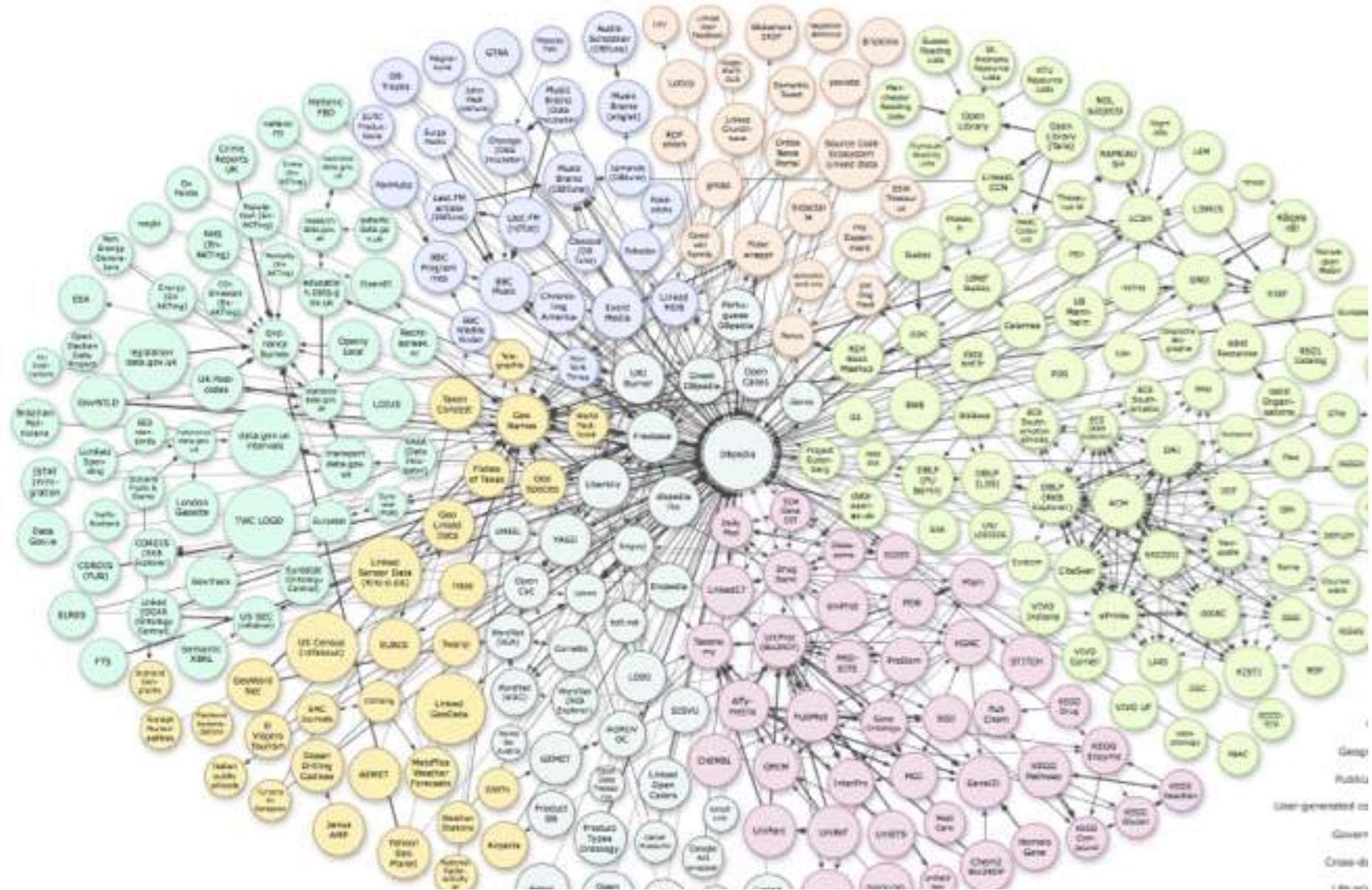


2010

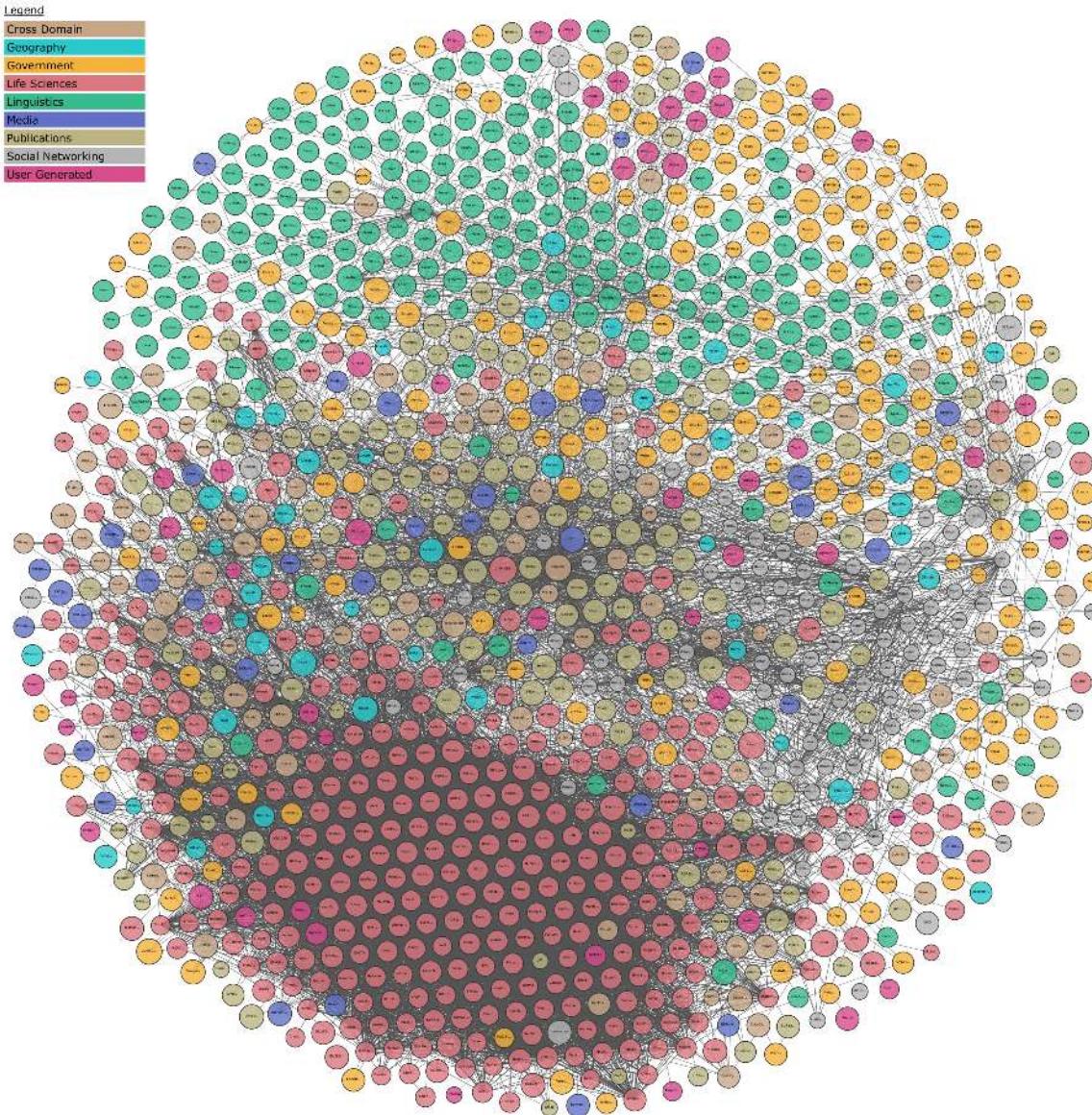
As of September 2010



Linked Open Data Project



Linked Open Data Project



The dataset contains
1269 datasets with
16201 links
(as of May 2020)

<https://lod-cloud.net>

neo4j

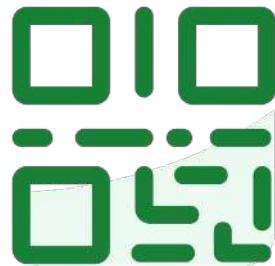
Modeling and querying for graph databases

Stefano Ottolenghi

! a cemetary



slido



Join at [slido.com](https://www.slido.com)
#3244815

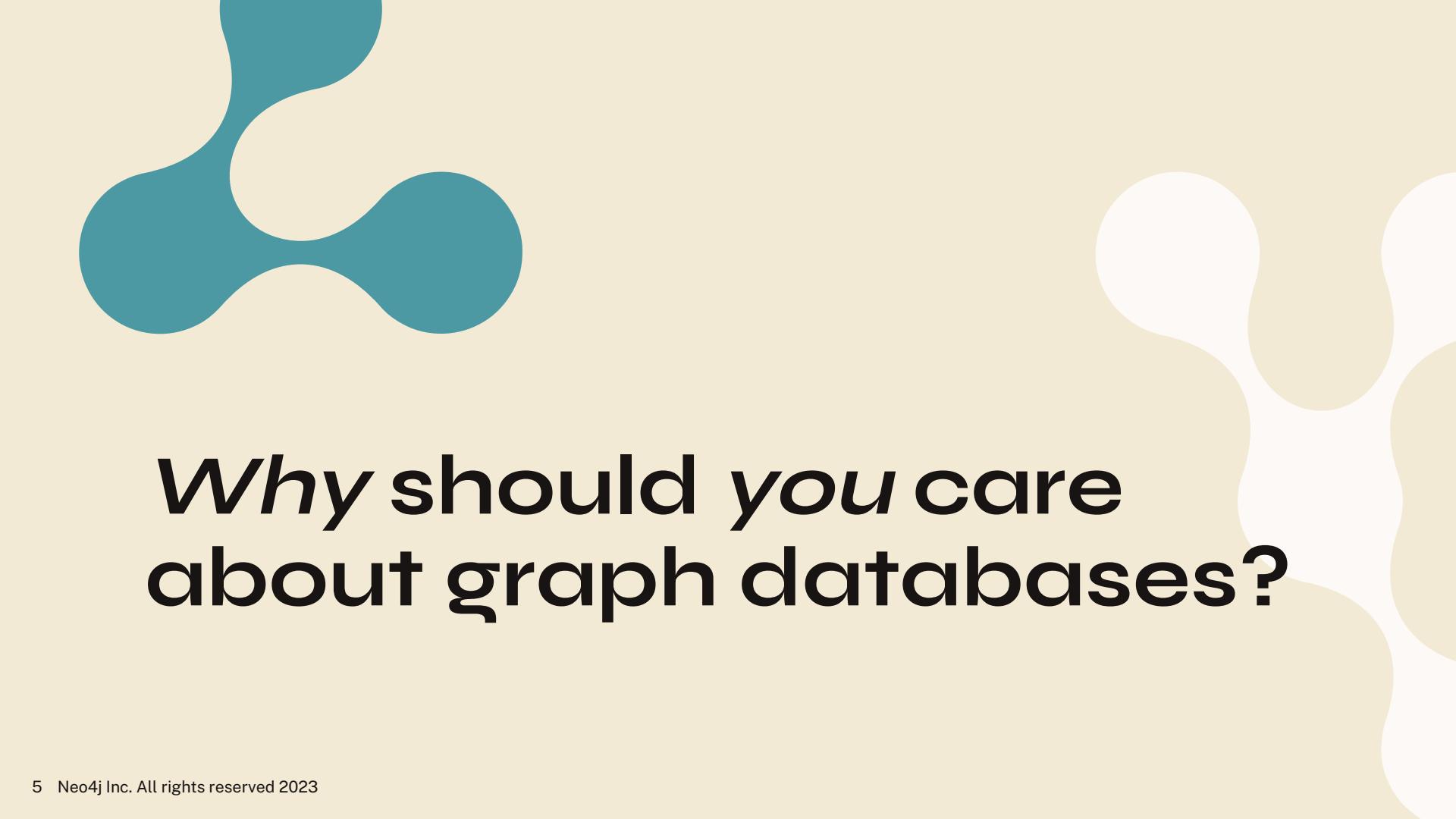
- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to display joining instructions for participants while presenting.

slido

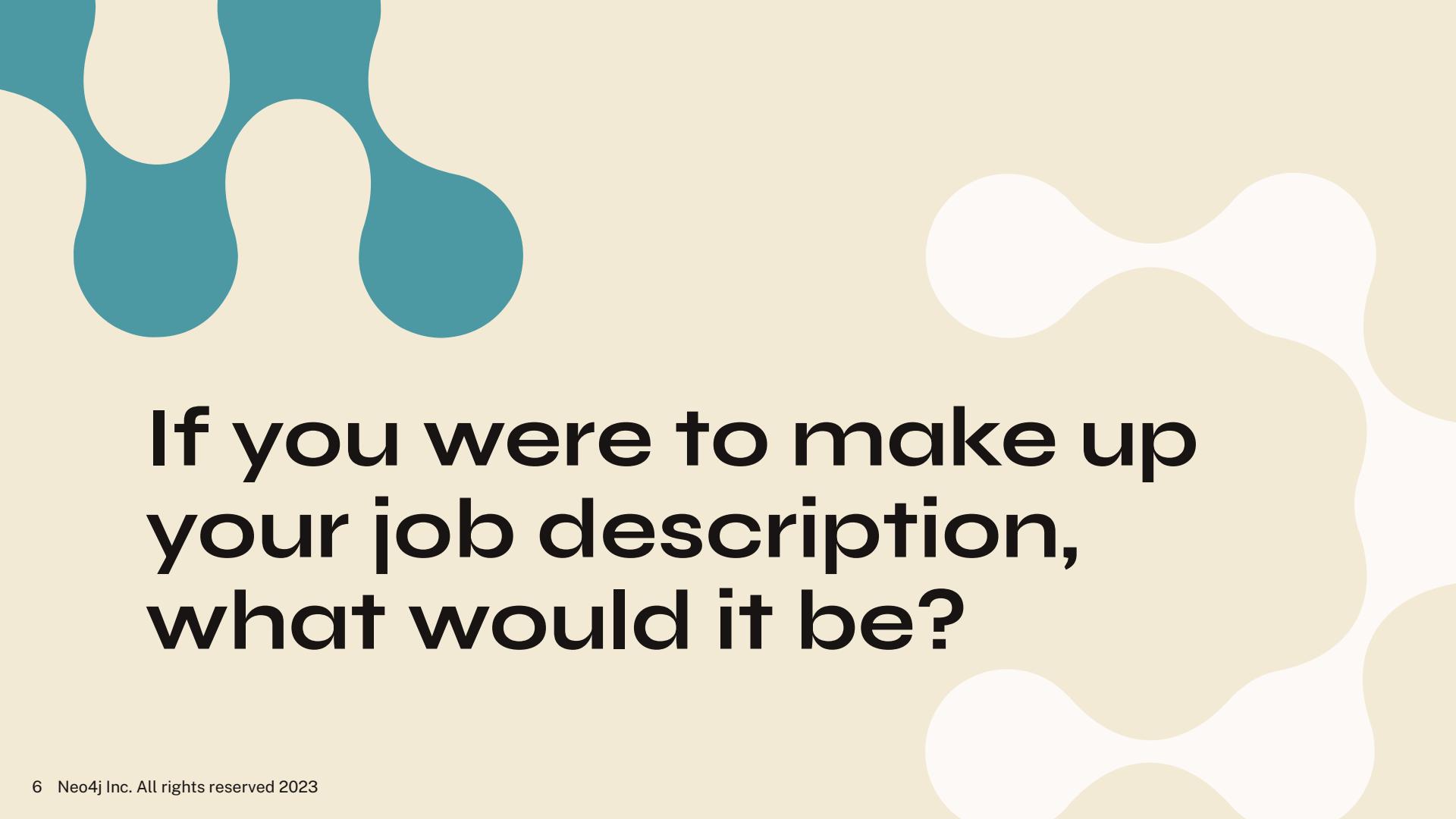


How is your Neo4j instance deployed?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

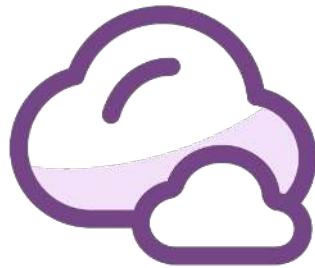


Why should you care about graph databases?



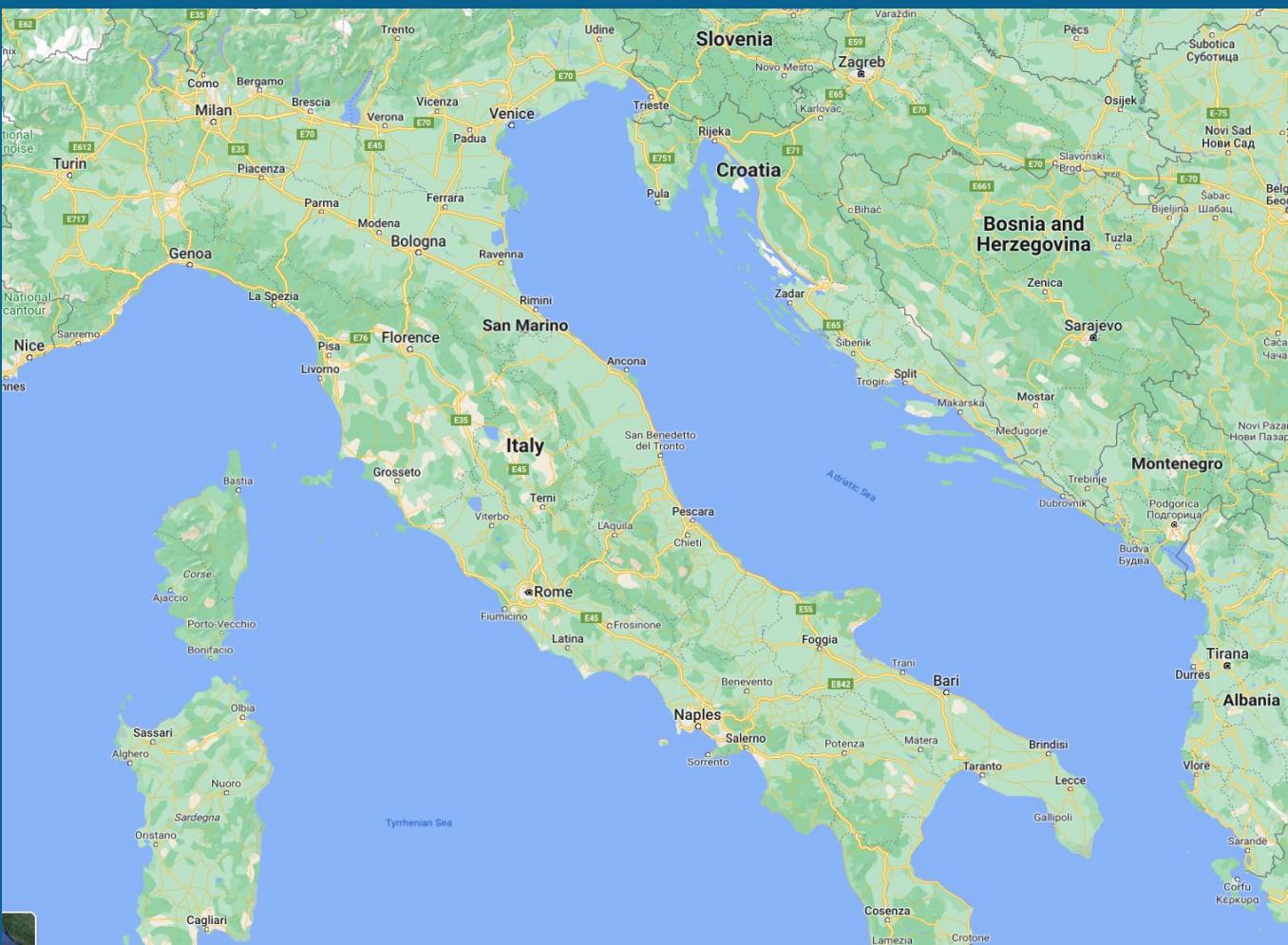
**If you were to make up
your job description,
what would it be?**

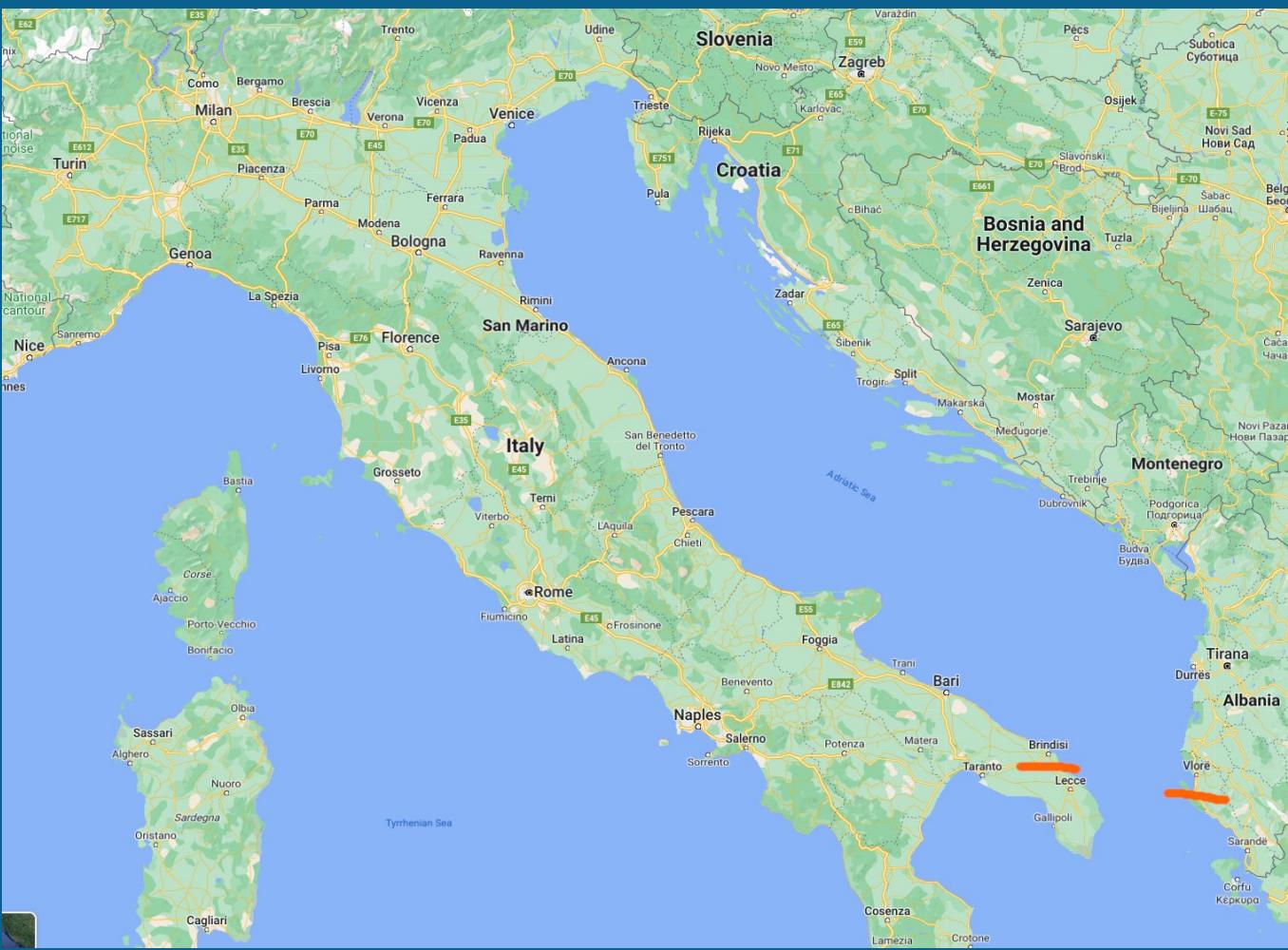
slido

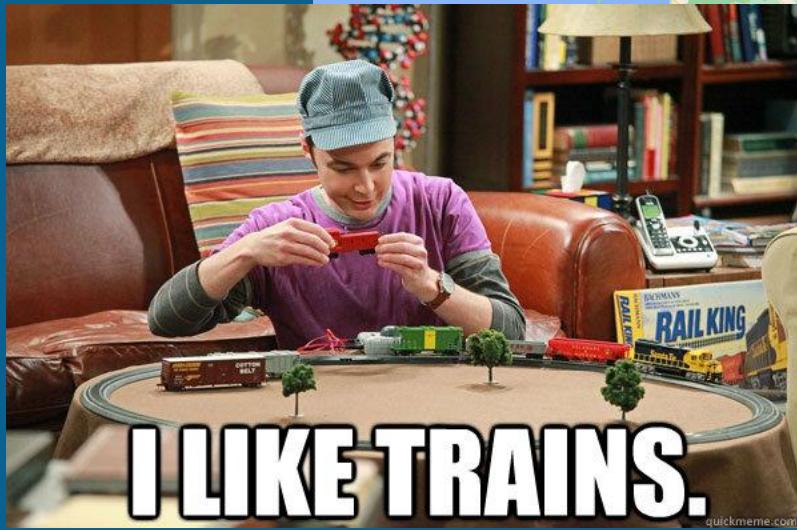


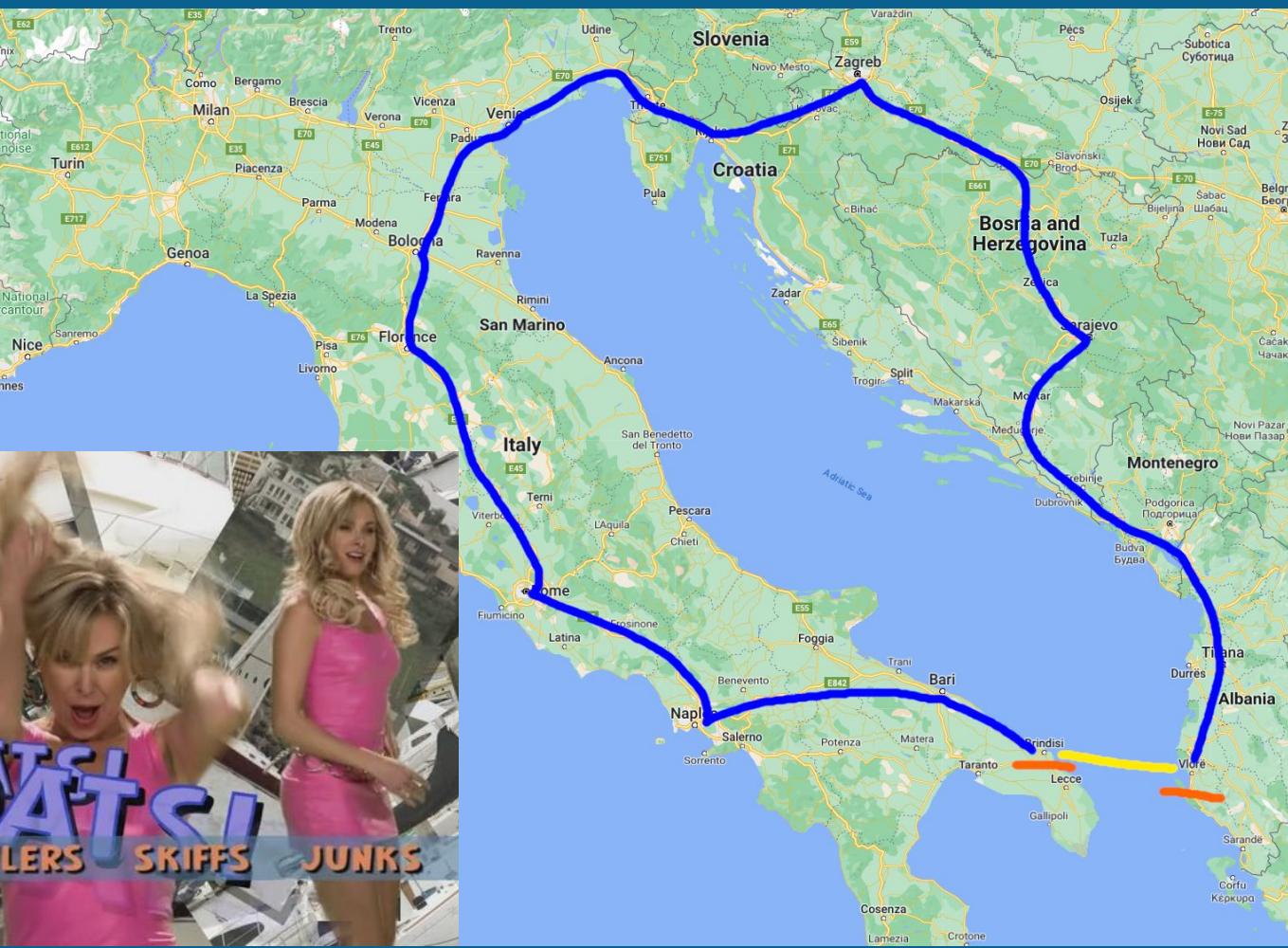
What's your dream job description?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.









Think of yourself as a ...



The more tools you know, the more problems you can solve (*efficiently*).

Questions?

I may have answers.

Twenty minute introduction to graph databases

Find the fraudster

Ash gonna catch them 'll



Not Sinatra



Blue Eyes



Terminology

Node - vertex, entity, object, thing



Terminology

Node - vertex, entity, object, thing

Label - concept, class(ification) of a node

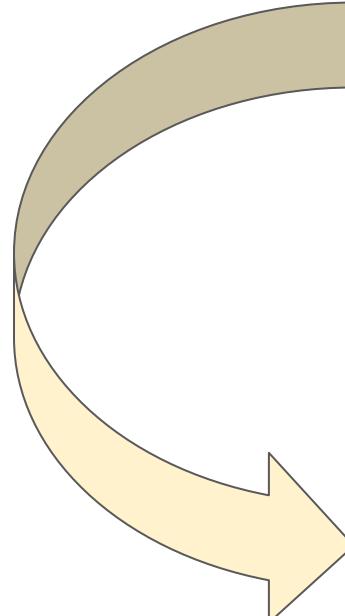


Terminology

Node - vertex, entity, object, thing

Label - concept, class(ification) of a node

Relationship - edge, a physical link between two nodes



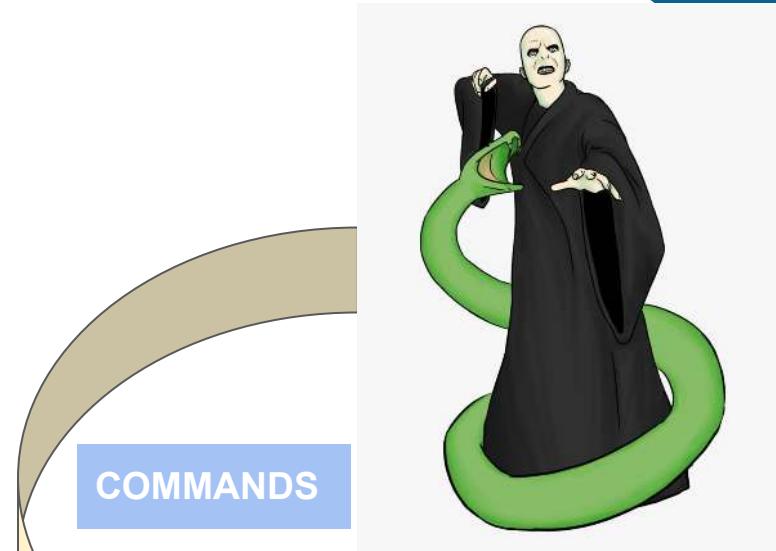
Terminology

Node - vertex, entity, object, thing

Label - concept, class(ification) of a node

Relationship - edge, a physical link between two nodes

Type - class(ification) of a relationship



Terminology

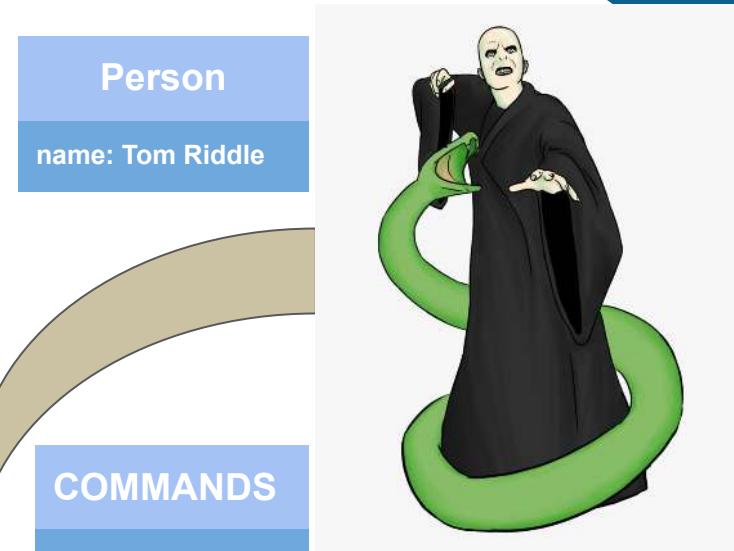
Node - vertex, entity, object, thing

Label - concept, class(ification) of a node

Relationship - edge, a physical link between two nodes

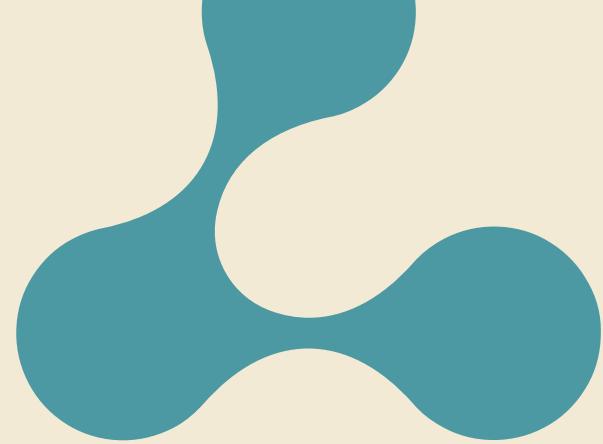
Type - class(ification) of a relationship

Property - key-value pair



Demo(n)





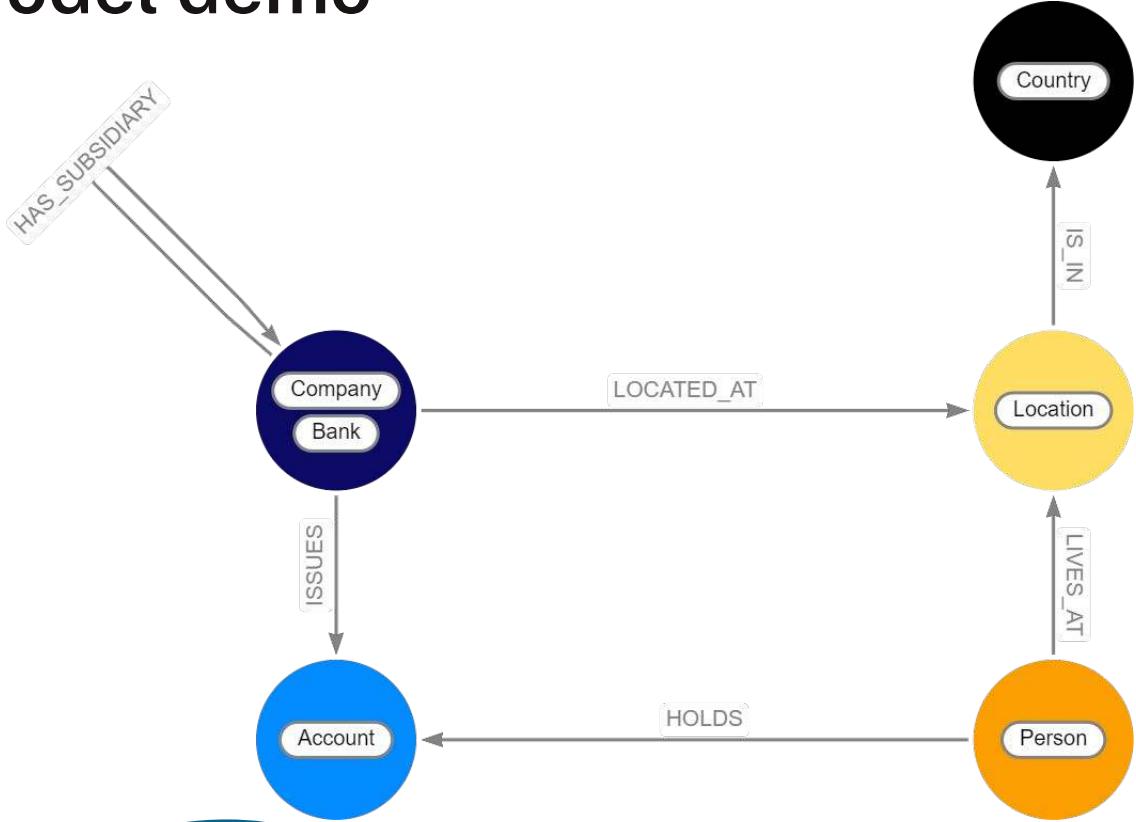
Play along!

bit.ly/neo4junige

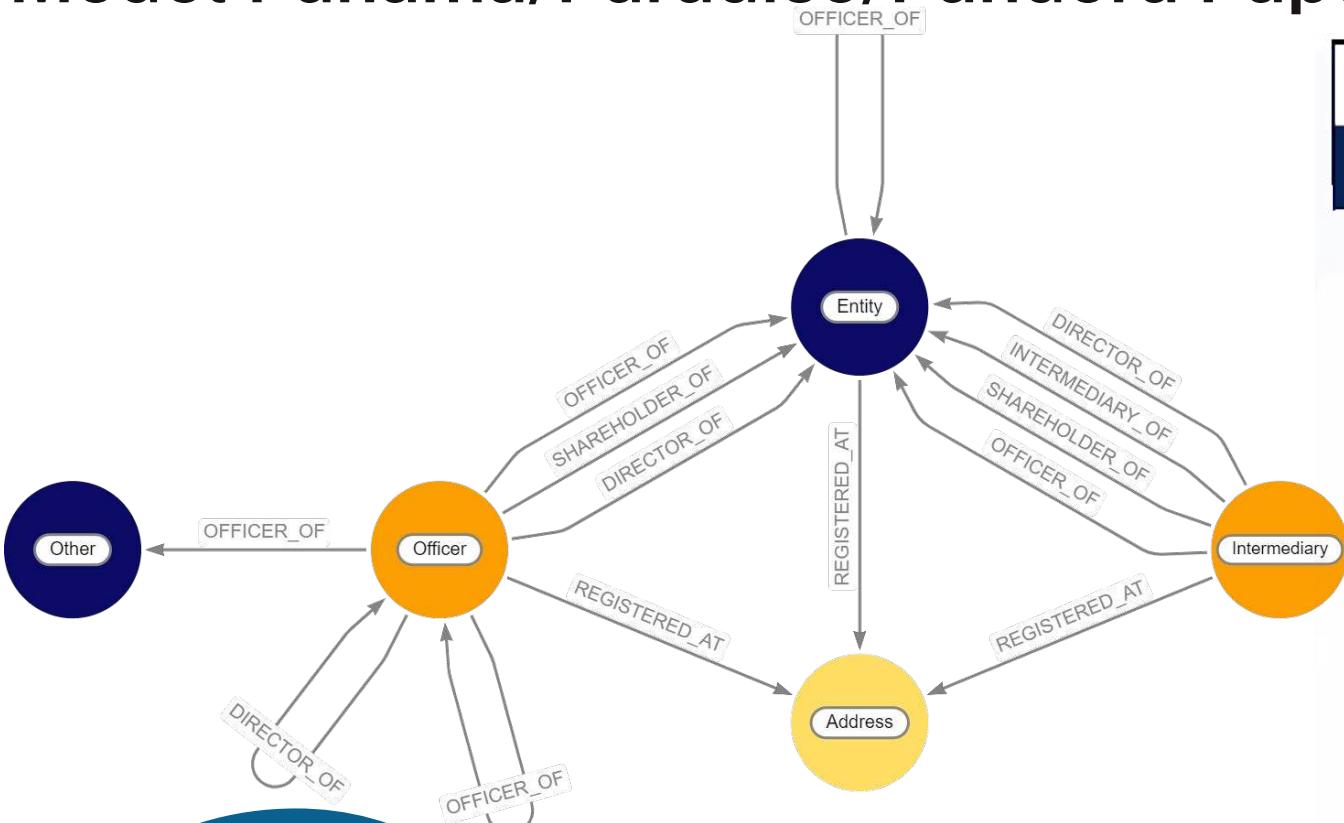
Drag unige-querying.csv
into “Saved Cypher”.



Model demo



Model Panama/Paradise/Pandora Papers



Takeaways

- **Connected data = differentiator & disruptor**

Google, Facebook, LinkedIn, ...



- **A graph database helps make sense of connected data**

The connections are *real*



- **It's not about the complexity of the *model*, it's about the complexity of the *query***

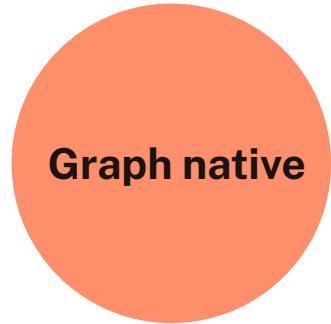
Questions?

I may have answers.

Neo4j in a



Secret Ingredients



Mister Ping
Noodle Expert

Native versus non-native

| | neo4j Native Graph DB | Non-Native Graph DB | RDBMS |
|-------------------------------|--|------------------------------|-------|
| Visualization | | | |
| Queries | Cypher <code>(graphs)-[are]->(everywhere)</code> | Cypher/Gremlin /Proprietary | SQL |
| Processing | | | |
| Storage | | Table Key-Value Column | |
| Optimized for graph workloads | | | |

Secret Ingredients

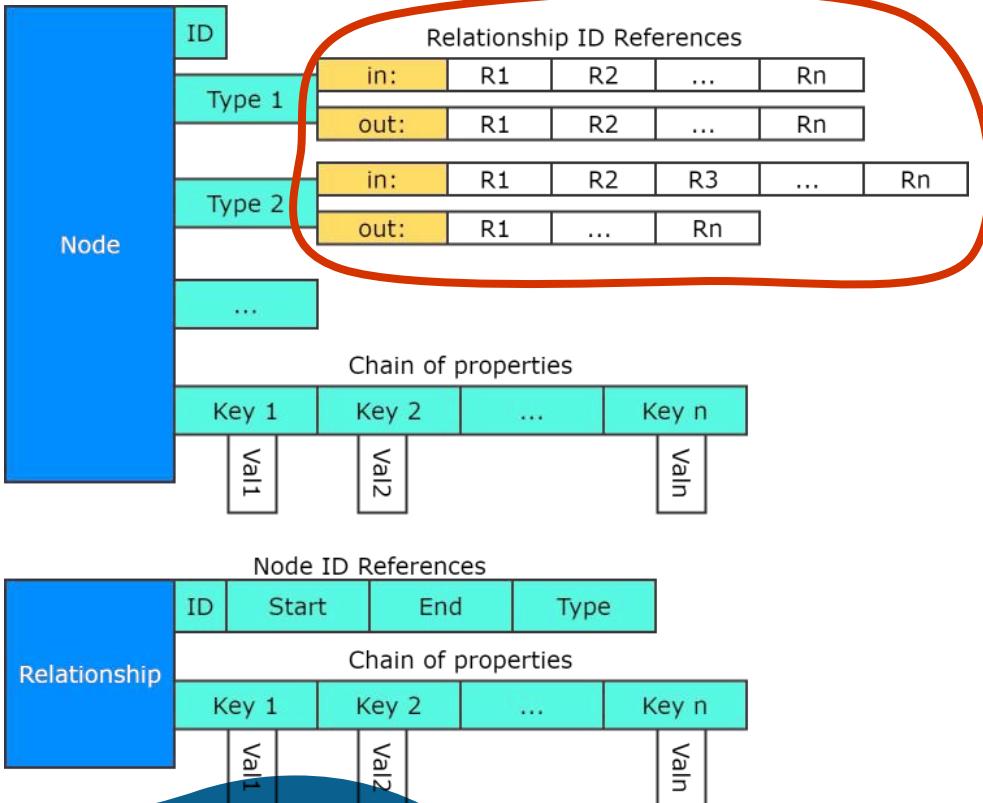
Graph native

Index-free
adjacency



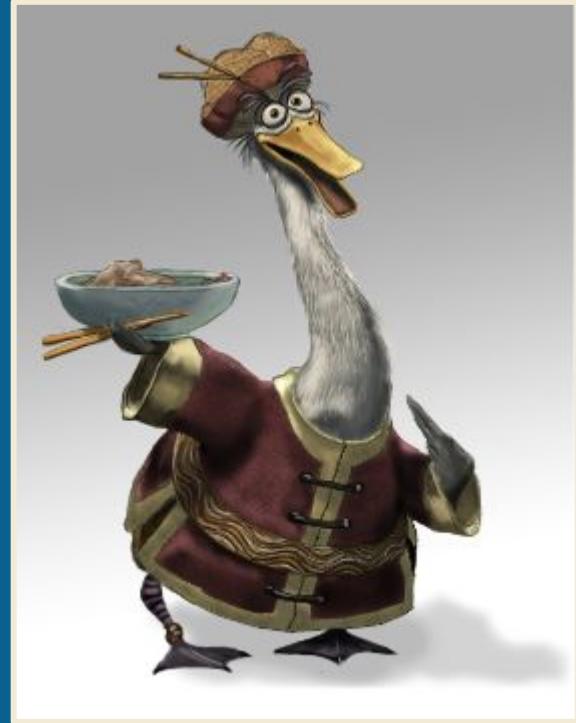
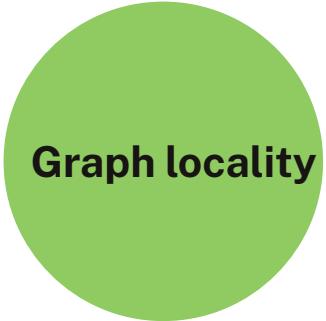
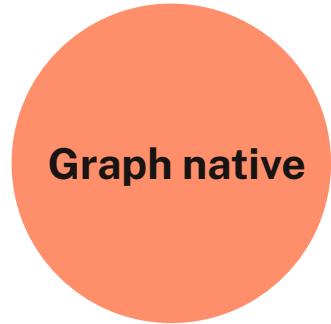
Mister Ping
Noodle Expert

Index-free adjacency



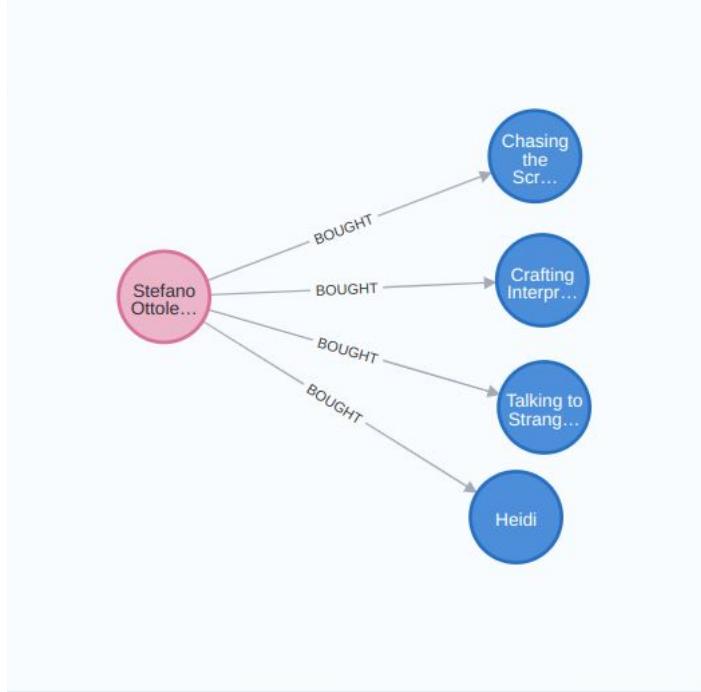
- Pointer hopping instead of index lookups
- Fixed size objects
- Joins are done on creation

Secret Ingredients



Mister Ping
Noodle Expert

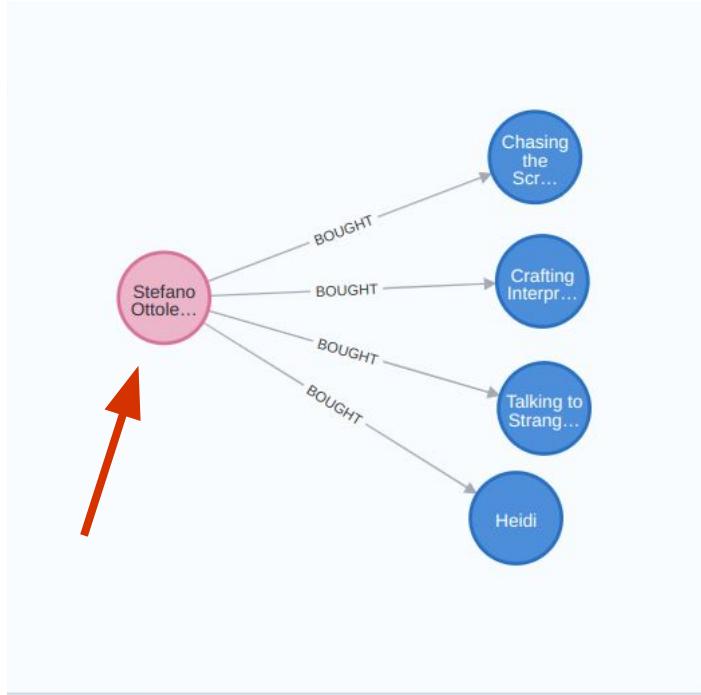
Graph locality



It doesn't matter if the data sits in a **huge** graph or a **small** graph.

A query performs in the same way (no big table problem) in all cases.

Gimme gimme gimme ... an entrypoint



Relational

1 + 2x index lookups
 $O(n \log n)$

VS

Graph

1 index lookup
x pointer hops
 $O(1)$

Downsides





Name two possible downsides of the Neo4j Secret Ingredients

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

Use cases



Real-Time
Recommendations



Fraud
Detection



Tracking mail
delivery



Graphs enabling
glycoscience



Knowledge
Graph Based
Chatbot



Identity & Access
Management

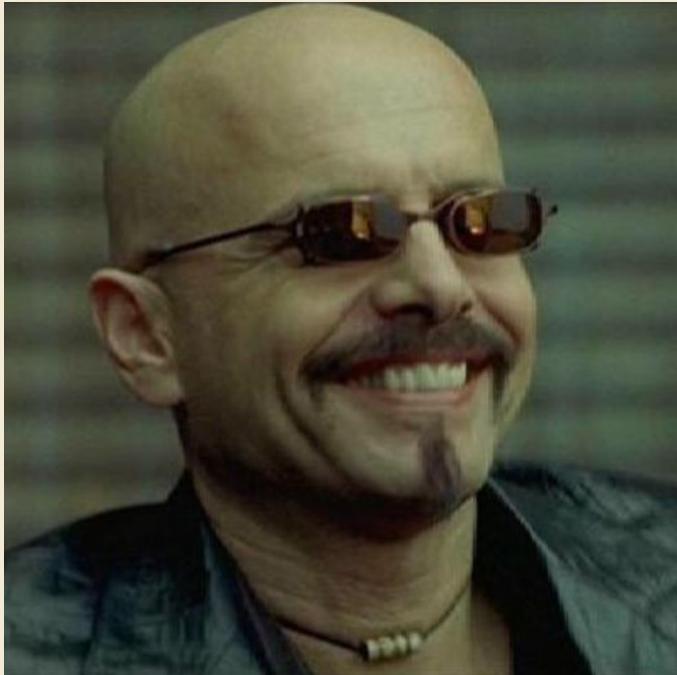
Questions?

I may have answers.

Cypher

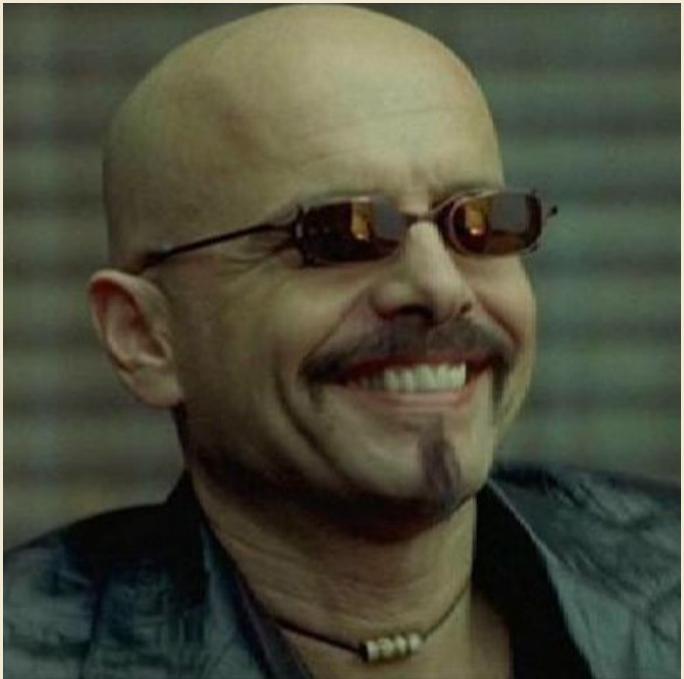
A new query language

Who is Cypher?



This is Cypher

Cypher ASCII art



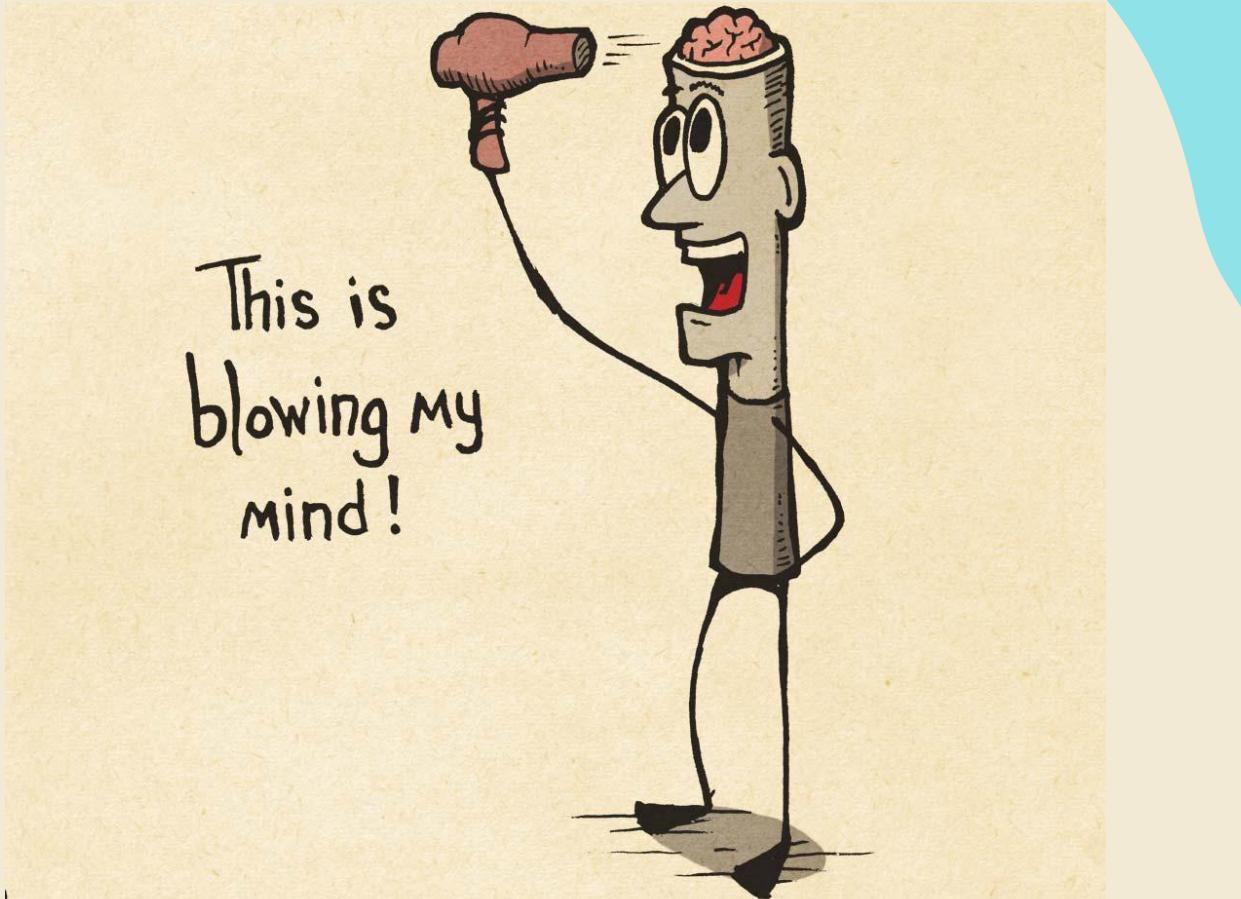
This is Cypher



```
( " ` - ' ' - / " ) . _ _ _ . . - - ' ' " ^ - . _  
` 6 _ 6 _ ) _ ` - . ( _ ) . ` - . _ . ` )  
( _ Y _ . ) ' . _ ) ` . _ ` . ` - . - - ' _  
_ . . ` - - ' _ . . - _ / / - - ' _ . ' _  
(( (( . - ' ' (( (( . ' (( (( . - '
```

This is ASCII Art

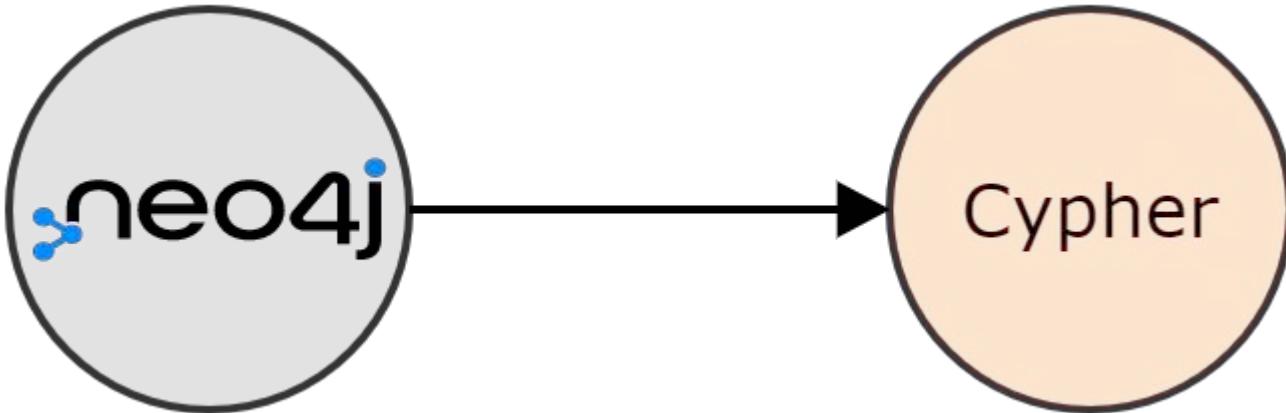
Cypher ASCII art



What is Cypher?

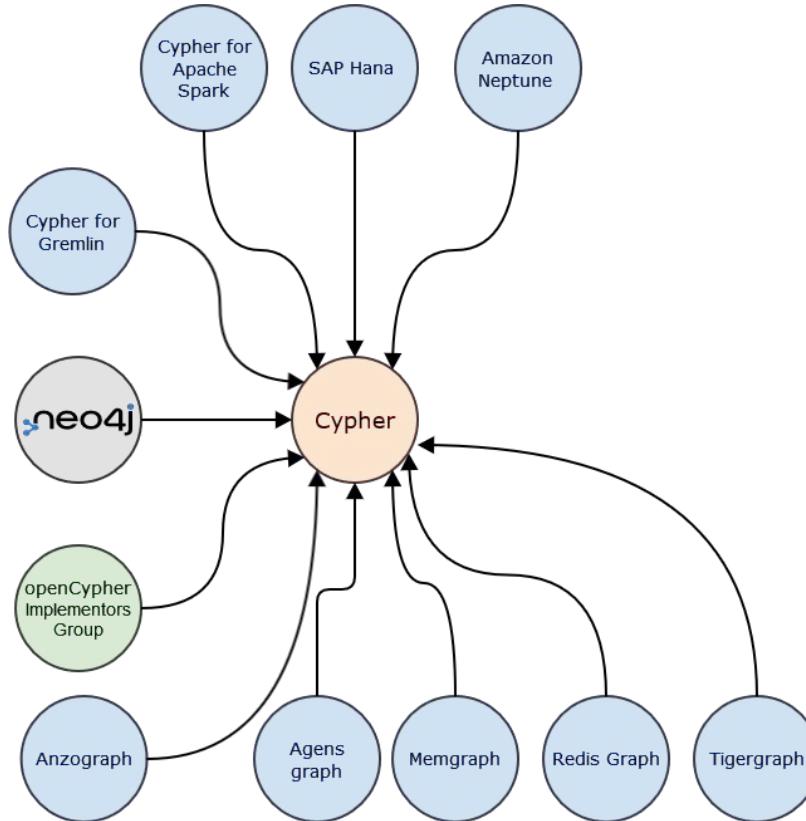
- Declarative (focus on *what*, not *how*) query language for property graphs
- Uses ASCII Art to visually describe patterns in a graph

Where is Cypher going?



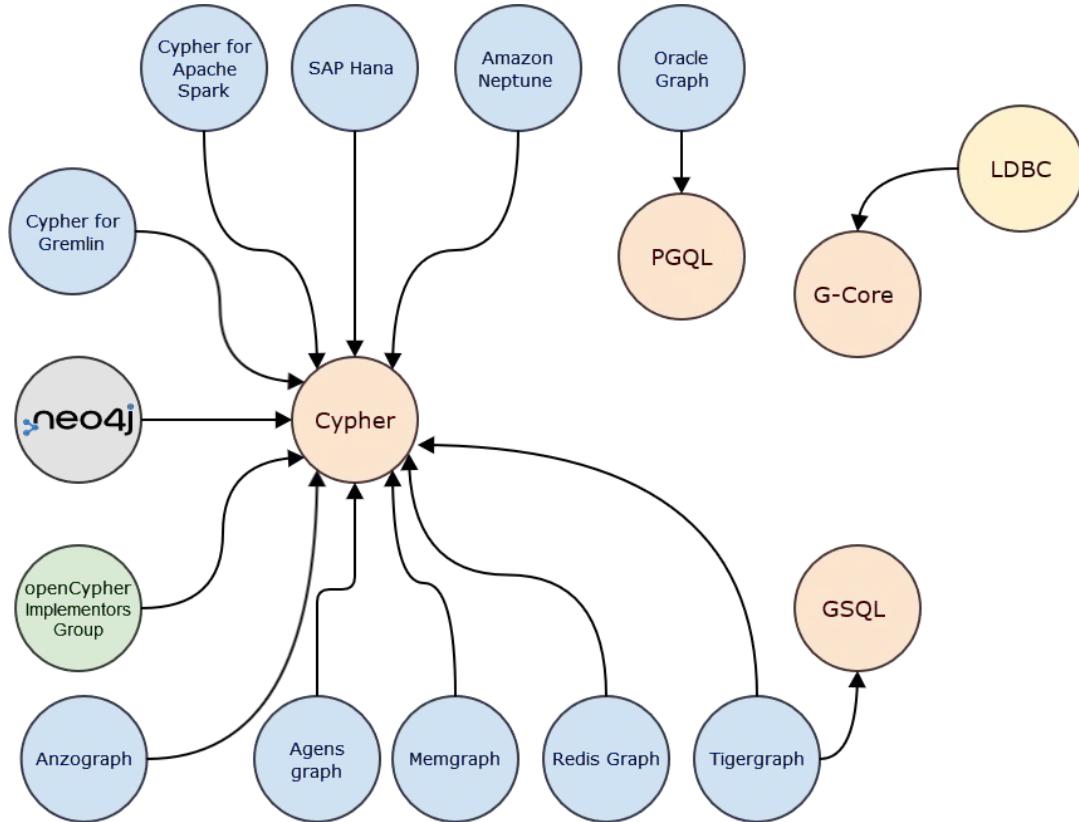
Where is Cypher going?

Cypher has been open from the start and has quite a few implementations!



Where is Cypher going?

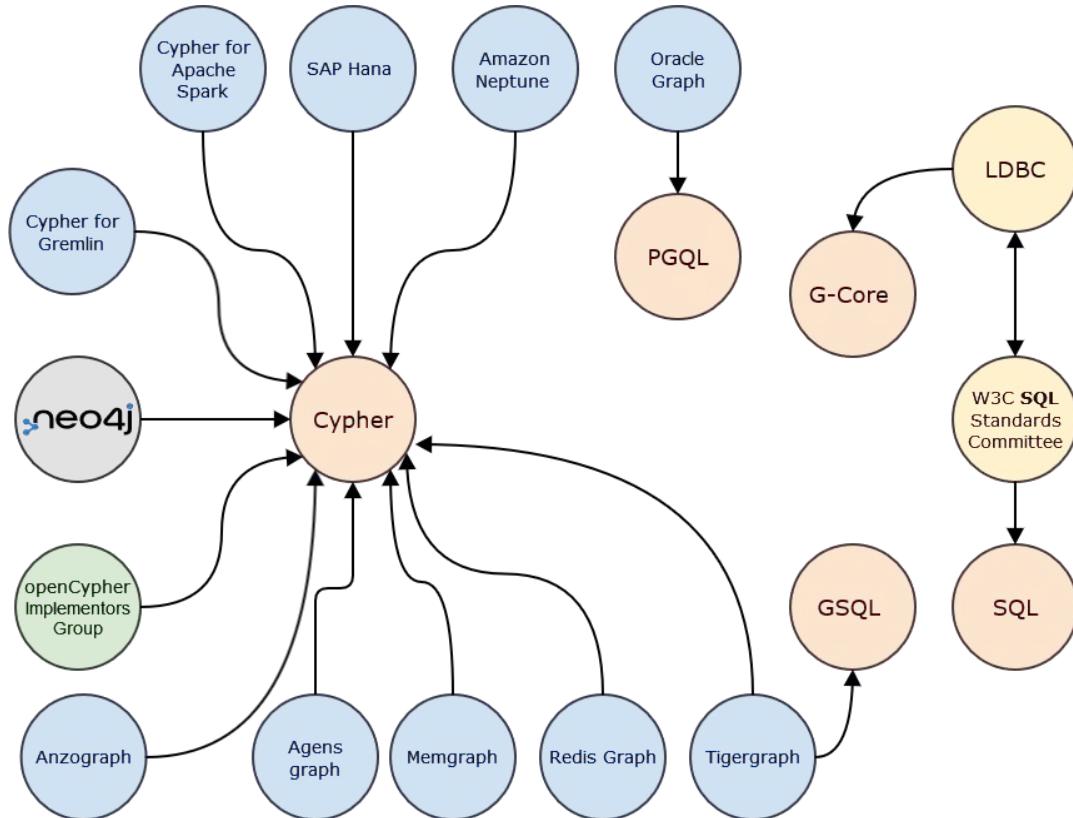
Realisation that a property graph requires a specific query language is much wider!



Where is Cypher going?

What does the Alpha & Omega
of standards think?

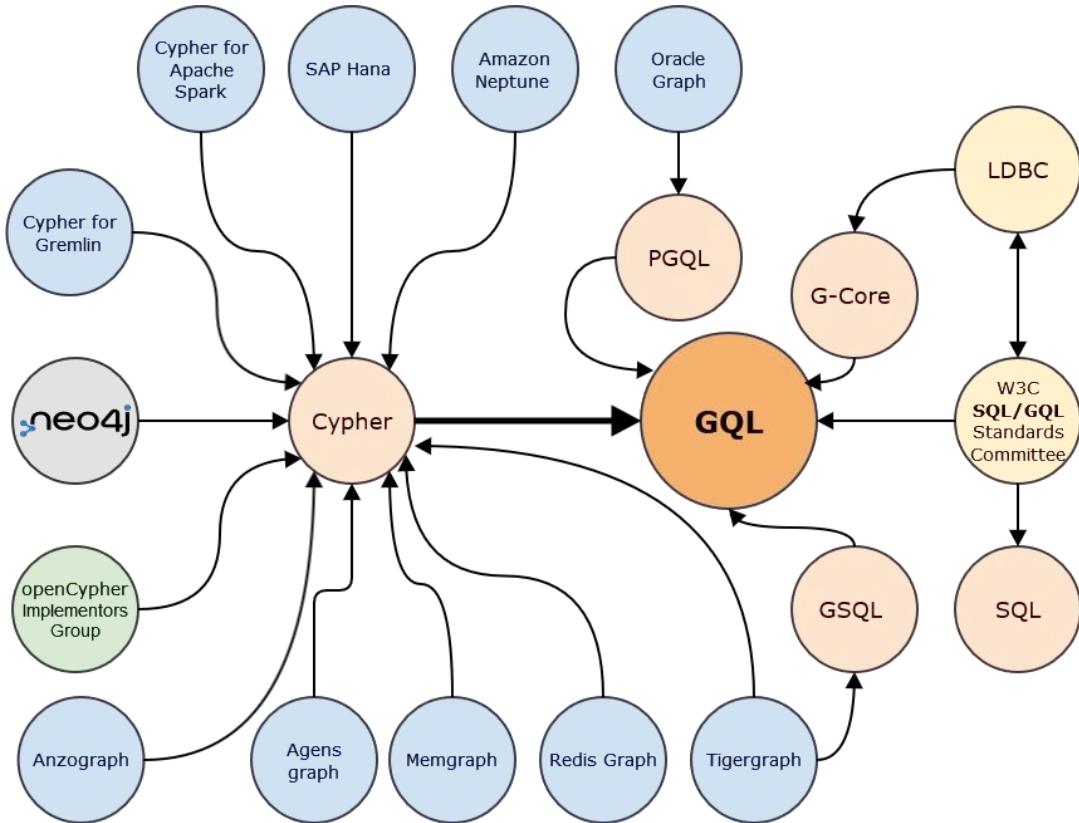
That committee hasn't budged in
over three decades ...



Where is Cypher going?

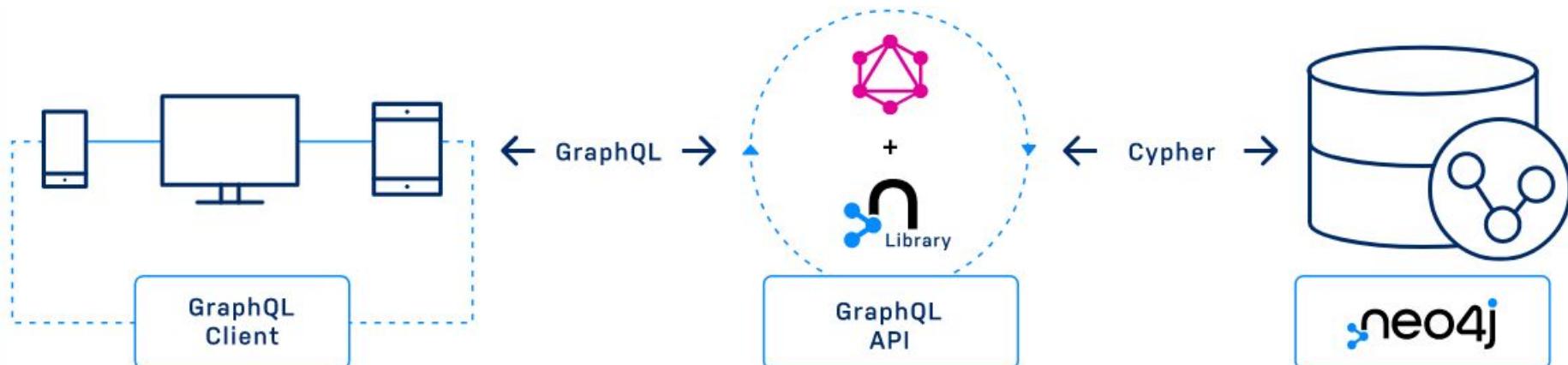
But it started moving in 2019,
including an official renaming!

GQL is now an official ISO
standard (in progress).



GQL ≠ GraphQL

Say the words and you'll hear the same thing twice ...
but they couldn't be more different.



Play along!



Practice time!

- Clean your database
 $\text{MATCH } (p) \text{ DETACH DELETE } p$
- Create a node representing yourself, with interests in at least 3 topics.
No duplicate nodes allowed.
...
- After doing it, you may want to check out the clause UNWIND, and adapt your query.



slido



Now that you are old and wise, if you could say one single thing to your children, what would it be? What is your life lesson?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

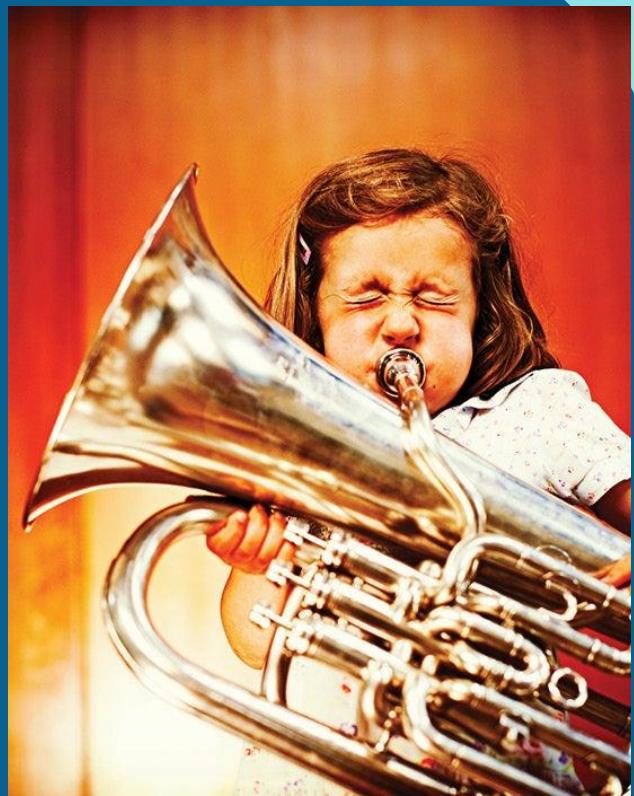
Cheaty solutions

```
MERGE (me:Person {name:'Stefano'})  
MERGE (t1:Topic {name:'Roller skating'})  
MERGE (t2:Topic {name:'Acroyoga'})  
MERGE (t3:Topic {name:'Rants'})  
MERGE (me)-[:LIKES]->(t1)  
MERGE (me)-[:LIKES]->(t2)  
MERGE (me)-[:LIKES]->(t3)  
RETURN me
```

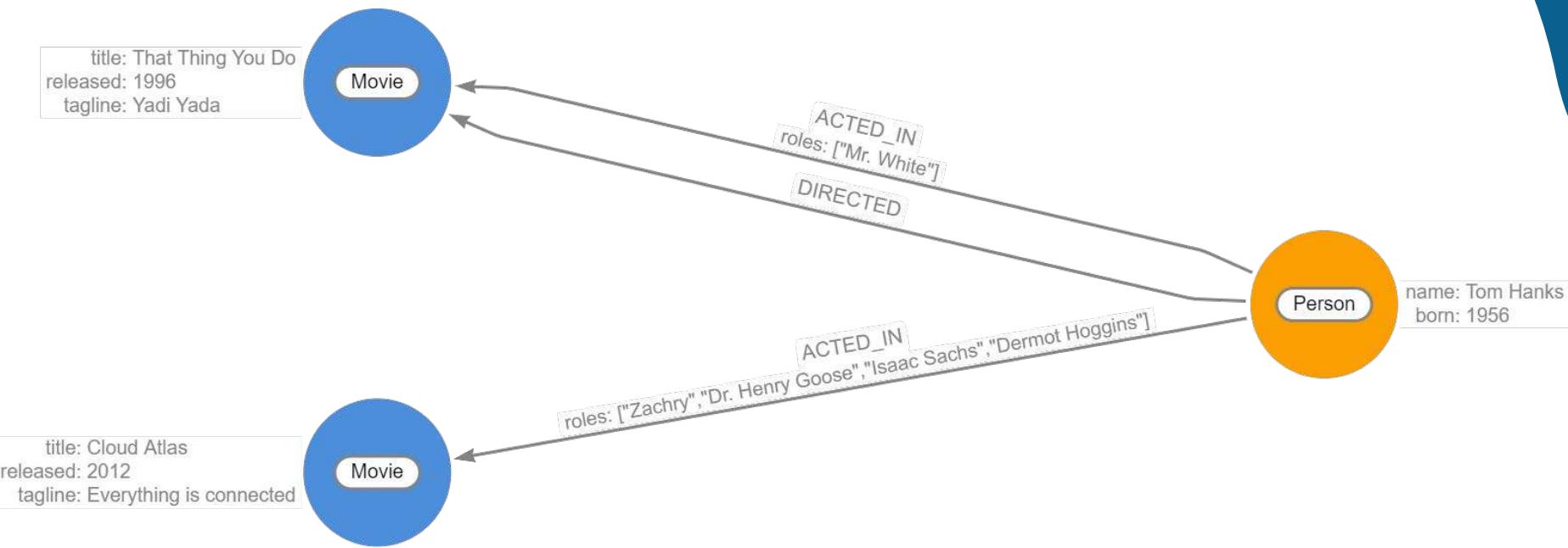
```
MERGE (me:Person {name:'Stefano'})  
WITH me AS me  
UNWIND ['Roller skating', 'Acroyoga', 'Rants'] as interest  
MERGE (t:Topic {name: interest})  
MERGE (me)-[:LIKES]->(t)  
RETURN me
```



More practice!



Movies model



MATCH Nodes

()

(:Person)

(:%)

(:!(Person)&(Actor|Actress))

(x WHERE x.name = "John Doe" AND x.age > 50)

(x:!(Person)&(Actor|Actress) WHERE x.name = "John Doe")

For an efficient pattern match, you should be as precise as possible!

Relationships – MATCH vs CREATE

- When CREATE-ing, you have to provide a direction...

CREATE (me)-[:LIKES]->(topic)

CREATE (me)<-[:LIKES]-(person)

- ... but when MATCH-ing, you can ask for undirected!

MATCH (me)-[:LIKES]->(topic)

MATCH (me)-[:LIKES]-(person)

MATCH Relationships

()-[]-()

- No direction = either direction
- Can also be written as ()--()

Matches any single hop relationship between any two nodes.

MATCH Relationships

`()-[]-()`

`()-[]->()` (or `()<-[]-()`)

- Can also be written as `()-->()`

Matches any single hop relationship between any two nodes.

MATCH Relationships

`()-[]-()`

`()-[]->()`

`()-[:ACTED_IN]->()`

Matches any single hop relationship between any two nodes filtering for a specific type of relationship.

MATCH Relationships

`()-[]-()`

`()-[]->()`

`()-[:ACTED_IN]->()`

`()-[:ACTED_IN|DIRECTED]->()`

Matches any single hop relationship between any two nodes filtering for several specific types of relationship.

MATCH Relationships

`()-[]-()`

`()-[]->()`

`()-[:ACTED_IN]->()`

`()-[:ACTED_IN|DIRECTED]->()`

`()-[ai:ACTED_IN WHERE ai.roles = ["Herself"]]->()`

Matches any single hop relationship between any two nodes filtering for a specific type of relationship and also filtering on the relationship property.

MATCH Relationships

There's more ...

```
MATCH sg=(b:Bank WHERE b.name =  
"Fideuram")-[*1..3]-(c:Country WHERE c.name = "Panama")  
RETURN sg;
```

Practice time!

1. Find 10 actor names.
2. Find how many movies have been released after 2005.
3. Create a WATCHED relationship between yourself and Cloud Atlas. Show this relationship.
4. Find who directed V for Vendetta.
5. Find people who have acted in movies released after 2005.
6. Find all people who have co-acted with Kevin Bacon (in any movie).
7. Find people 3 steps away from Tom Hanks.
8. Find (and show) the shortest path between Bill Paxton and Gary Sinise.



slido



If you had a magic wand, what is one thing you would like to make happen this saturday?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

Cheaty solutions!

1. MATCH (p:Person) RETURN p.name LIMIT 10
2. MATCH (m:Movie WHERE m.released > 2005) RETURN COUNT(*)
MERGE (ste:Person {name: 'Stefano'})
MERGE (film:Movie {title: 'Cloud Atlas'})
MERGE sg=(ste)-[:WATCHED]->(film)
RETURN sg
3. MATCH (p:Person)-[:DIRECTED]->(m:Movie {title:'V for Vendetta'}) RETURN p.name
4. MATCH (p:Person)-[:ACTED_IN]->(m:Movie WHERE m.released > 2005)
RETURN p.name
5. MATCH sg=(p:Person)-[r:ACTED_IN]->(m:Movie)<-[r:ACTED_IN]-(:Person
{name: 'Kevin Bacon'}) RETURN sg
6. MATCH sg=(p:Person)-[*1..3]-(:Person {name: 'Tom Hanks'}) RETURN sg
7. MATCH sg=shortestPath((p:Person {name: 'Bill Paxton'})-[*]-(:Person {name:
'Gary Sinise'})) RETURN sg



Interesting (?) queries...

This can get Messi ...

```
CREATE () - [:DOESMORETHANMOSTTHINK] -> () ;
```



Interesting (?) queries...

```
MERGE (p:Person  
       {name: 'Stefano'})
```

```
MERGE (p:Topic  
       {name: 'Rants'})
```

```
MERGE (p)-[:LIKES]->(t)
```

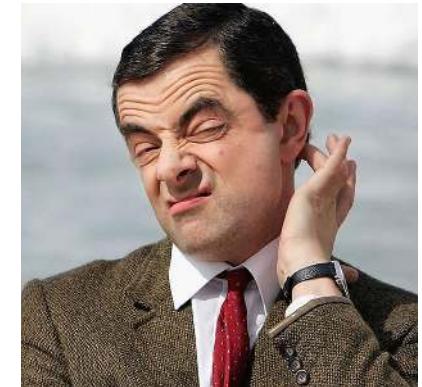
No error! Cypher queries are *pipelines*!

SQL query – Interpreted as a whole.

Cypher query – If at any stage the number of results drops to zero, the remainder of the pipeline is *not* executed.

VS

```
MATCH (p:Person  
       {name: 'Fabio'})  
MERGE (p:Topic  
       {name: 'Acroyoga'})  
MERGE (p)-[:LIKES]->(t)
```



Interesting (?) queries...

What is being counted here?

```
MATCH (ste:Person WHERE tom.name = "Stefano  
Hanks") -[ai:ACTED_IN]-> (m:Movie)  
RETURN count(ste);
```

Patterns. It counts matching patterns. Try with count(ai), count(m) or count(*), it makes absolutely no difference. It. Counts. Matching. Patterns.

What is this query doing?

```
MATCH (m:Member WHERE m.name = "Stefano") - [:MEMBER_OF] ->
(g:Group) - [:HAS_TOPIC] -> (t:Topic)
WHERE NOT EXISTS ((m) - [:INTERESTED_IN] -> (t))
WITH m, t, count(*) AS weight
WHERE weight > 2
CREATE (m) - [:INTERESTED_IN] -> (t);
```

It infers a relationship - i.e. gives a recommendation.

Magical Mr White



Magical Mr White

How many results does this query return?

```
MATCH () - [x:ACTED_IN WHERE x.roles = ["Mr. White"] ] - ()  
RETURN x;
```

Hint: I did not lie, there is only one relationship containing ["Mr. White"] as the roles property in the database ...



**How many results does the query `MATCH
()-[x:ACTED_IN WHERE x.roles = ["Mr.
White"]]-()
RETURN x;` return?**

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

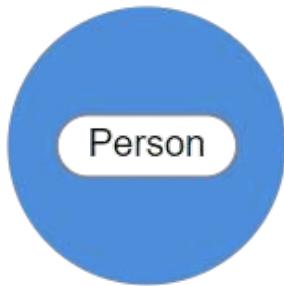
Questions?

