Graph stores

Ioana Manolescu ; Silviu Maniu
INRIA Saclay ; Université Paris-Sud

M2 Data and Knowledge
Université de Paris Saclay

Motivation

- Graphs correspond to a natural organization of knowledge
- They generalize
  - Relations
  - Trees (documents)
  - Key-value pairs …
- Graph stores simplify / facilitate data representation
- They do not simplify query evaluation (and may make it more complex)

Graph database models

- Graph = N (nodes) and E (edges, subset of E x E)
- Directed vs. undirected edges
- Nodes:
  - Unlabeled
  - With a single label (in some cases called *type*)
  - With a set of attribute-value pairs
  - With complex internal structure (persistent objects)
- Graphs may have semantics (RDF, RDFS)

Object-oriented databases

- 1980 – 2000 (approx)
- Idea: capitalize on the flexibility of OO programming languages such as C++ and Java to handle databases of persistent objects
- Object Database Management Group (ODMG): consortium of OODB vendors which produced a standard
  1. Object Model // classes, attributes, methods…
  2. Object Definition Language (ODL)
     // persistency roots (persistent collections)
  3. Object Query Language (OQL)
     // navigation from one object to its attribute
     // method invocation
     // structured query language
  4. C++ and Java Bindings

Sample OQL queries

- **select** a.number

  from a in ATM_MACHINE.accounts_list

  where a.balance > 0
- **select** max(**select** c.age **from** p.children c) // nested queries
  from Persons p
  where p.name = "Paul"
- **select** p.oldest_child.address.street
  from Persons p
  where p.lives_in("Paris") // method invocation
- **select** ((Student)p).grade // explicit type test
  from Persons p
  where "course of study" in p.activities // set attribute

Where are OODBs now?

- Object-oriented extensions are present in all major (relational) databases → Object-Relational Database Management Systems (ORDBMS)
  - Mostly relational
  - Modest but useful object extensions
- E.g. complex types in Postgres:
  - **create type** inventory_item **as** (name text, supplier_id integer, price numeric);
  - **create table** on_hand (item inventory_item, count integer);
  - **insert into** on_hand **values** (ROW('fuzzy dice', 42, 1.99), 1000);

Working with composite type in the Postgres ORDBMS

create type inventory_item **as** (name text, supplier_id integer, price numeric);

create table on_hand (item inventory_item, count integer);

select (on_hand.item).name // ( ) specific to composite type
from on_hand
where (on_hand.item).price > 9.99;

This would have required a join in a classical RDBMS!

create type complex as (r double precision, i double precision);

insert into mytab (complex_col) **values** ((1.1,2.2));

update mytab **set** complex_col = row(1.1,2.2) where ...;

The first (graph) semistructured data model: OEM [PGW95]

OEM: Object Exchange Model, introduced as a global data model for mediator systems

E.g. scenario where several product databases are integrated under a unique global schema

- Some have one **price**, some have several (e.g. price reductions)
- Some have a **description**, some have a **technical_description**, some have **description.text**, **description.price**…
- Some have a **photo**, some do not

OEM: Labeled, directed, unordered **graph** of **objects**

Every **object** has a unique identity

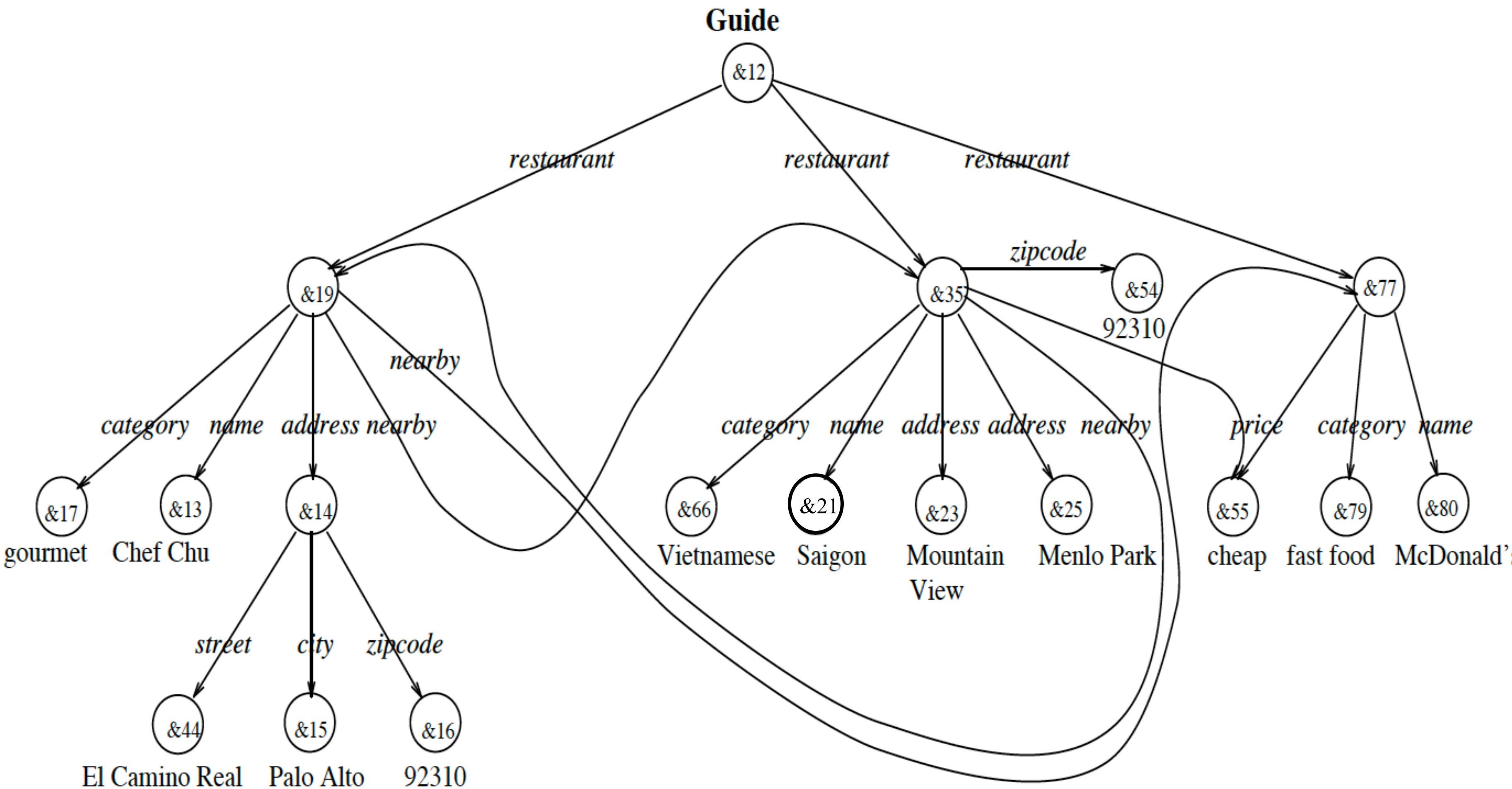
Every **edge** has a direction and a label

