Big Data Infrastructures

Natalia Ostapuk

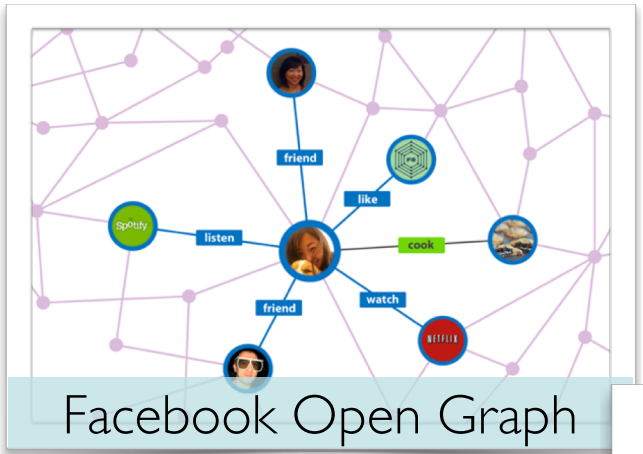
Fall 2018

Lecture 6 – Graph Databases

Today's Agenda

- Graphs
- Storing Graphs
- Graph Databases
- Introduction to Cypher
- Lab: Neo4j

Everyone is talking about graphs…



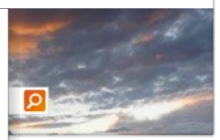
Facebook Open Graph

Google Just Got A Whole Lot Smarter, Launches Its Knowledge Graph

FREDERIC LARDINOIS

Bing one-ups knowledge graph, hires Encyclopaedia Britannica to supply results

By Daniel Cooper posted Jun 8th 2012 3:02PM



Neo4j
The World's Leading Graph Database

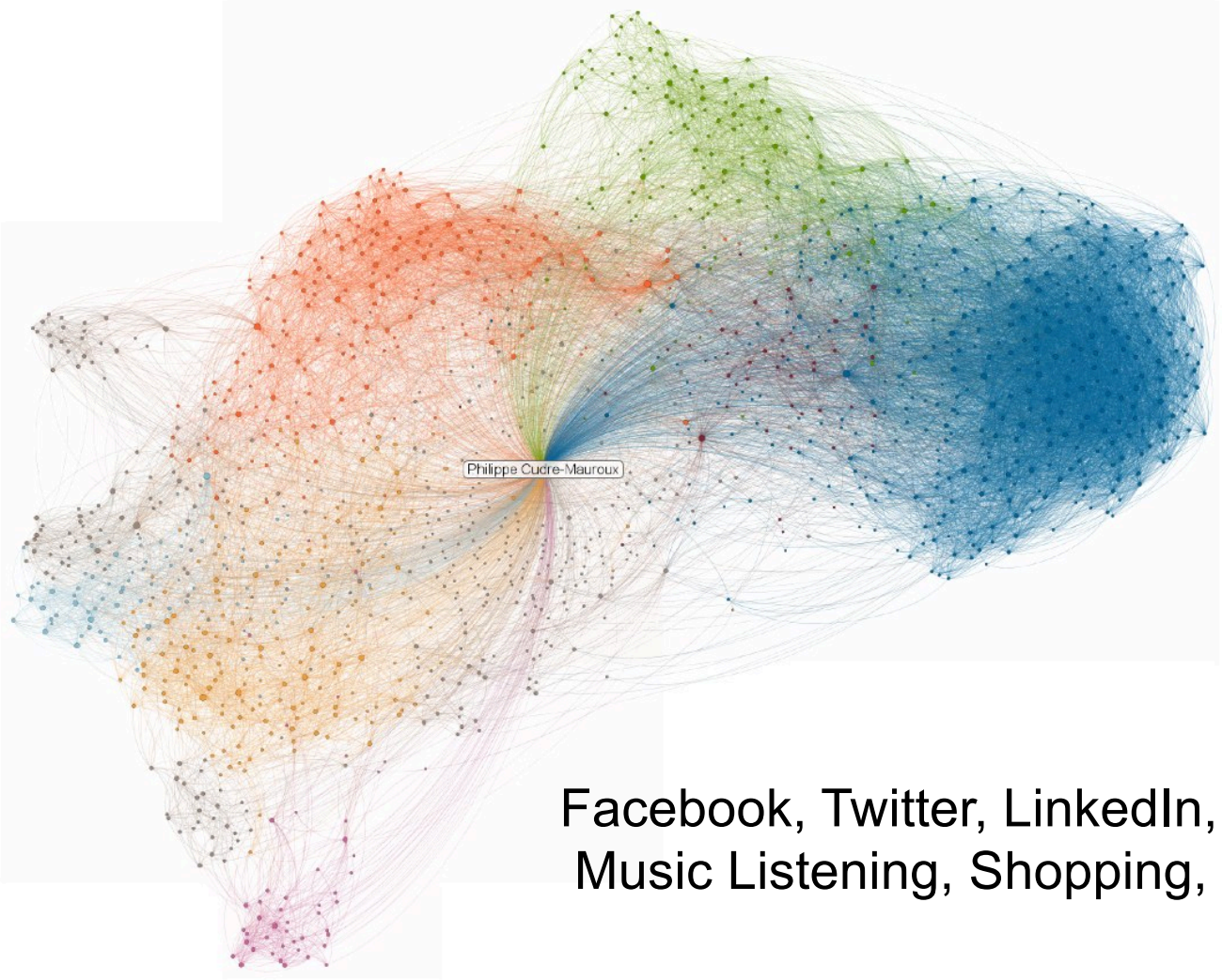
Why the Interest Graph Is a Marketer's Best Friend

1 day ago by Nadim Hossain

5

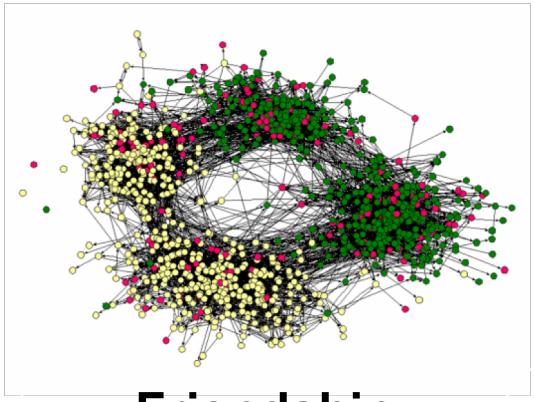
Introducing Graph Search

We all have our own graphs...