I may have answers.

Modeling for graph databases

My favorite pair

name: Brad Pitt
born: 1963



name: Angelina Jolie
born: 1975
shoesize: 9



Well, trio really ...

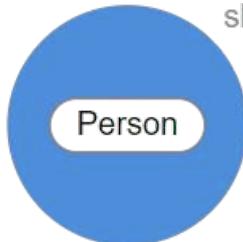
name: Lassie
born: 1938



name: Brad Pitt
born: 1963



name: Angelina Jolie
born: 1975
shoesize: 9

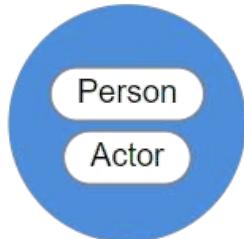


All three have a job

name: Lassie
born: 1938



name: Brad Pitt
born: 1963



name: Angelina Jolie
born: 1975
shoesize: 9



Two got married



But not for long



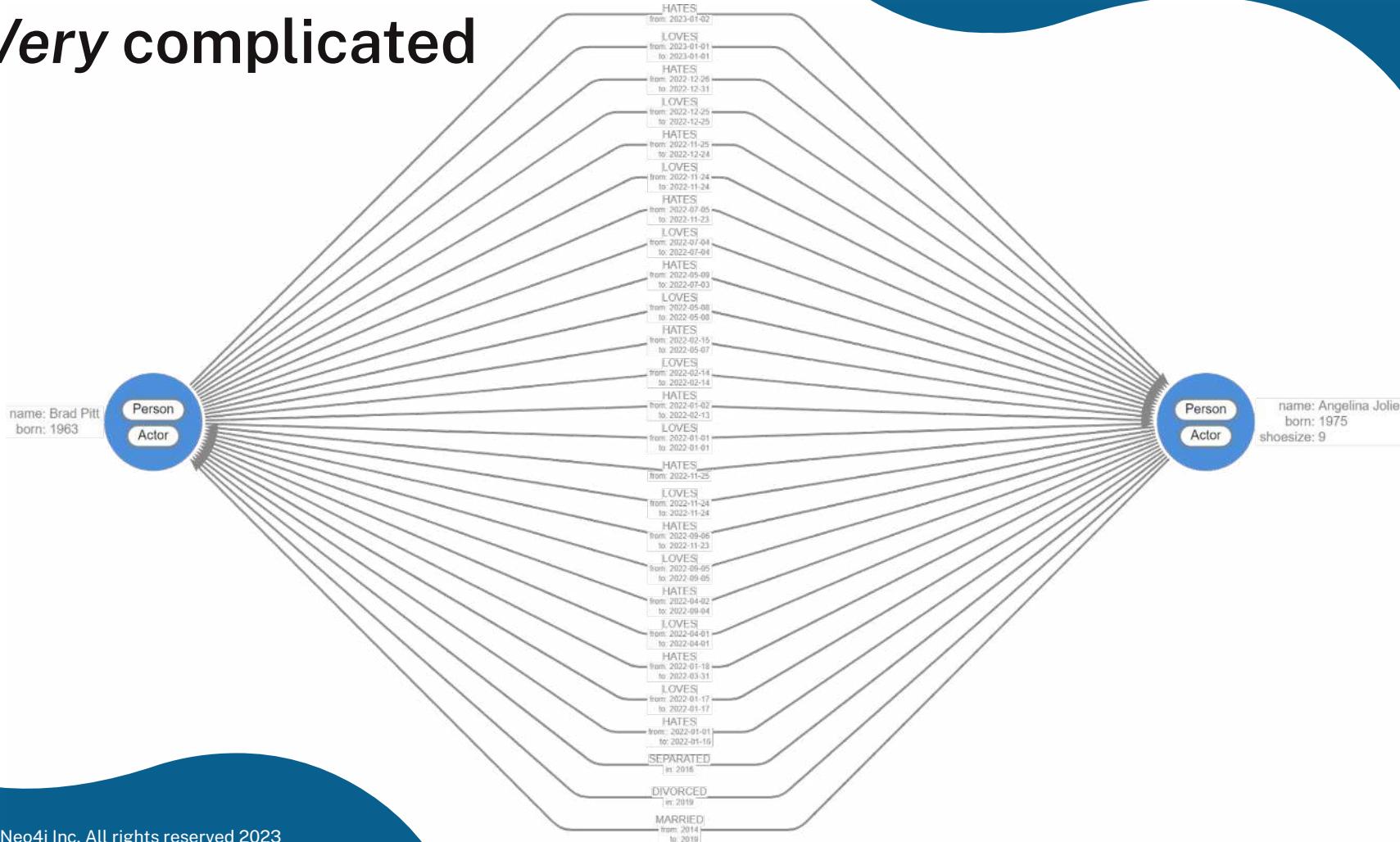
Inconsistency crops up ...



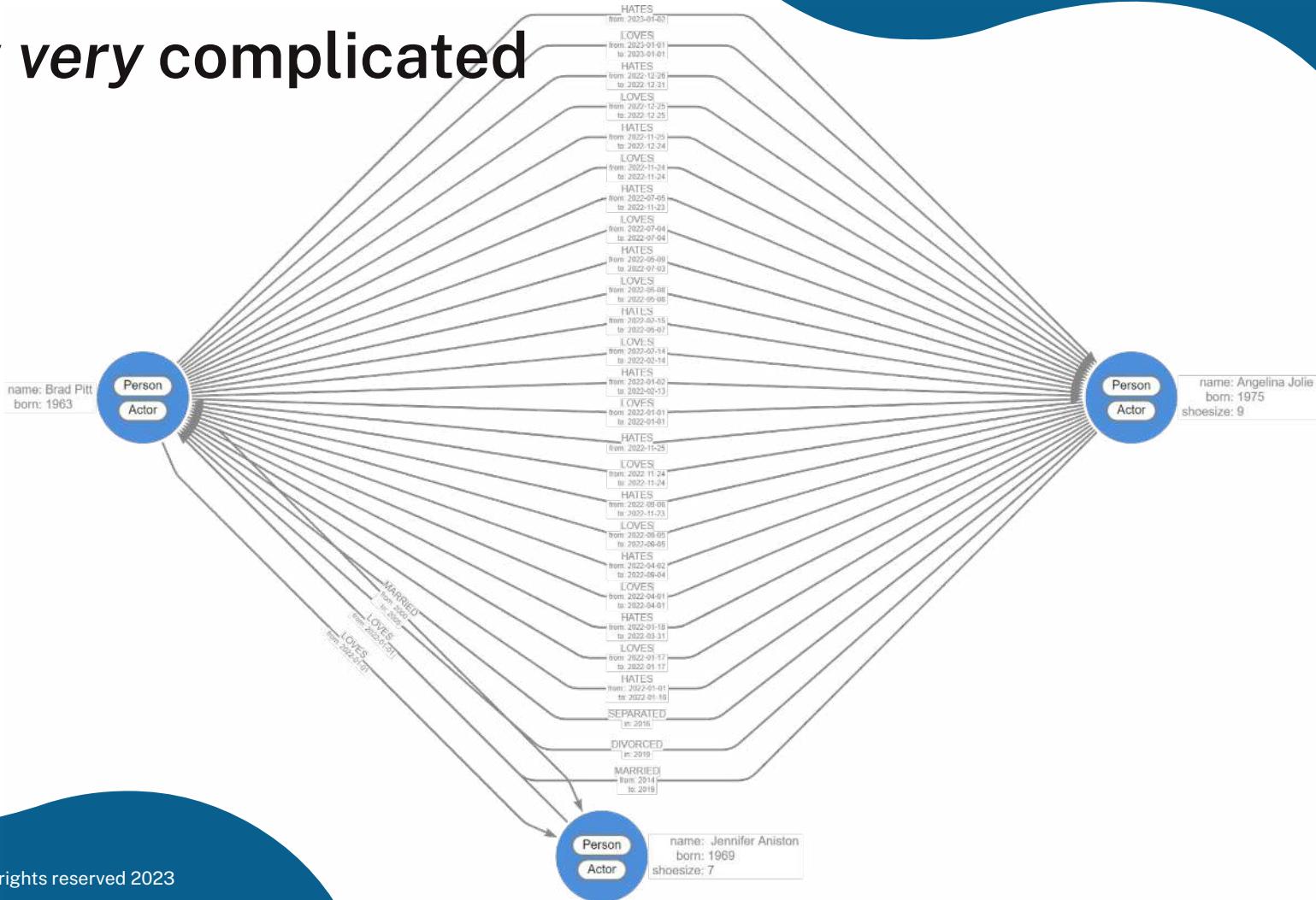
It is complicated



Very complicated



Very very complicated



Remodeling

A marriage is between two persons, but more people are involved (witnesses, officers, ...)!

How to include them?



Help the demon remodel!



No amount of remodeling can fix this...

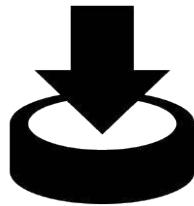


Graph modeling workflow

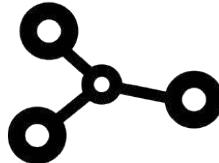
Ch-ch-ch-changes



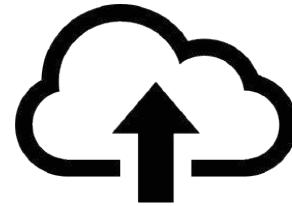
1. Derive the question



2. Obtain the data



3. Develop a model



4. Ingest the data



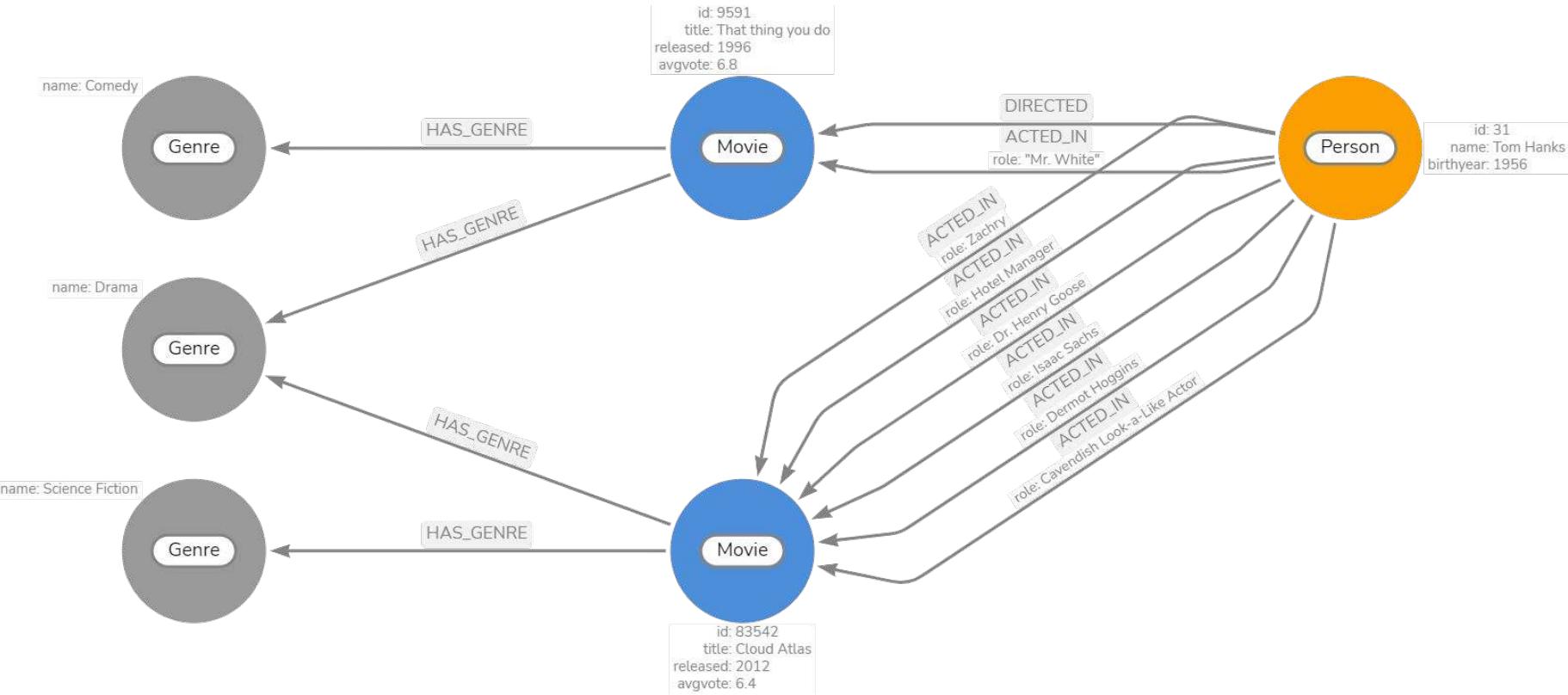
5. Query/Prove our model

Demo(n)



bit.ly/neo4junige

Model change



In conclusion - Intermediate nodes

Nothing wrong with

- Getting more intermediate nodes over time
- Seeing the model evolve towards what looks like a model that would also fit an RDBMS. There are only so many ways to model a problem domain, that's fine.

But! Allow the model to evolve based on the questions.

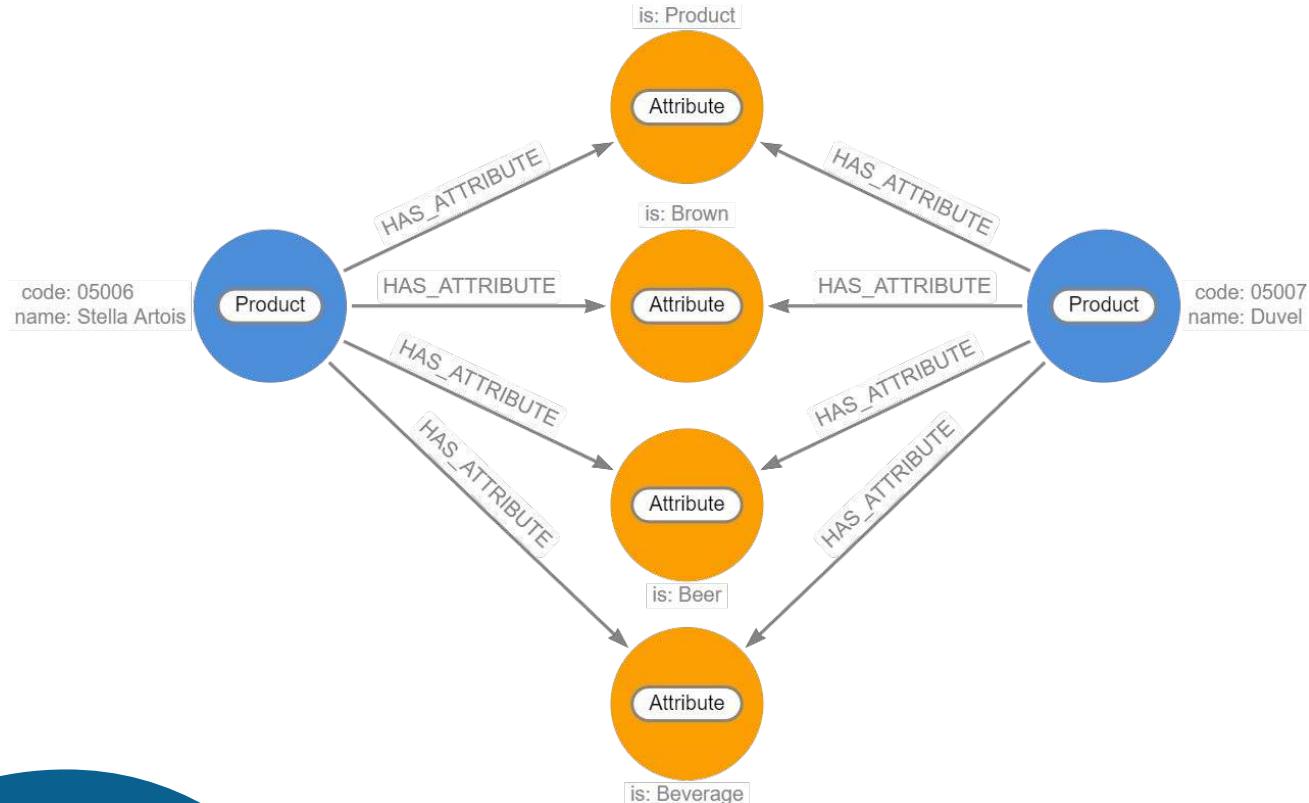
Business evolves ever faster. A graph helps you keep up with that!

The Justin Bieber problem

Nothing funny
to say about
this guy ...



Discussing the problem



Cost of change II - Demo(n)



Questions?

I may have answers.

Practice - Modeling

Create a graph model for an application that manages and coordinates a university campus.

You get 10 minutes to do it.

I will then pick two random people to present their solutions.



Everybody that answered is wrong!

There is no question in the statement!

Practice - Modeling

Create a graph model for an application to manage and coordinate a university campus and allows to answer/model the following:

1. What courses does a department offer?
2. What courses has/is a student completed/attending?
3. Get in touch with students who completed a course to receive feedback.
4. What courses have the lowest average exam score?
5. Which professors are performing worst, when looking at exam scores across all courses they teach?
6. What (new) research topics could be recommended to professors?



slido



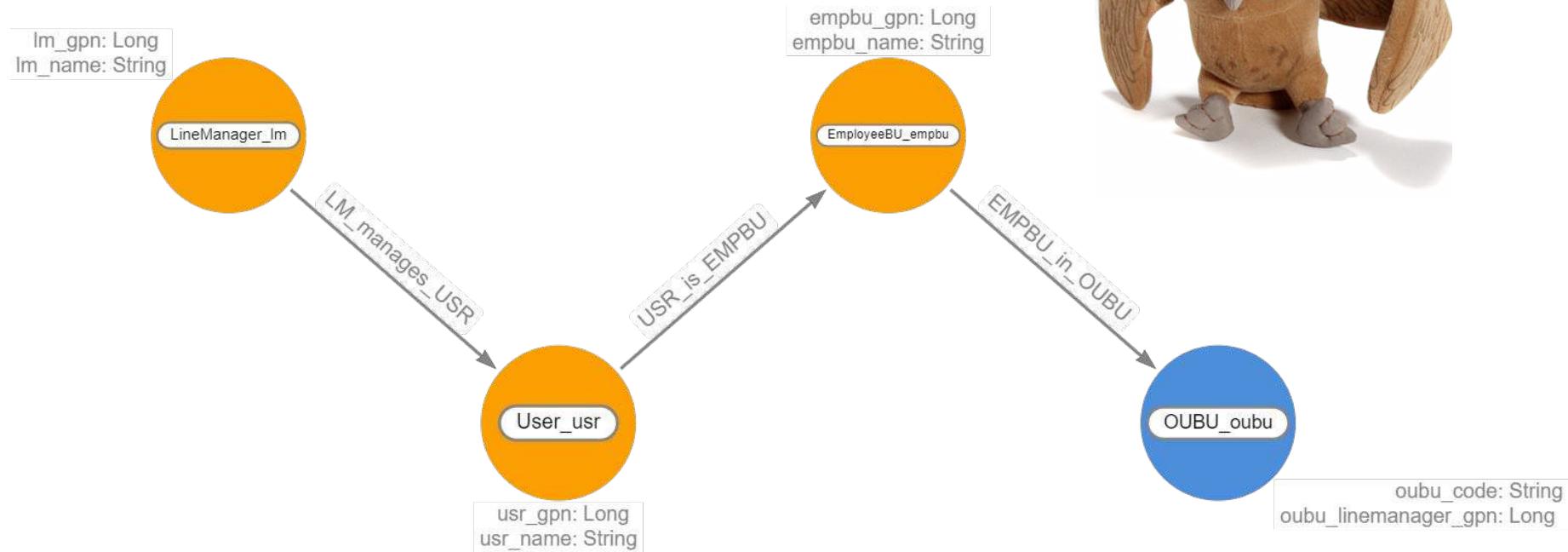
If you could be the best in
the world in one skill, what
would it be?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

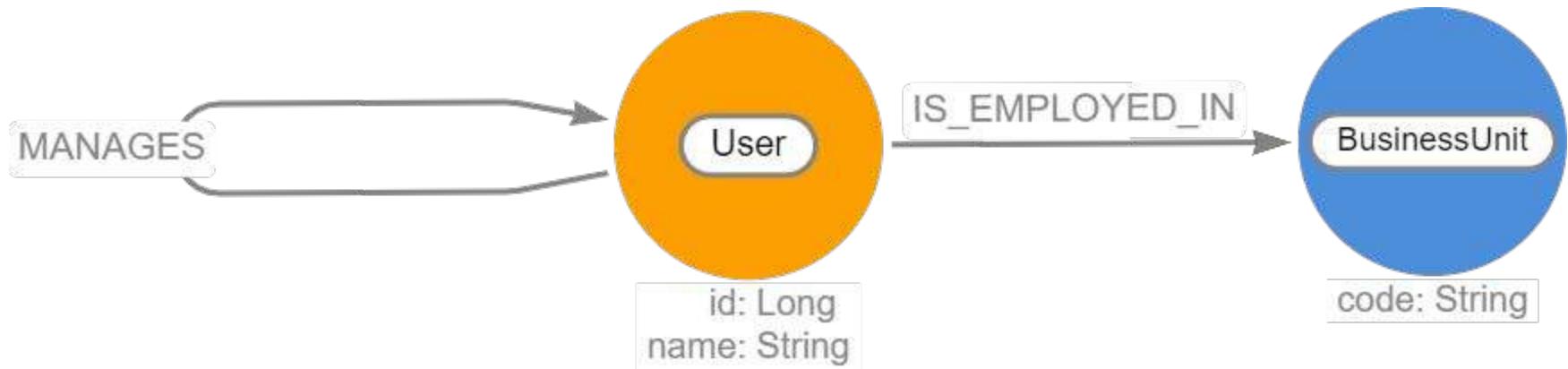
Modeling in practice

aka Stefano's Rants

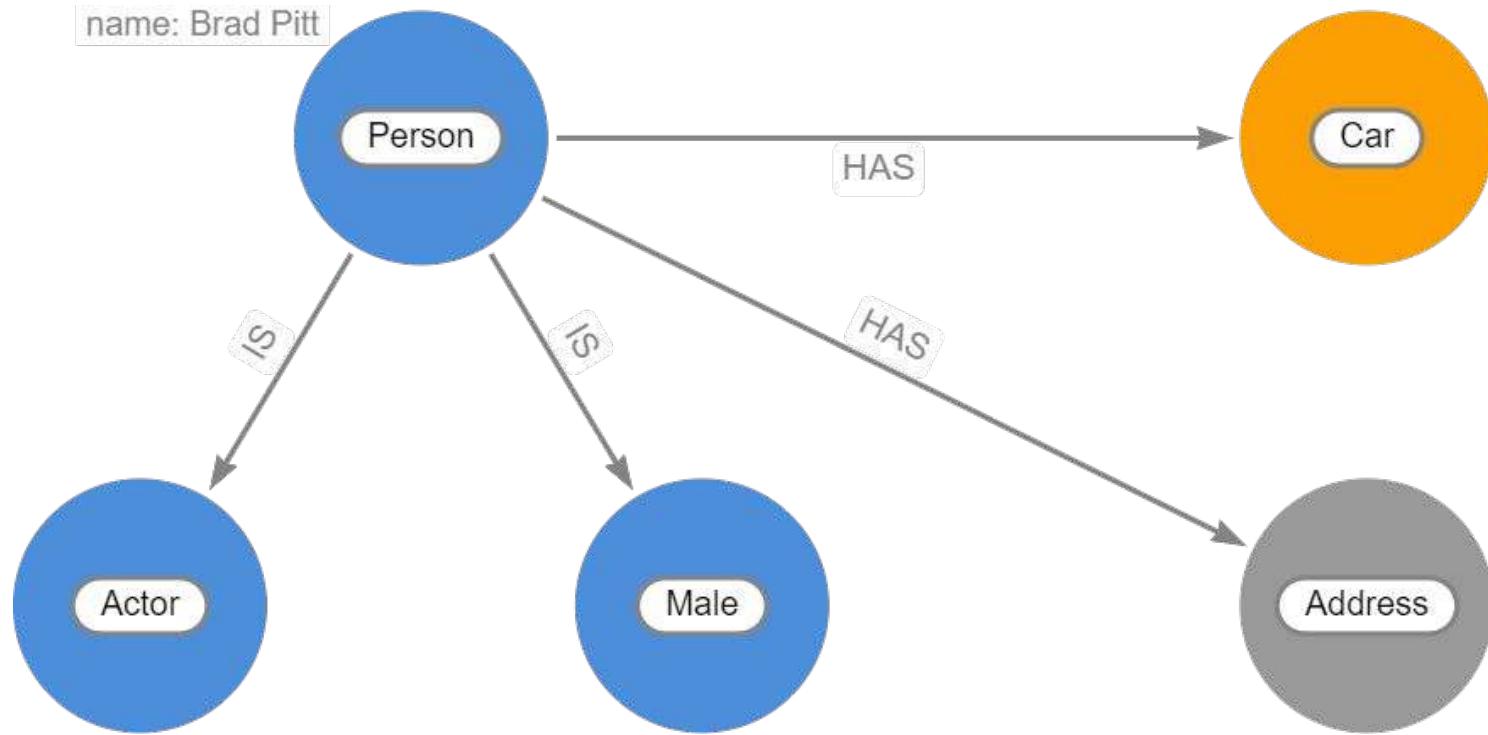
Let's discuss this



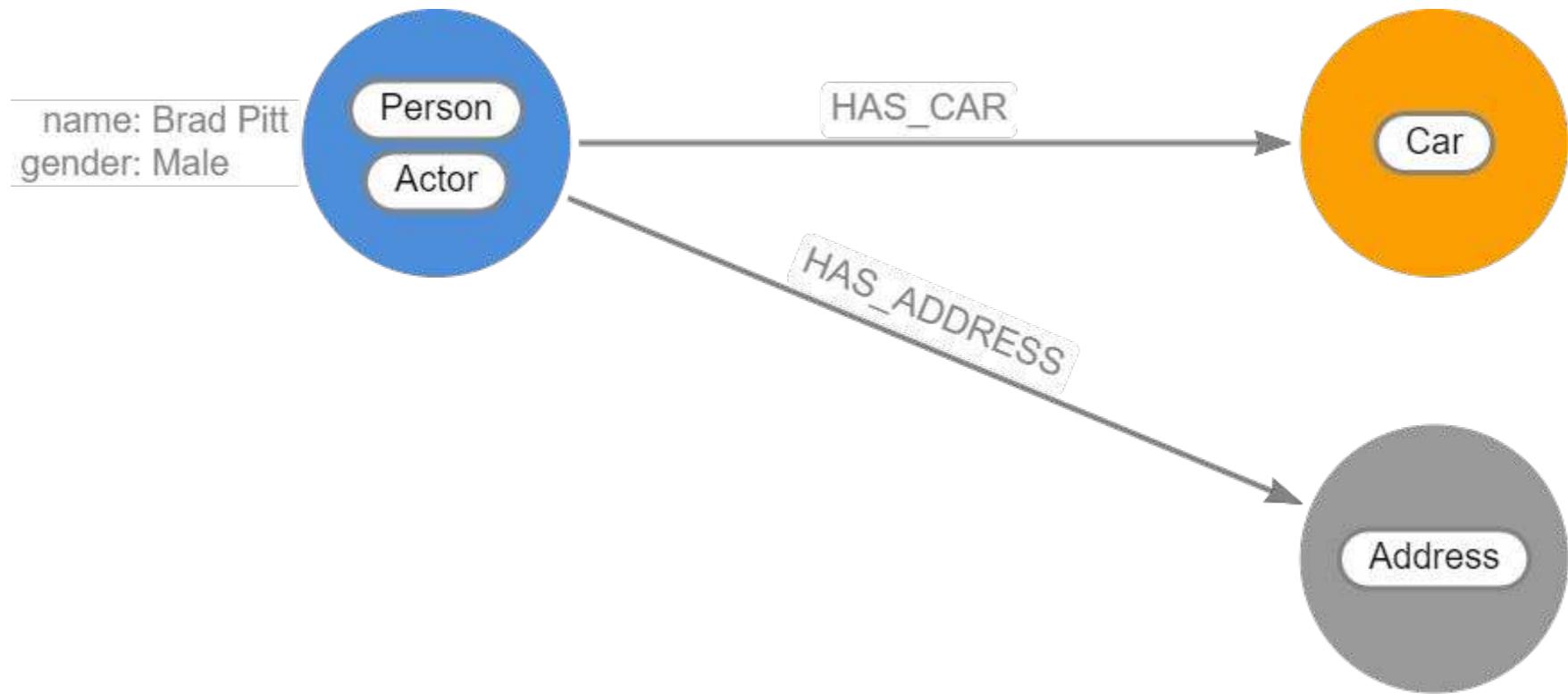
Let's discuss this



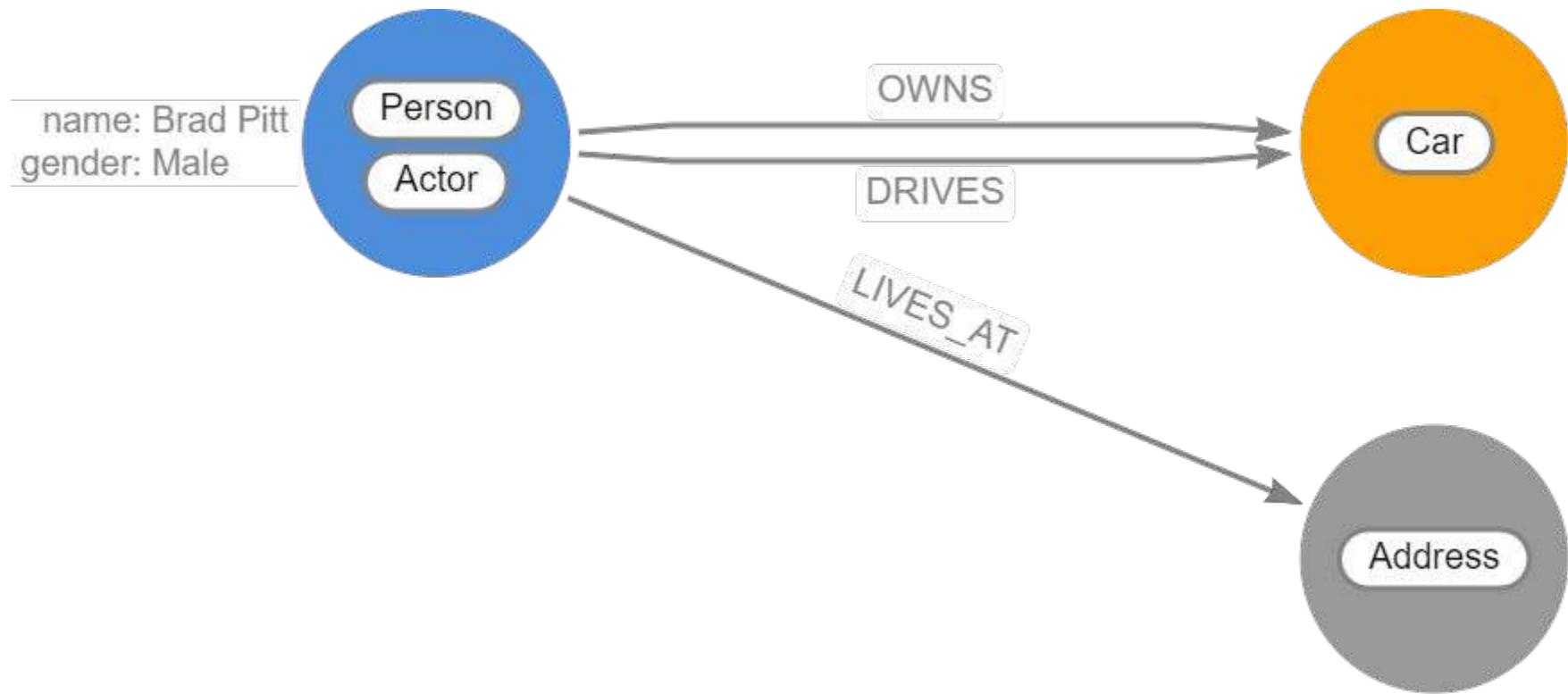
IS'ms and HAS'ms



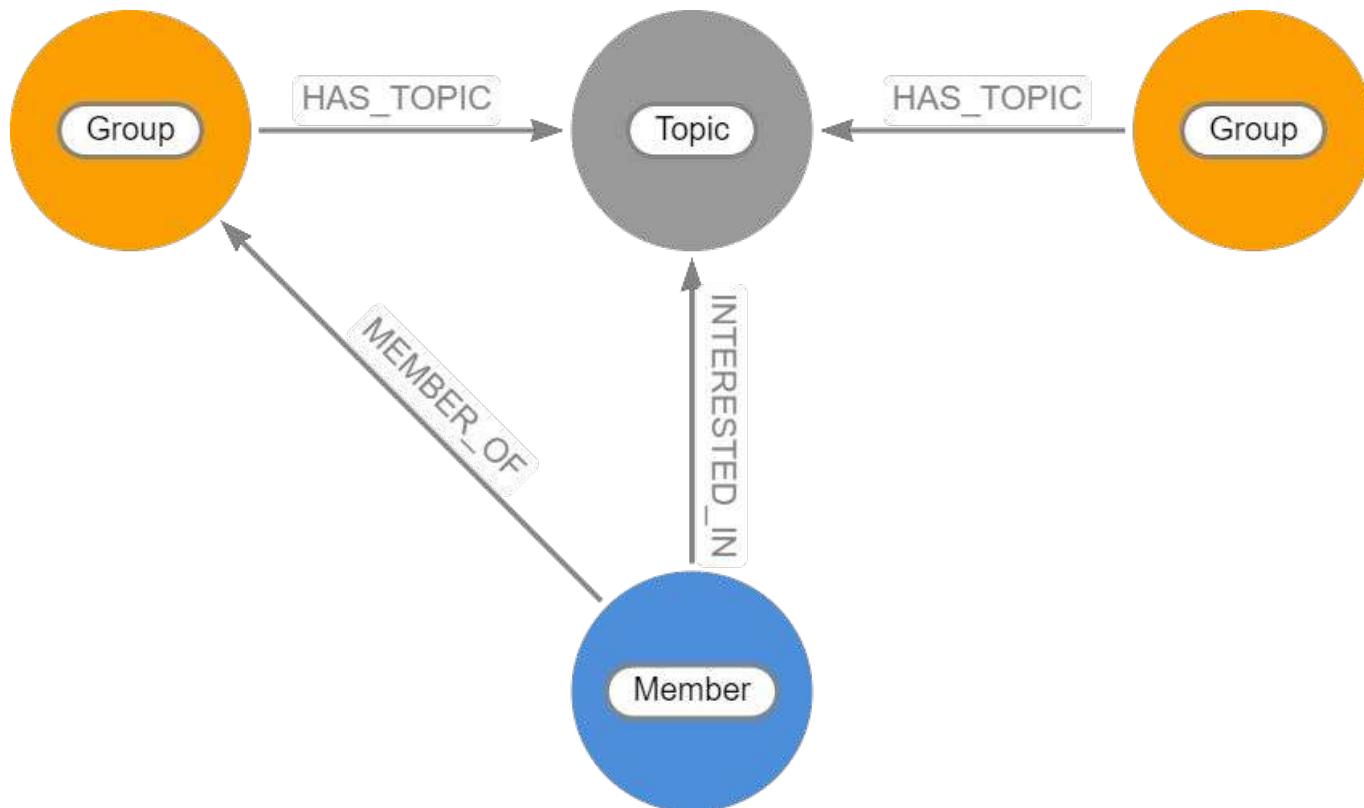
IS'ms and HAS'ms



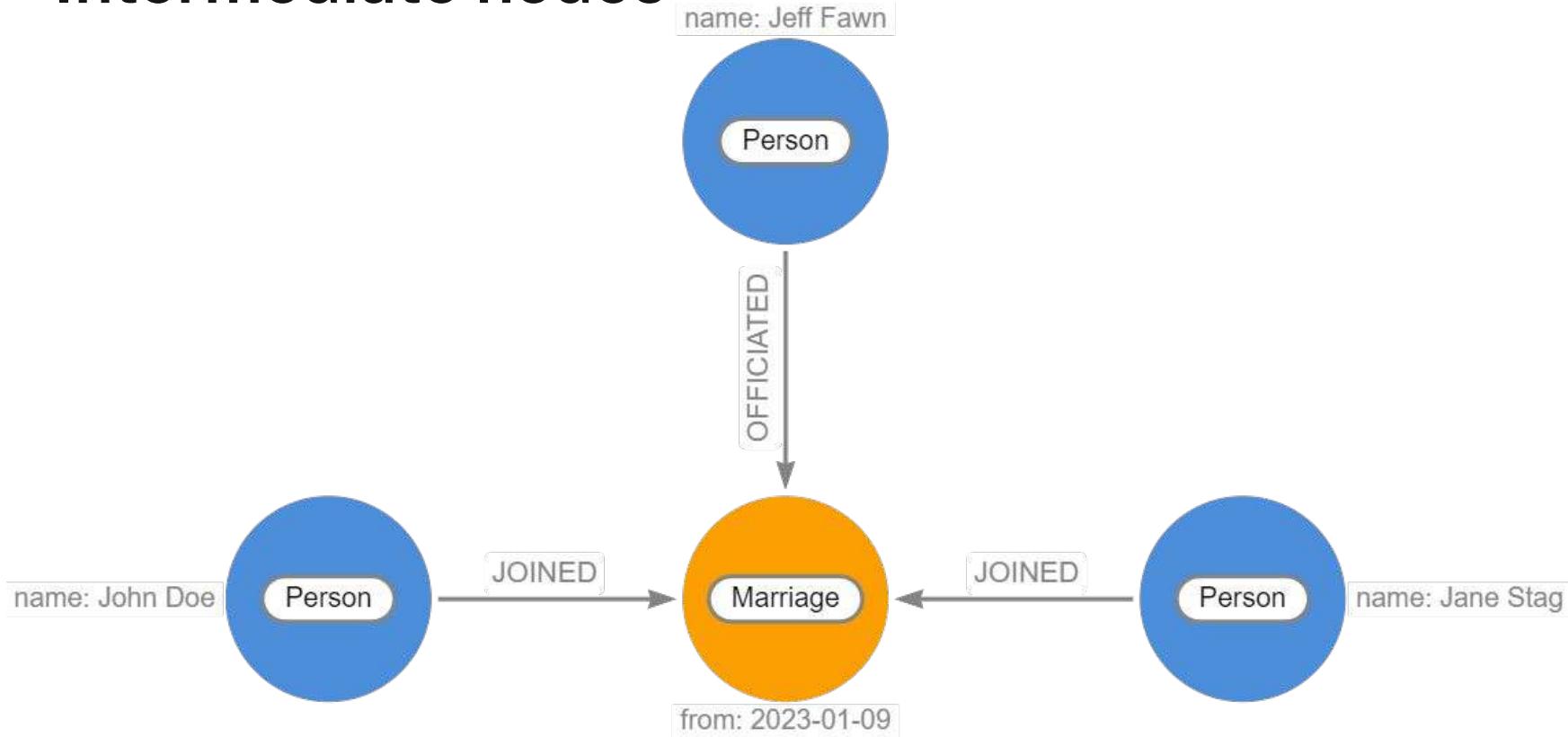
IS'ms and HAS'ms



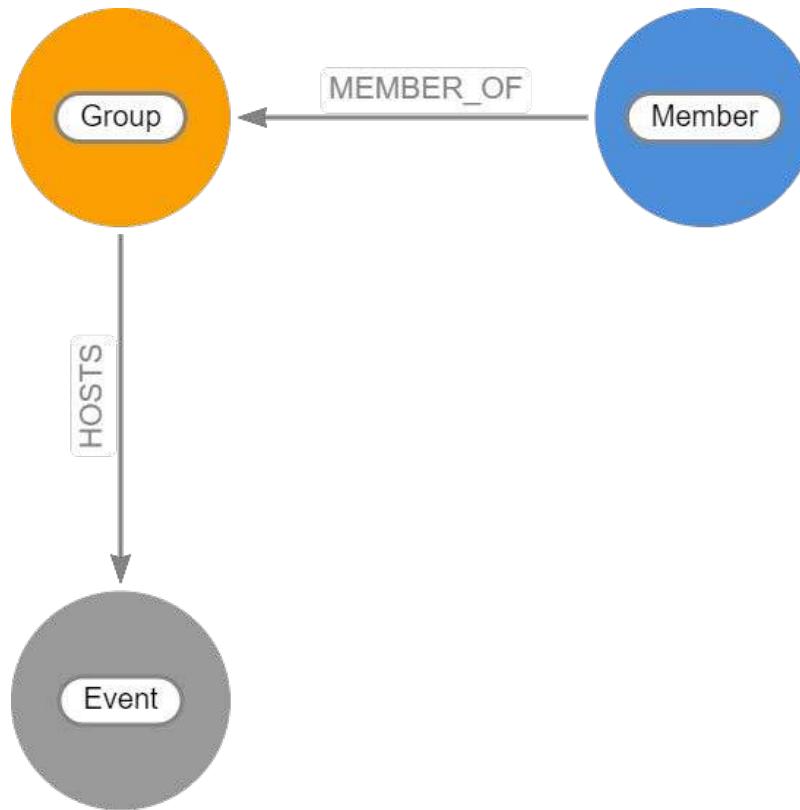
Inferring a relationship



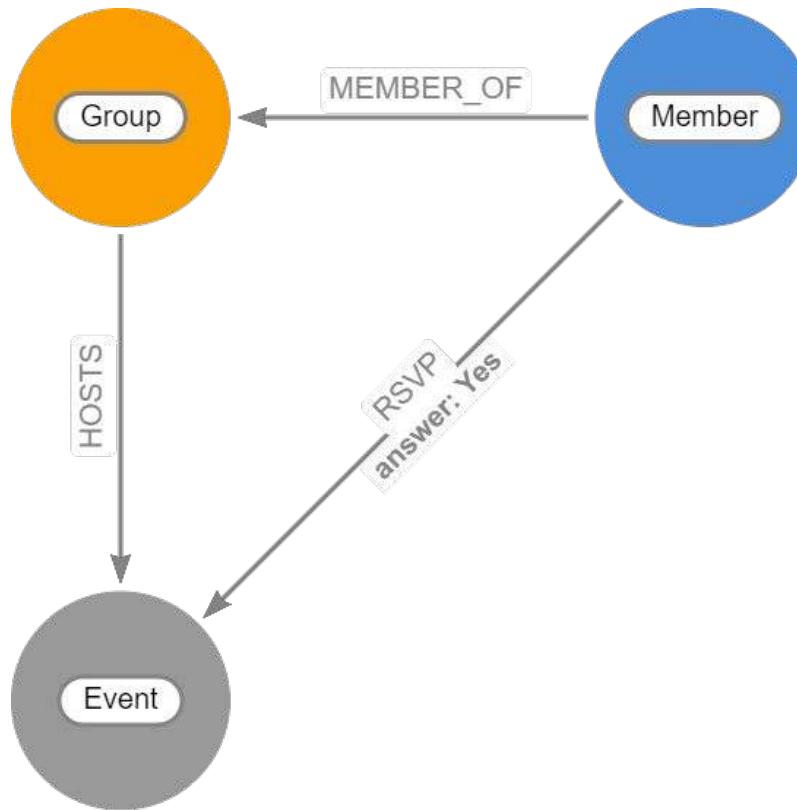
Intermediate nodes



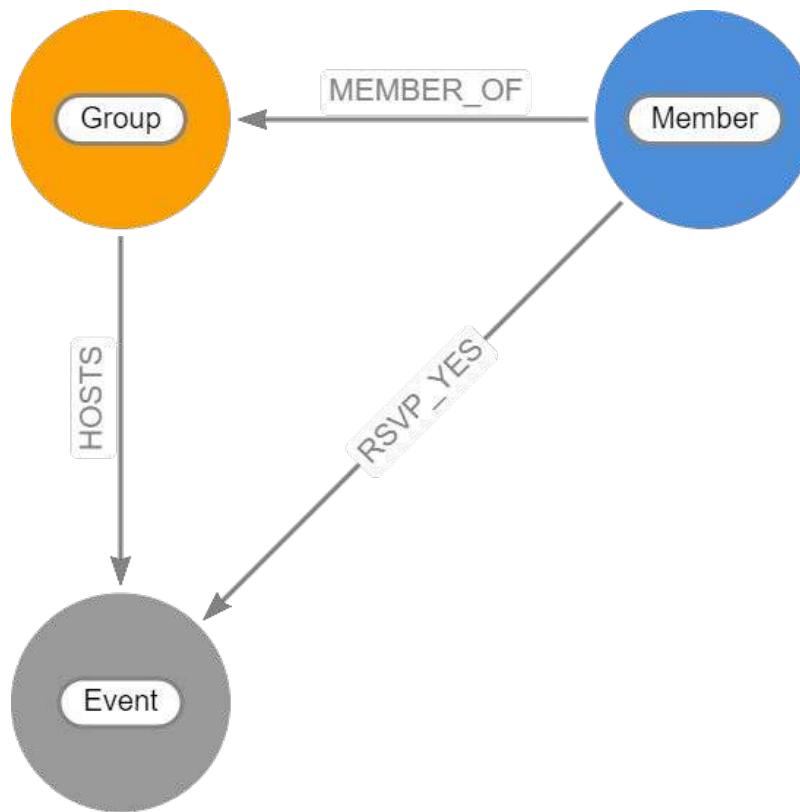
Leveraging relationship types



Leveraging relationship types



Leveraging relationship types



Relational to graph

Translation

- Rows in tables -> nodes
- Foreign keys -> relationships
- Link tables -> relationships (possibly with properties)
- Remove artificial constructs (extra primary and foreign keys, for example)

Nothing is lost in this translation.

However, do verify if there is something that can be **gained** in the translation!

A word on primary/foreign keys

Unique/Primary keys



Secondary/Foreign keys



Feedback



*“Can we make
the pig sexier?”*



stefano.ottolenghi@neo4j.com
stejey@gmail.com



Overall, how would you rate your experience today?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

slido



What has the largest friction point with using Neo4j been?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.



One thing you particularly liked/appreciated in today's session?

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.



**One thing you'd have liked
done differently in today's
session?**

- ⓘ Click **Present with Slido** or install our [Chrome extension](#) to activate this poll while presenting.

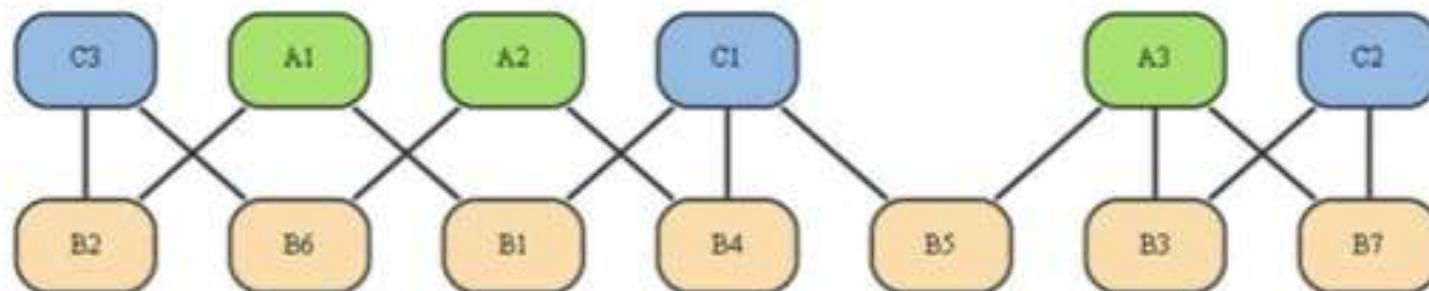
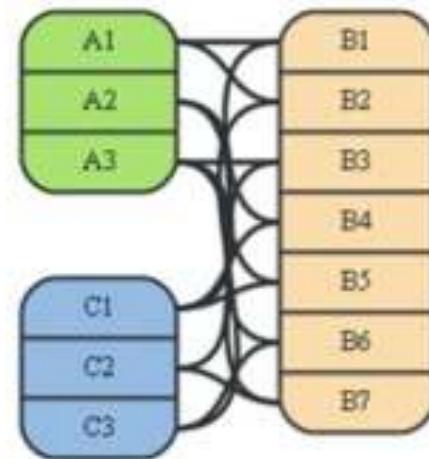
Graph Databases - neo4j

Outline

- Short intro to graph databases
- Neo4j
 - Introduction
 - Data model & interaction
 - Architecture
- Use cases

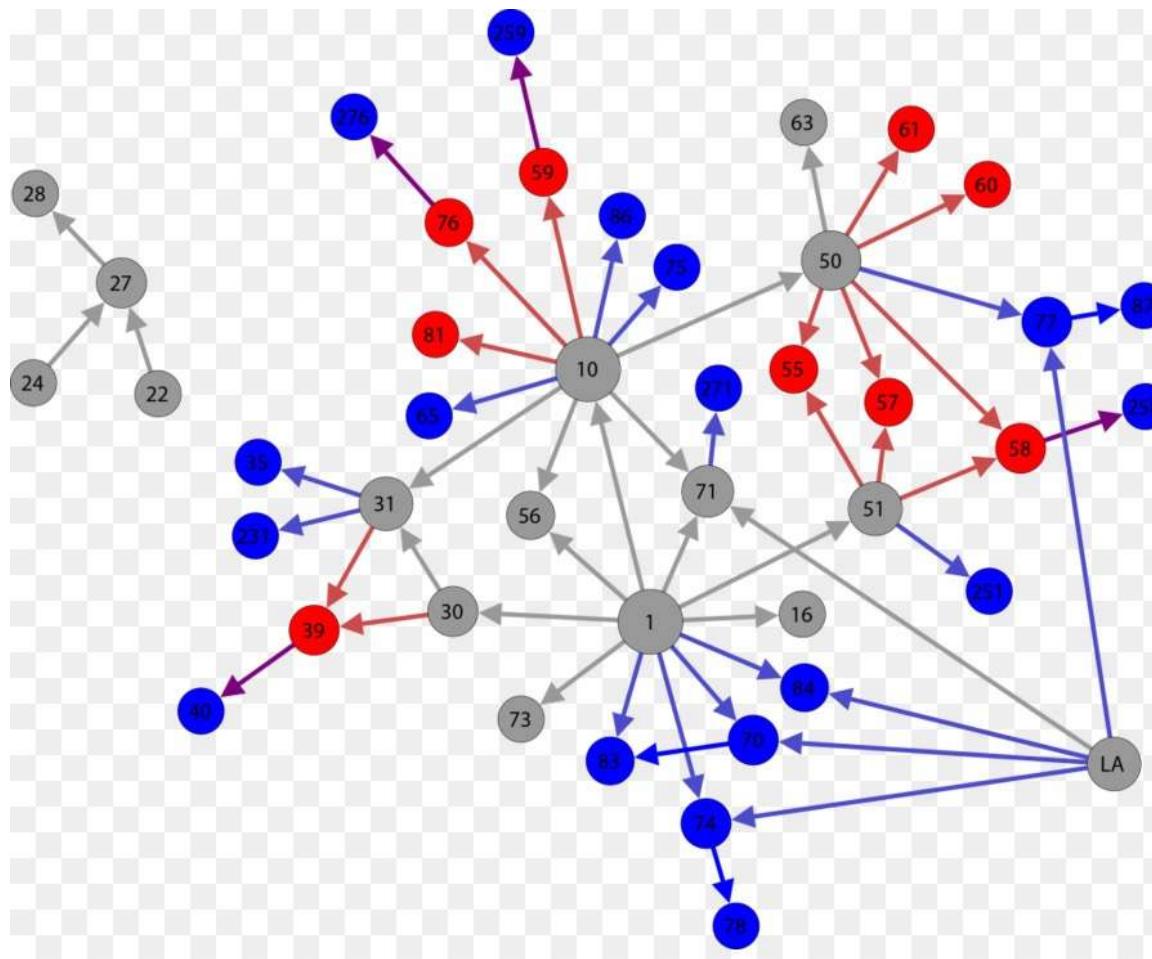
Graph databases

- Graph databases are motivated by the issues of relational databases in coping with complex relationships
- Complex relationships require complex join



Graph databases

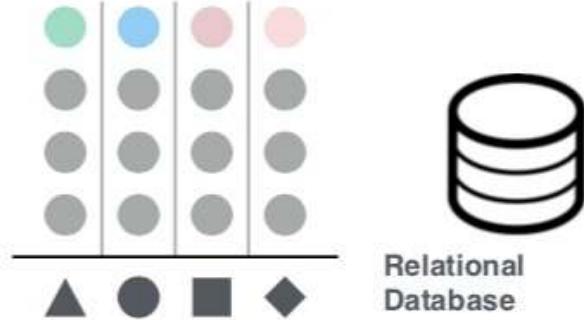
- Capture data consisting of complex relationships
- Focus on data inter-connection and topology



Why Graphs? (recall)

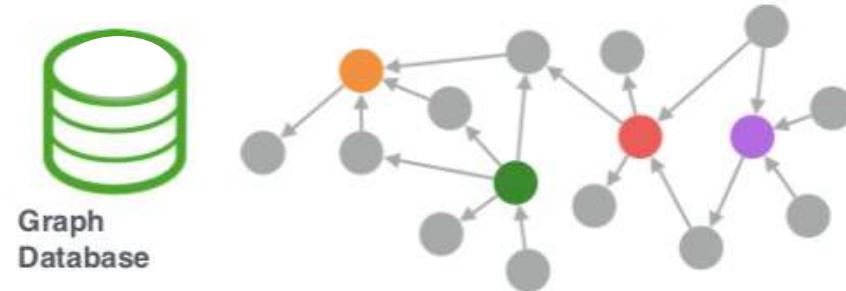
- Well-known generic data structure
- One way to address the impedance mismatch
(objects in your Java don't match the structure of a DB)
- Maths and algorithms are well understood
- «blackboard friendly»
- Data naturally modelled as graphs

Which is Good at What?



Good for:

- Well-understood data structures that don't change too frequently
- Known problems involving discrete parts of the data, or minimal connectivity



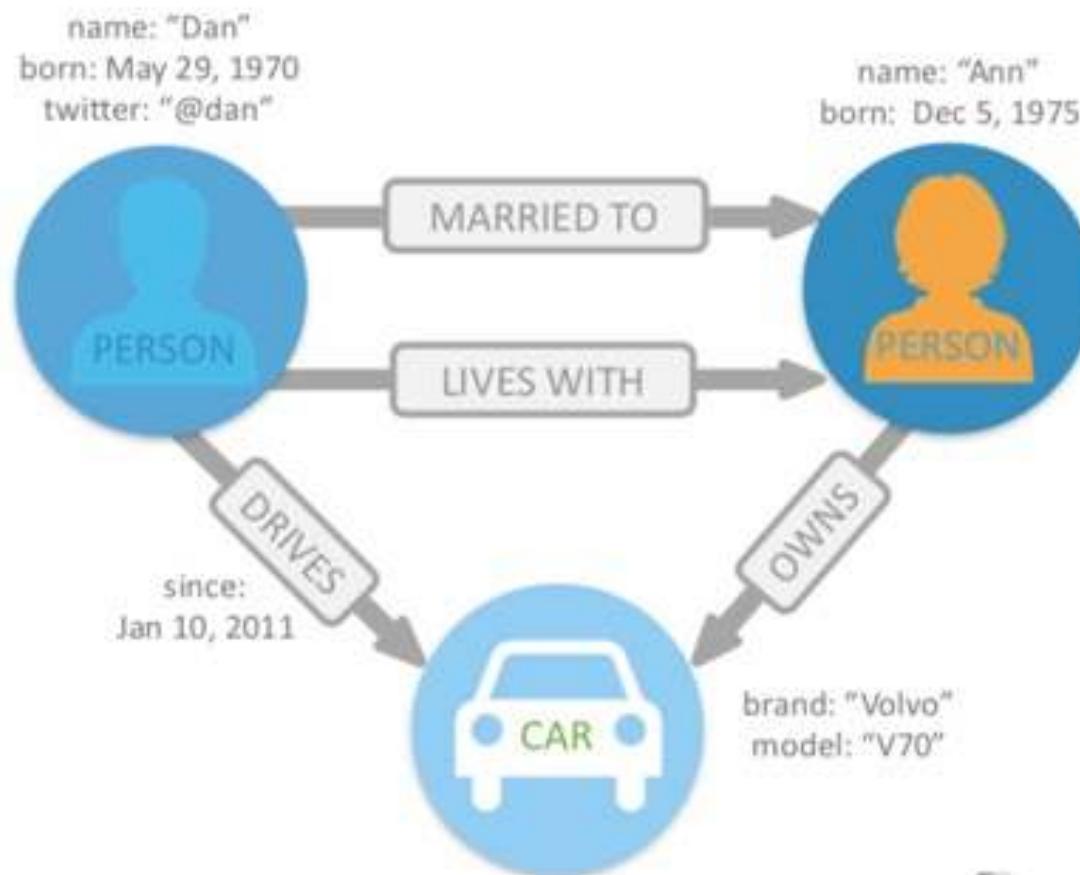
Good for:

- Dynamic systems: where the data topology is difficult to predict
- Dynamic requirements: that evolve with the business
- Problems where the relationships in data contribute meaning & value

Graph Model

- Complex, densely-connected domains
 - Lots of join tables? Relationships
 - Lots of sparse tables? Semi-structure
- Messy data
 - Ad hoc exceptions
- Relationships as **first-class elements**
 - Semantic clarity: named, directed
 - Not simply constraints
 - Can have properties

Graph Data Model (property graph)



NoSQL databases

graph vs aggregate-oriented databases

Graph databases

- graph-based query languages
- **partitioning is difficult:**
more likely to run on a single server
- transactions maintain **consistency** over multiple nodes and edges

Aggregate-oriented

- simple query languages
- distributed across clusters
- no ACID guarantees

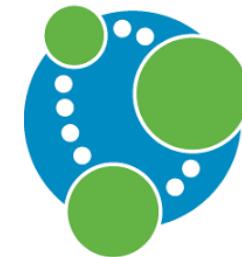
Popular graph dbs/systems



Property graph model: Neo4j, Titan, InfiniteGraph, ...

Triple store model (RDF): AllegroGraph, Virtuoso, ...

Neo4j - Introduction

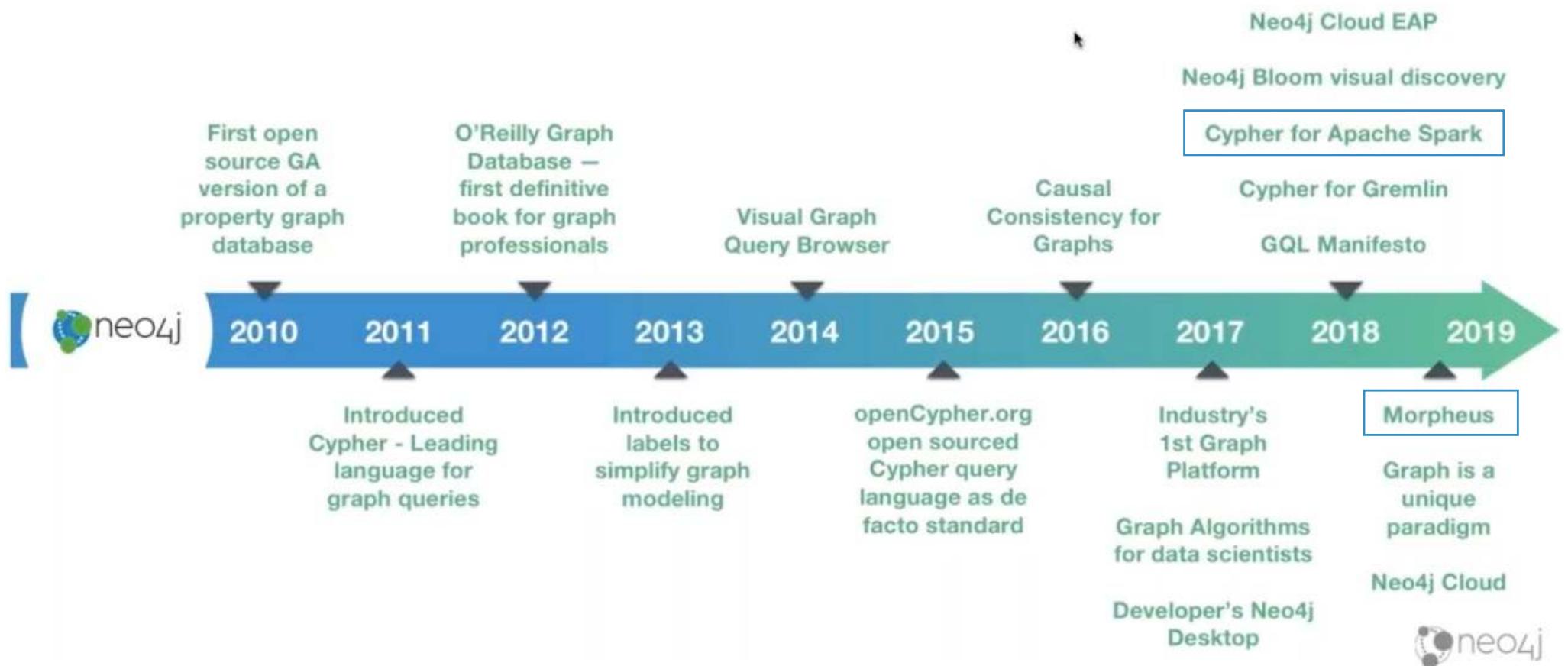


neo4j in short

| Feature | neo4j |
|---------------------|---|
| Model | Graph-based |
| Query language | Supported, Cypher |
| Reference scenarios | transactional (read intensive) & analytical |
| Partitioning | Difficult (application-level) |
| Indexes | on properties (simple and composite), full-text |
| Replication | Master-slave, single-leader |
| Consistency | Strong |
| Availability | Load balancing among read replicas |
| Fault tolerance | By re-electing a master in case it goes down |
| Transactions | ACID (maintain consistency over multiple nodes and edges) |
| CAP theorem | CA |
| Distributed by | Neo4j Inc. |

neo4j

- First released in 2010
- Developed in Java
- Community and Enterprise versions, dual-licensed: GPL v3 and a commercial license
 - the Community Edition is free but is limited to running on one node only due to the lack of clustering
- Query language: Cypher
- Java traversal API



Who uses neo4j?



v o l v o



<https://neo4j.com/customers>

Data model and interaction

Property Graph, Traversals, and Cypher Query Language

Property Graph Model

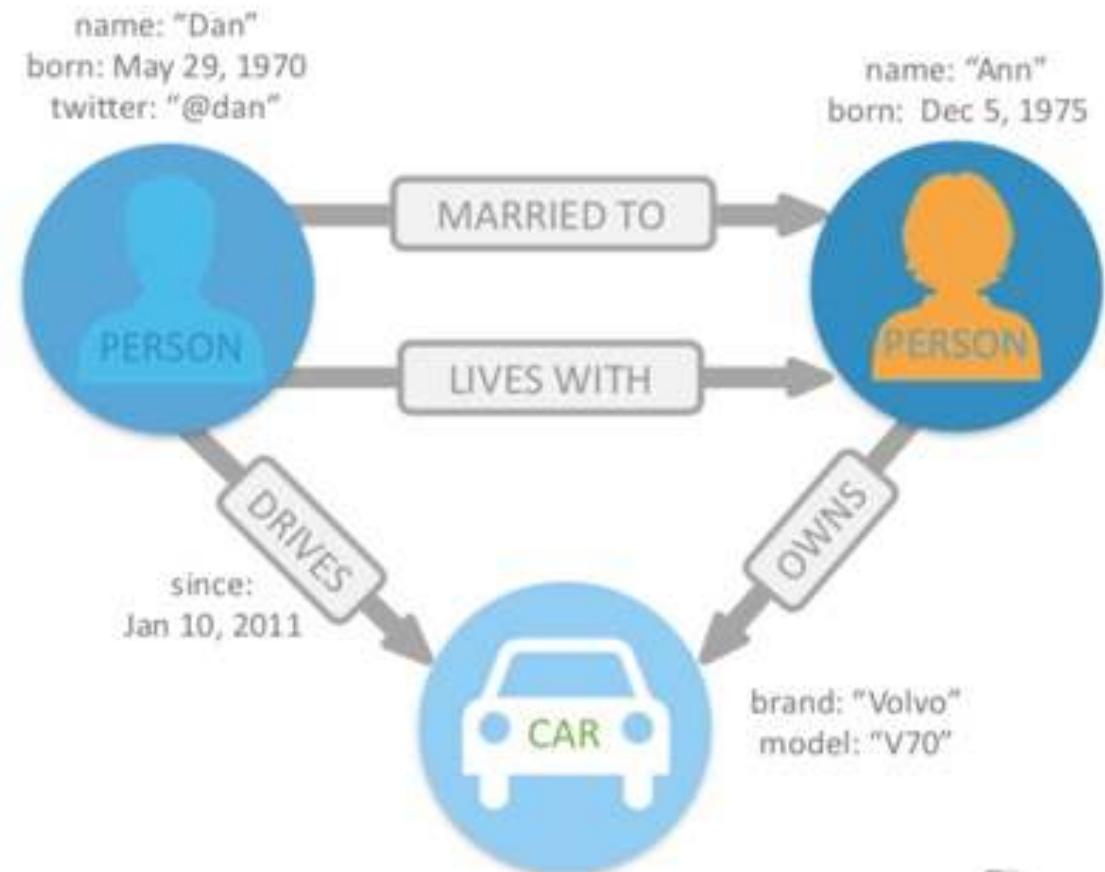
A property graph $G = (V, E)$ is a **directed multigraph** where:

- every node $v \in V$ and every edge $e \in E$ can be associated with a set of pairs $\langle \text{key}, \text{value} \rangle$, called **properties**
- value can be of primitive type or an array of primitive type
- every node $v \in V$ can be tagged with one or more **label(s)**
- every edge $e \in E$ is associated with a **type**

Property Graph Model

Edges have incoming/outgoing edges but traversal is equally efficient in both directions

Schema-less model



Graph vs Relational Data Model

- The **relational model** is designed for a single type of **relationship**
 "
- **Adding** another relationship usually means a lot of **schema changes**
- In RDBMS **we model** the graph beforehand based on the **traversal** we want
 - If the traversal changes, the data will have to change

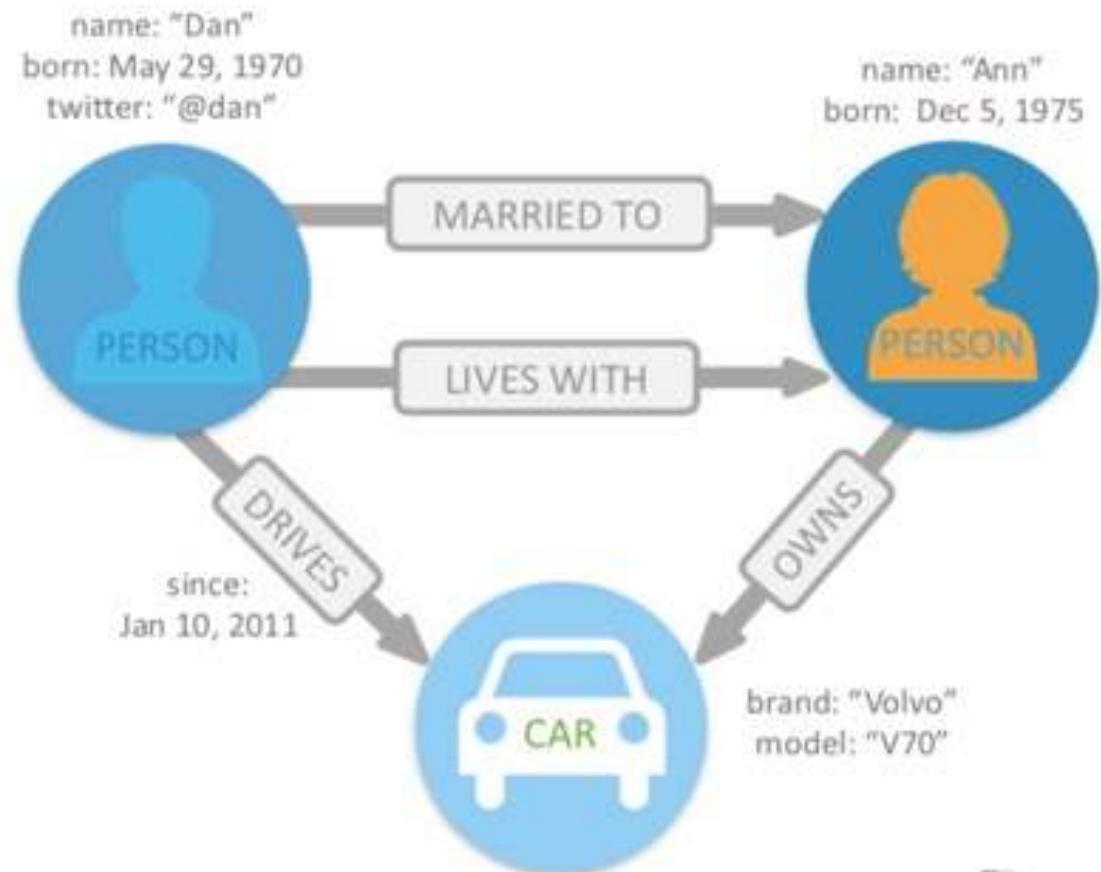
Schemaless Model: Flexibility

- Flexible property graph schema
- Neither node labels nor relationship types prescribe properties
 - freely add new edges
 - freely add properties to nodes and edges
- Schema can adapt over time
 - easy to add new nodes and relationships when the business needs change
 - changing existing nodes and their relationships is costly! (similar to data migration)

Queries as Traversals

Queries = Traversals

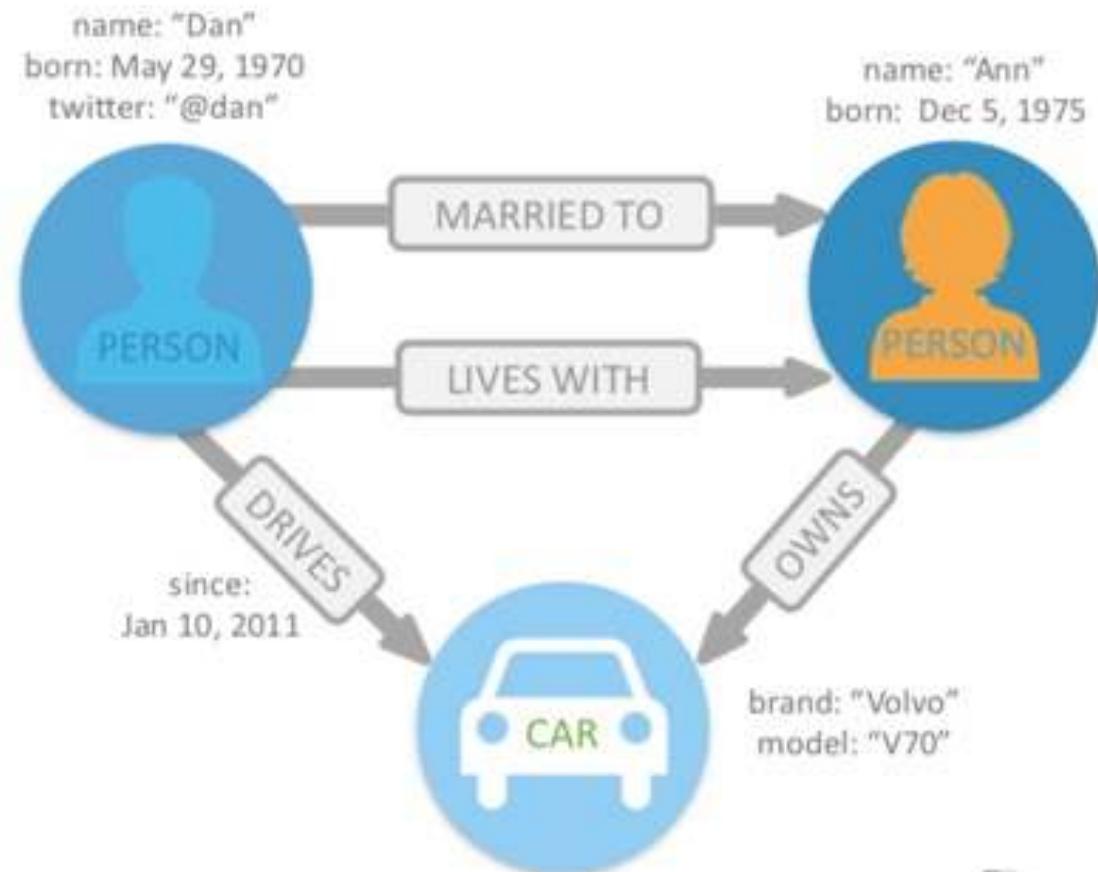
get the spouse of the owner of a Volvo



Queries = Traversals

get the spouse of the owner of a Volvo

get the owner of a car a person drives

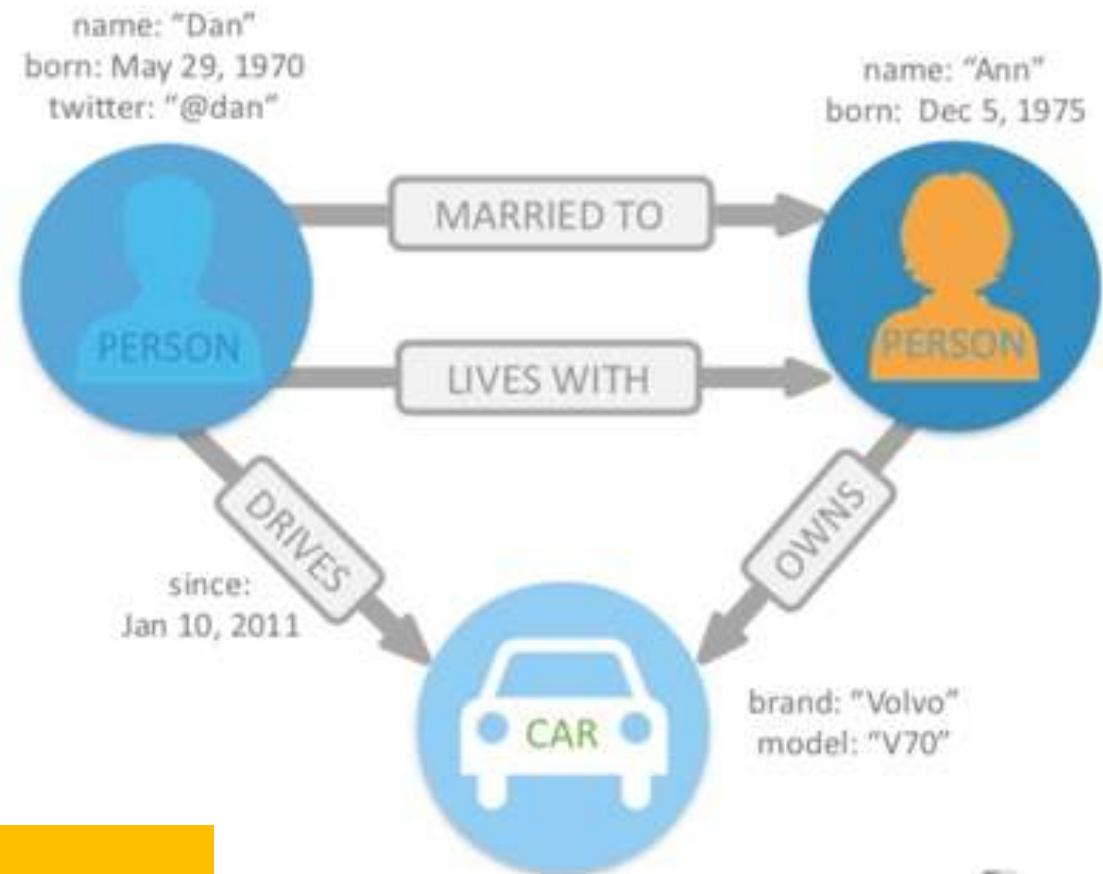


Queries = Traversals

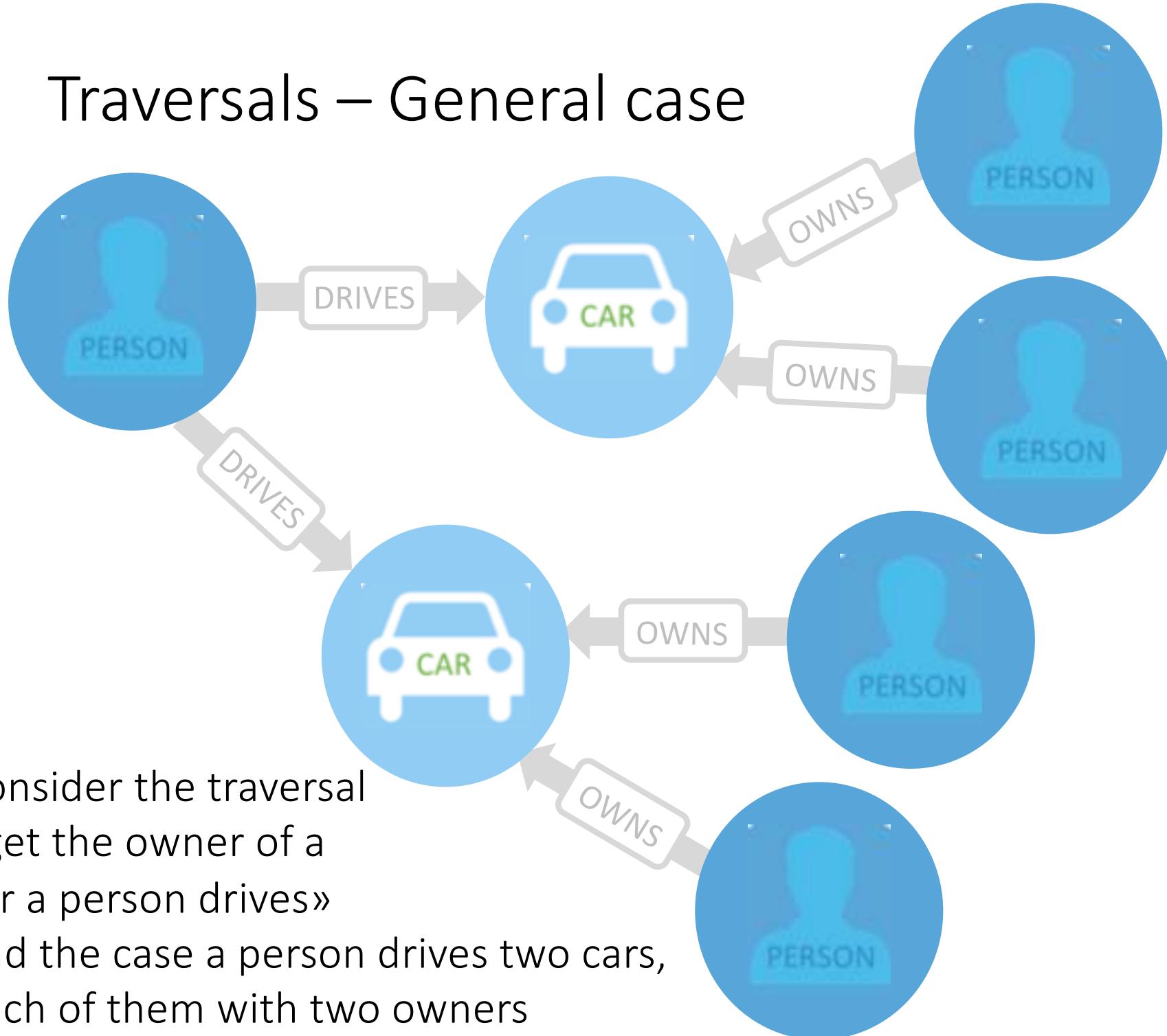
get the spouse of the owner of a Volvo

get the owner of a car a person drives

get the spouse of the owner of a car a person drives

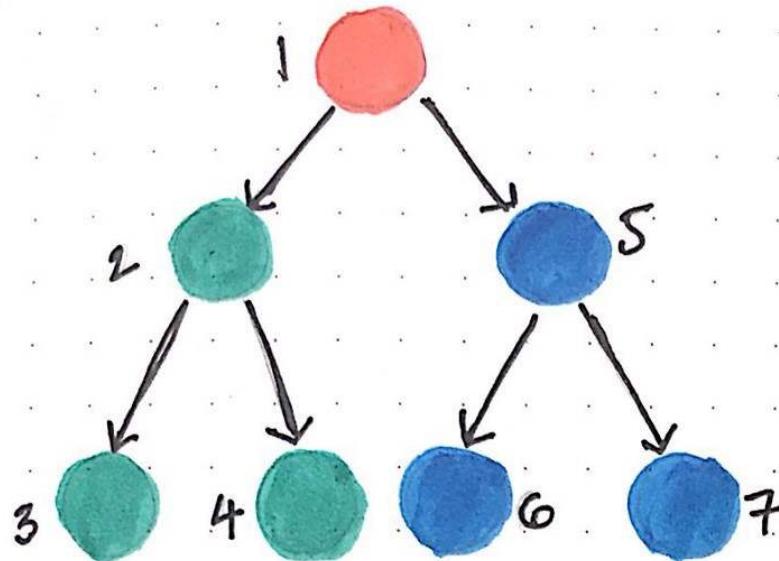


Traversals – General case



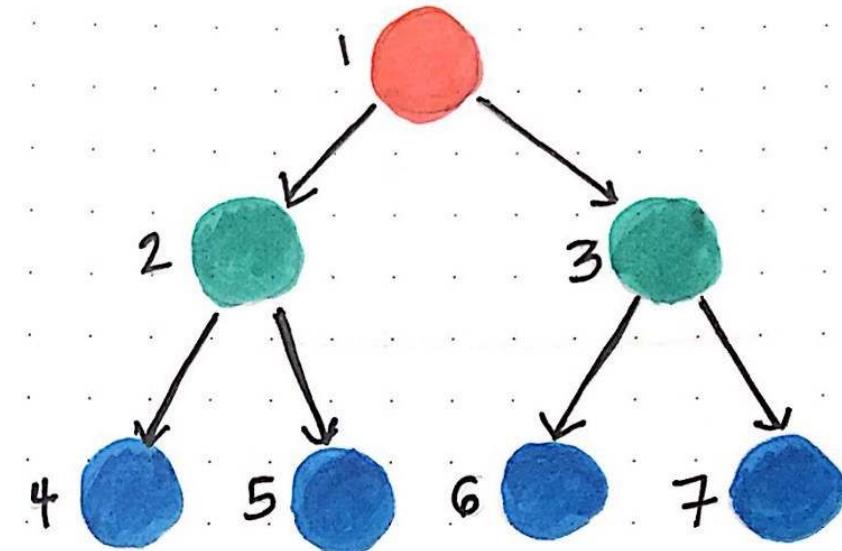
Consider the traversal
«get the owner of a
car a person drives»
and the case a person drives two cars,
each of them with two owners

Graph Traversals



Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).



Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Constant Time Traversal

- Constant time traversals in big graphs for both depth and breadth due to efficient representation of nodes and relationships
- Enables scale-up to billions of nodes on moderate hardware

Graph traversal

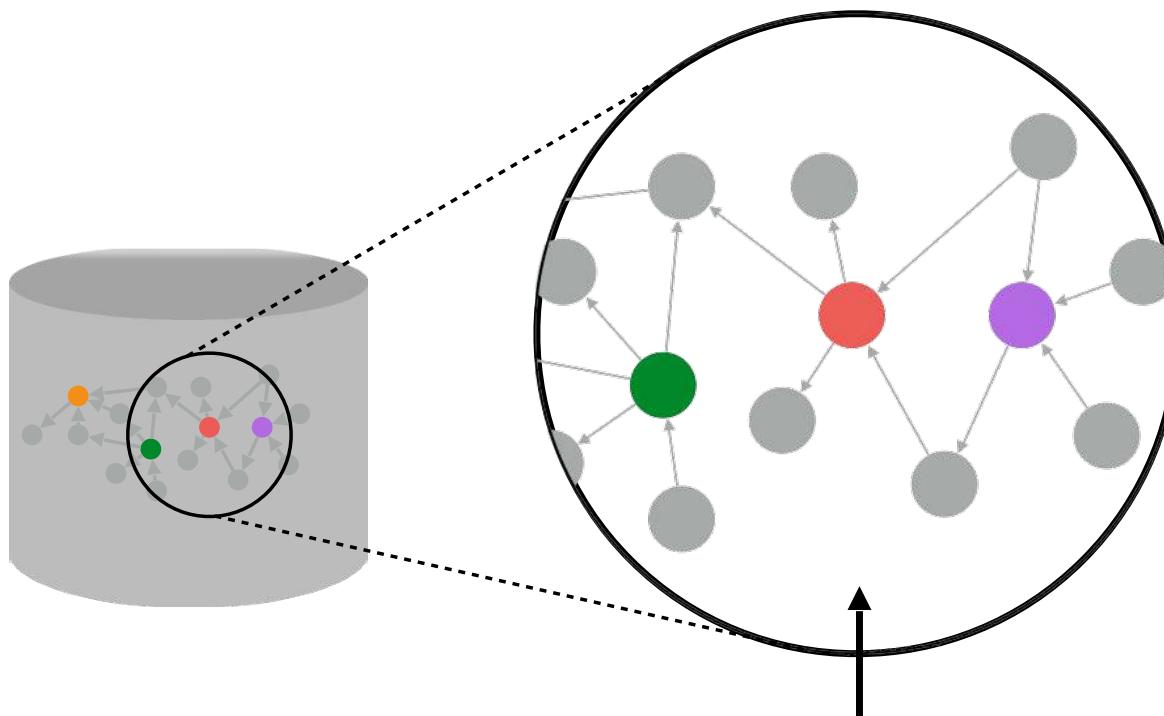
- Traversing relationships is very fast
- The relationship between nodes is not calculated at query time but it is actually persisted
- Traversing persisted relationships is faster than calculating them for every query

Index-free adjacency

We say that a (graph) database g satisfies the index-free adjacency if the existence of an edge between two nodes n_1 and n_2 in g can be tested on those nodes and does not require to access an external, global, index.

The cost of a basic traversal is independent of the size of the database

Index-free adjacency



At Write Time:
Data is *connected*
as it is stored

At Read Time:
Lightning-fast retrieval of data and relationships
via pointer chasing

Fast Graph Traversal

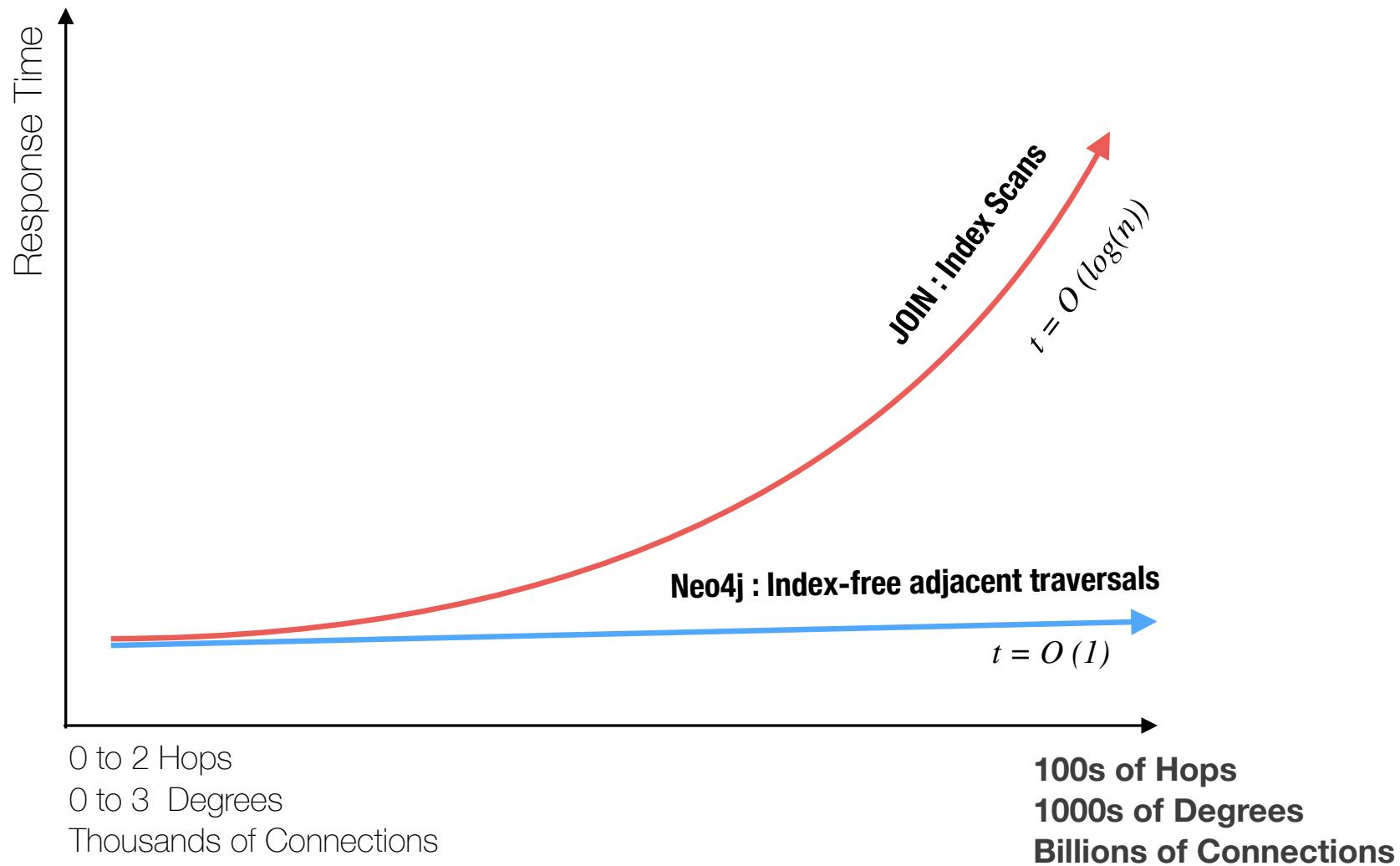
How Fast is Fast?

- Sample Social Graph with roughly 1,000 persons
- On average each person has 50 friends
- `pathExists(a,b)` limited to depth 4
- Caches warmed up to eliminate disk I/O

| DATABASE | # OF PERSONS | QUERY TIME |
|----------|--------------|------------|
| MySQL | 1,000 | 2,000 ms |
| Neo4j | 1,000 | 2 ms |
| Neo4j | 10,000,000 | 2 ms |

Index-Free Traversal

Connected-Data Query Performance



Graph vs relational data model

- Relational databases
 - implement relationships using foreign keys
 - joins require to navigate around and can get quite expensive
- Graph databases
 - make traversal along the relationships very cheap
 - performance is better for highly connected data
 - shift most of the work from query time to insert time
 - good when querying performance is more important than insert speed

Cypher Query Language

Declarative

- Most of the time, Neo4j knows better than you

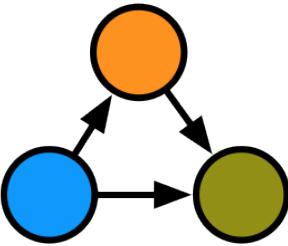
Imperative

- follow relationship
breadth-first vs depth-first
- explicit algorithm

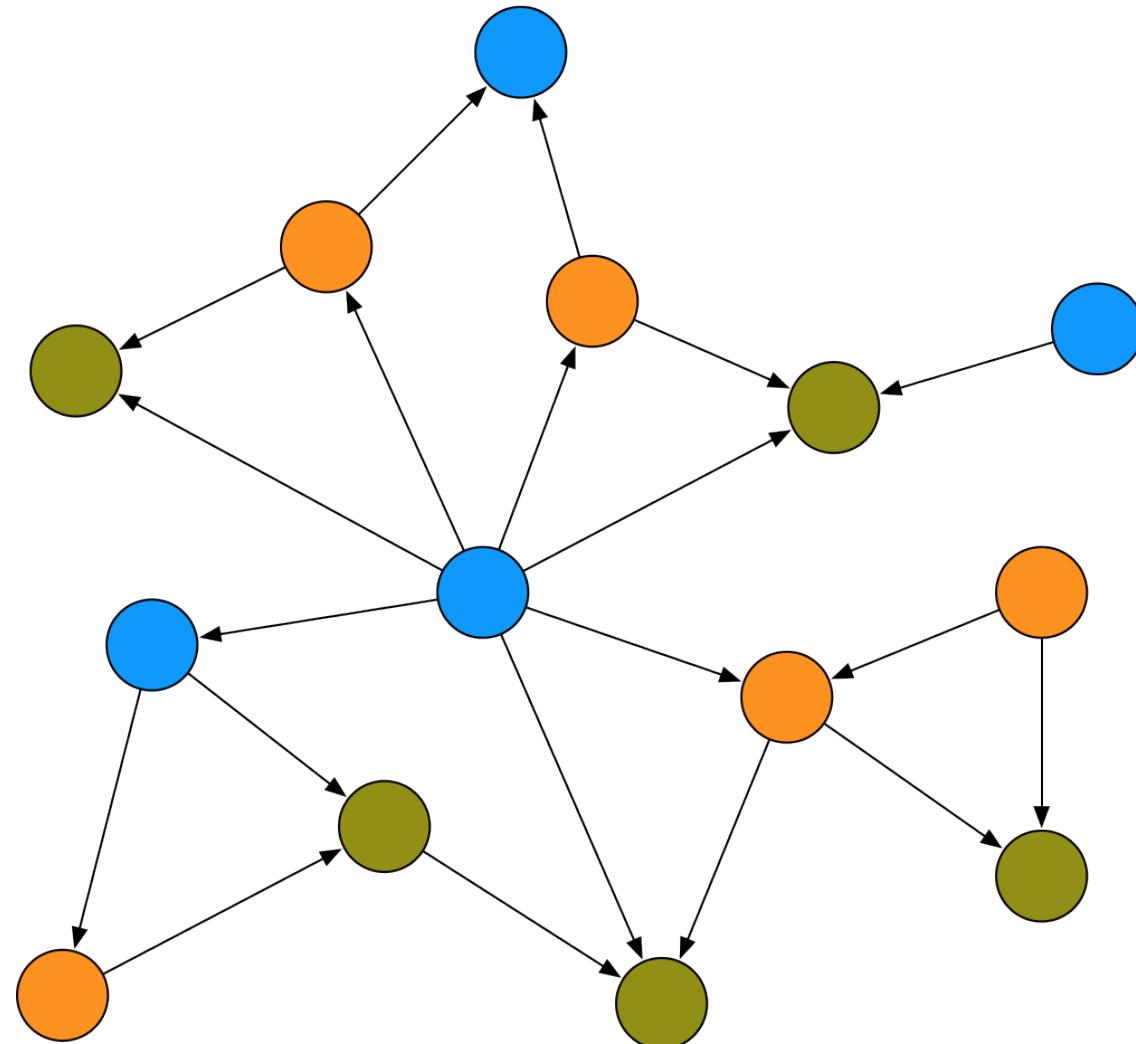
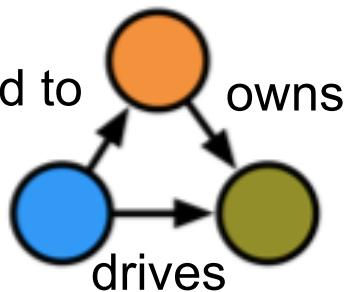
Declarative

- specify starting point
specify desired outcome
- algorithm adaptable based
on query

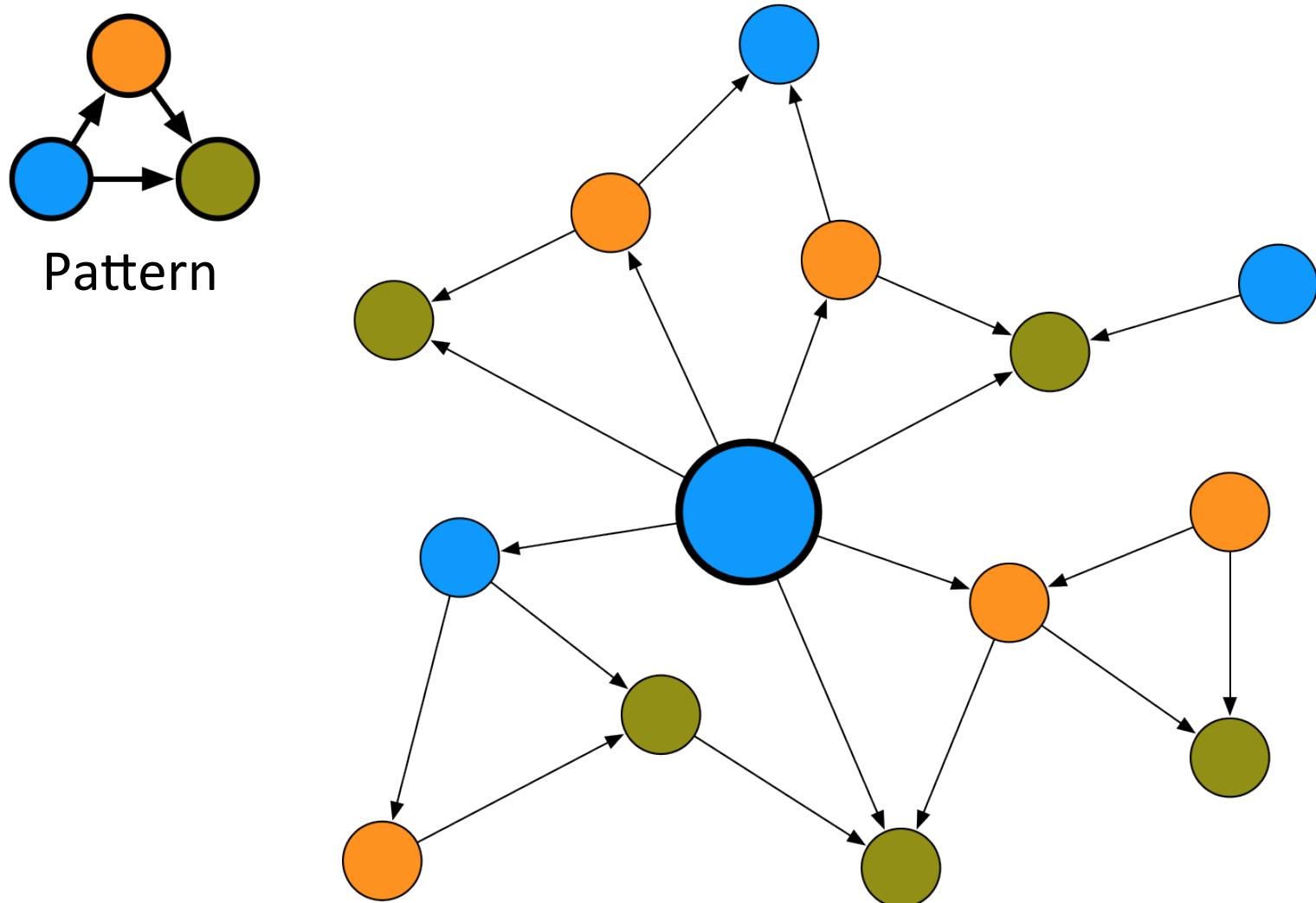
Pattern Matching



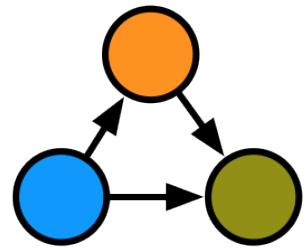
Pattern



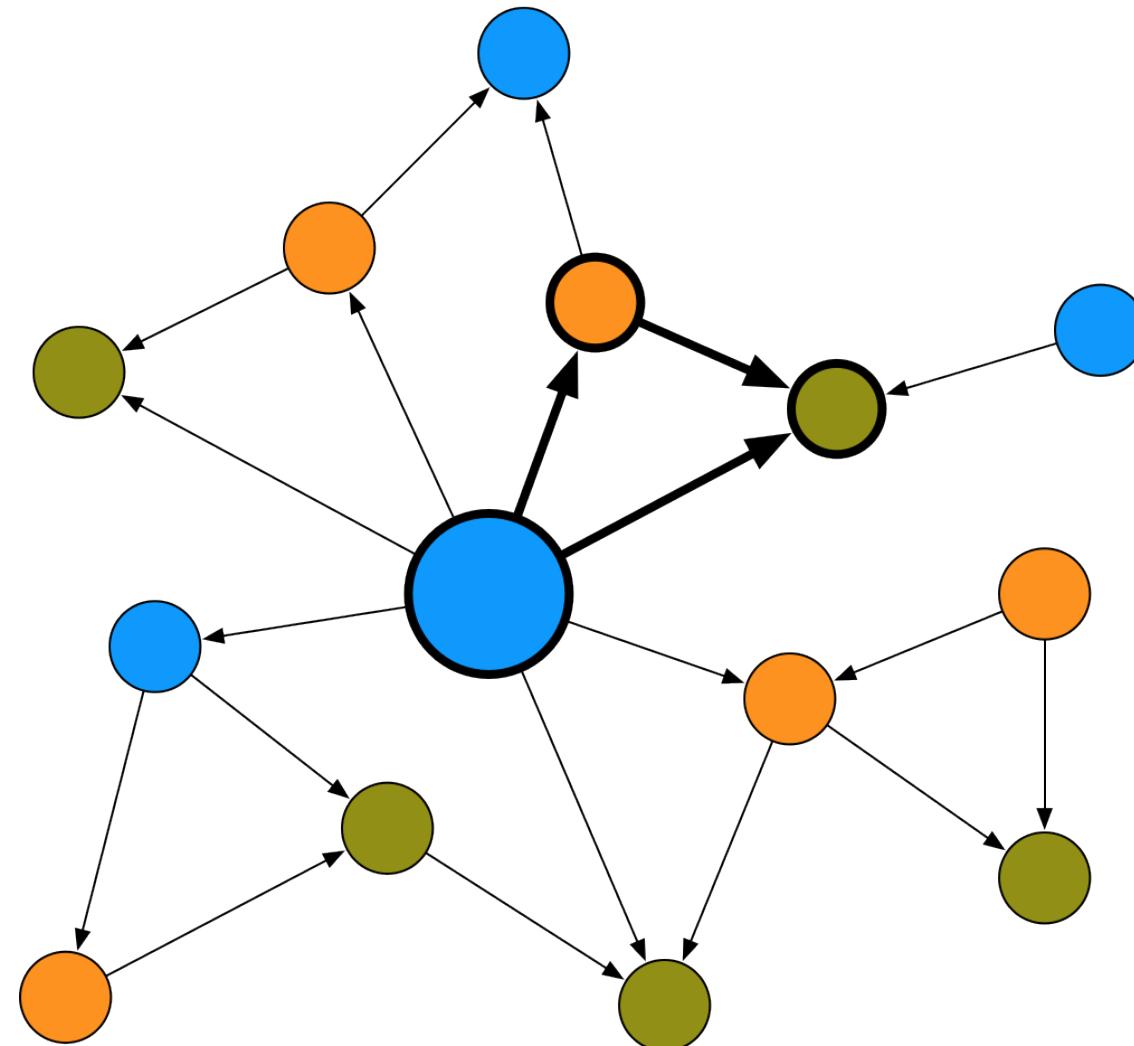
Pattern Matching – Start Node



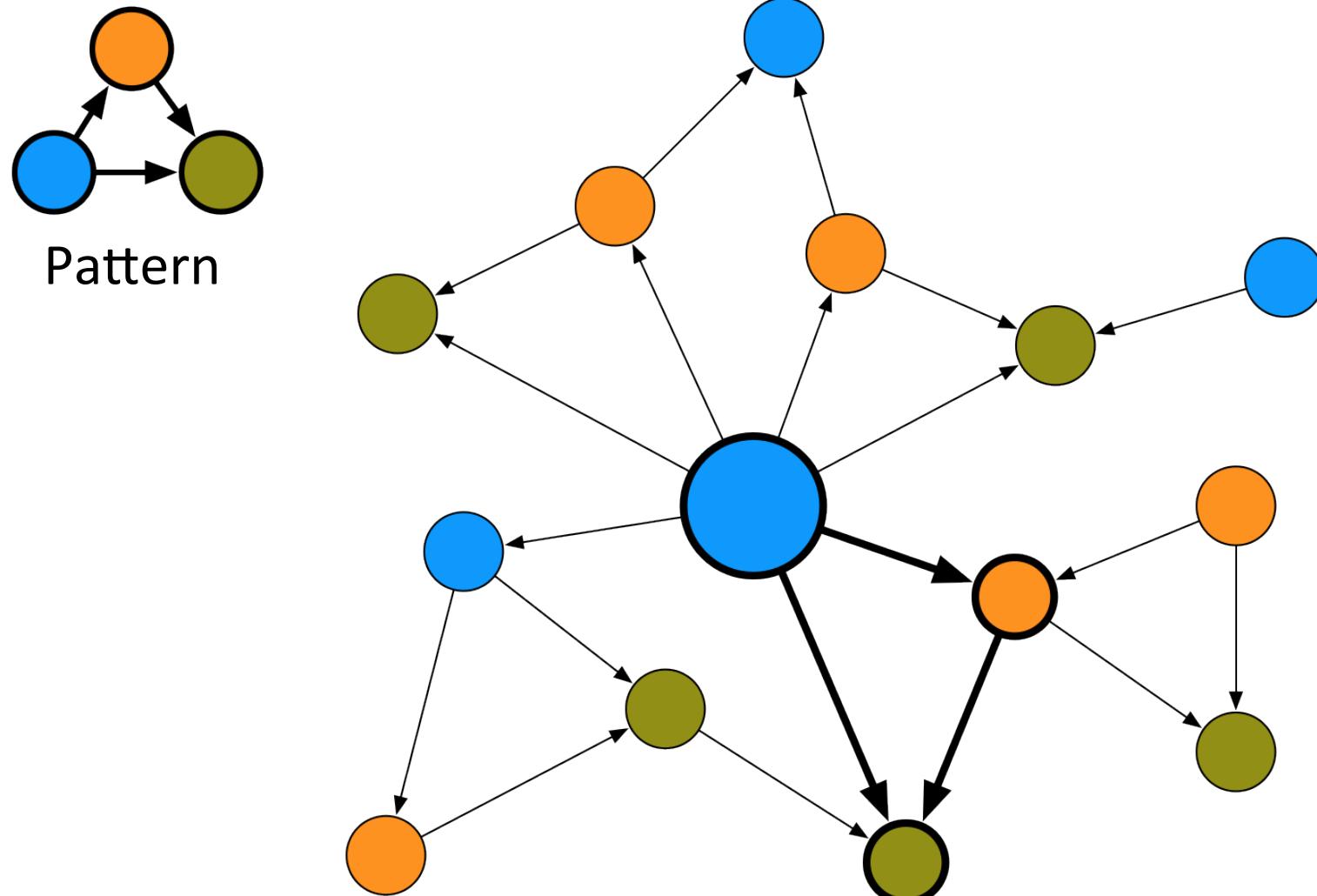
Pattern Matching - Match



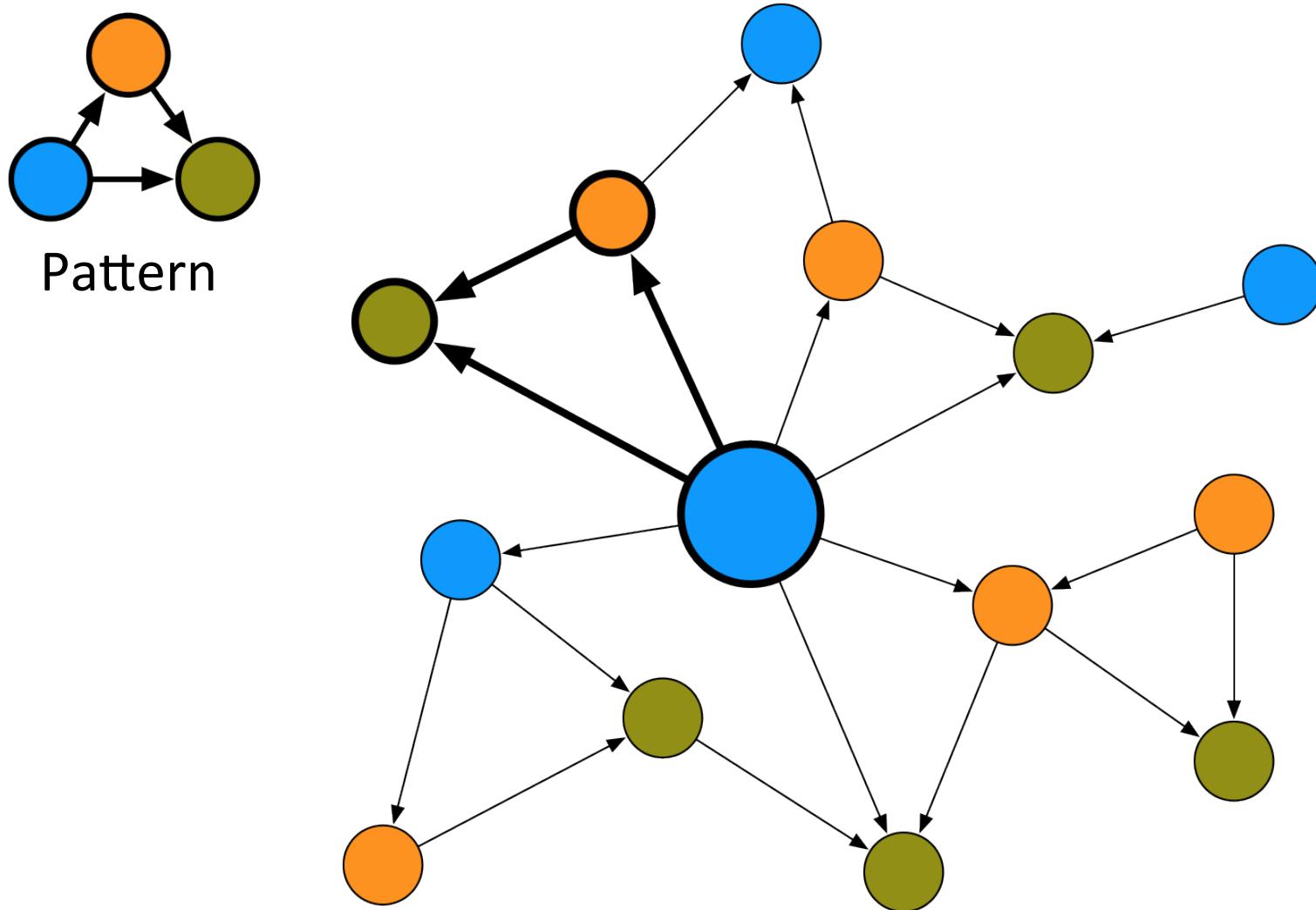
Pattern



Pattern Matching - Match



Pattern Matching - Match



ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location



ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location



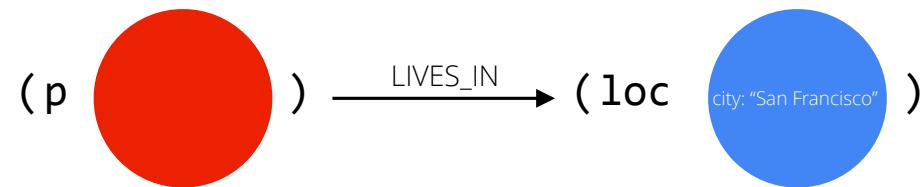
node ()

ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location



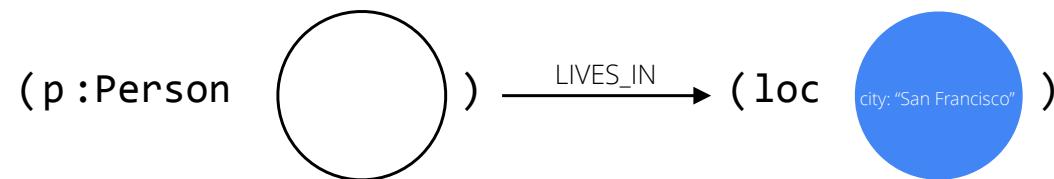
node variable (p)

ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location



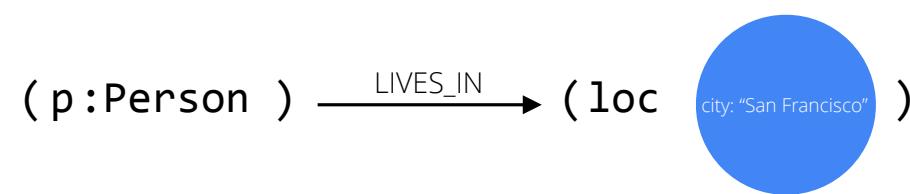
node label (:Person)

ASCII Art Patterns

Pattern: Persons who live in San Francisco

 Person

 Location



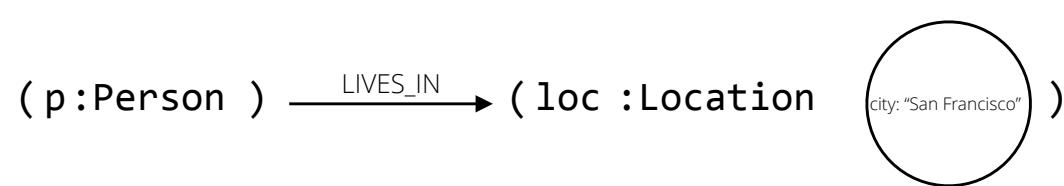
node label (:Person)

ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location



node label

(:Location)

ASCII Art Patterns

Pattern: Persons who live in San Francisco

- Person
- Location

(p :Person) —————→ (loc :Location {city: “San Francisco”})

property (city: ' 'San Francisco' ')

ASCII Art Patterns

Pattern: Persons who live in San Francisco

- Person
- Location

(p:Person) $\overset{\text{LIVES_IN}}{\dashrightarrow}$ (loc :Location {city: "San Francisco"})

edge \dashrightarrow

ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location

```
( p:Person ) - [:LIVES_IN] -> ( loc :Location {city: "San Francisco"} )
```

edge type - [:LIVES_IN] ->

ASCII Art Patterns

Pattern: Persons who live in San Francisco

● Person

● Location

```
MATCH (p:Person) - [:LIVES_IN] -> (loc:Location {city: "San Francisco"} )  
RETURN p
```

```
MATCH (node1:Label1 {property:value}) -->(node2:Label2)  
RETURN node1.property
```

```
MATCH (n1:Label1)-[rel:TYPE {property:value} ]->(n2:Label2)  
RETURN rel.property, type(rel)
```

Structural Patterns

“ASCII art”

variable
:node label
property values {k:v}

() - - ()

() <- - ()

() - -> ()

Conditions on the relationship may be specified inside []

variable
:edge type
property values

Structural Patterns - Nodes

- ()
- (matrix)
- (:Movie)
- (matrix:Movie)
- (matrix {title: "The Matrix"})
- (:Movie {title: "The Matrix", released: 1999})
- (matrix:Movie {released: 1999})
- ...

Structural Patterns - Relationships

More “ASCII art” (combines with <, >, :)

variable length path

() – [*] – ()

constrained length path

() – [*min..max] – ()

Structural Patterns - Paths

- $(a)-[*]->(b)$ traverse any depth
- $(a)-[*\ depth]->(b)$ exactly \textit{depth} steps long
- $(a)-[*1..4]->(b)$ from one to four levels deep
- $(a)-[:KNOWS*3]->(b)$ relationships of type KNOWS at 3 levels distance
- $(a)-[:KNOWS*..5]->(b)$ relationships of type KNOWS at most 5 levels distance
- $(a)-[:KNOWS|:LIKES*2..]->(b)$ relationships of type KNOWS or LIKES from 2 levels distance

Structural Patterns

- friend-of-a-friend

(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)

- shortest path:

path = shortestPath((user)-[:KNOWS*..5]-(other))

- collaborative filtering

(user)-[:PURCHASED]->(product)<-[:PURCHASED](otherUser)

NOTE we need to impose that otherUser <> user !

- tree navigation

(root)<-[:PARENT*]-(leaf:Category)-[:ITEM]->(data:Product)

Structural Patterns

- collaborative filtering

(user)-[:PURCHASED]->(product)<- [:PURCHASED] (otherUser)

otheruser <> user

:Product

:Customer

What if we are looking for a product bought by (at least) 3 users?

We may have multiple paths in the same MATCH, comma separated, sharing variables

Structural Patterns

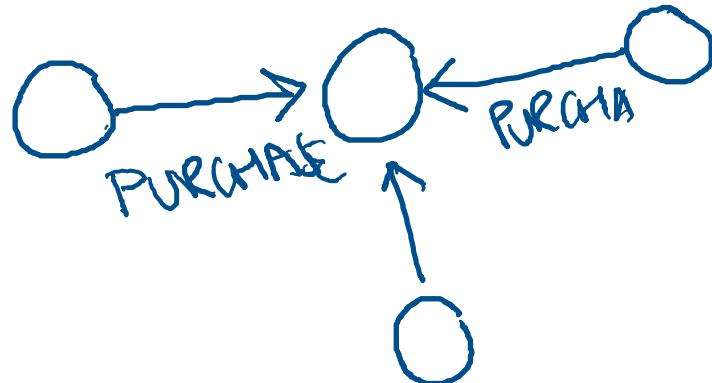
- collaborative filtering

(user)-[:PURCHASED]->(product)<- [:PURCHASED] (otherUser)
otheruser <> user

:Product

:Customer

What if we are looking for a product bought by (at least) 3 users?



We may have multiple paths in the same MATCH, comma separated, sharing variables

```
MATCH (user)-[ :PURCHASED ]->(product)<-[ :PURCHASED ](otherUser),  
(product)<- [ :PURCHASED ]-(thirdUser)
```

otheruser <> user, thirduser <> user, thirduser <> otheruser

Structural Patterns

- collaborative filtering

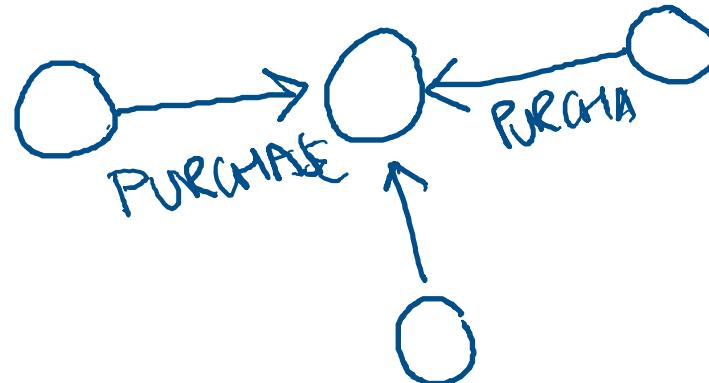
(user)-[:PURCHASED]->(product)<- [:PURCHASED] (otherUser)

otheruser <> user

:Product

:Customer

What if we are looking for a product bought by (at least) 3 users?



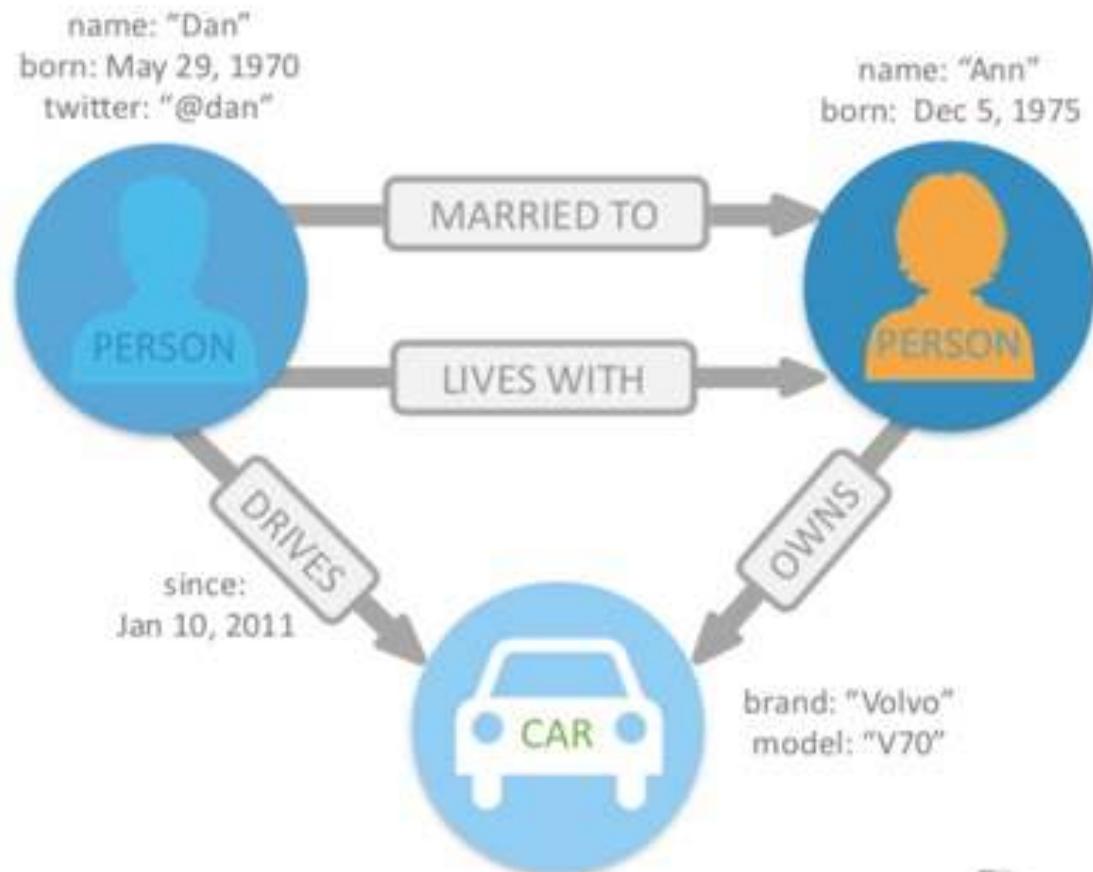
We may have multiple paths in the same MATCH, comma separated, sharing variables

```
MATCH (user)-[ :PURCHASED ]->(product),  
(product) <- [ :PURCHASED ] - (otherUser),  
(product) <- [ :PURCHASED ] - (thirdUser)
```

otheruser <> user, thirduser <> user, thirduser <> otheruser

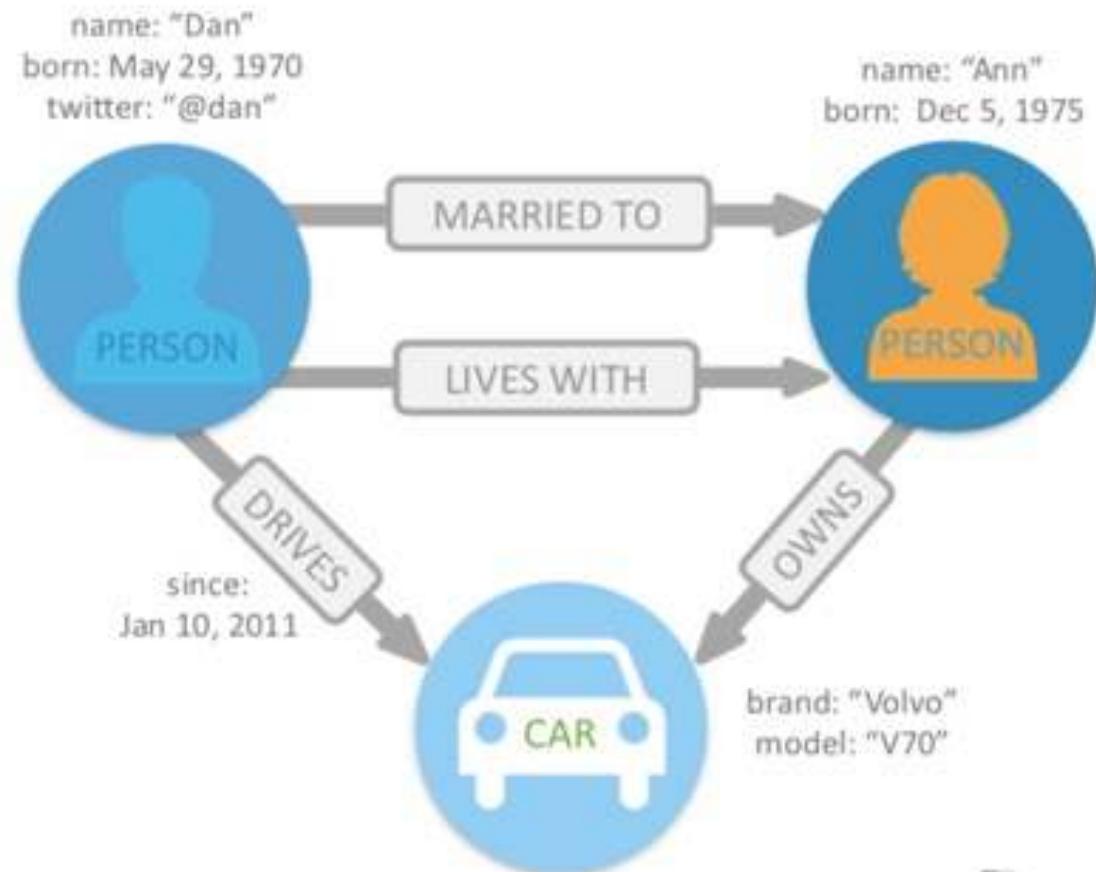
Queries = Traversals

get the spouse of the owner of a Volvo



Queries = Traversals

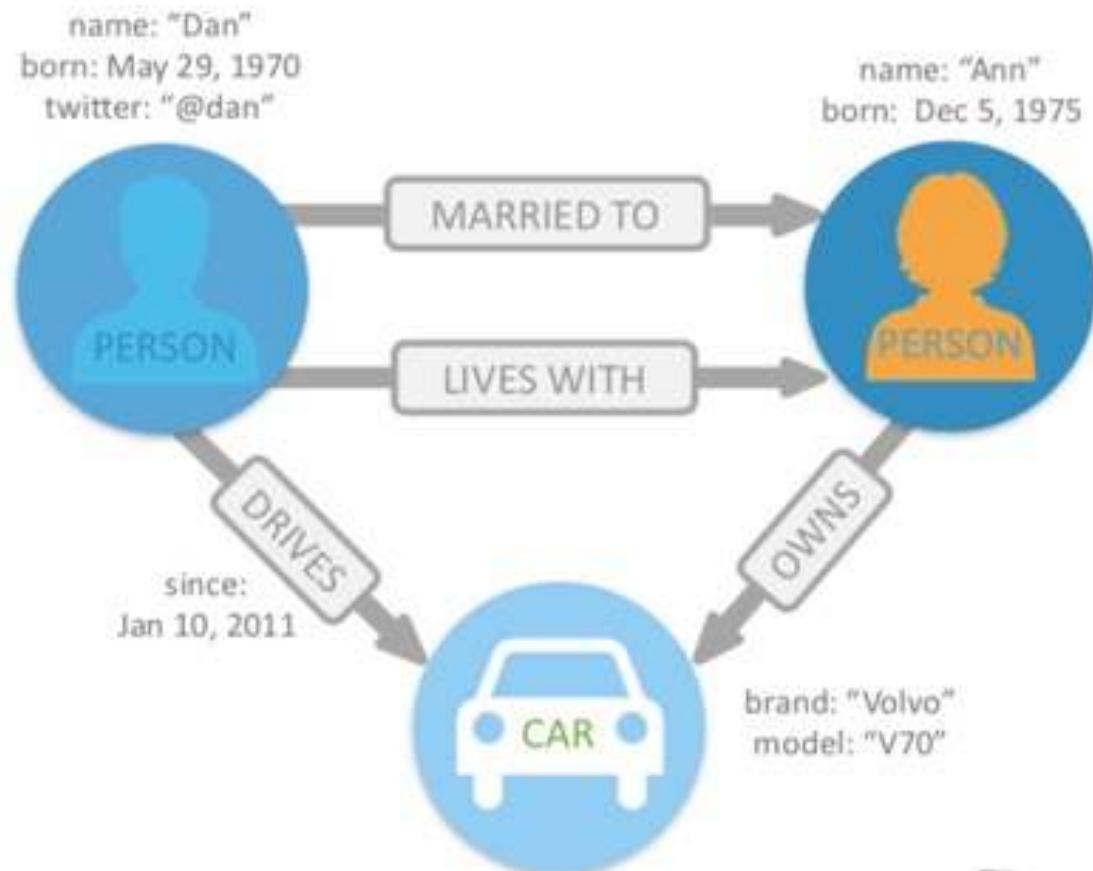
get the spouse of the owner of a Volvo



```
MATCH (:Car {brand:'Volvo'})<-[ :OWNS ]-  
(:Person)-[ :MARRIED_TO ]-(p:Person)  
RETURN p
```

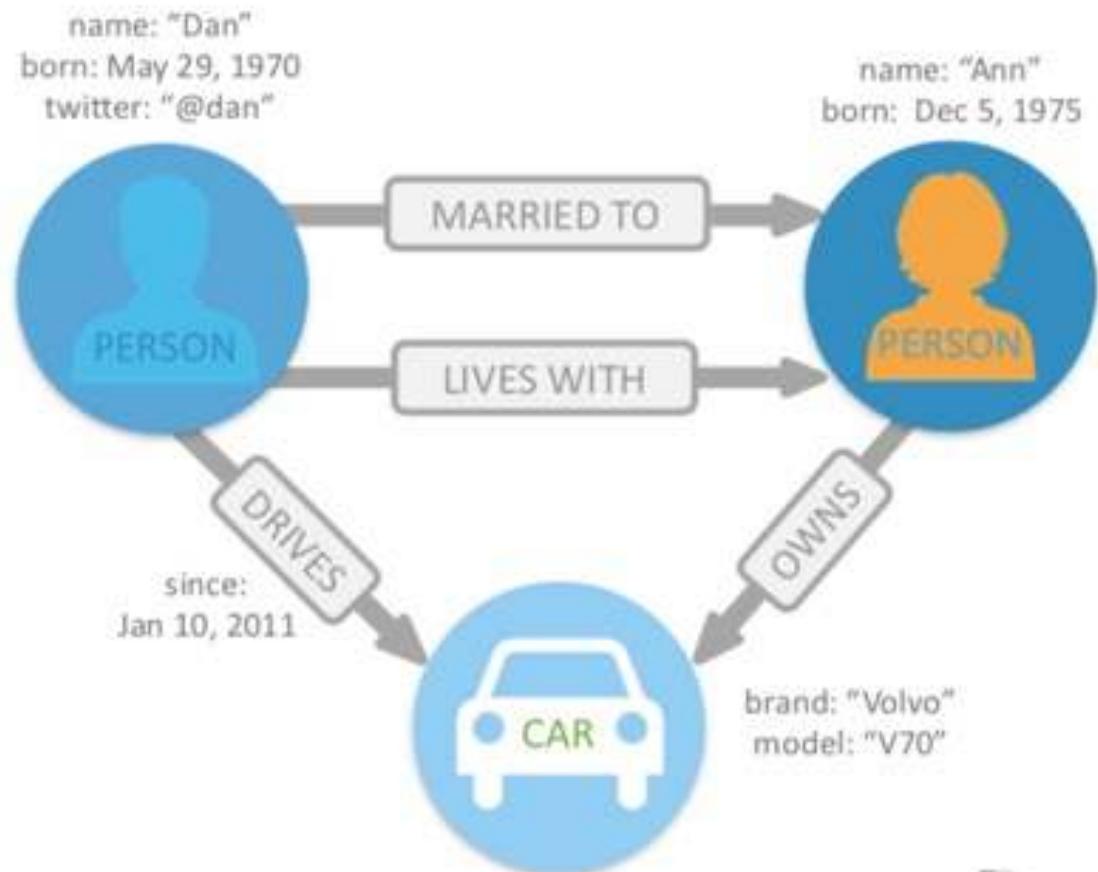
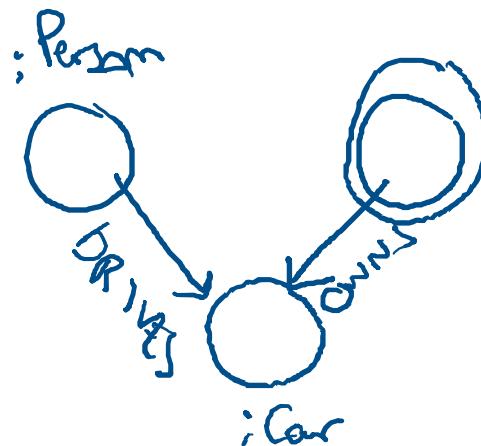
Queries = Traversals

get the owner of a car a person drives



Queries = Traversals

get the owner of a car a person drives

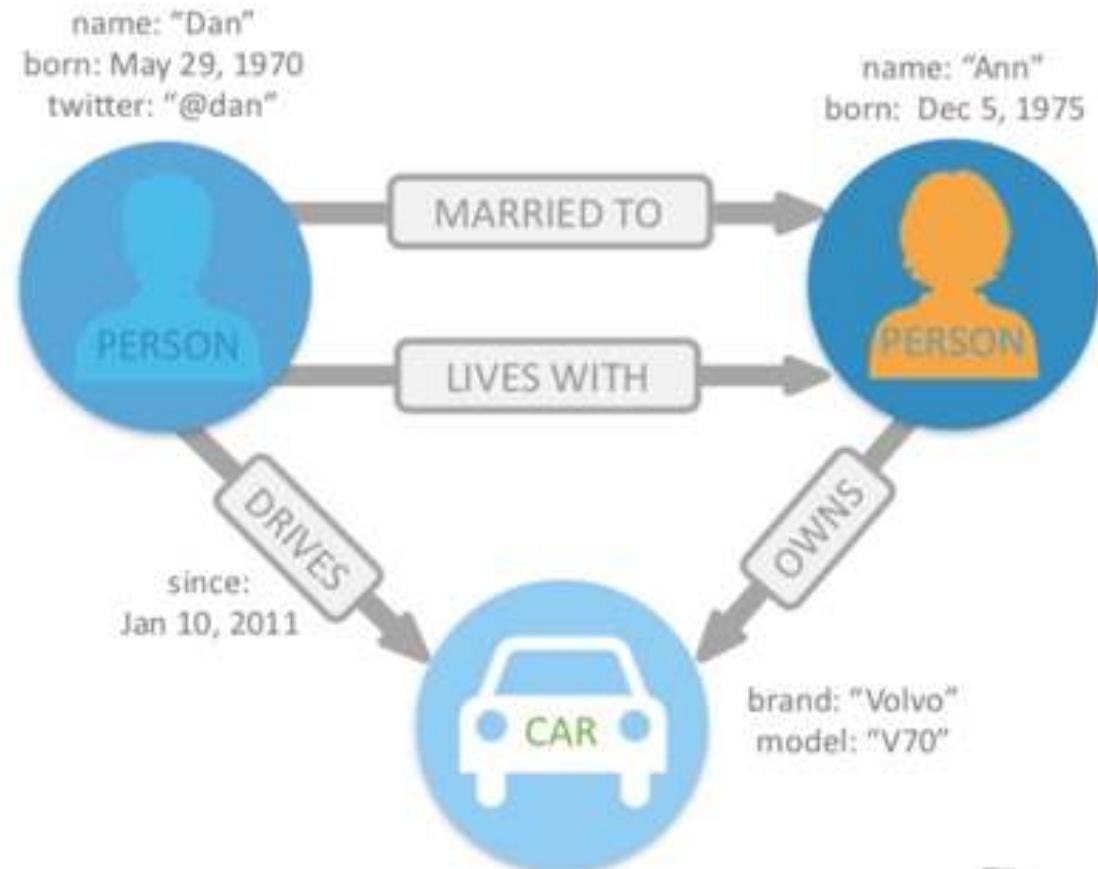


```
MATCH (:Person)-[:DRIVES]->(:Car)<-[:OWNS]-(p:Person)  
RETURN p
```

```
MATCH (c:Car)<-[:DRIVES]-(:Person), (p:Person)-[:OWNS]->(c)  
MATCH (p:Person)-[:OWNS]->(:Car)<-[:DRIVES]-(:Person)
```

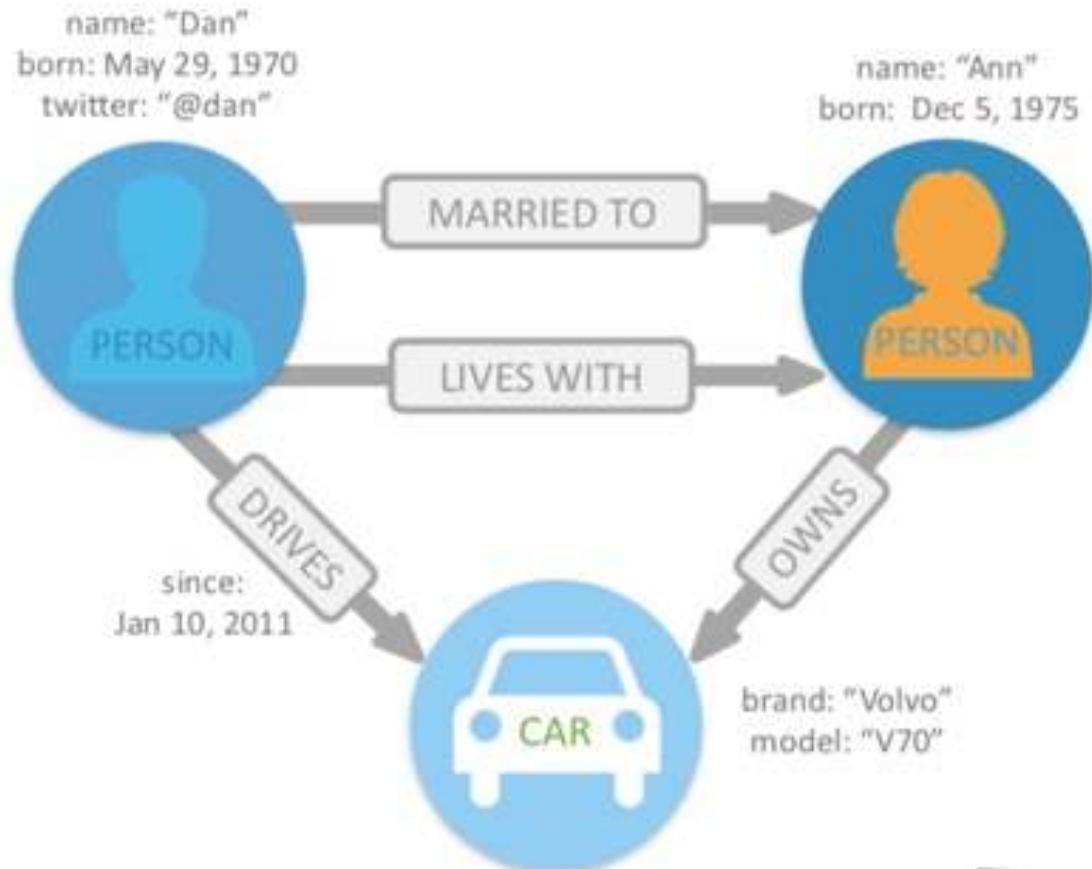
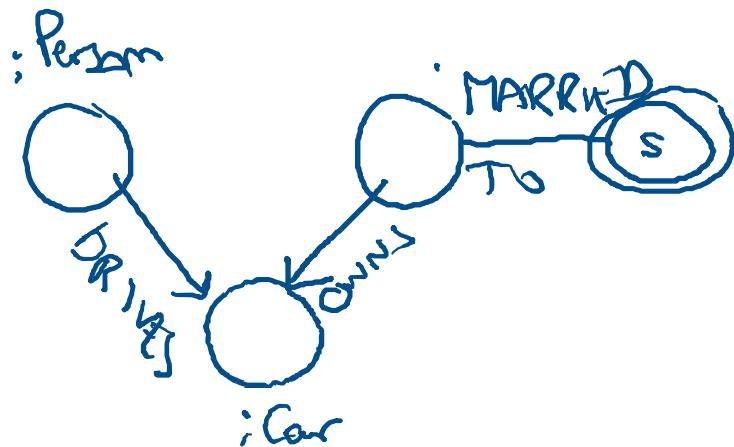
Queries = Traversals

get the spouse of the owner of a car a person drives



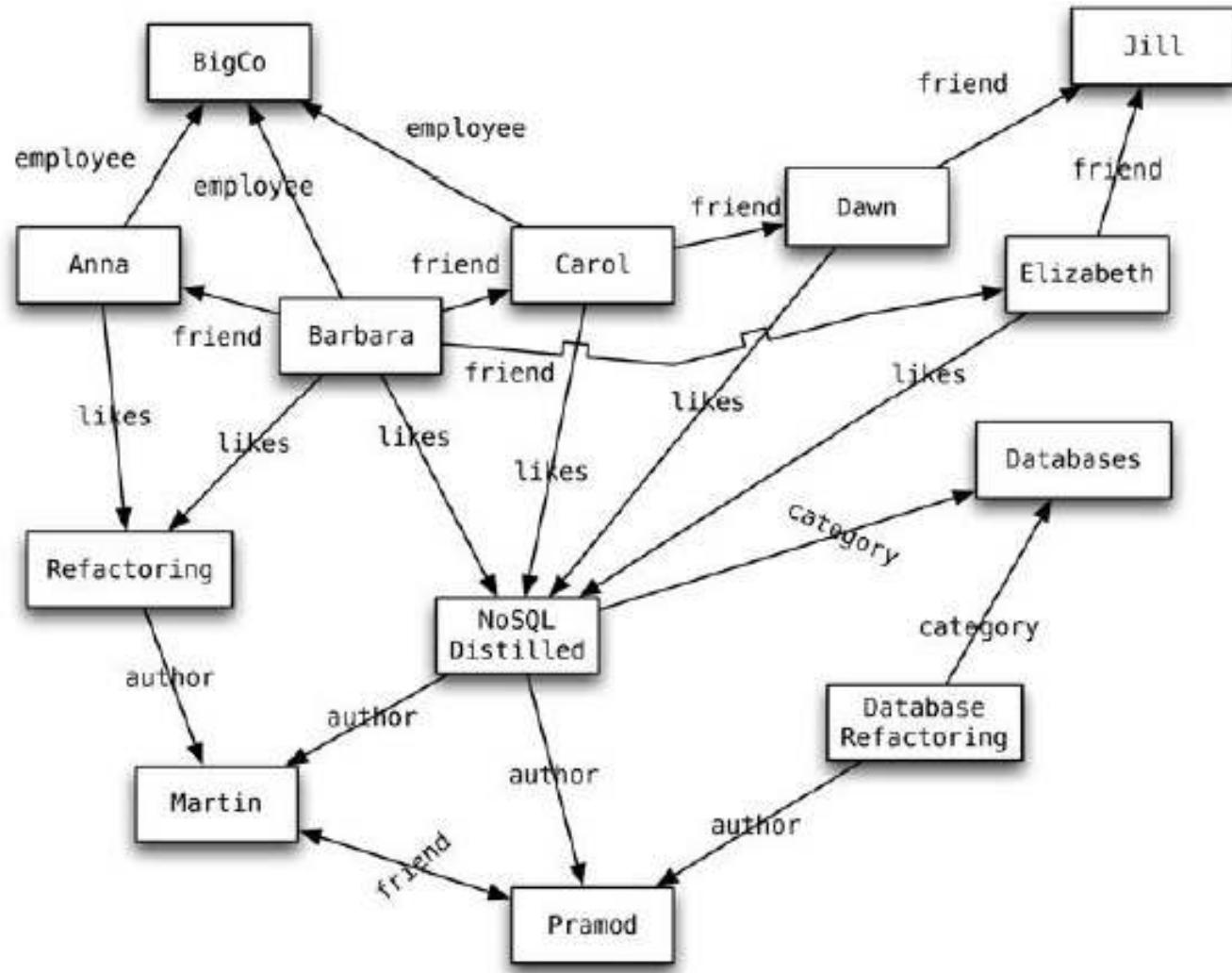
Queries = Traversals

get the spouse of the owner of a car a person drives

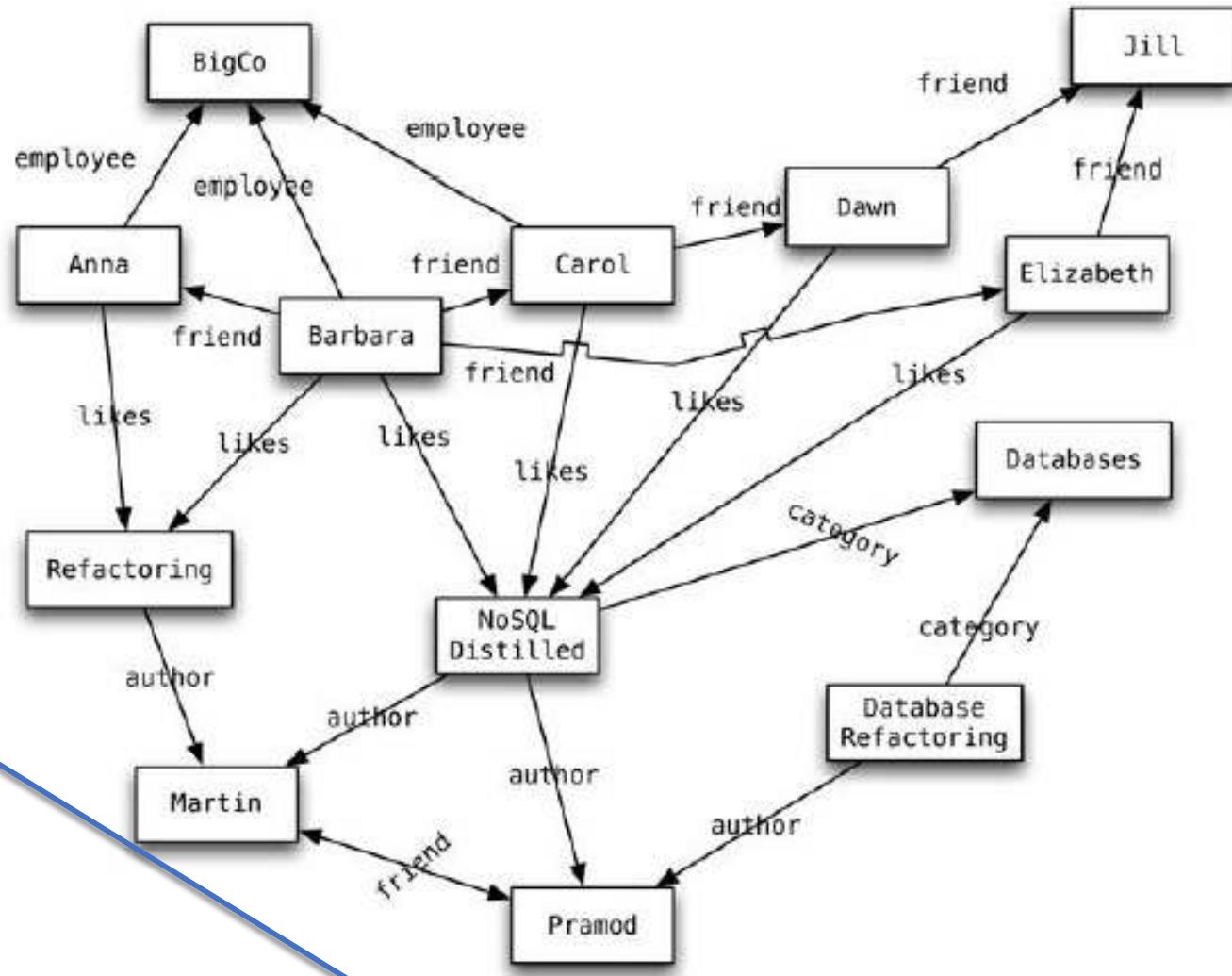


```
MATCH (:Person)-[:DRIVES]->(:Car)<-[:OWNS]-  
(:Person)-[:MARRIED_TO]-(s:Person)  
RETURN s
```

“find the friends of Barbara that like a node of database category.”

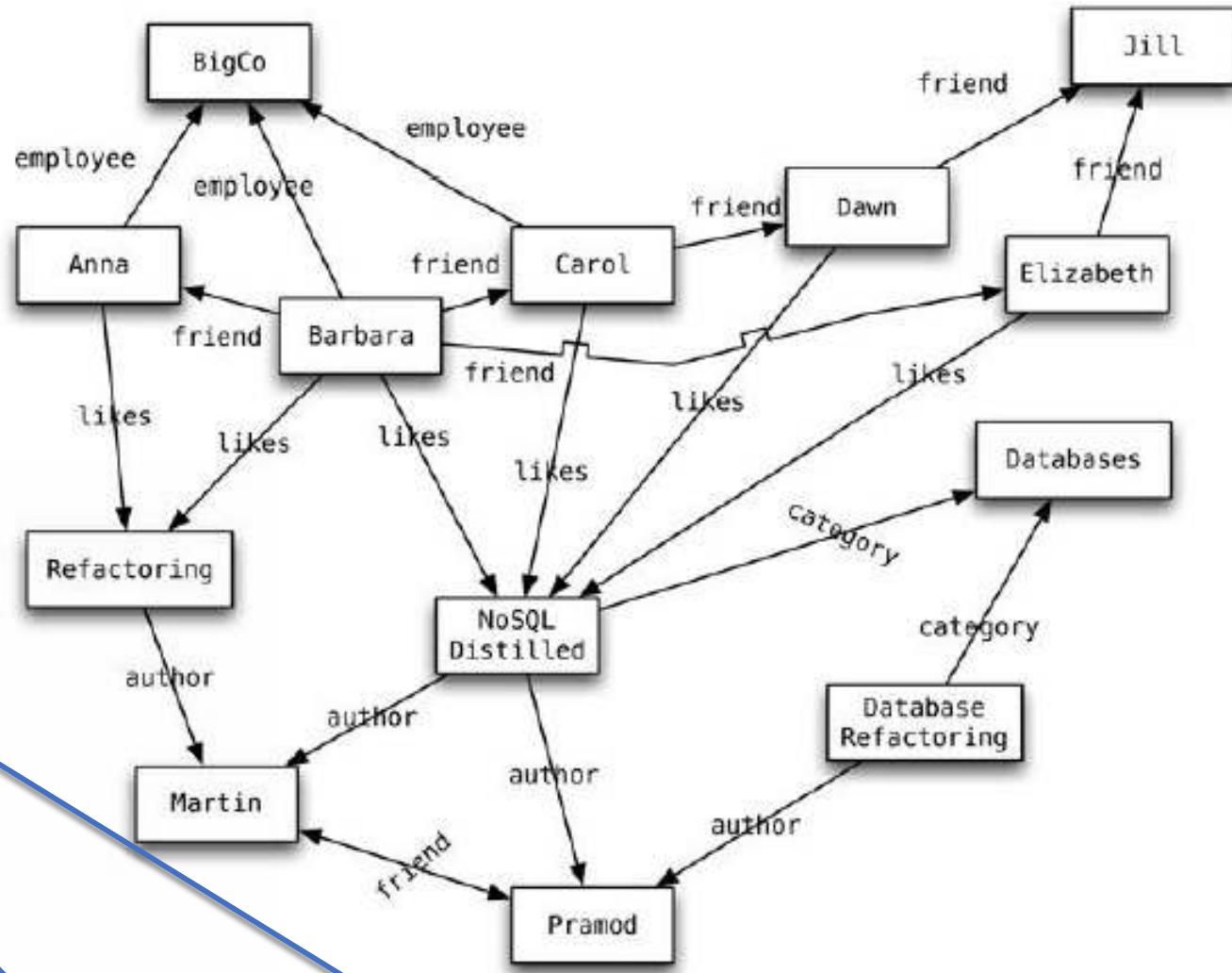


“find the friends of Barbara that like a node of database category.”



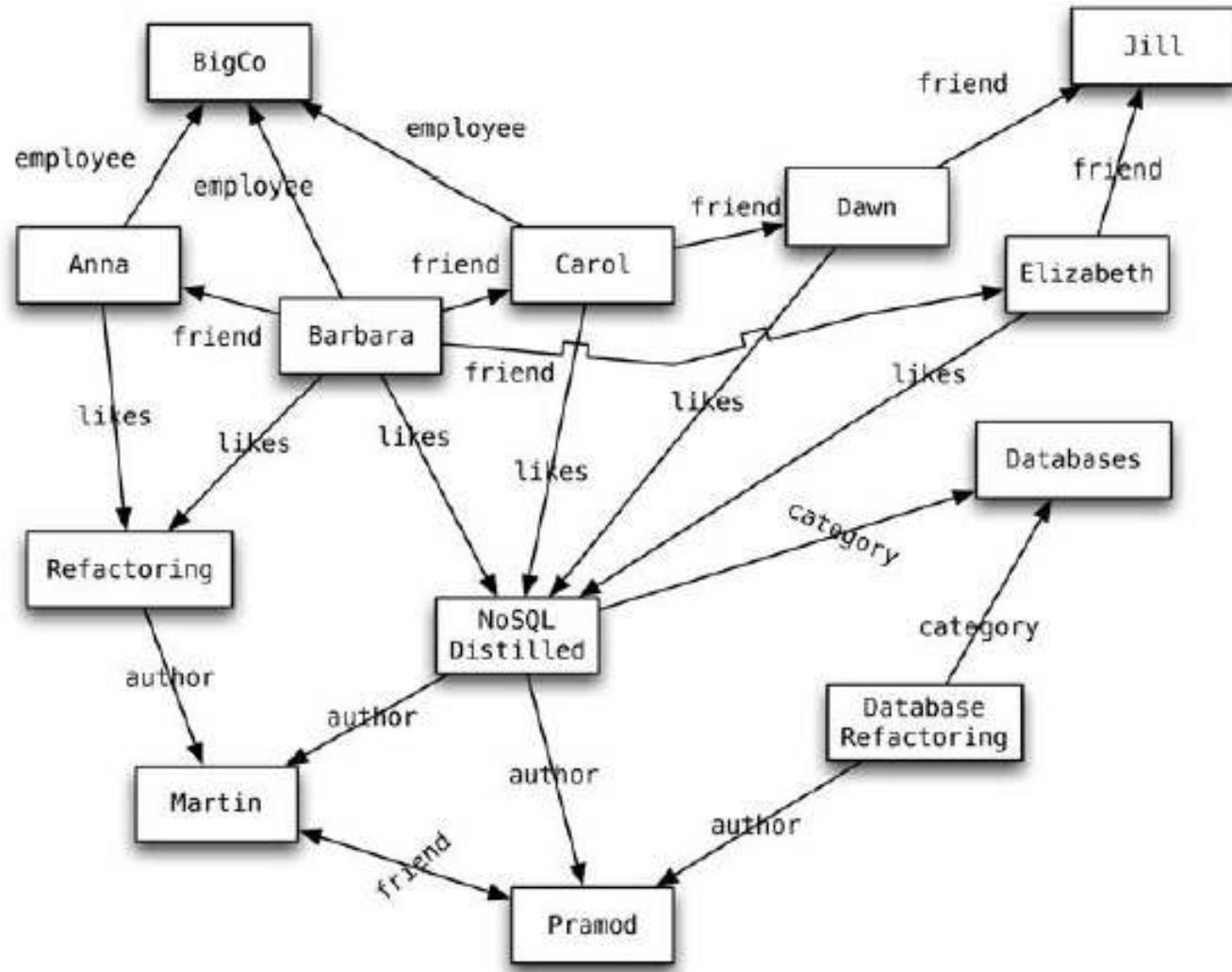
```
MATCH (:Barbara)-[:friend]-(node)-[:likes]->()-[:category]->(:Databases)  
RETURN node ;
```

“find the friends of Barbara that like a node of database category.”



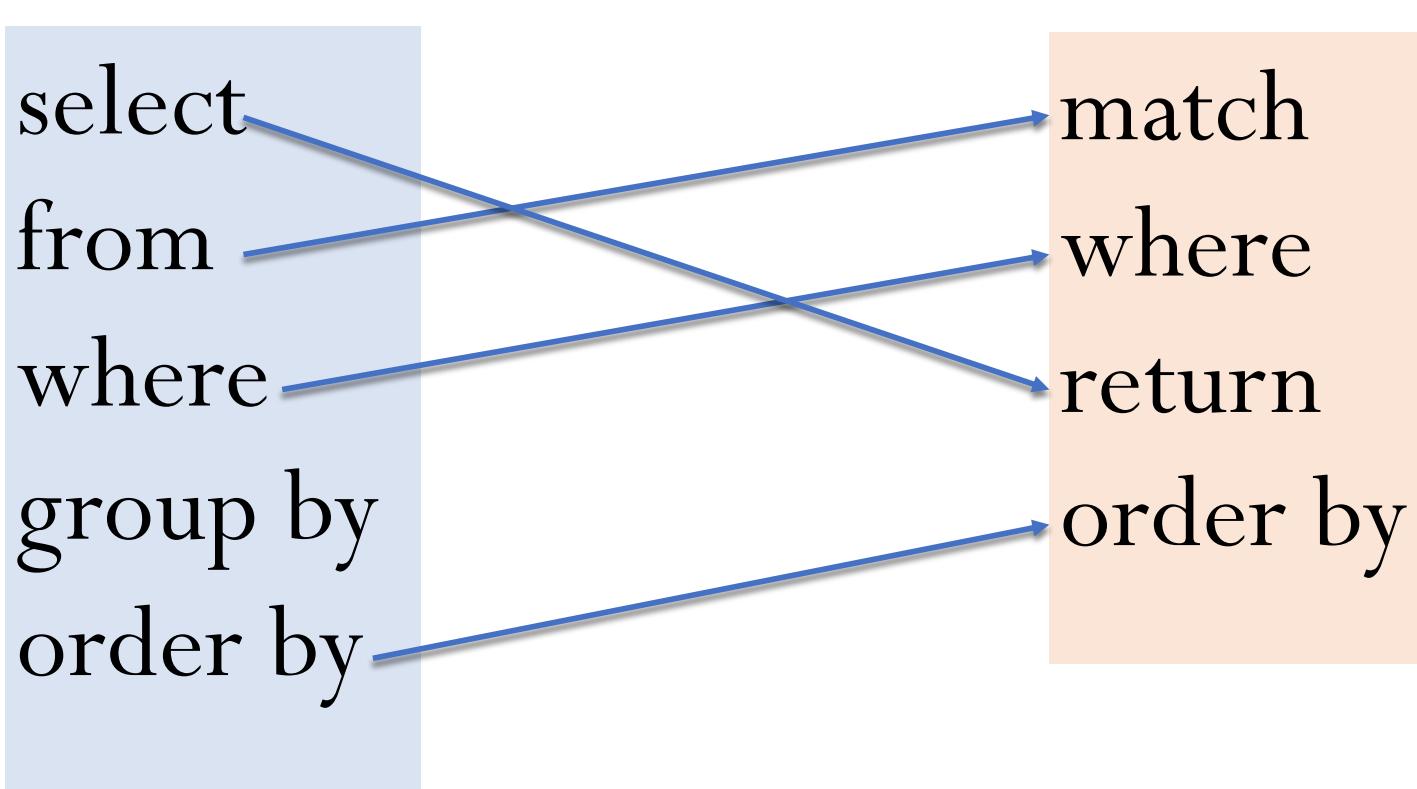
MATCH (:Barbara)-[:friend]-(node)-[:likes]->()-[:category]->(:Databases)
RETURN node ;

MATCH (:Databases)<-[:category]-()<-[:likes]-(node)-[:friend]-(:Barbara)
RETURN node ;



Cypher queries – Structure

Familiar for SQL Users



Where

```
MATCH (node:Label {property:value})  
MATCH (node:Label)  
WHERE property=value
```

- usual big 6 (=, <, >, <=,>=, <>)
- Usual Boolean (AND, OR, NOT)
- Property existence checking: exists
 - MATCH (n)
 - WHERE exists(n.marriedTo)
 - RETURN n.name, n.marriedTo
- Collection membership: IN
- ...

Returning nodes and values

```
MATCH (:Barbara)-[:friend]-(node)-  
[:likes]->()- [:category]-> (:Databases)  
RETURN node
```

```
MATCH (n)  
WHERE exists(n.marriedTo)  
RETURN n.name, n.marriedTo
```

Returning Paths

- `MATCH p=(:Barbara)-[:friend]-(node)-[:likes]->()-[:category]->(:Databases)`
`RETURN p`
all of the nodes and relationships for each path,
including all of their properties
- Just nodes in the path:
`RETURN nodes(p)`
- Just relationships in the path:
`RETURN rels(p)`

Cypher: Outputs (from yesterday)

A node

```
MATCH (p:Person {name:"Tom"}) RETURN p
```

A value

```
MATCH (p:Person {name:"Tom"}) RETURN p.age
```

A list of values

```
MATCH (p:Person) RETURN p.name LIMIT 5
```

An array

```
MATCH p=shortestPath((a)-[*]->(b)) WHERE a.name="Axel" AND b.name="Tom" RETURN p
```

A list of arrays

```
MATCH p=((a)-[*]->(b)) WHERE a.name="Axel" AND b.name="Frank" RETURN p
```

Return – aggregate functions

- MATCH (n:Person)

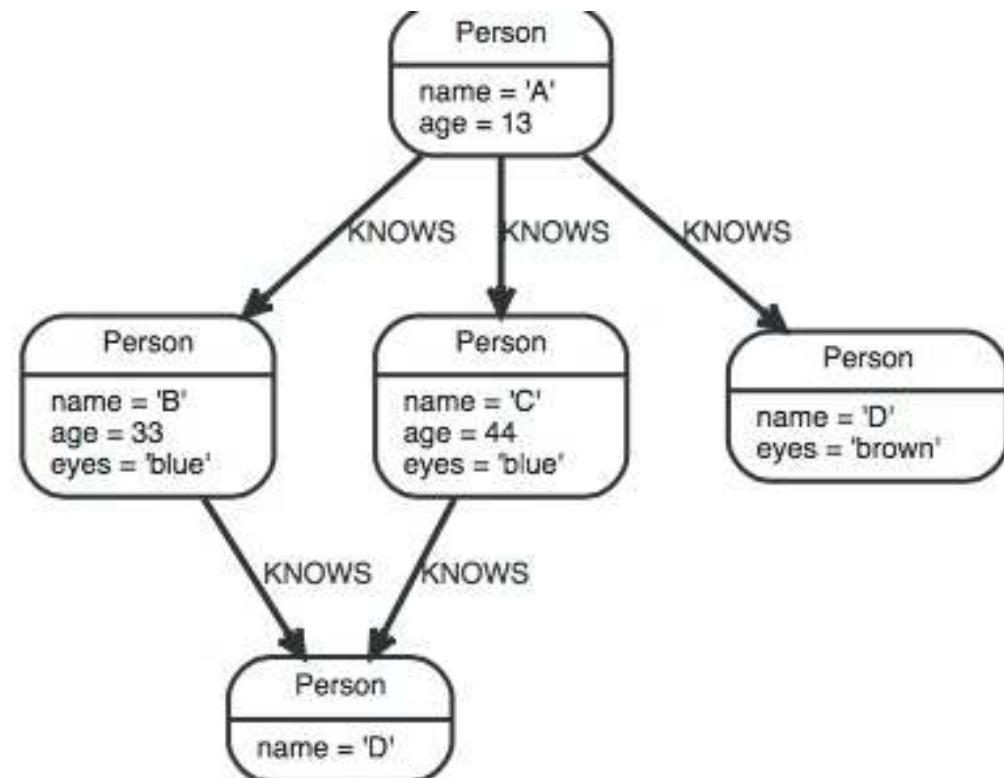
RETURN avg(n.age)

30

- MATCH (n:Person)

RETURN collect(n.age)

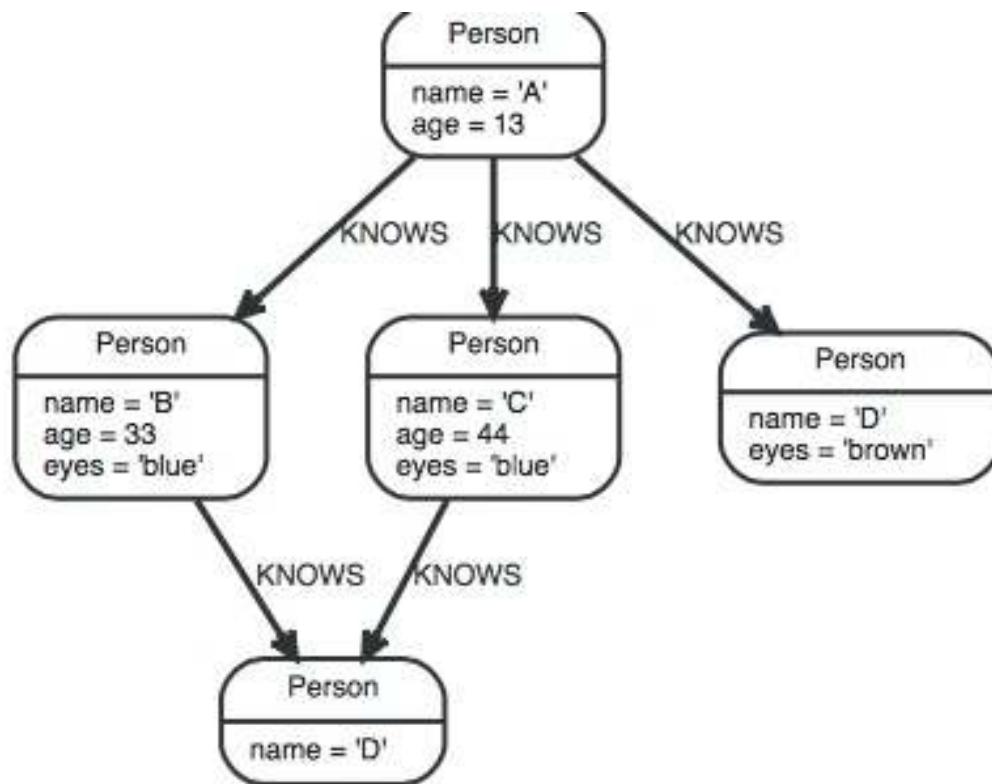
(13, 33, 44)



[avg\(\), collect\(\), count\(\), max\(\), min\(\), sum\(\), ...](#)

GROUP BY

- In Cypher, GROUP BY is done implicitly by all of the aggregate functions
- In a RETURN clause (as in a WITH, see next slides), any columns not part of an aggregate will be the GROUP BY key



- MATCH (n:Person)
RETURN COUNT(n), n.eyes
2,'blue' 1,'brown'
- MATCH (n:Person)
RETURN COUNT(n), n.eyes,
AVG(age)
2,'blue',38.5 1,'brown'

GROUP BY – how many friends?

```
MATCH (p:Person)-[:Knows]->(friend)  
WHERE p.age = 20  
RETURN p.name, count(friend)
```

What if we only want people with at least 10 friends?
(a sort of SQL HAVING)
We need a WITH clause ...