Atomic object = value (simple atomic type)

No (a priori) schema

Semistructured data: the data has internal structure (as opposed to a BLOB) but the structure is not regular, some parts may be more structured than others

A restaurant OEM database



Restaurant database, serialized

```
Guide &12
    restaurant &19
        category &17 "gourmet"
        name &13 "Chef Chu"
        address &14
            street &44 "El Camino Real"
            city &15 "Palo Alto"
            zipcode &16 92310
        nearby_eating_place &35
        nearby_eating_place &77
```

```
restaurant &35
    category &66 "Vietnamese"
    name &21 "Saigon"
    address &23 "Mountain View"
    address &25 "Menlo Park"
    nearby_eating_place &19
    zipcode &54 "92310"
    price &55 "cheap"
restaurant &77
    category &79 "fast food"
    name &80 "McDonald's"
    price &55
```

Querying OEM data with LOREL [AQH+97]

Semistructured database principle: no query should fail; query evaluation should adapt gracefully

```
select Guide.restaurant.address
where Guide.restaurant.address.zipcode=92310
```

Guide is a *persistence root* (name starts with a capital)

Empty results if expected labels are not found

Tries to convert zipcode to an integer; also accepts strings

```
select  Guide.restaurant.name,
        Guide.restaurant.(address?).zipcode
where Guide.restaurant.% grep "cheap"
```

Address is optional; "cheap" can occur anywhere in the restaurant object

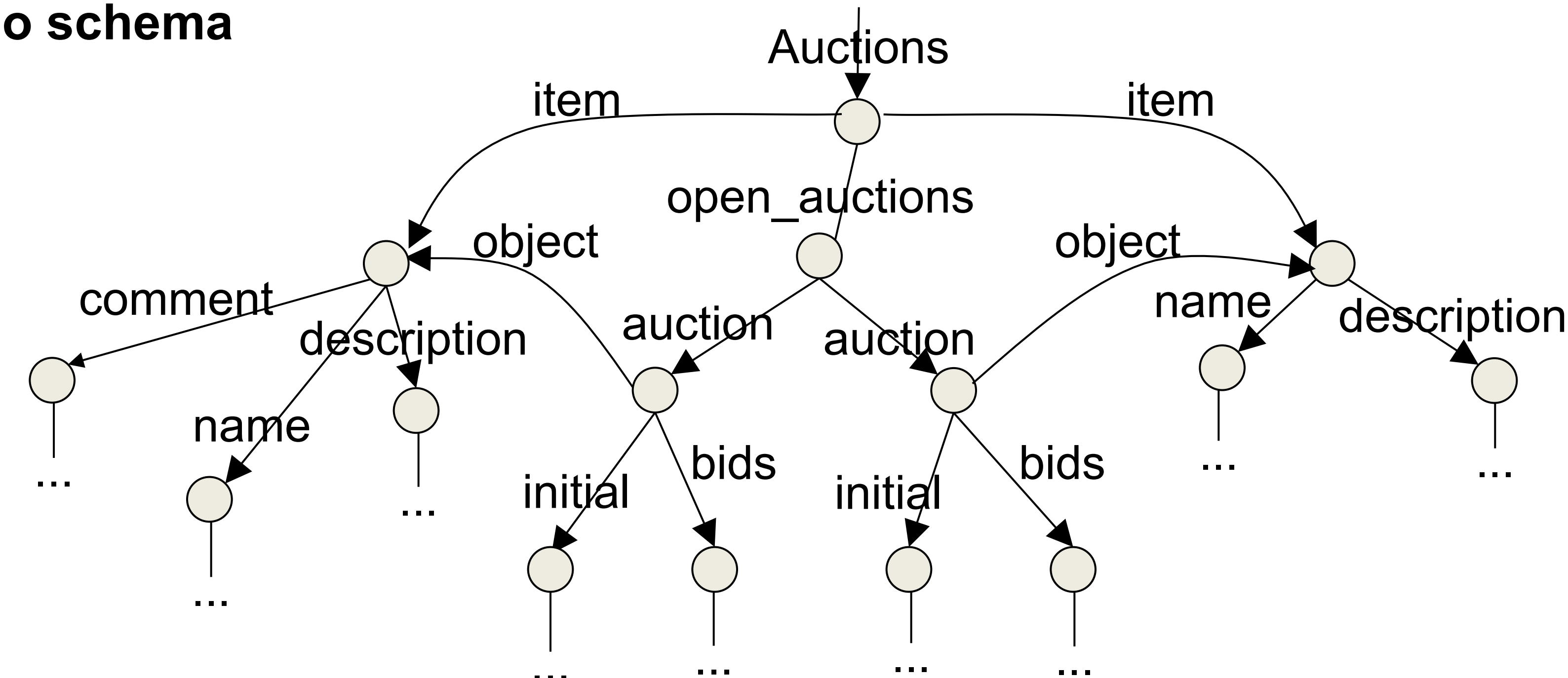
The first (graph) semistructured data model: OEM [PGW95]