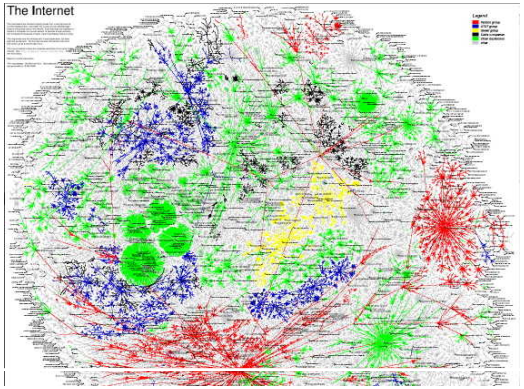
Facebook, Twitter, LinkedIn, Web browsing,
Music Listening, Shopping, Working…

Further Types of Graphs



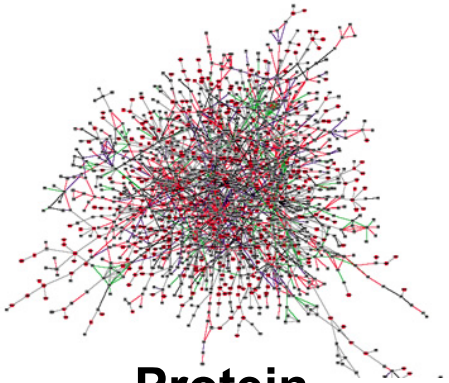
Friendship Network

[Moody'01]



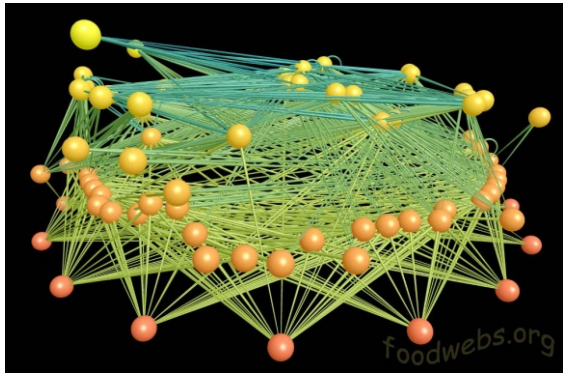
Internet Map

[lumeta.com]



Protein Interactions

[genomebiology.com]



Food Web

[foodwebs.org]

Graphs: Common Use Cases

- Social networks (Facebook, Twitter, LinkedIn)
- Recommender systems (MovieLens, Amazon)
- Geospatial applications (Google Maps Navigation)
- Network and data center management

Graphs: Small and Large

- Small graphs
  - Manage a collection of small graphs
  - Bioinformatics and cheminformatics
  - Well studied

- Large graphs
  - One large graph, aka “network”
  - Social network, and knowledge representation
  - Active area of research

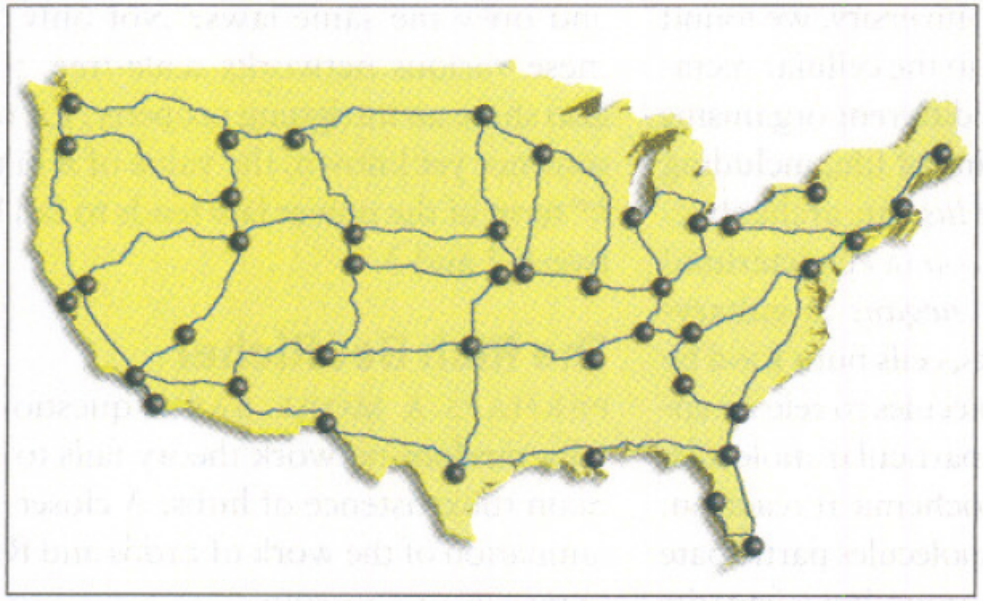
Classes of Large Graphs

- Random graphs
  - Nodes are paired with a uniform probability
  - Easiest to create
  - Less common

