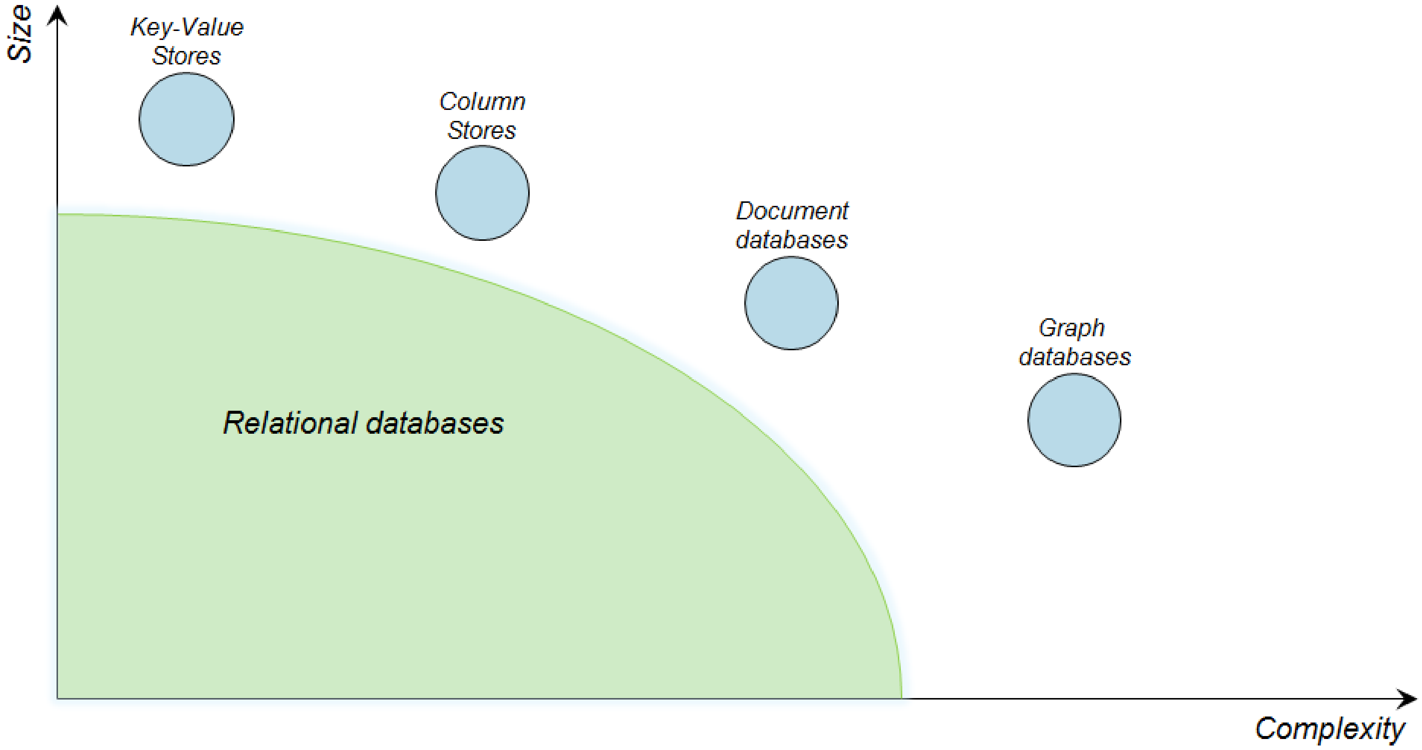
Introduction to Graph Databases

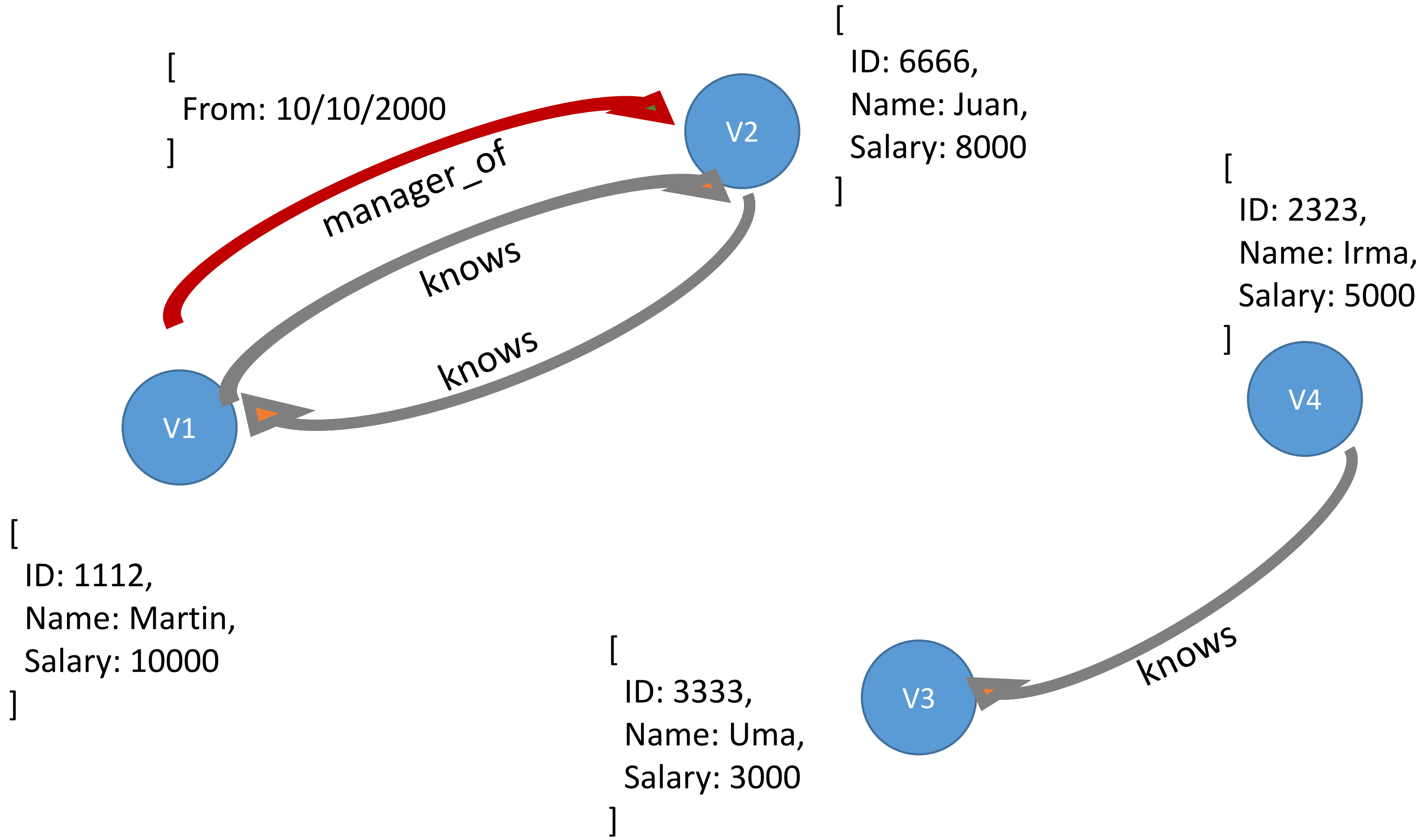
Fundamentals & Implememntations

Alejandro Vaisman
avaisman@itba.edu.ar

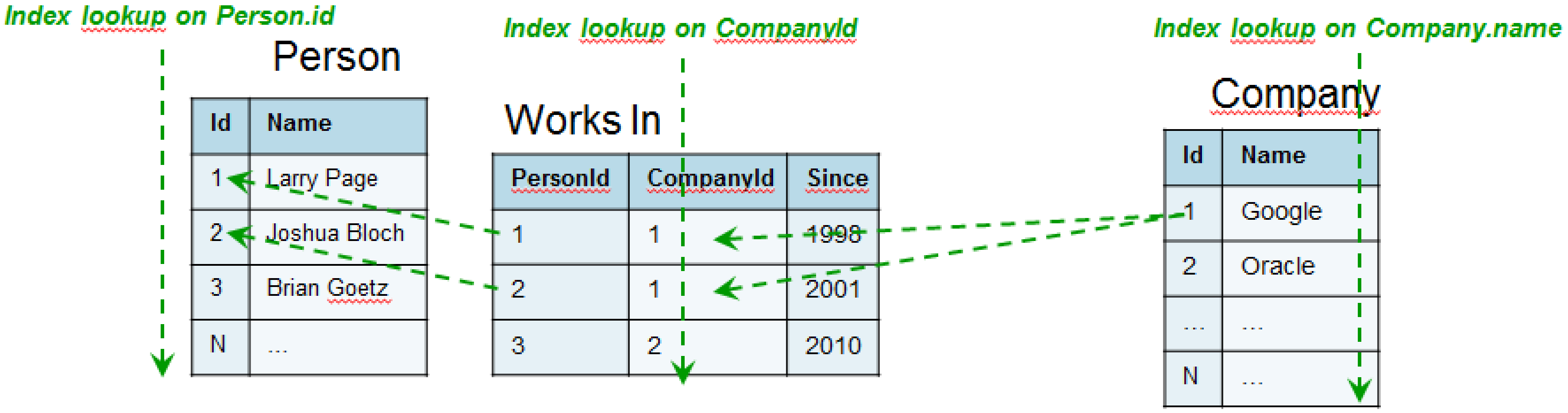
NoSQL databases



Property graphs revisited

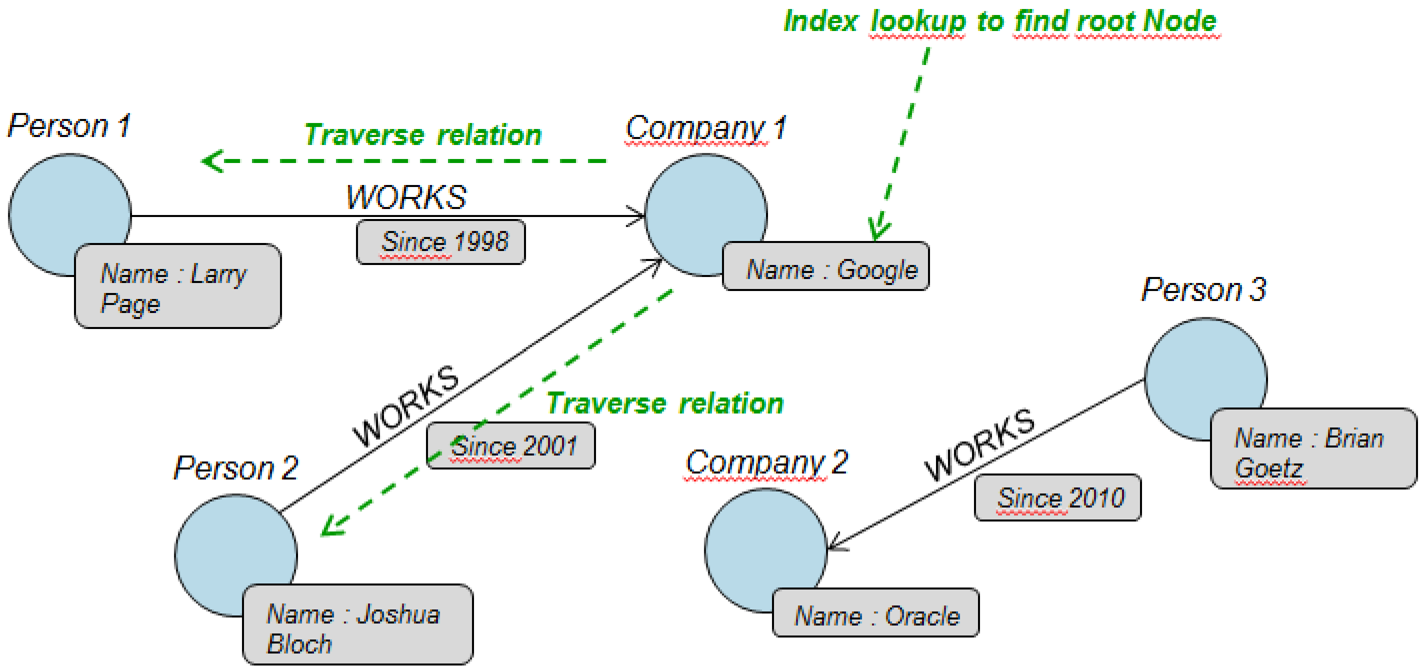


Typical SQL query



Select Person.Name
from Person, Company, WorksIn
where Company.name='Google'
and WorksIn.CompanyId = Company. Id
and WorksIn.PersonId = Person.Id

Same query on graphs



The deepest the navigation, the largest the difference with RDBs

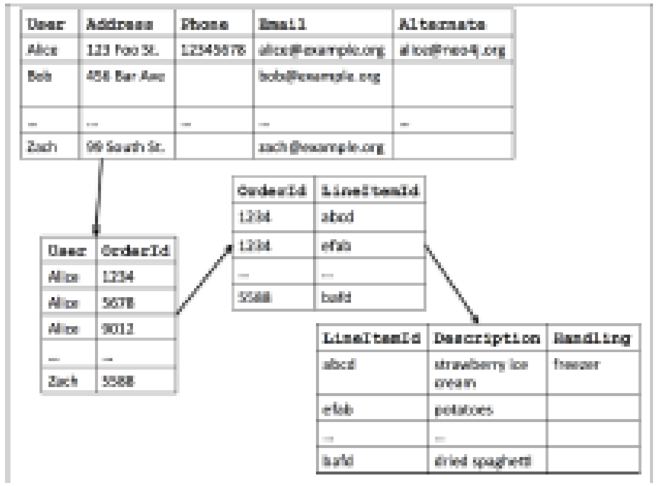
Traversal navigation: key to GDBs

- A graph traversal pattern is the ability to rapidly traverse structures to an arbitrary length (e.g., tree structures, cyclic structures), and with an arbitrary path description (e.g., Friends that work together, roads below a certain congestion threshold).