Semistructured data: the data has internal structure (opposed to e.g. unstructured *text* or *blob – Binary Large OBject*) but the structure is not regular

Some items have comments/bids, others do not

One description may be just text, another one have complex structure

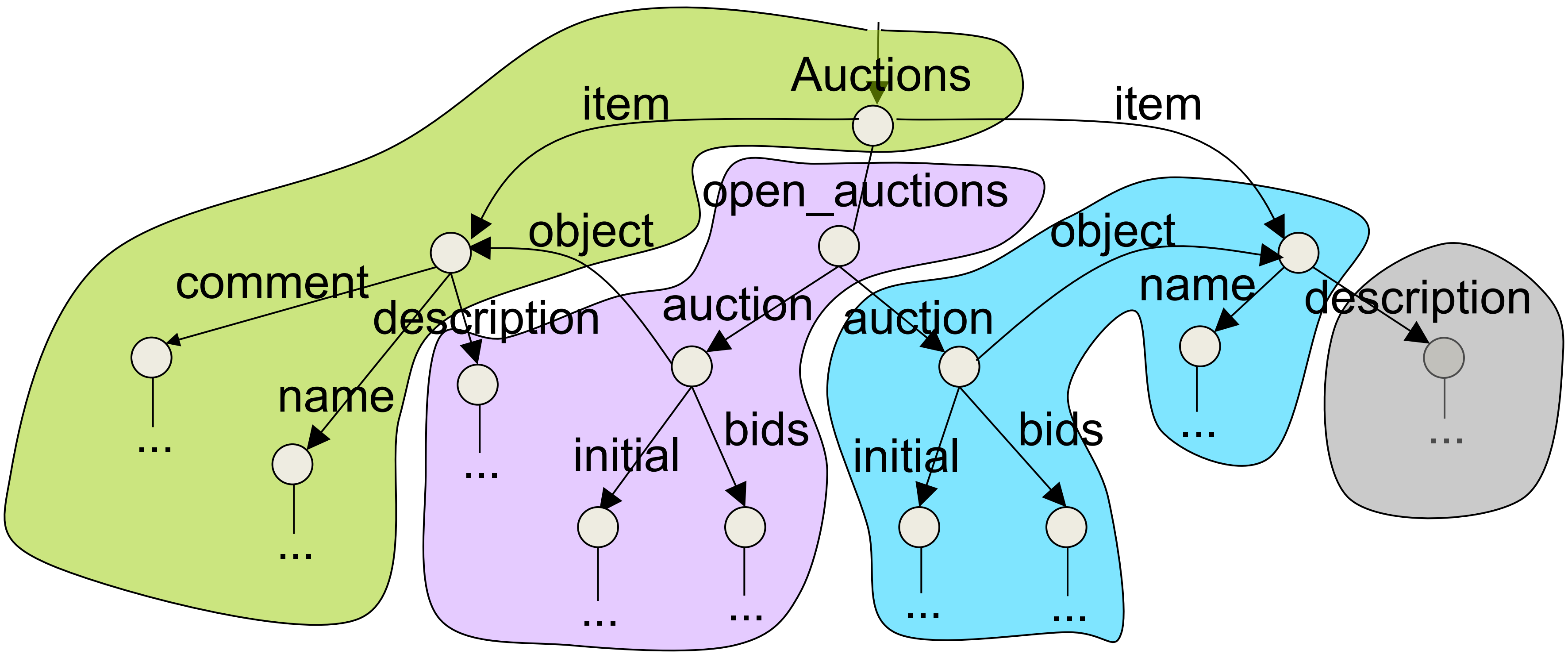
No schema



Storing OEM objects in LORE [MAG+97]