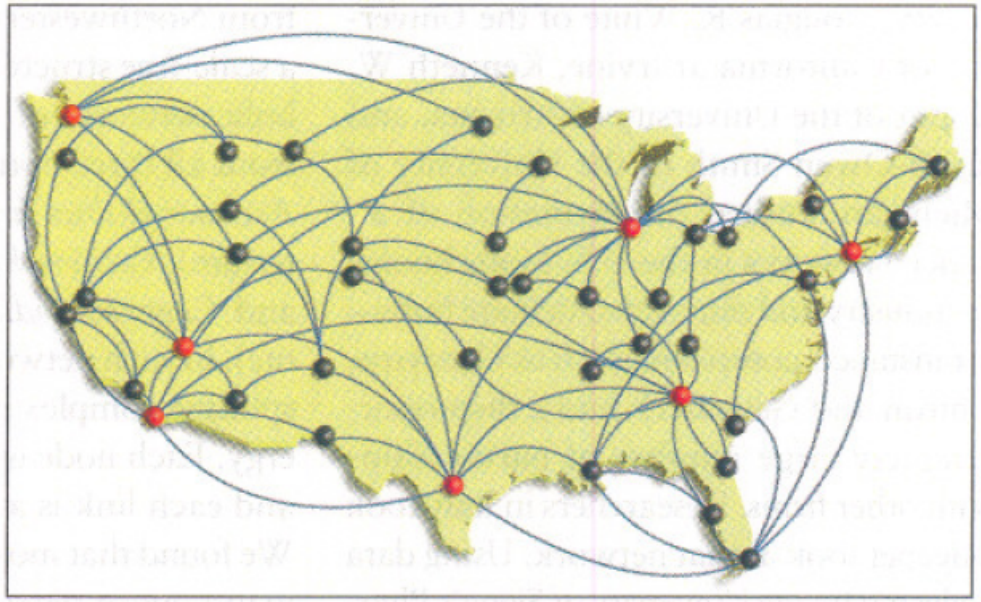
- Scale-free graphs
  - Distribution of node degree follows a power law distribution
  - Most large graphs are scale-free
  - Small world phenomena & hubs
  - Hard to partition