- Opposite to set theory, operated by means of relational algebra

Traversing data in a RDBMS

- Based on joining and selecting data



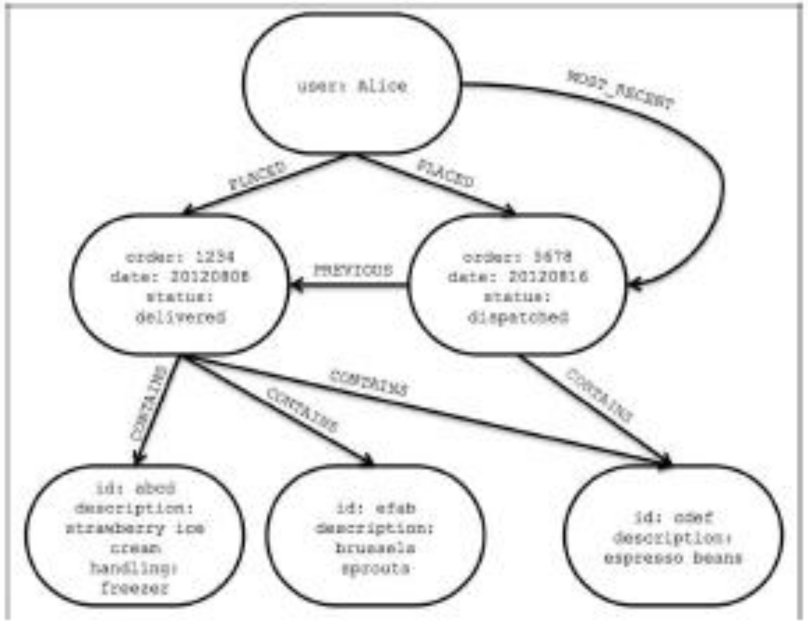

```
SELECT *
FROM user u, user_order uo,
orders o, items i
WHERE u.user = uo.user AND
uo.orderId = o.orderId AND
i.lineItemId = i.LineItemId
AND u.user = 'Alice'
```

Cardinalities:

|User|: 5.000.000
|UserOrder|: 100.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Query Cost?!

Traversing data in a GDB



Cardinalities:

|User|: 5.000.000
|Orders|: 1.000.000.000
|Item|: 35.000

Query Cost?!

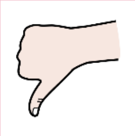
ReportsTo		
Boss	**Employee**	
A	C	Iteration 1
A	D	
C	E	
C	M	
E	T	
A	**E**	Iteration 2
A	M	
C	T	
A	**T**	Iteration 3

ReportsDirectlyTo	
Boss	**Employee**
A	C
A	D
C	E
C	M
E	T

We want to compute the closure of the relation "ReportsDirectlyTo", that is, to whom someone "ReportsTo", either directly or indirectly. SQL supports these kinds of recursive queries. Recursively joining ReportsTo and ReportsDirectlyTo on RT.Employee=RDT.Boss.

```
WITH recursive ReportsTo(Boss, Employee) AS
   (SELECT Boss, Employee
    FROM ReportsDirectlyTo
    UNION ALL
    SELECT ReportsTo.Boss, ReportsDirectlyTo.Employee
    FROM ReportsTo, ReportsDirectlyTo
    WHERE  ReportsTo.Employee = ReportsDirectlyTo.Boss )
SELECT * FROM ReportsTo
```

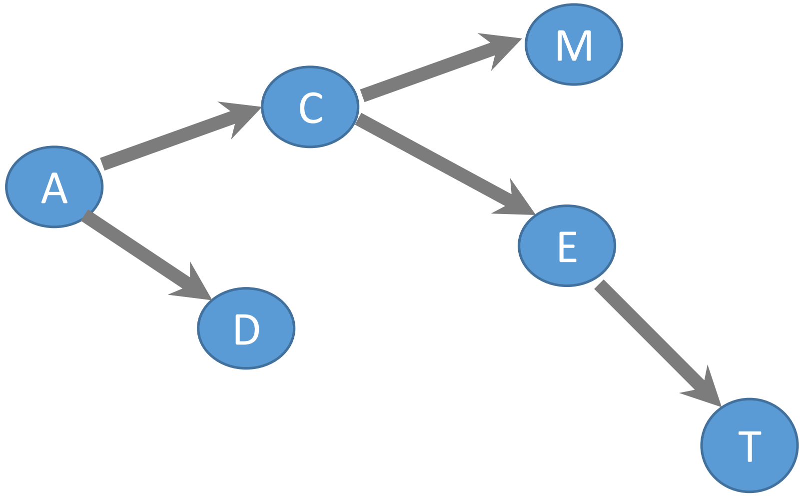
These queries are normally more expensive in the Relational Model, since they imply MULTIPLE JOINS.



Joins are expressed at the schema level rather than at the instance level.

How would we represent this in a Graph Data Model?

ReportsDirectlyTo	
Boss	**Employee**
A	C
A	D
C	E
C	M
E	T

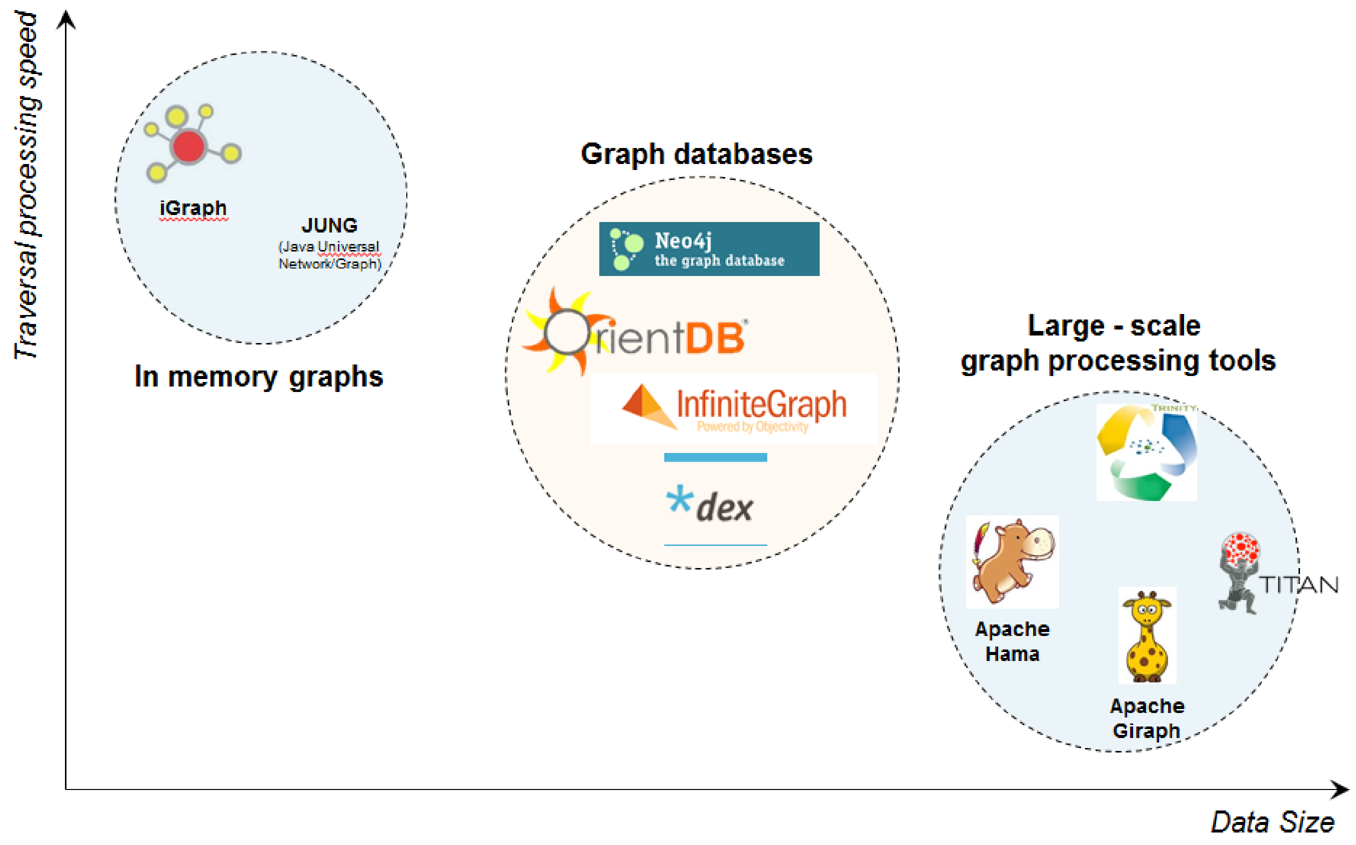


Paths in a graph are expressed at the instance level (there is no schema). Just check if there is an outgoing edge.

Graph storage vs graph processing

- Graph databases <> large-scale graph processing frameworks (e.g., Pregel from Google)
- Same data representation, but
  - graph processing tools focus on exploitation
  - graph databases focus on storage and transactions
- Graph databases scale vertically
- GBD problem: partitioning a graph is HARD
- Pregel executes distributed processing in commodity servers
- Two product families:
  - A GDB appropriate to traverse the graph, compute shortest paths
  - A graph processing framework adequate for clustering, graph mining, etc.

Graph DB vs. Graph processing



Graph database models

- Types of relationships supported by graph data models

Attributes
- Properties,
- Mono- or multi-valued.

Entities
- Groups of real-world objects.

Neighborhood relations
- Structures to represent neighborhoods of an entity.

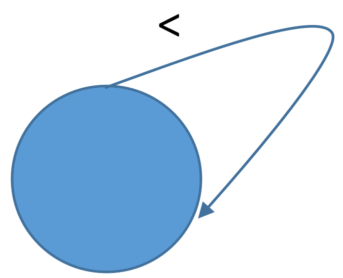
Standard abstractions
- Part-of, composed-by, n-ary associations.

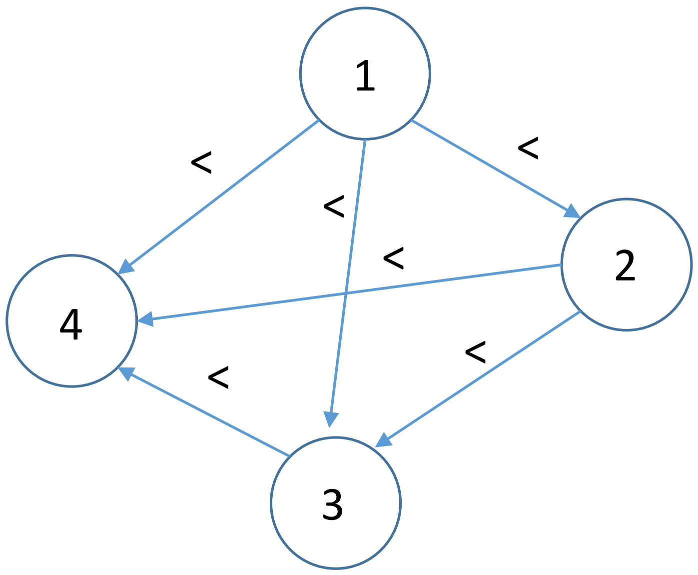
Derivation and inheritance
- Subclasses and superclasses,
- Relations of instantiations.

Nested relations
- Recursively specified relations.