WITH – to divide a query into multiple parts

```
MATCH (p:Person)-[:Knows]->(friend)
```

```
WHERE p.age = 20
```

```
WITH p, count(friend) as friends
```

```
WHERE friends > 10
```

```
RETURN p.name, friends
```

Ex: Reformulate the 3 users queries

Cumulative

```
reduce(accumulator = initial,  
variable IN list | expression)
```

1. iterate through each element e in the given list
2. run the expression on e and
3. store the new partial result in the accumulator

```
MATCH p =(a:{name:'Alice'})-[ :KNOWS*..3 ]->  
      (c: {name:'Bob'})  
RETURN reduce(totalAge = 0, n IN nodes(p) |  
totalAge + n.age) AS reduction
```

A few more Cypher queries

Return the pairs of nodes that are related by two different relationships

```
MATCH(n1)-[r1]-(n2)-[r2]-(n1) WHERE type(r1)<>type(r2)  
RETURN n1, n2
```

Return the three nodes with the highest number of relationships.

```
MATCH (a)-[r]->()  
RETURN a, COUNT(r) ORDER BY COUNT(r) DESC LIMIT 3
```

Return the length of the shortest path between two nodes of your choice.

```
MATCH p = shortestPath((p1:Person{name:'Alice'})-[*]-  
                      (p2:Person{name:'Bob'}))  
RETURN length(p)
```

Cypher: Clauses

- **MATCH**: The graph **pattern** to match
- **WHERE**: **Filtering** criteria
- **RETURN**: What to return
- **CREATE**: Creates nodes and relationships
- **DELETE**: Remove nodes, relationships, properties
- **SET**: Set values to **properties**
- **WITH**: Divides a query into multiple parts

Cypher vs SQL

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
    SELECT manager.pid AS directReportees, 0 AS count
        FROM person_reportee manager
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
    SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
        FROM person_reportee manager
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
      GROUP BY directReportees
UNION
    SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
        FROM person_reportee manager
       JOIN person_reportee reportee
          ON manager.directly_manages = reportee.pid
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
      GROUP BY directReportees
UNION
    SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
        FROM person_reportee manager
       JOIN person_reportee L1Reportees
          ON manager.directly_manages = L1Reportees.pid
       JOIN person_reportee L2Reportees
          ON L1Reportees.directly_manages = L2Reportees.pid
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
      GROUP BY directReportees
    ) AS T
   GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
    SELECT manager.directly_manages AS directReportees, 0 AS count
        FROM person_reportee manager
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
    SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
        FROM person_reportee manager
       JOIN person_reportee reportee
          ON manager.directly_manages = reportee.pid
       WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
      GROUP BY directReportees
UNION
```

```
SELECT depth1Reportees.pid AS directReportees,
       count(depth2Reportees.directly_manages) AS count
  FROM person_reportee_manager
     JOIN person_reportee L1Reportees
       ON manager.directly_manages = L1Reportees.pid
     JOIN person_reportee L2Reportees
       ON L1Reportees.directly_manages = L2Reportees.pid
      WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
        GROUP BY directReportees
    ) AS T
   GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
   FROM(
    SELECT reportee.directly_manages AS directReportees, 0 AS count
  FROM person_reportee_manager
     JOIN person_reportee reportee
       ON manager.directly_manages = reportee.pid
      WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
        GROUP BY directReportees
UNION
    SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages) AS count
  FROM person_reportee_manager
     JOIN person_reportee L1Reportees
       ON manager.directly_manages = L1Reportees.pid
     JOIN person_reportee L2Reportees
       ON L1Reportees.directly_manages = L2Reportees.pid
      WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
        GROUP BY directReportees
    ) AS T
   GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
  FROM person_reportee_manager
     JOIN person_reportee L1Reportees
       ON manager.directly_manages = L1Reportees.pid
     JOIN person_reportee L2Reportees
       ON L1Reportees.directly_manages = L2Reportees.pid
      WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
    )
```

Cypher vs SQL

```
MATCH (boss) - [:MANAGES*0..3] -> (sub),  
      (sub) - [:MANAGES*1..3] -> (report)  
WHERE boss.name = "John Doe"  
RETURN sub.name AS Subordinate,  
count(report) AS Total;
```

Cypher vs SQL

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
  SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
UNION
  SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
  FROM person_reportee manager
  JOIN person_reportee reportee
  ON manager.directly_manages = reportee.pid
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
UNION
  SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
  FROM person_reportee manager
  JOIN person_reportee L1Reportees
  ON manager.directly_manages = L1Reportees.pid
  JOIN person_reportee L2Reportees
  ON L1Reportees.directly_manages = L2Reportees.pid
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
  FROM person_reportee manager
  JOIN person_reportee reportee
  ON manager.directly_manages = reportee.pid
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
UNION
  SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages) AS count
  FROM person_reportee manager
  JOIN person_reportee L1Reportees
  ON manager.directly_manages = L1Reportees.pid
  JOIN person_reportee L2Reportees
  ON L1Reportees.directly_manages = L2Reportees.pid
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
  FROM person_reportee manager
  JOIN person_reportee reportee
  ON manager.directly_manages = reportee.pid
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
  GROUP BY directReportees
)) AS T
GROUP BY directReportees)
UNION
(SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
```

```
MATCH (boss)-[:MANAGES*0..3]->(sub),
      (sub)-[:MANAGES*1..3]->(report)
WHERE boss.name = "John Doe"
RETURN sub.name AS Subordinate,
       count(report) AS Total
```

Less time writing queries

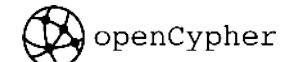
- More time understanding the answers
- Leaving time to ask the next question

Less time debugging queries:

- More time writing the next piece of code
- Improved quality of overall code base

Code that's easier to read:

- Faster ramp-up for new project members
- Improved maintainability & troubleshooting

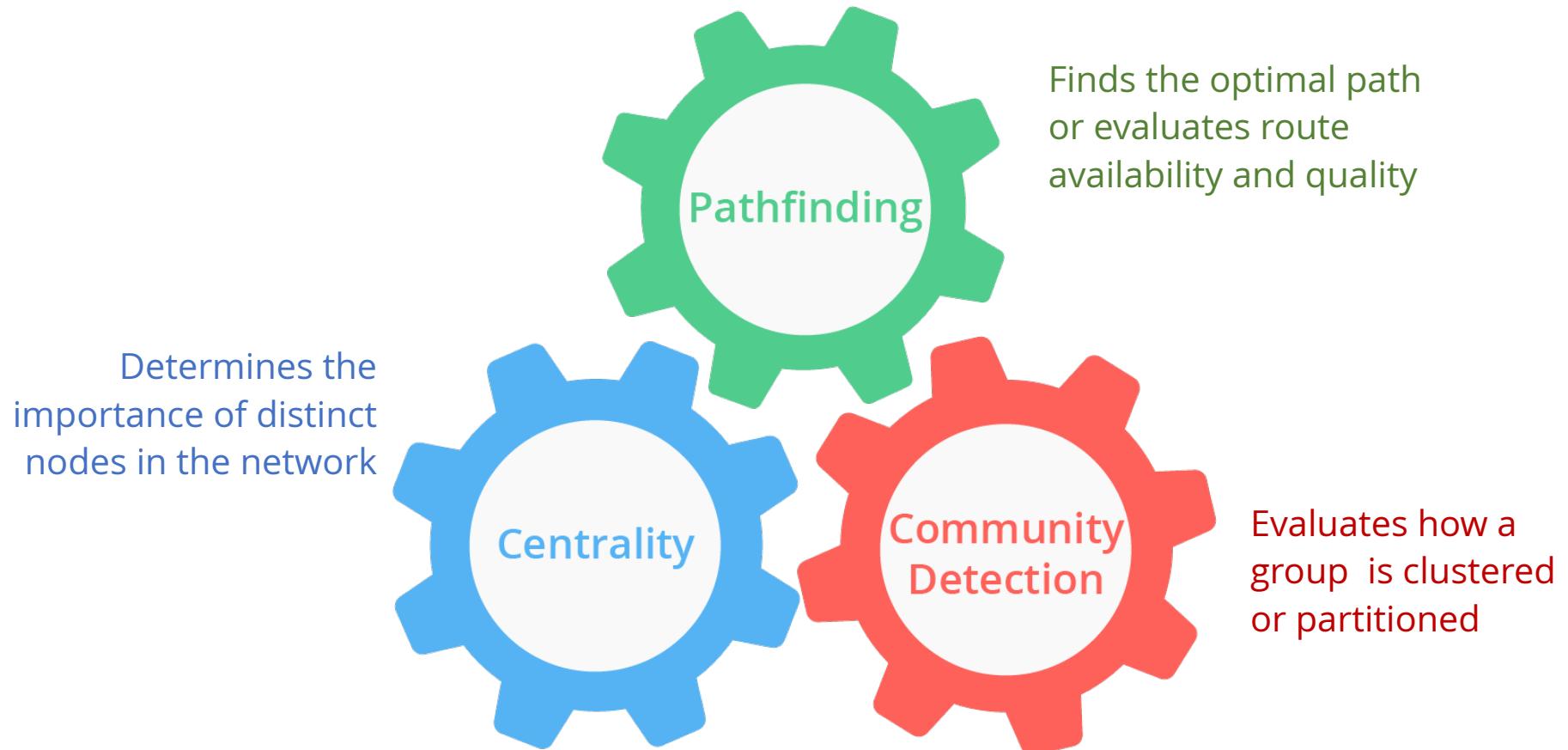


Indexes and Query Tuning

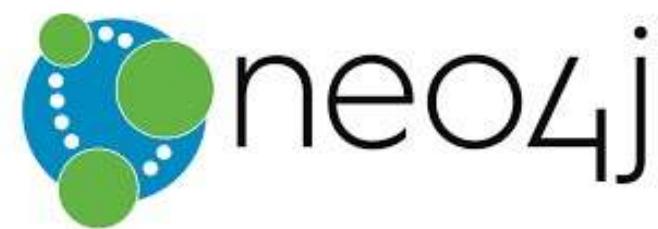
- Indexes can be added to improve search performance
`CREATE INDEX [index_name]
FOR (n:LabelName)
ON (n.propertyName)`
 - Different indexes, recommended BTREE
 - Composite (multiproperty) indexes can be defined as well
 - Full-text indexes can be defined as well (Apache Lucene)
- The use of indexes by the query optimizer can be checked by looking at the query plan (`EXPLAIN`) or suggested by `USING`

Other querying mechanisms

- Java API
- Graph algorithm library



Architecture



neo4j in short

| Feature | neo4j |
|---------------------|---|
| Model | Graph-based |
| Query language | Supported, Cypher |
| Reference scenarios | transactional (read intensive) & analytical |
| Partitioning | Difficult (application-level) |
| Indexes | on properties (simple and composite), full-text |
| Replication | Master-slave, single-leader |
| Consistency | Strong |
| Availability | Load balancing among read replicas |
| Fault tolerance | By re-electing a master in case it goes down |
| Transactions | ACID (maintain consistency over multiple nodes and edges) |
| CAP theorem | CA |
| Distributed by | Neo4j Inc. |

Indexes and Query Tuning

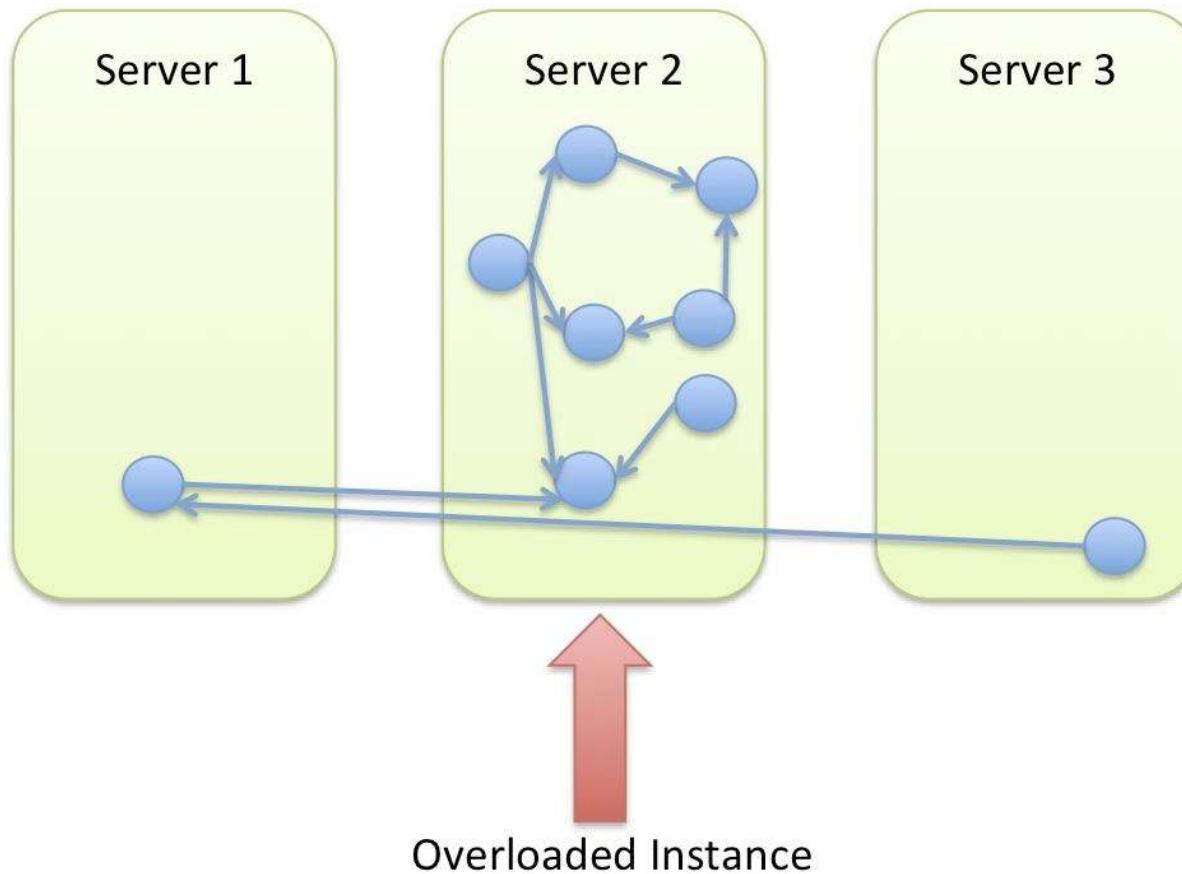
- Indexes can be added to improve search performance
`CREATE INDEX [index_name]
FOR (n:LabelName)
ON (n.propertyName)`
 - Different indexes, recommended BTREE
 - Composite (multiproperty) indexes can be defined as well
 - Full-text indexes can be defined as well (Apache Lucene)
- The use of indexes by the query optimizer can be checked by looking at the query plan (`EXPLAIN`) or suggested by `USING`

Graph Partitioning and Federated Access

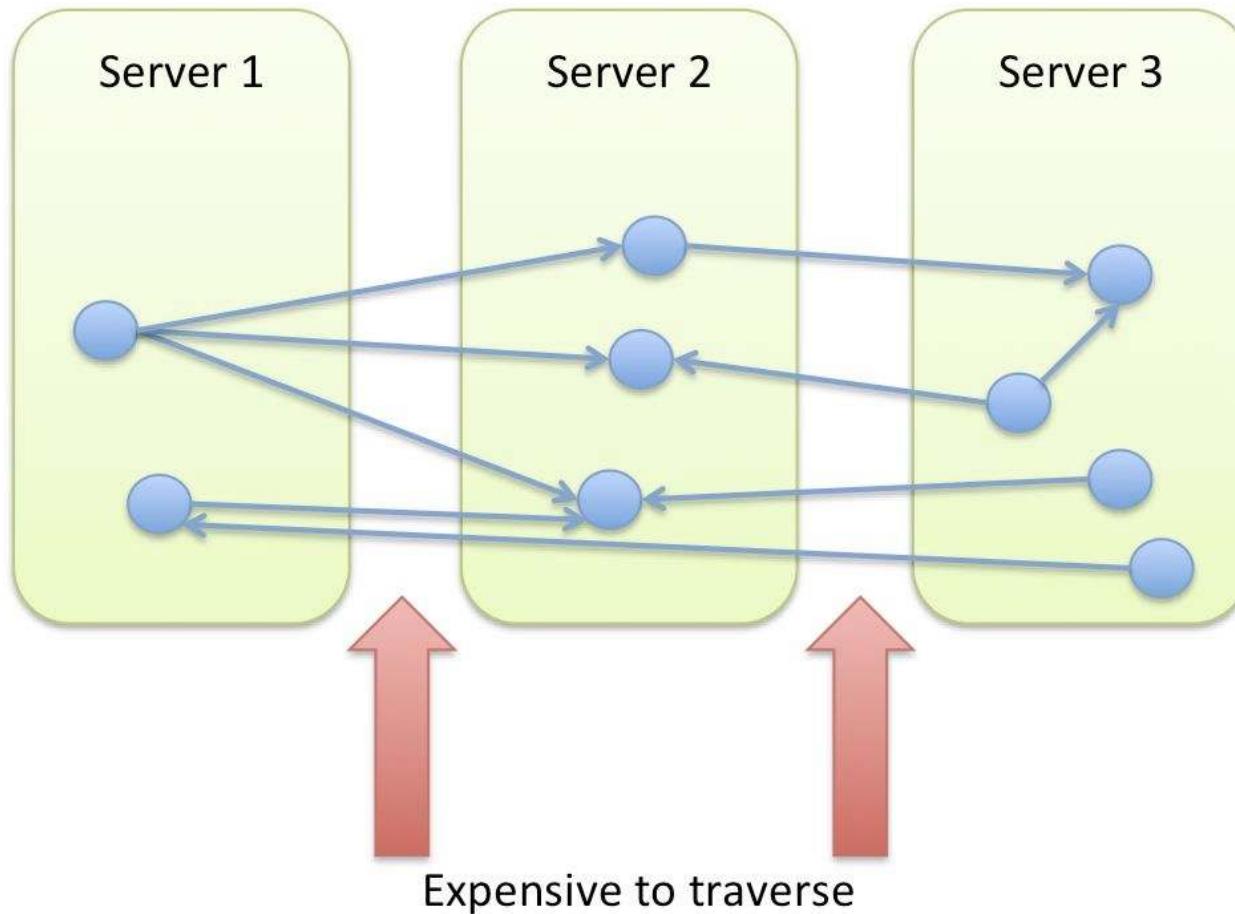
Partitioning

- Partitioning is difficult
- Storing related nodes on the same server is better for graph traversal
- Traversing a graph when the nodes are on different machines is not good for performance
- But any node can be related to any other node ...

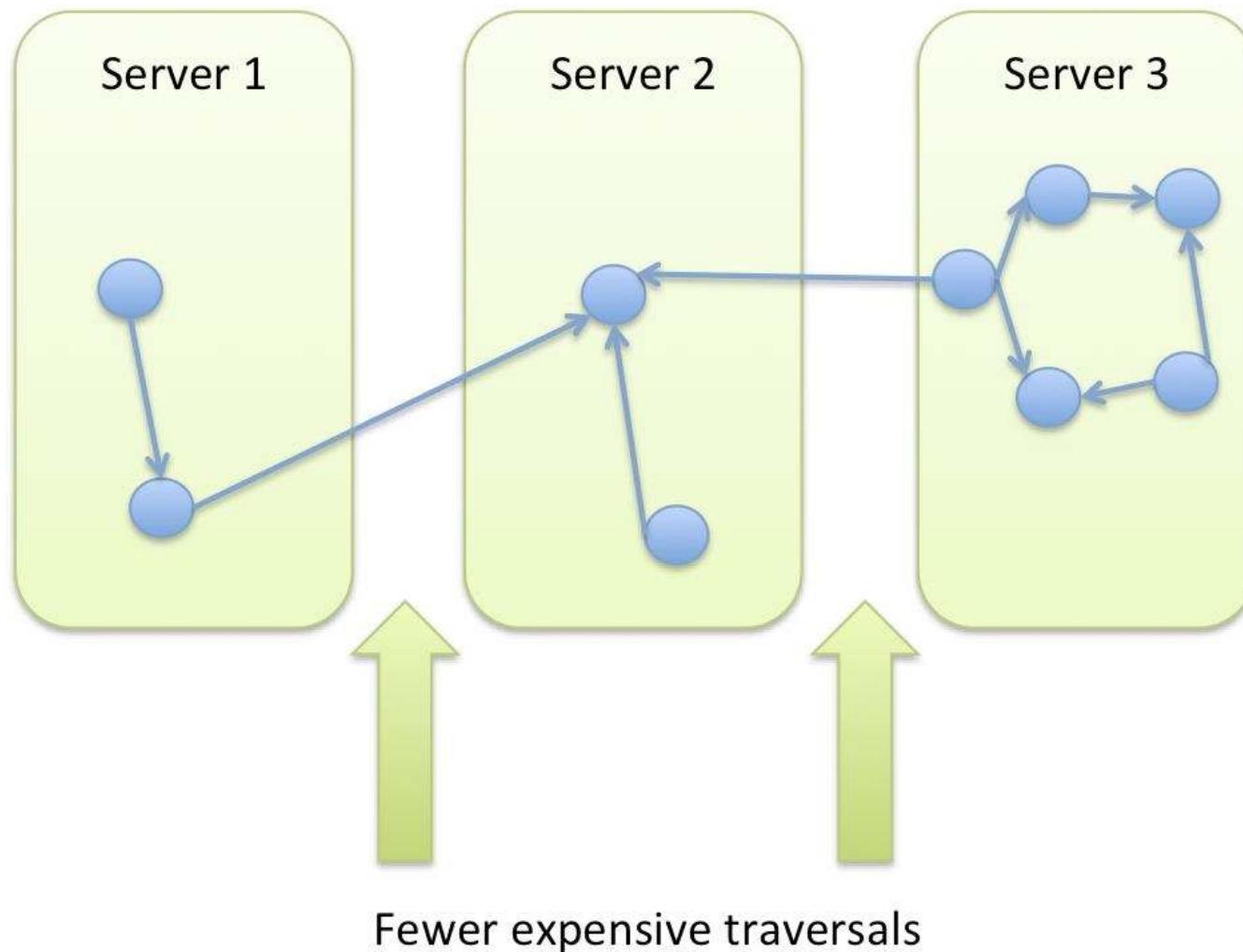
Partitioning - “Black Hole” server



Partitioning - Chatty Network



Partitioning - Minimal Point Cut

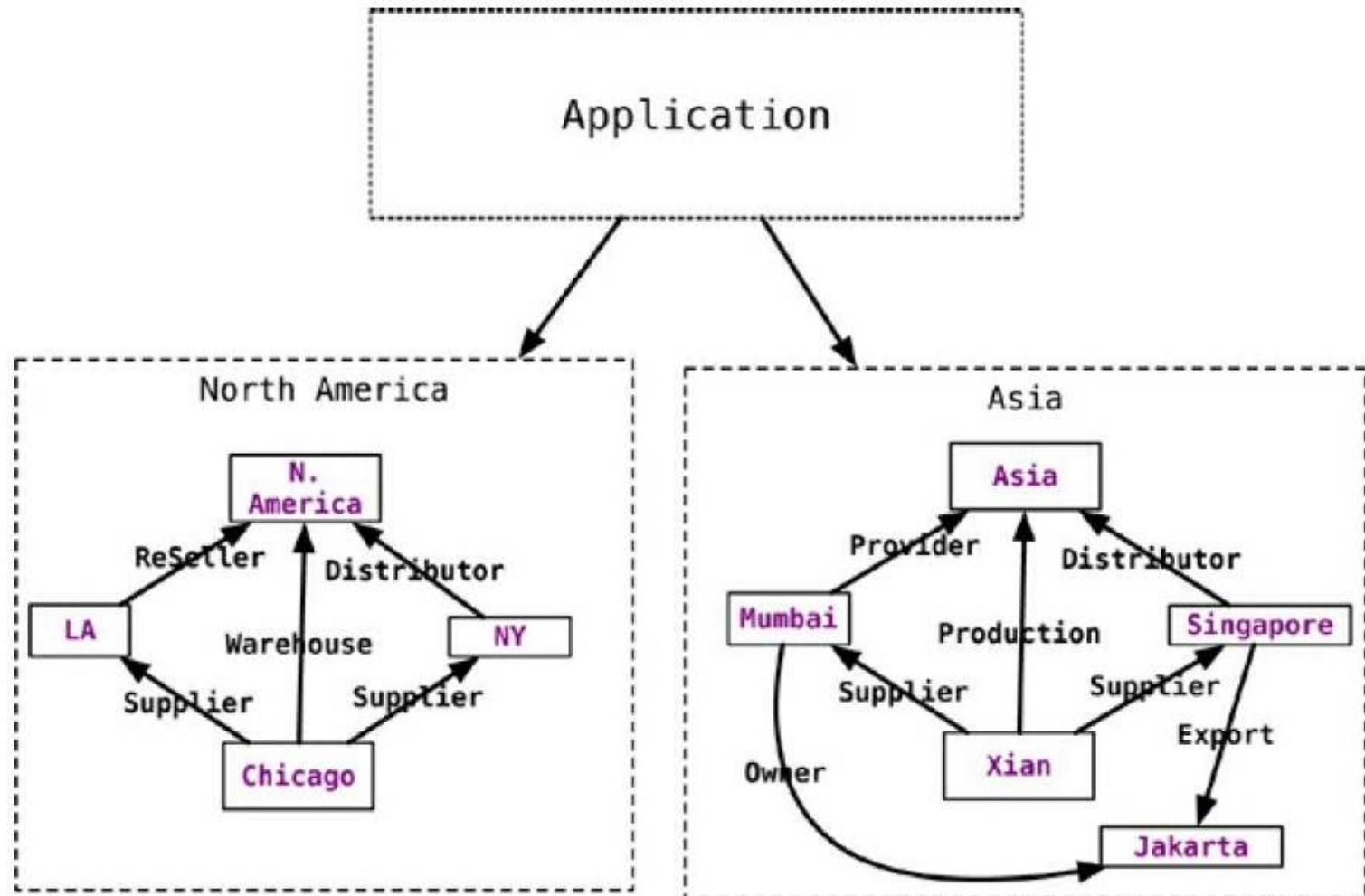


Application-level Partitioning

Partition the data from the application side using domain-specific knowledge

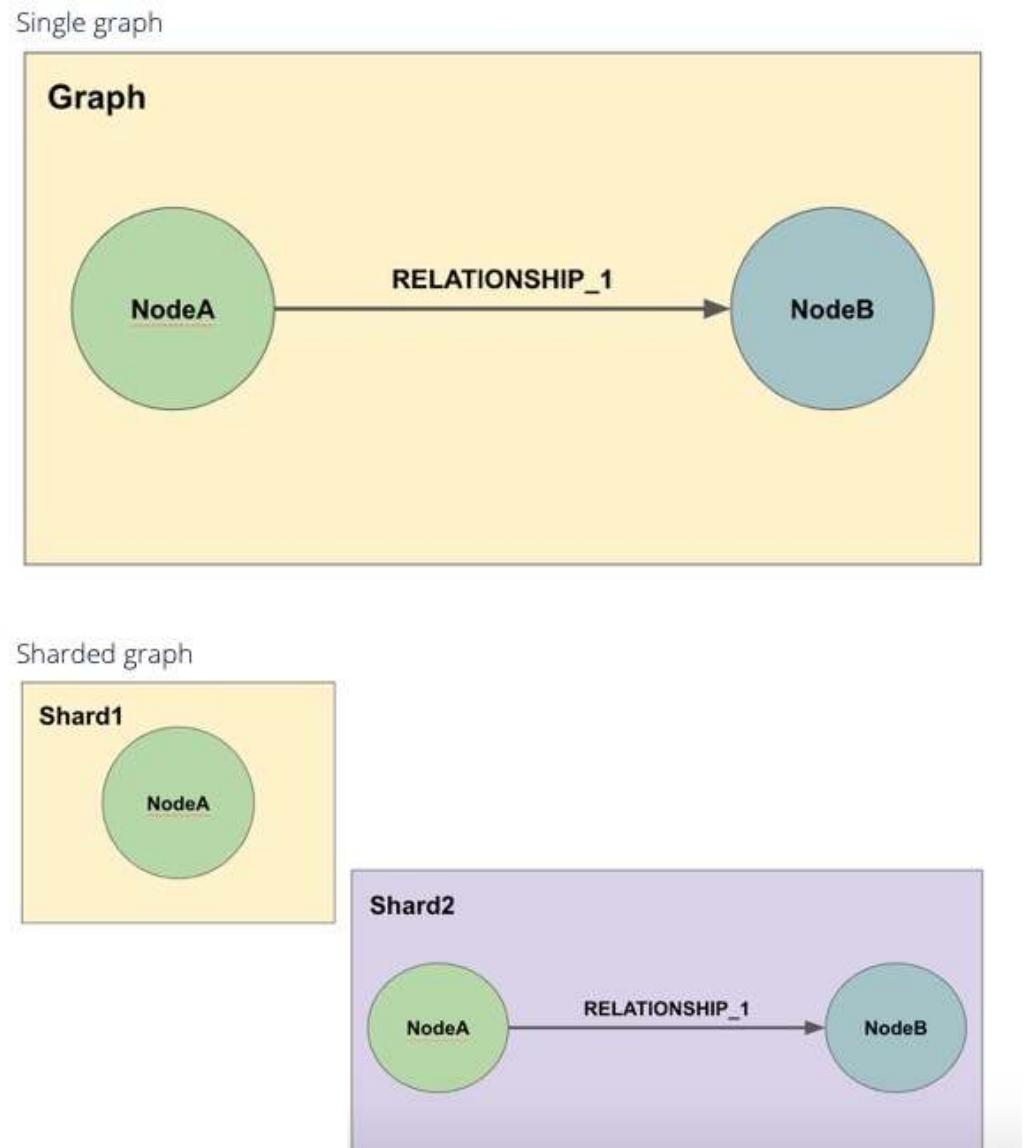
- For example, nodes that relate to the North America can be created on one server while the nodes that relate to Asia on another
- This application-level partitioning needs to understand that nodes are stored on physically different databases

Application-Level Partitioning



Partitioning Data with Neo4j Fabric

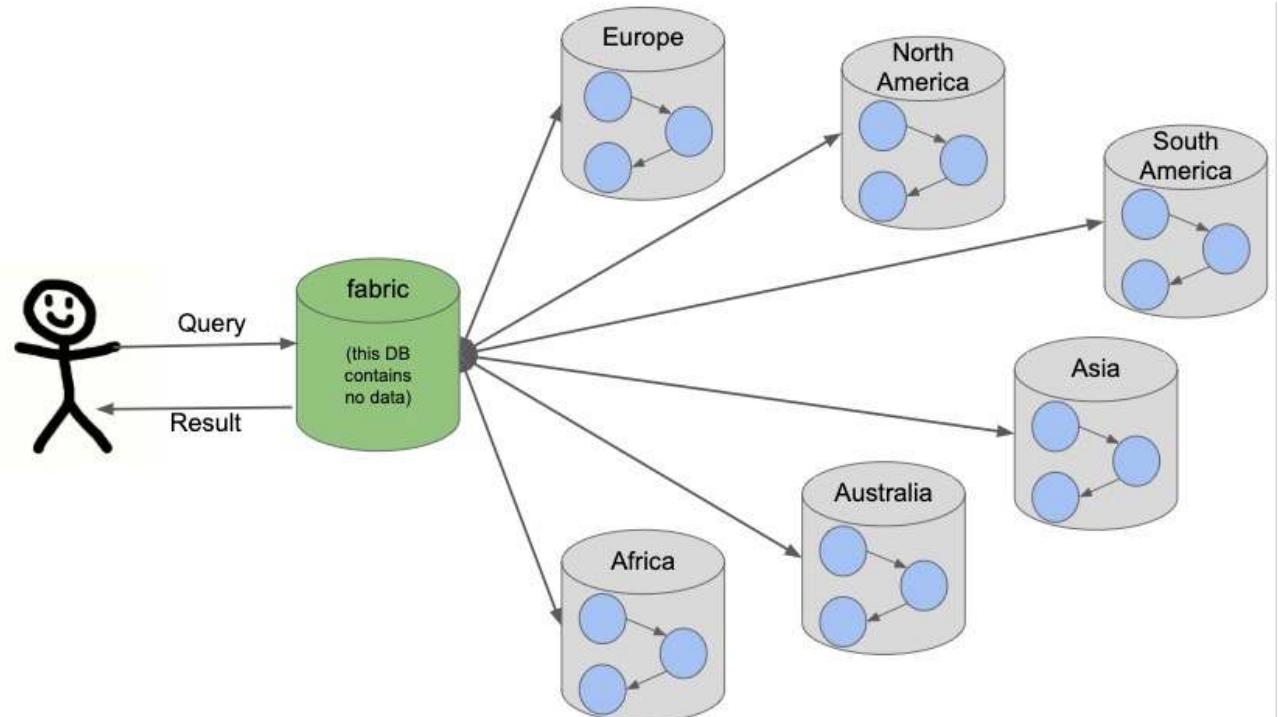
- Enable users to break a larger graph down into individual, smaller graphs and to store them in separate databases
- For highly-connected graphs, some **redundancy** to maintain the relationships between entities



<https://neo4j.com/developer/neo4j-fabric-sharding/>

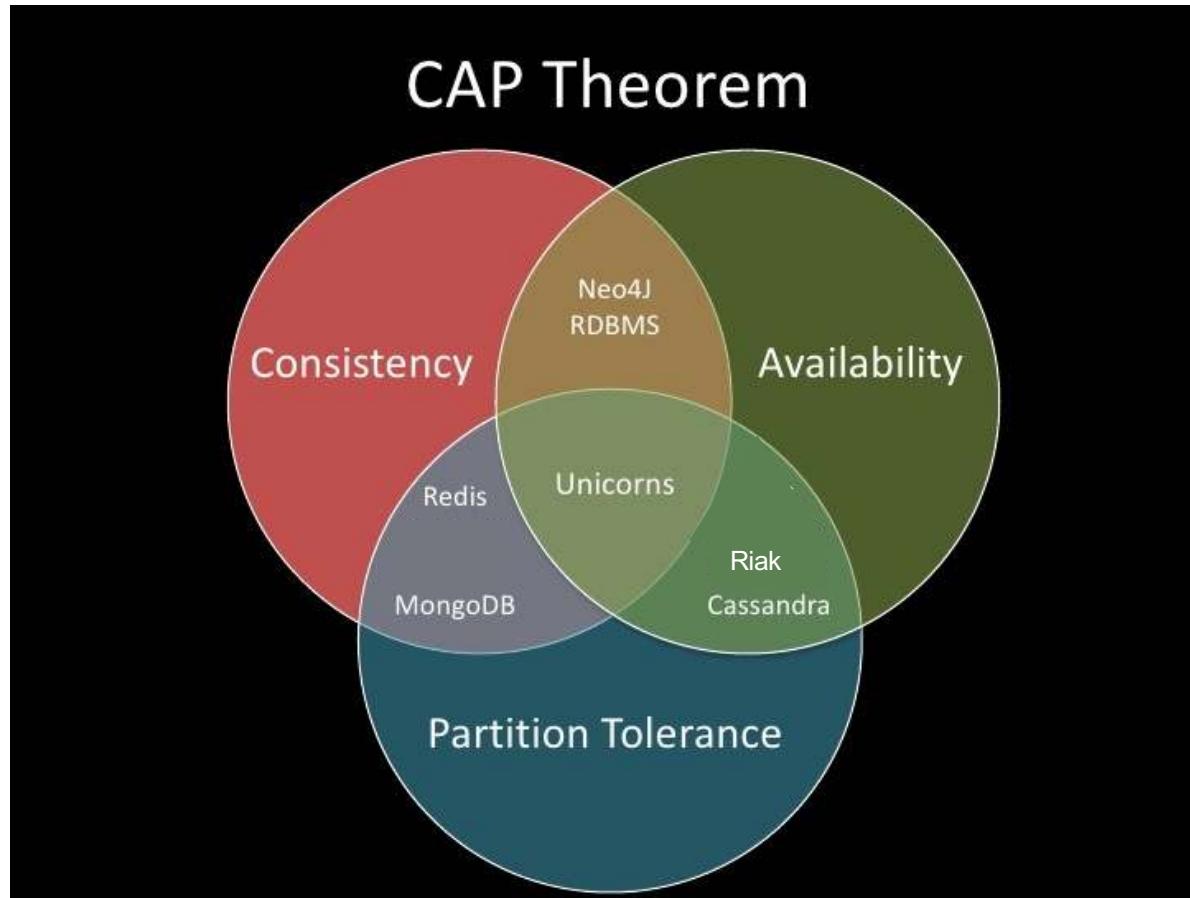
Partitioning with Neo4j Fabric

- The Fabric database is actually a virtual database
 - it cannot store data, but acts as the entrypoint into the rest of the graphs
 - like a proxy server that handles requests and connection information
 - it helps distribute load and sends requests to the appropriate endpoint



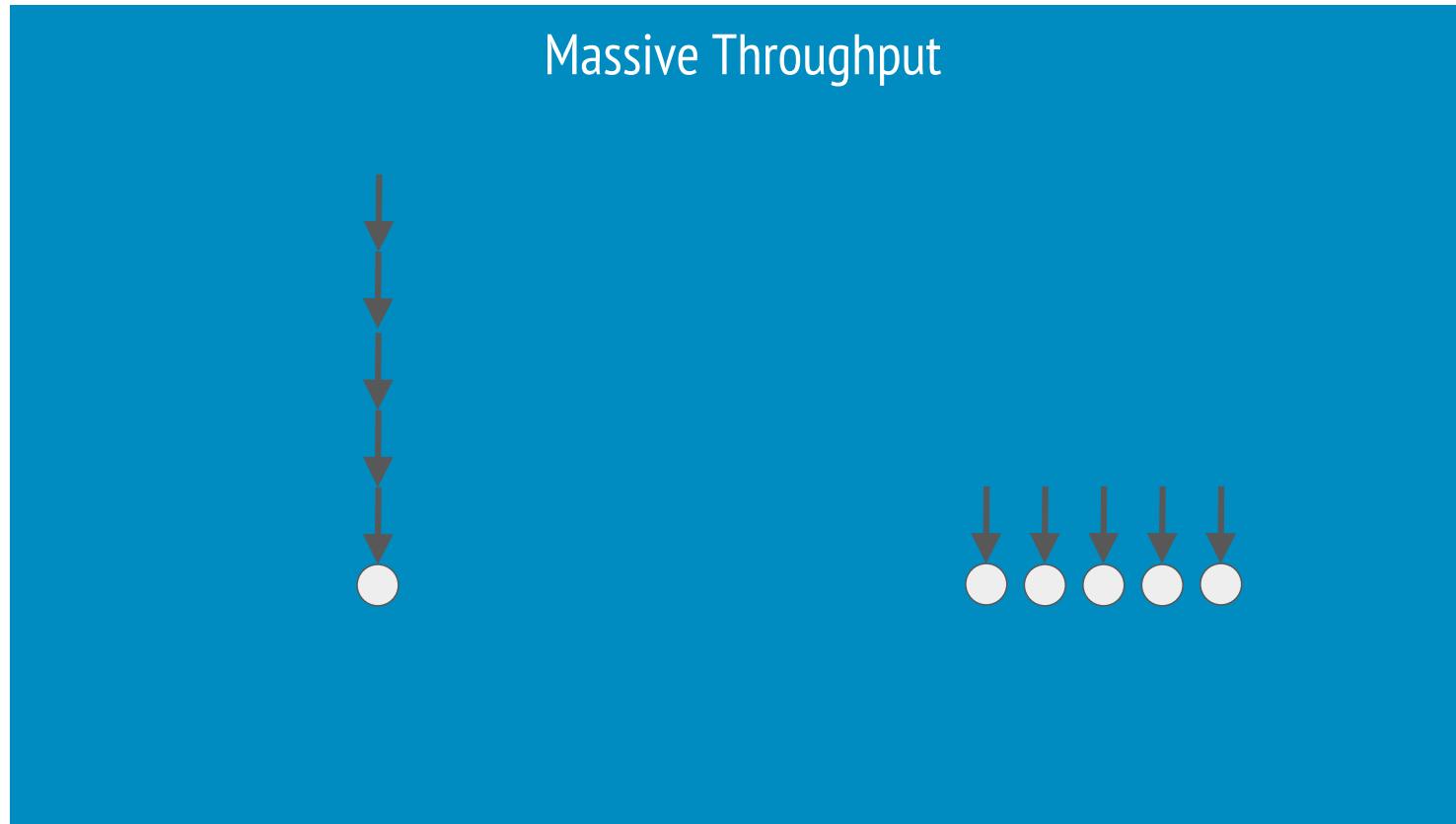
Neo4j on a cluster?
Scalability and availability

Consistency & Availability



Master-slave replication model
ACID-compliant transactions

Even if we cannot partition,
can graph dbs exploit running on a cluster?

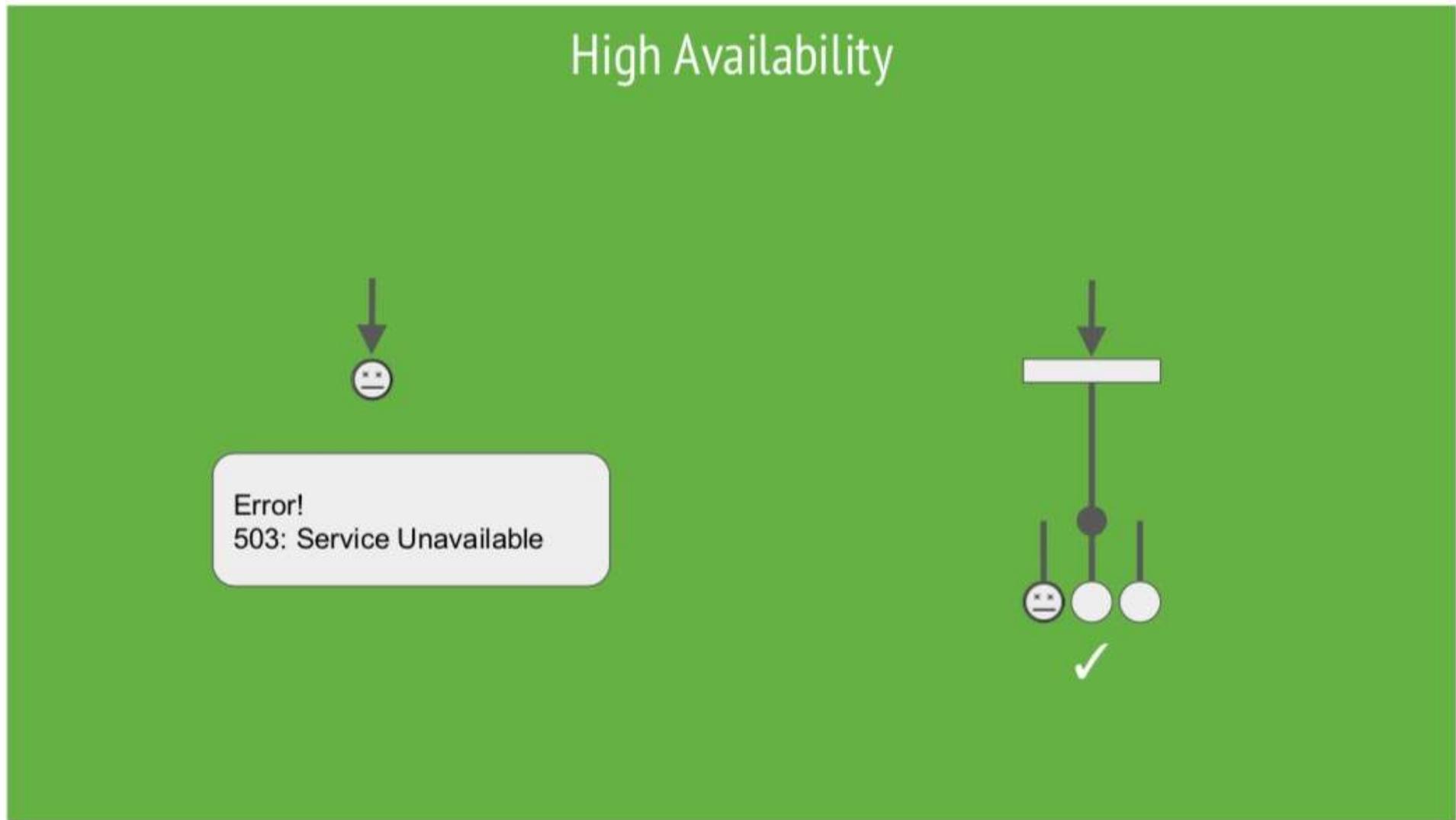


Even if we cannot partition,
can graph dbs exploit running on a cluster?

Data Redundancy



Even if we cannot partition,
can graph dbs exploit running on a cluster?



Even if we cannot partition,
can graph dbs exploit running on a cluster?



Availability & Scaling

- Master-slave **replication**
 - Several slave databases can be configured to be **exact replicas** of a single **master** database
- **Speed-up** of read operations
 - A horizontally scaling **read-mostly** architecture
 - Enables to handle more read load than a single node
- **Fault-tolerance**
 - In case a node becomes unavailable
 - **Transactions** are still **atomic**, consistent and durable, but eventually **propagated** to the slaves

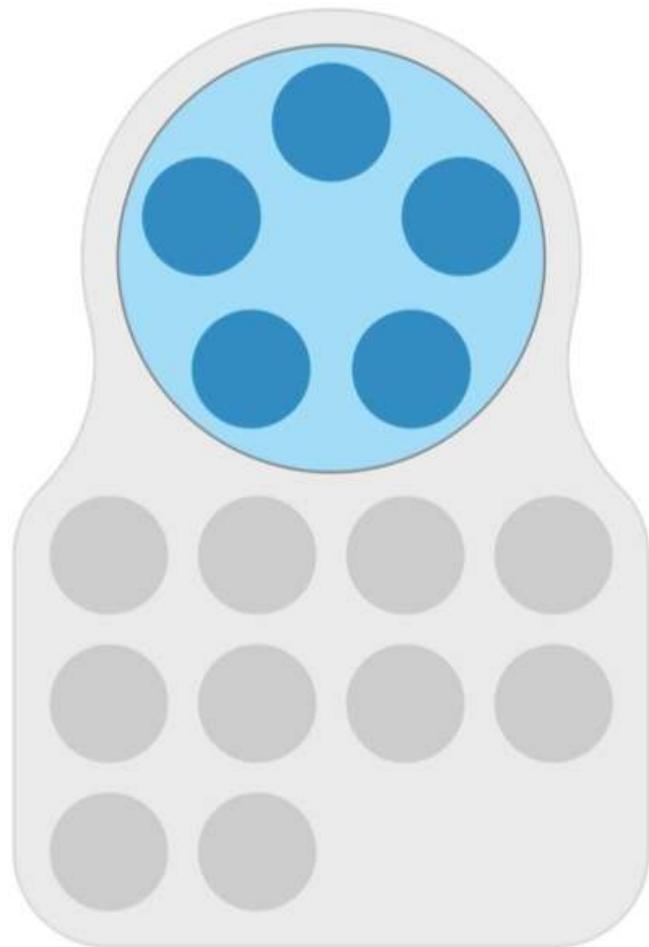
Consistency

- Consistency ensured through transactions
- All Cypher statements are explicitly run within a transaction
 - (full ACID-compliance holds for most graph databases as well)
- When running in a cluster
 - a write to the master is eventually synchronized to the slaves
 - slaves are always available for read
 - writes to slaves maybe allowed
 - they are immediately synchronized to the master and result in success only after having been committed at the master
 - other slaves will not be synchronized immediately (they will have to wait for the data to propagate from the master)

Availability

- High availability can be achieved by providing for replicated slaves
- Slaves can also handle writes:
 - When they are written to, they synchronize the write to the current master, and the write is committed first at the master and then at the slave
 - Other slaves will eventually get the update
- Keeping track of the last transaction IDs persisted on each slave node and the current master node
 - Once a server starts up, it finds out which server is the master
 - If the server is the first one to join the cluster, it becomes the master
 - When a master goes down, the cluster elects a master from the available nodes

Consensus Commit: Core

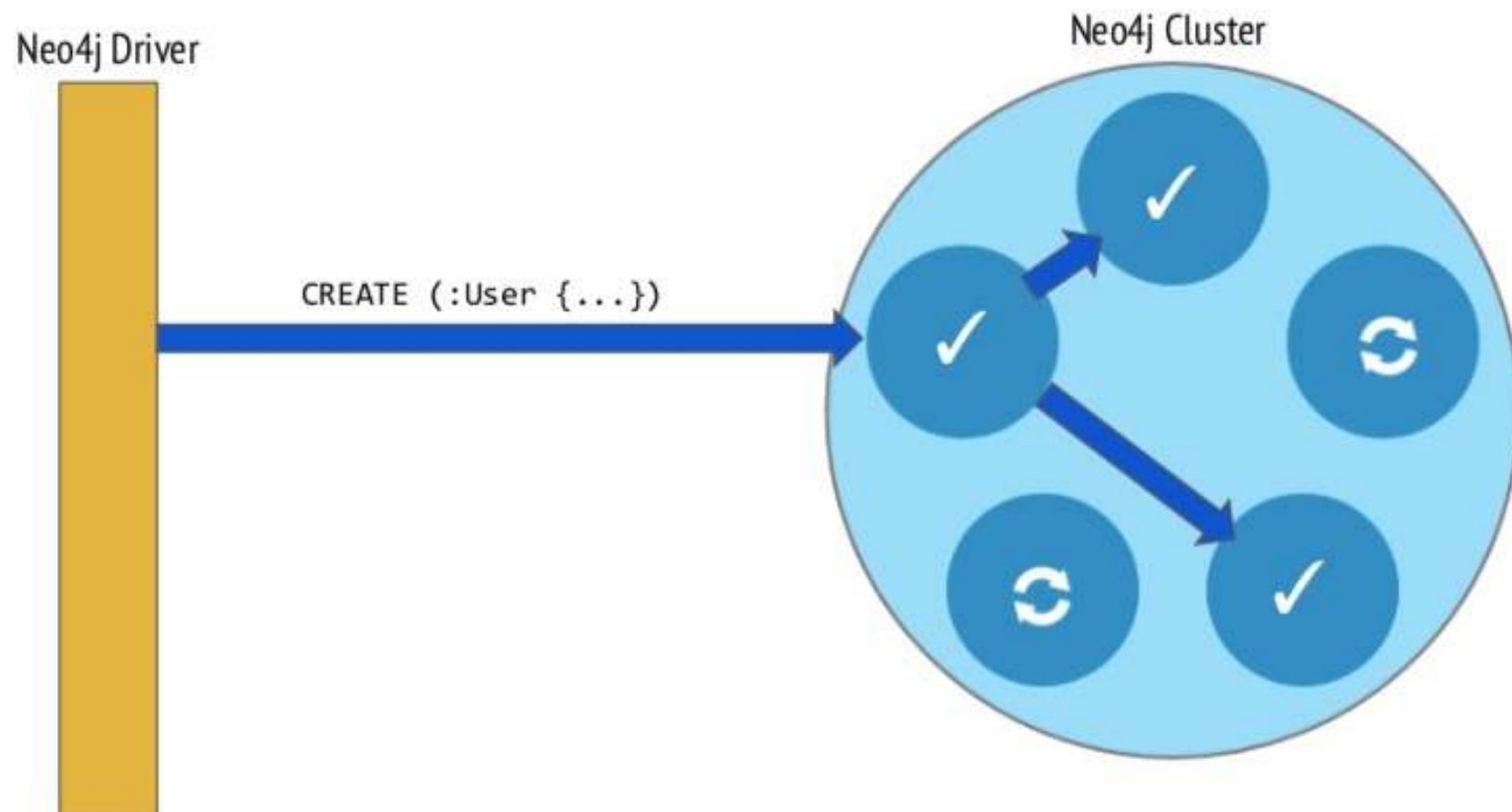


Core

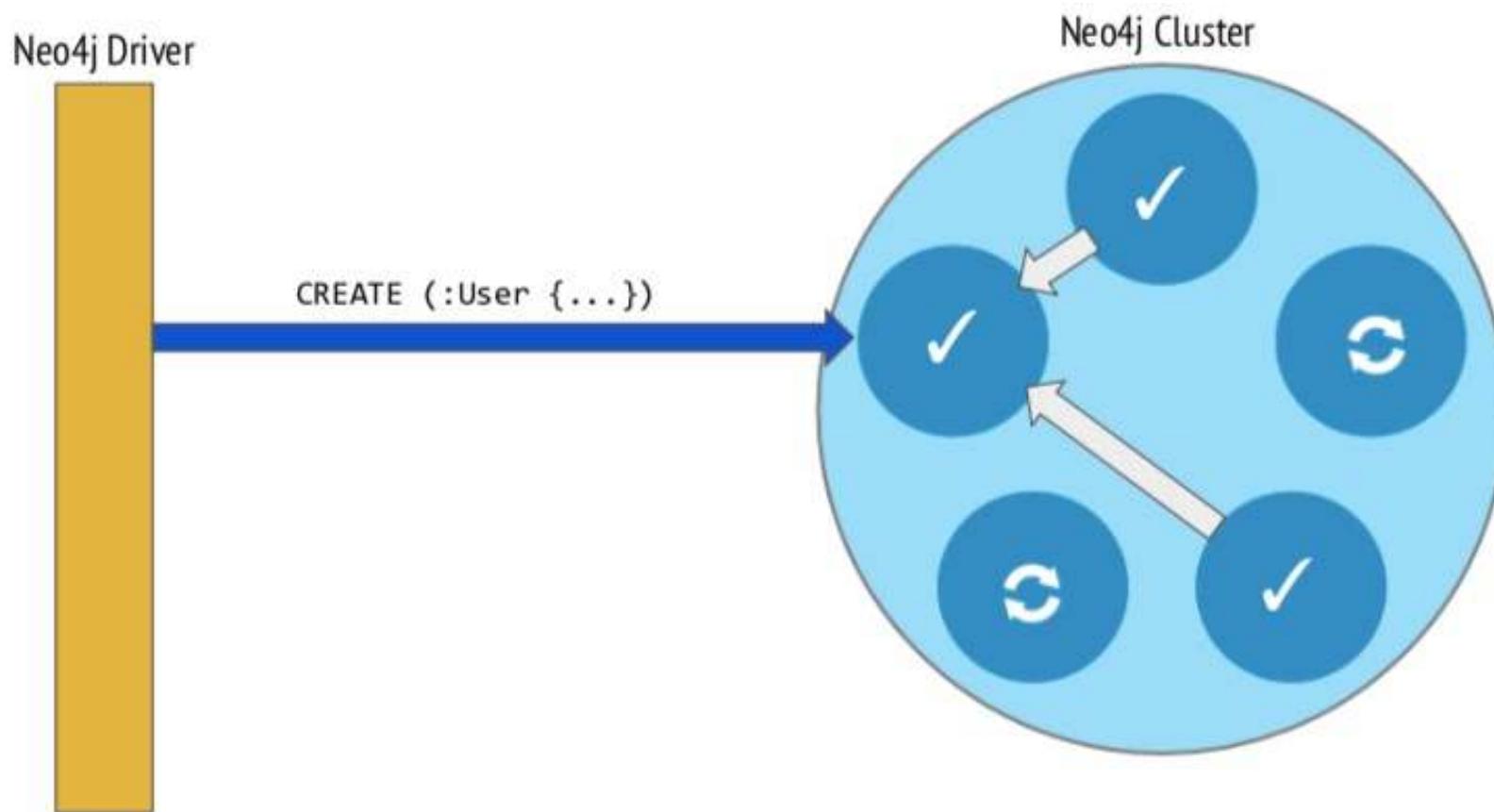
- Small group of Neo4j databases
- Fault-tolerant Consensus Commit
- Responsible for data safety

Raft consensus protocol

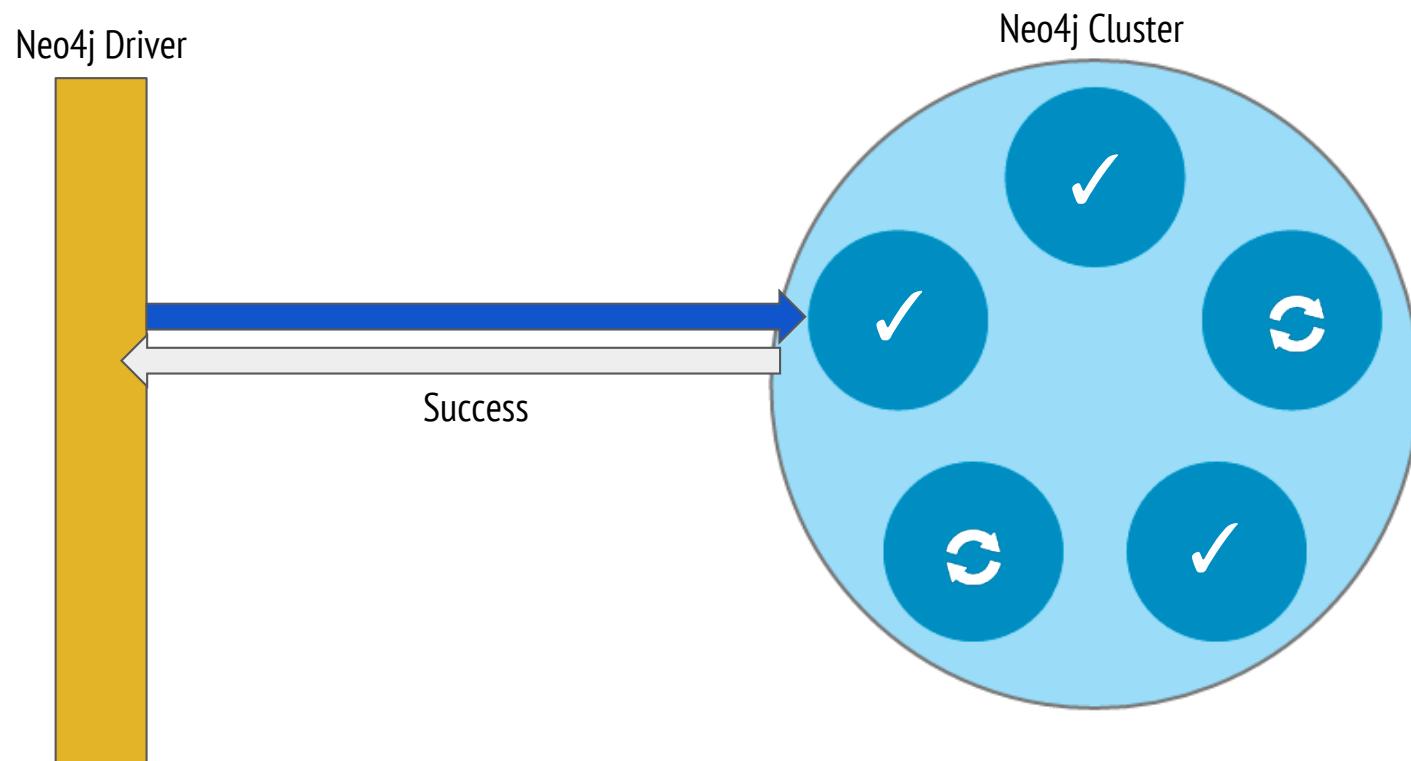
Writing to Core



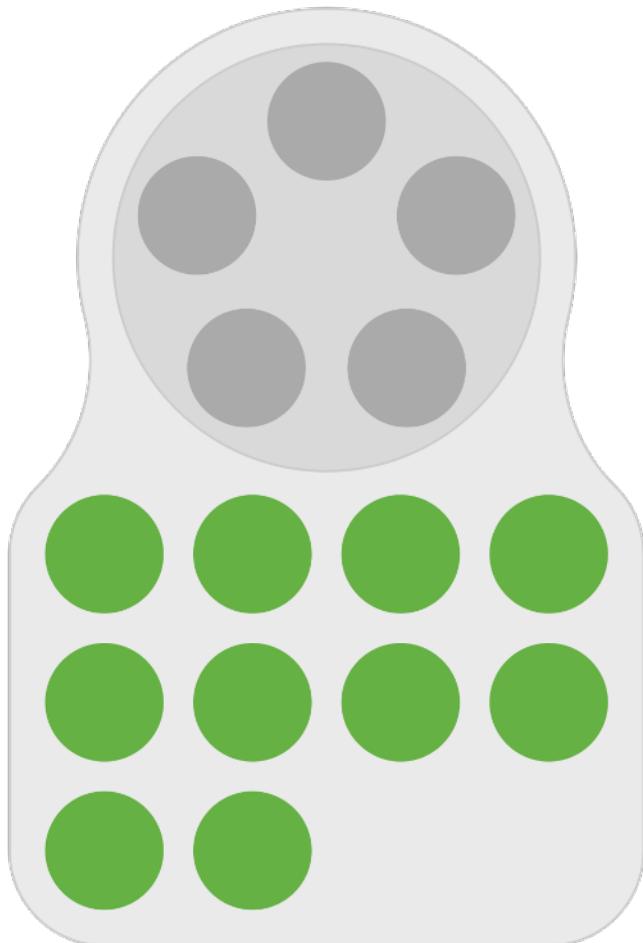
Writing to Core



Writing to Core



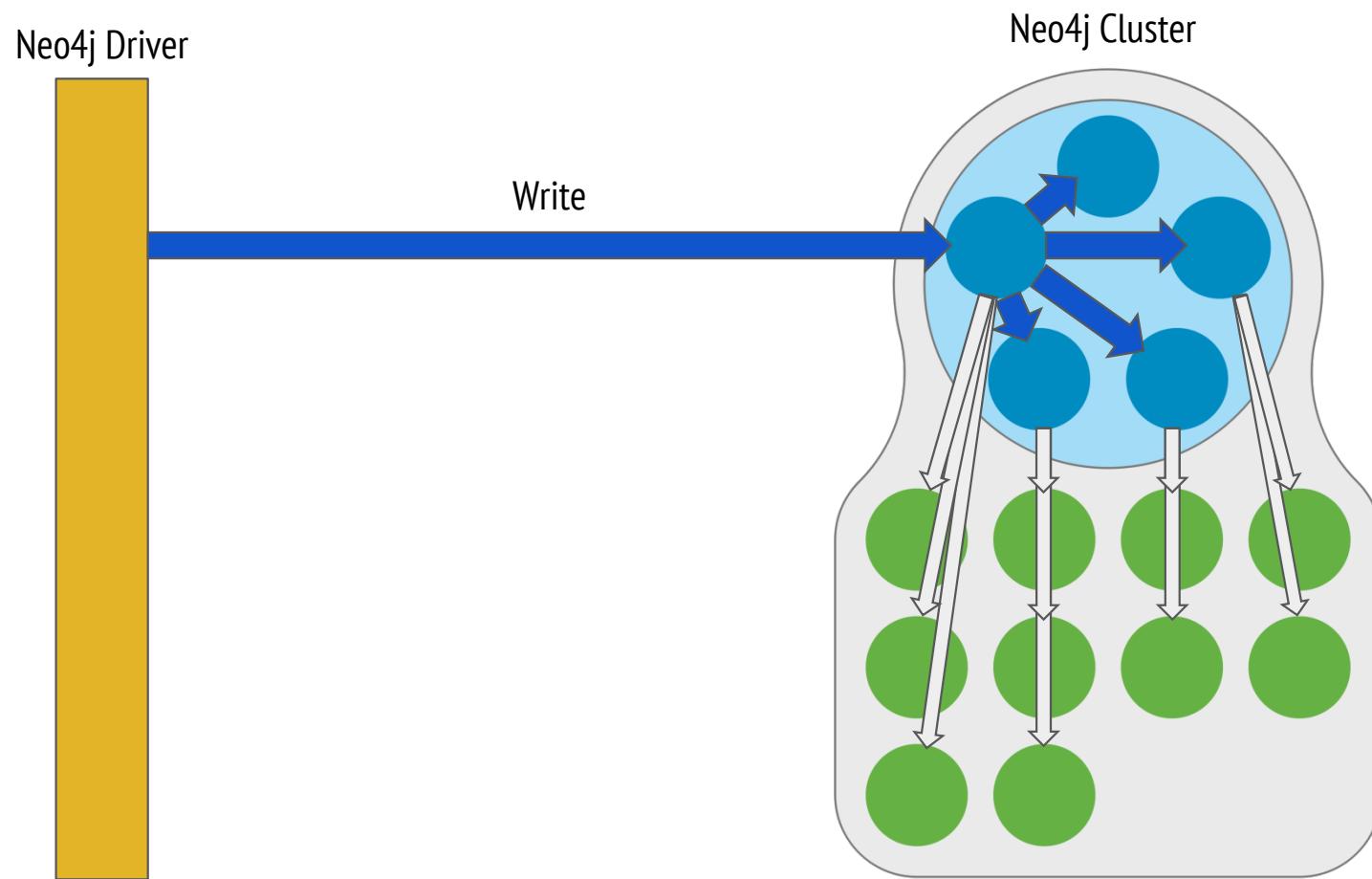
Read Replicas



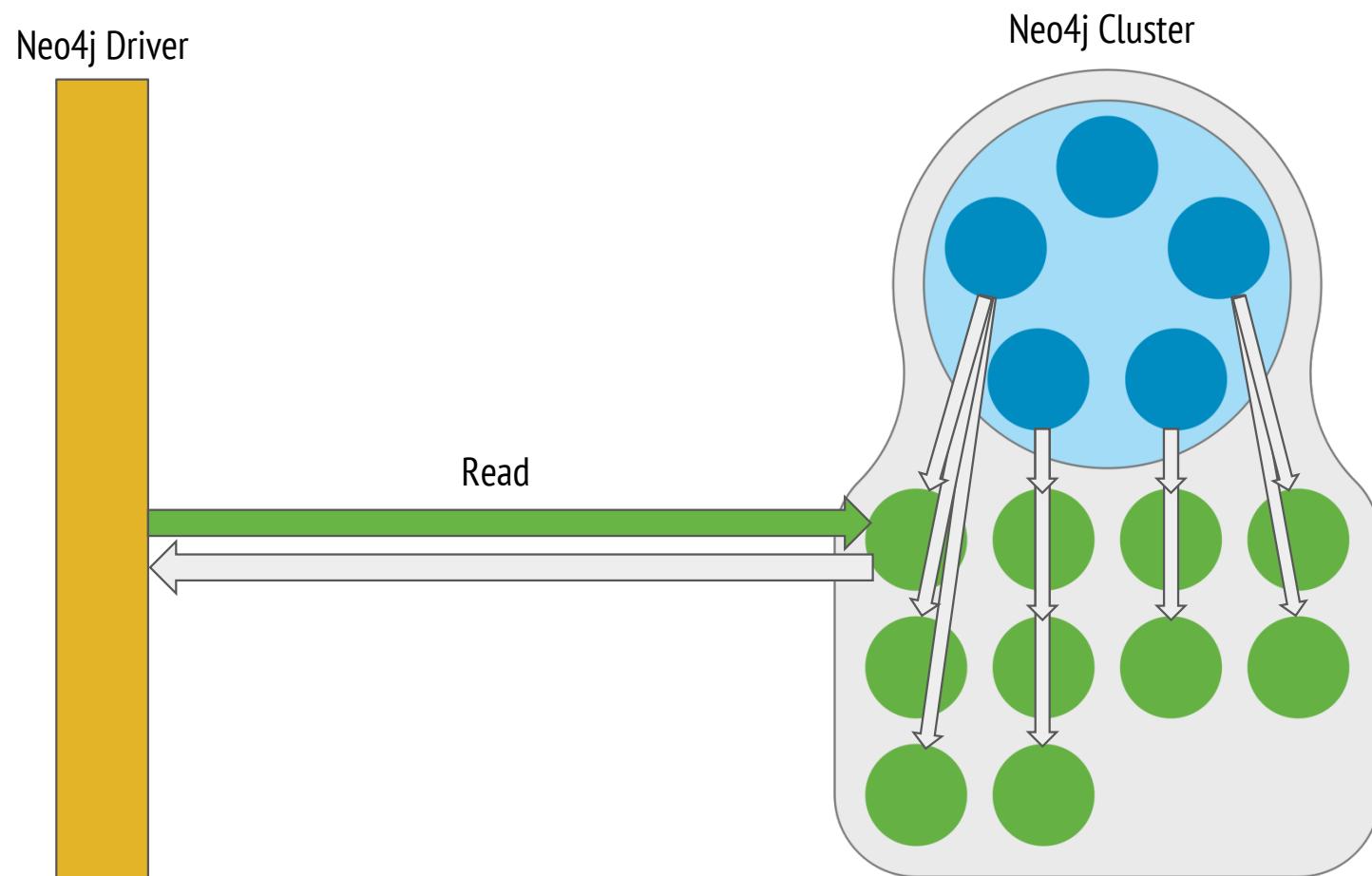
Read Replicas

- For massive query throughput
- Read-only replicas
- Not involved in Consensus Commit
- Disposable, suitable for auto-scaling

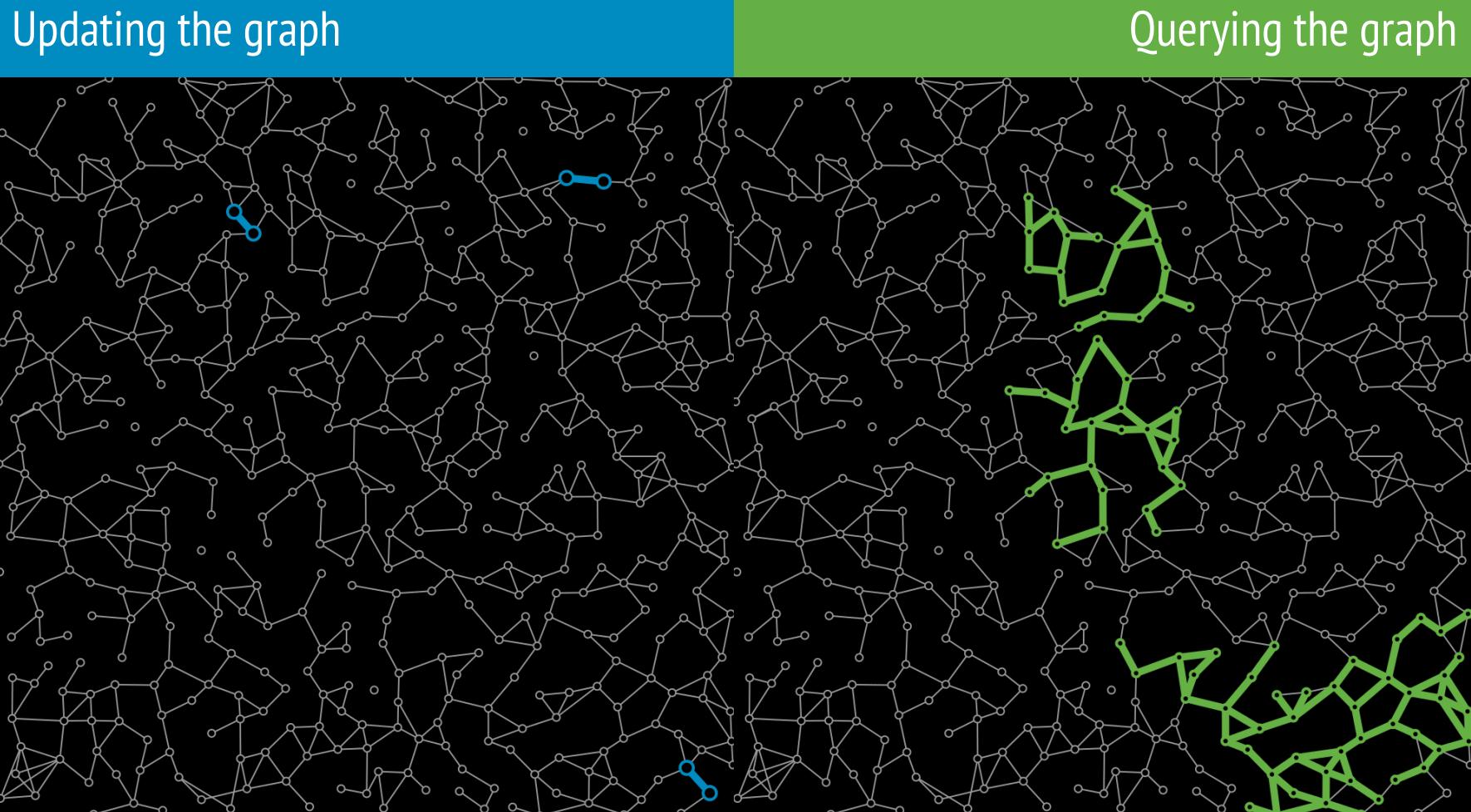
Propagating Updates to Read Replicas



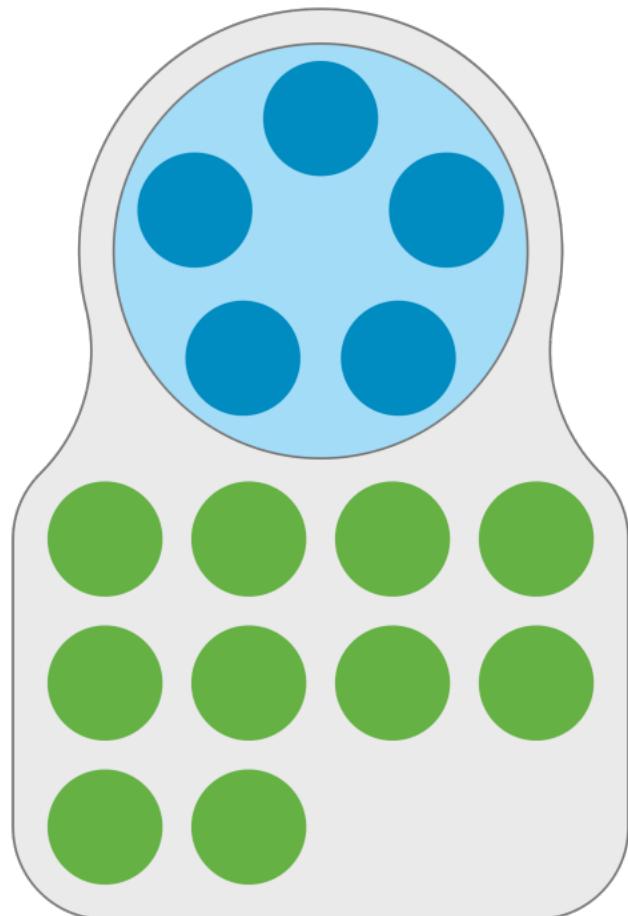
Reading from Read Replicas



Updating vs Reading



Updating vs Reading



Core

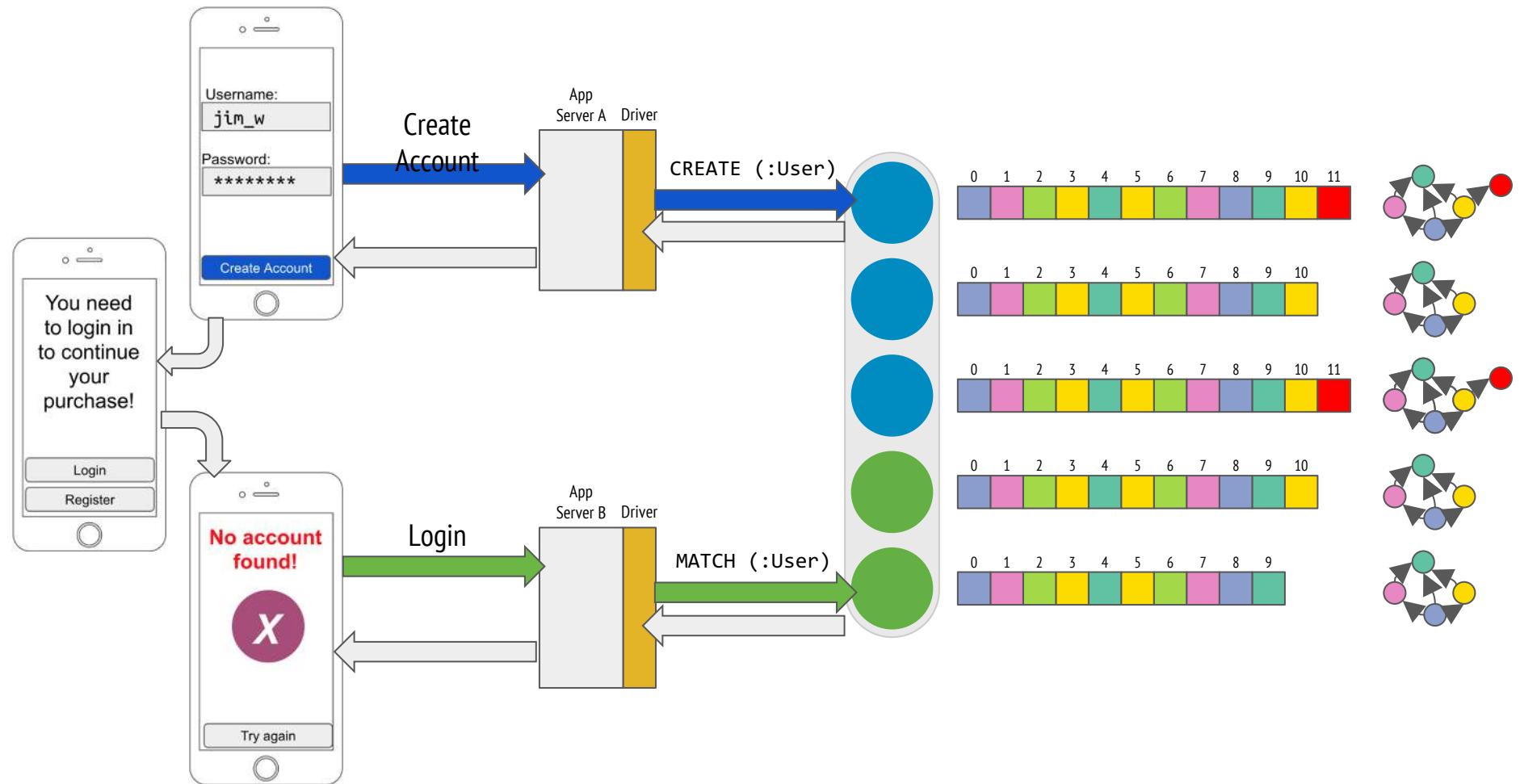
Updating the graph

Read

Queries, analysis, reporting

Replicas

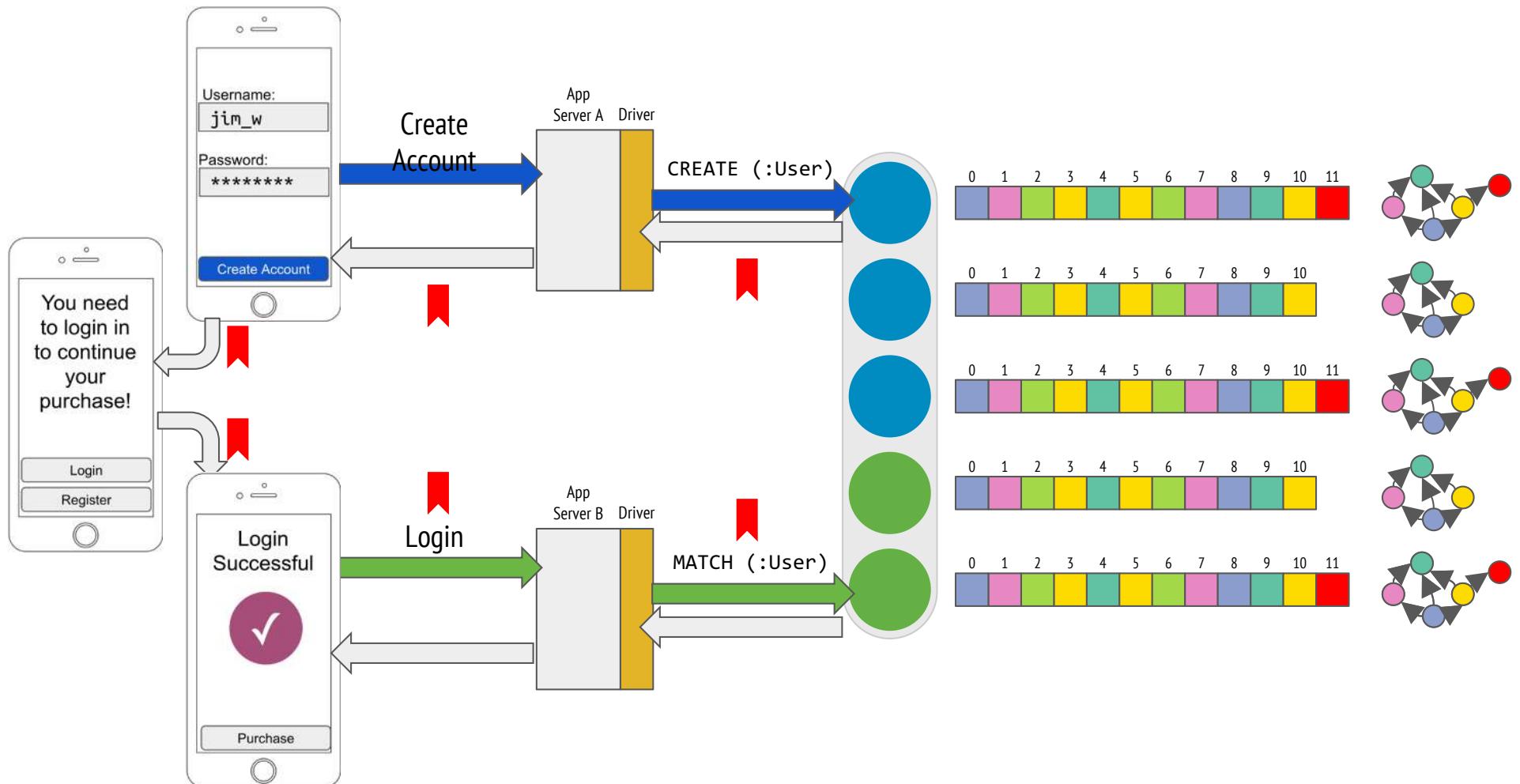
From Eventual to Causal Consistency



From Eventual Consistency to Causal Consistency

- On executing a transaction, the client can ask for a **bookmark** which it then presents as a parameter to subsequent transactions
- This way, the cluster can ensure that only servers which have processed the client's bookmarked transaction will run its next transaction
- This provides a *causal chain* which ensures correct **read-after-write semantics** from the client's point of view

From Eventual to Causal Consistency - Bookmarks



when invoked, a client application is guaranteed to read at least its own writes

| Feature | In Neo4j (EE) |
|-----------------|---|
| Partitioning | Difficult Application-level |
| Replication | Core nodes & read replicas |
| Consistency | Causal consistency (read your own writes) |
| Availability | Load balancing among read replicas for read (core nodes for writes) |
| Fault tolerance | Consensus commit among core nodes (Raft protocol) |
| Transactions | ACID (maintain consistency over multiple nodes and edges) |
| CAP theorem | CA |

Use Cases

Graph DBs: Suitable Use Cases

- Connected Data
 - Social networks
 - Any link-rich domain is well suited for graph databases
- Routing, Dispatch, and Location-Based Services
 - Node = location or address that has a delivery
 - Graph = nodes where a delivery has to be made
 - Relationships = distance
- Recommendation Engines
 - “your friends also bought this product”
 - “when buying this item, these others are usually bought”

Graph DBs: When Not to Use

- If we want to **update** all or a **subset** of entities
 - Changing a property on many nodes is not straightforward
 - e.g., analytics solution where all entities may need to be updated with a changed property
- **Some** graph databases may be **unable** to handle **lots** of data
 - **Distribution** of a graph is **difficult**

References

- Ian Robinson, Jim Webber & Emil Eifrem, Graph Databases, New Opportunities for Connected Data, 2nd Edition, O'Reilly, 2015
- Sadalage, P. J., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 192 p.
- Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications
- Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, Jonas Partner, Neo4J in Action, Manning, 2015
- <http://neo4j.com/docs>

Credits

- Credits for the slides:
 - Antonio Maccioni, Università Roma Tre
 - Kevin Swingler, University of Stirling
 - David Novak, Masaryk University, Brno
 - Irena Holubova, Charles University Praha
 - Jim Webber, Kevin Van Gundy, neo4j

Graph Data Modelling Exercise

Learning Management System

Graph Modelling – An Outline

- 1.Understand the domain.
- 2.Create high-level sample data.
- 3.Define specific questions for the application.
- 4.Identify entities.
- 5.Identify connections between entities.
- 6.Test the questions against the model.
- 7.Test scalability.