Classes of Large Graphs

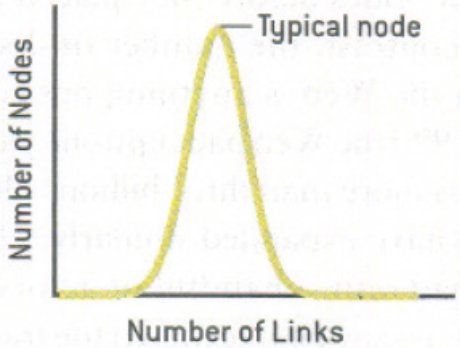
Random Network



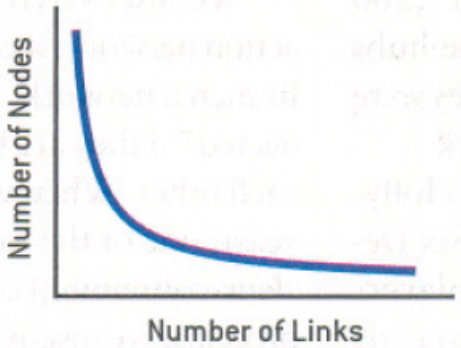
Scale-Free Network

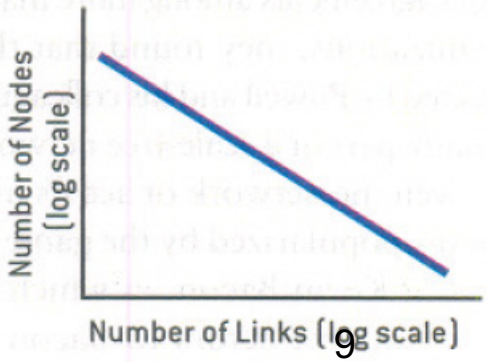


Bell Curve Distribution of Node Linkages



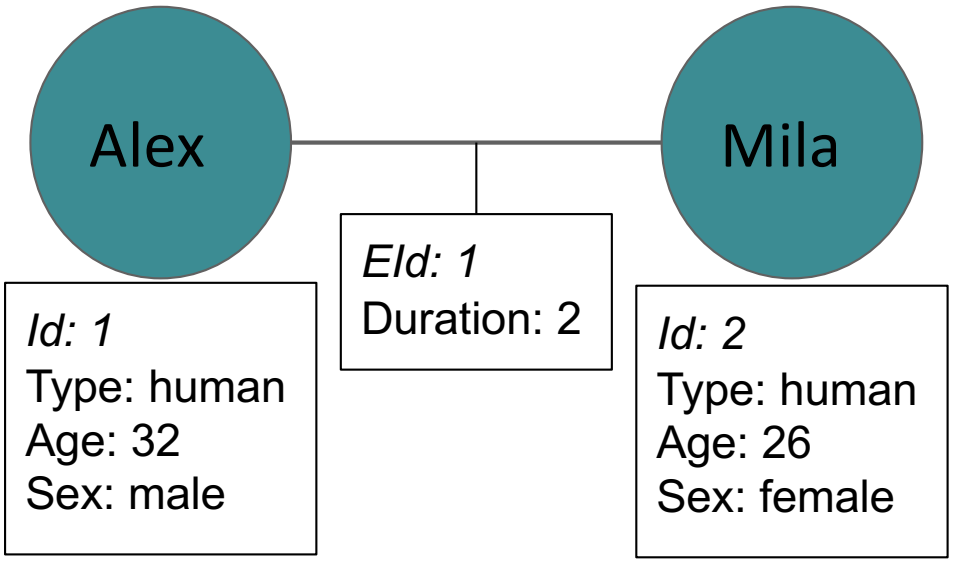
Power Law Distribution of Node Linkages



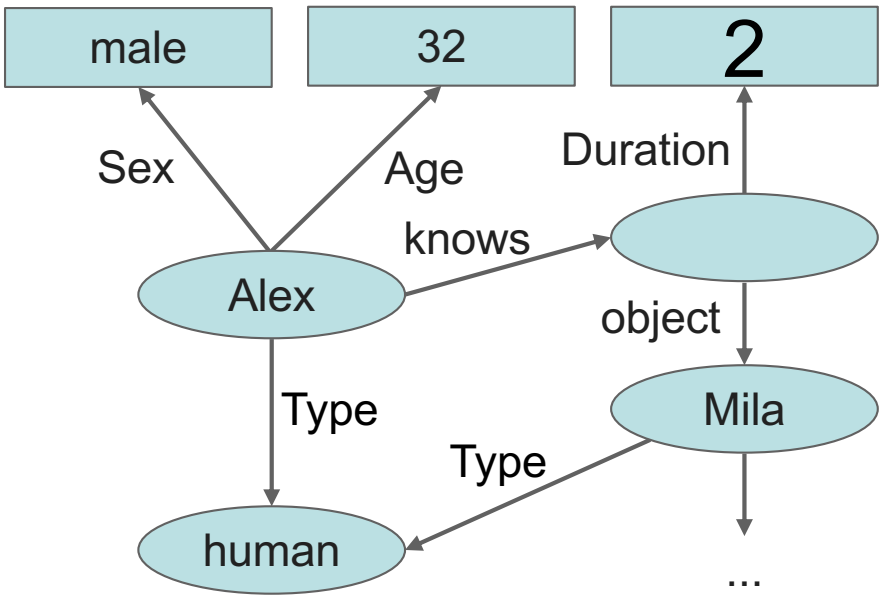


Graph Data Models

Property Graph



RDF Graph (triples)



Graph Data Models: Property Graph

- Vertices
  - Nodes: ID + label (optional) + set of key-value pairs
- Edges
  - Relationships: ID + Type + set of key-value pairs
- Nodes and relationships have internal structure
- Data storing and querying

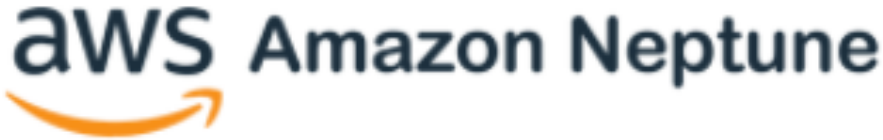
Graph Data Models: RDF Graph

- Vertices
  - Resources: URIs
  - Attribute Values: Literal Values
- Edges
  - Relationships: URIs
- Nodes or relationships have NO internal structure
- Key concept – triple: two vertices connected by an edge (*subject-predicate-object)*
- Data exchange

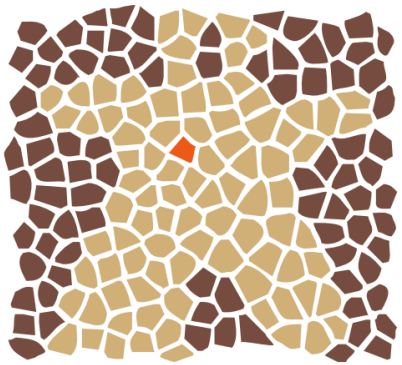
Technologies for Managing and Processing Graphs

- **Graph databases**
  - Equivalent of OLTP databases in the relational world
  - Typically accessed directly in real time from an application
  - Optimized for transactional performance
- Graph compute engines
  - Equivalent of OLAP databases
  - Used for offline graph analytics
  - Optimized for scanning and processing large amounts of information in batches
  - Normally don’t have a storage layer – processing data that is fed in from an external source

Graph Databases

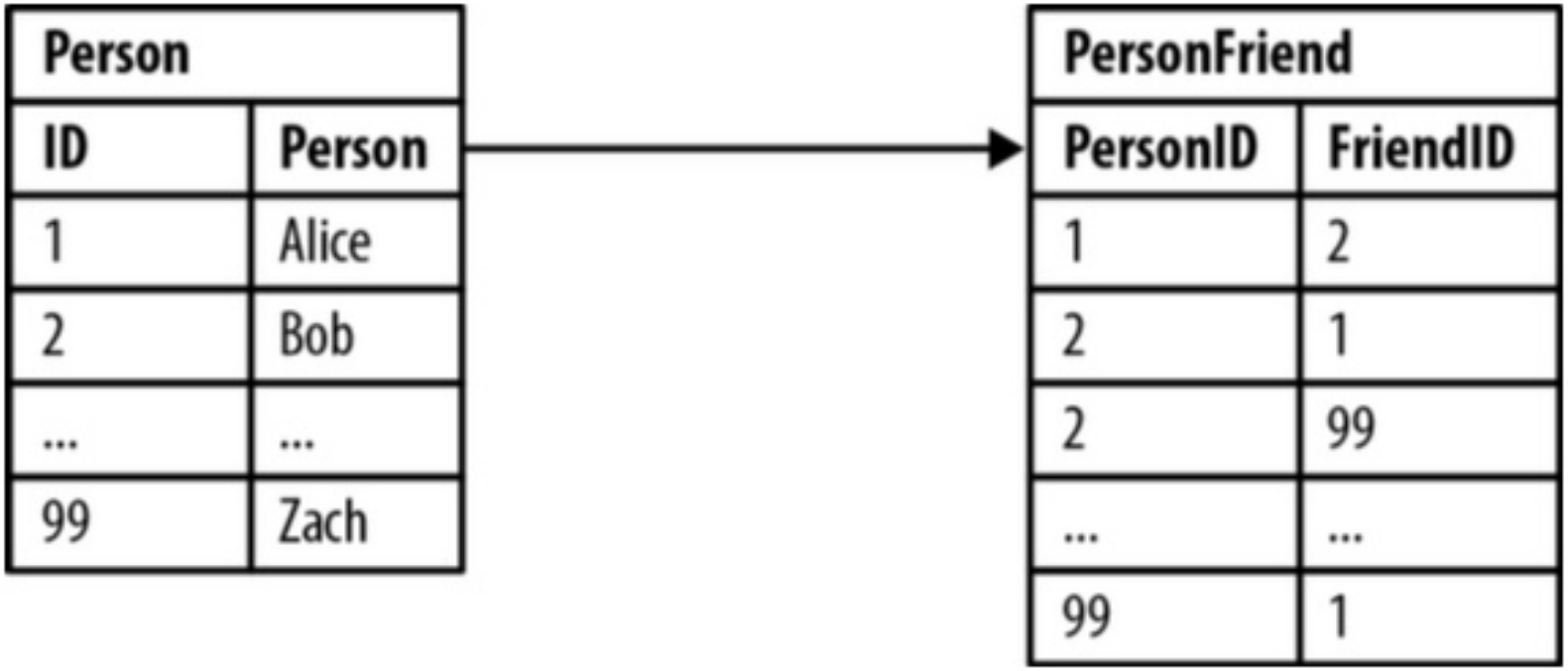


Graph Compute Engines



Why do we need something different to store and process graphs?