Graph database models

- A graph is a binary relation. Ex:





Highest: out-degree = 0

Lowest: in-degree = 0

Ex: write a Relational Algebra Expression for the “<“ relation (or an SQL query without using ORDER BY or MAX/MIN)

The abstract data type Graph (w/properties)

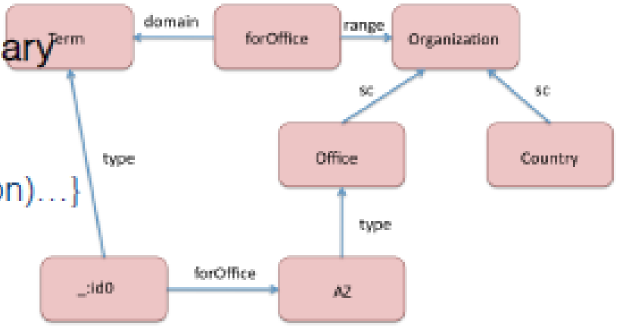
G=(V, E, Σ ,L) is a graph:

- V is a finite set of nodes or vertices,

  e.g. V={Term, forOffice, Organization,…}

- E is a set of edges representing binary relationship between elements in V,

  e.g. E={(forOffice,Term) (forOffice,Organization),(Office,Organization)…}



- Σ is a set of labels,

  e.g., Σ ={domain, range, sc, type, …}

- L is a function: V x V ➔ Σ ,

  e.g., L={((forOffice,Term),domain), ((forOffice,Organization),range)… }

The abstract data type Multigraph

G=(V, E, Σ ,L) is a multi-graph:

- V is a finite set of nodes or vertices,

  e.g. V={Term, forOffice, Organization,…}

- E is a set of edges representing binary relationship between elements in V,

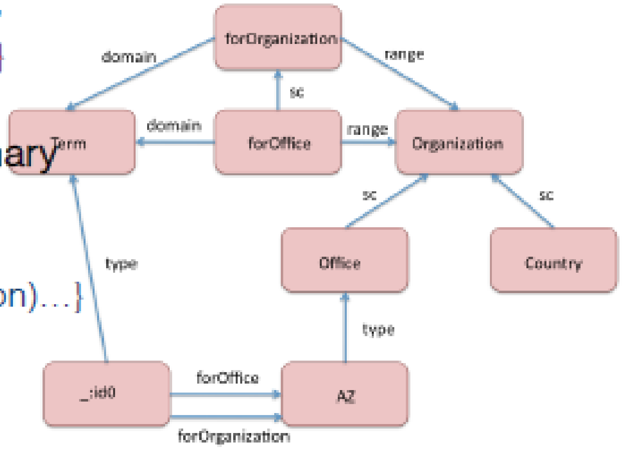
  e.g. E={(forOffice,Term) (forOffice,Organization),(Office,Organization)…}

- Σ is a set of labels,

  e.g., Σ ={domain, range, sc, type, …}

- L is a function: V x V ➔ PowerSet(Σ),

  e.g., L={((forOffice,Term),{domain}), ((forOffice,Organization),{range}), ((_id0,AZ),{forOffice, forOrganization})… }



Basic operations

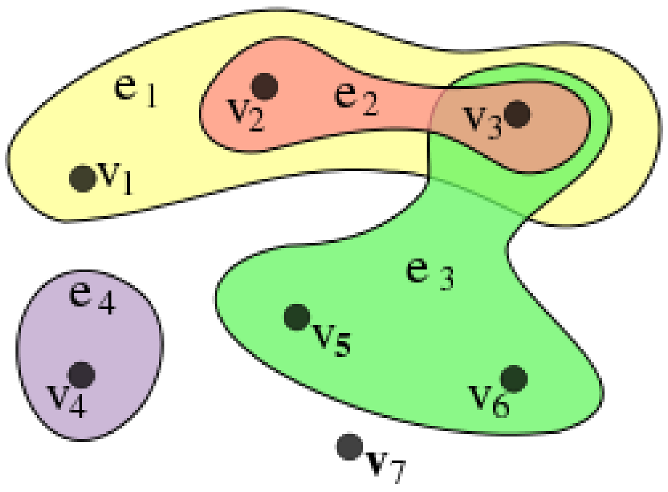
Given a graph G, the following are operations over G:

- AddNode(G,x): adds node x to the graph G.
- DeleteNode(G,x): deletes the node x from graph G.
- Adjacent(G,x,y): tests if there is an edge from x to y.
- Neighbors(G,x): nodes y s.t. there is an edge from x to y.
- AdjacentEdges(G,x,y): set of labels of edges from x to y.
- Add(G,x,y,l): adds an edge between x and y with label l.
- Delete(G,x,y,l): deletes an edge between x and y with label l.
- Reach(G,x,y): tests if there a path from x to y.
- Path(G,x,y): a (shortest) path from x to y.
- 2-hop(G,x): set of nodes y s.t. there is a path of length 2 from x to y, or from y to x.
- n-hop(G,x): set of nodes y s.t. there is a path of length n from x to y, or from y to x.

Graph generalization: (multi)Hypergraphs

H = (X , E), where X is a set of *nodes*, and E is a set of non-empty subsets of X called hyperedges **=>**
E $\subseteq$ P (X), where P (X) is the power set of X.

Undirected



$X = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E = \{e_1, e_2, e_3, e_4\} = \{\{v_1, v_2, v_3\}, \{v2,v3\}, \{v3, v5,v6\}, \{v_4\}\}$

Let X = (v*1*,...,v*n*), E = (*e1*,...,*em*).
Every hypergraph has an *m* x *n* incidence matrix $A = (a_{ij})$ where

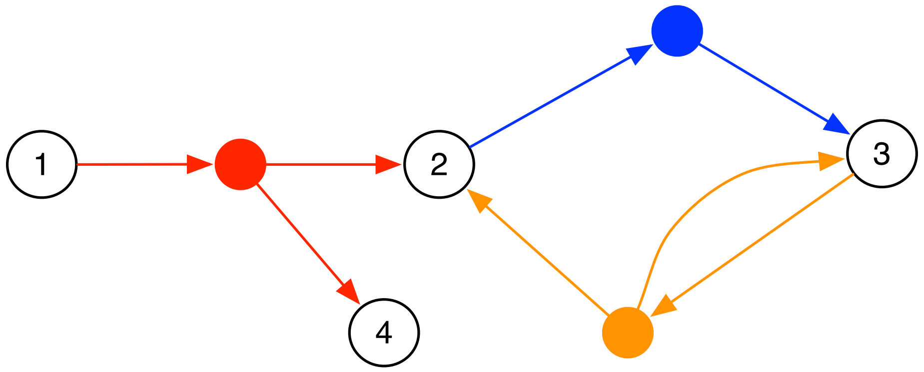
$$a_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise.} \end{cases}$$

	V1	v2	v3	v4	v5	v6	v7
e1	1	1	1	0	0	0	0
e2	0	1	1	0	0	0	0
e3	0	0	1	0	1	1	0
e4	0	0	0	1	0	0	0

Graph generalization: (multi)Hypergraphs

H = (X , E), where X is a set of *nodes*, and E is a set of non-empty subsets of X called hyperedges **=>** E is a subbag of P (X) x P(X), where P (X) is the power set of X.

Directed

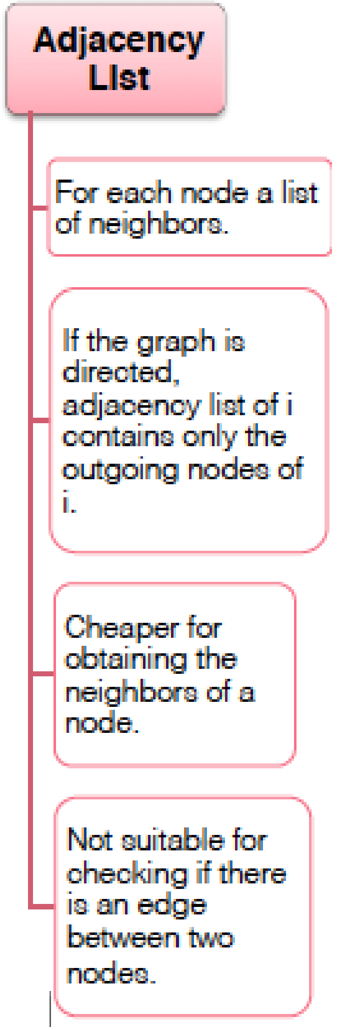


Graphically, S,T ⊆ X; A hyperedge is denoted S -> T

In the example:

X = {1,2,3,4}
E = {{1}->{2,4}, {2} -> {3}, {3} -> {2,3}}

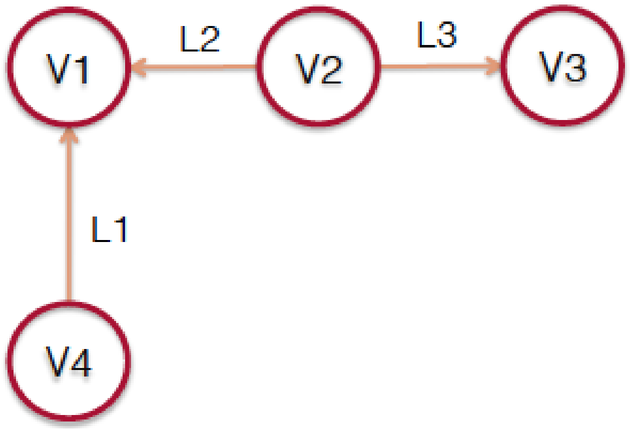
Implementation

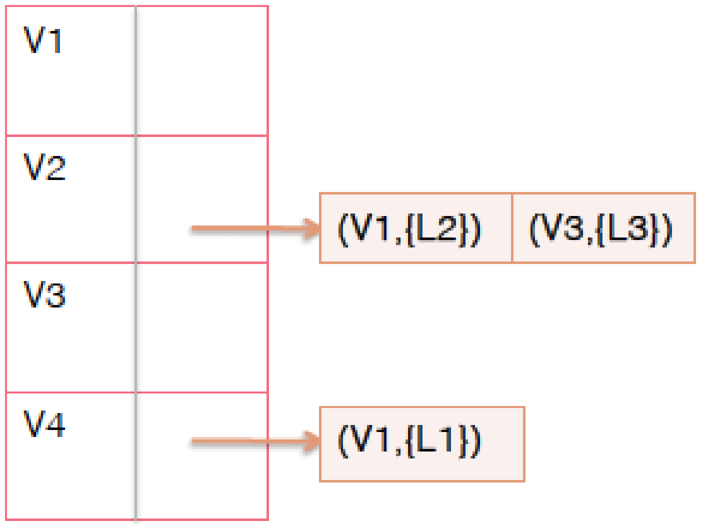


Implementation: adjacency list

Adjacency List

- For each node a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.



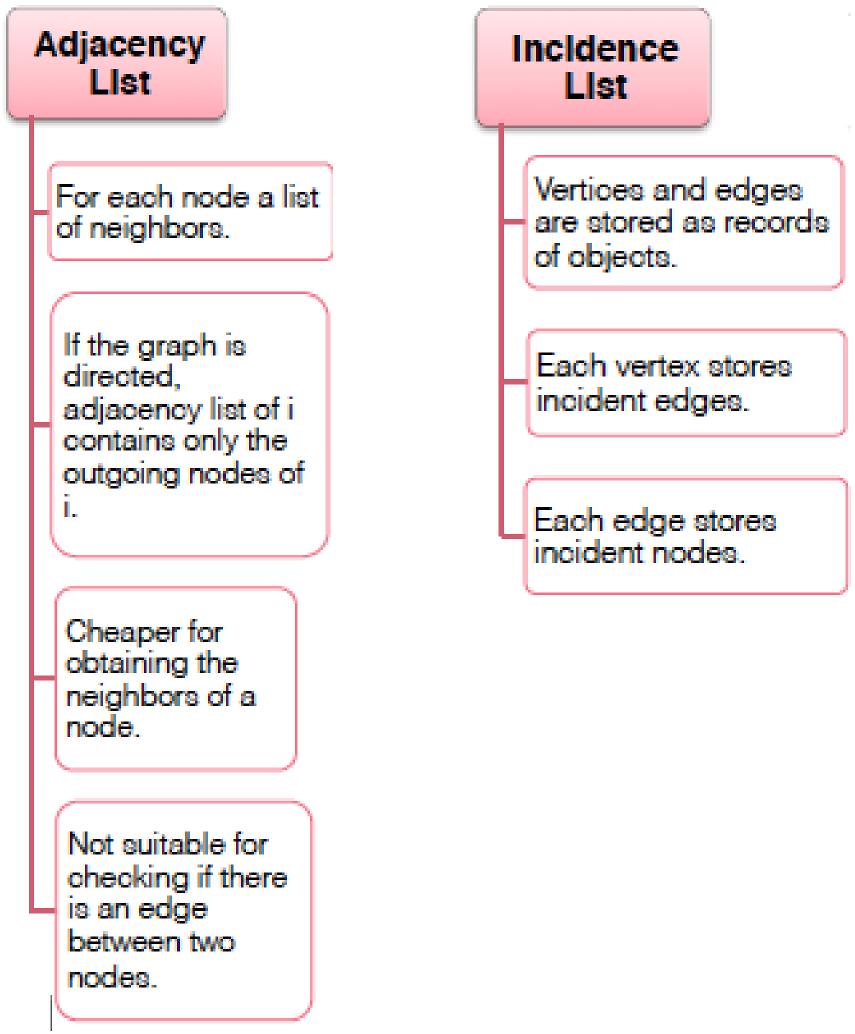


Implementation: adjacency list

Adjacency list of a directed graph

Call Adj an array of length |V|
Storage |V| X |E|
Is there a node between X and Y? O (V)
Out-degree of a vertex *u* = O (Adj[*u*]) = O (E) (worst case)
Out-degree for all vertices = O (V + E)
In-degree of a node = O (E)
In-degree of all vertices = O (V x E).
Alternative: allocate an array T of size |V | and initialize its entries to zero. Then scan the lists in Adj once, incrementing T[u] when we see u in the lists => O (V + E) time with (V) additional storage.