Graph Modelling – The domain

- There are many courses in the LMS, each of which contains a number of lessons that **must be completed in a specific order**.
- Every course grants a certificate upon completion.
- This certificate has a term of validity. When it expires, students must take the course again.
- Students can enroll in as many simultaneous courses as they want to.
- When a student logs in and chooses a course, the LMS must send them to their **latest unfinished lesson**.

Graph Modelling – Sample Data

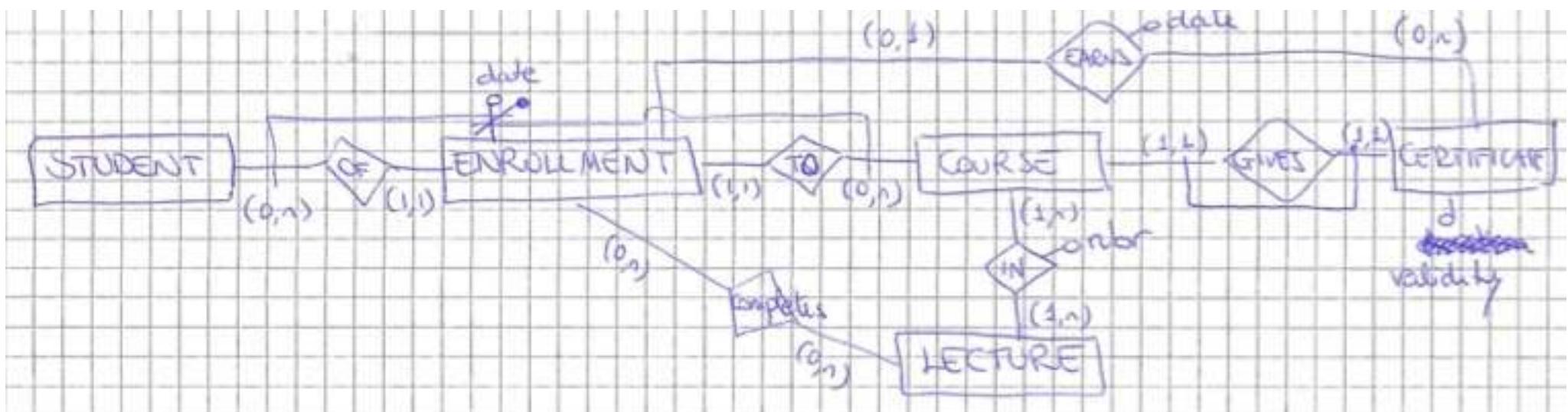
| Courses | Lessons | Certificate |
|-----------------------|---|------------------|
| Introduction to Neo4j | Graph Theory, Graph Databases, Basic Cypher | 2-year validity |
| Neo4j for Developers | Graph Theory, Property Graph, Graph Databases | 6-month validity |

| Students | Completed Courses | In-Progress Courses |
|----------|---|--|
| Alice | Introduction to Neo4j (2016), Introduction to Neo4j (2018) | Introduction to Neo4j (lesson 1) |
| Dan | | Introduction to Neo4j (lesson 3), Neo4j for Developers (lesson 2) |

Graph Modelling – Application Questions

- 1.Which lesson(s) is Dan currently working on?
- 2.What are Alice's current certifications?
- 3.Which lessons are in the Neo4j for Developers course?
- 4.What is the last lesson in the Introduction to Neo4j course?
- 5.Which lesson follows Graph Theory in the Neo4j for Developers course?
- 6.Who has completed Introduction to Neo4j?

An ER Conceptual Model for the Domain



Graph Modelling – List entities = labels for homogeneous sets of nodes

Graph Modelling – List entities = labels for homogeneous sets of nodes

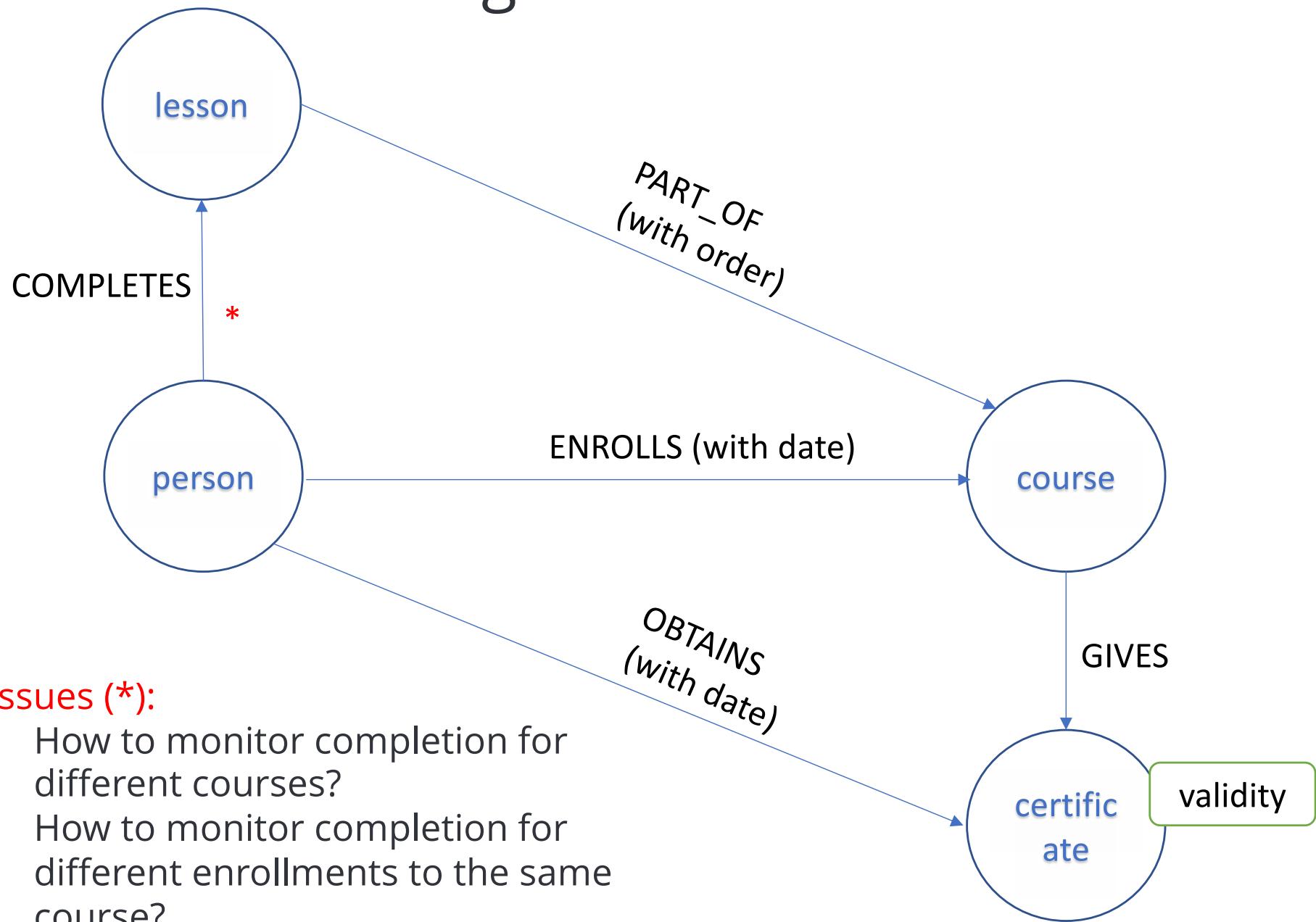
1. www.wooclap.com/GRAHMODEL

STUDENT
COURSE
LESSON
CERTIFICATE

TIME? -> just properties with domain date could be enough?

Latest/unfinished -> via properties/relationships

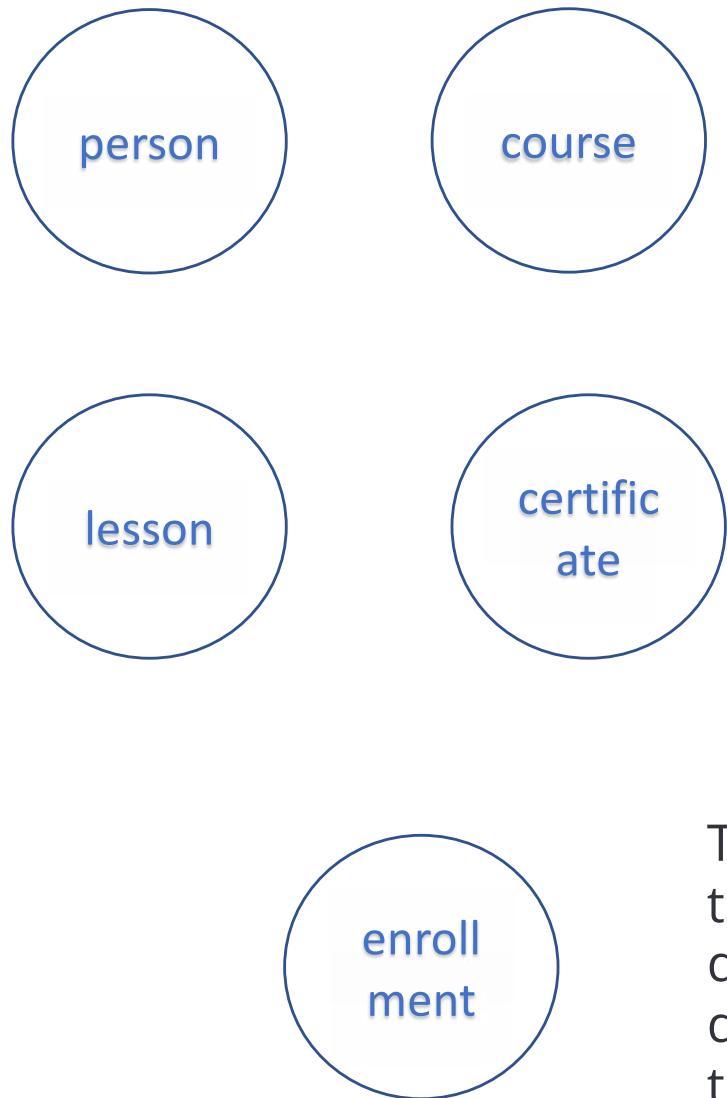
Graph Modelling – List entities = labels for homogeneous sets of nodes



Issues (*):

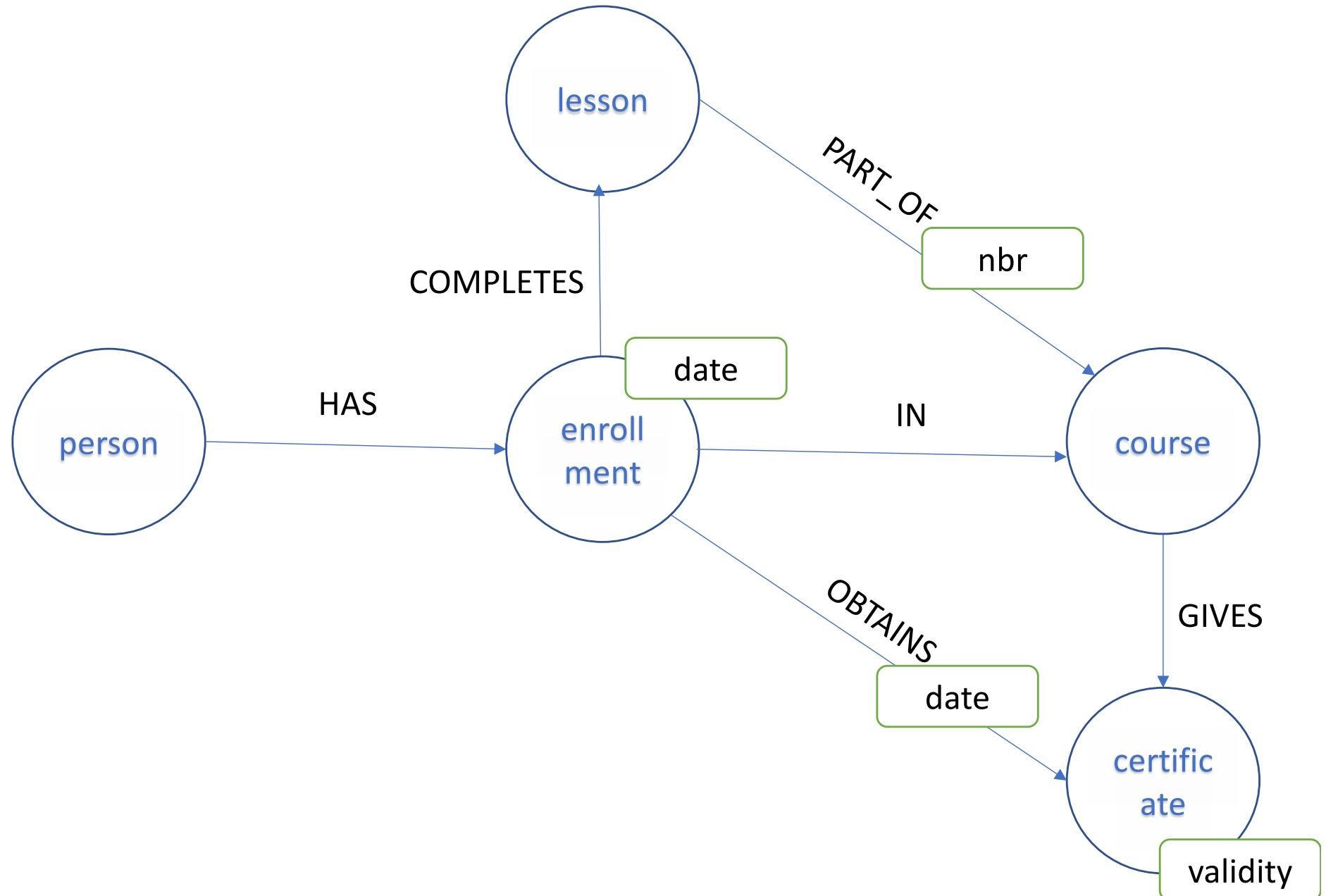
- How to monitor completion for different courses?
- How to monitor completion for different enrollments to the same course?

Graph Modelling – List entities = labels for homogeneous sets of nodes

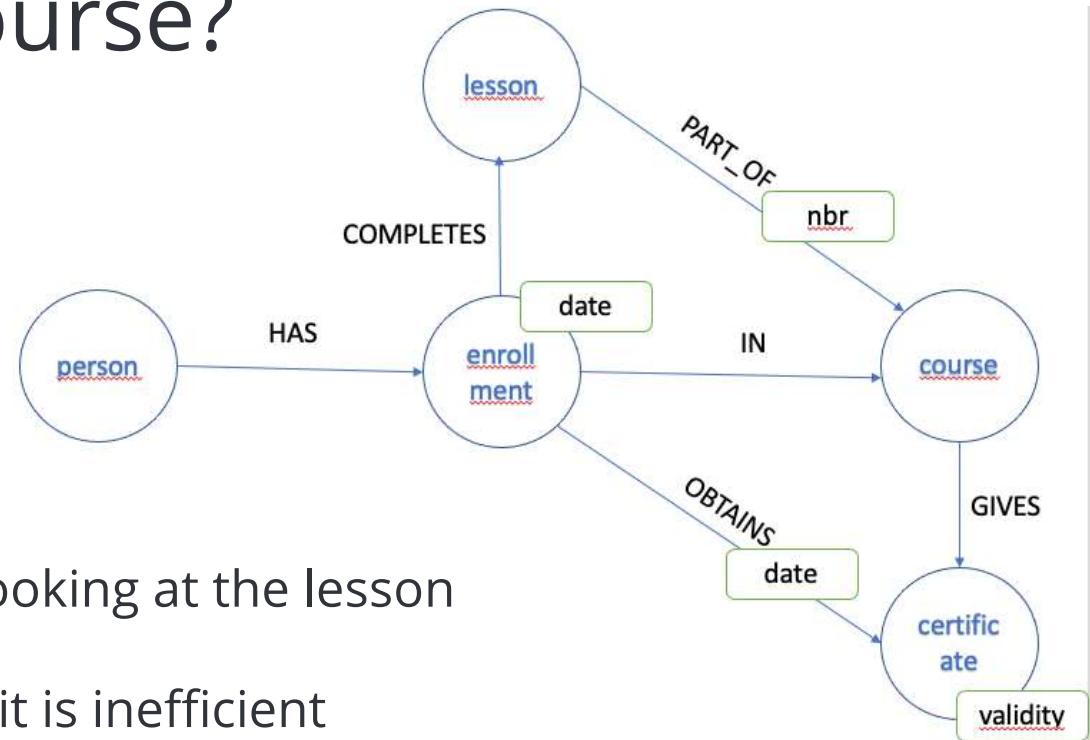


This intermediate node allows us to keep track of students' progress through different concurrent courses, and to differentiate multiple subsequent passes through the same course.

Graph Modelling – List entities = labels for homogeneous sets of nodes

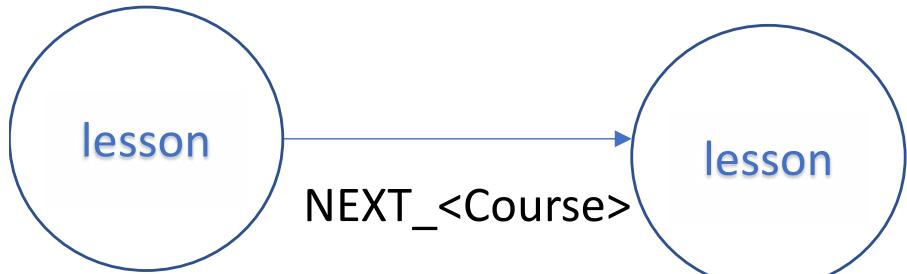


How to manage the order of lessons in a course?

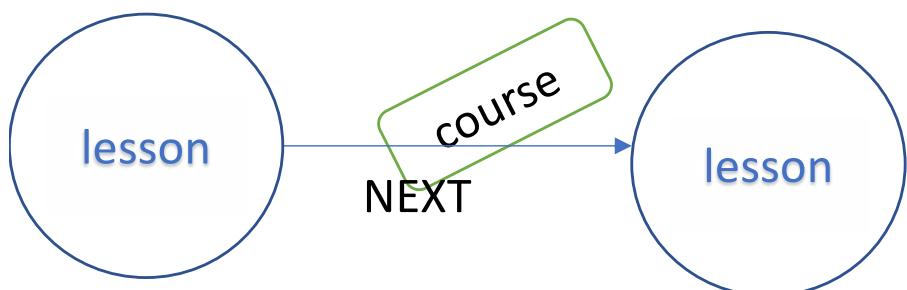


- The order can be reconstructed by looking at the lesson numbering
- But it is a property of a relationship, it is inefficient
 - E.g., given a lesson L PART_OF a course C with number n, to find the next lesson (frequent request) we need to look in all the relationships with type PART_OF, those for course C with the minimum number greater than n and find the corresponding lesson L_{next}
- To make this operation more efficient we can explicit model the next relationship between lessons inside a course
 - This can in turn be obtained through two different modelling

How to manage the order of lessons in a course?

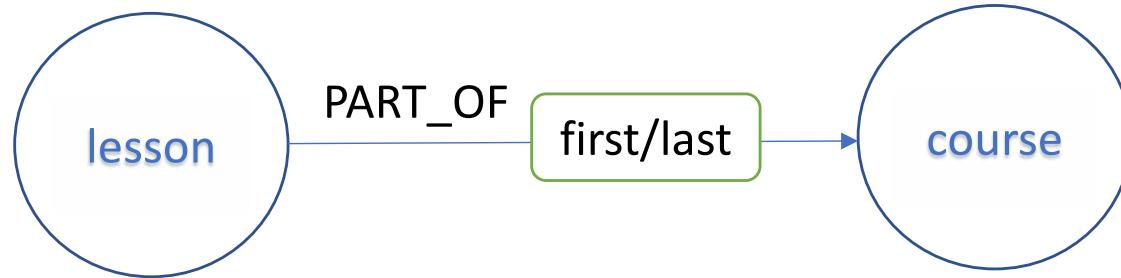


- Better for traversing from one lesson to the next one inside a given course
- A traversal to find the lessons that follows one lesson in any course is less efficient (and more difficult to express)
 - Still reasonable since a lesson likely is not part of too many courses
 - Query rewriting needed if we add a new course ...

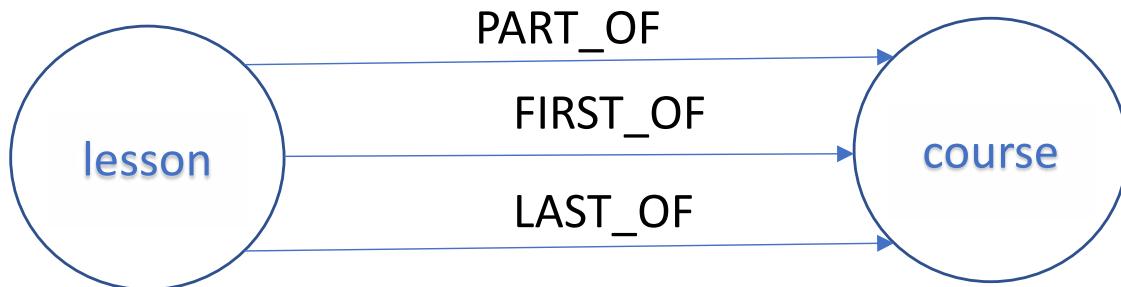


- Better for traversing from one lesson to the ones that follows it in any course
- Worse for traversing from one lesson to the next one inside a given course

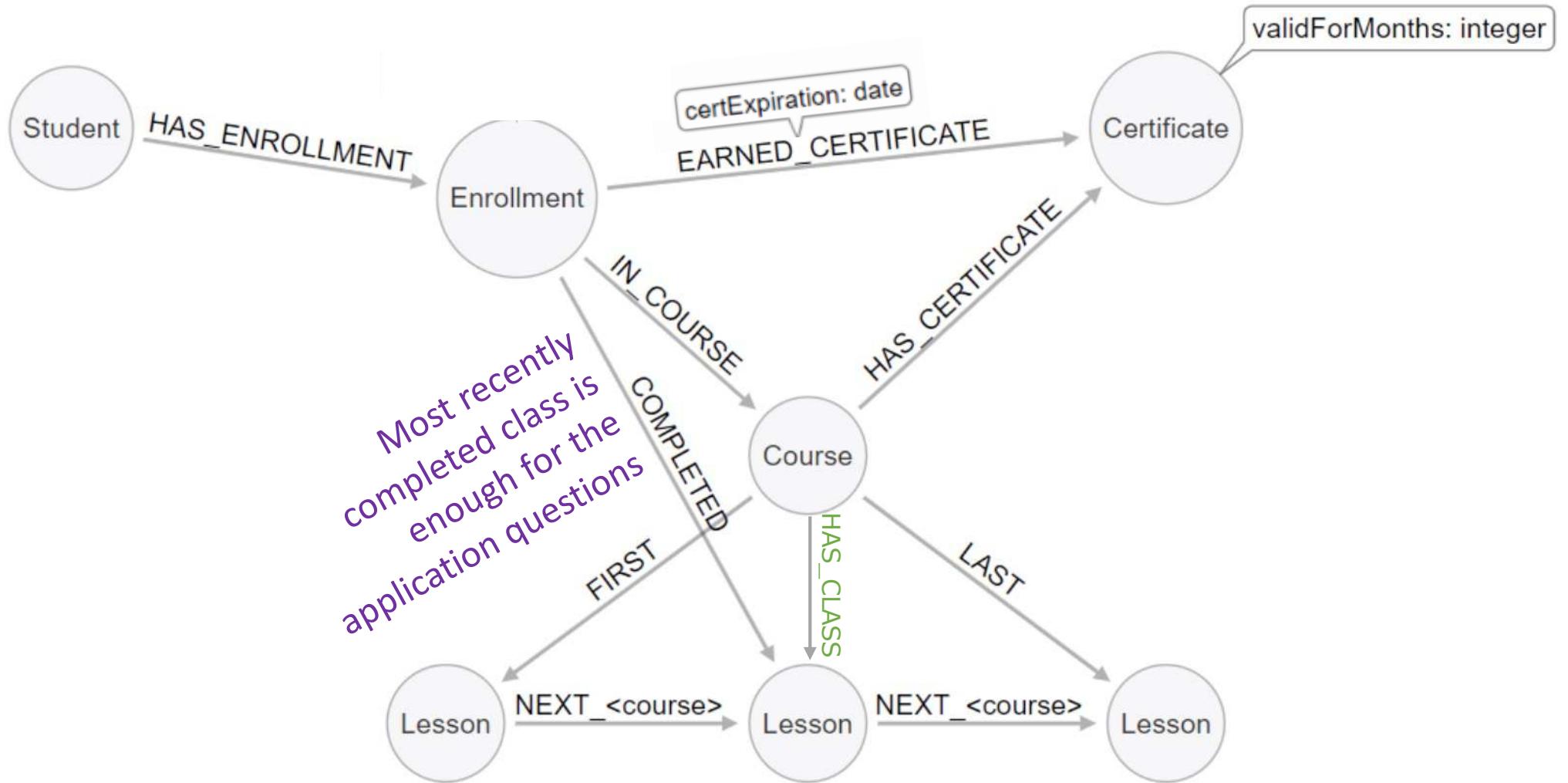
How to model the first (and last) lessons in a course?



- It works, but still we need to access a property of a relationship to solve a frequent request
 - better with last as boolean rather than with nbr (require to check whether nbr is max(nbr) for a course)

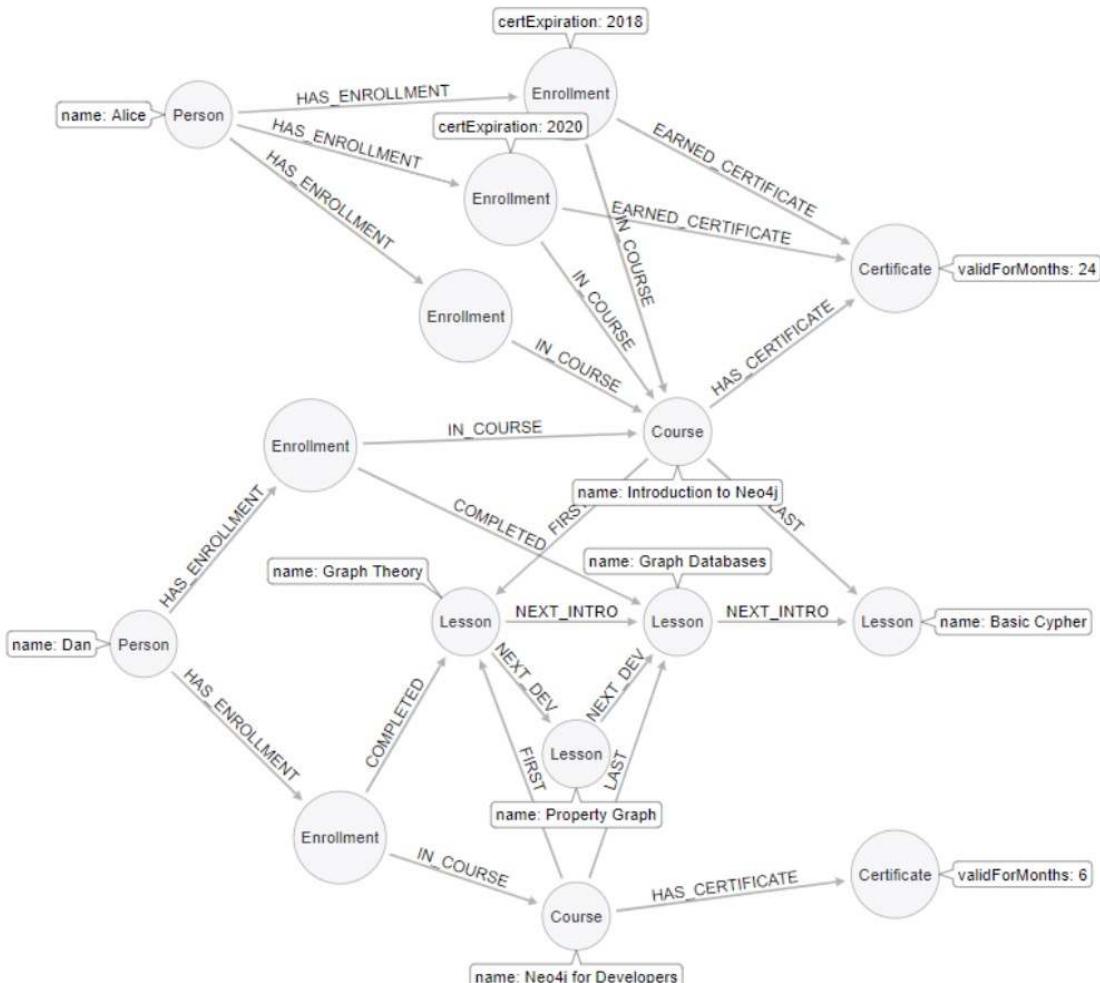


- More efficient to find first and last lesson in a course (frequent request)
- Some redundancy
 - Impact on space
 - Impact on insertion/update time



- No need to add it if we know we'll never ask for all the lessons of a course, but just traverse them from the first to the next one till the last one

Graph Modelling – Test questions against model



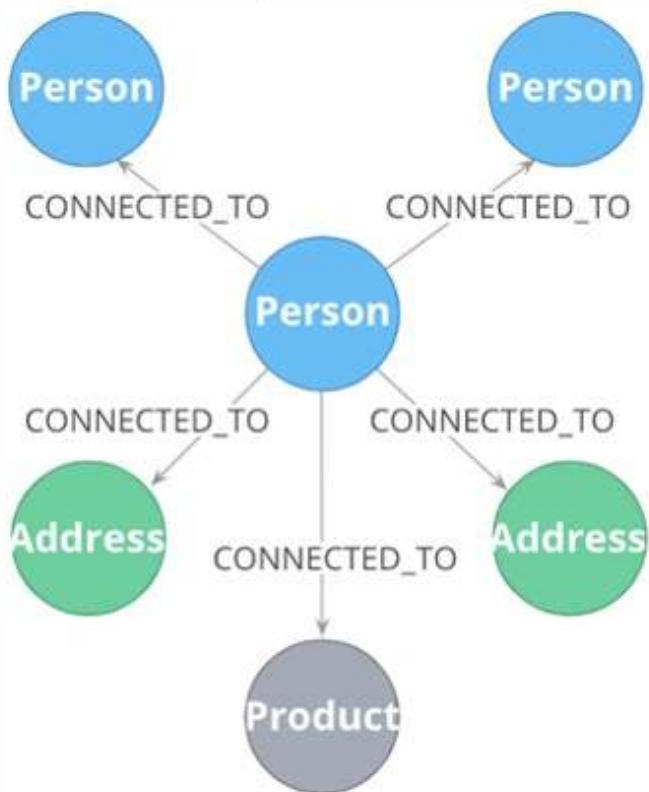
1. Which lesson(s) is Dan currently working on?
2. What are Alice's current certifications?
3. Which lessons are in the **Neo4j for Developers** course?
4. What is the last lesson in the **Introduction to Neo4j** course?
5. Which lesson follows **Graph Theory** in the **Neo4j for Developers** course?
6. Who has completed **Introduction to Neo4j**?

- Choose your favourite modelling
- Construct the graph containing the sample data
- Formulate the queries
- Check whether they are reasonably fast to evaluate

Graph Data Modelling

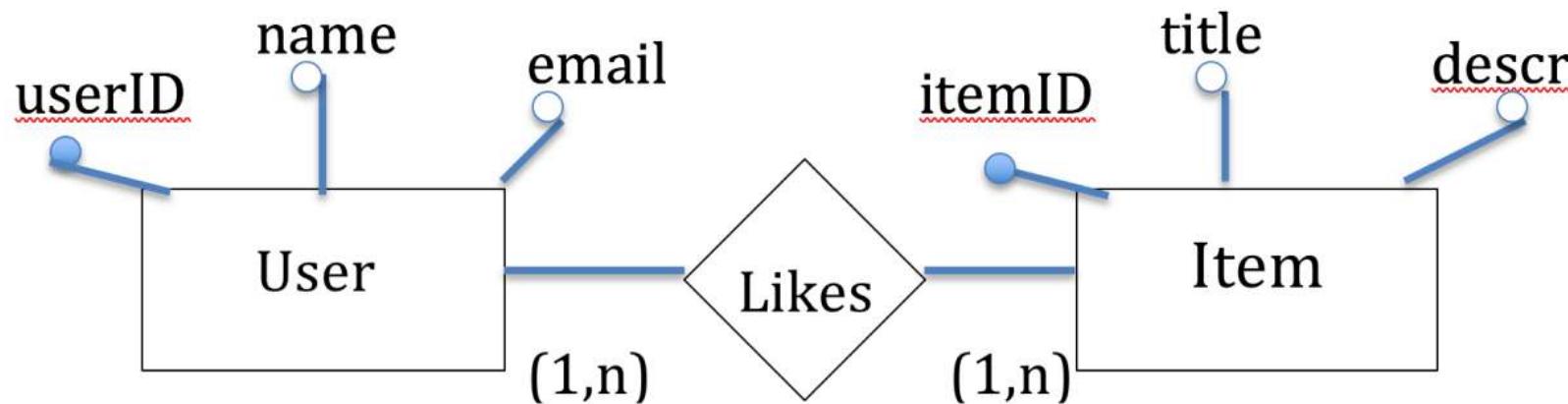
Wooclap

which problem (if any) do you see with this graph?



- The type of the relationship is too generic and thus useless
- Different kinds (living at, being friends, ...) of relationships are modelled in the same way
- This also means that the relationship relates heterogeneous pairs of nodes, this is not per se a problem, if the link represented by it is semantically the same

is a single type for relationships a problem?



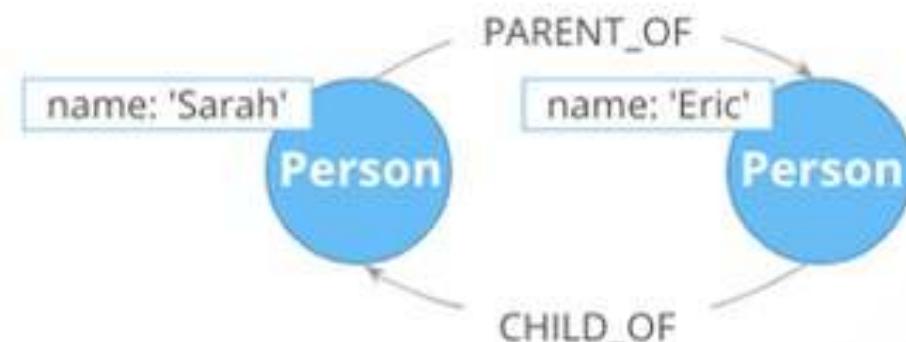
- this conceptual schema results in a graph with a single type for relationships, but the link represented by all of them is semantically the same (*liking*)

which problem (if any) do you see with this graph?



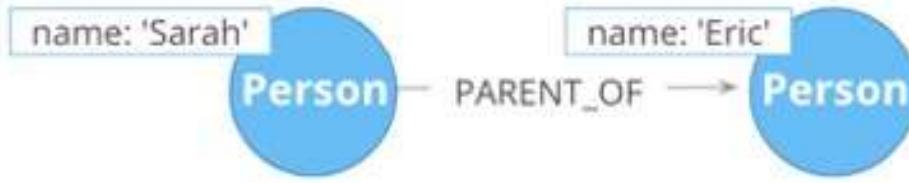
- The type of the relationship is too specific and refers to the value of a property of a connected node («peter»)
- This makes traversal over links of a semantically identical kind referring to another person (e.g., PAUL_WORKS_WITH) difficult
- Peter-centric modelling of relationships

which problem (if any) do you see with this graph?

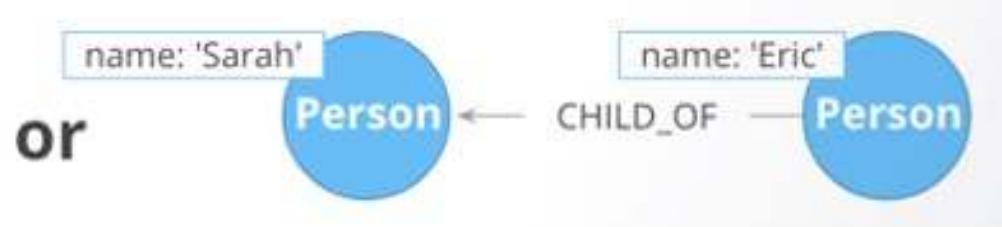


- The two relationships provide the same information
- This redundancy has a cost and provides no advantage
- The parent_of/child_of relations are one the inverse of the other

- One of these alternative graphs is better

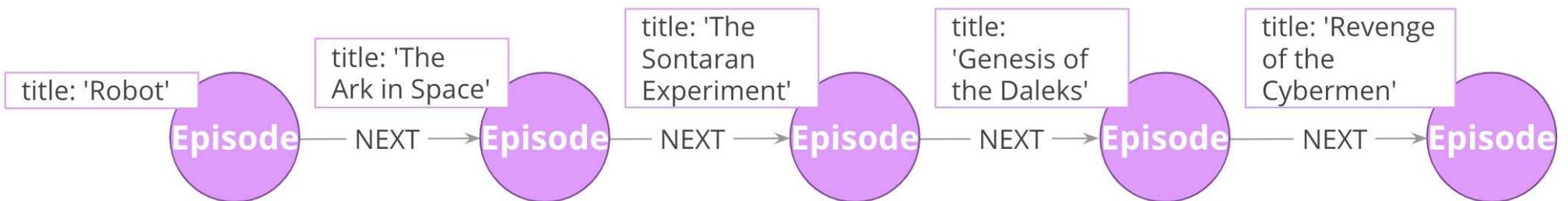


or

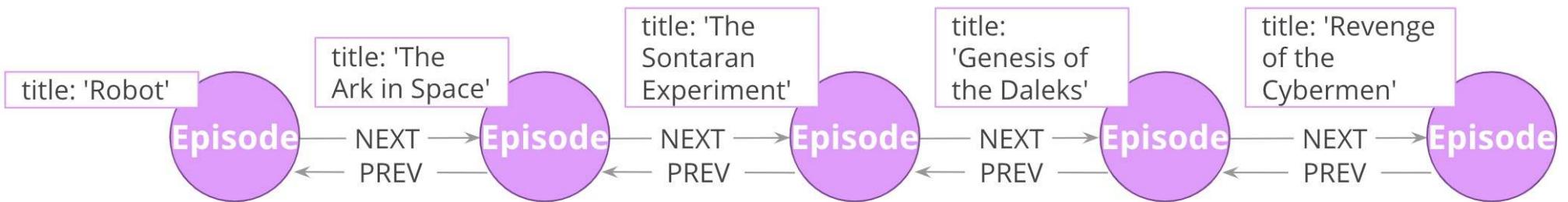


inverse relationships - a similar case

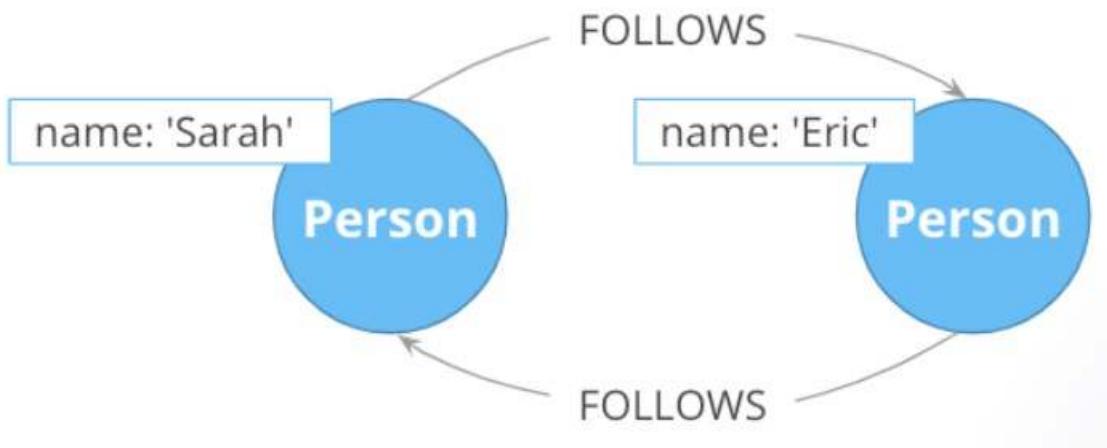
Episodes of the Dr.Who series:



Do NOT do this (doubly-linked list):

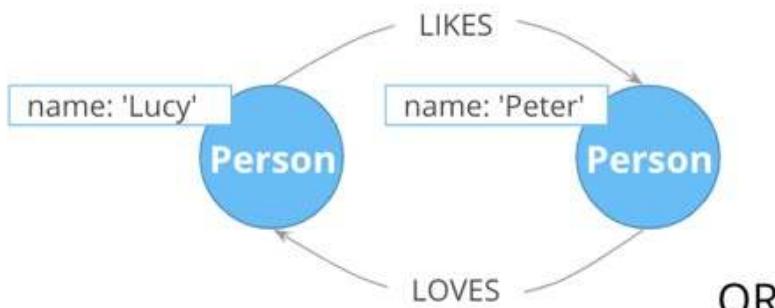


which problem (if any) do you see with this graph?

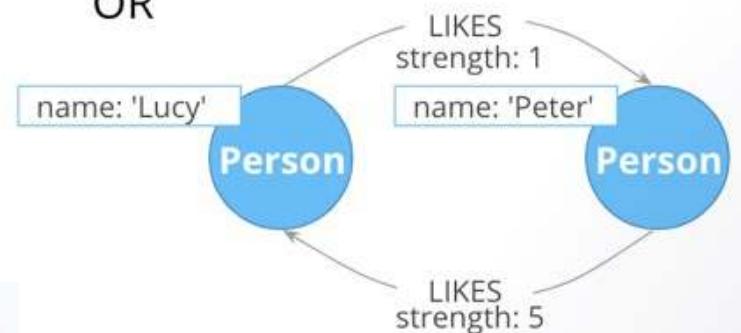


- The graph refers to a case like the ones in twitter or instagram or tiktok in which the fact that Sarah follows Eric guarantees that Eric is followed by Sarah but Eric does not necessarily follow Sarah back
- The relationship is not symmetric (unlike for instance the facebook friendship relationship)
- Thus, both the edges are needed here, since they represent different information

which of the two alternatives is better if we want to find all the strong relationships and discard the weaker ones?



OR



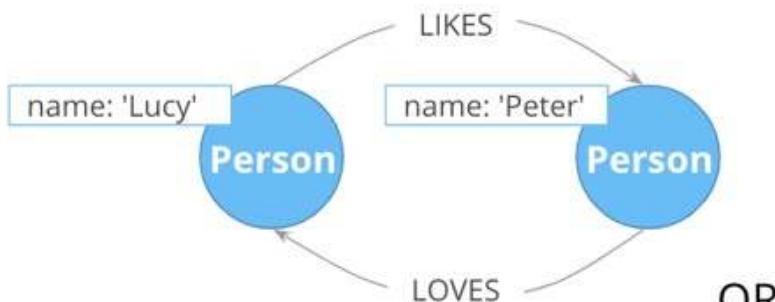
left one, since filtering on the type is more efficient

Traversal will not involve
any gather-and-inspect

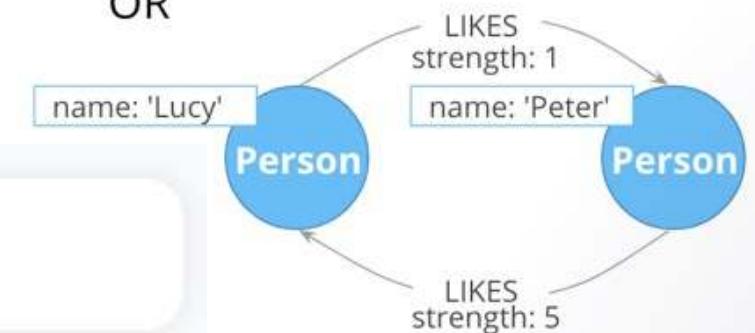
no difference

right one, since filtering on integers is more efficient than on string

which of the two alternatives is better if we want to find all the relationships (both weak and strong), ranked by strength (stronger first)?



OR



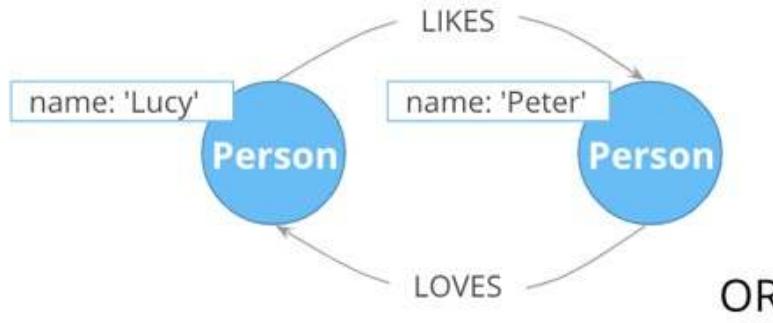
left one, since sorting on the type is more efficient

no difference

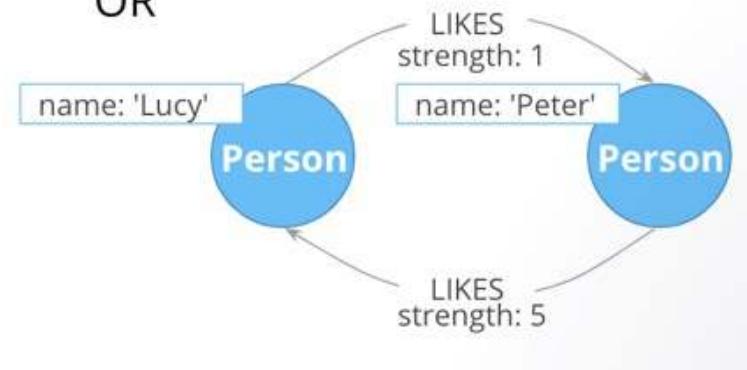
right one, since we want all the relationships anyway and sorting on integers is more efficient

it is also simpler to express in Cypher, and there will not be any gather-and-inspect discards because we want everything anyway

Any other difference among the two?



OR



- The right one is more flexible if we consider all the values for strength in the range 1..5 rather than just 1 and 5
- It allows to represent a wider set of nuances

for which of the following cases do you see an issue with the complex/multivalued address property?

```
firstName: 'Patrick'  
lastName: 'Scott'  
age: 34  
homeAddress: ['Flat 3B', '83  
Landor St.', 'Axebridge', 'DF3 OAS']  
workAddress: ['Acme Ltd.', '12  
Crick St.', 'Balton', 'DG4 9CD']
```

Person

address as an anchor (filtering that select the start node)

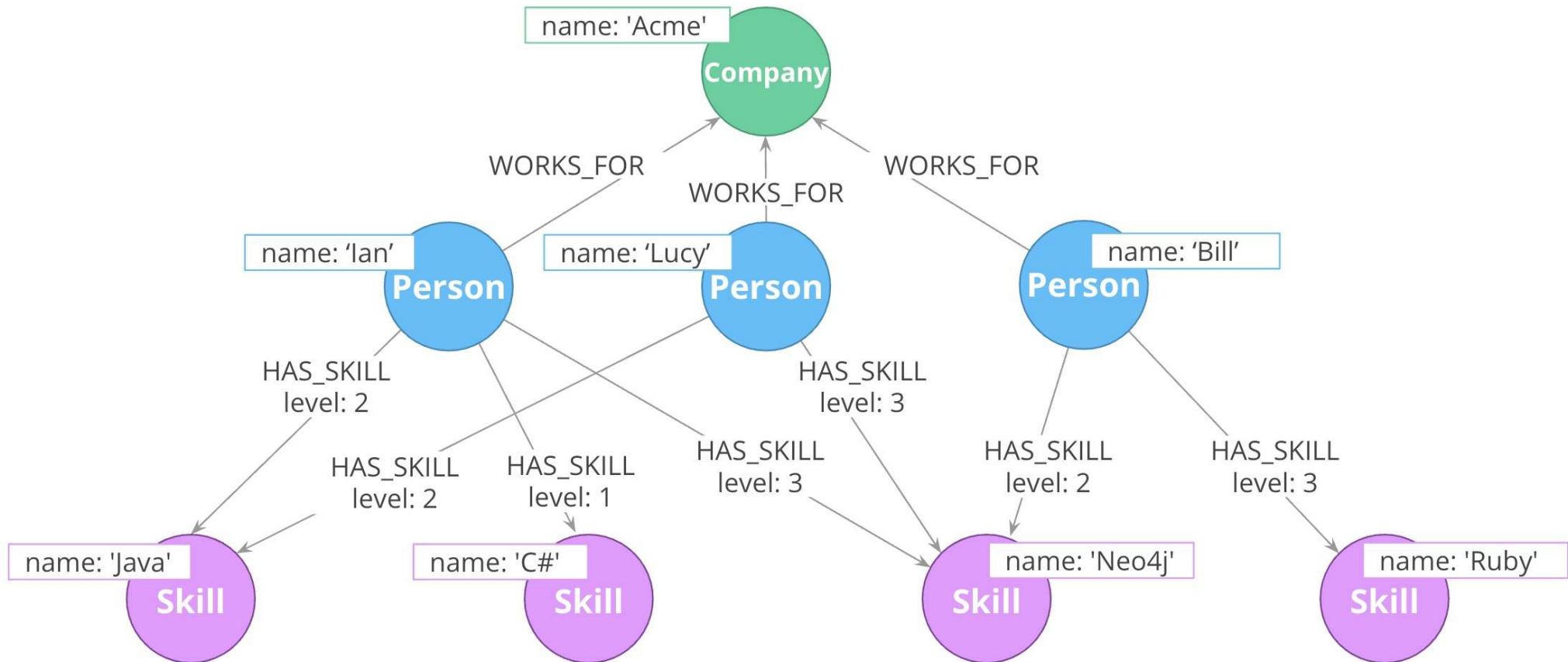
address as an output value

address as a decorator (not used for filtering, not returned as output)

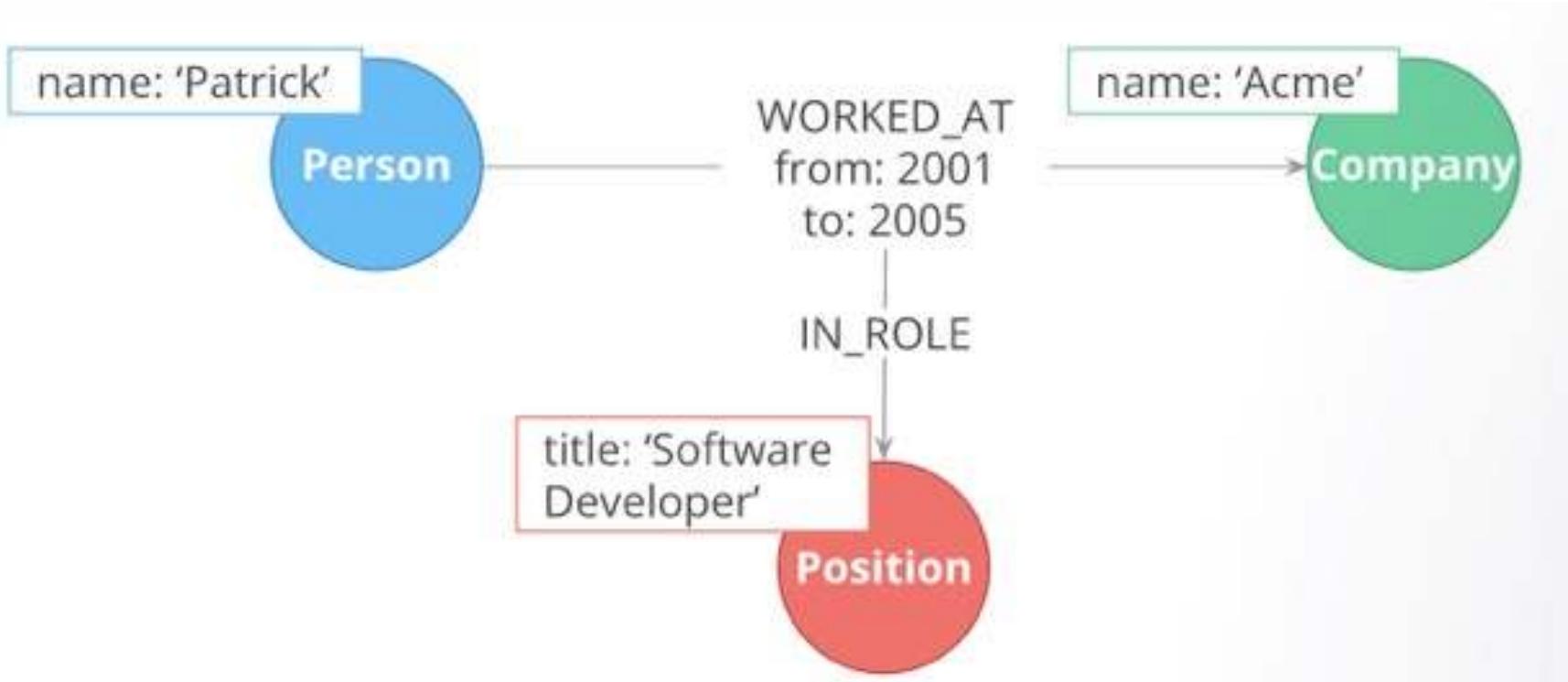
properties used
for anchoring
should be as
simple as
possible

Graph Traversals – Access Costs

1. Anchor node label, indexed anchor node properties
2. Relationship types
3. Non-indexed anchor node properties
4. Downstream node labels
5. Relationship properties, downstream node properties



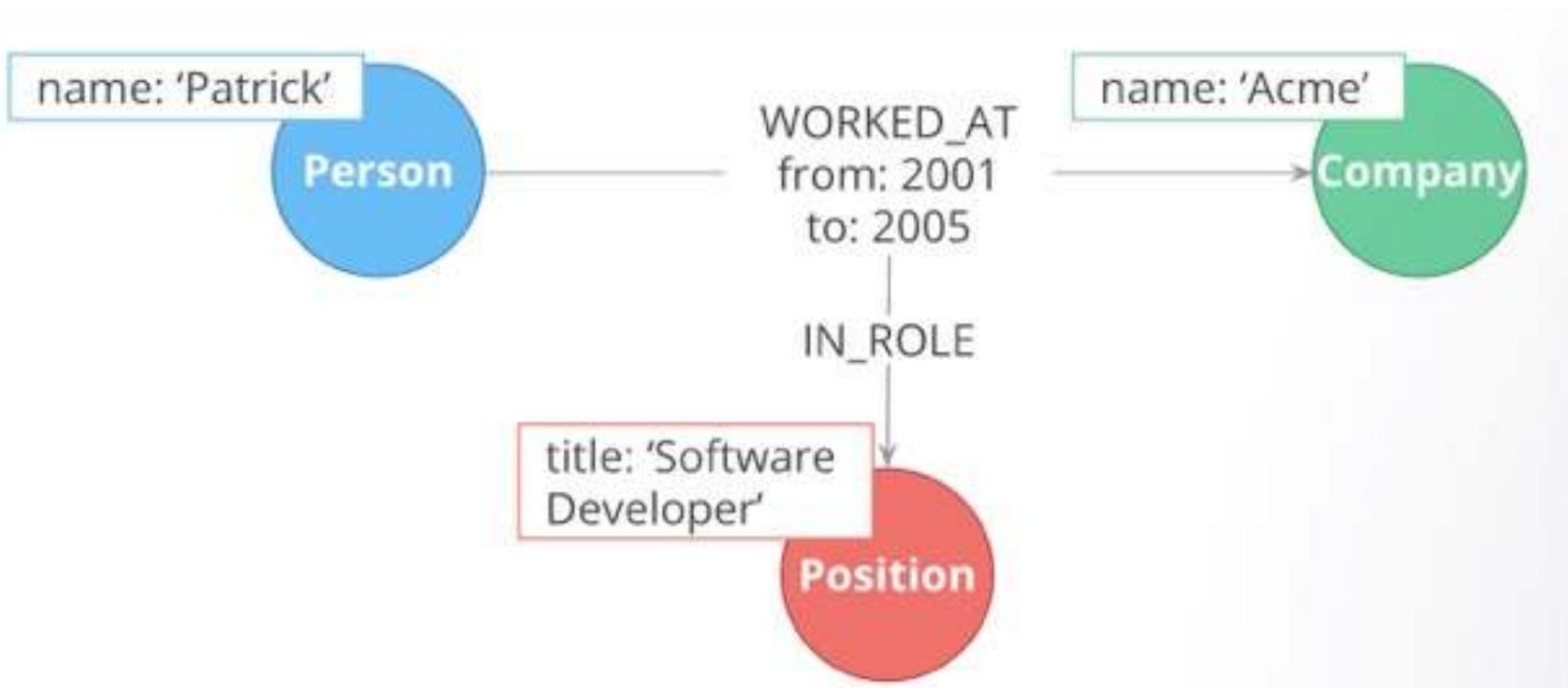
which problem (if any) do you see here?



This is not a graph!

Relationships cannot connect three nodes (an hyperedge would be needed, not supported by the property graph model)

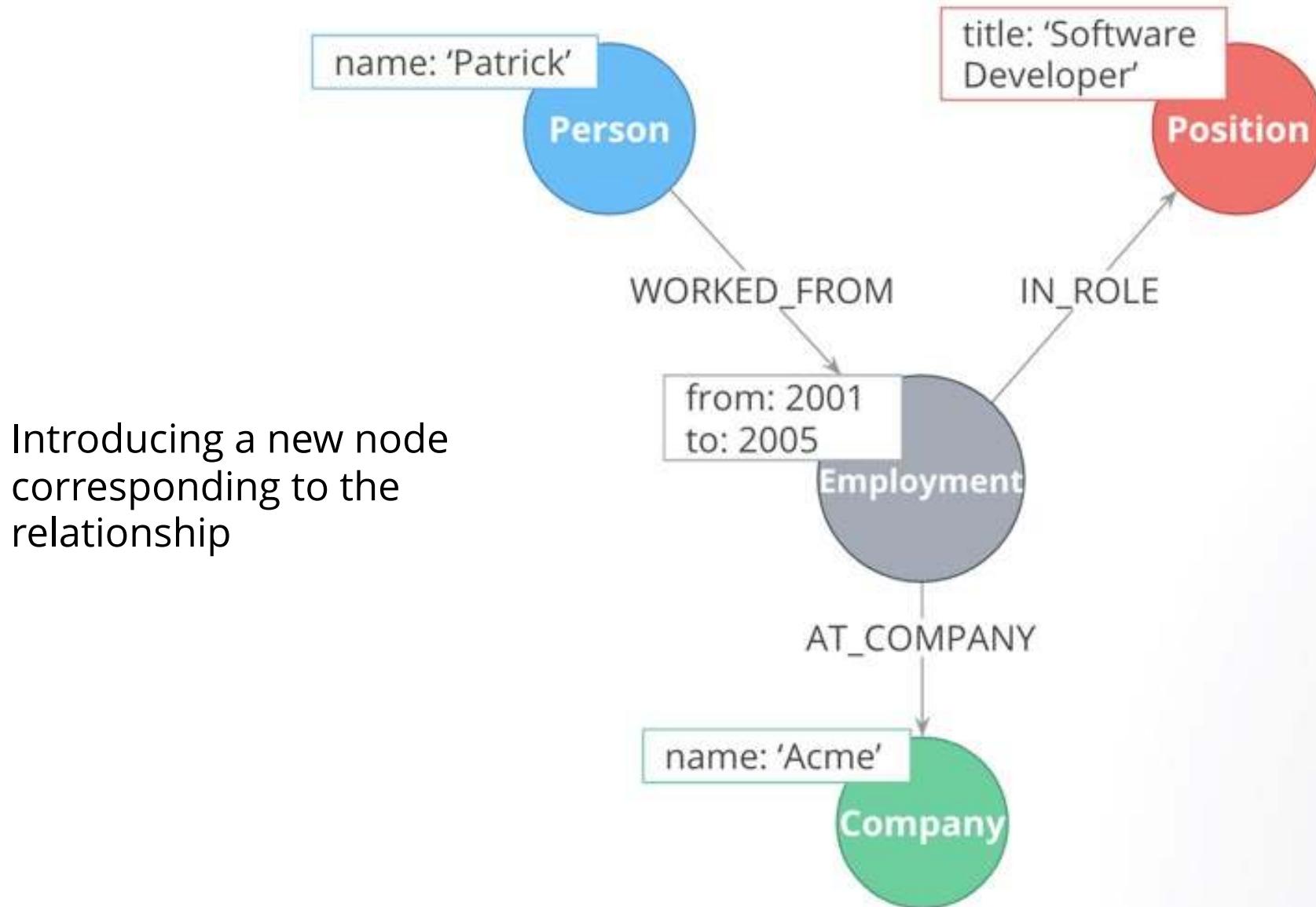
how can we represent it? (1)

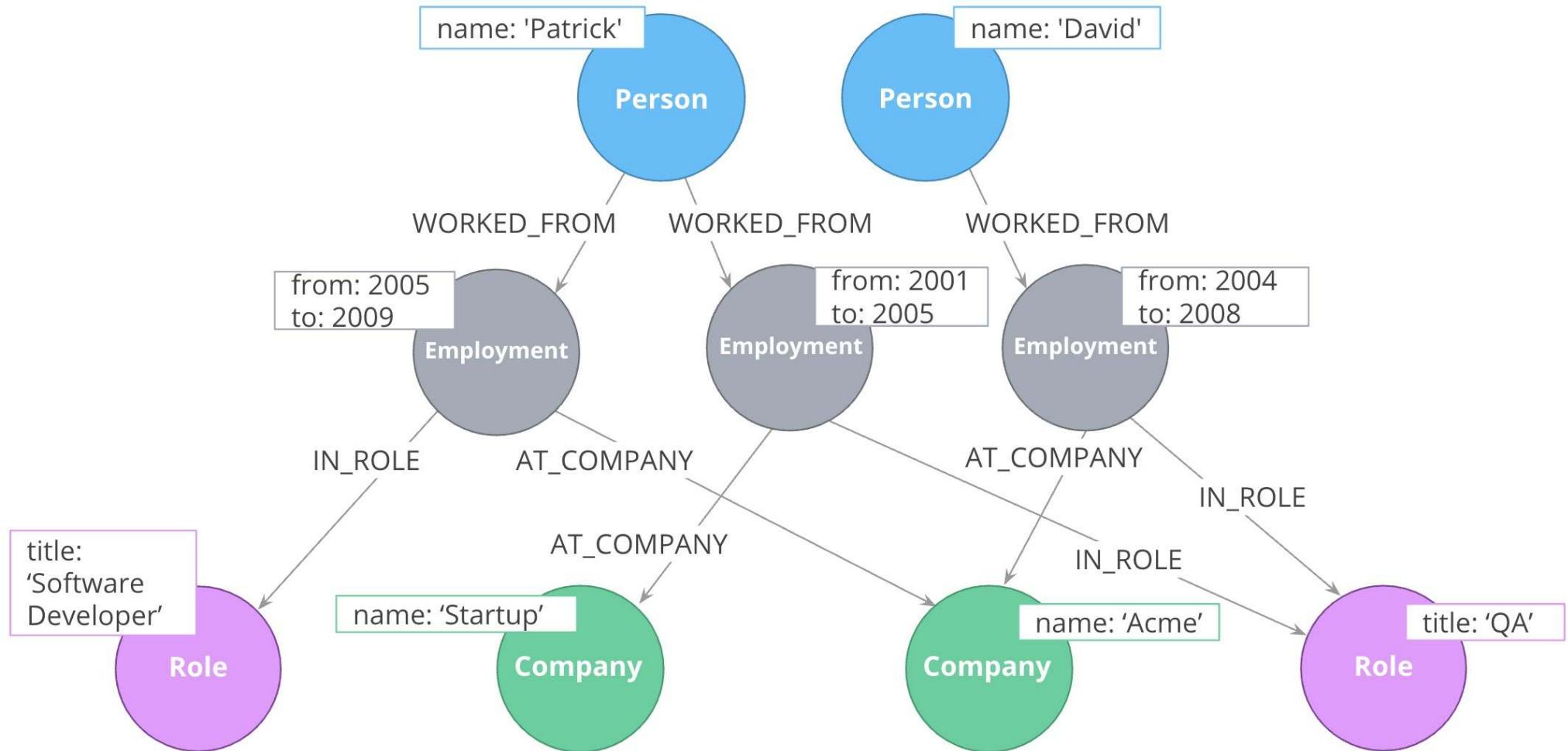


We could simply give up the position node and model roles as properties of the relationship

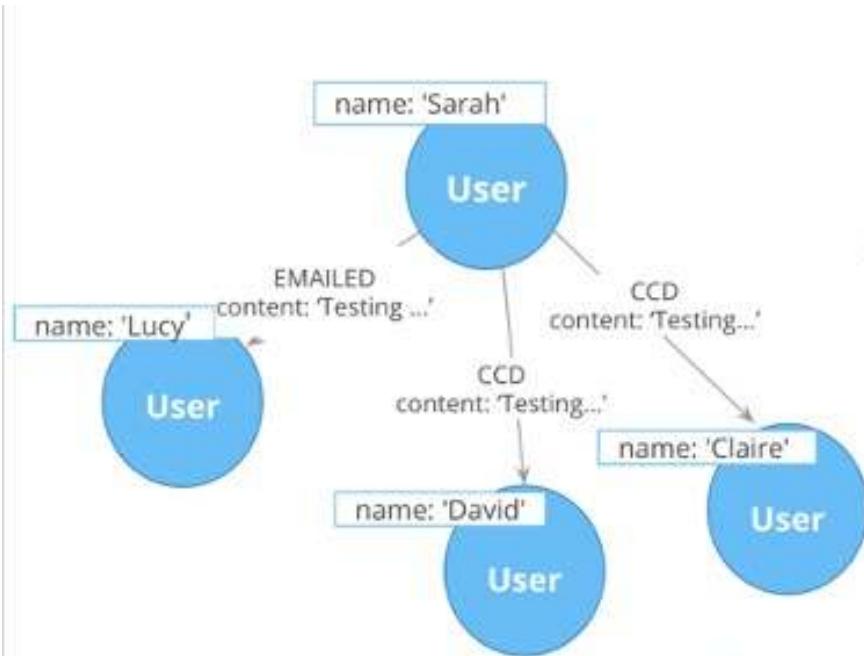
This makes queries like «find the employees that worked as Software Developer» or «find the companies with Software Engineering positions» less efficient

how can we represent it [if we want to keep position nodes and capture a ternary relationship]?



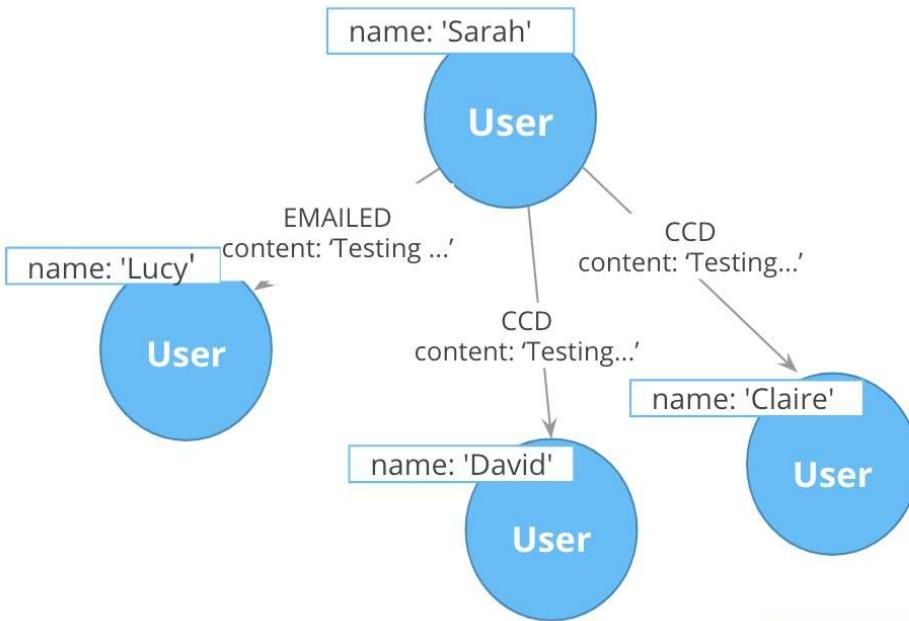


which problem (if any) do you see with this graph?



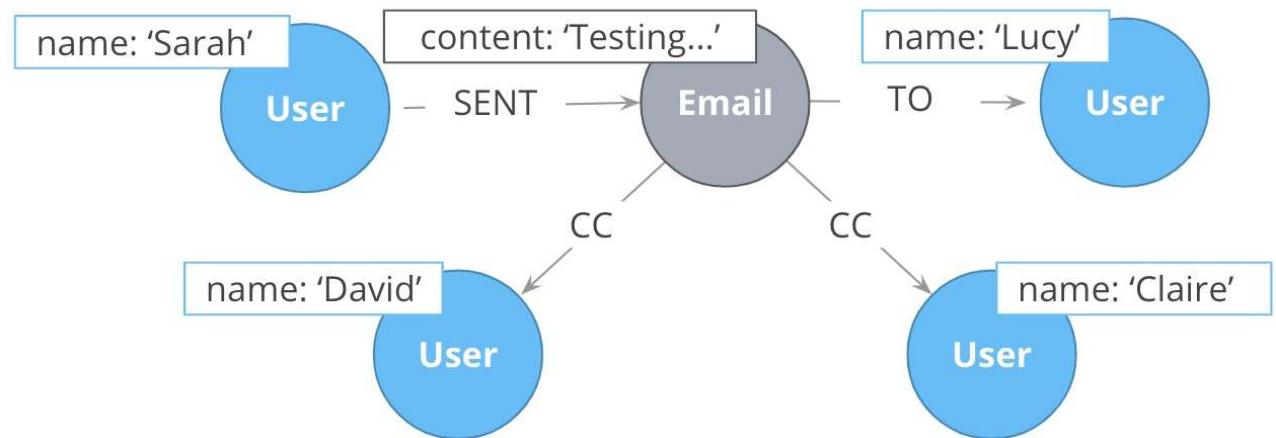
- Duplicate email text
- We cannot distinguish persons that are copied in the same email without looking at the content property of relationships
- The content of the email itself could not be enough to determine whether two emails are the same
- User nodes likely become dense since they will be connected to all the recipients and copied users of all the emails they send

a better modelling

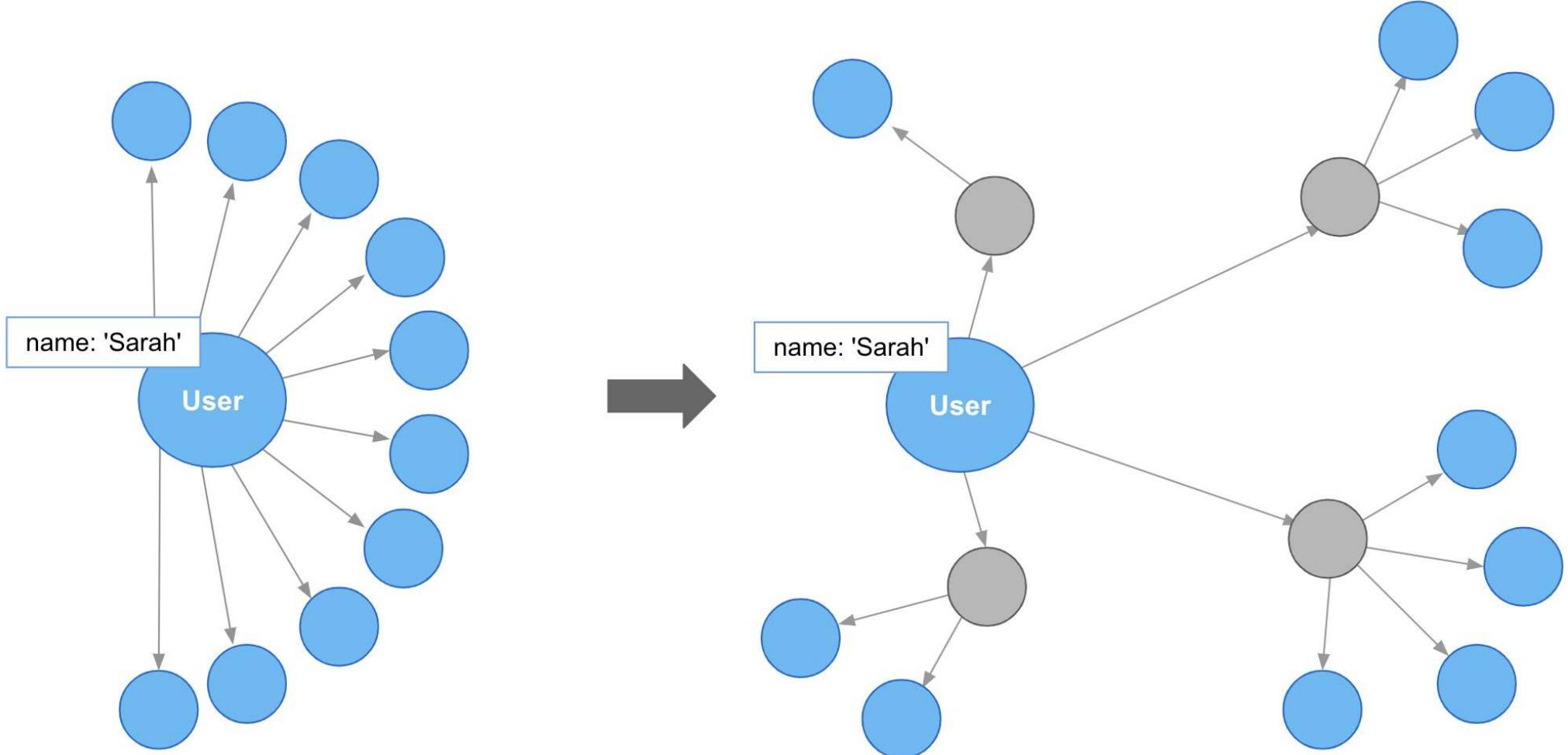


By introducing the email node we can fix the issues

Now it is much simpler and more efficient to determine
All recipients of an email about
«ADM project assignment»



a better modelling – node density



With the email node, each user is connected to the emails she wrote

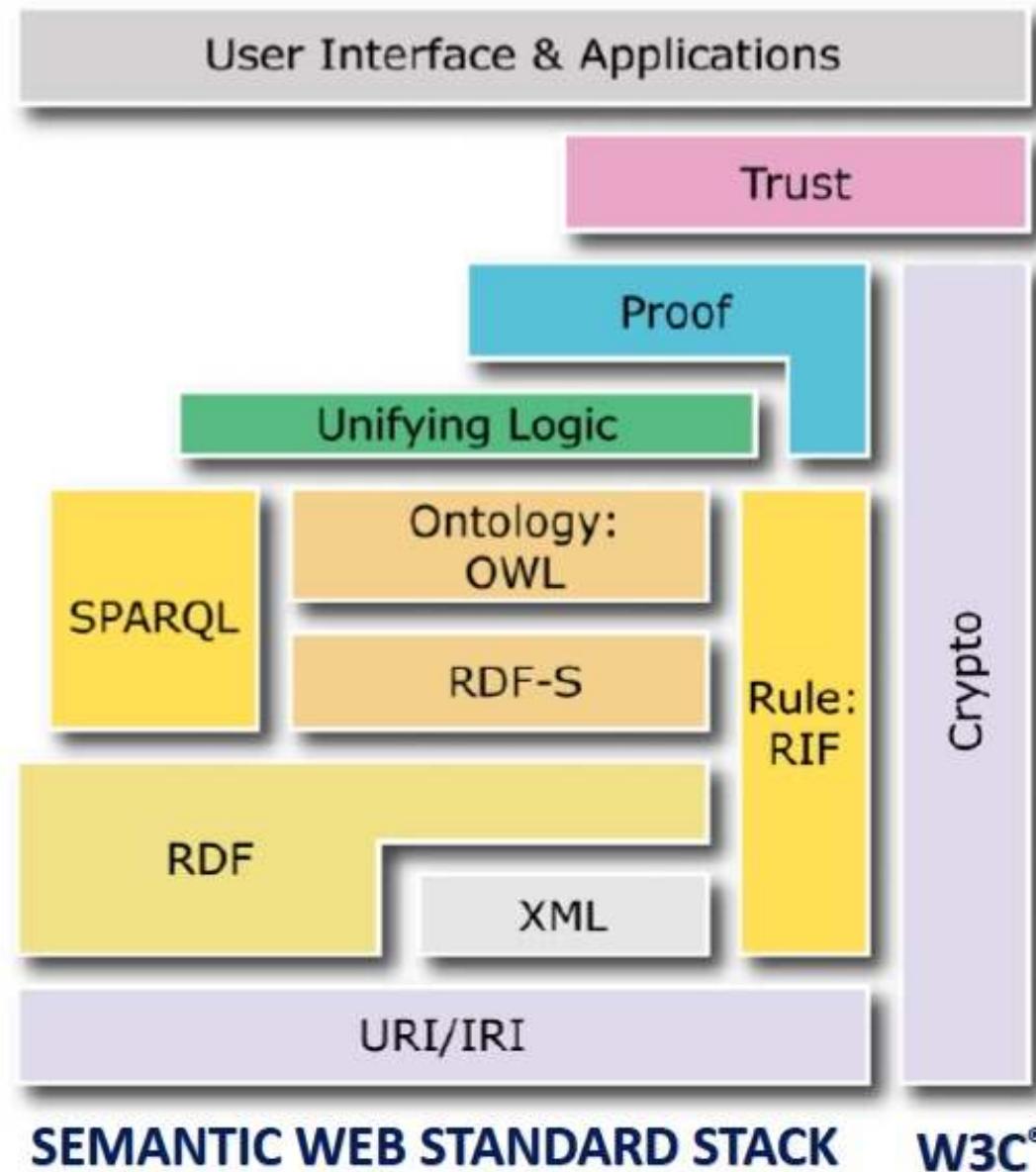
Resource Description Framework

The Semantic Web

- The Semantic Web is a Web in which the **resources** (things) are semantically described, through the usage of an **ontology**
- A resource is anything that can be referred to by a **URI** (Uniform Resource Identifiers)
 - a web page, identified by an URL
 - a fragment of an XML document, identified by an element node of the document or an XPath expression
 - a web service
 - a thing, an object, a property, etc.
- Examples
 - <http://www.example.org/file.html>
 - <http://www.example.org/file.html#home>
 - [http://www.example.org/file2.xml#xpath\(//q\[@a=b\]\)](http://www.example.org/file2.xml#xpath(//q[@a=b]))
 - <http://www.example.org/form?a=b&c=d>

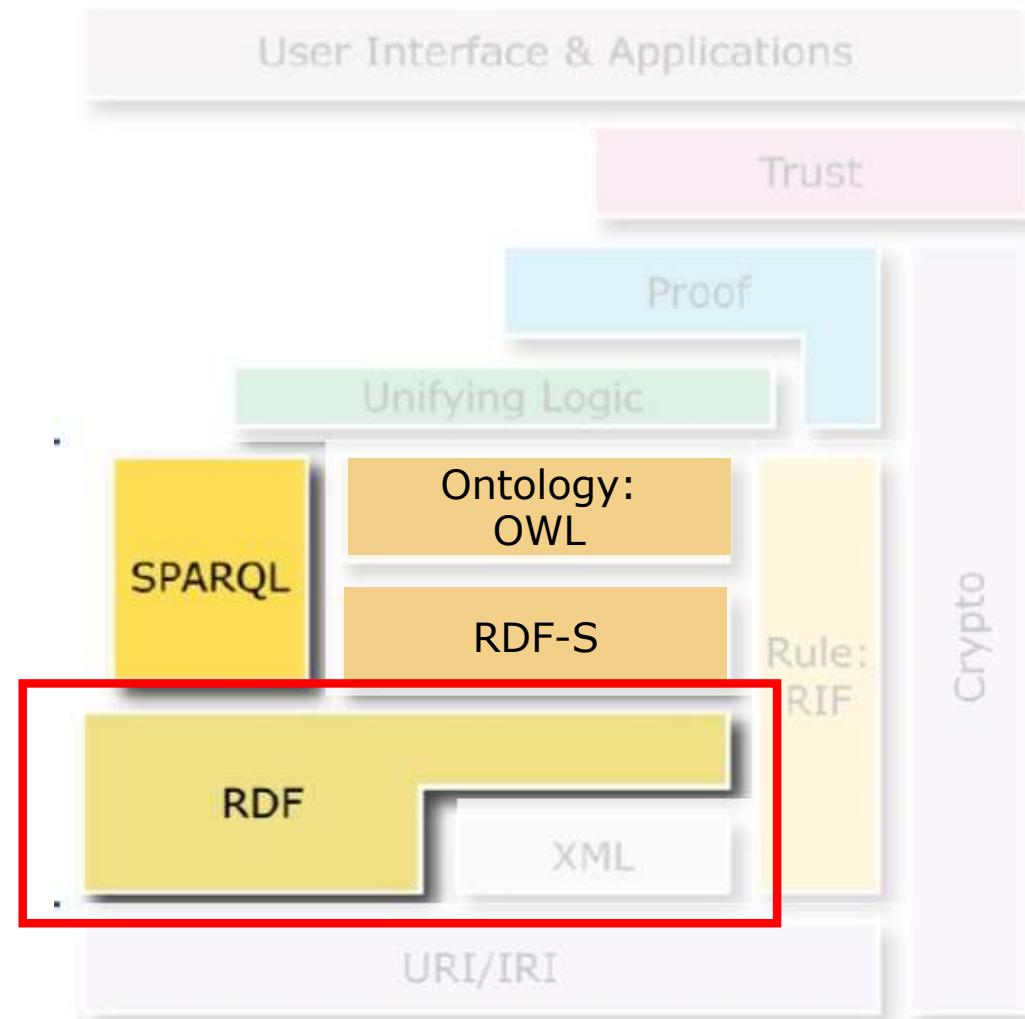


The Semantic Web Standard Stack



The Semantic Web Standard Stack

- RDF: a very simple ontology language
- RDFS: Schema for RDF
 - ▶ Can be used to define richer ontologies
- OWL: a much richer ontology language



Resource Description Framework

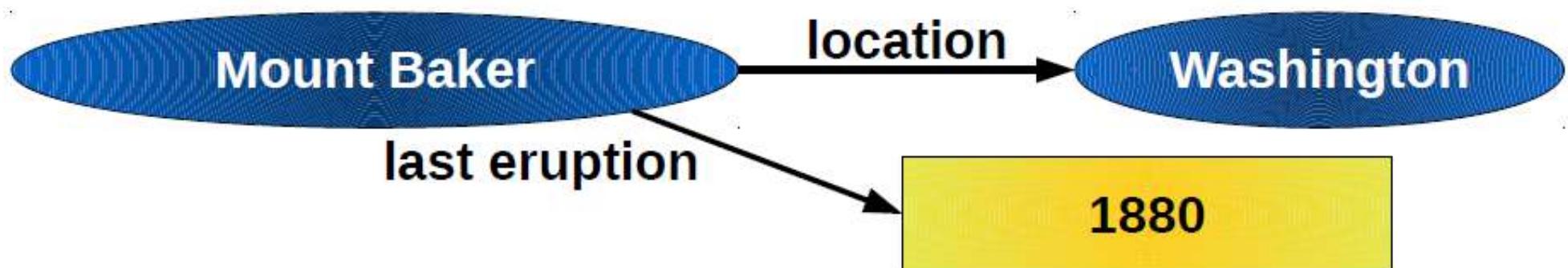
- RDF stands for
 - **Resource**: pages, dogs, ideas...everything that can have a URI
 - **Description**: attributes, features, and relations of the resources
 - **Framework**: model, languages and syntaxes for these descriptions
- A W3C standard since 2004
- Description of arbitrary things
- A very simple ontology language
- Models ontology instances, ontology concepts, ontology relations
- RDF is the data model for the Semantic Web
 - provides a simple language for describing annotations about Web resources identified by URIs
 - these are facts

RDF Data Model

- A schema-less data model that features
 - unambiguous identifiers and
 - named relations between pairs of resources
- Data are represented as a set of triples (**subject**, **predicate**, **object**)
 - subject: a **resource** (identified by an **URI**)
 - predicate: a **resource**, representing a **property** (identified by an **URI**)
 - object: a **resource** (identified by an **URI**) or a **literal** (a constant value with some annotation)
- when the object is a literal, the triple expresses that a given subject has a given value for a given property
- RDF triples can be represented as a graph (RDF graph)

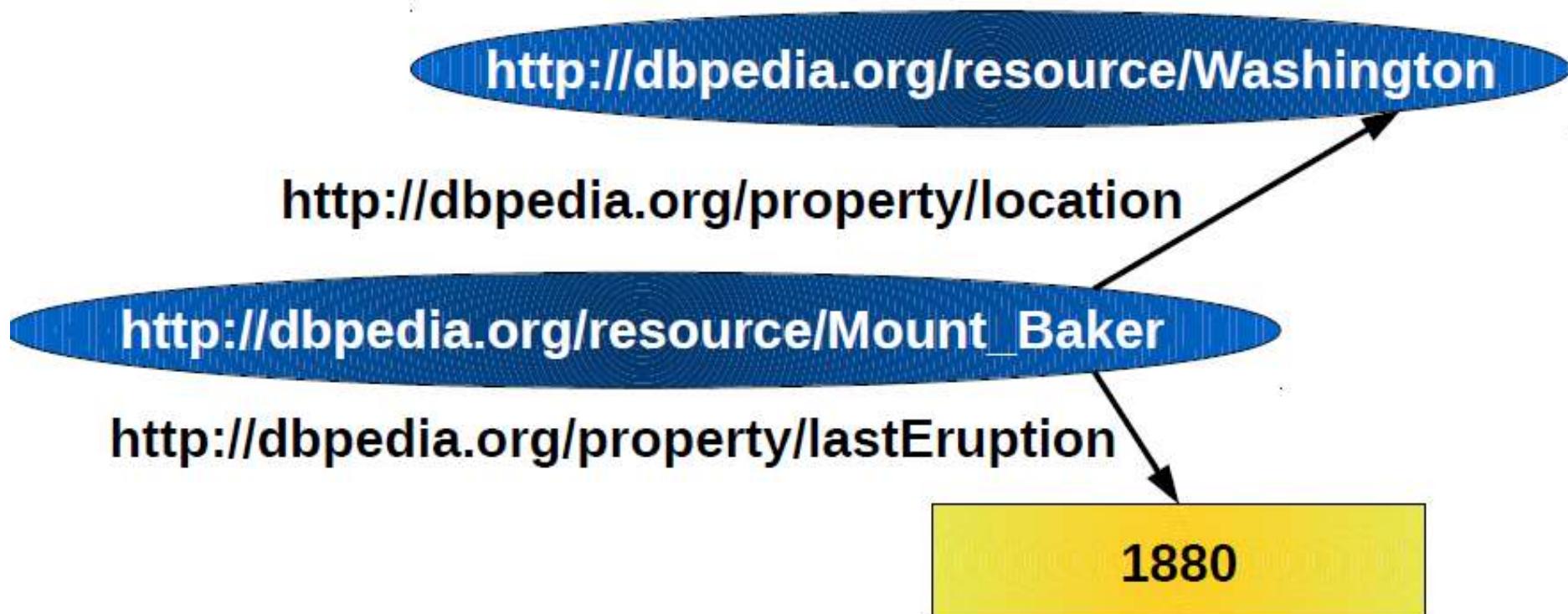
Example

- **Example:**
 - (Mount Baker , last eruption , 1880)
 - (Mount Baker , location , Washington)



RDF graph with URIs

- (http://dbpedia.org/resource/Mount_Baker,
<http://dbpedia.org/property/lastEruption>, 1880)
- (http://dbpedia.org/resource/Mount_Baker,
<http://dbpedia.org/property/location>,
<http://dbpedia.org/resource/Washington>)



RDF Data model: URI recap

- **URI:** Uniform Resource Identifier
 - a web page, identified by an URL
 - a fragment of an XML document, identified by an element node of the document or an XPath expression
 - a web service
 - a thing, an object, a property
 - ...
- Examples
 - <http://www.example.org/file.html>
 - <http://www.example.org/file.html#home>
 - [//q\[@a=b\]](http://www.example.org/file2.xml#xpath)
 - <http://www.example.org/form?a=b&c=d>
 - <http://dbpedia.org/resource/Berlin>

DBLP Example

Basic building blocks

Basic building blocks

- **Resources**

- denote things
- are identified by a URI
- can have one or multiple types

- **Literals**

- are values like strings or integers
- can only be objects, not subjects or predicates (graph view: they can only have incoming edges)
- can have a datatype or a language tag (but not both)

- **Properties (Predicates)**

- Link resources to other resources and to literals

Resource versus Literal

- A literal is an atomic value
 - can only be object
 - i.e., a literal terminates always a graph

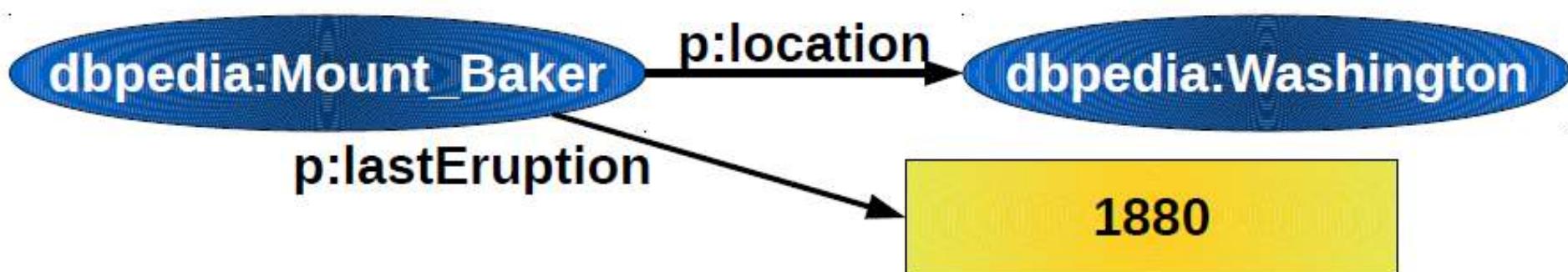


- A resource can be a subject itself



Example with namespaces

- **Using**
 - *dbpedia* for prefix *http://dbpedia.org/resource/*
 - *p* for prefix *http://dbpedia.org/property/*
- **we have**
 - (dbpedia:Mount_Baker, p:lastEruption, 1880)
 - (dbpedia:Mount_Baker, p:location, dbpedia:Washington)



Some standard namespaces

rdf: A namespace for RDF.