Objects clustered in pages in depth-first order, including simple value leaves

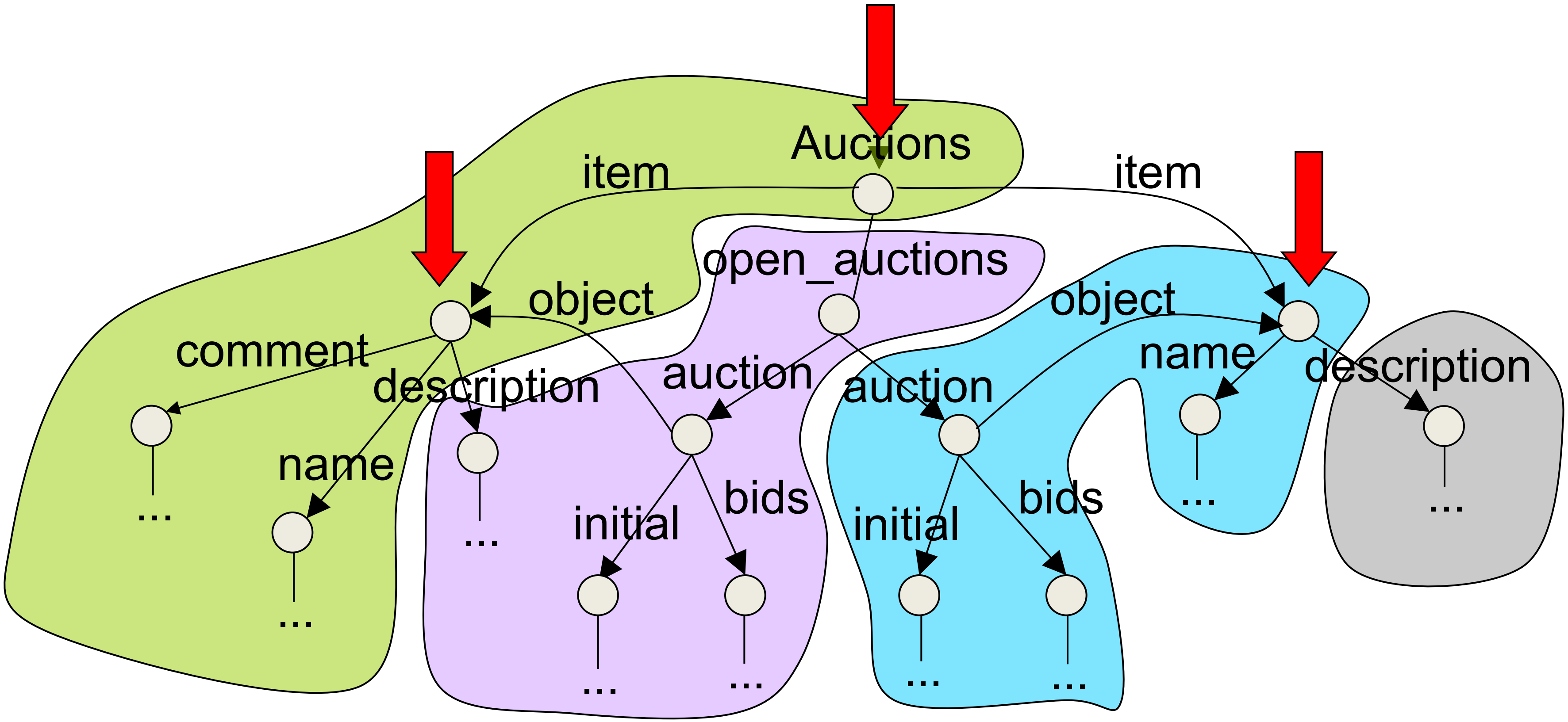
Basic physical operator: Scan(obj, path)



Navigation in a persistent graph

Navigation-based scan implementation (aka tuple-at-a-time, pointer-chasing)

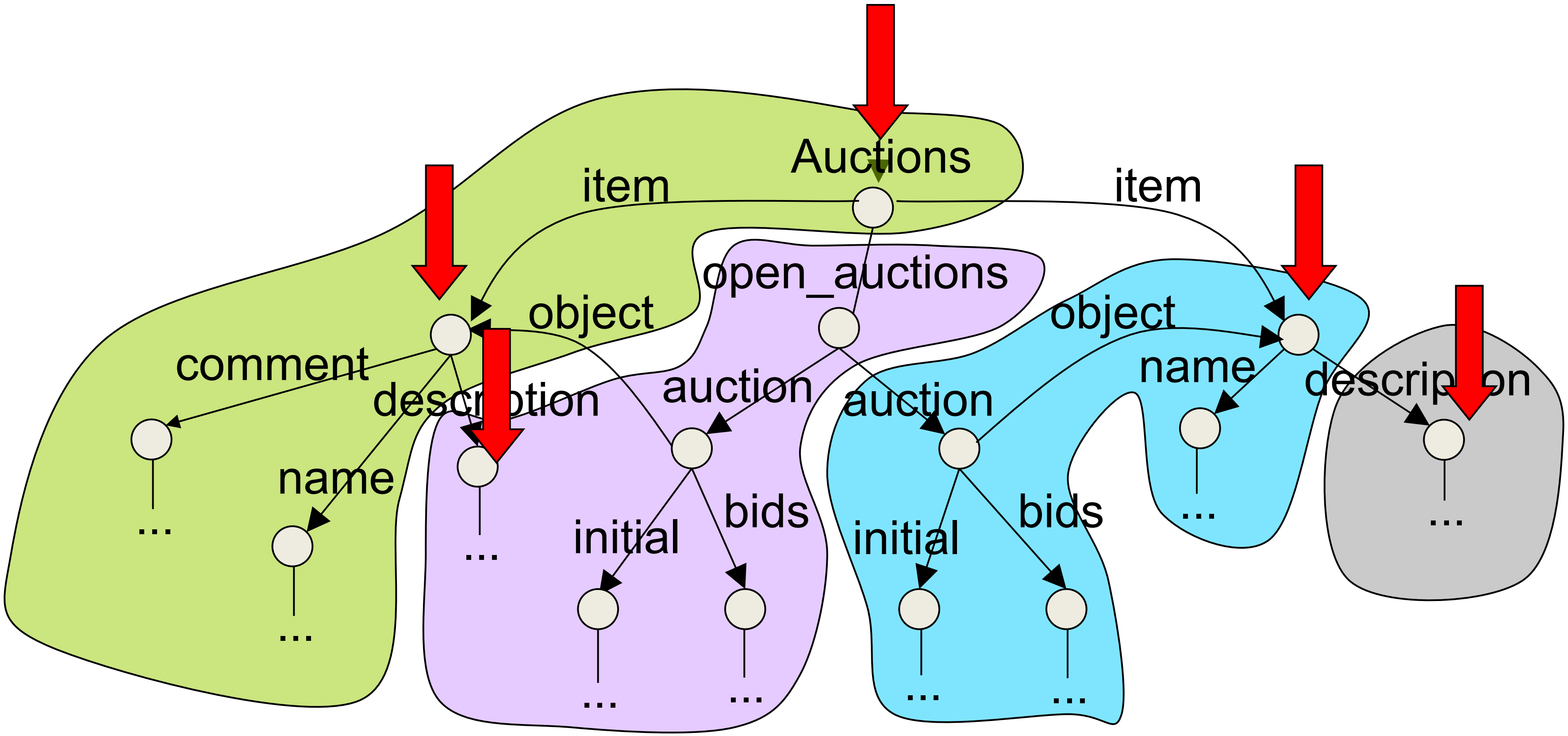
Scan(Auctions, "item"): 2 pages accessed