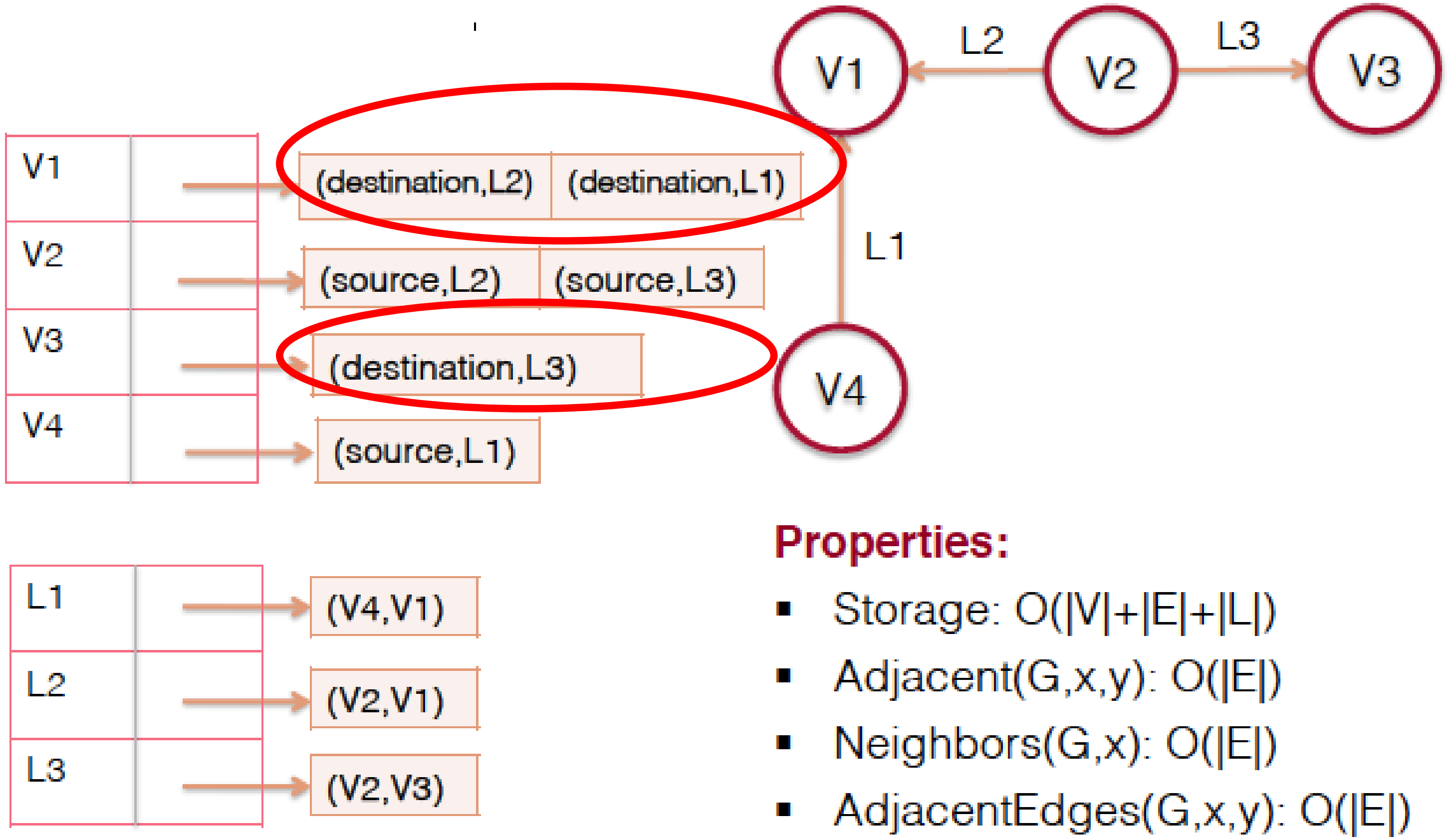
Implementation



Implementation: incidence list

Incidence List

- Vertices and edges are stored as records of objects.
- Each vertex stores incident edges.
- Each edge stores incident nodes.



Properties:

- Storage: O(|V|+|E|+|L|)
- Adjacent(G,x,y): O(|E|)
- Neighbors(G,x): O(|E|)
- AdjacentEdges(G,x,y): O(|E|)
- Add(G,x,y,l): O(|E|)
- Delete(G,x,y,l): O(|E|)

Implementation

Adjacency List
- For each node a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.

Incidence List
- Vertices and edges are stored as records of objects.
- Each vertex stores incident edges.
- Each edge stores incident nodes.

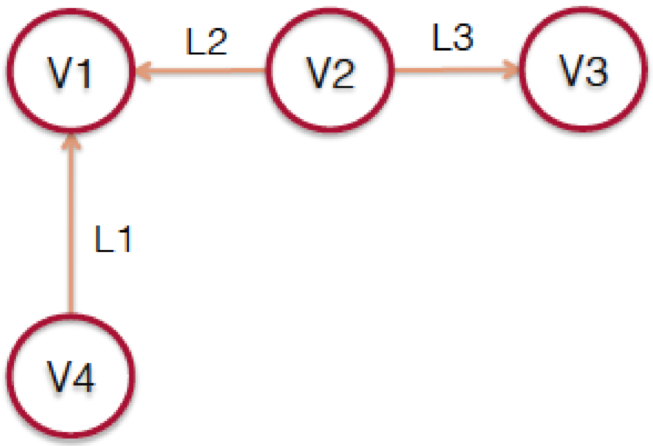
Adjacency Matrix
- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.

Implementation: adjacency matrix

Adjacency Matrix

- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.

	V1	V2	V3	V4
V1				
V2	{L2}		{L3}	
V3				
V4	{L1}			



Implementation: adjacency matrix

- Complexity
- Storage

Answer : |V| X |V|

- Is there an edge from X to Z?

Answer : O(1)

- Compute the out-degree of Z

Answer: O(|V|)

- Compute the in-degree of Z

Answer: O(|V|)

- Add an edge between two nodes

Answer: O(1)

- Compute all paths of length 4 between any pair of nodes (4-hop)

Answer: O($|V|^4$).

Implementation

Adjacency List

- For each node a list of neighbors.
- If the graph is directed, adjacency list of i contains only the outgoing nodes of i.
- Cheaper for obtaining the neighbors of a node.
- Not suitable for checking if there is an edge between two nodes.

Incidence List

- Vertices and edges are stored as records of objects.
- Each vertex stores incident edges.
- Each edge stores incident nodes.

Adjacency Matrix

- Bidimensional graph representation.
- Rows represent source vertices.
- Columns represent destination vertices.
- Each non-null entry represents that there is an edge from the source node to the destination node.

Incidence Matrix

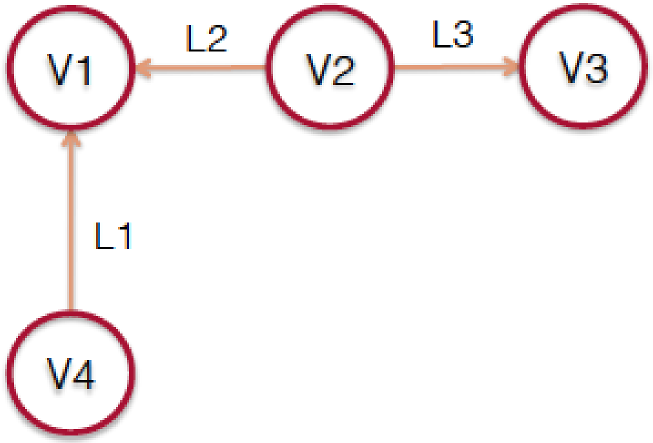
- Bidimensional graph representation.
- Rows represent vertices.
- Columns represent edges
- A non-null entry represents that the source vertex is incident to the edge.

Implementation: incidence matrix

Incidence Matrix

- Bidimensional graph representation.
- Rows represent vertices.
- Columns represent edges
- A non-null entry represents that the source vertex is incident to the edge.

	L1	L2	L3
V1	destination	destination	
V2		source	source
V3			destination
V4	source		

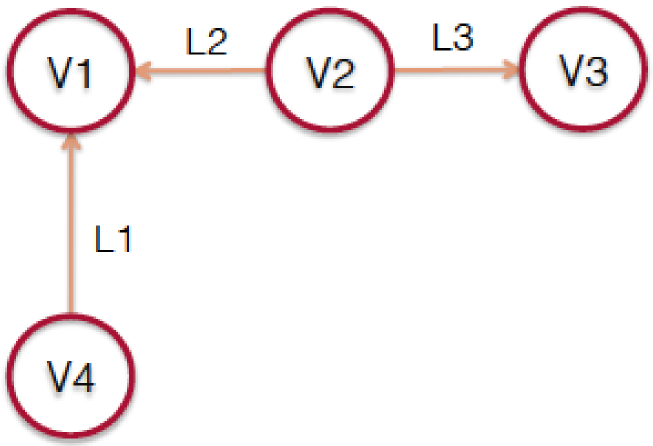


Implementation: incidence matrix

Incidence Matrix

- Bidimensional graph representation.
- Rows represent vertices.
- Columns represent edges
- A non-null entry represents that the source vertex is incident to the edge.

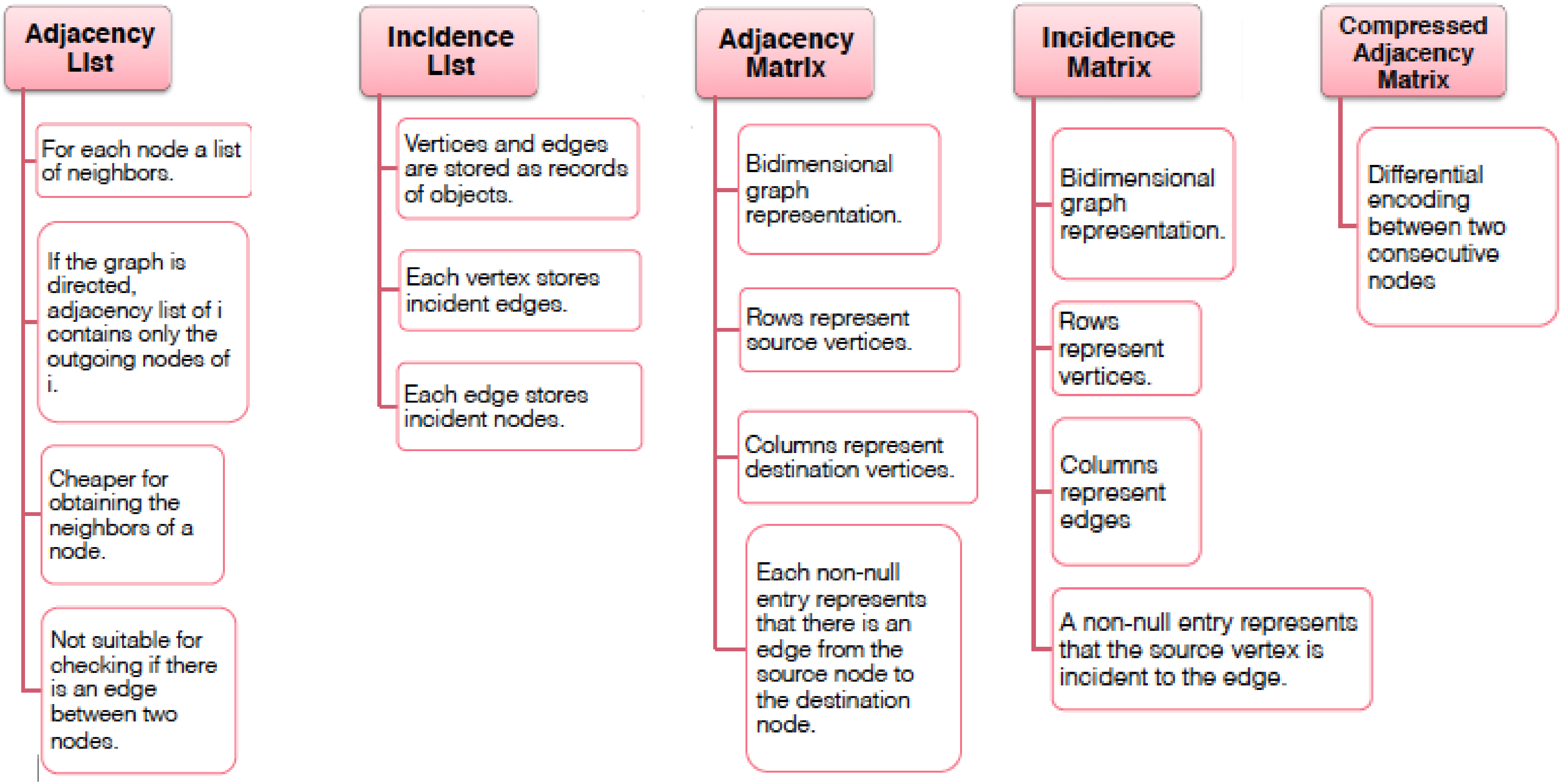
	L1	L2	L3
V1	destination	destination	
V2		source	source
V3			destination
V4	source		