Storing Connected Data: RDBMS



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Storing Connected Data: RDBMS

- Who are Bob's friend?

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
     ON PersonFriend.FriendID = p1.ID
JOIN Person p2
     ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Storing Connected Data: RDBMS

- Who is friend with Bob?

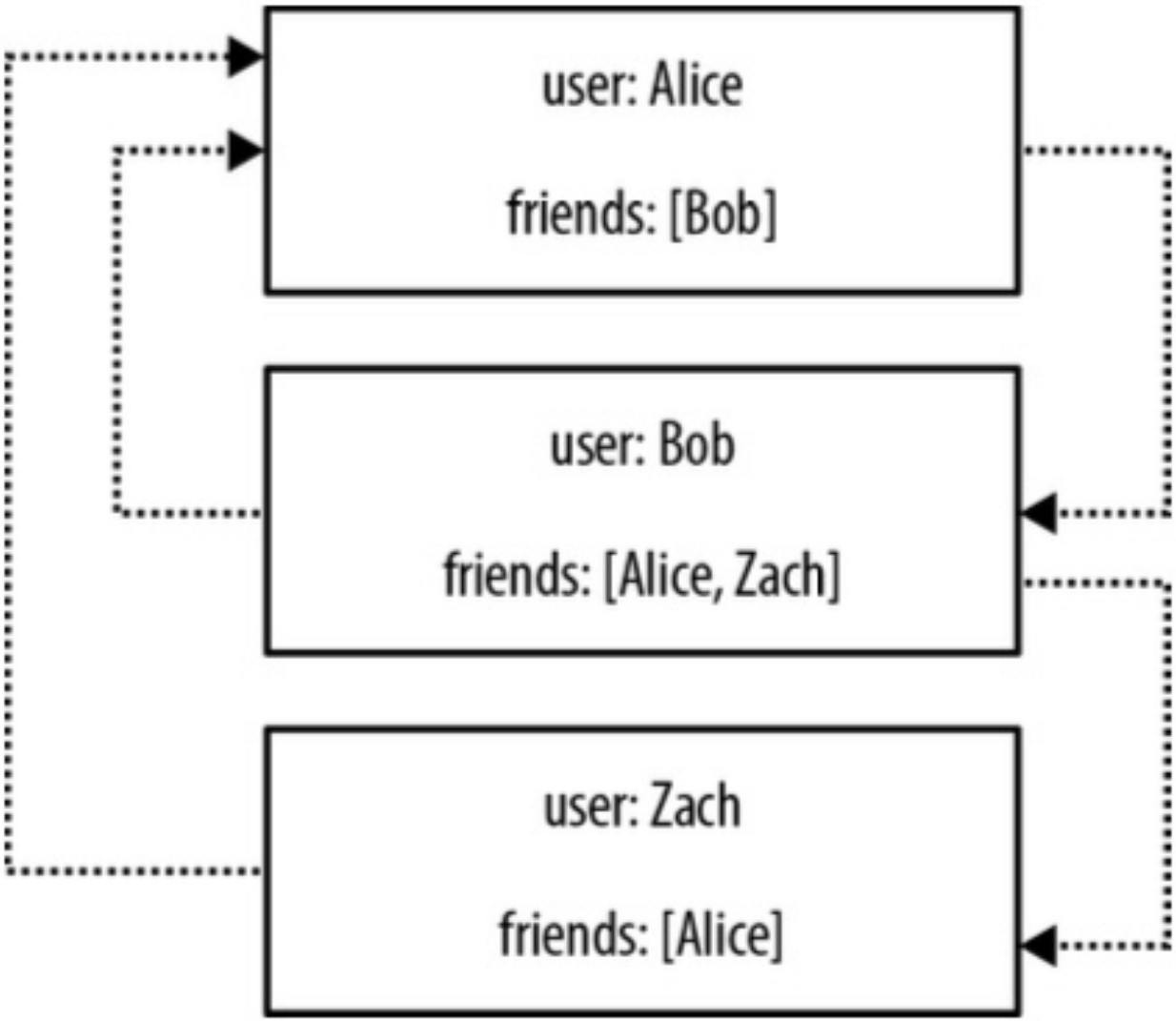
```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
     ON PersonFriend.PersonID = p1.ID
JOIN Person p2
     ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

Storing Connected Data: RDBMS

- Who are the friends of my friends?