Navigation in a persistent graph

Scan(Auctions, "item.description"): 4 pages accessed

Scan(Auctions, "open_auctions.auction.object"): 4 pages accessed



Indexing objects in a graph [MW97,MWA+98, MW99]

VIndex(**l**, o, **pred**): all objects o with an incoming **l**-edge, satisfying **pred**

LIndex(**o**, **l**, p): all parents of **o** via an **l**-edge

– "Reverse pointers"

BIndex(x, **l**, y): all edges labeled **l**

select X
from Auction.open_auctions.auction X
where X.initial < 10

Return(n2)

Name(n4,"Auctions")

LIndex(n3, "open_auctions", n4)

LIndex(n2, "auction",n3)

LIndex(n1, "initial", n2)

tuple at a time

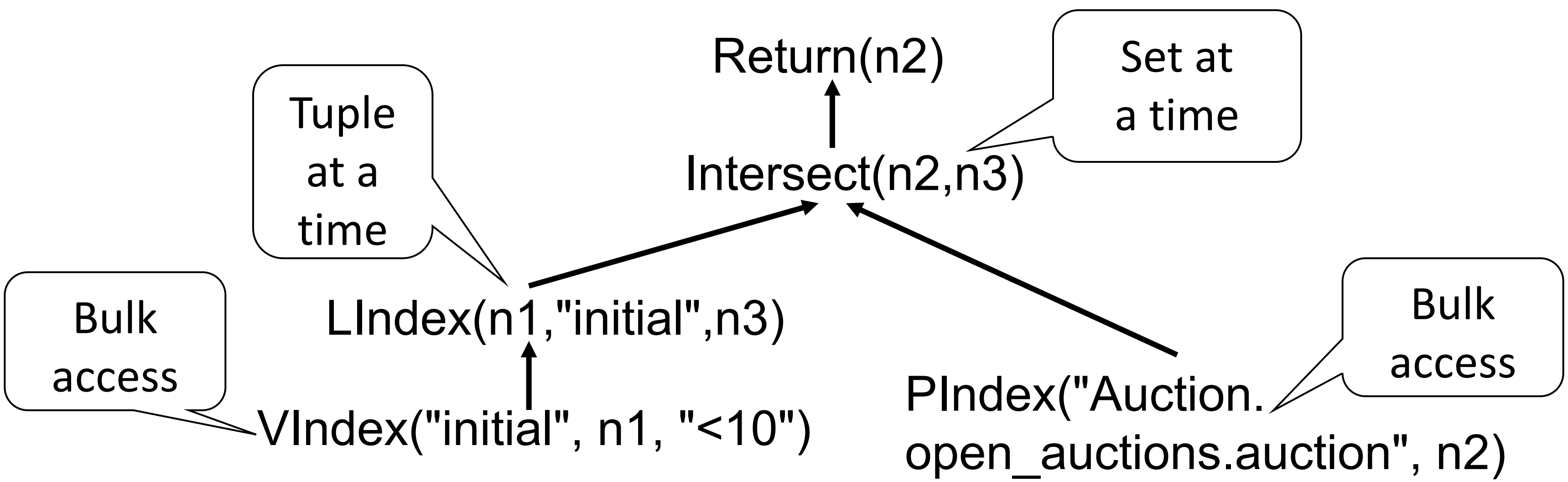
VIndex("initial", n1, "<10")