Properties:

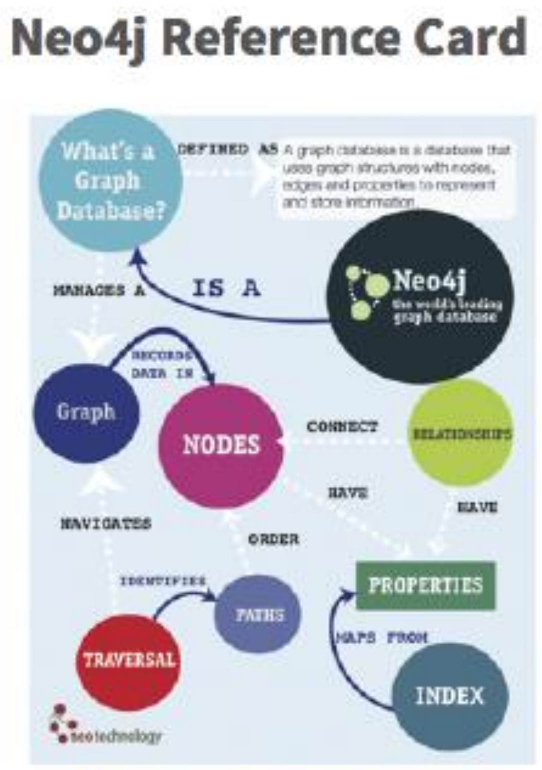
- Storage: O(|V|x|E|)
- Adjacent(G,x,y): O(|E|)
- Neighbors(G,x): O(|V|x|E|)
- AdjacentEdges(G,x,y): O(|E|)
- Add(G,x,y,l): O(|V|)
- Delete(G,x,y,l): O(|V|)

Implementation



Implementations

Graph databases – Representative approaches



http://www.neo4j.org

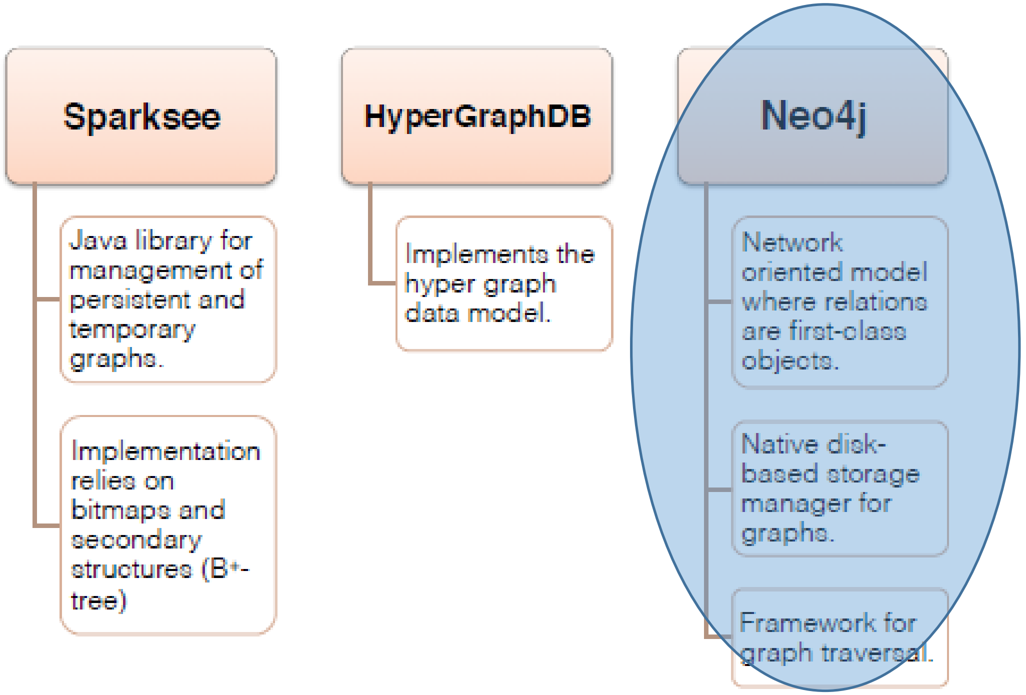
HyperGraphDB

http://www.hypergraphdb.org

*Sparksee

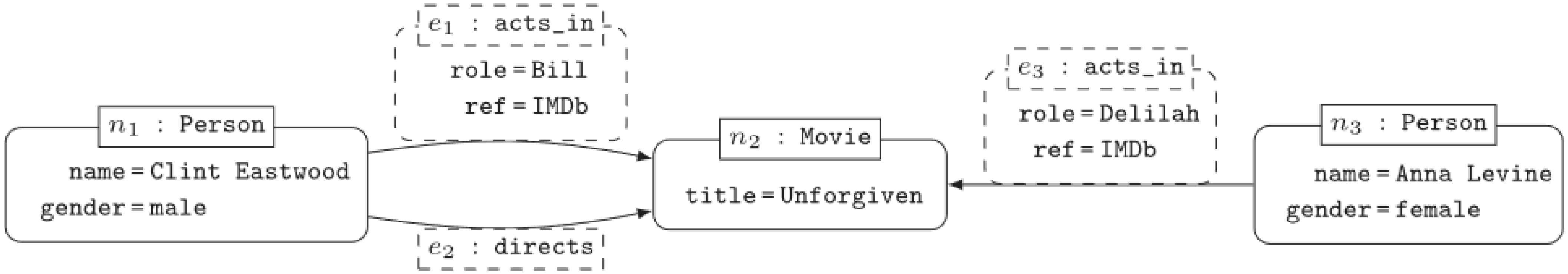
http://www.sparsity-technologies.com/

Some graph databases



- Some graph db implement an API rather than a query language

Property graph model again



Neo4j (Robinson et al., 2013)

- Labelled attributed multigraph
- Nodes and edges can have properties (property graphs)
- No restrictions on the # of edges between nodes
- Loops allowed
- Different types of traversal strategies
- APIs for Java and Python
- Embeddable and server
- Full ACID transactions

Neo4j (Robinson et al., 2013)

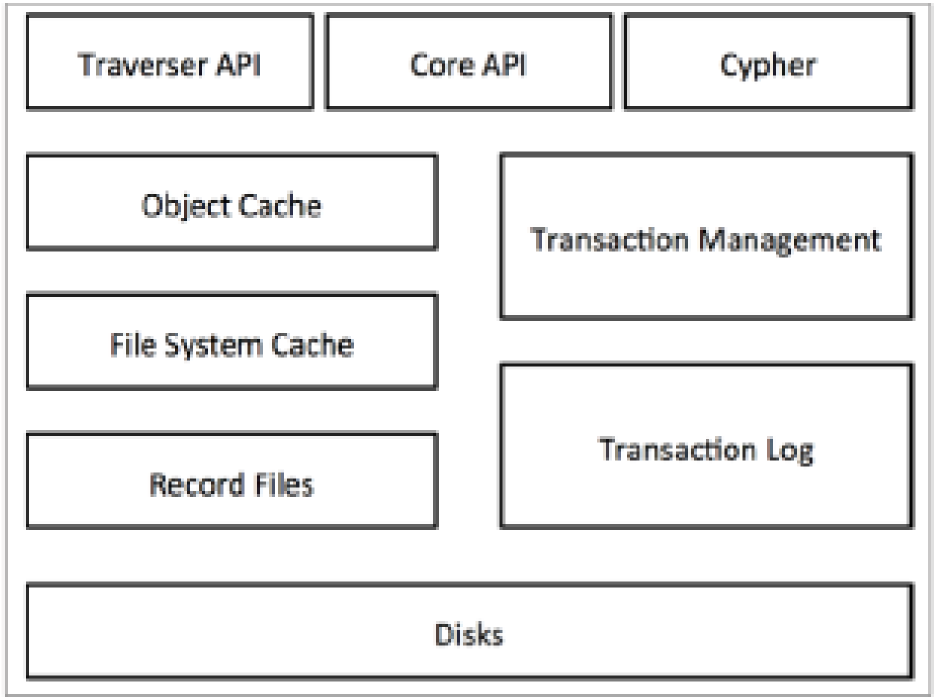
- Native graph processing and storage
  - Characterized by index-free adjacency:
    - Node keeps direct reference to adjacent nodes
    - Acts like a micro-index (or local index)
    - Makes query time independent from graph size for many queries
  - Joins are “precomputed” and stored as relationships
    - In non-native graph DBs, joins must be computed

Neo4j (Robinson et al., 2013)

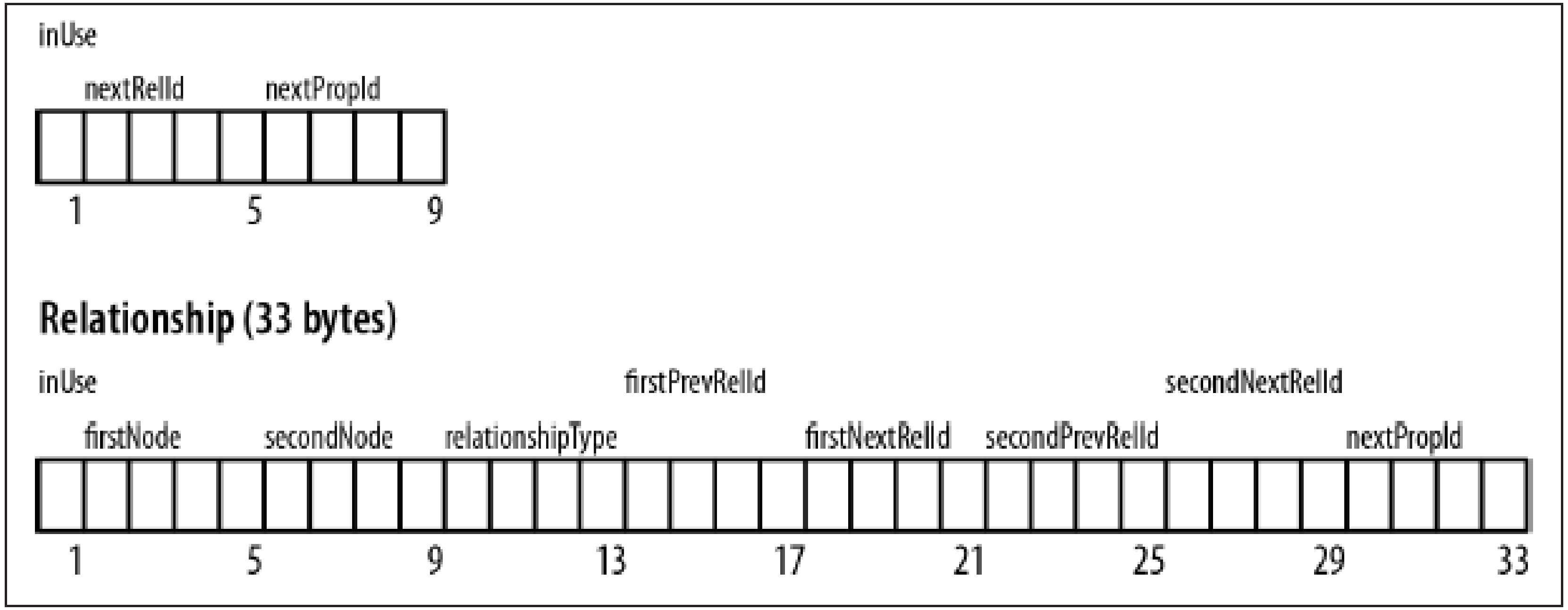
- Native graph storage
  - Storing graphs in files
  - Loading graphs into main memory
  - Caching graphs for fast querying