The URI is: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

rdfs: A namespace for RDFS.

The URI is: <http://www.w3.org/2000/01/rdf-schema#>

owl: A namespace for OWL.

The URI is: <http://www.w3.org/2002/07/owl#>

dc: A namespace for the Dublin Core Initiative.

The URI is: <http://dublincore.org/documents/dcmi-namespace/>

foaf: A namespace for FOAF.

The URI is: <http://xmlns.com/foaf/0.1/>.

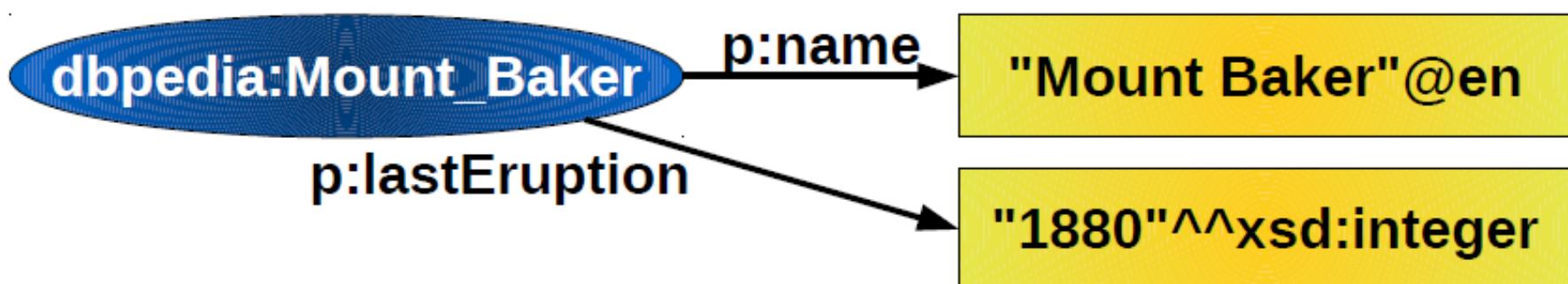
- **Dublin Core** is a popular standard in the field of digital libraries.
- **The Friend of a Friend (FOAF)** initiative aims at creating a “social” Web of machine-readable pages describing people, the links between them and the things they create and do.

Literals

- Literals may occur in the object position of triples
- Represented by strings
- Literal strings interpreted by datatypes
 - Datatype identified by a URI
 - Common to use the XML Schema datatypes
 - No datatype: interpreted as xsd:string
- Untyped literals may have language tags (e.g. @de)

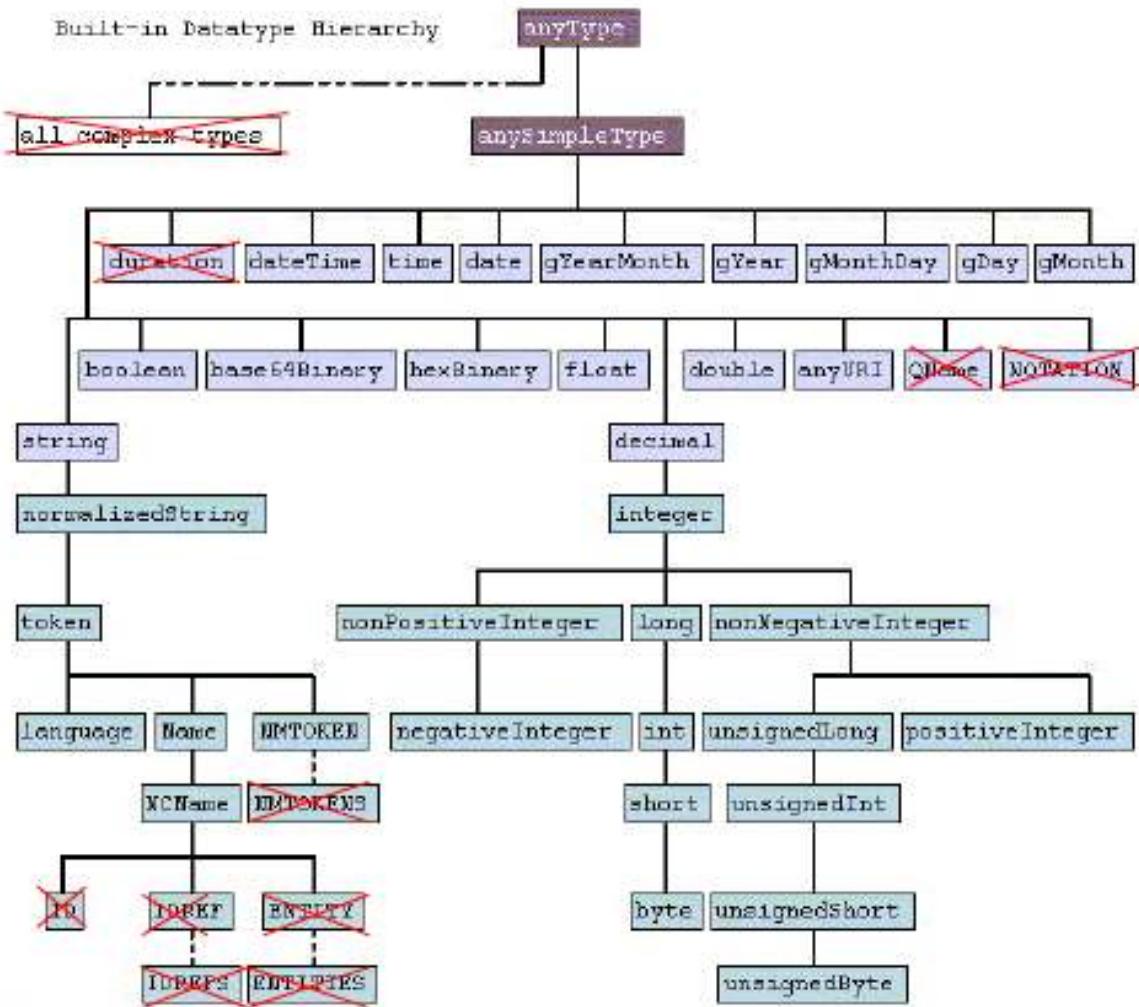
Language codes according to ISO 963

But also ..."Monte Baker"@it



Literals: type

- (Almost) all XML Schema datatypes may be used
- Exception:
 - XML specific types
 - The underspecified type "duration"
 - sequence types



RDF Data Model – Some «schema» information

- In RDF, one can distinguish between individuals (objects) and properties (relationships)
- This is not mandatory but it can be done using two RDF resources:
 - `rdf:type`, which can be used as a property
 - `rdf:Property`, which can be used as a resource
- Still triplets but providing a **very light schema information**
- Example
 - <location `rdf:type rdf:Property:`> the resource location is a property
 - <Mount_Baker `rdf:type Volcano`>; the resource Mount_Baker is an instance of class Volcano

Blank nodes

- Sometimes, you may not precisely know the resource which is involved in some relationship with some other resources
- but you do know that the relationship exists
- two options:
 - create an extra URI, but in this case the resource will be visible on the Web
 - create an “internal” resource, visible only to your set of triples, in terms of a **blank node**

Blank nodes

- A **blank node** (or anonymous resource) is a subject or an object in an RDF triplet or an RDF graph that is not identified by a URI and is not a literal
- A blank node is referred to by a notation `_:p` where `p` is a local name that can be used in triplets (in the context of the same RDF graph) for stating several properties of the corresponding blank node
- Blank nodes require attention when merging
 - blanks nodes with identical nodeIDs in different graphs are different

Example

RDF triples

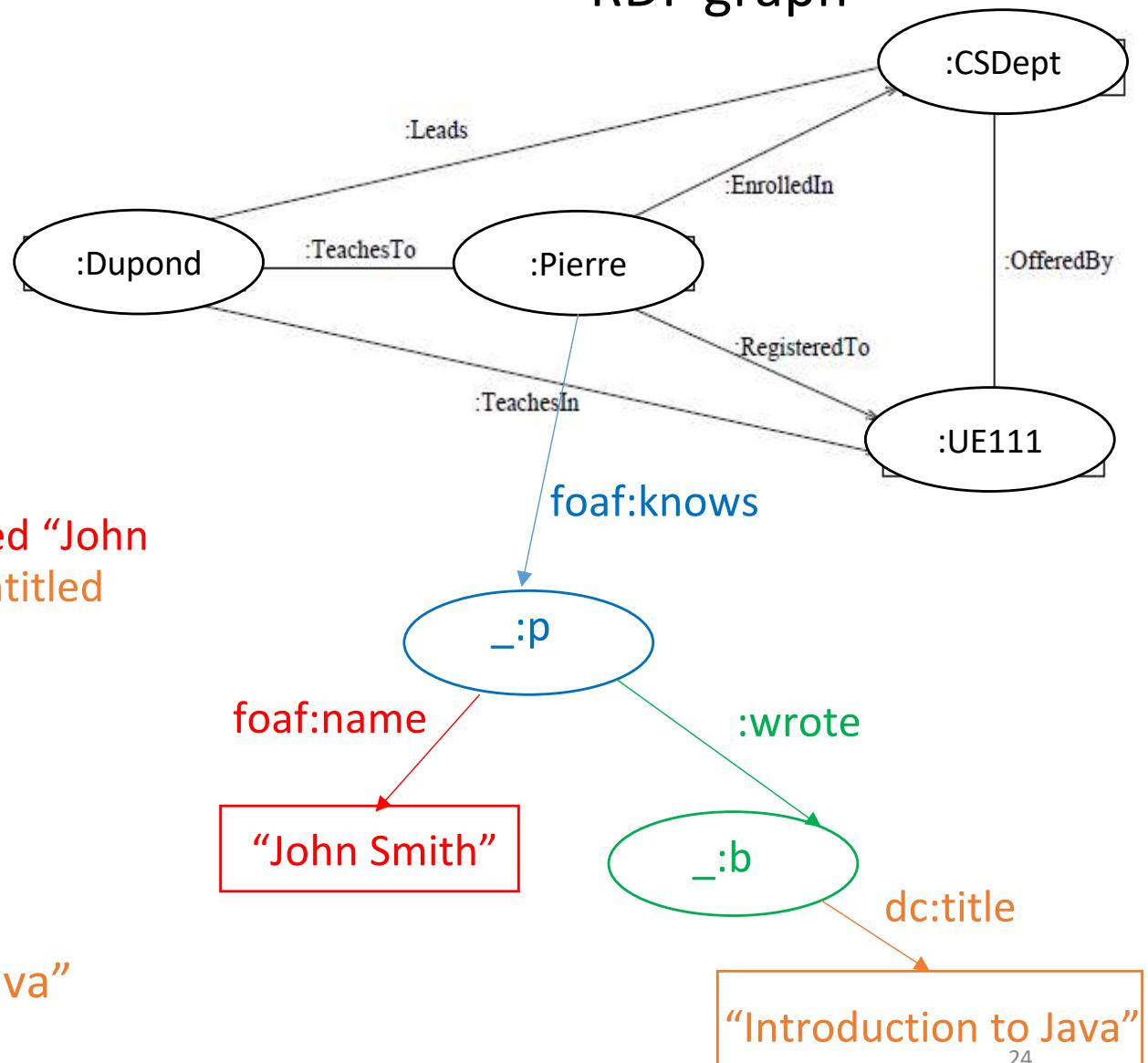
```
< :Dupond :Leads :CSDept >
< :Dupond :TeachesIn :UE111 >
< :Dupond :TeachesTo :Pierre >
< :Pierre :EnrolledIn :CSDept >
< :Pierre :RegisteredTo :UE111 >
< :UE111 :OfferedBy :CSDept >
```

You want to add

Pierre knows someone named “John Smith” that wrote a book entitled “Introduction to Java”

```
:Pierre foaf:knows _:p
_:p foaf:name "John Smith"
_:p :wrote _:b
_:b dc:title "Introduction to Java"
```

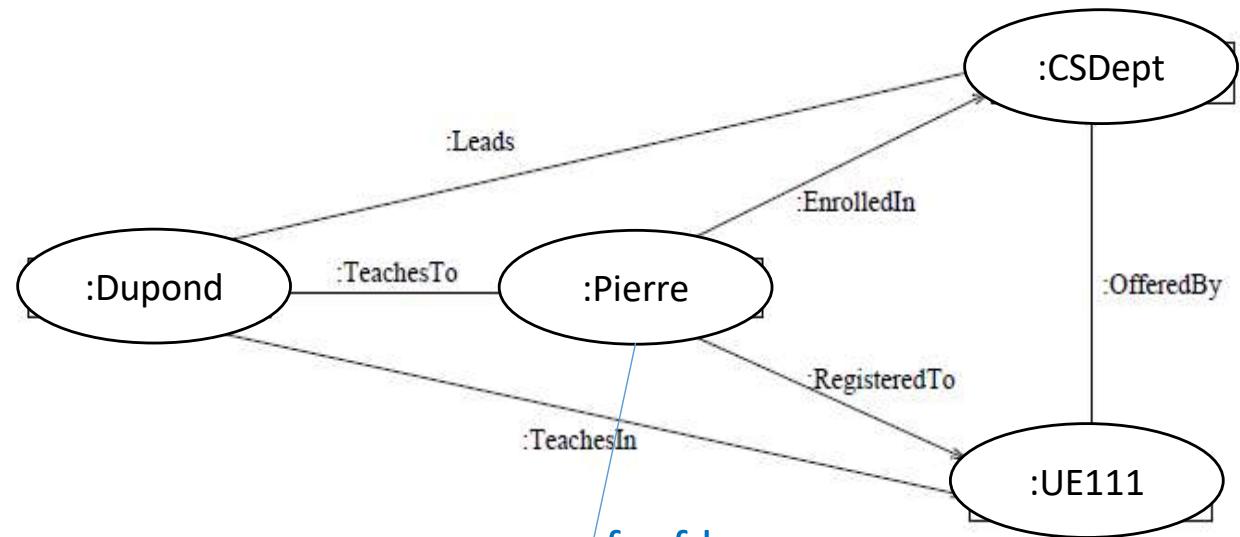
RDF graph



Blank nodes and n-ary associations

- RDF predicates always connect a subject and an object
- In the sense of predicate logic, they are binary predicates
 - Pierre¹ knows someone²
 - knows(Pierre, someone)
 - :Pierre foaf:knows _:p .
- Sometimes, n-ary predicates are needed
 - Pierre¹ knows someone² since 2000³
 - knows(Pierre, someone, 2000)
- N-ary predicates can be modeled using blank nodes

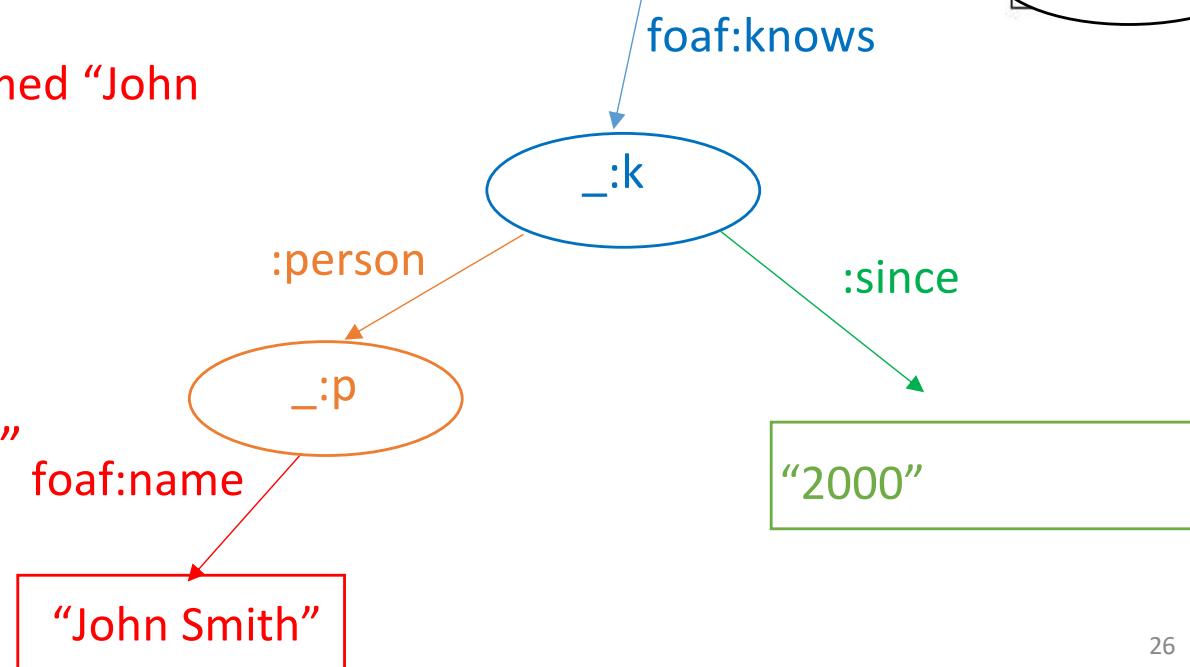
Example



You want to add

Pierre knows someone named “John Smith” since year 2000

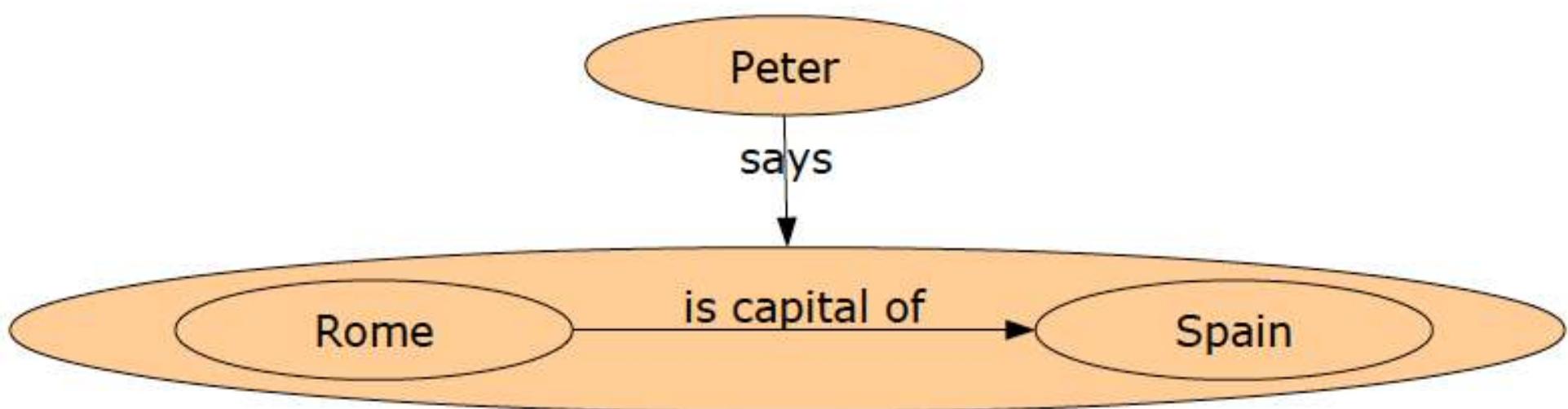
:Pierre foaf:knows _:k
_:k :person _:p
_:p foaf:name “John Smith”
_:k :since “2000”



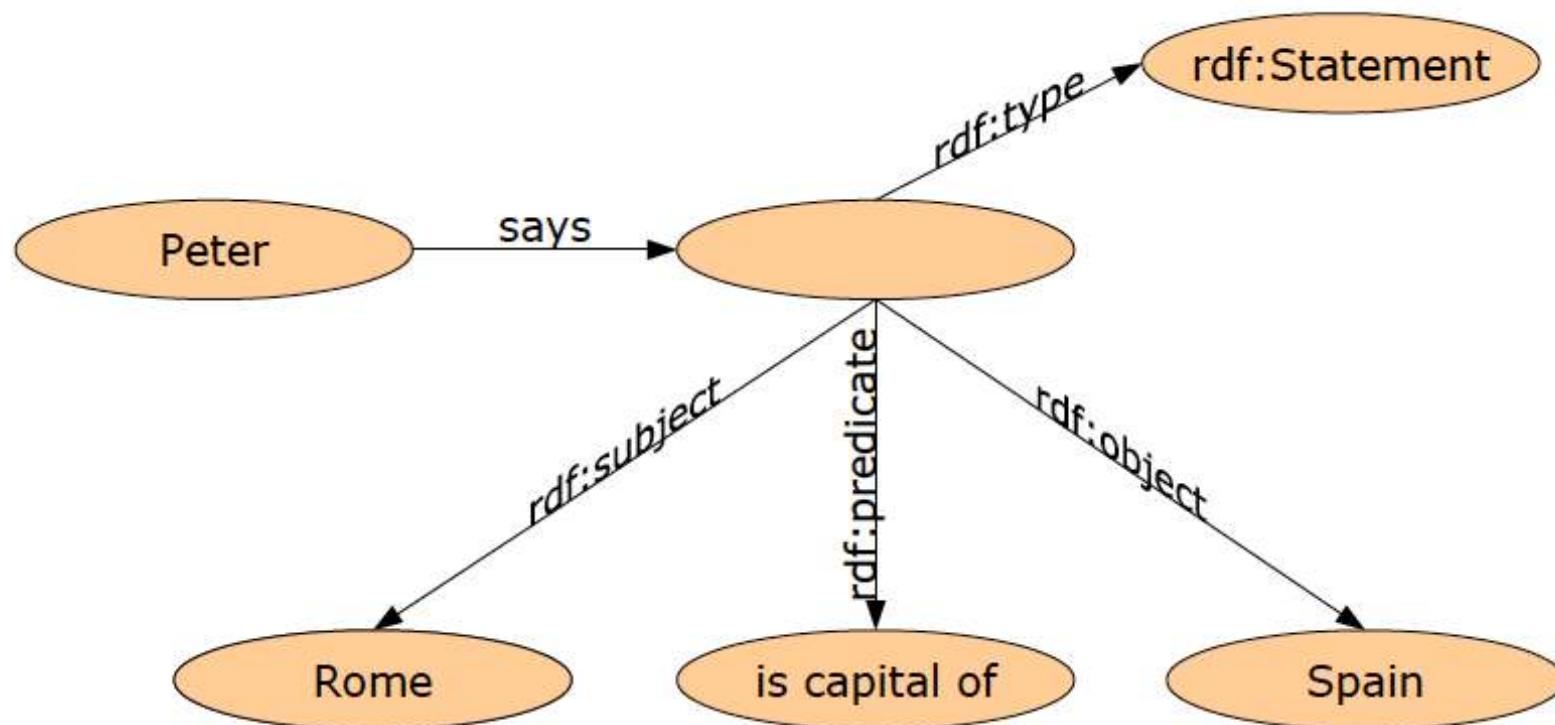
Reification in RDF

- Latin res ("Thing"), facere ("make")
 - an explication
 - making a statement, an opinion etc.
- In RDF: Statements about statements
- "Peter says that Rome is the capital of Spain.»
- Implementation:
 - RDF Statements are considered resources themselves
 - Can be subject or object of other statements
 - Reification can have multiple levels
 - “Peter says that Wikipedia states that Rome is the capital of Spain.”

Example



Example



RDF - Syntaxes

Abstract and concrete syntax

- Triples provides an abstract RDF syntax
- Several concrete languages have been provided to represent the same information
 - Triple notation
 - Turtle: simple, human readable notation for listing RDF tuples, introduce some shorthands
 - RDF/XML
 - ...

Triple notation

- A W3C standard (2004)
- Triples consist of a subject, predicate, and object
- An RDF document is an unordered set of triples

- Simple triple:

```
<http://dbpedia.org/resource/Mount_Baker>
<http://dbpedia.org/property/location>
<http://dbpedia.org/Washington> .
```

- Literal with language tag:

```
<http://dbpedia.org/resource/Mount_Baker>
<http://dbpedia.org/property/name>
"Mount Baker"@en .
```

- Typed literal:

```
<http://dbpedia.org/resource/Mount_Baker>
<http://dbpedia.org/property/lastEruption>
"1800"^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Turtle notation

- A simplified triple notation
- Namespaces as central definition

```
@prefix dbpedia : <http://dbpedia.org/resource/> .  
@prefix p : <http://dbpedia.org/property/> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
  
dbpedia:Mount_Baker p:lastEruption "1880"^^xsd:integer .  
dbpedia:Mount_Baker p:location dbpedia:Washington .  
  
dbpedia:Washington p:borderingstates dbpedia:Oregon .  
dbpedia:Washington p:borderingstates dbpedia:Idaho .
```

Olaf Hartig - ICWE 2012 Tutorial "An Introduction to SPARQL and Queries over Linked Data" - Chapter 1: Linked Data and RDF 22

- A default namespace

```
@prefix :http://www.example.org
```

Turtle notation

- A simplified triple notation
- Triples sharing
 - the same subject: lists separated by «;»
 - the same subject+predicate: lists separated by «,»

```
@prefix dbpedia : <http://dbpedia.org/resource/> .  
@prefix p : <http://dbpedia.org/property/> .  
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .  
  
dbpedia:Mount_Baker p:lastEruption "1880"^^xsd:integer ;  
                      p:location      dbpedia:Washington .  
  
dbpedia:Washington p:borderingstates dbpedia:Oregon ,  
                     dbpedia:Idaho .
```

- Shorthand notation for rdf:type:

dbpedia:Mount_Baker a dbpedia:StratoVolcano

Turtle notation: blank nodes

- **Variant 1:** explicitly named with an underscore

```
:Dieter_Fensel dc:creator _:x .  
_:x a :Book ;  
    dc:subject "Semantic Web" .
```

- **Variant 2:** unnamed with square brackets

```
:Dieter_Fensel dc:creator [ a :Book;  
                           dc:subject "Semantic Web" ] .
```

- Notes:

- both are equivalent
- changing blank node names does not change the semantics!

Turtle notation: reification

- Variant 1: Named Statement (with URI)

```
:triple1 rdf:type rdf:Statement ;
          rdf:subject :Rome ;
          rdf:predicate :isCapitalOf ;
          rdf:object :Spain .
:Peter :says :triple1 .
```

- Variant 2: Unnamed Statement (Blank Node)

```
:Peter :says [
  a rdf:Statement ;
  rdf:subject :Rome ;
  rdf:predicate :isCapitalOf ;
  rdf:object :Spain .
] .
```

Other notations

- XML/RDF (W3C standard)
 - Encodes RDF in XML
 - Suitable for machine processing (plenty of XML tools!)
- JSON-LD (W3C standard)
 - Encodes RDF in JSON
 - Useful for serializing RDF data

RDF Semantics

RDF Semantics

- A triplet $\langle s \ P \ o \rangle$ is interpreted in first-order logic (FOL) as a fact $P(s, o)$
- Example:

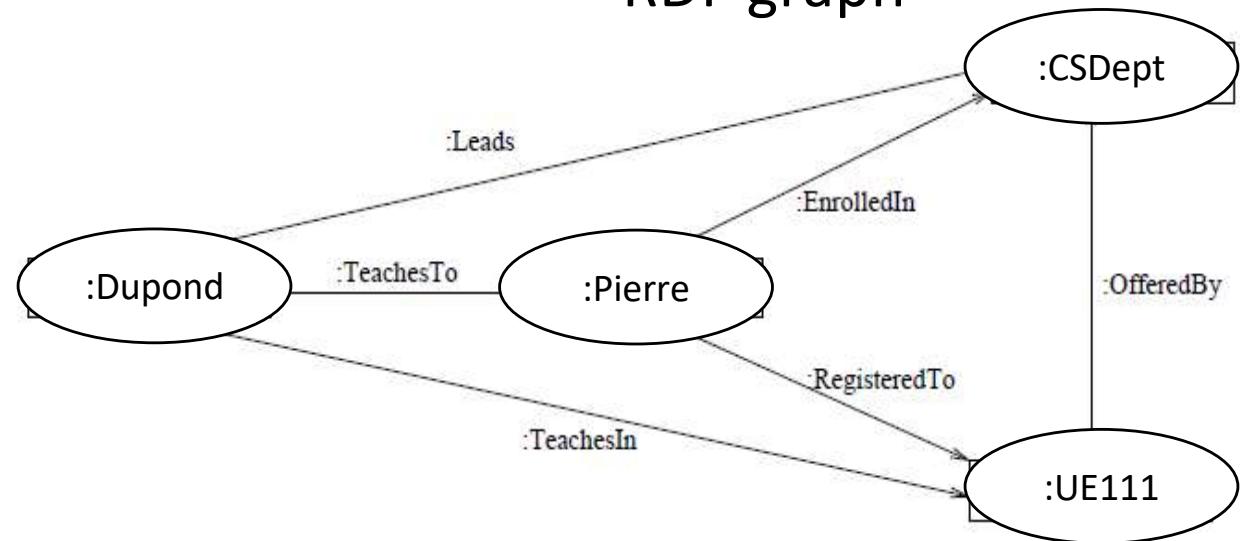
RDF triples

```
< :Dupond :Leads :CSDept >
< :Dupond :TeachesIn :UE111 >
< :Dupond :TeachesTo :Pierre >
< :Pierre :EnrolledIn :CSDept >
< :Pierre :RegisteredTo :UE111 >
< :UE111 :OfferedBy :CSDept >
```

RDF semantics

```
Leads(Dupond, CSDept)
TeachesIn(Dupond, UE111)
TeachesTo(Dupond, Pierre)
EnrolledIn(Pierre, CSDept)
RegisteredTo(Pierre, UE111)
OfferedBy(UE111, CSDept)
```

RDF graph



RDF Semantics

- Blank nodes, when they are in place of the subject or the object in triplets, are interpreted as **existential variables**
- Therefore a set of RDF triplets, possibly with blank nodes as subjects or objects, is interpreted as a **conjunction of positive literals in which all the variables are existentially quantified**
- Giving a FOL semantics to triplets in which the predicates can be blank nodes is also possible but a little bit tricky

Example

- Pierre knows someone named “John Smith” wrote a book entitled “Introduction to Java”

:Pierre foaf:knows _:p
_:p foaf:name “John Smith”
_:p wrote _:b
_:b dc:title “Introduction to Java”

$$\exists p \exists b [knows(Pierre, p) \wedge name(p, "John\ Smith") \wedge wrote(p, b) \wedge title(b, "Introduction\ to\ Java"))]$$

RDF vs Labeled Property Graph

RDF vs Labeled property graphs

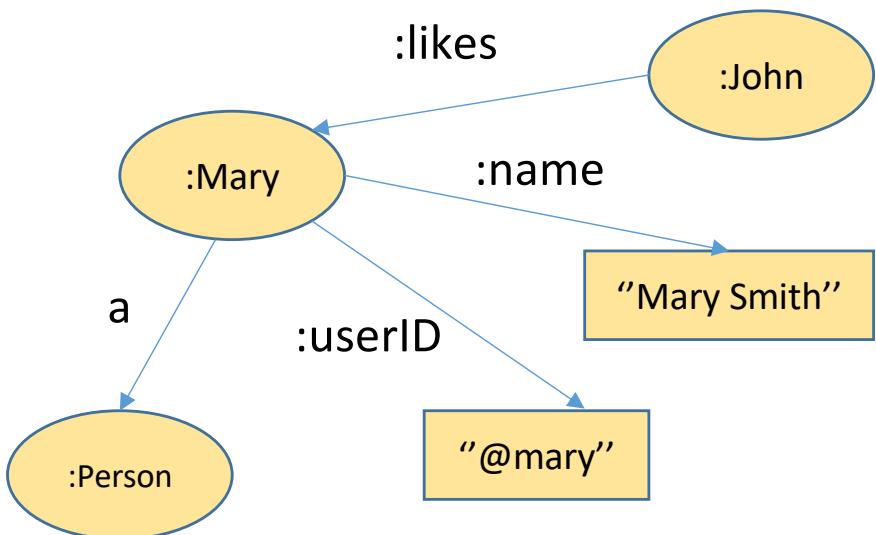
- The logical model is graph-based
- What are the differences between NoSQL labeled property graphs (LPG) and RDF graphs?
- More details at
 - <https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/>
 - <https://allegrograph.com/articles/rdf-graph-vs-property-graph-the-graph-show/>

RDF vs Labeled property graphs: general structure

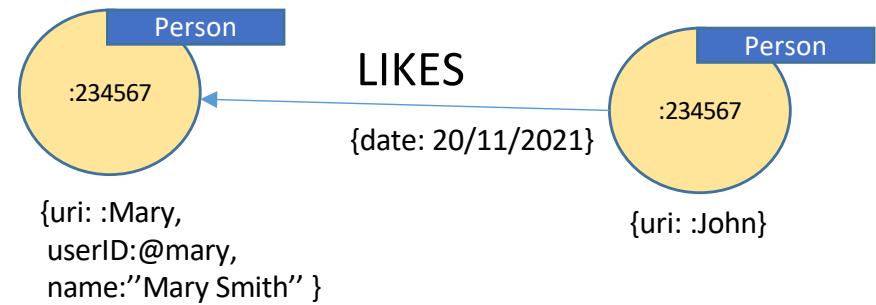
- LPG are more compact
- In LPG, nodes and edges have an internal structure representing part of the information as key-value pairs

Example

- Mary is a person
- Mary user ID is @mary
- Mary name is “Mary Smith”
- John likes Mary



- There is a person that is described by: her name, Mary Smith, her user ID, @mary. She has a globally unique identifier :Mary
- There is another person with a globally unique identifier, :John
- :John likes :Mary in date 20/11/2021

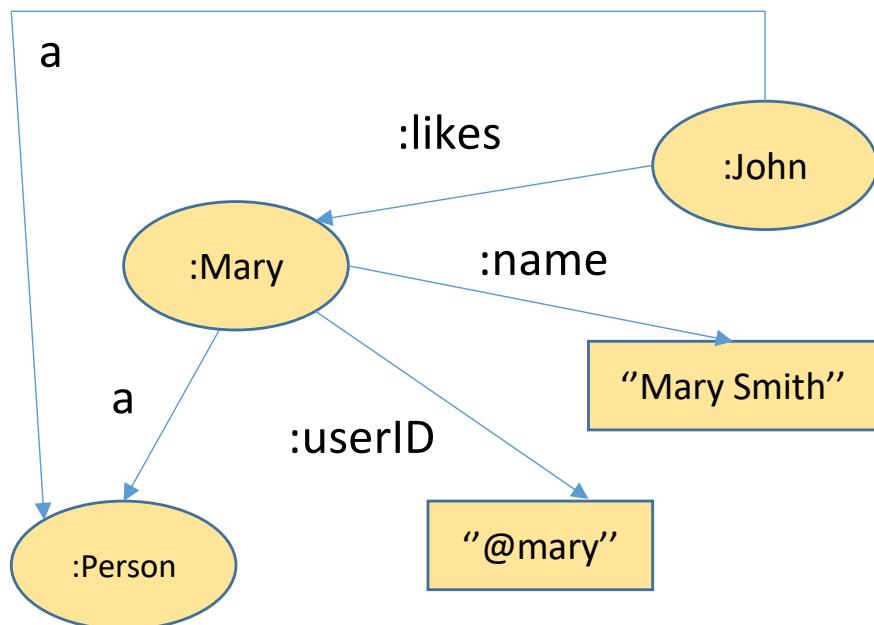


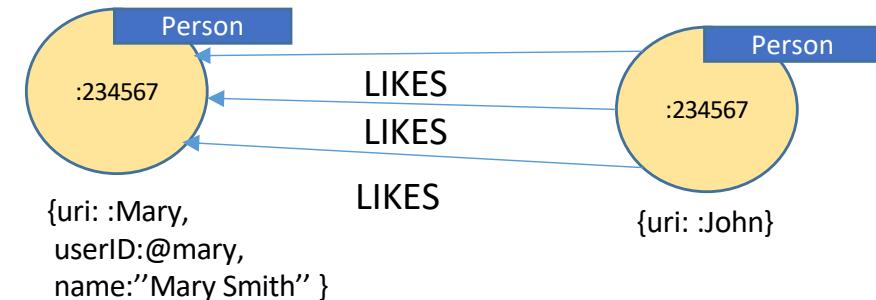
RDF vs Labeled property graphs: edge instances

- RDF does not uniquely identify instances of relationships of the same type
- It's not possible to have connections of the same type between the same pair of nodes because that would represent exactly the same triple, with no extra information

Example

- Given two nodes, one property resource can connect them just once
- Three instances of the same association between the same two nodes



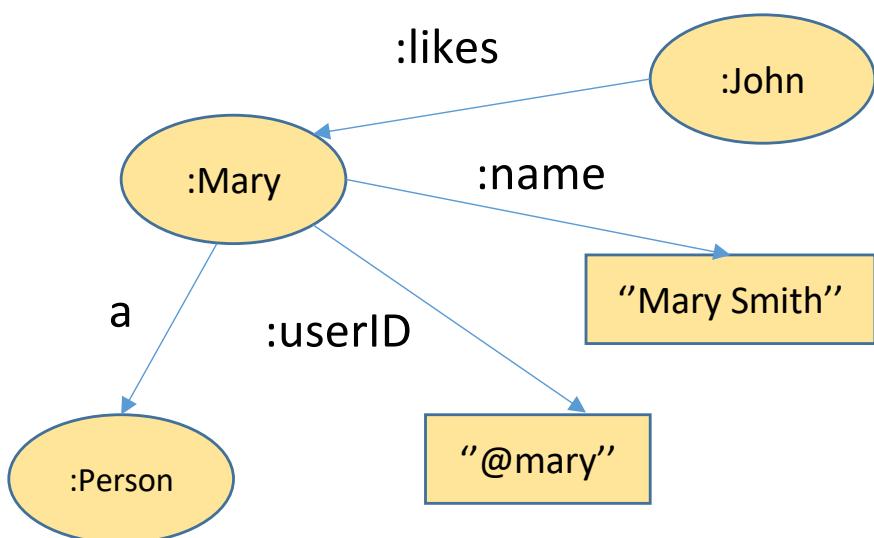


RDF vs Labeled property graphs: attributes associated with edges

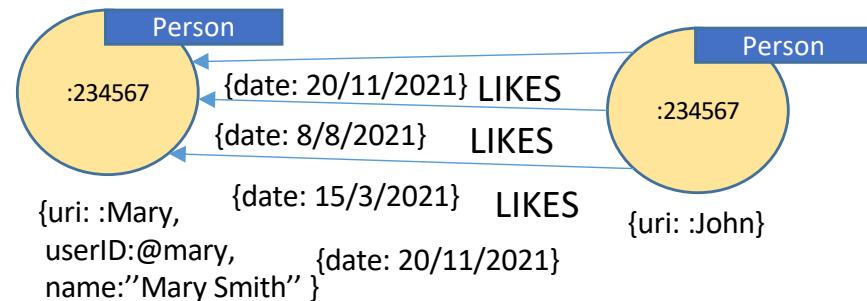
- Inability to qualify instances of relationships
- because you can't identify these unique instances in RDF, you cannot qualify them or give them attributes

Example

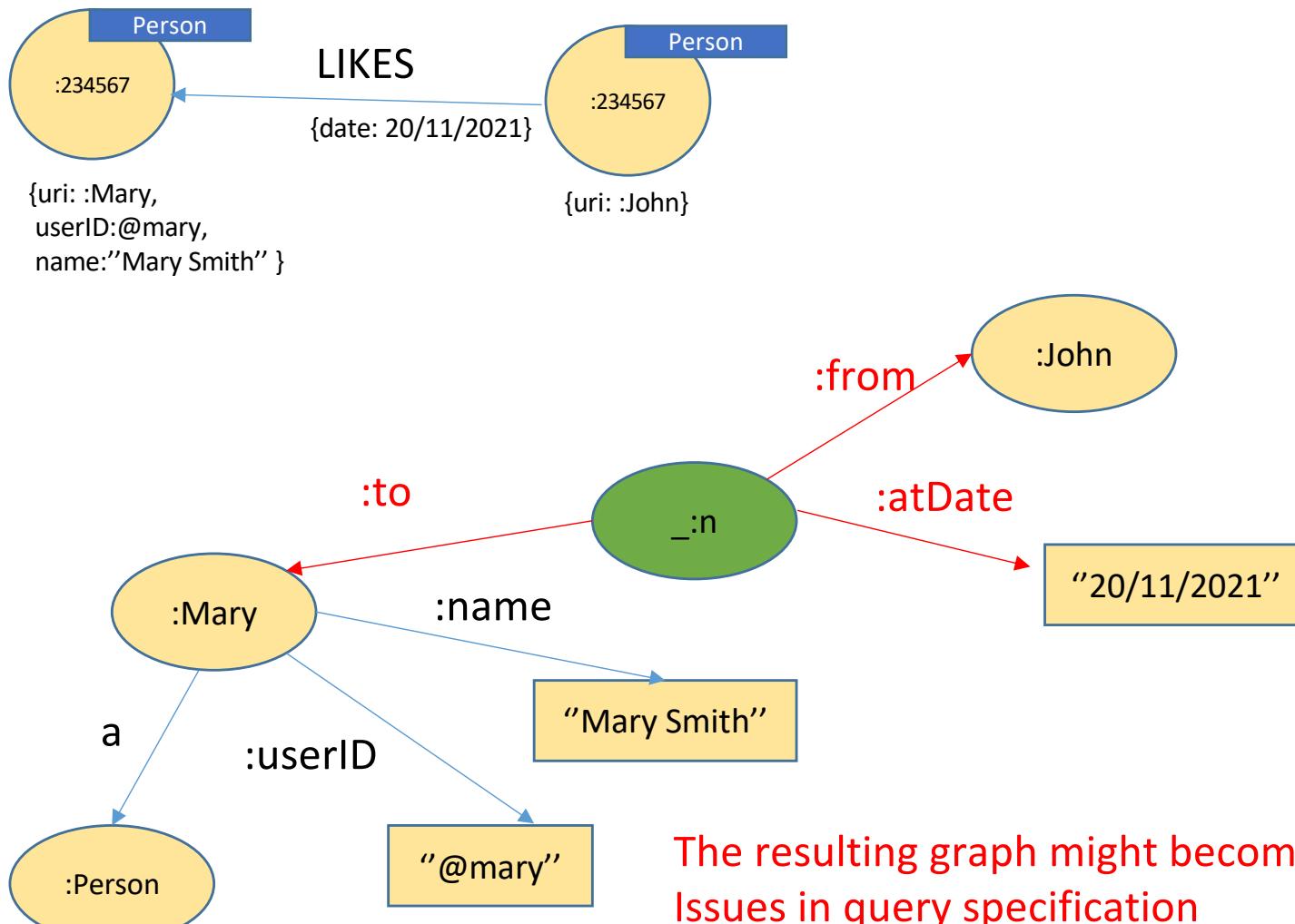
- You can add a triple like
:likes :givenAtDate "20/22/2011"
but this represent a general
property of resource :like and not
the specific instance used to
connect :Mary and :John



- Each 'likes' association, can be qualified by the reference date for the association

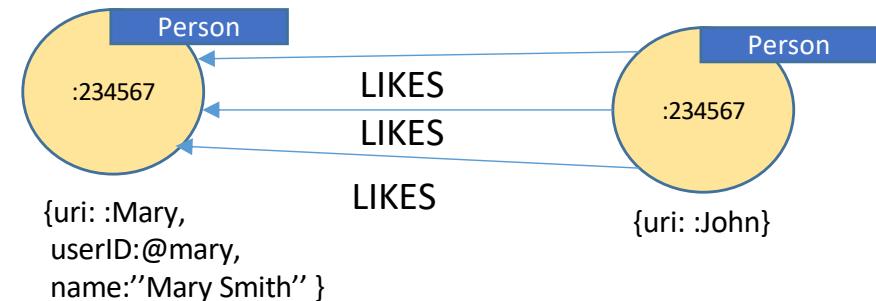
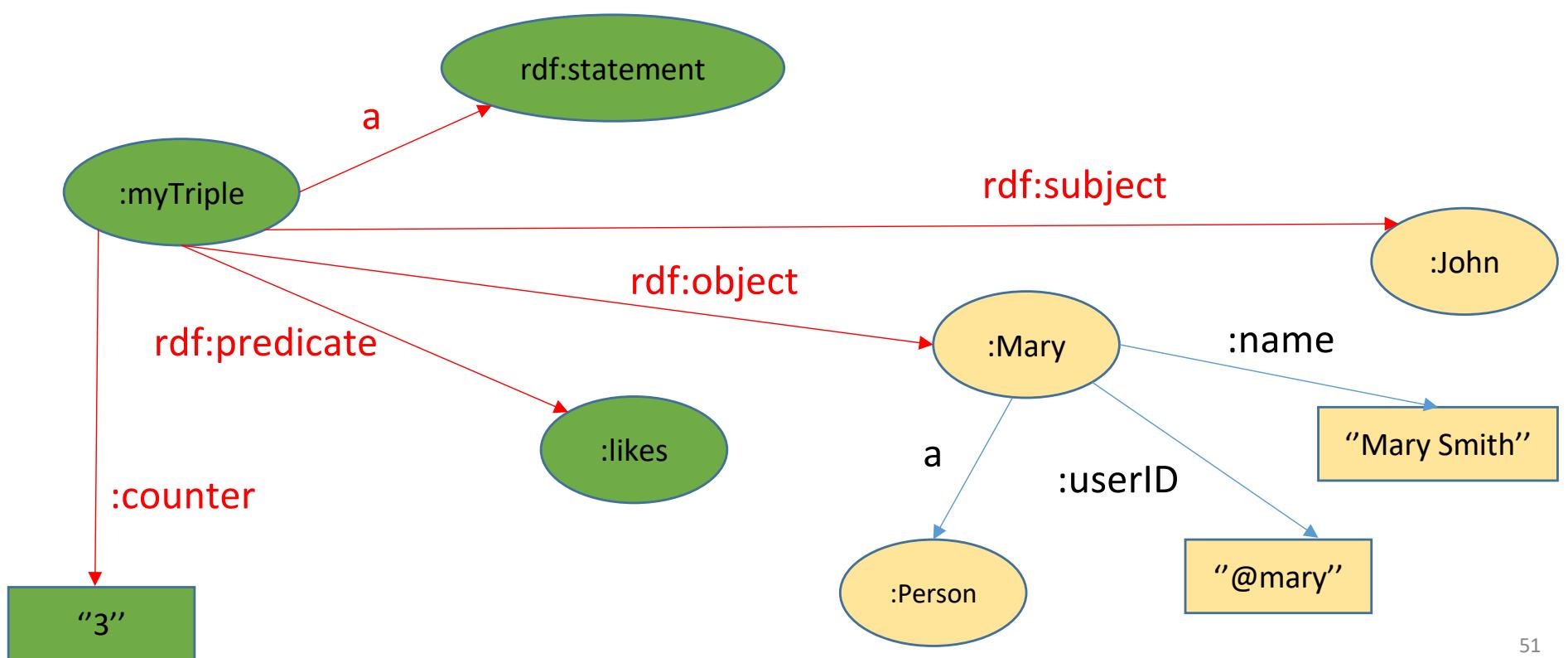


Solution 1: add new nodes



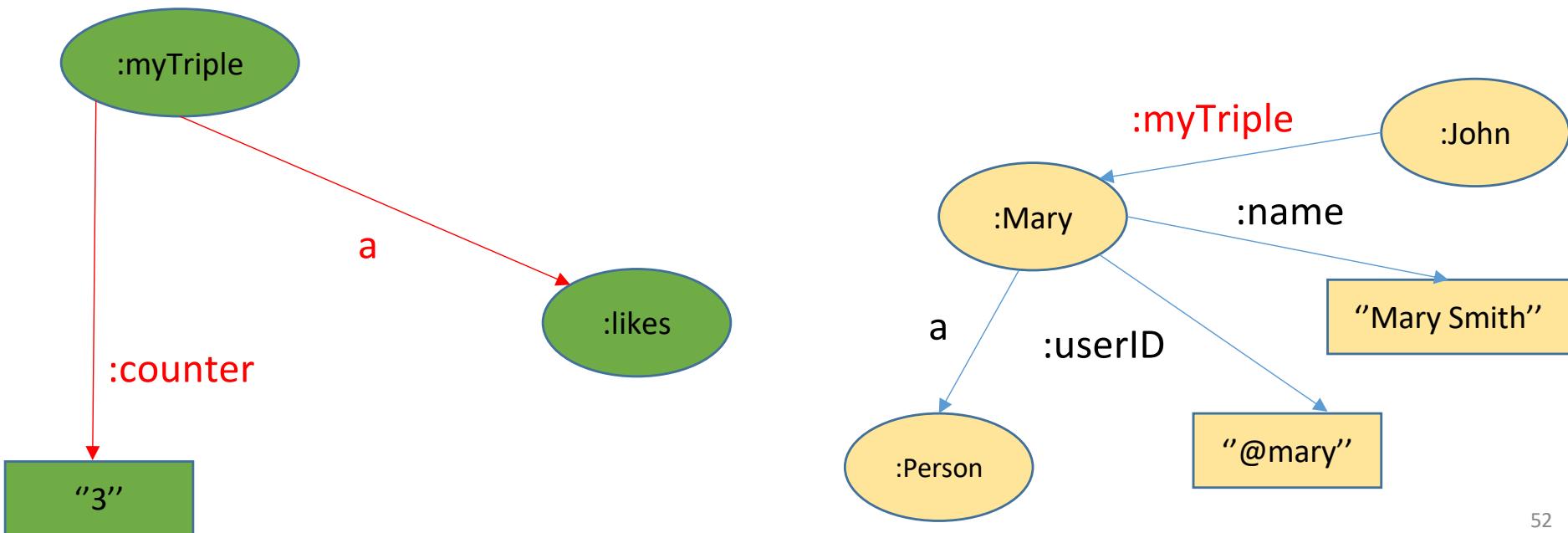
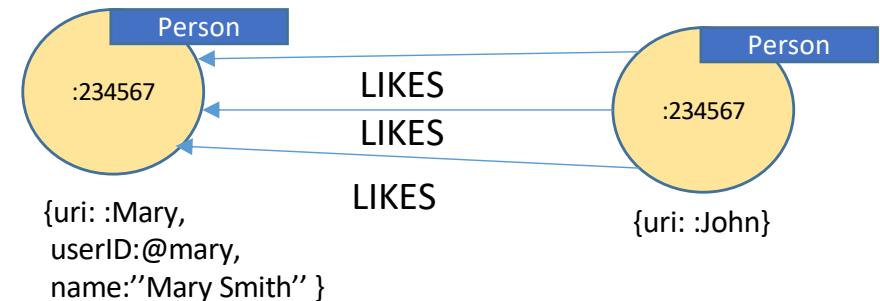
Solution 2: reification

With reification, we create a metagraph on top of our graph that represents the statement that we have here



Solution 3: Singleton property

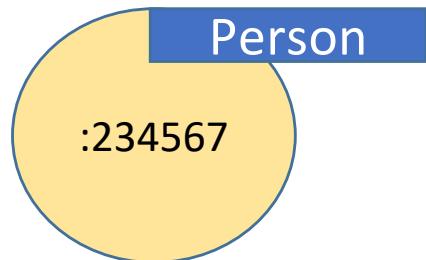
Similar to solution 2, more compact but two distinct graphs



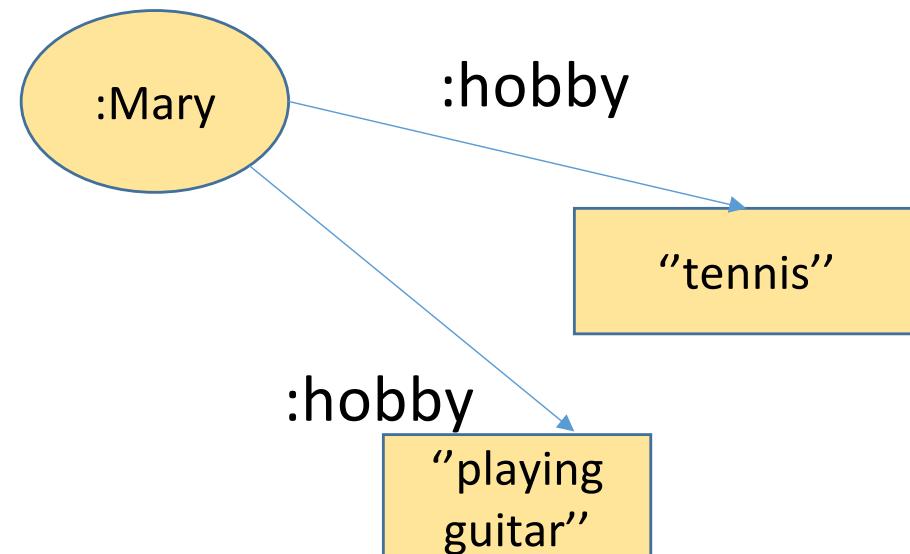
RDF vs Labeled property graphs: multivalued properties

- RDF can have multivalued properties
- In LPG you use arrays

Example



```
{uri: :Mary,  
hobbies: [ "tennis", "playing guitar" ] }
```



Property graph vs RDF

Main differences

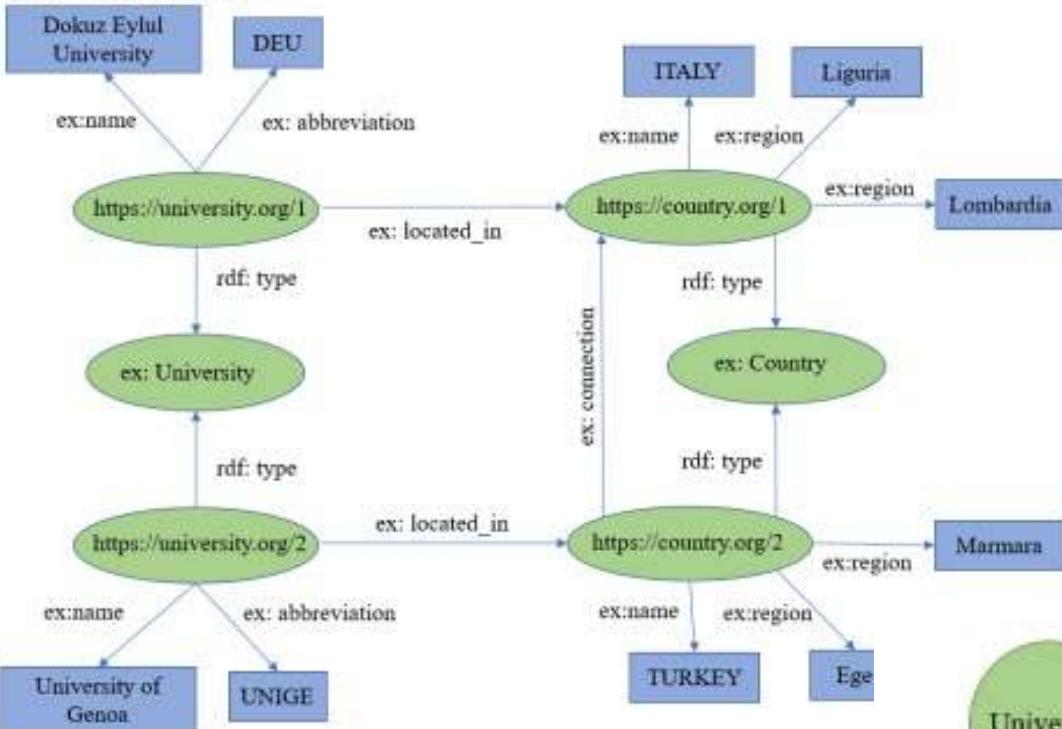
*From L10-
Graph Data Management*

- In RDF
 - Each property need to be modeled as a new, unique, node
 - No properties for relationships
 - No multiple relationships of the same type between the same nodes (they can exist in property graphs which are actually multigraphs)
 - Array-valued vs multivalue properties

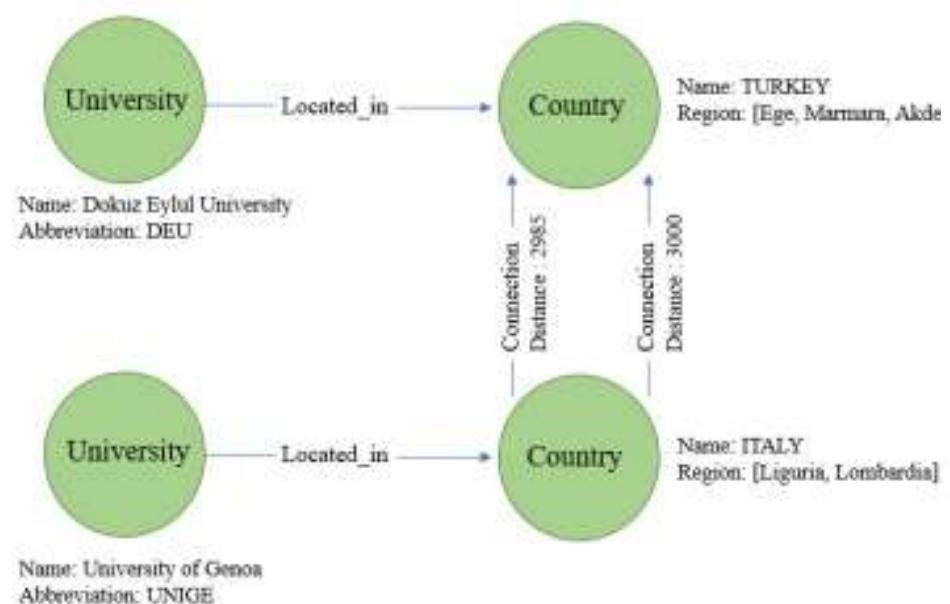
Example LPG vs RDF

From L10-
Graph Data Management

rdf: https://example.org/rdf-schema/
ex: https://example.org/



*Is connection information exactly the same?
How would you modify the RDF graph?*

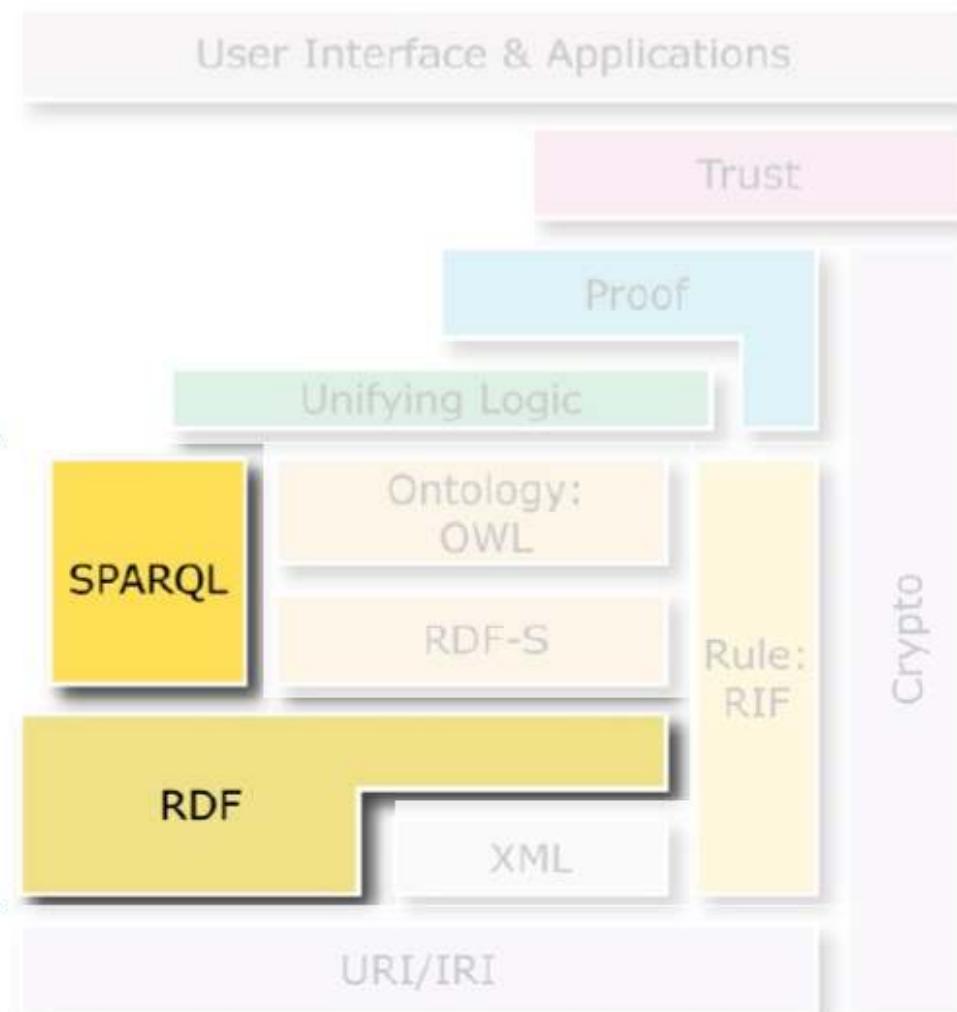


SPARQL

Query Language for RDF

Introduction

The Semantic Web Standard Stack

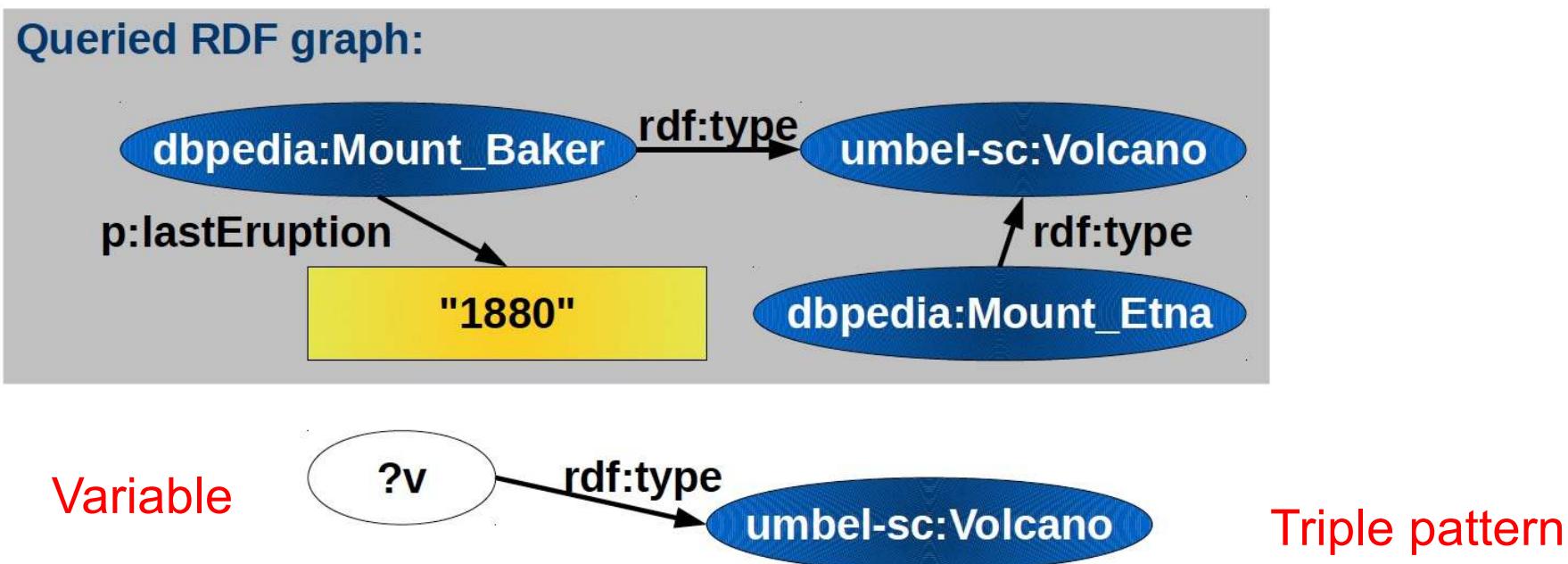


SPARQL

- Query language for getting information from RDF graphs
- It provides facilities to:
 - **extract** information in the form of URIs, blank nodes, plain and typed literals
 - **extract** RDF subgraphs
 - **construct** new RDF graphs based on information in the queried graphs
- Query terms and data description format: Turtle

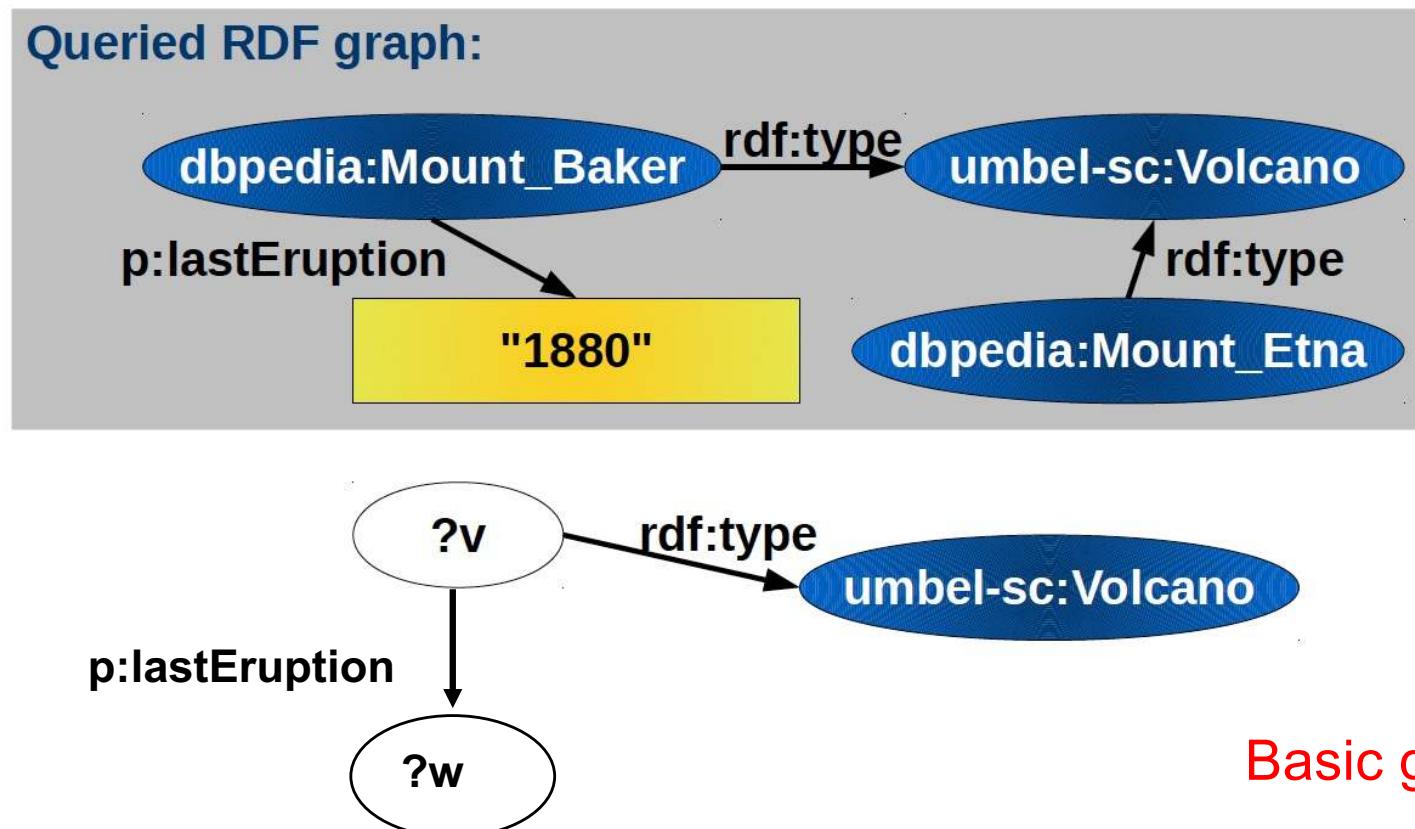
Main idea of SPARQL

- SPARQL queries are defined starting from **triple patterns** subject-property-object
 - similar to an RDF Triple (subject, predicate, object), but any component can be a **query variable** (denoted by ?<name> where <name> is the variable name)



Main idea of SPARQL

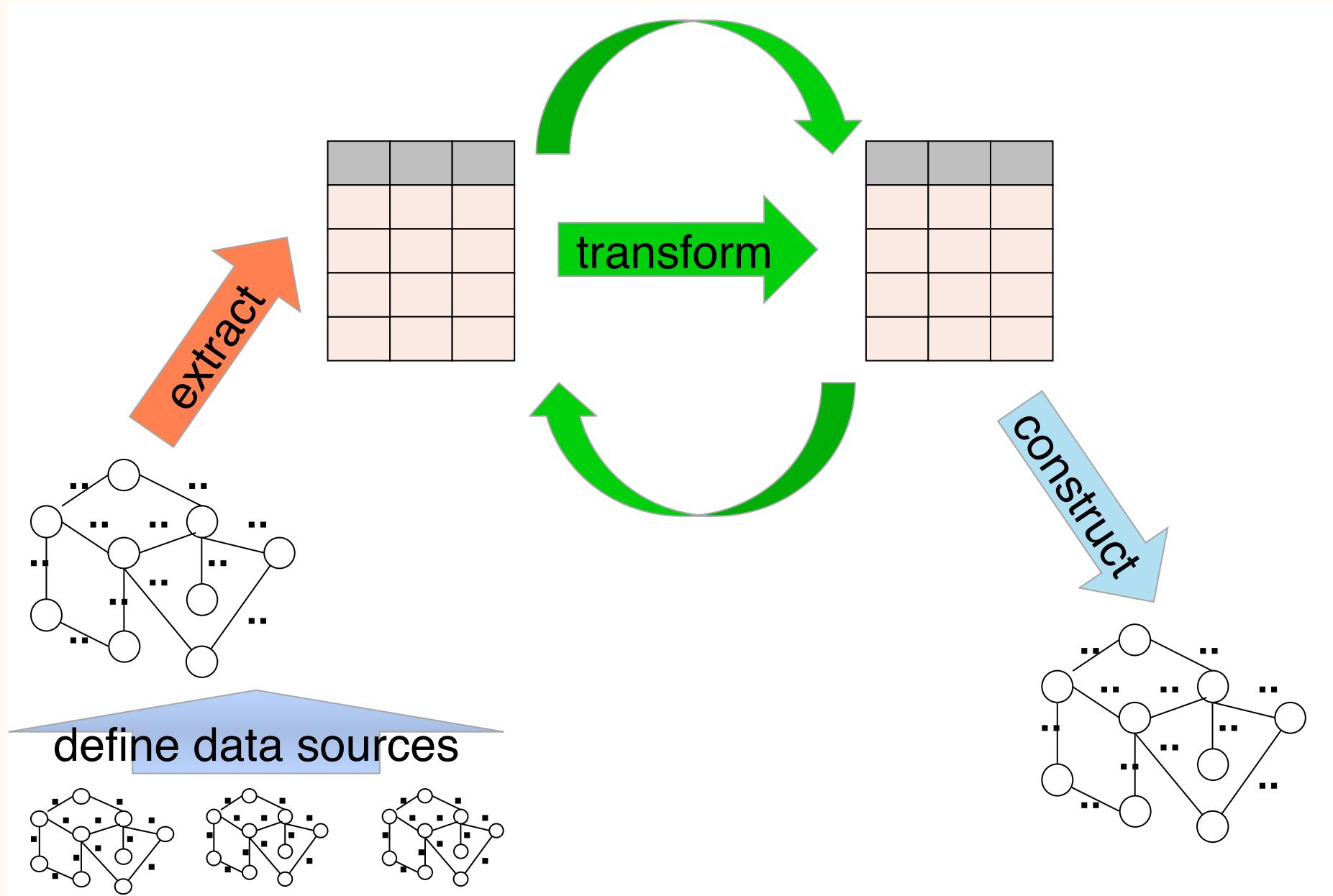
- Combining triple patterns gives a **basic graph pattern** (RDF graphs with variables)



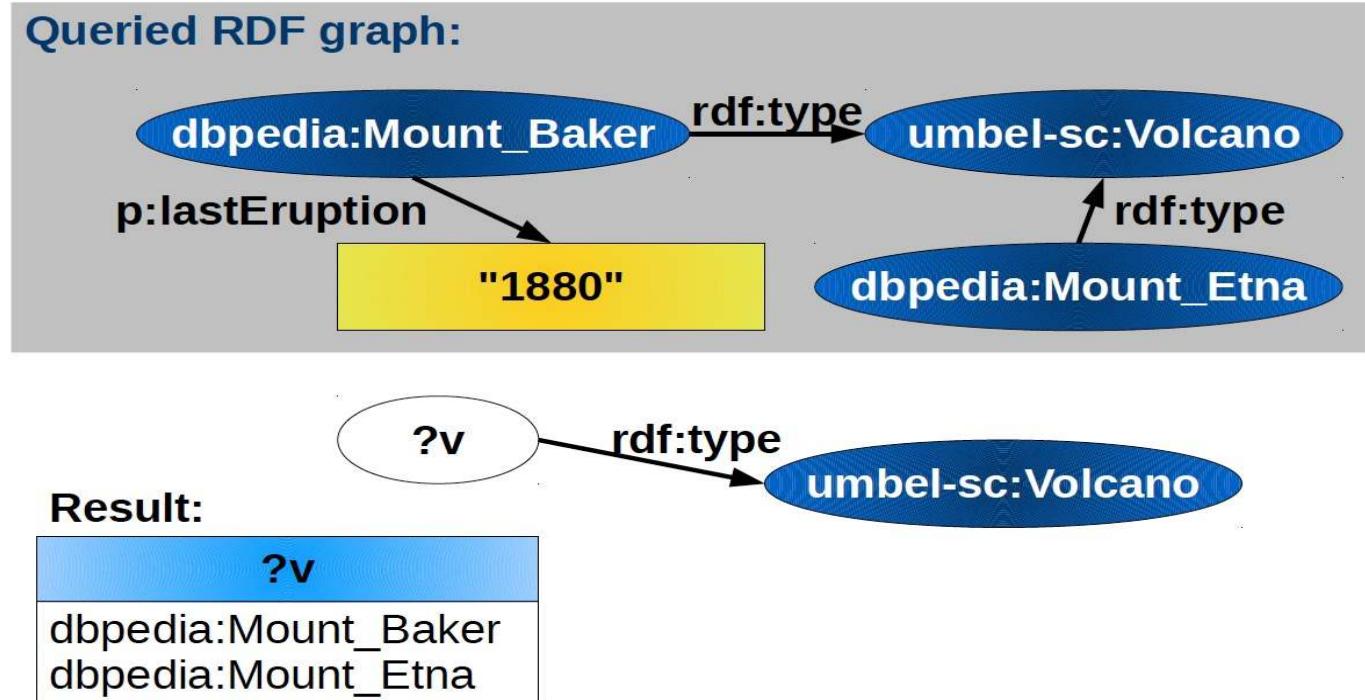
Main idea of SPARQL

- Matching a basic graph pattern to an RDF graph:
 - bindings between variables in graph pattern and RDF Terms (either resources or literals)
- A Pattern Solution of a Basic Graph Pattern GP on an RDF graph G is any substitution S of variables with RDF terms such that $S(GP)$ is a subgraph of G
- The results of SPARQL queries can be either a tabular result set or an RDF graph

SPARQL is a Graph Query Language



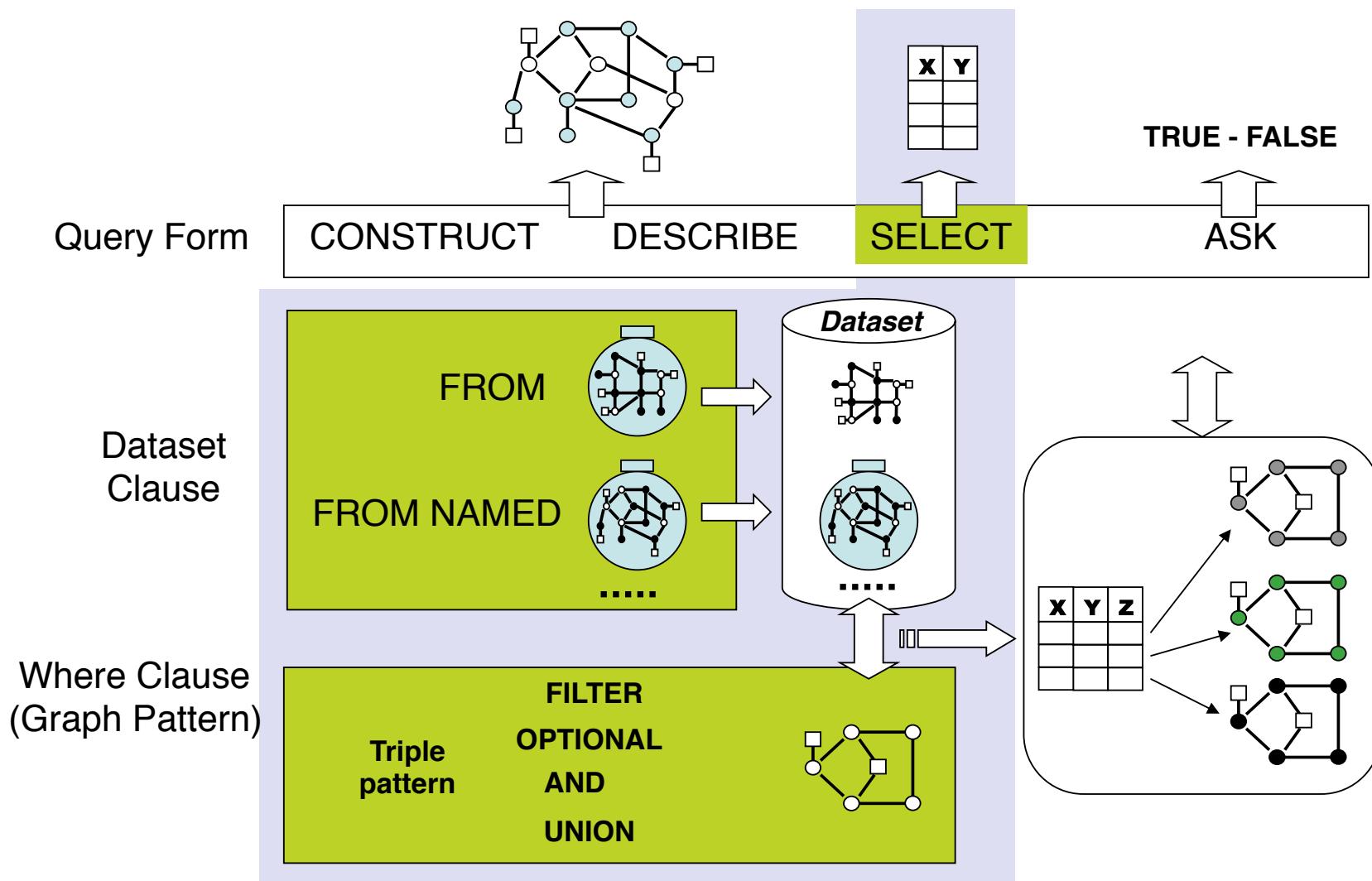
Example



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX umbel-sc: <http://umbel.org/umbel/sc/>
SELECT ?v
FROM <http://example.org/myGeoData>
WHERE {
    ?v rdf:type umbel-sc:Volcano .
}
ORDER BY ?v
```

Main SPARQL query forms

SPARQL Query



Query forms

- SPARQL has four query forms. These query forms use the solutions from pattern matching to return either a **result set** or an **RDF graph**
- **SELECT**
 - Returns all, or a subset of, the **variables bounds** by pattern matching
- **CONSTRUCT**
 - Returns an **RDF graph** constructed by substituting variables with the corresponding solution terms in a set of triple templates
- **ASK**
 - Returns a **boolean** indicating whether a query pattern matches or not the input RDF dataset
- **DESCRIBE**
 - Returns an **RDF graph** that represents all the triples **describing the resources found**

SELECT

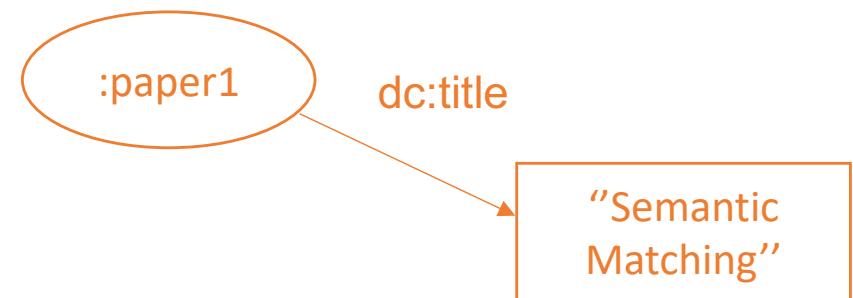
SELECT specifies the projection: the number and the order of retrieved data

FROM is used to specify the source (the RDF dataset) being queried (optional, if not specified a default RDF dataset is considered)

WHERE imposes constraints on solutions in form of graph pattern templates and boolean constraints

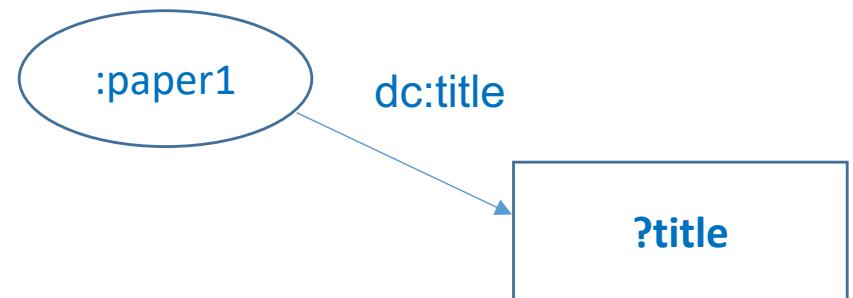
Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix : <http://example.org/book/> .  
:paper1 dc:title "Semantic Matching"
```



Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
SELECT ?title  
FROM <http://example.org/book/>  
WHERE { :paper1 dc:title ?title . }
```



SELECT (multiple matches and variables)

Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
_:a foaf:name "Tim Berners-Lee" .  
_:a foaf:homepage <http://www.w3.org/People/Berners-Lee/> .  
_:b foaf:name "Barbara Catania" .  
_:b foaf:homepage <http://www.dibris.unige.it/catania-barbara> .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
SELECT ?name ?homepage  
WHERE { ?x foaf:name ?name .  
       ?x foaf:homepage ?homepage . }
```

Result

| name | homepage |
|-----------------|--|
| Tim Berners-Lee | <http://www.w3.org/People/Berners-Lee/> |
| Barbara Catania | <http://www.dibris.unige.it/catania-barbara> |

SELECT (expressions)

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix : <http://example.org/book/> .  
@prefix ns: <http://example.org/ns#> .  
:book1 dc:title "SPARQL Tutorial" .  
:book1 ns:price 42 .  
:book1 ns:discount 0.2 .  
:book2 dc:title "The Semantic Web" .  
:book2 ns:price 23 .  
:book2 ns:discount 0.25 .
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
PREFIX ns: <http://example.org/ns#>  
SELECT ?title, (?p*(1-?discount) AS ?price)  
WHERE  
{ ?x ns:price ?p .  
?x dc:title ?title .  
?x ns:discount ?discount .  
}
```

Result

| title | price |
|--------------------|-------|
| "The Semantic Web" | 17.25 |
| "SPARQL Tutorial" | 33.6 |

SELECT (filtering, term restriction)

FILTER specifies how solutions are restricted to those RDF terms which match with the filter expression (only the bindings satisfying the FILTER condition are returned)

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix : <http://example.org/book/> .  
@prefix ns: <http://example.org/ns#> .  
:book1 dc:title "SPARQL Tutorial".  
:book1 ns:price 42 .  
:book1 ns:discount 0.2 .  
:book2 dc:title "The Semantic Web" .  
:book2 ns:price 23 .  
:book2 ns:discount 0.25 .
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
PREFIX ns: <http://example.org/ns#>  
SELECT ?title ?p  
WHERE  
{ ?x ns:price ?p .  
?x dc:title ?title .  
FILTER (?p >= 25)  
}
```

Result

| title | p |
|--------------------|----|
| "The Semantic Web" | 42 |

Joins

(Implicit join) Retrieve all the titles and the associated prices

```
SELECT ?title ?p
```

```
WHERE
```

```
{ ?x ns:price ?p .
```

```
?x dc:title ?title .
```

```
}
```

(Explicit join) Retrieve all the titles and the associated prices

```
SELECT ?title ?p
```

```
WHERE
```

```
{ ?x ns:price ?p .
```

```
?Y dc:title ?title .
```

```
FILTER (?x = ?y)
```

```
}
```

Blank Nodes

Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
_:a foaf:name "Alice" .  
_:b foaf:name "Bob" .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
SELECT ?x ?name  
WHERE { ?x foaf:name ?name }
```

Result

| x | name |
|-----|---------|
| _:c | "Alice" |
| _:d | "Bob" |

Alternative Graph Patterns

@prefix dc10: <<http://purl.org/dc/elements/1.0/>> .
@prefix dc11: <<http://purl.org/dc/elements/1.1/>> .
Data `_:a dc10:title "SPARQL Query Language Tutorial" .`
 `_:b dc11:title "SPARQL Protocol Tutorial" .`
 `_:c dc10:title "SPARQL" .`
 `_:c dc11:title "SPARQL (updated)" .`

PREFIX dc10: <<http://purl.org/dc/elements/1.0/>>

Query

PREFIX dc11: <<http://purl.org/dc/elements/1.1/>>

SELECT ?x ?y

WHERE { { ?book dc10:title ?x } UNION { ?book dc11:title ?y } }

Result

| x | y |
|----------------------------------|----------------------------|
| | "SPARQL (updated)" |
| | "SPARQL Protocol Tutorial" |
| "SPARQL" | |
| "SPARQL Query Language Tutorial" | |

CONSTRUCT

The **CONSTRUCT** query form returns a single RDF graph specified by a graph template.