bulk access

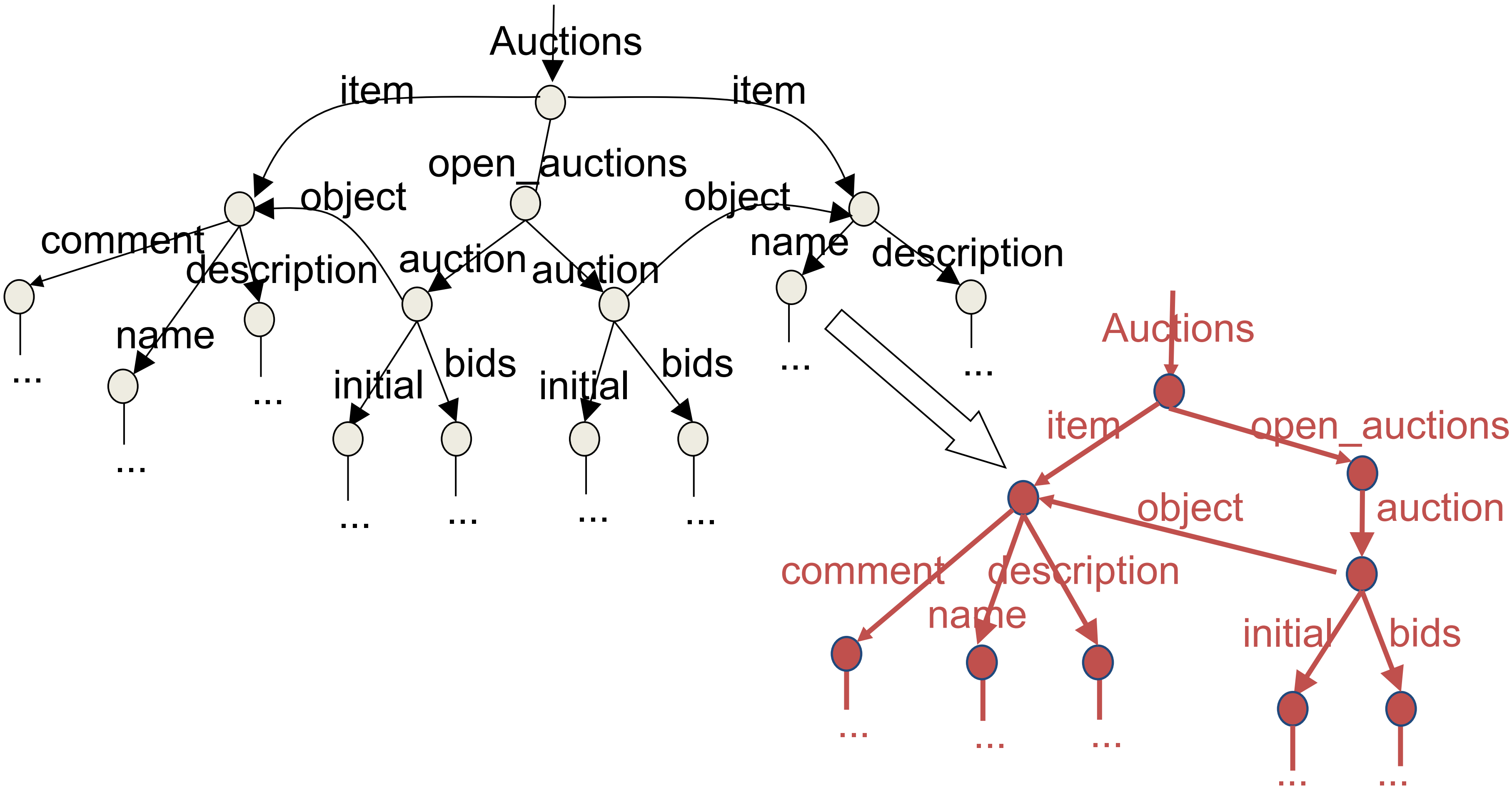
Indexing objects in a graph [MW97]

PIndex(**p**, o): all objects o reachable by the path p

```
select X
from Auction.open_auctions.auction.initial  X
where X.initial < 10
```



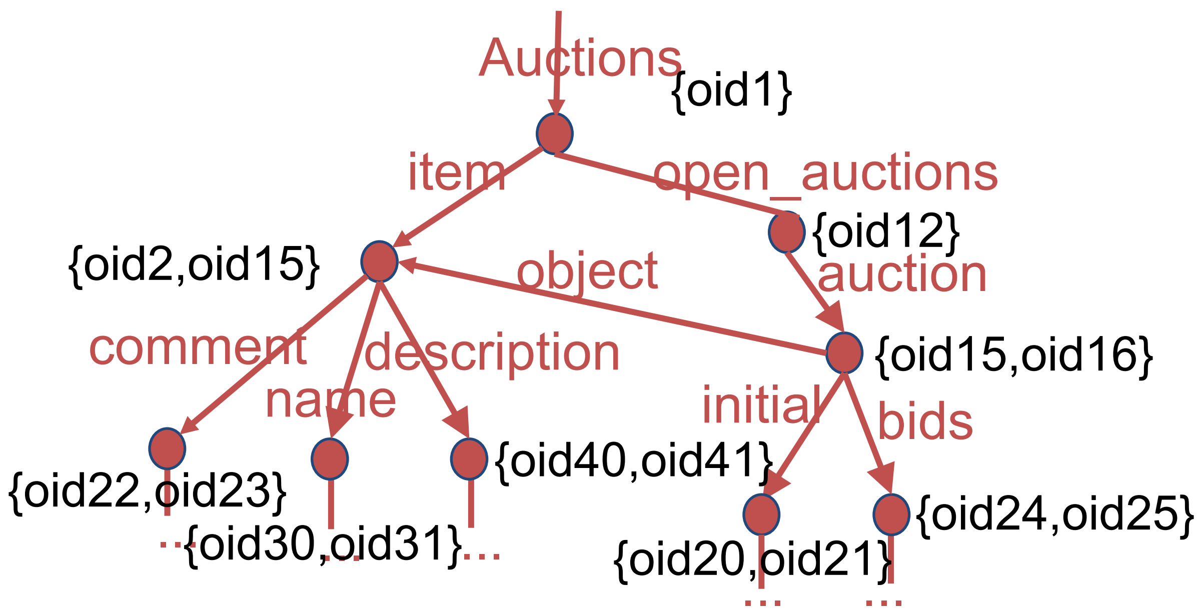
The idea behind path indexes: DataGuides [GW97]