Neo4j - architecture

Robinson et al., 2013

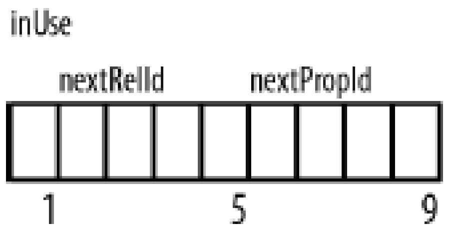


File storage



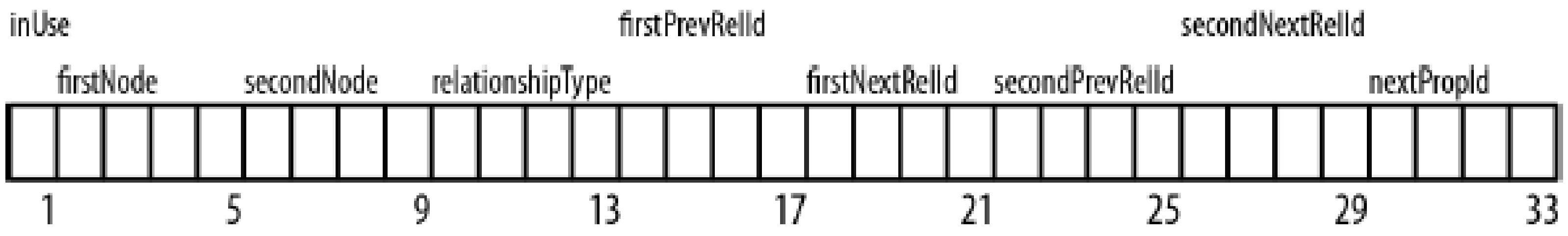
- Graphs stored in store files
    - Nodes (neostore.nodestore.db)
    - Relationships (neostore.relationshipstore.db)
    - Properties (neostore.propertystore.db)

File storage: nodes



- Stored in node records
  - Fixed length (9 bytes) to make search performant (find records with an offset from the node id)
    - Finding a node is O(1)
  - First byte: in-use flag
  - 4 bytes for the address of the first relationship
  - 4 bytes for the first property

File storage: relationships

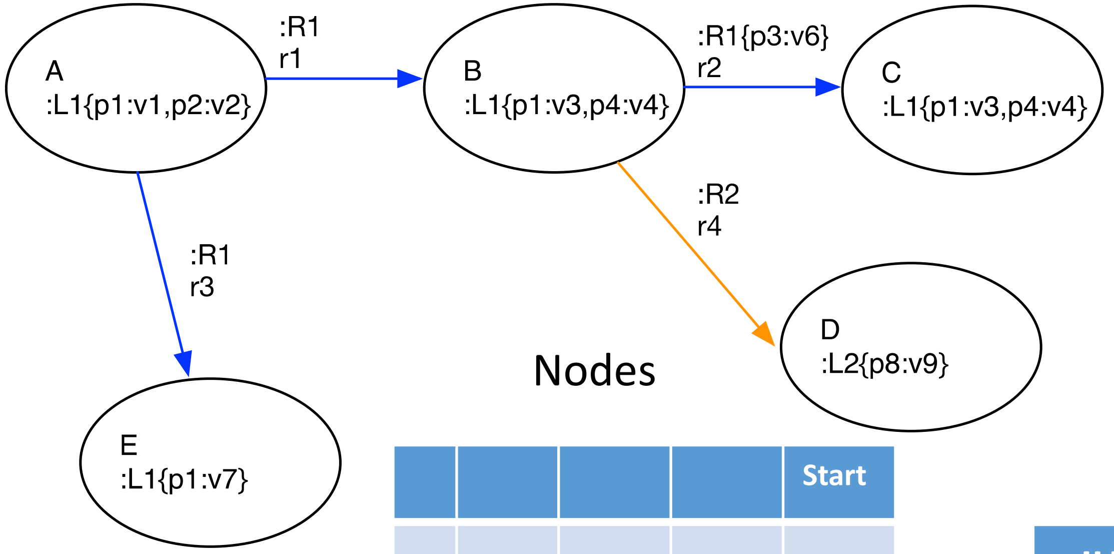


- Stored in relationship records
  - Fixed length (33 bytes)
  - First byte: in-use flag
  - Organized as a double-linked list
  - Each record contains the IDs of the two nodes in a relationship (start and end nodes)
  - A pointer to the relationship type
  - For each node, there is a pointer to the previous and next relationship records
    - E.g.: firstPrevRelID: previous relationship of the start node; firstNextRelID: next relationship of the start node (the one after the current relationship)
    - These form the relationship chain

File storage: properties

- Stored in property records
  - Fixed length
  - Each record consists of 4 property blocks and the ID of the next property in the property chain
  - Property chains: single-linked list
  - Each property: between 1 and 4 blocks
  - Each property record holds:
    - Property type
    - Pointer to the property index file, holding the property name
    - A value, or a pointer to a dynamic structure (string or array store)

File storage: example



Relationship Types

ID1	R1
ID2	R2

Properties

rp3	p3	v6	NIL
np1	p1	v1	np2
np2	p2	v2	NIL
np7	p1	v7	NIL
	...	...	...
	...	...	...

Nodes

				Start
1	A	**np1**	..	r1
2	B	..	..	r2
3	C	..	..	Nil
4	D	..	..	Nil
5	E	**np7**	..	Nil

Relationships

	IU	Fst	Snd	RT	FPrev	FNext	SPrev	SNext	NP
r1	1	A	B	ID1	NIL	r3	NIL	r2	NIL
r2	1	B	C	ID1	NIL	r4	NIL	NIL	rp3
r3	1	A	E	ID1	r1	NIL	NIL	NIL	NIL
r4	1	B	D	ID2	r2	NIL	NIL	NIL	NIL

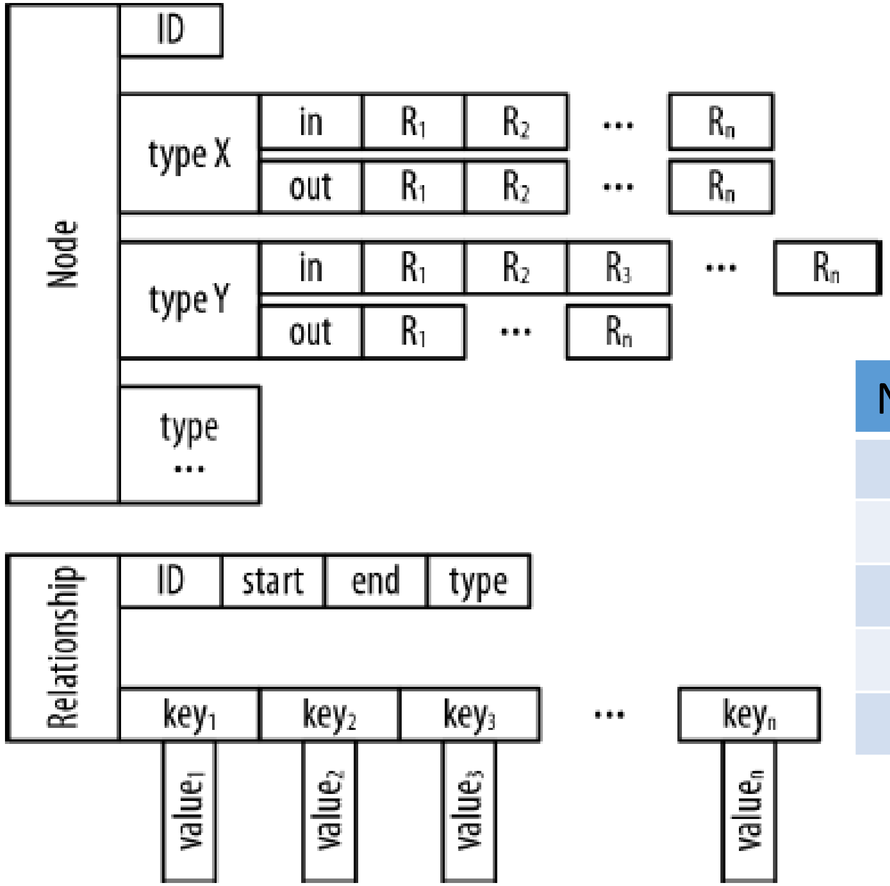
In a DFS, start from r1, then r2, r4, r3 (see table "Relationships"). We have all the information.

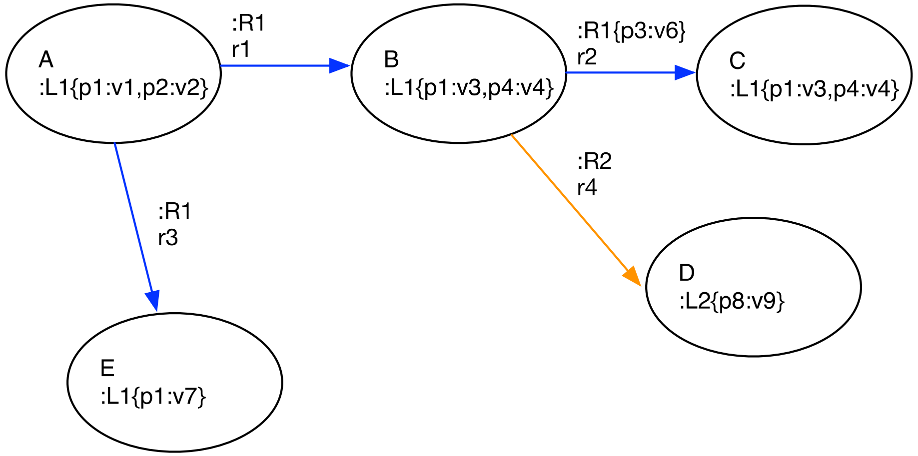
Caching

- File system cache (writing)
  - Cache divides each store into regions (pages)
  - Stores a fixed number of pages per file
  - Pages are replaced using Least Frequently Used pages

- Object cache
  - Optimized for reading
  - Stores object representations of nodes, relationships, and properties for fast path traversal
  - Node objects: contain properties and references to relationships
  - Relationship objects: contain only their properties
  - This is opposite to what happens in disk storage, where most information is in the relationship records

Object cache

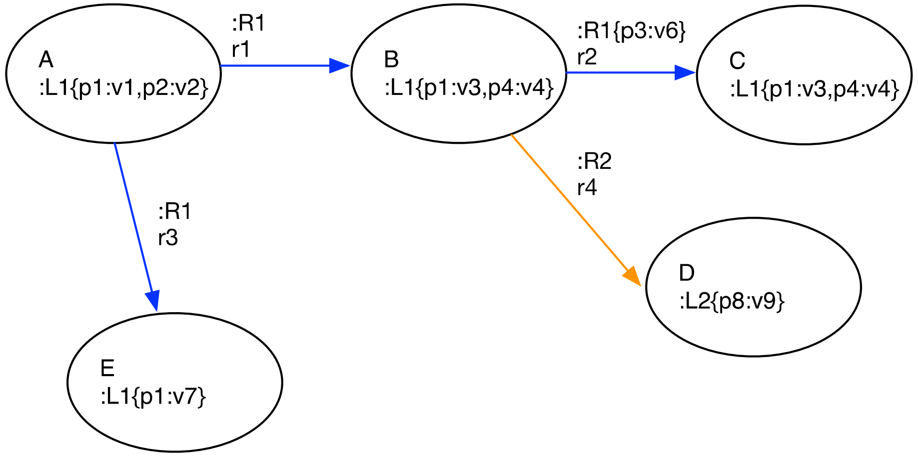




Node	Type	REL
n2	B	IN: r1
	B	OUT: r2
	B	OUT: r4
n3	C	IN: r2
…	…	…

REL	Start	End	Type
r1	A	B	R1
r2	B	C	R1
	(p3,v6)		
r3	A	E	R1
r4	B	D	R2

Traversal

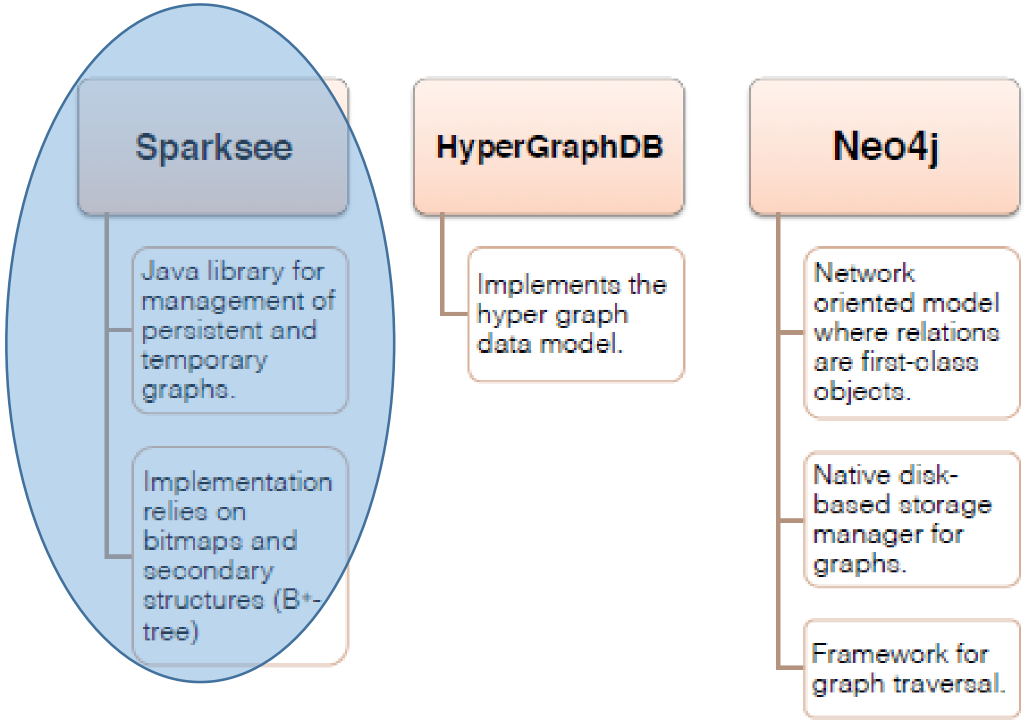


- Fetch node data from cache - non-blocking access
  - If not in cache, retrieve from storage, into cache
    - ▸If region is in FS cache: blocking but short duration access
    - ▸If region is outside FS cache: blocking, slower access
- Get relationships from cached node
  - If not fetched, retrieve from storage, by following chains
- Expand relationship(s) to end up on next node(s)
  - The relationship knows the node, no need to fetch it yet
- Evaluate
  - possibly emitting a Path into the result set
- Repeat