- **The result.** The result is an RDF graph formed by
 - taking each query solution in the solution sequence,
 - substituting for the variables in the graph template,
 - combining the triples into a single RDF graph by set union
- **Ignored triples.** If any such instantiation produces a triple containing an unbound variable or an illegal RDF construct (e.g., a literal in subject or predicate position) then that triple is not included in the output RDF graph
- **Ground triples.** The graph template can contain triples with no variables (known as ground or explicit triples), and these also appear in the output RDF graph returned by the CONSTRUCT query form

CONSTRUCT

Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
_:a foaf:name "Alice" .
```

```
_:a foaf:mbox <mailto:alice@example.org> .
```

Query

```
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
CONSTRUCT { <http://example.org/person#Alice> vcard:FN ?name }
```

```
WHERE { ?x foaf:name ?name }
```

Result

It creates vcard properties from the FOAF information:

```
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .  
<http://example.org/person#Alice> vcard:FN "Alice" .
```

ASK

- Applications can use the **ASK form** to test whether or not a query pattern has a solution.
- No information is returned about the possible query solutions, just whether or not a solution exists (a **Boolean value**)

Data

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
_:a foaf:name "Alice" .  
_:a foaf:homepage <http://work.example.org/alice/> .  
_:b foaf:name "Bob" .  
_:b foaf:mbox <mailto:bob@work.example> .
```

Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
ASK { ?x foaf:name "Alice" }
```

Result

true

DESCRIBE

The [DESCRIBE form](#) returns a single RDF graph containing RDF data about resources

- The query pattern is used to create a **result set**
- The DESCRIBE form takes each of the resources identified in a solution and returns a "description" of such resources, in the form of an RDF graph, which can come from any information available including the target RDF Dataset
- The description is determined by the query service
- The syntax DESCRIBE * is an abbreviation that describes all of the variables in a query.

Query 1

[DESCRIBE <http://example.org/>](#)

Query 2

PREFIX foaf: <<http://xmlns.com/foaf/0.1/>>

DESCRIBE ?x

[WHERE { ?x foaf:name "Alice" }](#)

Other clauses and modifiers

OPTIONAL

OPTIONAL allows binding variables to RDF terms to be included in the solution in case of availability (basically it allows for empty cells in the result set)

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
_:a dc:creator "Tim Berners-Lee" .  
_:a foaf:age 53 .  
_:a foaf:homepage <http://www.w3.org/People/Berners-Lee/> .  
  
_:b dc:creator "Alice Smith" .
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
SELECT ?author ?age  
WHERE { ?x dc:creator ?author .  
       OPTIONAL {?x foaf:age ?age}}
```

| author | Age |
|-------------------|-----|
| "Tim Berners-Lee" | 53 |
| "Alice Smith" | |

ORDER BY

ORDER BY is a facility to order a solution sequence

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
_:a dc:creator "Alice Smith" .  
_:a foaf:age 48 .  
_:b dc:creator "Tim Berners-Lee" .  
_:b foaf:age 53.
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
SELECT ?author  
WHERE { ?x dc:creator ?author .  
      ?x foaf:age ?age }  
ORDER BY ?author DESC
```

Result

| author |
|-------------------|
| "Tim Berners-Lee" |
| "Alice Smith" |

DISTINCT modifier

The **DISTINCT** solution modifier eliminates duplicate solutions. Only one solution that binds the same variables to the same RDF terms is returned from the query.

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
```

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
_:a dc:creator "Alice Smith" .
```

```
_:a foaf:age 22 .
```

```
_:b dc:creator "Alice Smith" .
```

```
_:b foaf:age 48.
```

Result

| |
|---------------|
| creator |
| "Alice Smith" |

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
SELECT DISTINCT ?creator
```

```
WHERE { ?x dc:creator ?creator}
```

LIMIT

The **LIMIT** clause puts an upper bound on the number of solutions returned. If the number of actual solutions, after OFFSET is applied, is greater than the limit, then at most the limit number of solutions will be returned. A LIMIT of 0 would cause no results to be returned.

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
_:a dc:creator "Alice Smith" .  
_:a foaf:age 48 .  
  
_:b dc:creator "Tim Berners-Lee" .  
_:b foaf:age 53.
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
SELECT ?author  
WHERE { ?x dc:creator ?author }  
ORDER BY ?author LIMIT 1
```

Result

| |
|---------------|
| auhor |
| "Alice Smith" |

OFFSET

The **OFFSET** clause causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

Data

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .  
@prefix foaf: <http://xmlns.com/foaf/0.1/> .  
  
_:a dc:creator "Alice Smith" .  
_:a foaf:age 48 .  
  
_:b dc:creator "Tim Berners-Lee" .  
_:b foaf:age 53.
```

Query

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>  
SELECT ?author  
  
WHERE { ?x dc:creator ?author }  
  
ORDER BY ?author OFFSET 1
```

| |
|-------------------|
| auhor |
| "Tim Berners-Lee" |

Additional features

- **New features of SPARQL 1.1 Query:**
 - Aggregate functions (e.g. COUNT, SUM, AVG)
 - Subqueries
 - Negation (EXISTS, NOT EXISTS, MINUS)
 - Assignments (e.g. BIND, SELECT expressions)
 - Property paths
 - Basic query federation (SERVICE, BINDINGS)
- **SPARQL 1.1 Update:**
 - Graph update (INSERT DATA, DELETE DATA, INSERT, DELETE, DELETE WHERE, LOAD, CLEAR)
 - Graph management (CREATE, DROP, COPY, MOVE, ADD)

Combining classes,
property restrictions and
intensional classes
(Optional further reading)

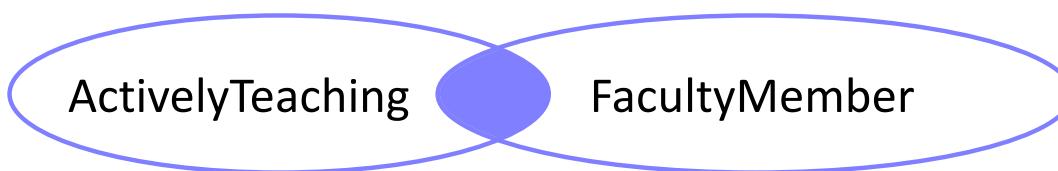
Class Union and Intersection, Enumeration, Complement

- Allow combining classes

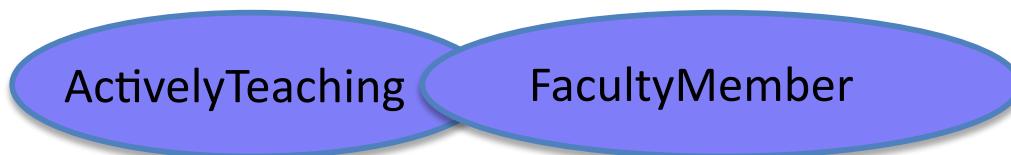
| OWL notation | FOL translation |
|--|--------------------------|
| <code>owl:intersectionOf (C,D...)</code> | $C(X) \wedge D(X) \dots$ |
| <code>owl:unionOf (C,D...)</code> | $C(X) \vee D(X) \dots$ |
| <code>owl:oneOf (e,f...)</code> | $X = e \vee X = f \dots$ |
| <code>owl:complementOf (C)</code> | $\neg C(X)$ |

Examples

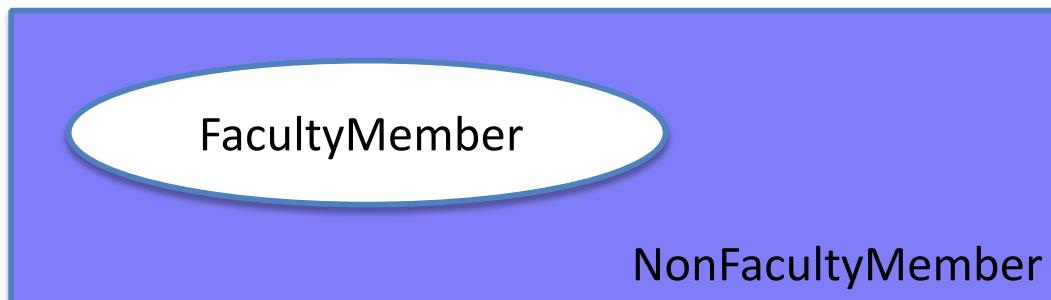
`_:X owl:intersectionOf (ex:ActivelyTeaching ex:FacultyMember) .`



`_:X owl:unionOf (ex:ActivelyTeaching ex:FacultyMember) .`



`_:X owl:complementOf (ex:FacultyMember) .`



Unnamed Classes

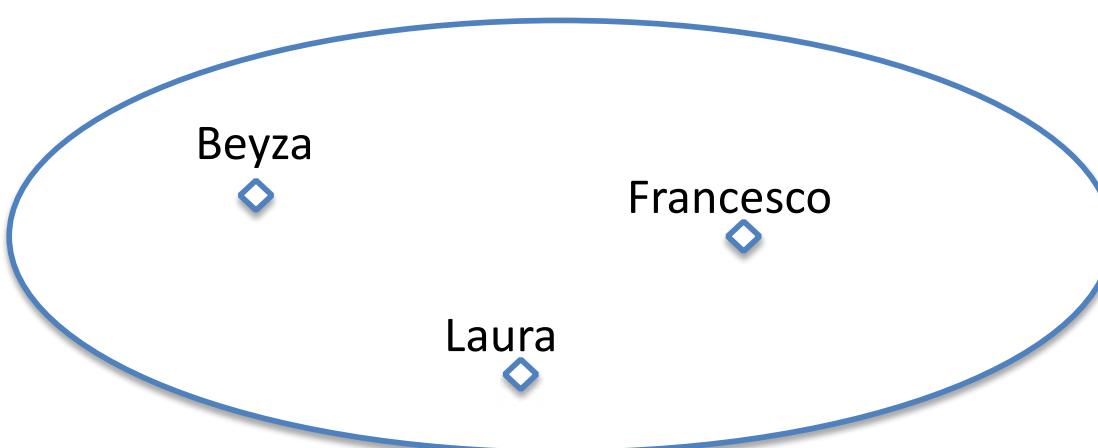
- The combination of classes with logical operators describes an anonymous (unnamed) class
- Other classes can be declared to be a subclass or equivalent to this anonymous class

```
ex:NonFacultyMember rdfs:subClassOf _:X .  
_:X rdf:type owl:Class .  
_:X owl:complementOf ex:FacultyMember .
```

```
ex:TeachingFaculty owl:equivalentClass [  
rdf:type owl:Class ;  
owl:intersectionOf (ex:ActivelyTeaching, ex:FacultyMember) ].
```

Closed Classes

```
:PhDStudents owl:equivalentClass [  
    rdf:type owl:Class ;  
    owl:oneOf ( :Beyza :Laura :Francesco ) ].
```



The OWL View of Life

- OWL is not like a database system
- no requirement that the only properties of an individual are those mentioned in a class it belongs to
- no assumption that everything is known
- classes and properties can have multiple “definitions”
- statements about individuals need not be together (in the same document)

Property Restrictions and Definition of Intensional Classes

- Goal: allow expressing complex constraints such as:
 - ▶ departments can be lead only by professors
 - ▶ only professors or lecturers may teach to undergraduate students.
- The keyword `owl:Restriction` is used in association with a `blank node class`, and some specific restriction properties:
 - ▶ `owl:someValuesFrom`
 - ▶ `owl:allValuesFrom`
 - ▶ `owl:minCardinality`
 - ▶ `owl:maxCardinality`
 - ▶ `owl:hasValue`

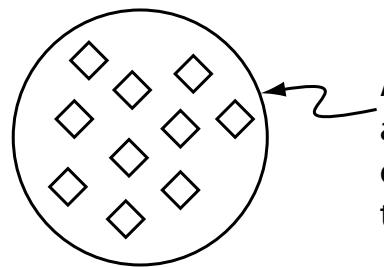
Property Restrictions

| OWL notation | FOL translation |
|---------------------------------------|---|
| <code>_:a owl:onProperty P</code> | |
| <code>_:a owl:allValuesFrom C</code> | $\forall Y (P(X, Y) \Rightarrow C(Y))$ |
| <code>_:a owl:onProperty P</code> | |
| <code>_:a owl:someValuesFrom C</code> | $\exists Y (P(X, Y) \wedge C(Y))$ |
| <code>_:a owl:onProperty P</code> | |
| <code>_:a owl:minCardinality n</code> | $\exists Y_1 \dots \exists Y_n (P(X, Y_1) \wedge \dots \wedge P(X, Y_n) \wedge \bigwedge_{i,j \in [1..n], i \neq j} (Y_i \neq Y_j))$ |
| <code>_:a owl:maxCardinality n</code> | $\forall Y_1 \dots \forall Y_n \forall Y_{n+1} (P(X, Y_1) \wedge \dots \wedge P(X, Y_n) \wedge P(X, Y_{n+1}) \Rightarrow \bigvee_{i,j \in [1..n+1], i \neq j} (Y_i = Y_j))$ |

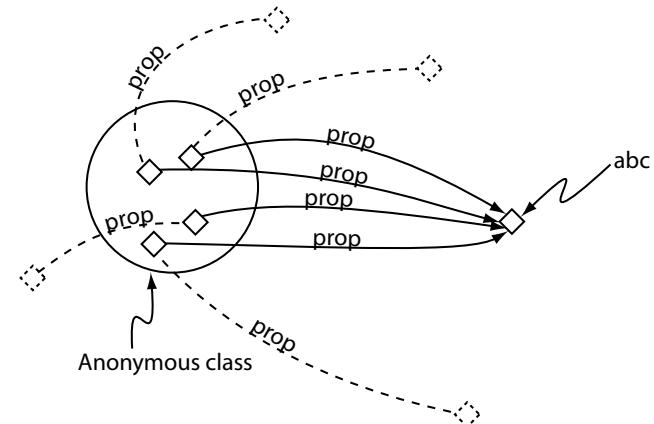
`_:a owl:onProperty P` $P(X, v)$

`_:a owl:hasValue v`

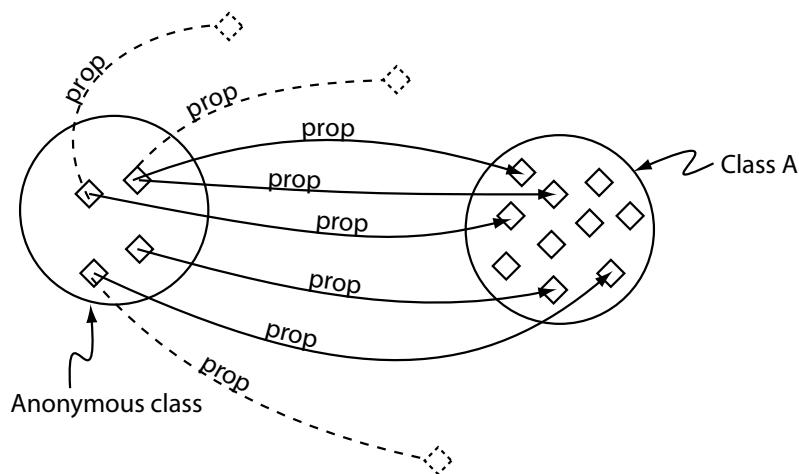
Restriction Classes



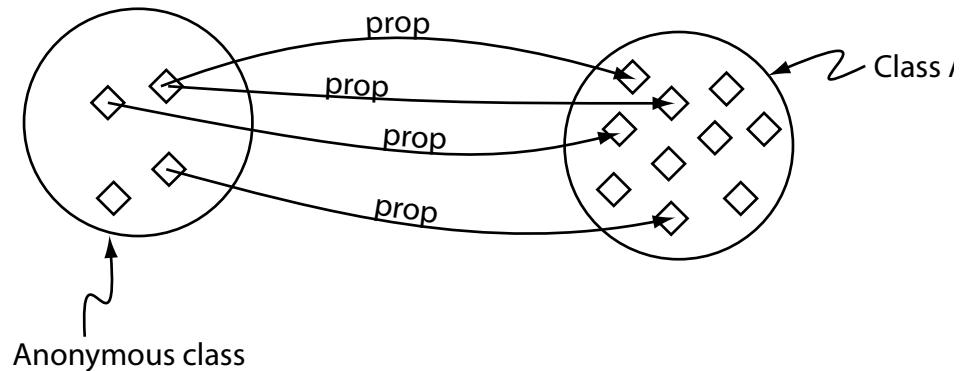
A set of individuals that satisfy a restriction - the restriction essentially describes an anonymous (unnamed) class that contains these individuals.



prop hasValue abc



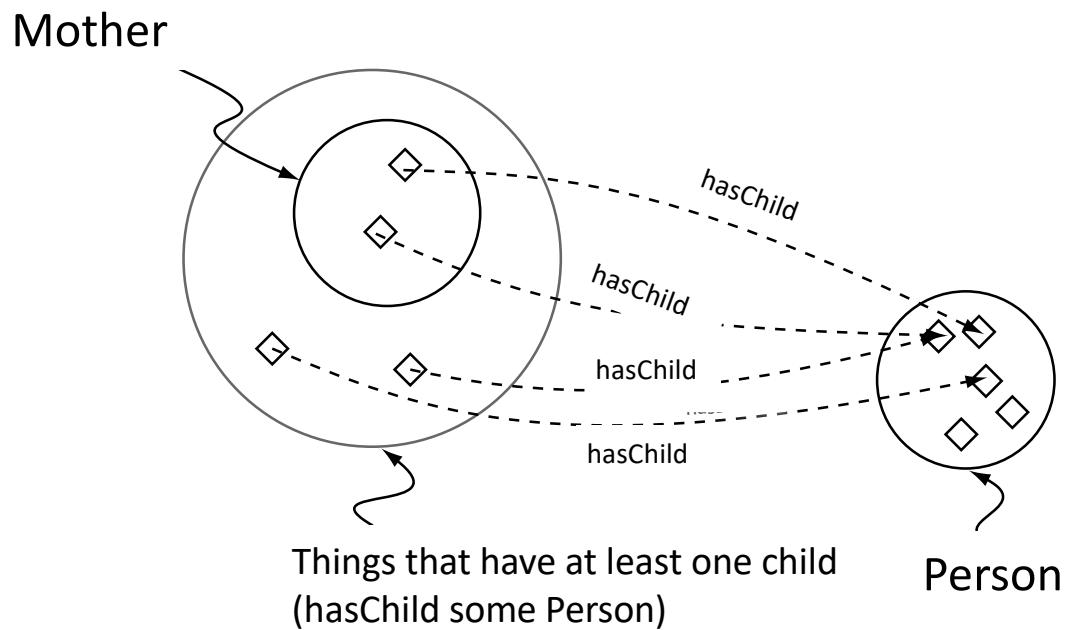
prop someValuesFrom A



prop allValuesFrom A

Property Restriction

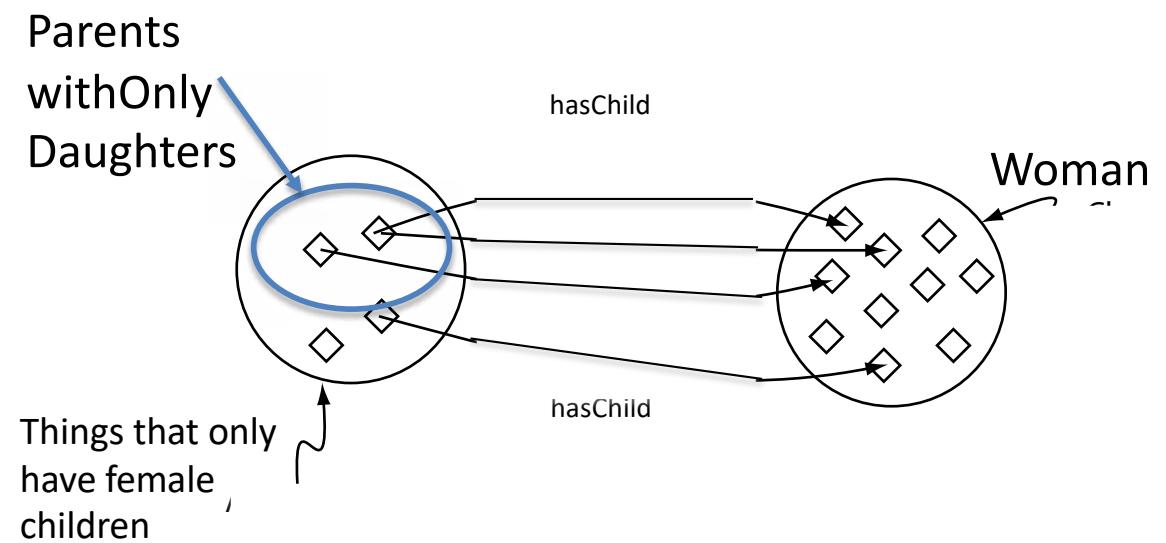
- A Mother is a Woman that has a child (some Person)
 - Set of objects that have a child (some Person)
 - Mother is a subclass



```
_ :a rdfs:subClassOf owl:Restriction .  
_ :a owl:onProperty ex:hasChild .  
_ :a owl:someValuesFrom ex:Person .  
ex:Mother rdfs:subClassOf _:a
```

Property Restriction

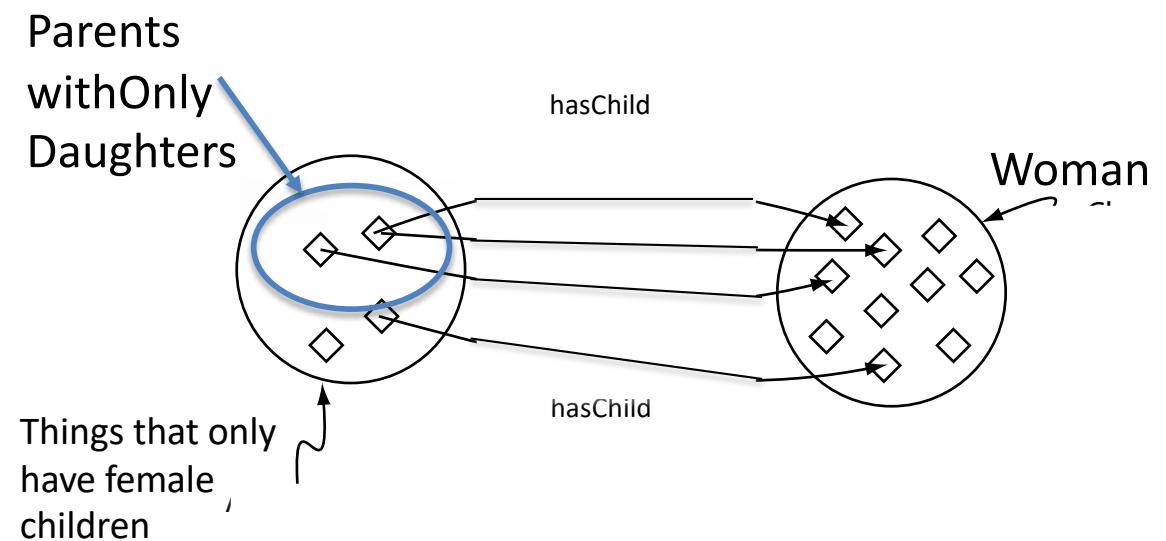
- Set of parents that only have female children (daughters)



```
_:a rdfs:subClassOf owl:Restriction .  
_:a owl:onProperty ex:hasChild .  
_:a owl:allValuesFrom ex:Woman .  
ex:ParentwithOnlyDaughters rdfs:subClassOf _:a
```

Property Restriction

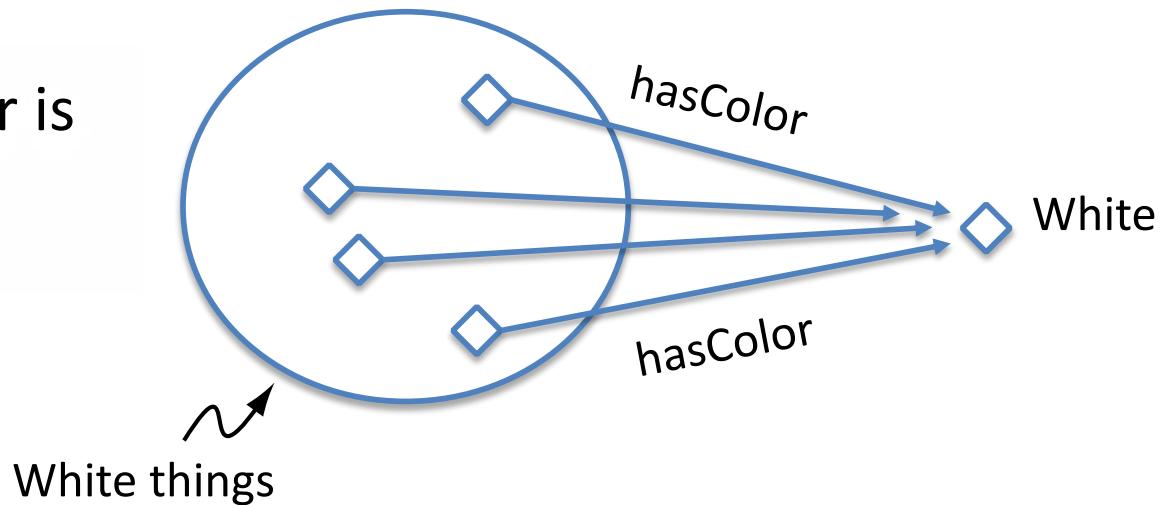
- Set of parents that only have female children (daughters)



```
_:a rdfs:subClassOf owl:Restriction .  
_:a owl:onProperty ex:hasChild .  
_:a owl:allValuesFrom ex:Woman .  
ex:ParentwithOnlyDaughters rdfs:subClassOf _:a
```

Property Restriction

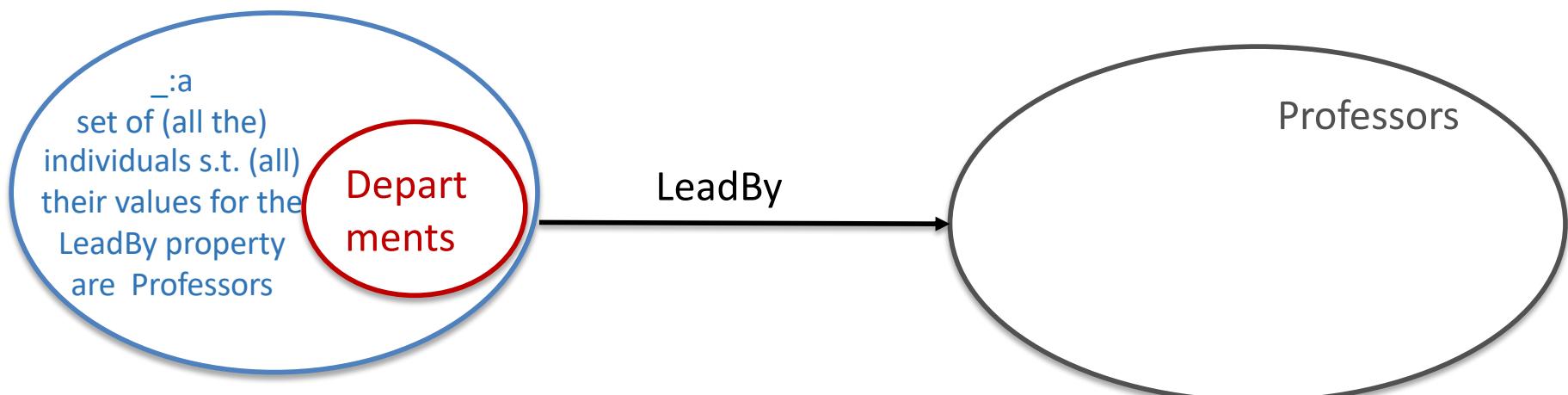
- Set of wines with color is white



```
_ :a rdfs:subClassOf owl:Restriction .  
_:a owl:onProperty ex:hasColor .  
_:a owl:hasValue ex:White .  
ex:WhiteWine rdfs:subClassOf _:a
```

Example

- Departments can be lead only by Professors

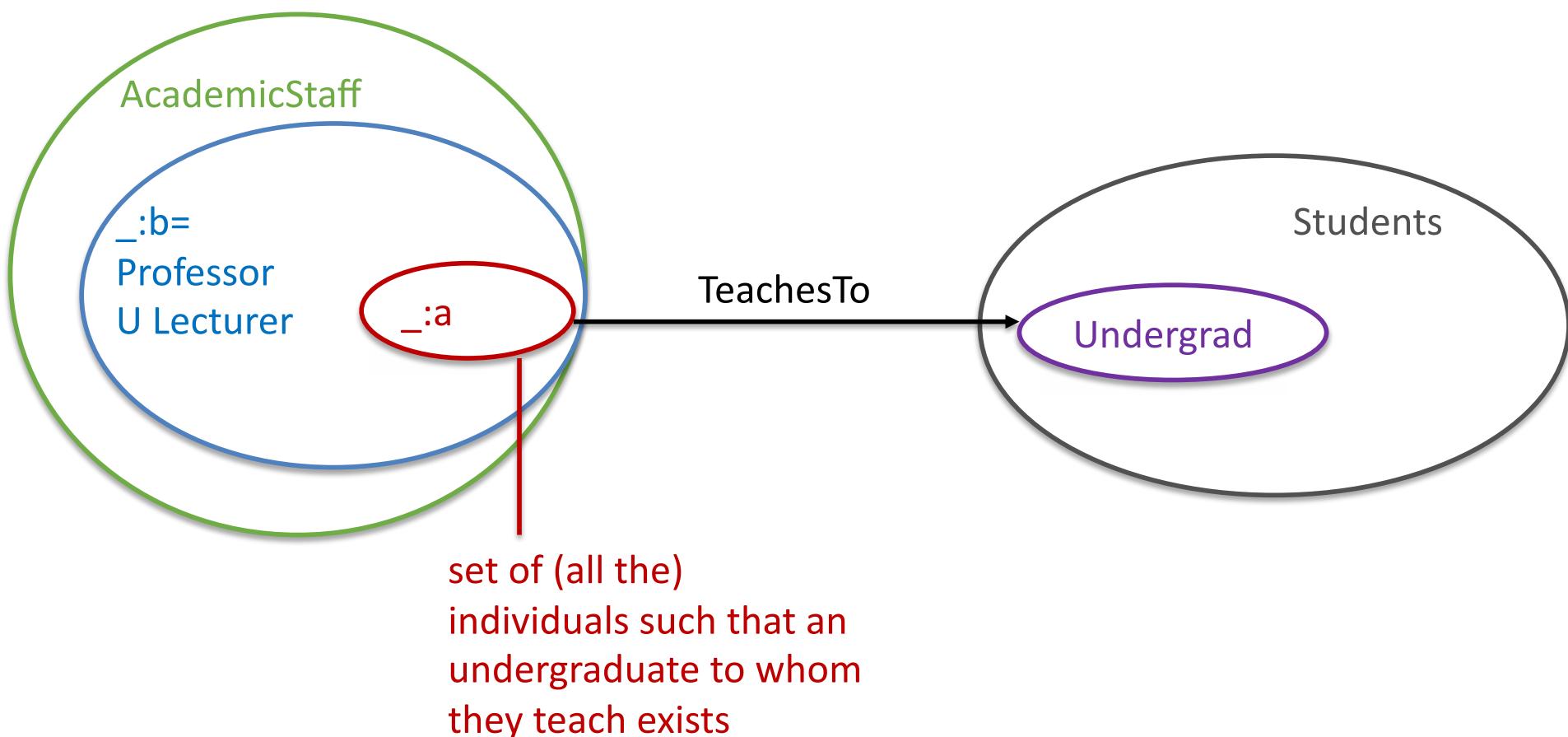


Example

- Departments can be lead only by professors
- Define the set of objects that are lead by professors
 - _a rdfs:subClassOf owl:Restriction
 - _a owl:onProperty LeadBy
 - _a owl:allValuesFrom Professor
- Now specify that all departments are lead by professors
Department rdfs:subClassOf _a

Example

- Only professors or lecturers can teach to undergraduate students



Example

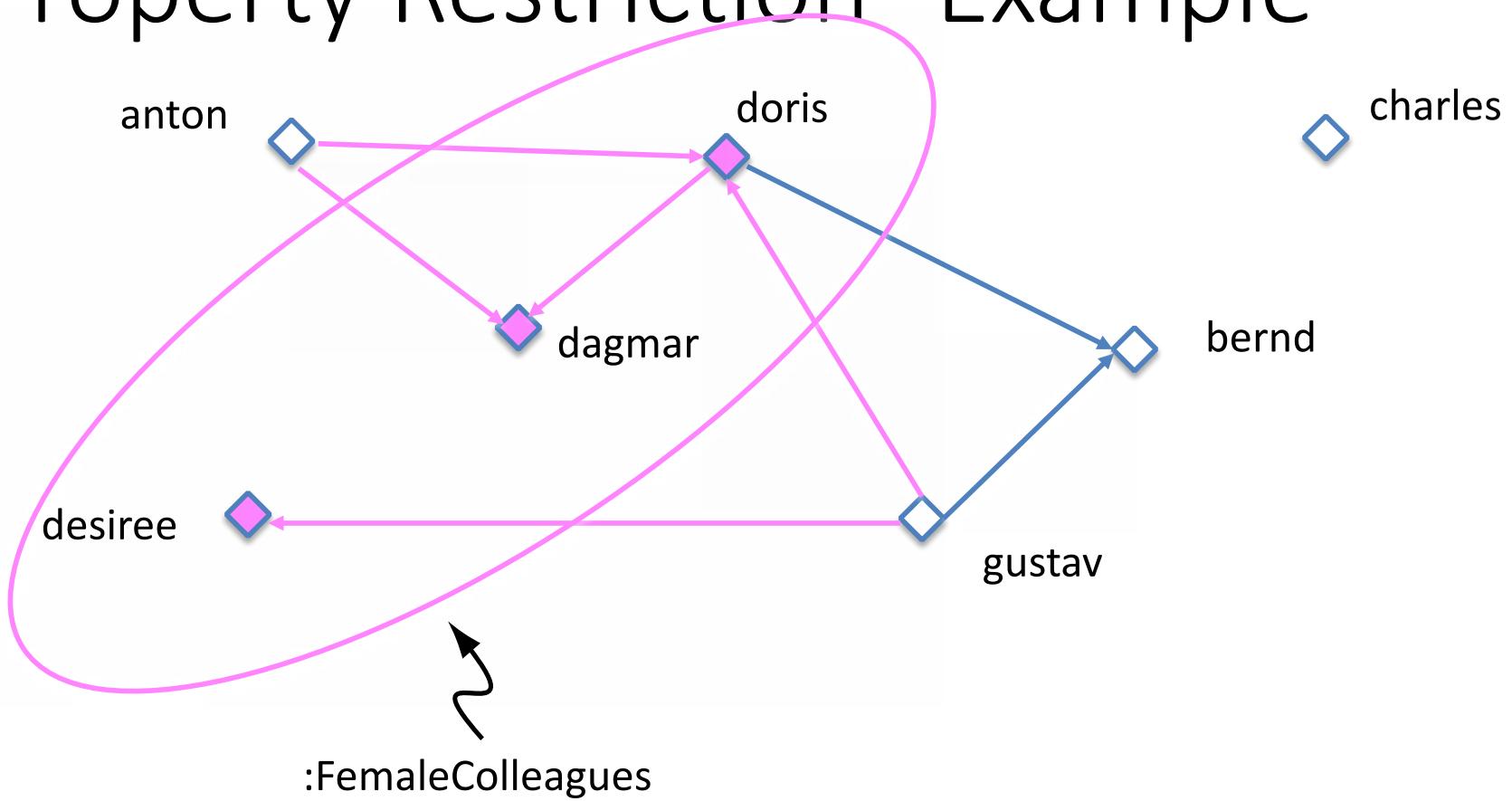
- only professors or lecturers may teach to undergraduate students

```
_a rdfs:subClassOf owl:Restriction
_a owl:onProperty TeachesTo
_a owl:someValuesFrom Undergrad
_b owl:unionOf (Professor, Lecturer)
_a rdfs:subClassOf _b
```

Property Restriction - Example

```
:anton a :Person;  
:likesToWorkWith :doris, :dagmar .  
  
:doris a :Person;  
:likesToWorkWith :dagmar, :bernd .  
  
:gustav a :Person;  
likesToWorkWith :bernd, :doris, :desiree .  
  
:charles a :Person .  
  
:FemaleColleagues owl:equivalentClass [  
rdf:type owl:Class ;  
owl:oneOf ( :dagmar :doris :desiree )].  
  
+ :anton owl:differentFrom :doris .  
... for all the person pairs ...
```

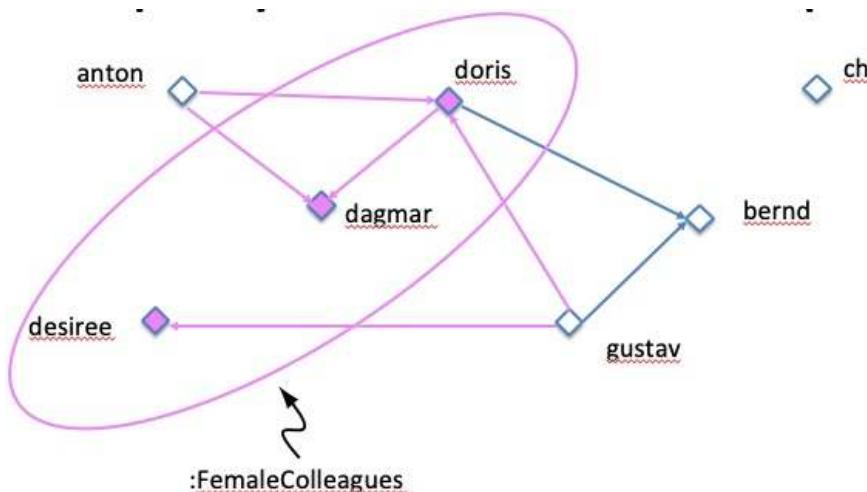
Property Restriction- Example



- All the properties are :likesToWorkWith properties

Property Restriction- Example

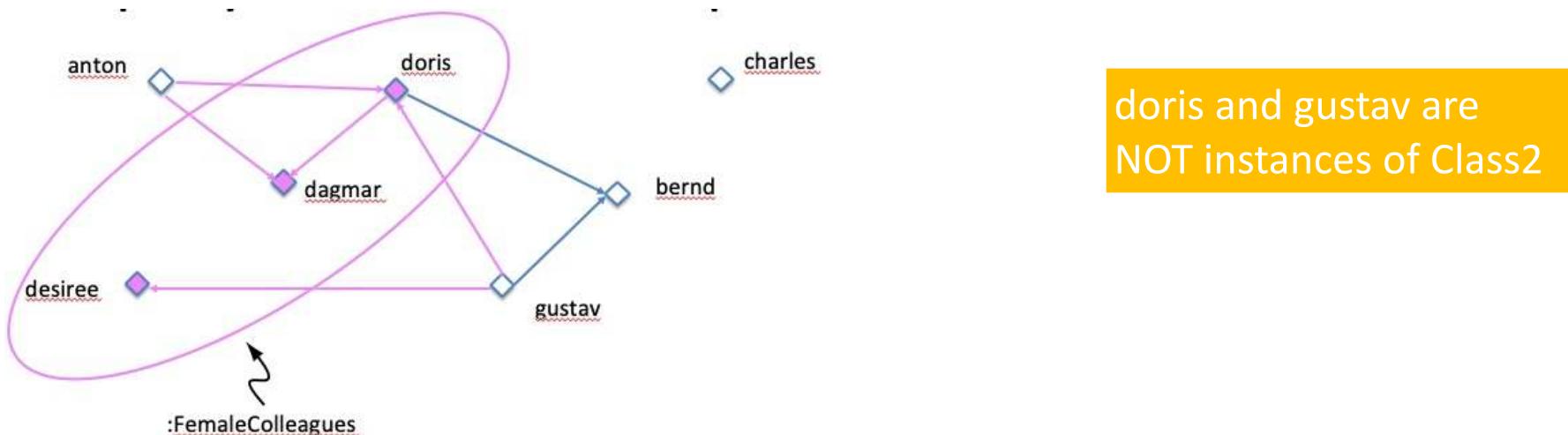
```
_:a1 rdfs:subClassOf owl:Restriction .  
_:a1 owl:onProperty :likesToWorkWith .  
_:a1 owl:someValuesFrom :FemaleColleagues  
.  
:Class1 owl:equivalentClass _:a1
```



anton
gustav
doris
are instances of Class1

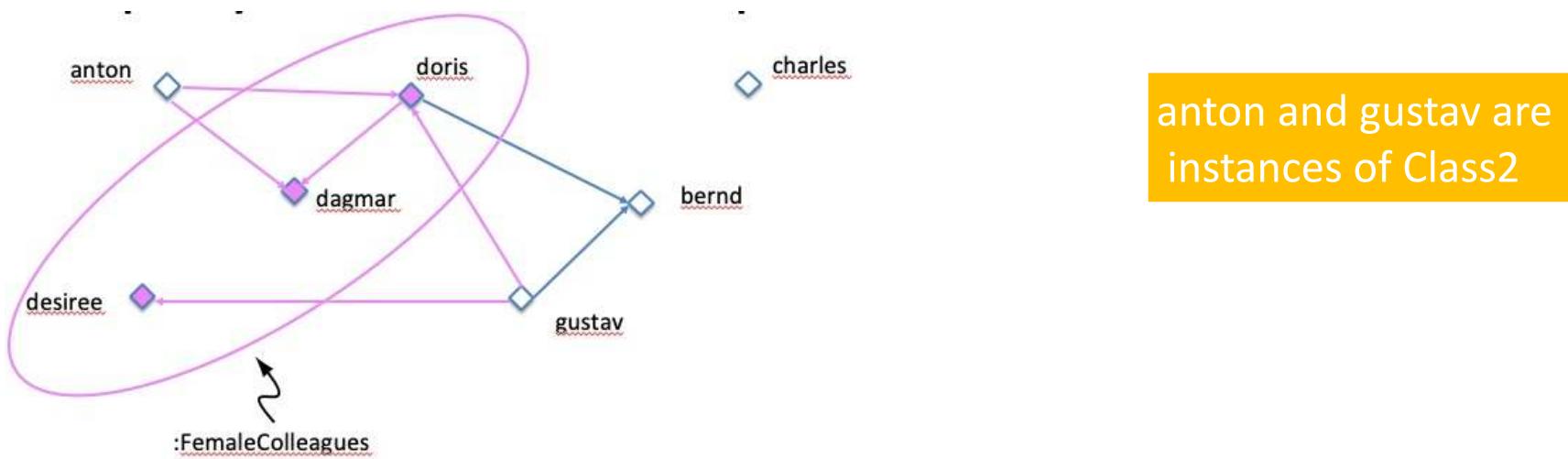
Property Restriction- Example

```
_:a2 rdfs:subClassOf owl:Restriction .  
_:a2 owl:onProperty :likesToWorkWith .  
_:a2 owl:allValuesFrom :FemaleColleagues .  
:Class2 owl:equivalentClass _:a2
```



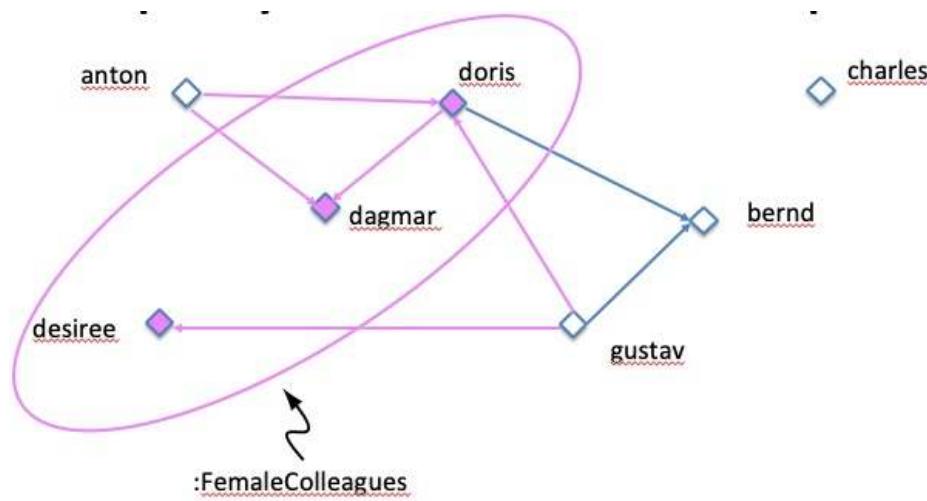
Property Restriction- Example

```
_:a3 rdfs:subClassOf owl:Restriction .  
_:a3 owl:onProperty :likesToWorkWith .  
_:a3 owl:hasValue :doris .  
:Class3 owl:equivalentClass _:a3
```



Property Restriction- Example

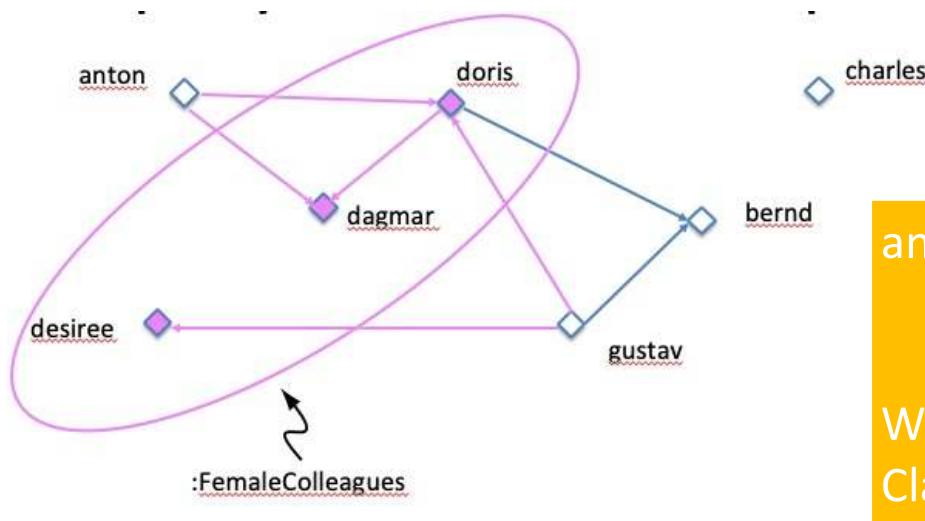
```
_:a4 rdfs:subClassOf owl:Restriction .  
_:a4 owl:onProperty :likesToWorkWith .  
_:a4 owl:minCardinality  
  
"3"^^xsd:nonNegativeInteger .  
:Class4 owl:equivalentClass _:a4
```



gustav is an
instances of Class2

Property Restriction - Example

```
_ :a5 rdfs:subClassOf owl:Restriction .  
_ :a5 owl:onProperty :likesToWorkWith .  
_ :a5 owl:maxCardinality  
  "0"^^xsd:nonNegativeInteger .  
:Class5 owl:equivalentClass _:a5
```



anton, doris, gustav are NOT part of Class5

We cannot infer that charles belongs to Class5 because of OWA

Cardinality Restrictions and UNA

- OWL does not use the Unique Name Assumption (UNA)
 - This means that different names may refer to the same individual
- Cardinality restrictions rely on ‘counting’ distinct individuals
- Consider class6 identical to class5 but with maxcardinality 2
- If we add anton to class6
 - Since we are stating that the individual anton is a member of the class of individuals that work with at the most two other individuals (people) doris and dagmar are inferred to be the same individual
- Rather than being viewed as an error, it will be inferred that two of the names refer to the same individual

Some basic modelling guidelines

- X must be Y, X is an Y that...

$$X \subseteq Y$$

X rdfs:subClassOf Y

- X is exactly Y, X is the Y that...

$$X \equiv Y$$

X owl:equivalentClass Y

- X is not Y

$$X \subseteq \neg Y$$

X owl:complementOf Y

(not the same as X is whatever it is not Y)

- X and Y are disjoint

$$X \cap Y \subseteq \perp$$

X owl:disjointWith Y

- X is Y or Z

$$X \subseteq Y \cup Z$$

X rdfs:subClass [
rdf:type owl:Class ;
owl:unionOf (Y,Z)]

Some basic DL modelling guidelines

- X is Y for which property P has only instances of Z as values

$$X \sqsubseteq Y \cap (\forall P.Z)$$

$_a$ rdfs:subClassOf owl:Restriction .
 $_a$ owl:onProperty P .
 $_a$ owl:allValuesFrom Z .
 X rdfs:subClassOf [
 rdf:type owl:Class ;
 owl:intersectionOf ($Y, _a$)]

- X is Y for which property P has at least an instance of Z as a value

$$X \sqsubseteq Y \cap (\exists P.Z)$$

- X is Y for which property P has at most 2 values

$$X \sqsubseteq Y \cap (\leq 2.P)$$

X rdf:type Y

- Individual X is a Y

$$X \in Y$$

X a Y

)OWL: Semantics and Reasoning

OWL Semantics

- Most of the OWL constructs come from Description Logics (DL)
- Therefore, we get for free all the positive and negative known results about reasoning in DLs

Ontologies and Description Logics

- First-order logic (FOL) is the formal foundation of the ontology languages for the Web
- First-order logic (also called predicate logic) is especially appropriate for knowledge representation and reasoning
 - ontologies are simply knowledge about classes and properties
 - from a logical point of view
 - classes are unary predicates
 - properties are binary predicates
 - constraints are logical formulas asserted as axioms on these predicates, i.e., asserted as true in the domain of interest

Ontologies and Description Logics

- Unfortunately, the implication problem in FOL is not decidable but only recursively enumerable:
 - an algorithm exists that given some formula F enumerates all the formulas G such that F implies G
 - no general algorithm exists that, when applied to two any input FOL formulas F and G , decides whether F implies G

Ontologies and Description Logics

- The simpler problem of deciding whether a FOL formula is satisfiable (i.e., there exists an assignment to the variables which make the formula true in the domain of interest) is also not decidable
- Said in other terms
 - an algorithm which takes a FOL formula F and returns yes if it is satisfiable and not if it not satisfiable does not exist

Ontologies and Description Logics

- **Issues:** exhibit fragments of FOL that are decidable, i.e., subsets of FOL formulas defined by some restrictions on the allowed formulas, for which checking logical entailment between formulas, possibly given a set of axioms, can be automatically performed by an algorithm
- **Description logics (DLs)** are decidable fragments of first-order logic allowing reasoning on complex logical axioms over unary and binary predicates
 - Exactly what is needed for handling ontologies

Ontologies and Description Logics

- DLs computational complexity depends on the set of constructs allowed in the language
- Research carried out on DLs provides a fine-grained analysis of the trade-off between expressive power and computational complexity of sound and complete reasoning
- **The semantics of RDF, RDFS, OWL can be described in terms of DL**

Ontologies in Description Logic Notation

- Classes and Instances

- $C(x)$ $\leftrightarrow x \text{ a } C .$
- $R(x,y)$ $\leftrightarrow x \text{ } R \text{ } y .$
- $C \sqsubseteq D$ $\leftrightarrow C \text{ rdfs:subClassOf } D$
- $C \equiv D$ $\leftrightarrow C \text{ owl:equivalentClass } D$
- $C \sqsubseteq \neg D$ $\leftrightarrow C \text{ owl:disjointWith } D$
- $C \equiv \neg D$ $\leftrightarrow C \text{ owl:complementOf } D$
- $C \equiv D \sqcap E \leftrightarrow C \text{ owl:intersectionOf } (D \text{ } E) .$
- $C \equiv D \sqcup E \leftrightarrow C \text{ owl:unionOf } (D \text{ } E) .$
- T $\leftrightarrow \text{owl:Thing}$
- \perp $\leftrightarrow \text{owl:Nothing}$

Ontologies in Description Logic Notation

- Domains, ranges, and restrictions
 - $\exists R.T \sqsubseteq C \leftrightarrow_R \text{rdfs:domain } C .$
 - $\forall R.C \leftrightarrow_R \text{rdfs:range } C .$
 $C \sqsubseteq \forall R.D \leftrightarrow_C \text{owl:subClassOf}$
[a owl:Restriction;
owl:onProperty R;
owl:allValuesFrom D] .
 - $C \sqsubseteq \exists R.D \leftrightarrow_C \text{owl:subClassOf}$
[a owl:Restriction;
owl:onProperty R;
owl:someValuesFrom D] .
 - $C \sqsubseteq \geq n R \leftrightarrow_C \text{owl:subClassOf}$
[a owl:Restriction;
owl:onProperty R;
owl:minCardinality n] .

Basic Inference Tasks

- ▶ Inconsistency of one class – Class has to be empty?
 $C \equiv \perp$
- ▶ Subsumption – Is C a subclass of D ?
 $C \sqsubseteq D$
- ▶ Equivalence of classes – Are two classes the same?
 $C \equiv D$
- ▶ Disjointness – Are two classes disjoint?
 $C \sqcap D \equiv \perp$
- ▶ Class membership – Is a an instance of C ?
 $C(a)$

OWL – Reasoning Tasks Revisited

- Proof method: Reductio ad absurdum
 - "Invent" an instance i
 - Check for contradictions
- E.g. Subclass Relations
 - Student subclass of Person „Every student is a person“
 - Define $\text{Student}(i)$ and $\neg\text{Person}(i)$
 - Check for contradictions
 - If there is one: Student subclass of Person has to hold
 - If there is none: Student subclass of Person cannot be derived
 - Note: it may still hold!
- All reasoning tasks can be reduced to the same basic tasks
 - i.e., consistency checking
- Tableaux algorithm

Tableaux Reasoning

- A tableaux reasoner shows the inconsistency (unsatisfiability) of a knowledge base (KB)
- Tableaux reasoners contain rules for handling all logical connectives and quantifiers
- Some rules cause the tableau to branch (divide into two alternatives)
- If any branch of a tableau leads to an evident contradiction, the branch closes
- If all branches close, the proof is complete, the KB is inconsistent (unsatisfiable)
- Else, if the algorithm finishes when no more rules can be applied, then the KB is consistent (satisfiable)

Inference Tasks as Unsatisfiability Problems

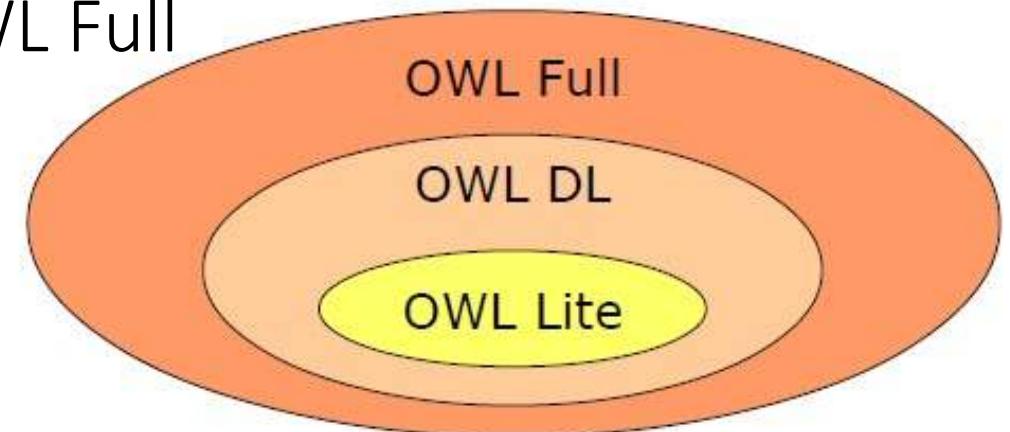
- ▶ Inconsistency of one class – Class has to be empty?
 $C \equiv \perp$ iff $KB \cup \{C(a)\}$ unsatisfiable (a is a new individual)
- ▶ Subsumption – Is C a subclass of D ?
 $C \sqsubseteq D$ iff $KB \cup \{(C \sqcap \neg D)(a)\}$ unsatisfiable (a new)
- ▶ Equivalence of classes – Are two classes the same?
 $C \equiv D$ iff $C \sqsubseteq D$ and $D \sqsubseteq C$
- ▶ Disjointness – Are two classes disjoint?
 $C \sqcap D \equiv \perp$ iff $KB \cup \{(C \sqcap D)(a)\}$ unsatisfiable (a new)
- ▶ Class membership – Is a an instance of C ?
 $C(a)$ iff $KB \cup \{\neg C(a)\}$ unsatisfiable

Tableaux Reasoners and Inference Tasks

- ▶ Our objective is to check whether a statement is true.
- ▶ To do this, we prove that the “reverse” of the statement cannot be satisfied.
- ▶ This means we take the KB we have and input the negation of the statement we want to prove.
- ▶ If the tableau ends with an inconsistent KB, the proof is complete and the original statement is true.
- ▶ If the tableau ends with a consistent KB, we have not been able to prove the truth of the statement.

OWL – Why so many variants?

- More expressive/powerful than RDF schema
- Trade-off:
 - Expressive power
 - Complexity of reasoning
 - Decidability
- Solution: different variants of OWL, e.g.,
 - OWL Lite, OWL DL, OWL Full
 - OWL2 profiles



OWL Full

OWL 2 Full

- All OWL features added on top of RDF(S)
 - Allows, e.g., *redefining the meaning of RDF(S) and OWL primitives*
- Advantages
 - *Fully upward compatible with RDF*
 - Any RDF document is an OWL 2 Full document
 - Any RDF(S) conclusion is an OWL 2 Full conclusion
 - *RDF-based semantics*
- Disadvantages
 - *Undecidable, as RDFS already has some very powerful modeling primitives*
 - *Complete and efficient reasoning not possible*

OWL DL

OWL 2 DL (Description Logic)

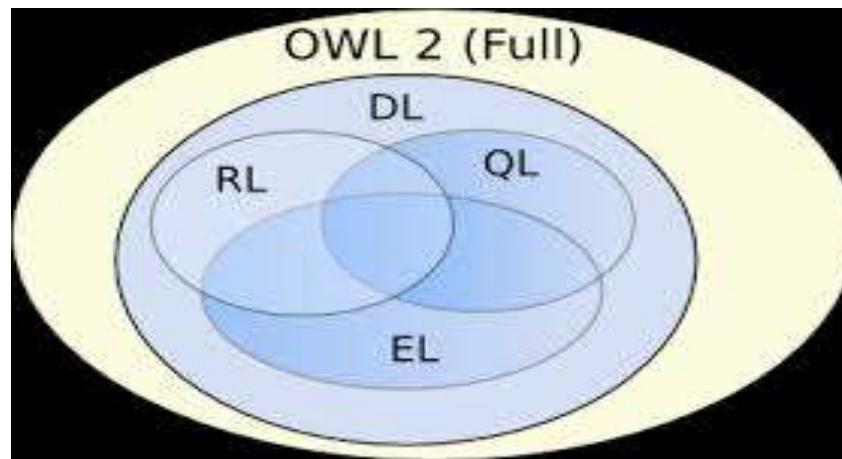
- Restricted form of OWL Full for which decidable, efficient support for reasoning is possible
 - *OWL 2 primitives cannot be applied to themselves*
 - *Only classes of non-literal resources considered*
 - *Strict separation between datatype and object properties*
 - *Strict separation between an individual, a class, or a property*
 - Using "punning" the same name may be used for different purposes, but treated as different views on the same IRI, interpreted semantically as if they were distinct
- Direct semantics, based on Description logics (Terminology logics)
 - *Subsets of predicate logic*
 - *But also RDF-based semantics can be applied to OWL 2 DL ontologies*
- Reasoning engines are available for DL
 - *Pellet, FaCT, RACER, Hermit*

OWL 2 Profiles

OWL 2 DL includes three specific profiles for different use cases

- OWL 2 EL
- OWL 2 QL
- OWL 2 RL

Each profile includes a subset of OWL DL features



OWL 2 EL

- Good for ontologies with lots of classes and/or properties
- Polynomial complexity of standard inference types: satisfiability, classification, instance checking
- Used for large scale class ontologies, e.g., Snomed CT
- Limitations include:
 - *Negation and disjunction not supported*
 - *Universal quantification on properties*
 - *E.g., “all children of a rich person are rich” cannot be stated*
 - *All kinds of role inverses are not available*
 - *E.g., parentOf and childOf cannot be stated as inverses*

OWL 2 QL

- Good for querying large numbers of individuals
- Relational Query Languages (conjunctive queries)
 - *Can be implemented efficiently using relational databases*
- Limitations include:
 - *Existential quantification of roles to a class expression*
 - *E.g., it can be stated that every person has a parent, but not that every person has a female parent*
 - *Property chain axioms and equality are not supported*

OWL 2 RL (Rule Language)

- Good for rule-based reasoning, database focus
- Can be implemented using logic programming
 - *First-order implications: IF certain triples exist THEN add additional triples*
 - See partial axiomatization of the OWL 2 RDF-based semantics as rules in the [OWL 2 Profiles](#) specification (Section 4.3)
- Limitations include:
 - *Statements where the existence of an individual enforces the existence of another individual*
 - E.g., the statement “every person has a parent” is not expressible
 - *Restricts class axioms asymmetrically*
 - *Constructs for a subclass cannot necessarily be used as a superclass*

Suggested Reading

- Pascal Hitzler, Markus Krötzsch and Sebastian Rudolph. Foundations of Semantic Web Technologies. Chapman & Hall/CRC, 2009. (Chapter 4)
- Matthew Horridge. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools. Edition 1.3.
- Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, Sebastian Rudolph. OWL 2 Web Ontology Language Primer (Second Edition), 2012

SCHEMAS: RDFS + AN INTRODUCTION TO OWL

Schema information in RDF

- RDF allows to model the instance level of an ontology
- On the other hand, RDF provides very simple mechanisms to constrain the instance level through ontology schema information

Which means have you already discussed to specify schema constraints?

Schema information in RDF

- RDF allows to model the instance level of an ontology
- On the other hand, RDF provides very simple mechanisms to constrain the instance level through ontology schema information
 - we can specify the type of a resource of a predicate (**rdf:type** as property)
 - we can specify that a given predicate is indeed a property (**rdf:Property** as object)
- modeling complex domains requires more sophisticated ontology schema information

RDF Typing

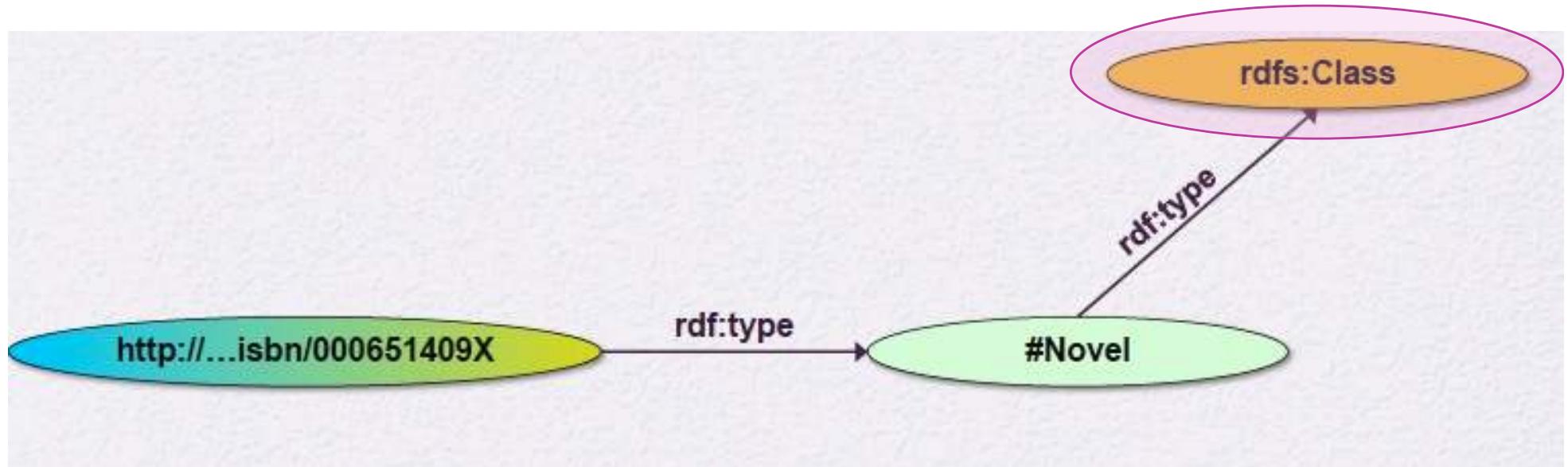
- Languages for specifying
 - constraints over individuals (subjects, objects) and predicates used in RDF
 - i.e., an ontology *schema* an RDF graph is an instance of
- Two main proposals
 - RDF Schema (RDFS)
 - Web Ontology Language (OWL)
- Different expressive power
- Bringing the **Semantics** to the Semantic Web!

RDFS - RESOURCE DESCRIPTION FRAMEWORK SCHEMA

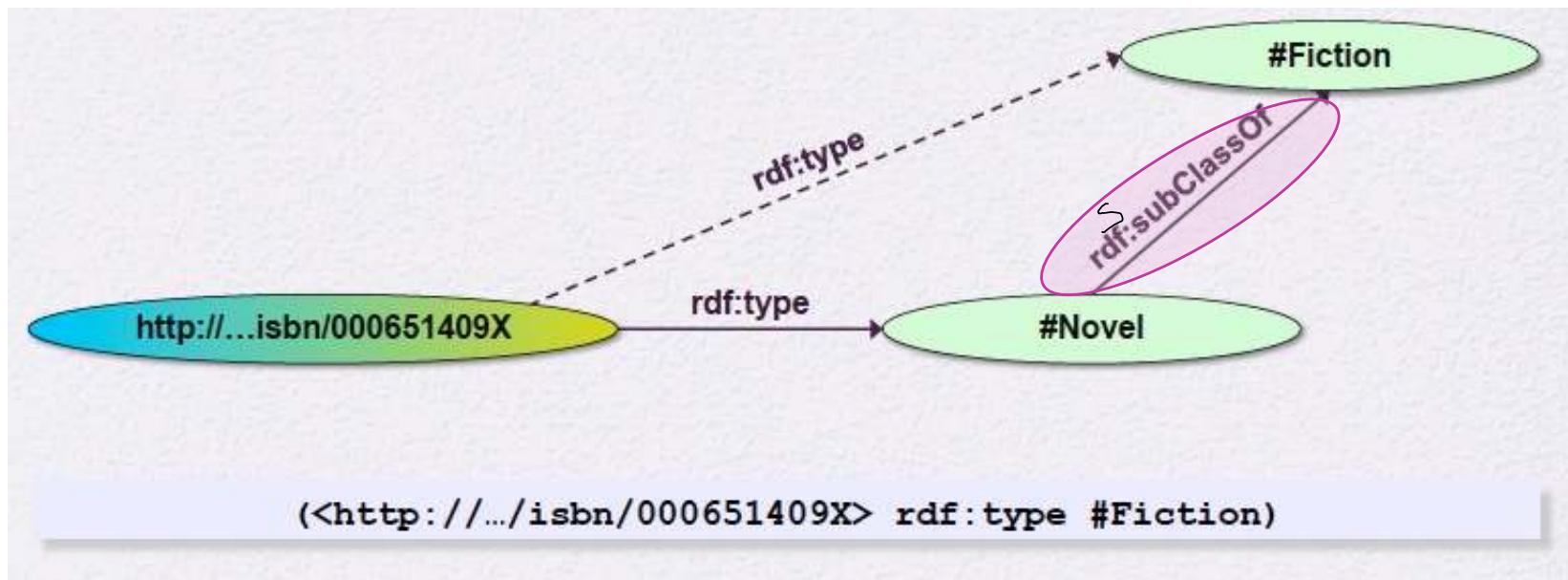
RDF Schema (RDFS)

- First step towards the “extra knowledge”
- Vocabulary (defining terms)
 - use the term “novel” (a class)
- Schema (defining types)
 - “The Glass Palace” is a novel
to be more precise: <<http://.../000651409X>> is a novel
- Taxonomy (defining hierarchies)
 - Any “novel” is a “fiction”
- RDFS defines resources and classes:
 - everything in RDF is a “resource”
 - “classes” are also resources, but...
 - ...they are also a collection of possible resources (i.e., “individuals” “fiction”, “novel”, ...)

Classes, resources in RDFS



Inferred properties IN RDFS



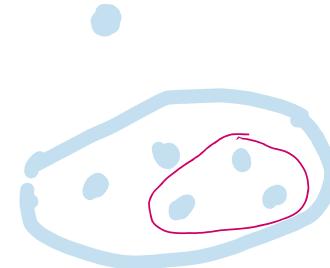
- is not in the original RDF data...
- ...but can be inferred from the RDFS rules
- RDFS environments return that triple, too

RDF Schema (RDFS)

- Type of constraints
 - *[objects and subjects as instances of certain classes]*
→ already in RDF, beyond the pure schema
 - **inclusion statements** between classes and between properties
 - **semantic relations** between the “domain”/the “range” of a property and some classes
- RDFS formalizes these notions as RDF triples, using some specific **properties** and **objects**

RDF Schema

- Declaration of instances (beyond the pure schema)
 - ▶ ⟨ Dupond **rdf:type** AcademicStaff ⟩



- Declaration of classes and subclass relationships
 - ▶ ⟨ Staff **rdf:type** rdfs:Class ⟩
 - ▶ ⟨ Java **rdfs:subClassOf** CSCourse ⟩

- Declaration of relations (properties in RDFS terminology)

- ▶ ⟨ RegisteredTo **rdf:type** rdf:Property ⟩

- Declaration of subproperty relationships

- ▶ ⟨ LateRegisteredTo **rdfs:subPropertyOf** RegisteredTo ⟩

- Declaration of domain and range restrictions for predicates

- ▶ ⟨ TeachesIn **rdfs:domain** AcademicStaff ⟩

subject

- ▶ ⟨ TeachesIn **rdfs:range** Course ⟩

object

- ▶ TeachesIn(AcademicStaff , Course)

Domain and range

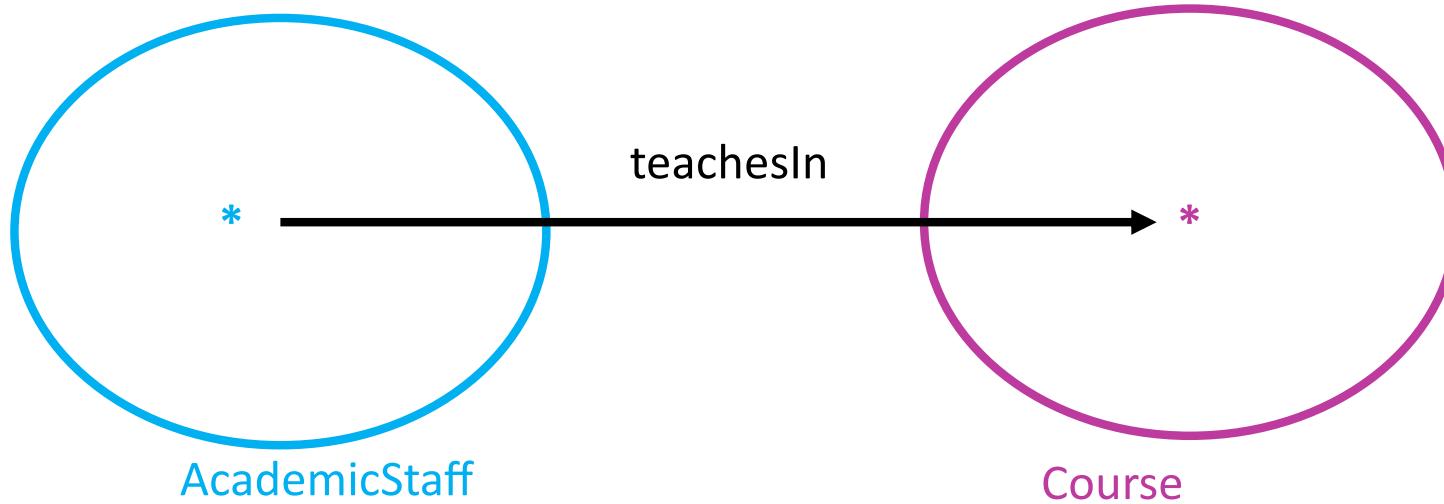
domain

range

s, teachesIn, o

AcademicStaff

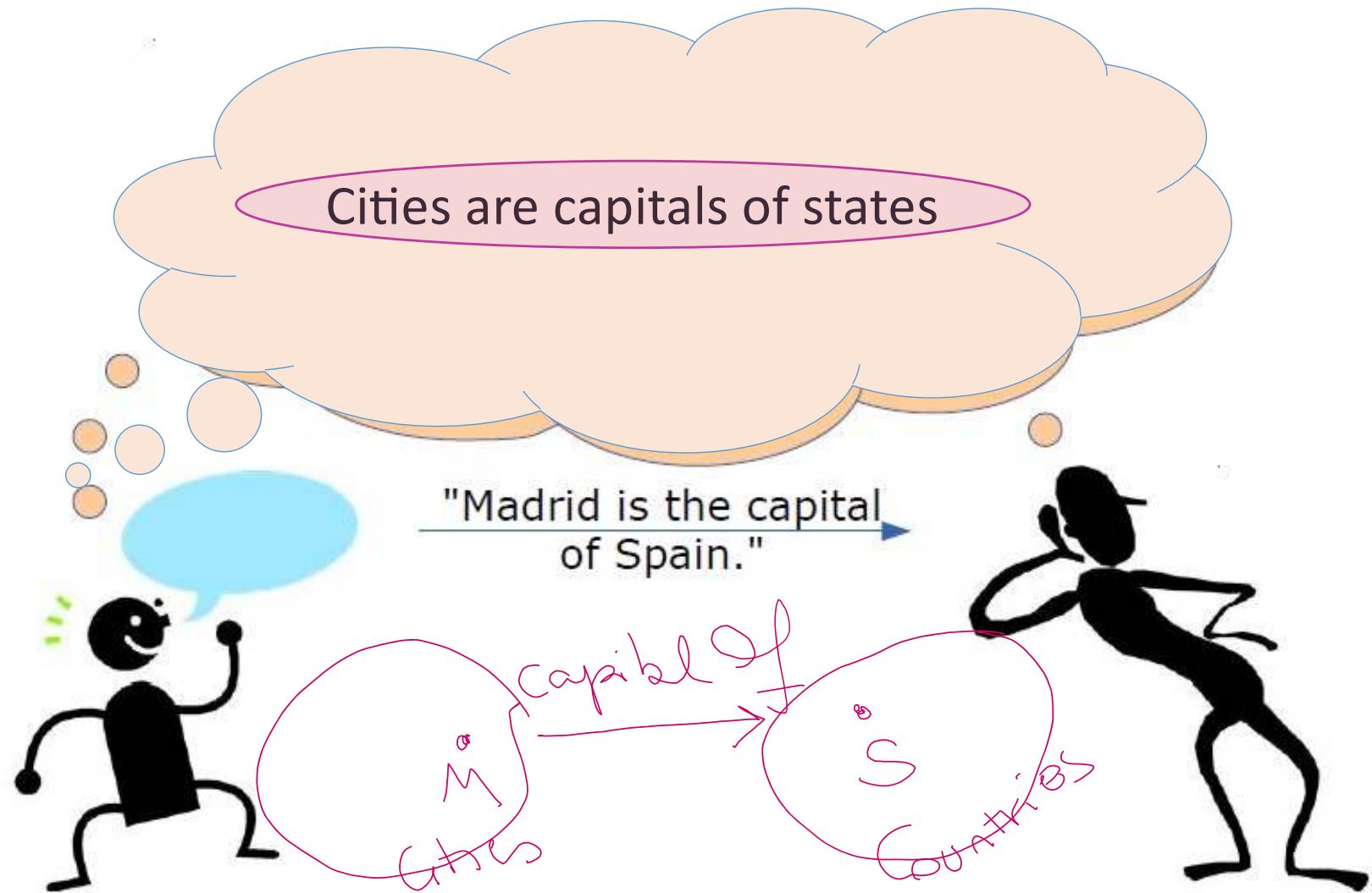
Course



A very simple example

- Let's look at that sentence:
 - "Madrid is the capital of Spain."
- Published on the Semantic Web (i.e., using RDF):
 - :Madrid :capitalOf :Spain .
- How many pieces of information can we (i.e., humans) derive from that sentence?

A very simple example – adding simple semantics (schema information)



A very simple example

- States, cities, and capitals

a is an alternative syntax
(qname, shorthand) for
rdf:type

```
:State a rdfs:Class .  
:City a rdfs:Class .  
:locatedIn a rdf:Property .  
:capitalOf rdfs:subPropertyOf :locatedIn .  
:capitalOf rdfs:domain :City .  
:capitalOf rdfs:range :State .  
  