The idea behind path indexes: DataGuides [GW97]

Graph-shaped summaries of graph data

- "A-posteriori schema"
- Groups all nodes *reachable by the same paths*



More on graph indexing

Graph indexing:

1. Partition nodes into equivalence classes
2. Store the extent of each equivalence class, use it as "pre-cooked" answer to some queries

Equivalence notions:

1. Reachable by some common paths: *DataGuide* [MW97]
2. Reachable by exactly the same paths: *1-index* [MS99] or, equivalently, indistinguishable by any forward path expression
3. Indistinguishable by any (forward and backward) path expression: *F&B Index* [KBN+02]
4. Indistinguishable by the (forward and backward) path expressions in the set Q: *covering index* [KBN+02]
5. Indistinguishable by any path expression of length < k: *A(k) index* [KSB+02]

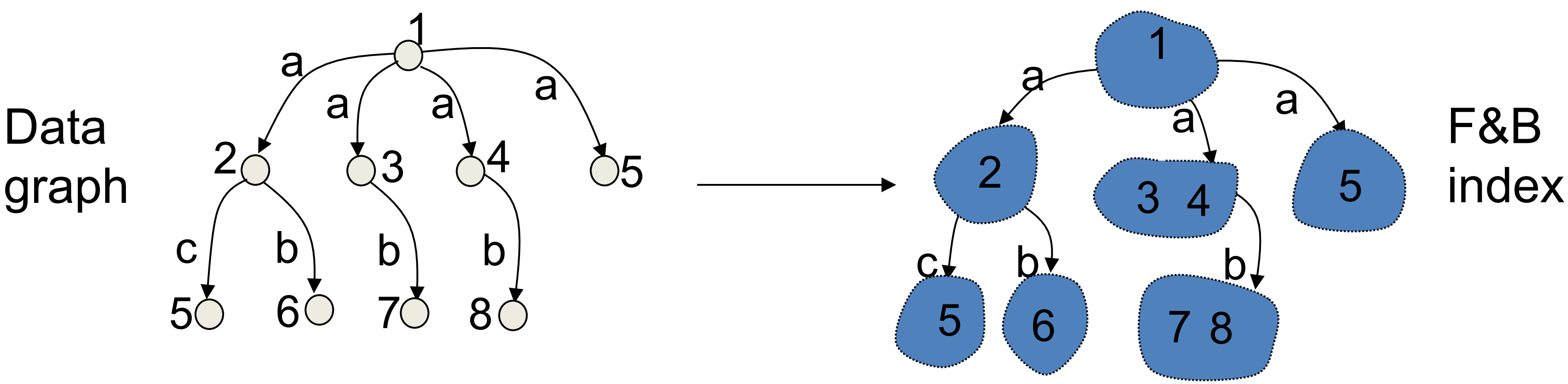
F&B index

Group together nodes reachable by exactly the same paths

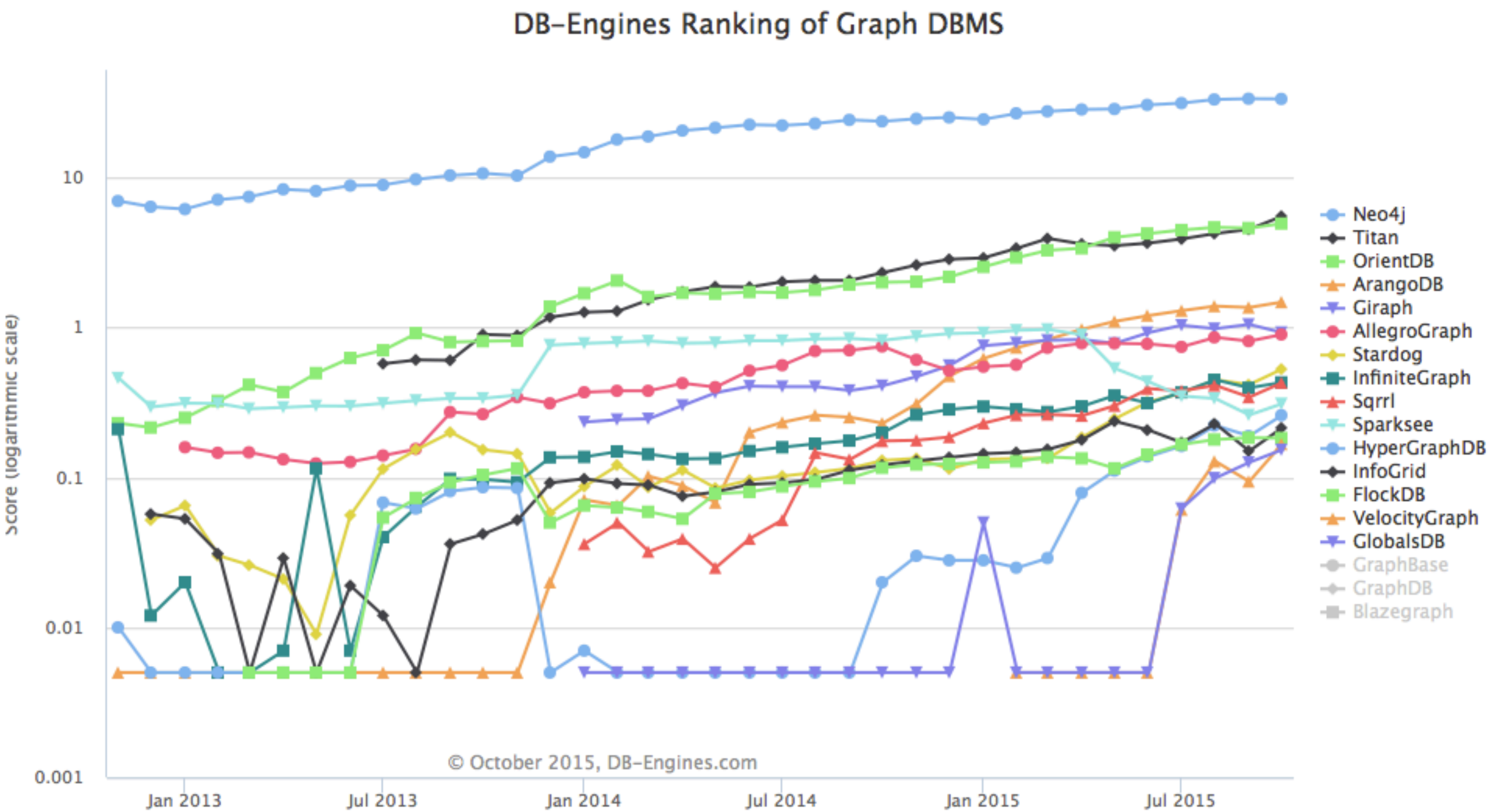
Path language:

- Navigate along one edge in both directions
- Navigate along any number of edges, in both directions

n1 ~ n2: *for any path expression p, either n1 and n2 are in the answer of p, or neither are in the answer of p.*



Current graph stores



Neo4J basics

Data model: labeled, directed graphs

Data manipulation language (CRUD): Cypher, used to describe data and patterns to be matched

Node descriptions in Cypher:

```
()                // empty anonymous node
(matrix)          // node whose identifier is matrix.
(:Movie)          // node of type Movie
(matrix:Movie)    // node whose ID is matrix and type Movie
(matrix:Movie {title: "The Matrix"})
                  // node with an attribute
(matrix:Movie {title: "The Matrix", released: 1997})
                  // node with two attributes
```

Identifiers can be used to refer to this node in another place in the same statement