Node	Type	REL
n2	B	IN: r1
	B	OUT: r2
	B	OUT: r4
n3	C	IN: r2
…	…	…

REL	Start	End	Type
r1	n1	n2	R1
r2	n2	n3	R1
	(p3,v6)		
r3	n1	n5	R1
r4	n2	n4	R2

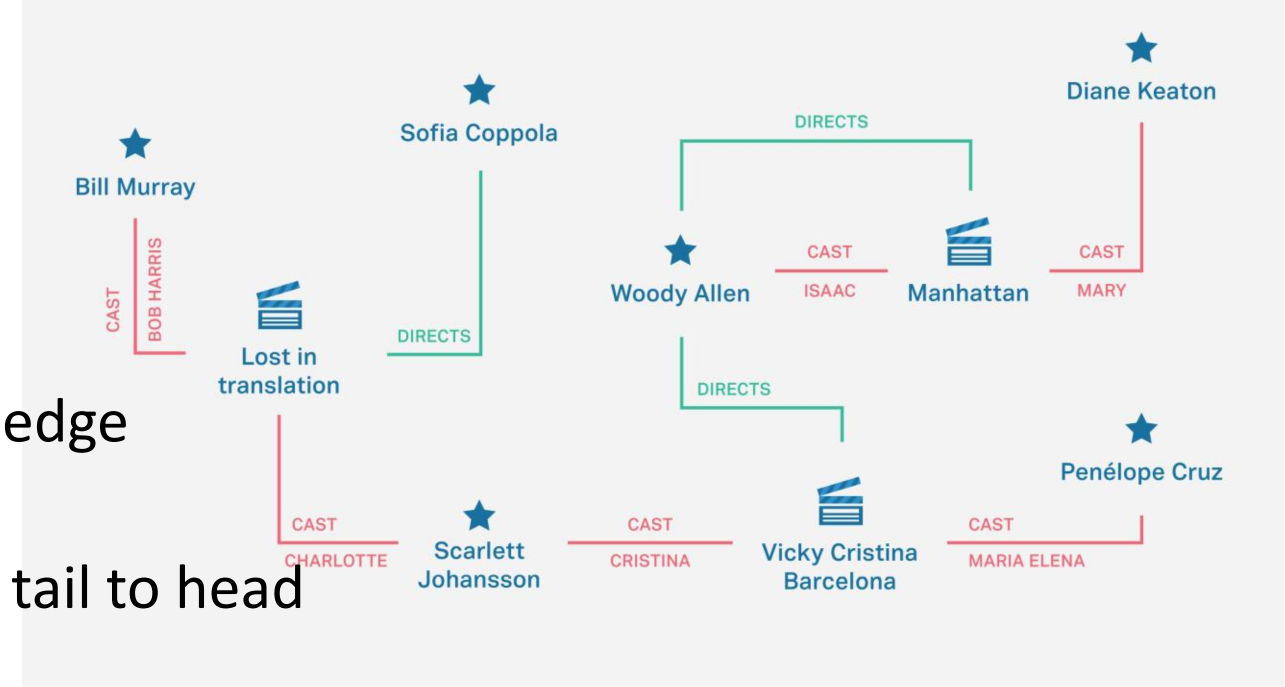
Some graph databases



- Some graph db implement an API rather than a query language

Sparksee

- Logical model
  - Labeled
    - a label for each vertex and edge
  - Directed
    - fixed direction edges, from tail to head
  - Attributed
    - variable # for each vertex)
  - Multigraph
    - possibly more than one edge between nodes
  - Embedded graph dbms
    - tightly integrated with the application at code level



Sparksee

- Nodes and edges have a sparksee-generated OID
- Node, edge and global attributes
  - Not restricted to an edge or node type (e.g., NAME can belong to all node objects)
  - Global attributes belong to the graph
- Attributes can have different **indexes**
    - Basic attributes
    - Indexed attributes
    - Unique attributes
    - Neighborhood index
- Persistent database in a single file
- Can manage very large graphs

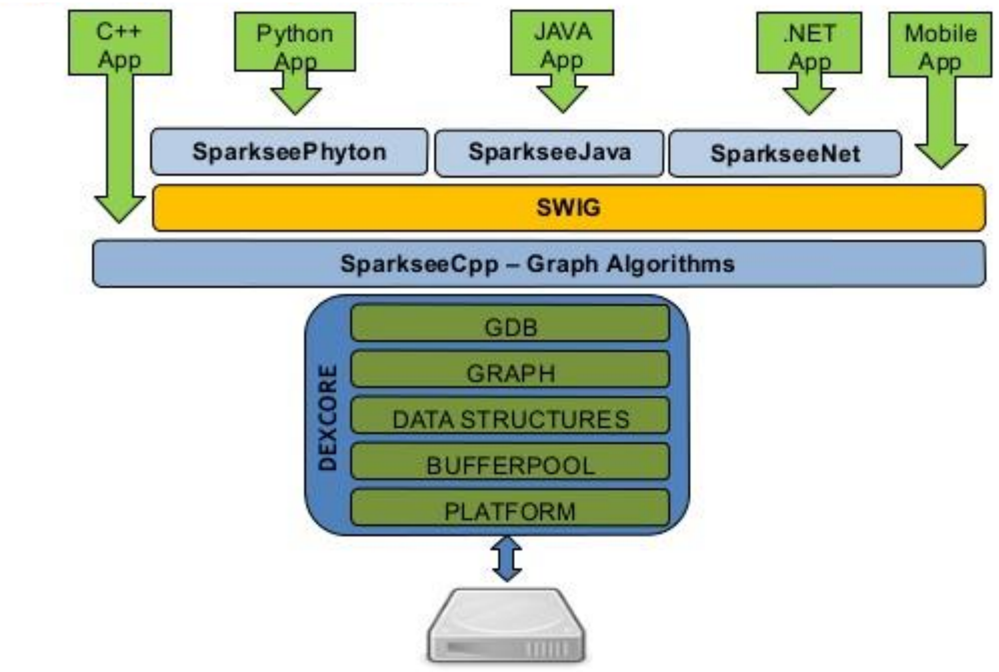
Sparksee

- A graph $G = (V,E,L,T,H, A1,\ldots\ldots,An)$ is defined as:

  - Labels $L = \{(o,l) \mid o \in (V \cup E) \wedge l \in \mathit{string}\}$
  - Heads $H = \{(e,h) \mid e \in E \wedge h \in V\}$
  - Tails $T = \{(e,t) \mid e \in E \wedge t \in V\}$
  - Attributes $Ai = \{(o,c) \mid o \in (V \cup E) \wedge c \in (\mathit{int},\mathit{string},\ldots\}$

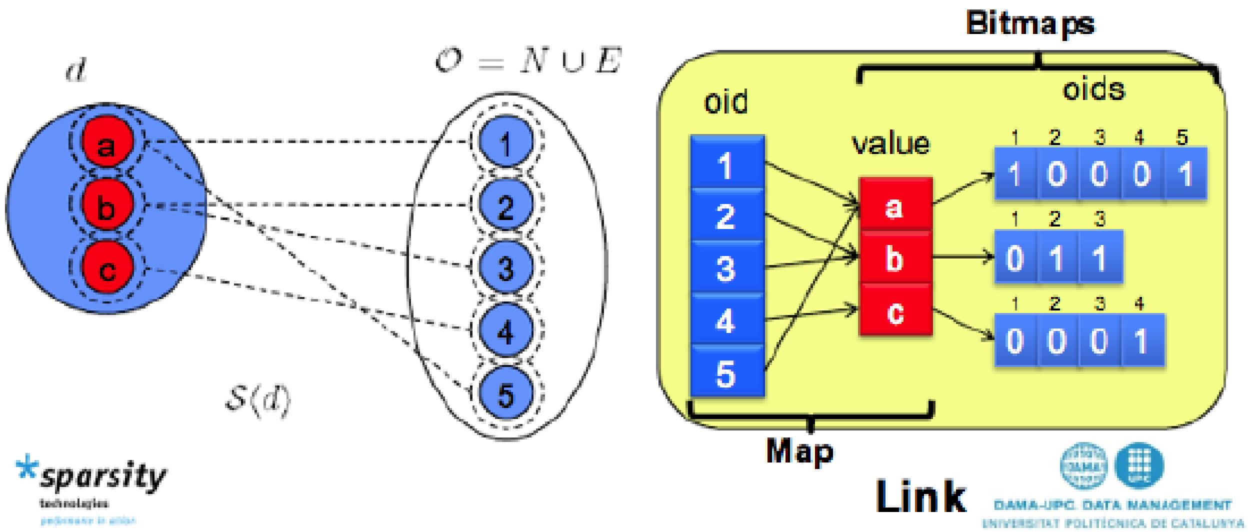
  - The graph is split into multiple lists of pairs
  - The first element in a pair is always an edge or a vertex

Sparksee - architecture



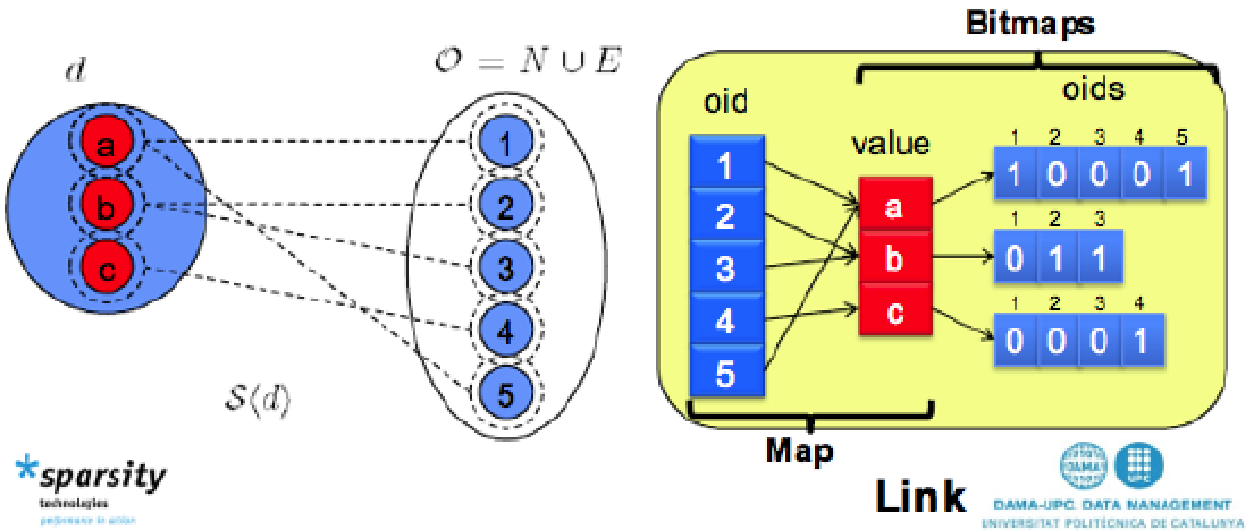
*SWIG = Simplified Wrapper and Interface Generator. Open source tool used to connect programs/libraries written in C/C++ with other languages.