```
SELECT p1.Person AS PERSON, p2.Person AS
FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
     ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
     ON pf2.PersonID = pf1.FriendID
JOIN Person p2
     ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID
<> p1.ID
```

Storing Connected Data: NoSQL



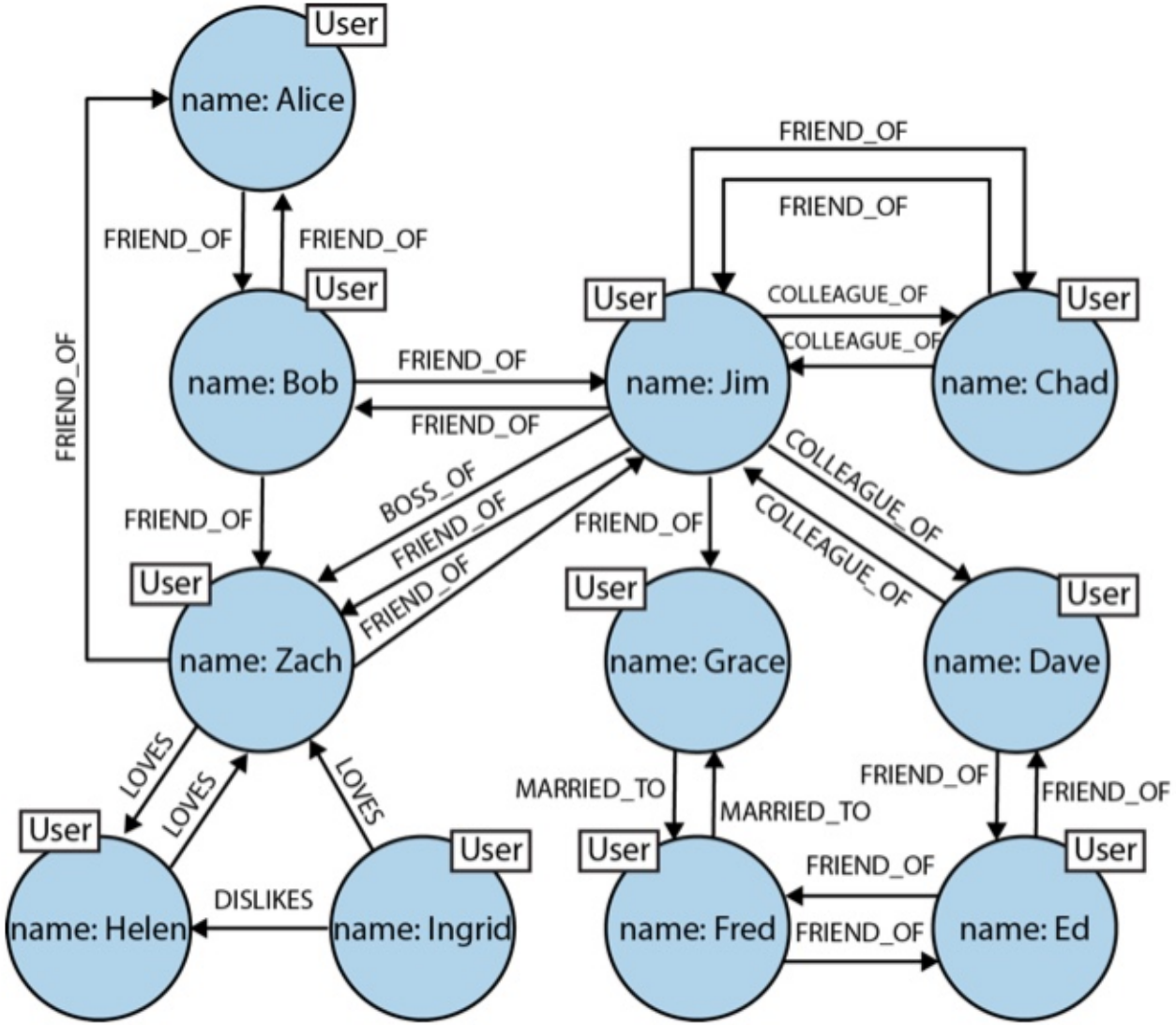
Storing Connected Data: NoSQL

- Who are Bob's friends?
  Straightforward.

- Who is friends with Bob?
  Scan across the whole dataset.

- Who are the friends of my friends?
  Multiple dataset scans.

Storing Connected Data: Graph DB



Storing Connected Data: Graph DB

- Constant-time lookup!
- Use index-free adjacency to ensure that traversing connected data is fast and efficient

Storing Connected Data: Comparison

Friends-of-friends request (1 million users, each has approx. 50 friends):

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

Vukotic, A., Watt, N., Abedrabbo, T.: Neo4J in Action (1st edition). Manning (2014)

Storing Connected Data: Summary

- Relational and NoSQL databases deal with *implicitly* connected data
- In Graph databases connected data is stored as *connected data*
  - Fast querying across records
- Major difference is in the use case:
  - A relational database may tell you how many books your read last year
  - A graph database will tell which books you and the friends of your friends have read in common

Graph Database

- Performance
  - RDBMS – join-intensive query performance decrease as the data gets bigger.
  - Graph DB – performance remains constant.
- Flexibility & agility
  - Schema-free nature of the graph data.
  - Can add new kinds of relations without disturbing existing queries and application functionality.
  - Don’t need to model our domain ahead of time.

Graph Database

- Index-free adjacency – each node maintain direct references to its adjacent nodes

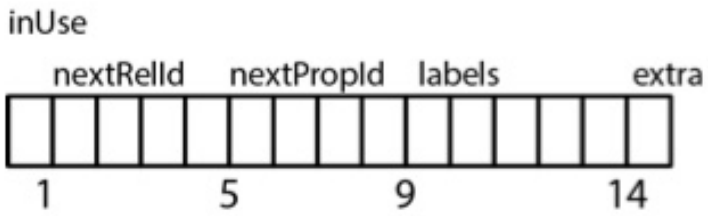
- Native Graph DB engine – exhibits index-free adjacency. Each node acts as a micro-index of other nearby nodes.
- Nonnative Graph DB engine – uses global index to link nodes together.

Graph Database Storage

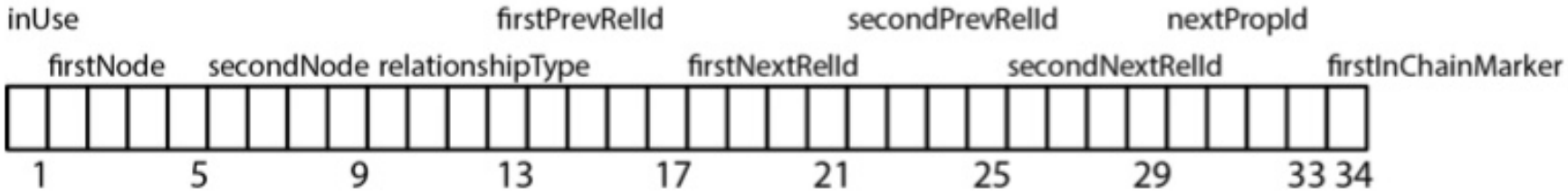
- Different types of data stored in different files (nodes, relationships, properties, labels)
- Records are fixed-size
- Node records point to lists of relationships, labels and properties
- Relationships stored in doubly linked lists
- Properties and labels stored in singly linked lists

Graph Database Storage

Node (15 bytes)



Relationship (34 bytes)



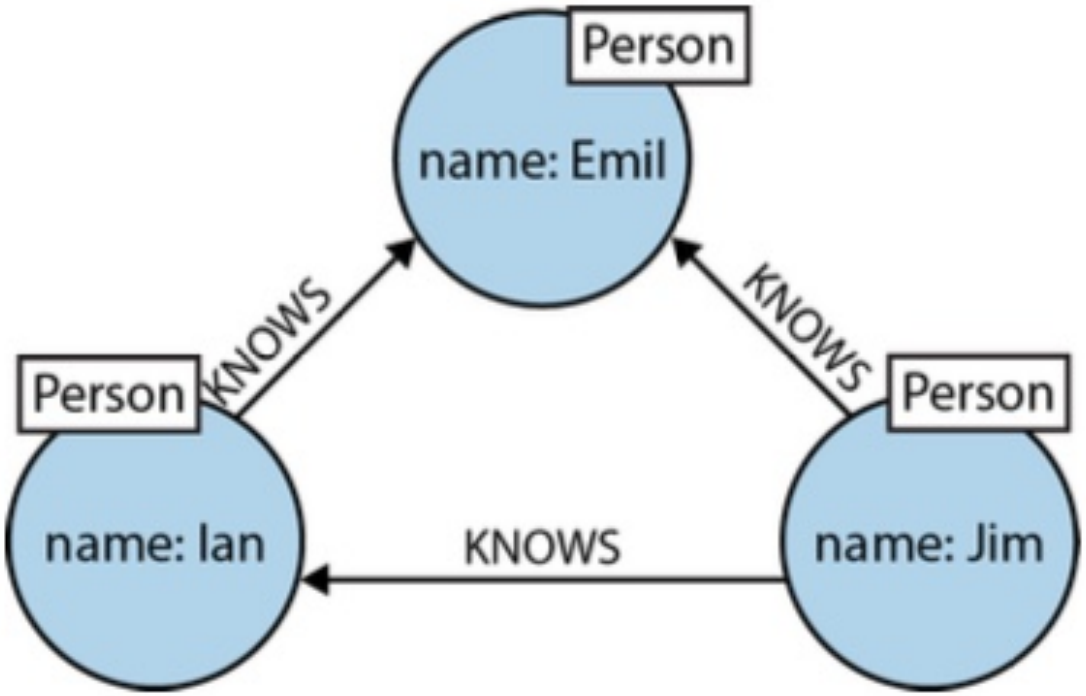
Query Languages

- **Cypher** (Neo4j)
- SPARQL (W3C)
- GraphQL (Facebook)
- Gremlin (Apache TinkerPop)

Introduction to Cypher

- Describes patterns in graphs visually using an ASCII-art syntax.
  - Nodes – surrounded with parenthesis (aka circles)
  - Relationships – expressed by arrow --> or <-- between two nodes, where < and > indicates the direction

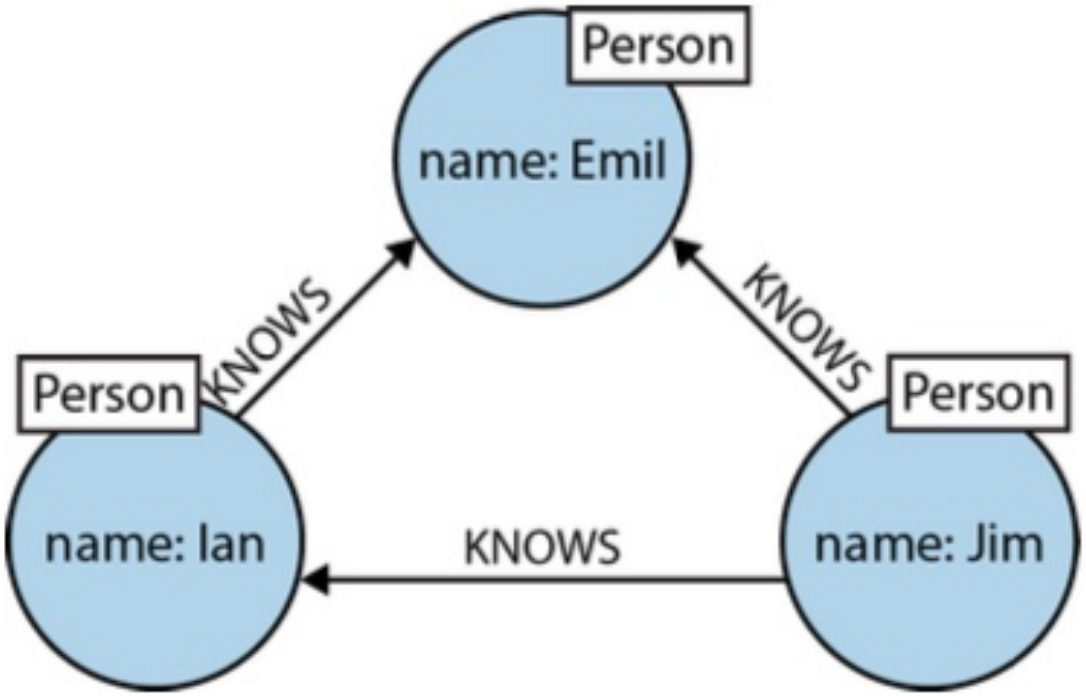
Introduction to Cypher



Pattern:

```
(emil)<-[:KNOWS]-(jim)-[:KNOWS]->(ian)-[:KNOWS]->(emil)
```

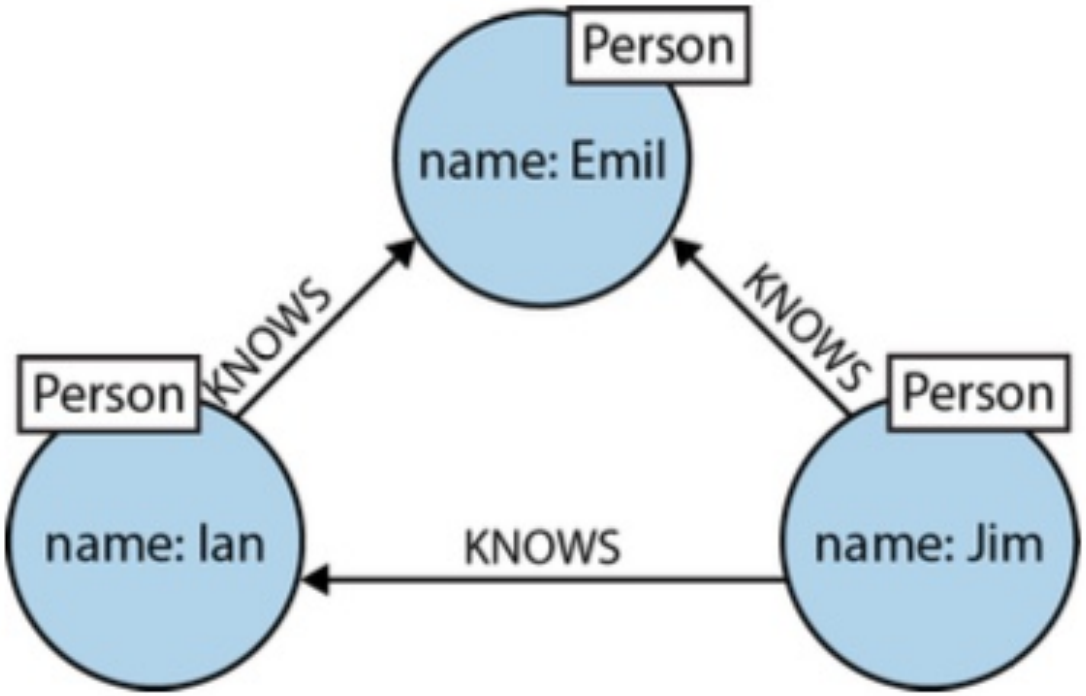
Introduction to Cypher



Localized pattern:

```
(emil:Person {name:'Emil'})<-[:KNOWS]-(jim:Person
{name:'Jim'})-[:KNOWS]->(ian:Person {name:'Ian'})-
[:KNOWS]->(emil)
```

Introduction to Cypher



Cypher query:

```
MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]-
>(c), (a)-[:KNOWS]->(c)
RETURN b, c
```

Introduction to Cypher: Syntax

- Node labels – prefixed by a colon.

```
(:Person)
```

- Relationship name – in square brackets inside an arrow, prefixed by a colon.

```
(:Person)-[:ATTEND]->(:Course)
```

- Properties – key-value pairs within curly braces.

```
(:Person {name:'Tom'})-[:ATTEND {semester:
'SA2018'}]->(:Course {title:'Databases'})
```

Introduction to Cypher: Syntax

- Nodes and relationships can be bound to identifiers

```
(tom:Person {name:'Tom'})-[attend:ATTEND
{semester: 'SA2018'}]->(db_course:Course
{title:'Databases'})
```

- We can specify distance between nodes

```
(tom:Person {name:'Tom'})-[:ATTEND
{semester: 'SA2018'}]->(db_course:Course
{title:'Databases'})-
[:UNIVERSITY|CITY*1..2]->(fribourg:City
{name:'Fribourg'})
```

Introduction to Cypher: Syntax

- Anonymous nodes – empty parenthesis

- We can constrain results of current match in WHERE clause

```
MATCH (s:Student)-[:ATTEND {semester:
'SA2018'}]->(db_course:Course
{title:'Databases'})
WHERE s.age < 25
```

- RETURN clause: `COUNT, ORDER BY, DISTINCT, LIMIT`

Introduction to Cypher: Clauses

- Reading clause:
  - MATCH – specify the patterns to search for in the database.
- Projecting clauses:
  - RETURN – define what to include in the query result set.
  - WITH – chain together query parts, piping the results from one to be used as starting points or criteria in the next.
- Reading sub-clauses:
  - WHERE – constrain the current pattern match.
  - ORDER BY [ASC|DESC] – specify that the output should be sorted. Follows RETURN or WITH
  - LIMIT – constrain the number of rows in the output.

Introduction to Cypher: Clauses

- Writing clauses:
  - CREATE – create nodes and relationships.
  - DELETE – remove nodes, relationships or paths.
  - SET – update labels and properties.
  - REMOVE – remove properties and labels.
  - FOREACH – update data within a list.
- Reading/writing clauses:
  - MERGE – create without duplicates.
  - CREATE UNIQUE – match what it can, and create what is missing

Introduction to Cypher: Clauses

- Writing clauses:
  - CREATE – create nodes and relationships.
  - DELETE – remove nodes, relationships or paths.
  - SET – update labels and properties.
  - REMOVE – remove properties and labels.
  - FOREACH – update data within a list.
- Reading/writing clauses:
  - MERGE – create without duplicates.
  - CREATE UNIQUE – match what it can, and create what is missing

Introduction to Cypher: Clauses

- Set operations:
  - UNION – combine the result of multiple queries into a single result set. Duplicates are removed.
  - UNION ALL – combine the result of multiple queries into a single result set. Duplicates are retained.
- Importing data:
  - LOAD CSV – import data from CSV files.
- Schema clauses:
  - CREATE | DROP CONSTRAINT – create or drop a constraint (label, relationship type, property).
  - CREATE | DROP INDEX – create or drop an index on all nodes with a particular label and property.