:Madrid :capitalOf :Spain .
```

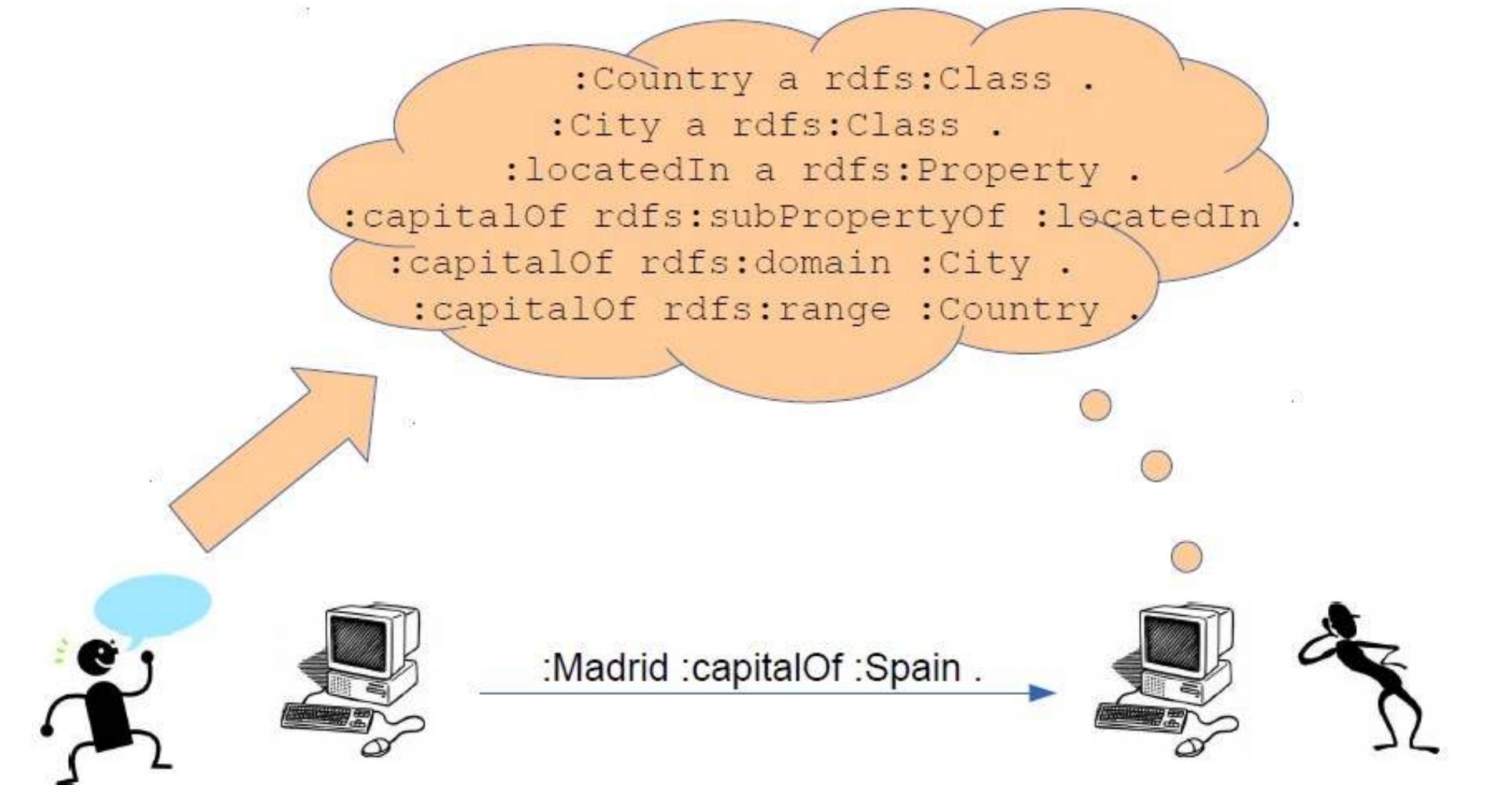
Definition of the Terminology (T-Box)
RDFS

Definition of the Assertions (A-Box)
RDF

RDFS Logical Semantics

| | RDF and RDFS statements | FOL translation |
|---|---|---|
| | $\langle i \text{ rdf:type } C \rangle$ | $C(i)$ |
| | $\langle i P j \rangle$ | $P(i,j)$ |
| $\forall \dots (\dots \Rightarrow \dots)$ | $\langle C \text{ rdfs:subClassOf } D \rangle$ | $\forall X (C(X) \Rightarrow D(X))$ |
| | $\langle P \text{ rdfs:subPropertyOf } R \rangle$ | $\forall X \forall Y (P(X, Y) \Rightarrow R(X, Y))$ |
| | $\langle P \text{ rdfs:domain } C \rangle$ | $\forall X \forall Y (P(X, Y) \Rightarrow C(X))$ |
| | $\langle P \text{ rdfs:range } D \rangle$ | $\forall X \forall Y (P(X, Y) \Rightarrow D(Y))$ |

What do we gain now?



What do we gain now?

| $\langle P \text{ rdfs:domain } C \rangle$ | $\mid \forall X \forall Y (P(X, Y) \Rightarrow C(X)) \mid$ |
|---|---|
| $:Madrid :capitalOf :Spain .$ | |
| <u>+ :capitalOf rdfs:domain :City</u> | |
| $\rightarrow :Madrid \text{ a } :City .$ | |
| $\langle P \text{ rdfs:range } D \rangle$ | $\mid \forall X \forall Y (P(X, Y) \Rightarrow D(Y)) \mid$ |
| $:Madrid :capitalOf :Spain .$ | |
| <u>+ :capitalOf rdfs:range :Country</u> | |
| $\rightarrow :Spain \text{ a } :Country .$ | |
| $:Madrid :capitalOf :Spain .$ | |
| <u>+ :capitalOf rdfs:subPropertyOf :locatedIn</u> | <u>$\mid \forall X \forall Y (P(X, Y) \Rightarrow R(X, Y)) \mid$</u> |
| $\rightarrow :Madrid :locatedIn :Spain .$ | |

RDFS Operational Semantics

- The logical formulas representing the semantics of RDFS are very useful in practice and are very adapted to deductive reasoning on RDF
- This means that
 - given facts and rules
 - we can derive new facts
- The corresponding tools are called **reasoners**
- The logical formulas can be interpreted as rules (**tuple generating dependencies**) that may be thought of as a factory for generating new facts

RDFS Operational Semantics

- Deduction rules are an interpretation function
- Simple reasoning algorithm (a.k.a. *forward chaining*):

```
Given: an RDF Graph G
a set of deduction rules R
Entailment E = G
Repeat
    M := { }
    For all rules in R
        For each statement S in G
            Apply R to S
            If E does not contain consequence
                Add consequence to M
    Add all elements in M to E
bis M = { }
```

RFDS Operational semantics: Example

- if $\langle r \text{ rdf:type } A \rangle$ and $\langle A \text{ rdfs:subClassOf } B \rangle$
then $\langle r \text{ rdf:type } B \rangle$
where r, A, and B are variables
- This means that:
 - if we know two triplets matching the patterns
 - $\langle r \text{ rdf:type } A \rangle$ and
 - $\langle A \text{ rdfs:subClassOf } B \rangle$
 - for some values of r, A, B, then
 - we can infer the triplet $\langle r \text{ rdf:type } B \rangle$ with the values of r, B taken to be those of the match

RDFS Operational Semantics

1. if $\langle r \text{ rdf:type } A \rangle$ and $\langle A \text{ rdfs:subClassOf } B \rangle$
then $\langle r \text{ rdf:type } B \rangle$

2. if $\langle r \text{ P } s \rangle$ and $\langle P \text{ rdfs:subPropertyOf } Q \rangle$
then $\langle r \text{ Q } s \rangle$

3. if $\langle P \text{ rdfs:domain } C \rangle$ and $\langle x \text{ P } y \rangle$
then $\langle x \text{ rdf:type } C \rangle$

4. if $\langle P \text{ rdfs:range } D \rangle$ and $\langle x \text{ P } y \rangle$
then $\langle y \text{ rdf:type } D \rangle$

A very simple example

- States, cities, and capitals

```
:State a rdfs:Class .  
:City a rdfs:Class .  
:locatedIn a rdf:Property .  
:capitalOf rdfs:subPropertyOf :locatedIn .  
:capitalOf rdfs:domain :City .  
:capitalOf rdfs:range :State .  
  
:  
:Madrid :capitalOf :Spain .  
  
:Madrid a :City  
:Spain a :Country  
  
:Madrid :locatedIn :Spain
```

The diagram illustrates the classification of the provided statements into four categories:

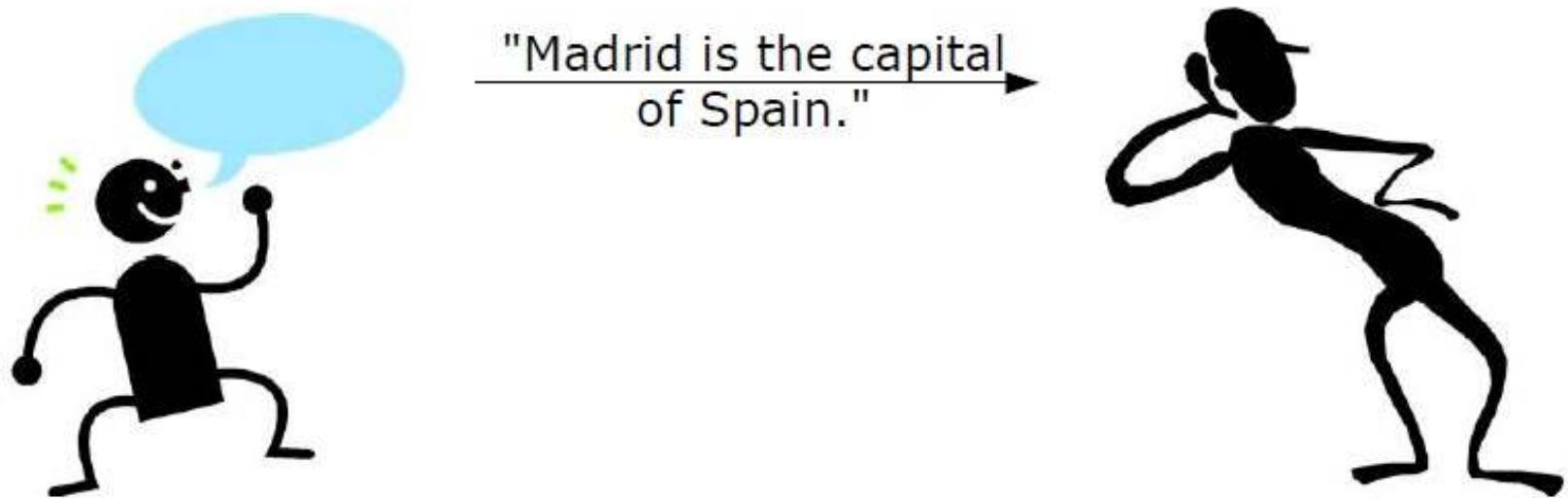
- Definition of the Terminology (T-Box):** This category is represented by a blue brace on the right side of the first five statements. It includes the definitions of the classes `:State`, `:City`, and the properties `:locatedIn` and `:capitalOf`.
- RDFS:** This category is represented by a blue brace on the right side of the last three statements. It includes the statement that `:capitalOf` is a subproperty of `:locatedIn`, and the domain and range declarations for `:capitalOf`.
- Definition of the Assertions (A-box):** This category is represented by a blue brace on the right side of the statement `:Madrid :capitalOf :Spain .`
- RDF:** This category is represented by a blue brace on the right side of the statements `:Madrid a :City` and `:Spain a :Country`.
- Inferred Assertions:** This category is represented by a blue brace on the right side of the inferred statement `:Madrid :locatedIn :Spain`, which is derived from the existing assertions.

OWL

WEB ONTOLOGY LANGUAGE

What is missing up to now?

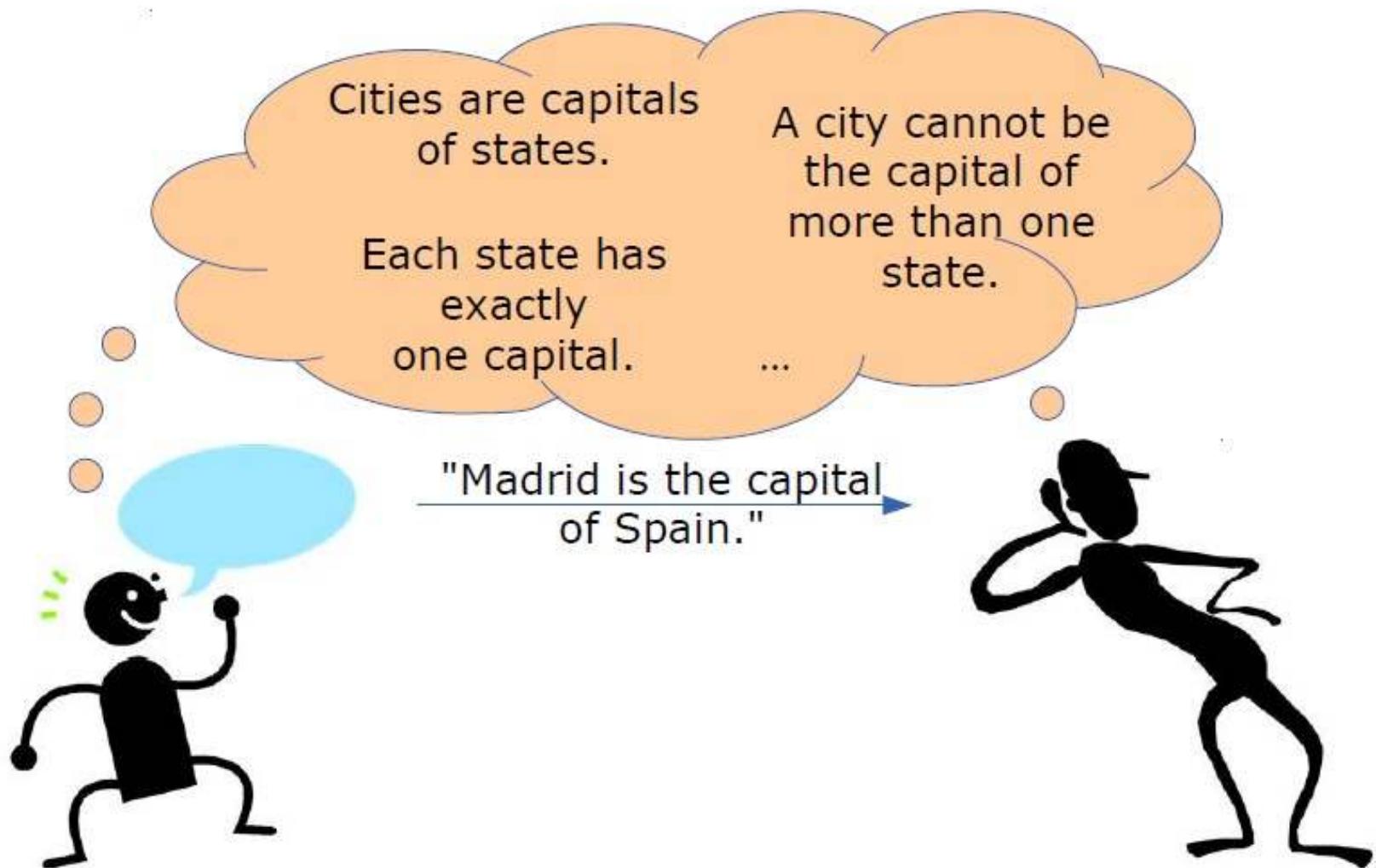
- Our mission: make computers understand information on the Web
- But what does *understand* actually mean?



Semantics

- Let's look at that sentence:
 - "Madrid is the capital of Spain."
- Published on the Semantic Web (i.e., using RDF):
 - :Madrid :capitalOf :Spain .
- How many pieces of information can we (i.e., humans) derive from that sentence?
 - (1 piece of information = 1 statement <S,P,O>)

Semantics – How does it work?



Semantics

- Let's look at that sentence:
 - "Madrid is the capital of Spain."
- We can get the following information:
 - "Madrid is the capital of Spain."
 - "Spain is a state."
 - "Madrid is a city."
 - "Madrid is located in Spain."
 - "Barcelona is not the capital of Spain."
 - "Madrid is not the capital of France."
 - "Madrid is not a state."
 - ...

What have we gained?

- Let's look at that sentence:
 - "Madrid is the capital of Spain."
- We can get the following information:
 - "Madrid is the capital of Spain." ✓
 - "Spain is a state." ✓
 - "Madrid is a city." ✓
 - "Madrid is located in Spain." ✓
 - "Barcelona is not the capital of Spain." ✗
 - "Madrid is not the capital of France." ✗
 - "Madrid is not a state." ✗
 - ...

What we cannot express (up to now)?

- "Every state has *exactly one* capital"
 - Property cardinalities
- "Every city can only be the capital of one state."
 - Functional properties
- "A city cannot be a state at the same time."
 - Class disjointness
- ...
- For those, we need more expressive languages than RDFS!

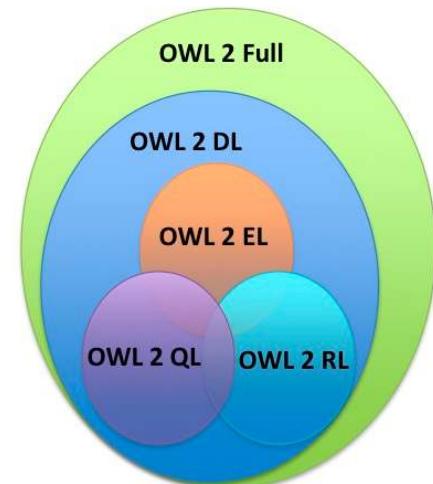
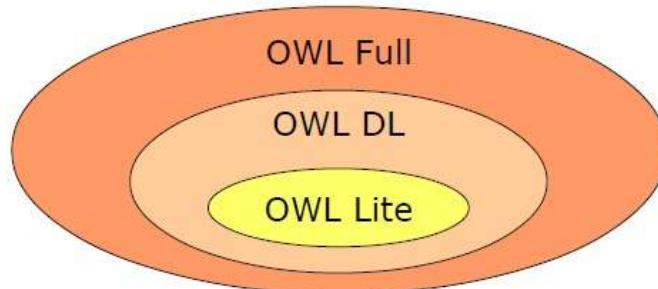
Expressive Ontologies using OWL

OWL extends RDFS with the possibility to express additional constraints

OWL was designed to find a reasonable balance between

- expressivity of the language and
- efficient reasoning, i.e. scalability

This also motivates the different sublanguages



Expressive Ontologies using OWL

- "Barcelona is not the capital of Spain." ✗
- Why not?
 - Countries have exactly one capital
 - Barcelona and Madrid are not the same
- In OWL:

```
:capitalOf a owl:InverseFunctionalProperty .  
:Madrid :capitalOf :Spain .  
:Madrid owl:differentFrom :Barcelona .
```

```
ASK { :Barcelona :capitalOf :Spain . } → false
```

Expressive Ontologies using OWL

- "Madrid is not the capital of France."
- Why not?
 - A city can only be the capital of one country
 - Spain and France are not the same
- Also:

```
:capitalOf a owl:FunctionalProperty .  
:Madrid :capitalOf :Spain .  
:Spain owl:differentFrom :France .  
  
ASK { :Madrid :capitalOf :France . } → false
```

Expressive ontologies using OWL

- Madrid is not a state 
- Why not?
 - A capital is a city and a city cannot be a state (states and cities are disjoint)

```
:Madrid :capitalOf :Spain .  
:capitalOf rdfs:domain :City .  
:City owl:disjointWith :State .
```

```
ASK { :Madrid :a :State .} -> false
```

What is OWL?

- Standard language (W3C) for representing vocabularies/ ontologies/schemas
- Much richer than RDF Schema and SKOS (W3C recommendations for light-weight vocabularies)
- Original OWL
 - *Published as W3C recommendation on 10.2.2004*
- OWL 2
 - *Latest W3C recommendation on 11.12.2012*
Extends and replaces the old recommendation

OWL 2 Syntaxes

Turtle `ex:JohnSmith rdf:type ex:Person .`

RDF/XML `<ex:Person rdf:about="#JohnSmith"/>`

OWL/XML `<ClassAssertion>
 <Class IRI="Person" />
 <NamedIndividual IRI="JohnSmith" />
</ClassAssertion>`

Functional-style `ClassAssertion(:Person :JohnSmith)`

Manchester Individual: `JohnSmith`
Types: `Person`

Many examples, translated into all syntaxes:
OWL 2 Web Ontology Language: Primer
<http://www.w3.org/TR/owl2-primer/>

«Schema knowledge» in RDFS

rdf:type

rdf:Property

rdfs:Class

rdfs:range

rdfs:domain

rdfs:subClassOf

rdfs:subPropertyOf

«Schema knowledge» in RDFS

```
ex:isMarriedTo rdfs:domain ex:Person .  
ex:isMarriedTo rdfs:range ex:Person .  
ex:instituteAIFB rdf:type ex:Institution
```

```
ex:pascal ex:isMarriedTo ex:instituteAIFB
```

- It likely is a modeling flaw
- It would result in rejecting the insertion in a database
- In RDFS it results in inferring

```
ex:instituteAIFB rdf:type ex:Person
```

- Multiple class membership is allowed
- No inconsistency

RDFS (and OWL) Semantics

- **Open World Assumption** The absence of a triple in a graph does not imply that the corresponding statement does not hold
- The fact that a statement is not true cannot be described in RDFS
- No **Unique Name Assumption**: differently named individuals can denote the same thing

The OWL View of Life

- OWL is not like a database system
- no requirement that the only properties of an individual are those mentioned in a class it belongs to
- no assumption that everything is known
- classes and properties can have multiple “definitions”
- statements about individuals need not be together (in the same document)

OWL Constructs

- Class (&instance) relations
- Property relations & characteristics
- Combining classes (set opns)
- Property restrictions and intensional classes

Class Relations

- Disjointness between classes:

| OWL notation | FOL translation |
|---|---|
| <code><C owl:disjointWith D></code> | $\forall X(C(X) \Rightarrow \neg D(X))$ |

inconsistency

ex:isMarriedTo rdfs:domain ex:Person .
ex:isMarriedTo rdfs:range ex:Person .
ex:instituteAIFB rdf:type ex:Institution .
ex:Institution owl:disjointWith ex:Person

ex:pascal ex:isMarriedTo ex:instituteAIFB

ex:instituteAIFB rdf:type ex:Person



Class Relations (back to our initial example)

- Madrid is not a state 
- Why not?
 - A capital is a city and a city cannot be a state (states and cities are disjoint)

```
:Madrid :capitalOf :Spain .  
:capitalOf rdfs:domain :City .  
:City owl:disjointWith :State .
```

```
ASK { :Madrid :a :State .} -> false
```

Other class & instance relations

- owl:equivalentClass
 - owl:sameAs
 - owl:differentFrom
- Needed because of lack of
Unique Name Assumption

Property Relations

- Constraints of functionality and symmetry on predicates:

| OWL notation | FOL translation |
|--|--|
| $\langle P \text{ rdf:type owl:FunctionalProperty} \rangle$ | $\forall X \forall Y \forall Z (P(X, Y) \wedge P(X, Z) \Rightarrow Y = Z)$ |
| $\langle P \text{ rdf:type owl:InverseFunctionalProperty} \rangle$ | $\forall X \forall Y \forall Z (P(X, Y) \wedge P(Z, Y) \Rightarrow X = Z)$ |
| $\langle P \text{ owl:inverseOf } Q \rangle$ | $\forall X \forall Y (P(X, Y) \Leftrightarrow Q(Y, X))$ |
| $\langle P \text{ rdf:type owl:SymmetricProperty} \rangle$ | $\forall X \forall Y (P(X, Y) \Rightarrow P(Y, X))$ |

owl:TransitiveProperty

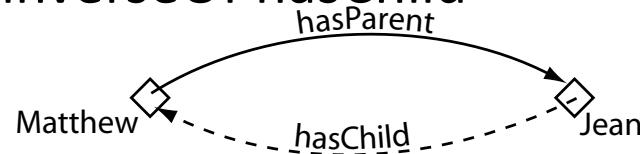
...

owl:equivalentProperty

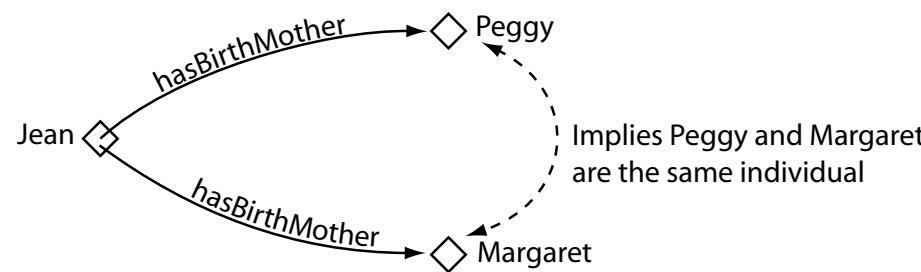
owl:propertyDisjointWith

Property Relations

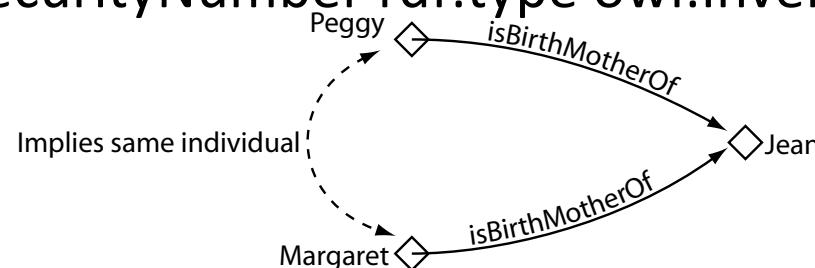
- hasParent owl:inverseOf hasChild



- hasBirthMother rdf:type owl:FunctionalProperty



- hasSocialSecurityNumber rdf:type owl:InverseFunctionalProperty

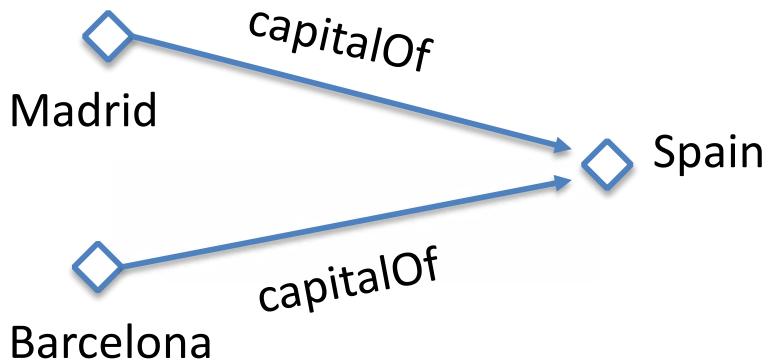


Property Relations (back to our initial example)

- "Barcelona is not the capital of Spain." **X**
- Why not?
 - Countries have exactly one capital
 - Barcelona and Madrid are not the same
- In OWL:

```
:capitalOf a owl:InverseFunctionalProperty .  
:Madrid :capitalOf :Spain .  
:Madrid owl:differentFrom :Barcelona .
```

ASK { :Barcelona :capitalOf :Spain . } → false



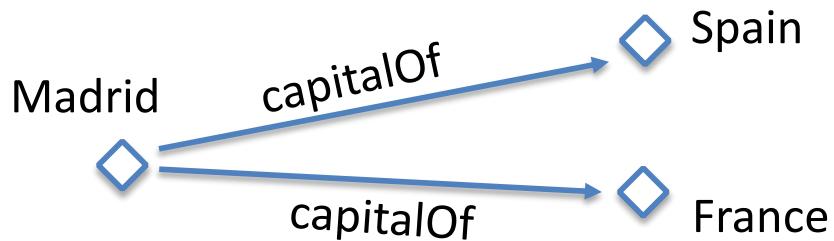
inverseFunctionality leads to the fact that any city capital of Spain is the same thing as Madrid (Barcelona = Madrid)
If we add owl:differentFrom we get an inconsistency

Property Relations (back to our initial example)

- "Madrid is not the capital of France." ✗
- Why not?
 - A city can only be the capital of one country
 - Spain and France are not the same
- Also:

```
:capitalOf a owl:FunctionalProperty .  
:Madrid :capitalOf :Spain .  
:Spain owl:differentFrom :France .
```

```
ASK { :Madrid :capitalOf :France . } → false
```



Functionality leads to the fact that any country which capital is Madrid is the same thing as Spain (France = Spain)
If we add owl:differentFrom we get an inconsistency

Key Notions

- An ontology is a clear, unambiguous, formal model of some part of the real world that is shared by several people
- OWL can be used to describe ontologies
- OWL adds to RDFS the possibility to
 - clarify that two individuals are identical or different
 - express that two classes are equivalent or disjoint
 - declare a property to be the inverse of another property
 - construct new classes by taking the union, intersection or complement of other existing classes
 - define classes by specifying restrictions on the cardinality or possible values of properties
- OWL is based on description logics, which makes its semantics machine accessible

Suggested Reading

- Pascal Hitzler, Markus Krötzsch and Sebastian Rudolph. Foundations of Semantic Web Technologies. Chapman & Hall/CRC, 2009. (Chapter 4)
- Matthew Horridge. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools. Edition 1.3.
- Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, Sebastian Rudolph. OWL 2 Web Ontology Language Primer (Second Edition), 2012

Semantic data systems

Data storage and processing
(no reasoning)

RDF DATA STORAGE

RDF data store

**RDF as an
interface
format**



RDF Enabled Systems
RDF is not the logical model

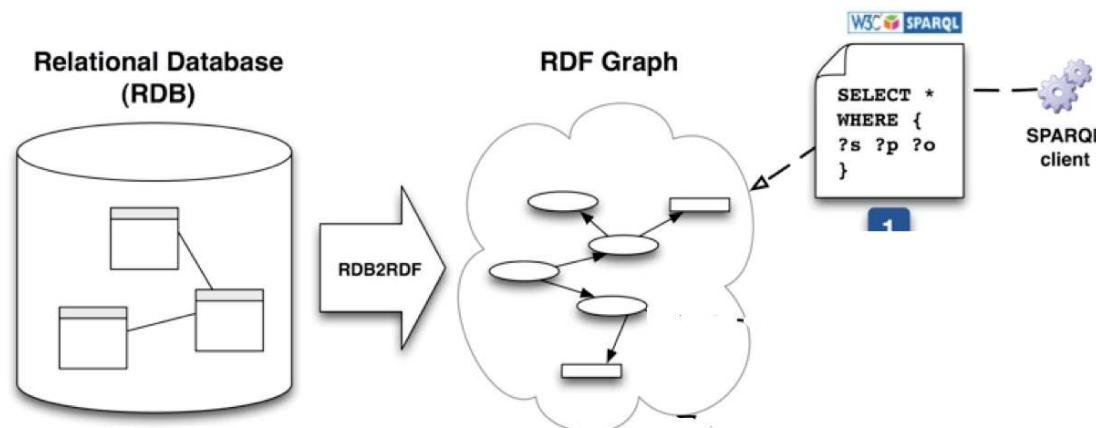
**RDF as a
storage format**



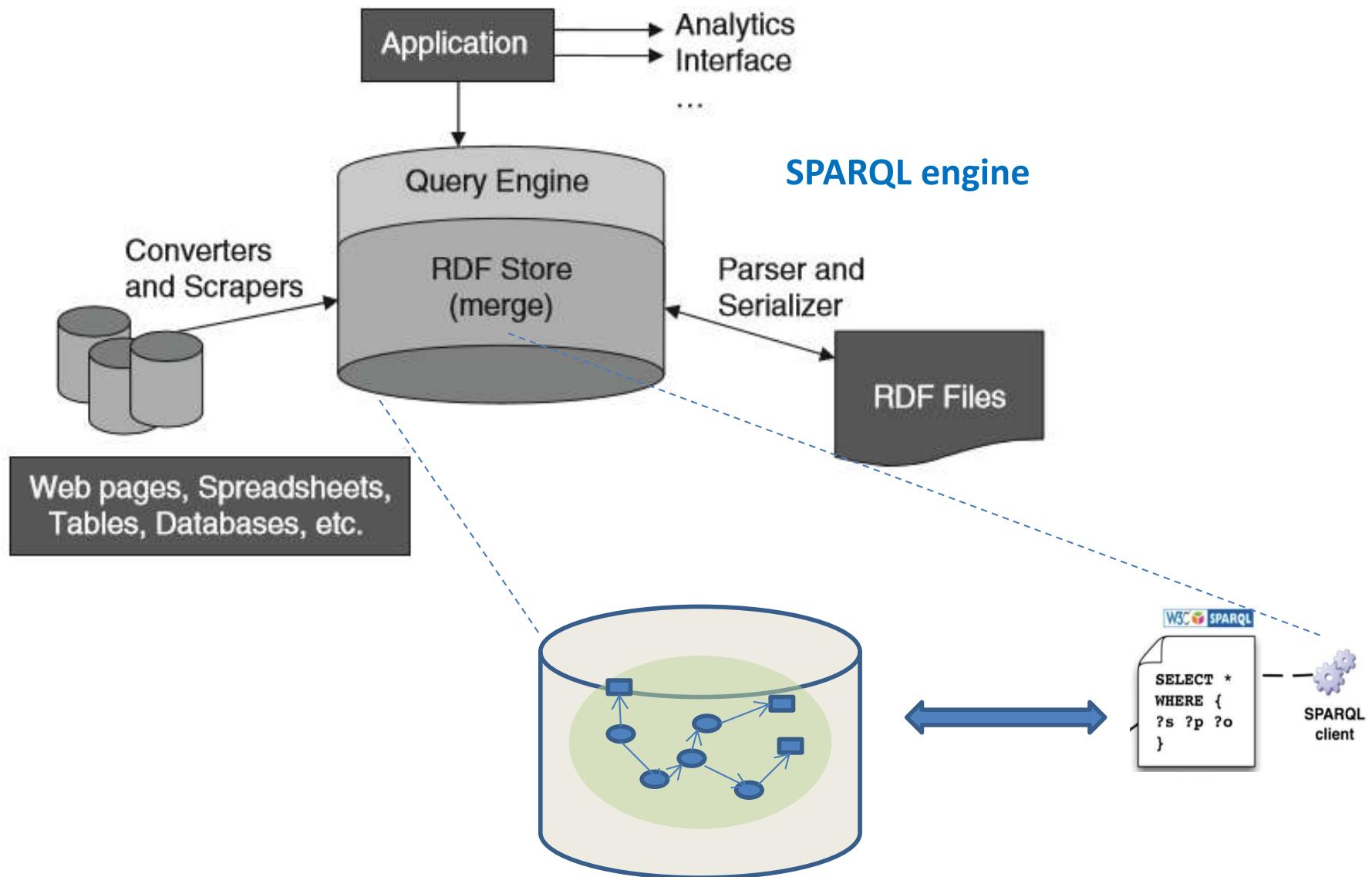
RDF Native Systems
RDF (triples, graphs) as
logical data model

RDF enabled systems

- Implemented as a middleware on top of RDBMS
- Relational data contained in traditional relational databases are translated into RDF data (and viceversa) and stored into an RDBMS
- SPARQL queries must be translated into SQL
- RDF is just a view for relational data, the system is not aware of RDF
- **Pros:** you can rely on DBMS functionalities
- **Cons:** external and internal model do not coincide, the DBMS is not optimized for RDF storage and SPARQL access, slower



RDF Native Systems



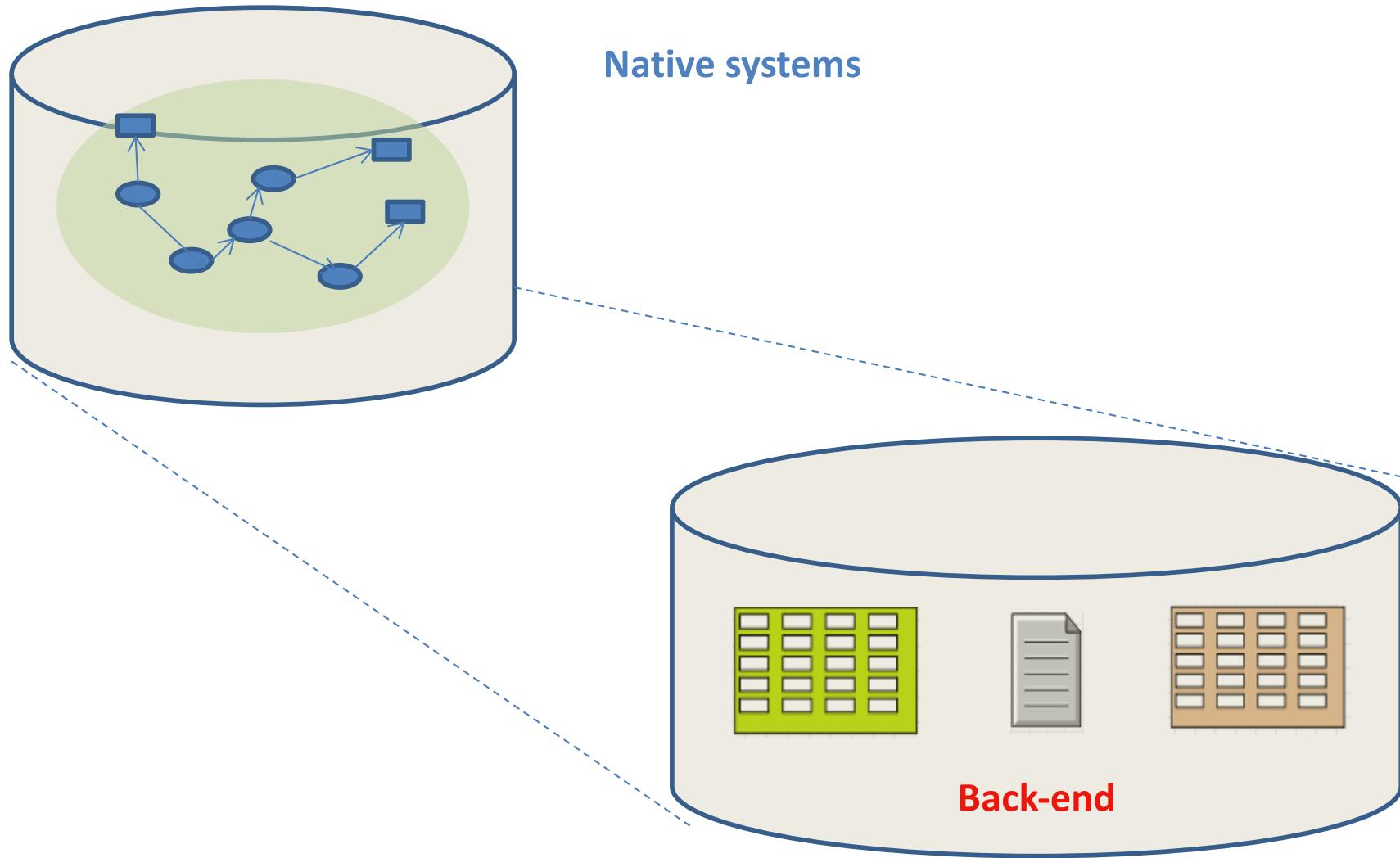
RDF Native Systems

- New systems, tailored to RDF data management → **TripleStore**
- RDF (triples, graph) is the logical model
- SPARQL engine available
- Triples stored **in memory** or in a **persistent back-end**
- Persistence can be provided by
 - a relational DBMS or
 - a NoSQL graph-based system or
 - other systems (e.g., XML based)
- The backend system often supports large-scale functionalities
- **Pros:** faster, the system is optimized for RDF storage and SPARQL access
- **Cons:** less consolidated technologies with respect to RDBMS
- <https://www.w3.org/2001/sw/wiki/Tools> for a list of existing tools

RDF Native Systems with Relational backend

- Backend: RDBMS
- Basic principles:
 - store triples in table (but the system is aware that some tables represent triples), many alternatives available
 - convert SPARQL to equivalent
 - the database will do the rest
- Used by many TripleStores

RDF Native Systems with Relational backend



RDF Native Systems with Relational backend: Single Triple Table

- Store triples in one single giant three-attribute table (subject, predicate, object)

Example: Single Triple Table

| | | |
|-----------|--------------|--|
| ex:Katja | ex:teaches | ex:Databases; |
| | ex:works_for | ex:MPI_Informatics; |
| | ex:PhD_from | ex:TU_Ilmenau. |
| ex:Martin | ex:teaches | ex:Databases; |
| | ex:works_for | ex:MPI_Informatics; |
| | ex:PhD_from | ex:Saarland_University. |
| ex:Ralf | ex:teaches | ex:Information_Retrieval; |
| | ex:PhD_from | ex:Saarland_University; |
| | ex:works_for | ex:Saarland_University, ex:MPI_Informatics. |

Table **Triples**

| subject | predicate | object |
|-----------|--------------|--------------------------|
| ex:Katja | ex:teaches | ex:Databases |
| ex:Katja | ex:works_for | ex:MPI_Informatics |
| ex:Katja | ex:PhD_from | ex:TU_Ilmenau |
| ex:Martin | ex:teaches | ex:Databases |
| ex:Martin | ex:works_for | ex:MPI_Informatics |
| ex:Martin | ex:PhD_from | ex:Saarland_University |
| ex:Ralf | ex:teaches | ex:Information_Retrieval |
| ex:Ralf | ex:PhD_from | ex:Saarland_University |
| ex:Ralf | ex:works_for | ex:Saarland_University |
| ex:Ralf | ex:works_for | ex:MPI_Informatics |

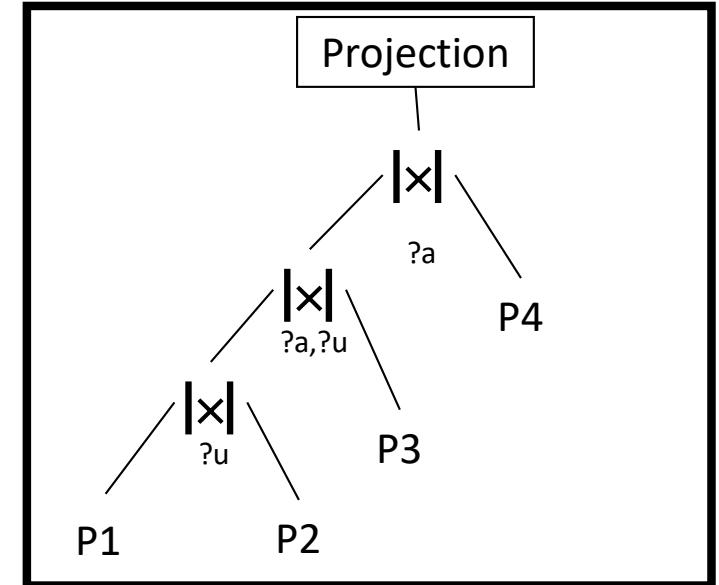
Conversion of SPARQL to SQL

General approach to translate **SPARQL** into **SQL**:

- (1) Each **triple pattern** is translated into a (self-) **JOIN** over the triple table
- (2) **Shared variables** create **JOIN conditions**
- (3) **Constants** create **WHERE conditions**
- (4) **FILTER conditions** create **WHERE conditions**
- (5) **OPTIONAL clauses** create **OUTER JOINS**
- (6) **UNION clauses** create **UNION expressions**

Example: from SPARQL to SQL

```
SELECT ?a ?b WHERE  
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
```



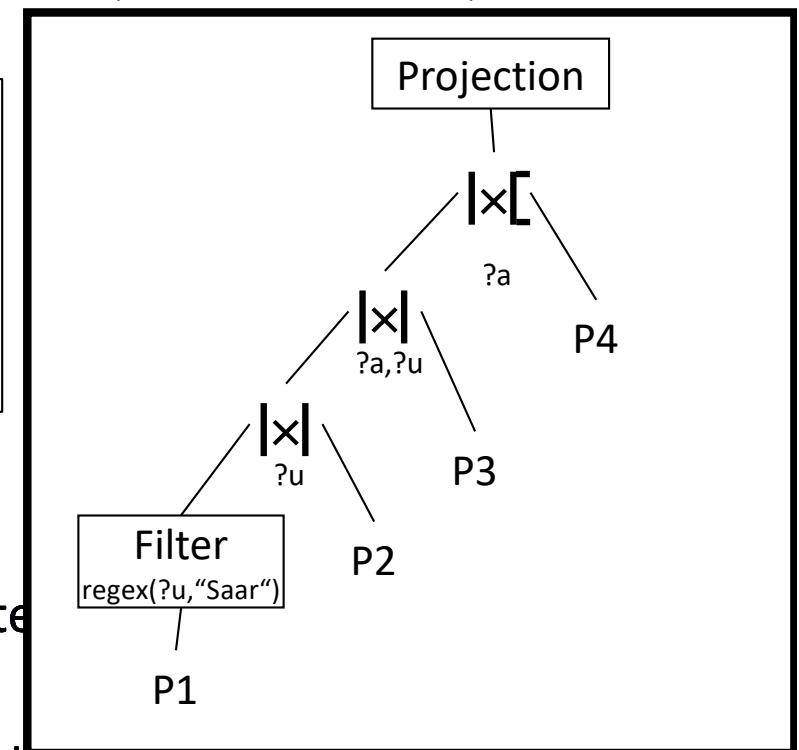
```
SELECT P1.subject as A, P2.subject as B  
FROM Triples P1, Triples P2, Triples P3  
WHERE P1.predicate="works_for" AND P2.predicate="works_for"  
AND P3.predicate="phd_from"  
AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
```

Many self and outer joins

Example: from SPARQL to SQL

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, "Saar"))
```

```
SELECT P1.subject as A, P2.subject as B
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate="works_for" AND P2.predicate="phd_from"
      AND P3.predicate="teaches"
      AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
      AND REGEXP_LIKE(P1.object, "Saar")
```

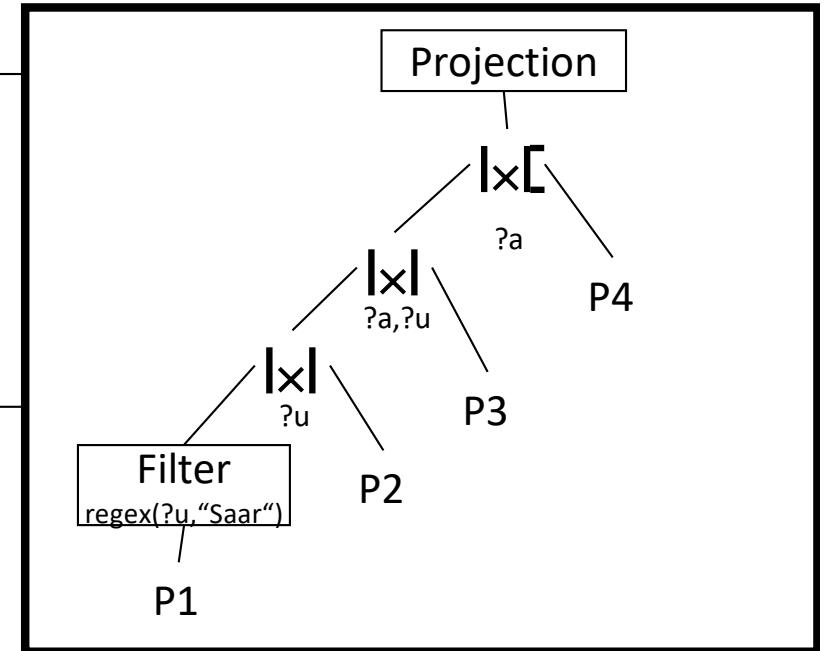


Many self and outer joins

Example: from SPARQL to SQL

```
SELECT ?a ?b ?t WHERE
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
OPTIONAL {?a teaches ?t}
FILTER (regex(?u, "Saar"))
```

```
SELECT R1.A, R1.B, R2.T FROM
( SELECT P1.subject as A, P2.subject as B
FROM Triples P1, Triples P2, Triples P3
WHERE P1.predicate="works_for" AND P2.predicate="works_for"
AND P3.predicate="phd_from"
AND P1.object=P2.object AND P1.subject=P3.subject AND P1.object=P3.object
AND REGEXP_LIKE(P1.object, "Saar")
) R1 LEFT OUTER JOIN
( SELECT P4.subject as A, P4.object as T
FROM Triples P4
WHERE P4.predicate="teaches") AS R2
) ON (R1.A=R2.A)
```



Many self and outer joins

Single Triple Table: Pros and Cons

Advantages:

- No restructuring is required if the ontology changes (e.g., new classes, etc., realized by a simple INSERT command in the table)

Disadvantages:

- Performing a query means searching the whole database and queries involving joins become very expensive

Dictionary for Strings

Map all strings to unique integers (e.g., via hashing)

- Regular size (4-8 bytes), much easier to handle
- Dictionary usually small, can be kept in main memory

| | |
|----------------------------|----------|
| <http://example.de/Katja> | → 194760 |
| <http://example.de/Martin> | → 679375 |
| <http://example.de/Ralf> | → 4634 |

This may break original lexicographic sorting order
⇒ RANGE conditions are difficult!
⇒ FILTER conditions may be more expensive!

RDF Native Systems with Relational backend: Property Tables

Observations and assumptions

- Not too many different predicates
- Triple patterns usually have fixed predicate
- Need to access all triples with one predicate

Design consequence

- Use one two-attribute table for each predicate
- From one giant three-attribute table to many property tables

Example

ex:Katja ex:teaches ex:Databases;
 ex:works_for ex:MPI_Informatics;
 ex:PhD_from ex:TU_Ilmenau.

ex:Martin ex:teaches ex:Databases;
 ex:works_for ex:MPI_Informatics;
 ex:PhD_from ex:Saarland_University.

ex:Ralf ex:teaches ex:Information_Retrieval;
 ex:PhD_from ex:Saarland_University;
 ex:works_for ex:Saarland_University,
 ex:MPI_Informatics.

| works_for | |
|-----------|------------------------|
| subject | object |
| ex:Katja | ex:MPI_Informatics |
| ex:Martin | ex:MPI_Informatics |
| ex:Ralf | ex:Saarland_University |
| ex:Ralf | ex:MPI_Informatics |

| teaches | |
|-----------|--------------------------|
| subject | object |
| ex:Katja | ex:Databases |
| ex:Martin | ex:Databases |
| ex:Ralf | ex:Information_Retrieval |

| PhD_from | |
|-----------|------------------------|
| subject | object |
| ex:Katja | ex:TU_Ilmenau |
| ex:Martin | ex:Saarland_University |
| ex:Ralf | ex:Saarland_University |

Example: from SPARQL to SQL

```
SELECT ?a ?b ?t WHERE  
{?a works_for ?u. ?b works_for ?u. ?a phd_from ?u. }
```

```
SELECT W1.subject as A, W2.subject as B  
FROM works_for W1, works_for W2, phd_from P3  
WHERE W1.object=W2.object  
    AND W1.subject=P3.subject  
    AND W1.object=P3.object
```

Fragmentation-based solutions and columnstores

Columnstores store *each* column (or group of columns) of a table separately

| PhD_from | |
|-----------|------------------------|
| subject | object |
| ex:Katja | ex:TU_Ilmenau |
| ex:Martin | ex:Saarland_University |
| ex:Ralf | ex:Saarland_University |



| PhD from:subject |
|------------------|
| ex:Katja |
| ex:Martin |
| ex:Ralf |

| PhD_from:object |
|------------------------|
| ex:TU_Ilmenau |
| ex:Saarland_University |
| ex:Saarland_University |

Advantages:

- Fast if **only subject or object** are accessed, not both
- Allows for a very compact representation

Problems:

- Need to recombine columns if subject and object are accessed
- Inefficient for triple patterns with predicate variable
- Space overhead in case the same subject is replicated among triples several times

Compression in Columnstores

General ideas:

- Store subject the minimum number of times
- Use same order of subjects for all columns, including NULL values when necessary

| subject | PhD_from | teaches | works_for |
|-----------|------------------------|--------------------------|------------------------|
| ex:Katja | ex:TU_Ilmenau | ex:Databases | ex:MPI_Informatics |
| ex:Martin | ex:Saarland_University | ex:Databases | ex:MPI_Informatics |
| ex:Ralf | ex:Saarland_University | ex:Information_Retrieval | ex:Saarland_University |
| ex:Ralf | NULL | NULL | ex:MPI_Informatics |

- Additional compression to get rid of NULL values

| PhD_from: bit[1110] | Teaches: range[1-3] |
|------------------------|--------------------------|
| ex:TU_Ilmenau | ex:Databases |
| ex:Saarland_University | ex:Databases |
| ex:Saarland_University | ex:Information_Retrieval |

Other Solutions: Property Tables

Group entities with similar predicates into a relational table
(for example using RDF types or a clustering algorithm).

ex:Katja ex:teaches ex:Databases;
ex:works_for ex:MPI_Informatics;
ex:PhD_from ex:TU_Ilmenau.

ex:Martin ex:teaches ex:Databases;
ex:works_for ex:MPI_Informatics;
ex:PhD_from ex:Saarland_University.

ex:Ralf ex:teaches ex:Information_Retrieval;
ex:PhD_from ex:Saarland_University;
ex:works_for ex:Saarland_University,
 ex:MPI_Informatics.

| subject | teaches | PhD_from |
|-----------|--------------|------------------------|
| ex:Katja | ex:Databases | ex:TU_Ilmenau |
| ex:Martin | ex:Databases | ex:Saarland_University |
| ex:Ralf | ex:IR | ex:Saarland_University |

| subject | predicate | object | | |
|-----------|--------------|------------------------|--|--|
| ex:Katja | ex:works_for | ex:MPI_Informatics | | |
| ex:Martin | ex:works_for | ex:MPI_Informatics | | |
| ex:Ralf | ex:works_for | ex:Saarland_University | | |
| ex:Ralf | ex:works_for | ex:MPI_Informatics | | |

← “Leftover triples”

Property Tables: Pros and Cons

Advantages:

- More in the spirit of existing relational systems
- Saves many self-joins over triple tables

Disadvantages:

- Query mapping depends on schema
- Schema changes very expensive

RDF Native Systems with Relational backend: is that all?

Well, no.

- Which **indexes** should be built?
(to support efficient evaluation of triple patterns)
- How can we **reduce storage space**?
- How can we find the **best execution plan**?

Traditional RDBMS has to take into account RDF features:

- flexible, extensible, generic storage not needed here
- cannot deal with multiple self-joins of a single table
- often generate bad execution plans

RDF Native Systems with graph-based backend

- Backend: NoSQL graph-based system
- Implementing triple stores with a relational backend makes the system **associative**
 - Triples stored in different tables can be combined together only through joins
- On the other hand, NoSQL graph-based systems are **navigational** and provide native graph storage
 - Connections between nodes are stores and can be directly navigated through pointers
- NoSQL graph-based systems more suitable for deep or variable-length traversals and path queries
 - Lead to a very large number of joins in TripleStore with relational backend
- NoSQL graph databases should support specialized graph index structures , tailored to RDF

RDF USE CASES

RDF use cases

- Two possible scenarios
 - Local access
 - Store your RDF dataset in a given Triplestore
 - use the interaction protocol, based on SPARQL, available in the TripleStore
 - Remote access (more interesting)
 - Single RDF dataset available on the Web
 - Processing a single RDF dataset or a federation of many RDF datasets

Processing a single RDF dataset

- SPROT = SPARQL Protocol for RDF
- SPARQL endpoint
 - A service, conformant to SPROT, that accepts SPARQL queries and returns results via HTTP, in one or more machine-processable formats
 - Either generic (fetching data on the Web as needed) or specific (querying an associated TripleStore)
 - Issuing a SPARQL query is an HTTP GET request with parameter query
- A SPARQL endpoint is mostly conceived as a machine-friendly interface towards a knowledge base
- <https://www.w3.org/wiki/SparqlEndpoints> for a list of available SPARQL endpoints
- <https://dbpedia.org/sparql>: SPARQL endpoint for DBpedia (the semantic version of Wikipedia, <https://www.dbpedia.org/>)

Processing a single RDF dataset: SPARQL Client Libraries

- **More convenient than on the protocol level:**

- SPARQL JavaScript Library

http://www.thefigtrees.net/lee/blog/2006/04/sparql_calendar_demo_a_sparql.html

- ARC for PHP <http://arc.semsol.org/>

- RAP – RDF API for PHP

<http://www4.wiwiss.fu-berlin.de/bizer/rdfapi/index.html>

- Jena / ARQ (Java) <http://jena.sourceforge.net/>

- Sesame (Java) <http://www.openrdf.org/>

- SPARQL Wrapper (Python)

<http://sparql-wrapper.sourceforge.net/>

- PySPARQL (Python)

<http://code.google.com/p/pysparql/>

Processing a federation of many RDF datasets

- Many RDF datasets, stored in many independent repositories
- For some applications, you might need to use all the datasets in the context of the same query
- Two approaches
 - Controlled integration approach
 - Link traversal-based query execution

Controlled integration approach

- Assumptions
 - you know in advance the RDF data sources to be queried
 - each data source is exposed via a SPARQL endpoint
- Specify the service (i.e., the SPARQL endpoint) you want to use in the context of your query

Controlled integration approach: SPARQL 1.1 Federation Extension

- **SERVICE pattern in SPARQL 1.1**
 - Explicitly specify query patterns whose execution must be distributed to a remote SPARQL endpoint

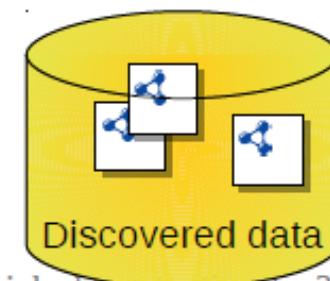
```
SELECT ?v ?ve WHERE
{
  ?v rdf:type umbel-sc:Volcano ;
       p:location dbpedia:Italy .
  SERVICE <http://volcanos.example.org/query> {
    ?v p:lastEruption ?ve }
}
```

Controlled Integration Solutions

- Pros:
 - Queried data is up to date: you do not care about their storage and update
- Cons:
 - All relevant datasets must be exposed via a SPARQL endpoint
 - You have to know the relevant data sources beforehand
 - You restrict yourselves to the selected sources
 - You do not tap the full potential of the Web

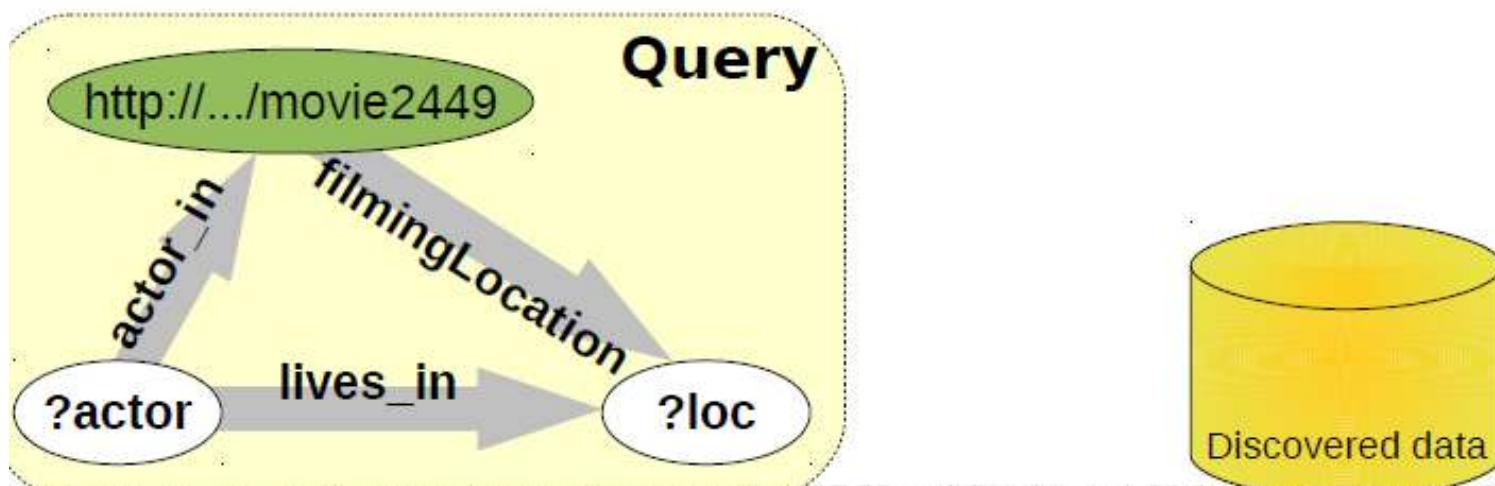
Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



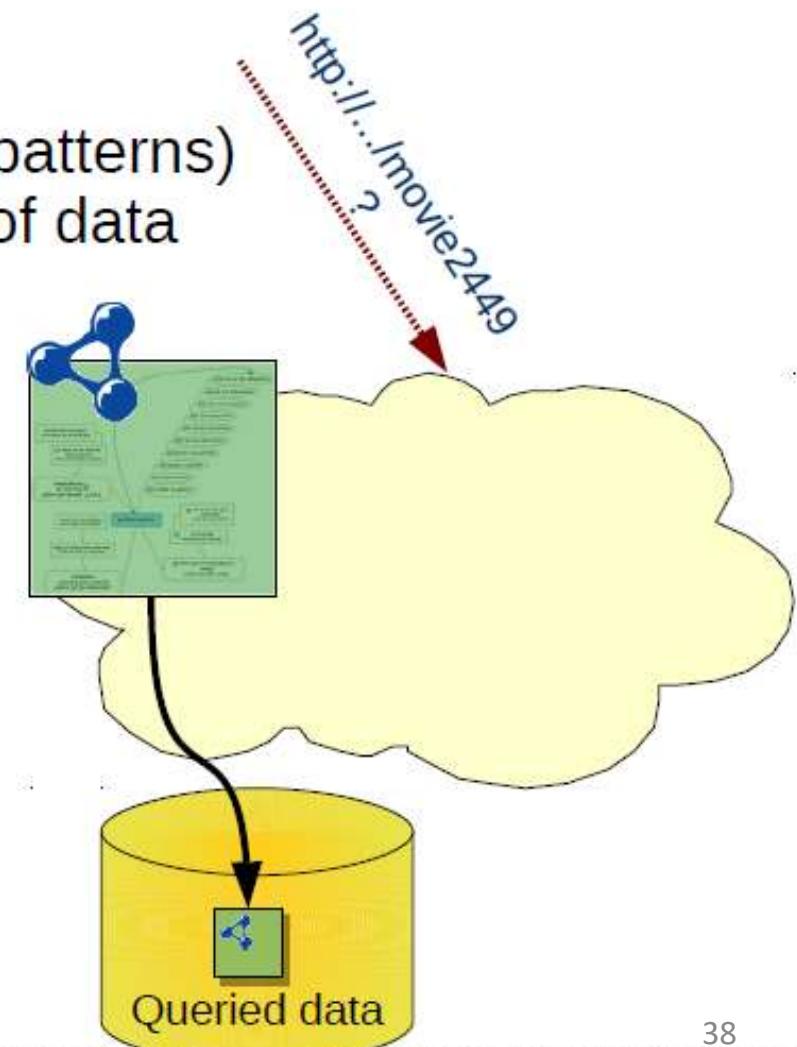
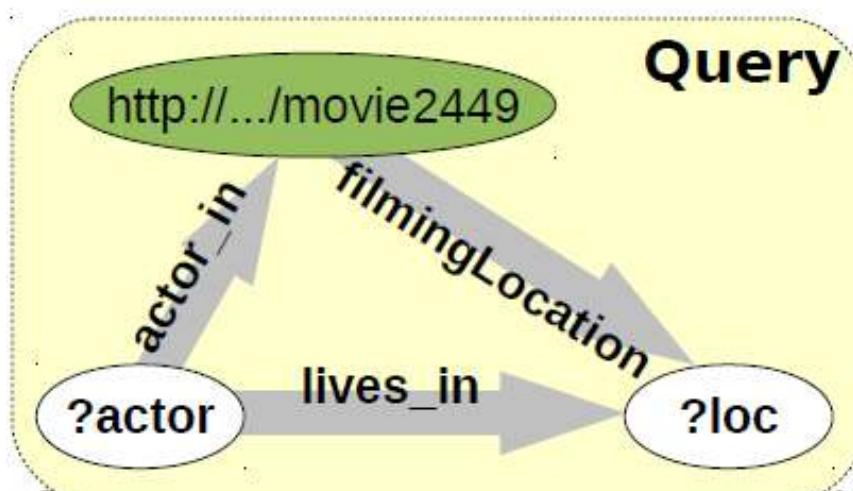
Link traversal-based query execution

- Intertwine query evaluation with traversal of data links
- We alternate between:
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



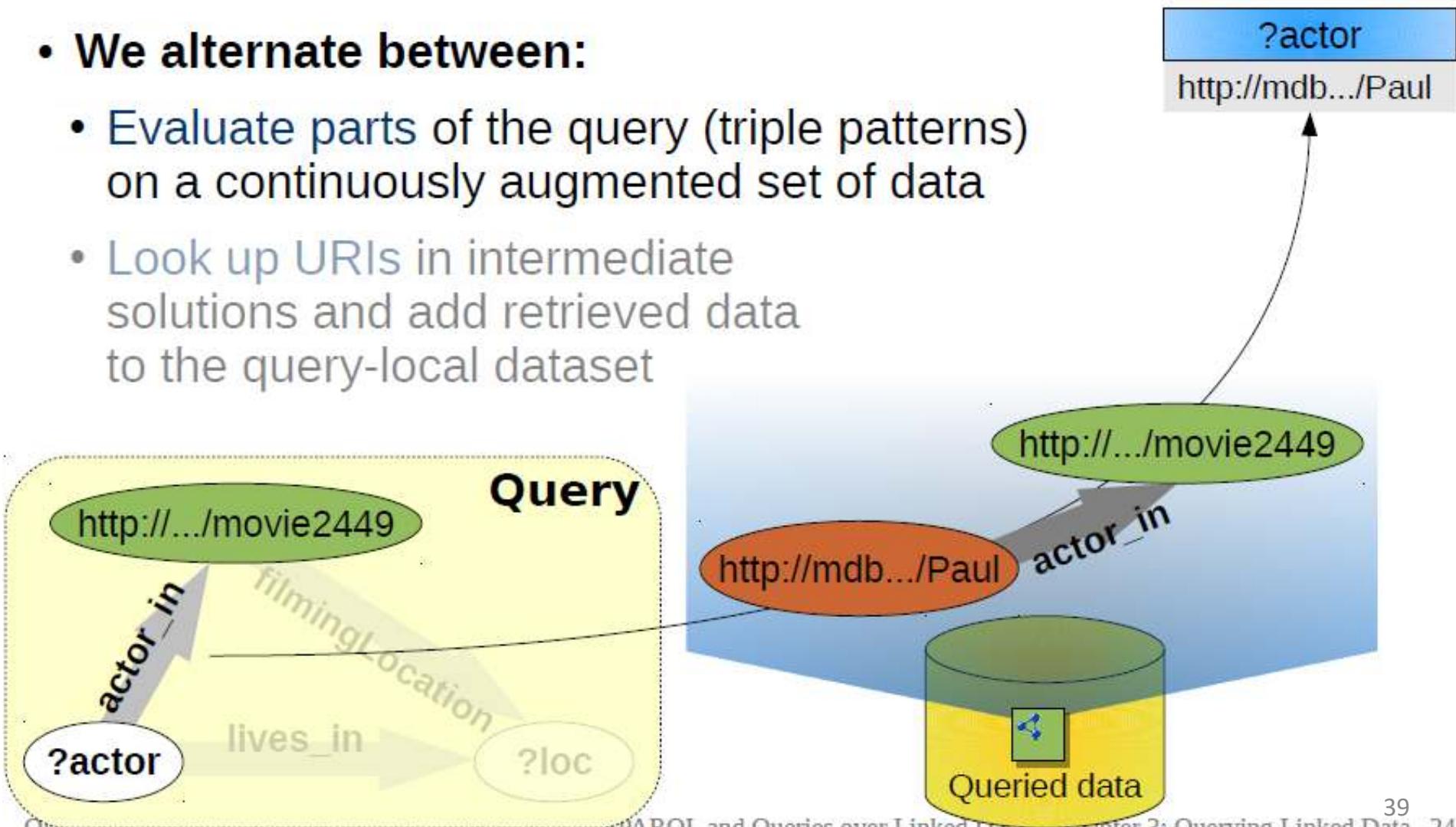
Link traversal-based query execution

- Intertwine query evaluation with traversal of data links
- We alternate between:
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



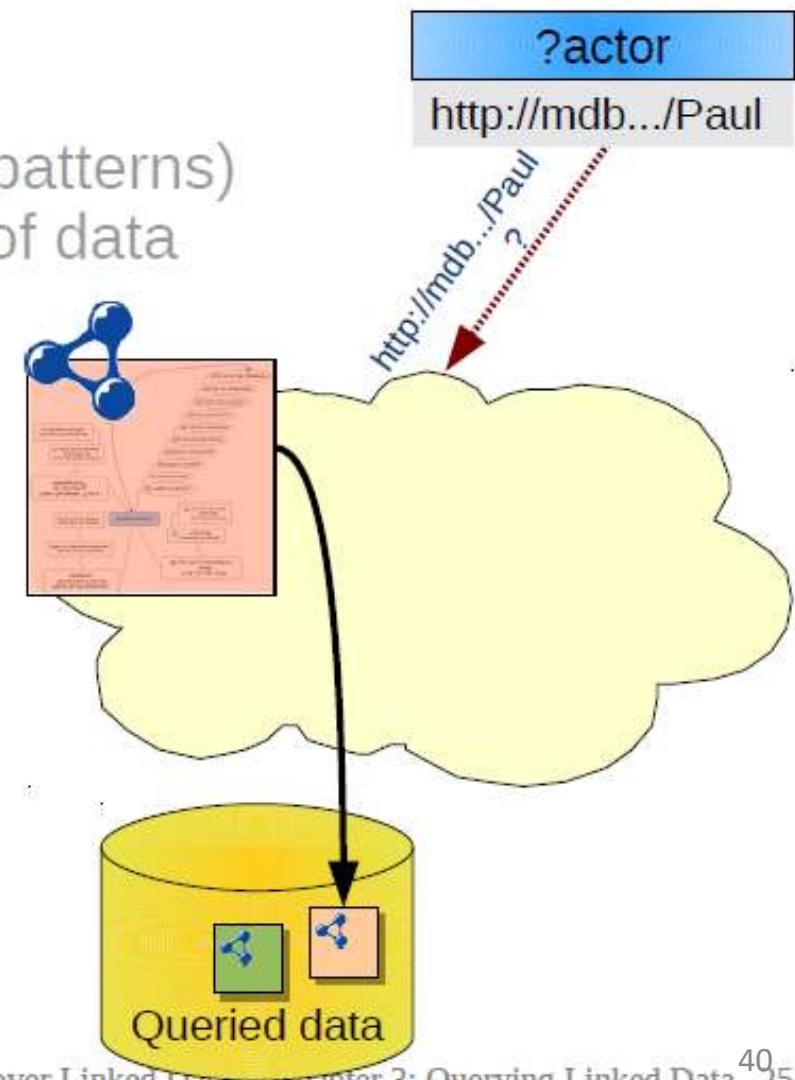
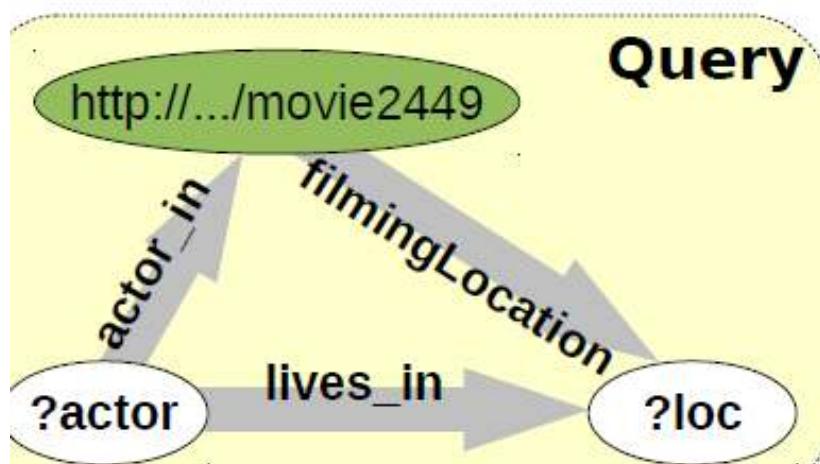
Link traversal-based query execution

- Intertwine query evaluation with traversal of data links
- We alternate between:
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



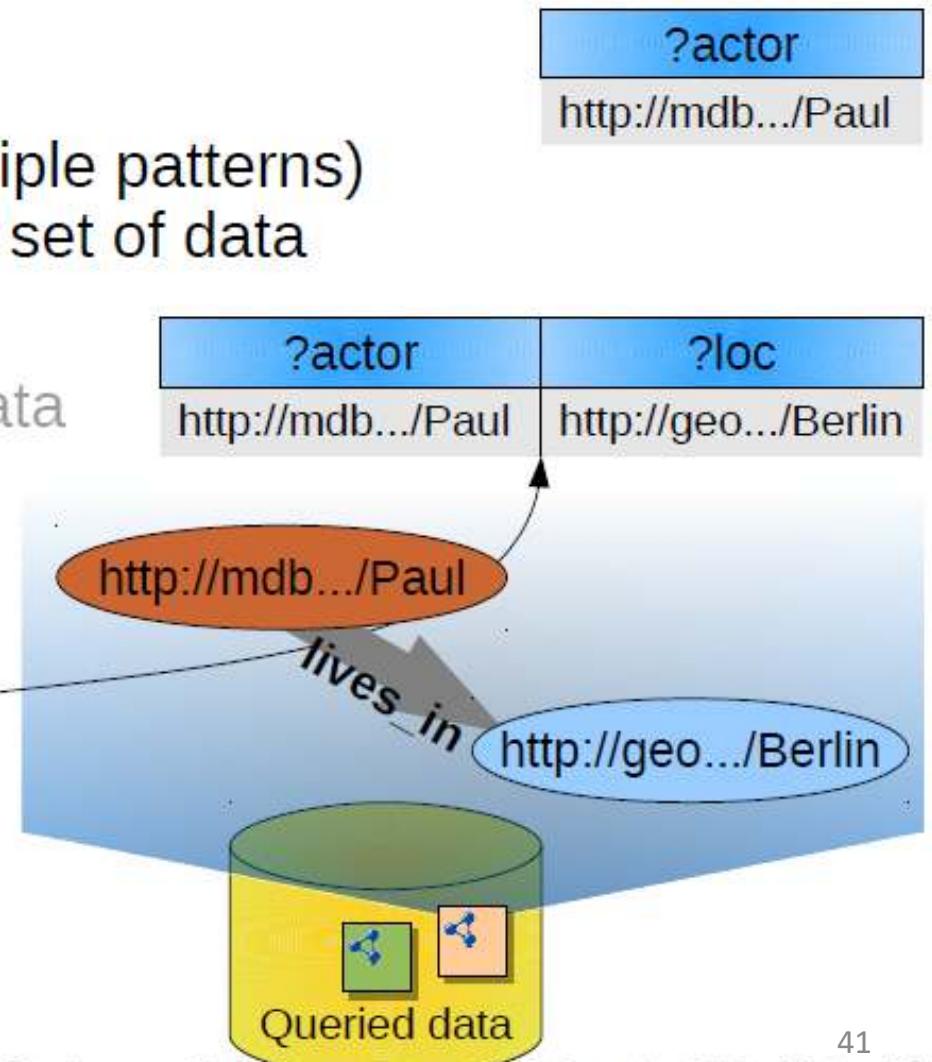
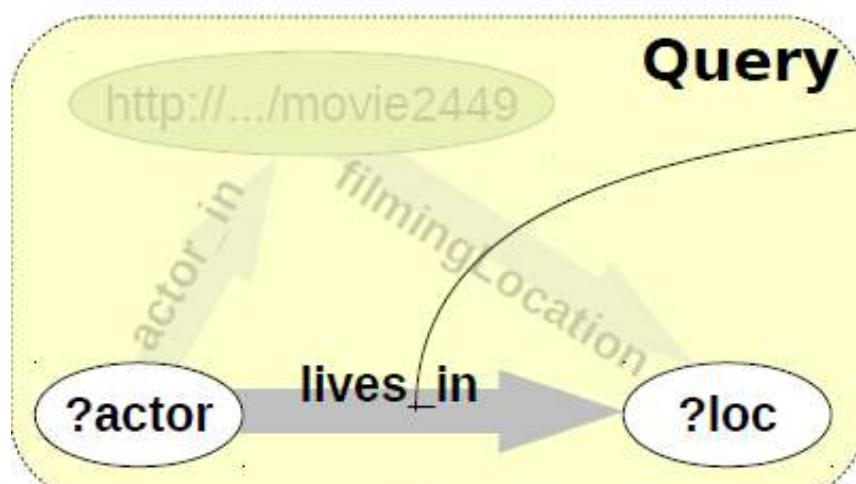
Link traversal-based query execution

- Intertwine query evaluation with traversal of data links

- We alternate between:

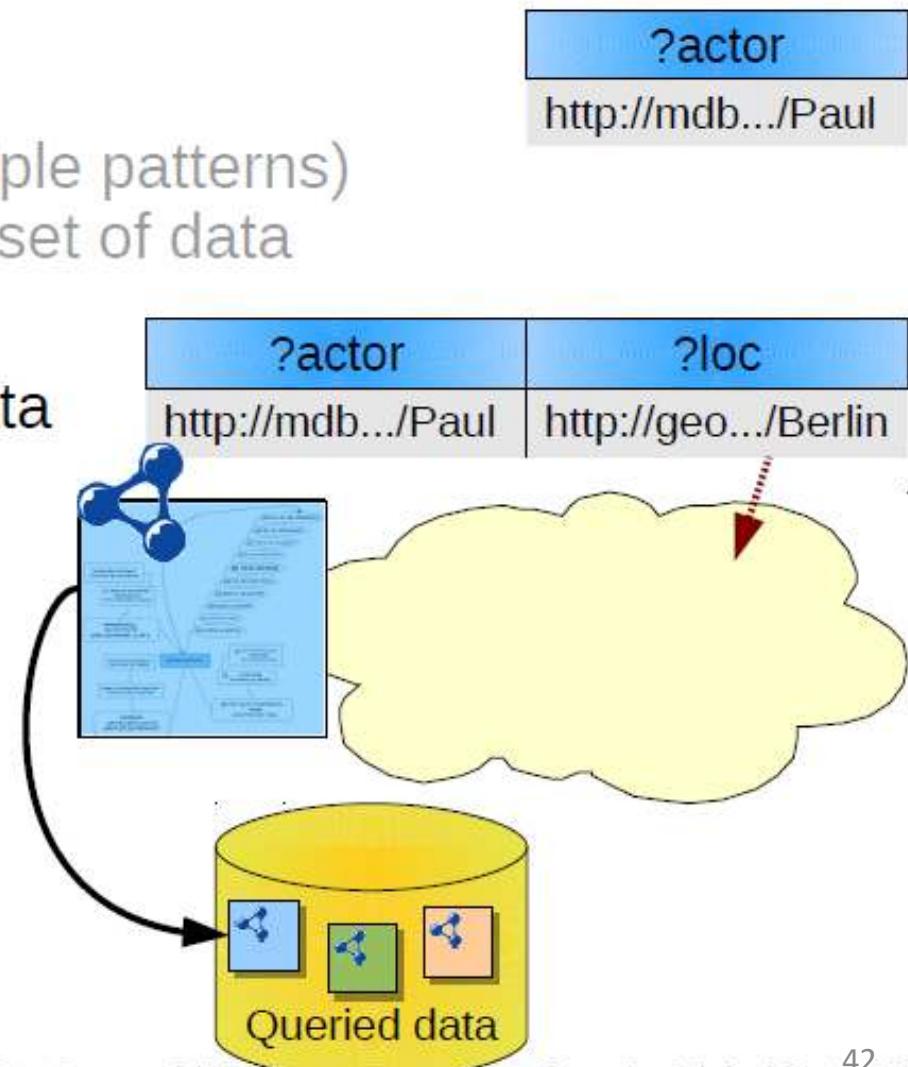
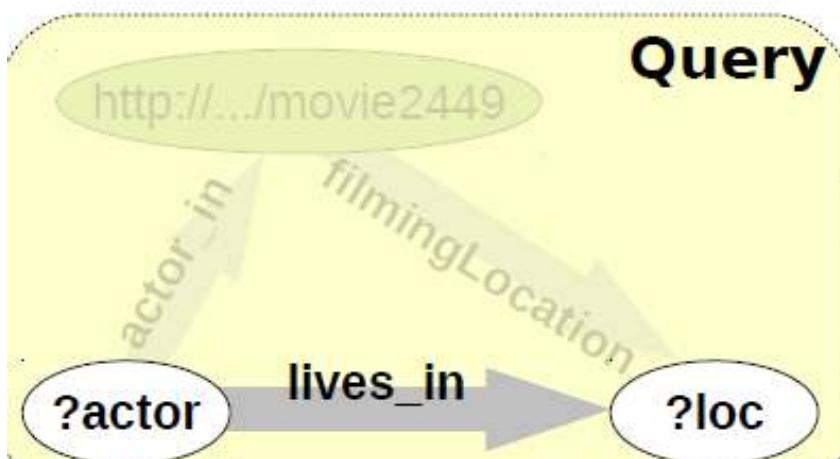
- Evaluate parts of the query (triple patterns) on a continuously augmented set of data

- Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



Link traversal-based query execution

- **Intertwine query evaluation with traversal of data links**
- **We alternate between:**
 - Evaluate parts of the query (triple patterns) on a continuously augmented set of data
 - Look up URIs in intermediate solutions and add retrieved data to the query-local dataset



Link traversal-based query execution

- Pros:
 - You might not know in advance the datasets to be queried
 - You tap the full potential of the Web
- Cons:
 - Slow: many navigations to get the RDF data sources related to a given resource (.e., describing a given resource)
 - Local RDF dataset incrementally generated
 - Very interesting from a research point of view, not very used in practice