Sparksee – internal representation



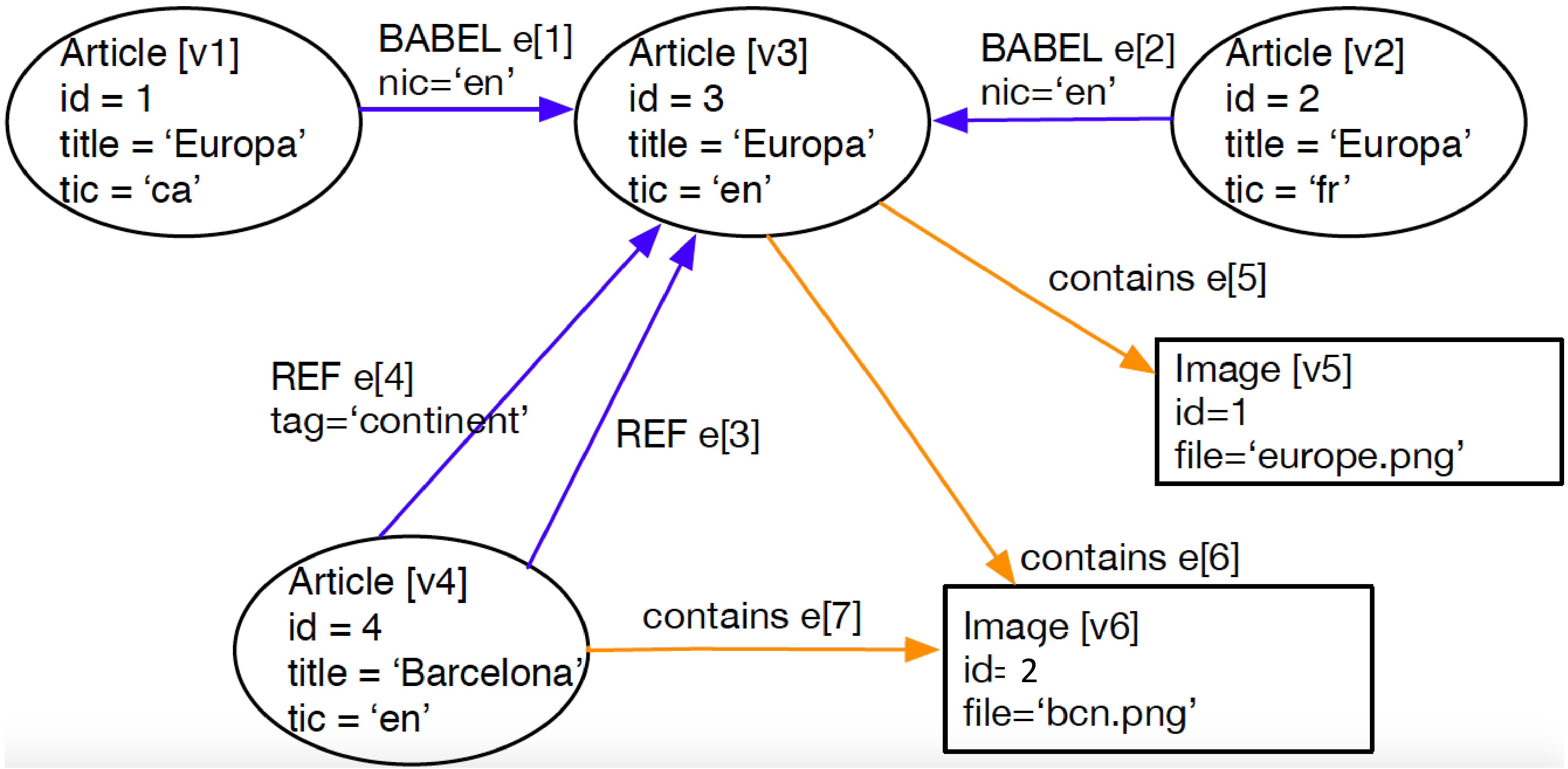
- Each vertex/edge is identified with an immutable oid.
- Links: bidirectional
  - Value - > set of OIDs.
  - Given an OID -> a value.
- Two maps: (a) from a value to a vertex or edge set; (b) from a vertex or edge to an oid.
- Maps are B-trees.

Sparksee – internal representation

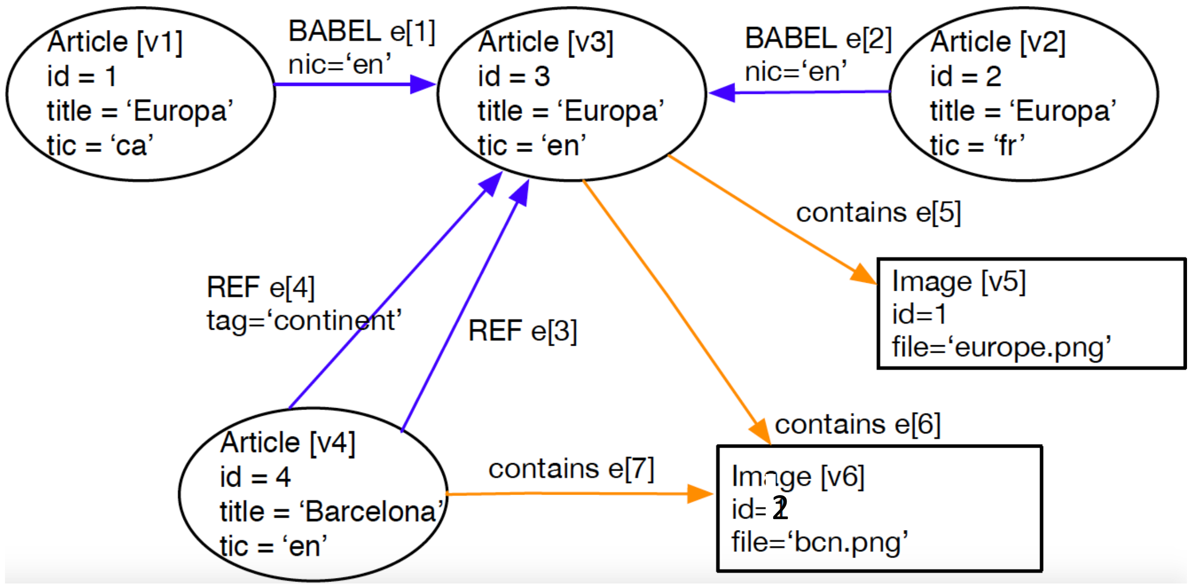


- A Sparksee Graph is a combination of **Bitmaps**:
  - Bitmap for each node or edge set (type).
  - Each position in the bitmap corresponds to the oid.
  - One link for each attribute.
  - Two links for each type: Outgoing and in-going edges.
- Maps are B+trees
  - A compressed UTF-8 storage for UNICODE string.

Sparksee – example



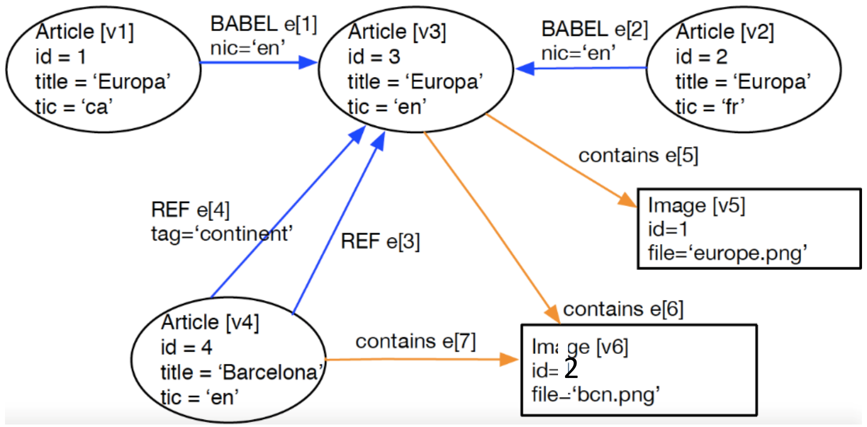
Sparksee – example

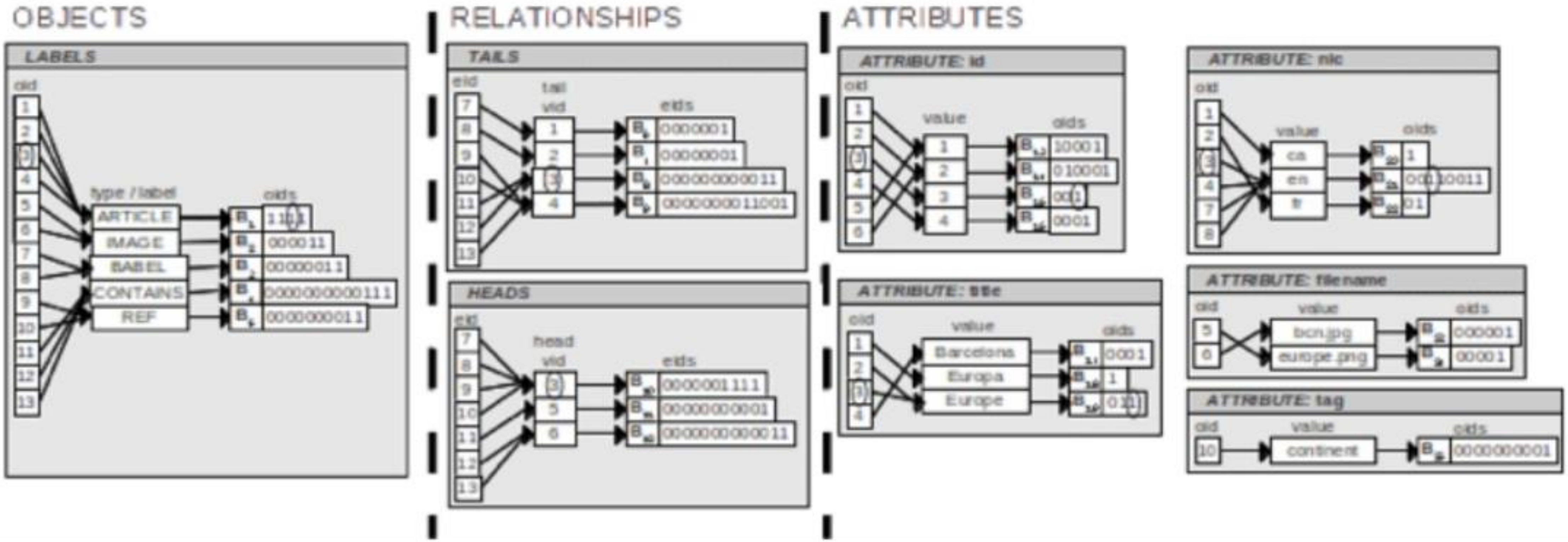


Value sets: group all pairs of the original set with the same value, as a pair between the value and the set of objects with that value

L	v_1, ARTICLE), (v_2, ARTICLE), (v_3, ARTICLE), (v_4, ARTICLE), (v_5, IMAGE), (v_6, IMAGE), (e_1, BABEL), (e_2, BABEL), (e_3, REF), (e_4, REF), (e_5, CONTAINS), (e_6, CONTAINS), (e_7, CONTAINS)	(ARTICLE, {v_1, v_2, v_3, v_4}), (BABEL, {e_1, e_2}), (CONTAINS, {e_5, e_6, e_7}), (IMAGE, {v_5, v_6}), (REF, {e_3, e_4})
T	(e_1, v_1), (e_2, v_2), (e_3, v_4), (e_4, v_4), (e_5, v_3), (e_6, v_3), (e_7, v_4)	(v_1, {e_1}), (v_2, {e_2}), (v_3, {e_5, e_6}), (v_4, {e_3, e_4, e_7})
H	(e_1, v_3), (e_2, v_3), (e_3, v_3), (e_4, v_3), (e_5, v_5), (e_6, v_6), (e_7, v_6)	(v_3, {e_1, e_2, e_3, e_4}), (v_5, {e_5}), (v_6, {e_6, e_7})
Aid	(v_1, 1), (v_2, 2), (v_3, 3), (v_4, 4), (v_5, 1), (v_6, 2)	(1, {v_1, v_5}), (2, {v_2, v_6}), (3, {v_3}), (4, {v_4})
Atitle	(v_1, Europa), (v_2, Europe), (v_3, Europe), (v_4, Barcelona)	(Barcelona, {v_4}), (Europa, {v_1}), (Europe, {v_2, v_3})
Anlc	(v_1, ca), (v_2, fr), (v_3, en), (v_4, en), (e_1, en),(e_2, en)	(ca, {v_1}), (en, {v_3, v_4, e_1, e_2}), (fr, {v_2})
Afilename	(v_5, europe.png), (v_6, bcn.jpg)	(bcn.jpg, {v_6}), (europe.png, {v_5})
Atag	(e_4, continent)	(continent, {e_4})

Sparksee – example





Sparksee – example

Graph query examples

— Number of articles

$$|objects\ (LABELS, \text{'ARTICLE'})|$$

— Out-degree of English article 'Europe'

$$|objects\ (TAILS, objects(\ TITLE, \text{'Europe'}) \cap\ objects\ (NLC, \text{'en'}) \cap\ objects\ (LABELS, \text{'ARTICLE'}))|$$

— Articles with references to the image with filename 'bcn.jpg'

$$\{lookup(TAILS, x\ )\ |x \in objects\ (HEAD, objects\ (FILENAME, \text{' bcn.jpg'})\ \cap\ objects\ (LABELS, \text{' IMAGE'})))\}$$

— Count the articles of each language

$$\{(x\ ,\ y\ )\ |\ x \in domain(NLC) \wedge y = |(objects\ (NLC, x\ ) \cap objects\ (LABELS, \text{' ARTICLE'})))|\}$$