Identifiers are not stored in the database (they are related to "variables")

Strings vs. integers

Neo4J basics

Relationship descriptions in Cypher

-- (undirected) vs. --> or <-- (directed)

Sample relationship descriptions:

-->

-[role]-> // relationship ID

-[:ACTED_IN]-> // relationship type

-[role:ACTED_IN]->

-[role:ACTED_IN {roles: ["Neo"]}]-> // relationship with attributes

Data manipulation with Cypher

Patterns combine node and relationship descriptors:

```
(keanu:Person:Actor {name: "Keanu Reeves"} )
-[role:ACTED_IN {roles: ["Neo"] } ]-> (matrix:Movie {title: "The
Matrix"} )
```

Data **creation**:

```
CREATE (a:Person { name:"Tom Hanks", born:1956 })
        -[r:ACTED_IN { roles: ["Forrest"]}]->
        (m:Movie { title:"Forrest Gump",released:1994 })
CREATE (d:Person { name:"Robert Zemeckis", born:1951 })
        -[:DIRECTED]->(m)
```

Data manipulation with Cypher

Querying data: **MATCH** pattern **RETURN** matched variables

```
MATCH (p:Person { name:"Tom Hanks" })
        -[r:ACTED_IN]->(m:Movie)
RETURN m.title, r.roles
```

Successive match-create-return steps can be used to update the data:

```
MATCH (p:Person { name:"Tom Hanks" })
CREATE (m:Movie { title:"Cloud Atlas",released:2012 })
CREATE (p)-[r:ACTED_IN { roles: ['Zachry']}]->(m)
RETURN p,r,m
```

Data manipulation with Cypher

Inserting data only if it didn't exist:

```
MERGE (m:Movie { title:"Cloud Atlas" })
            // create or check the existence of movie node m
ON CREATE SET m.released = 2012
            // if we had to create it, set the release year
RETURN m
```

Insert relationship only if it did not exist:

```
MATCH (m:Movie { title:"Cloud Atlas" })
MATCH (p:Person { name:"Tom Hanks" })
MERGE (p)-[r:ACTED_IN]->(m)
ON CREATE SET r.roles =['Zachry']
RETURN p,r,m
```

Returning results with Cypher

MATCH (a { name: "A" })-[r]->(b)

RETURN *

a	b	r
Node[0]{name:"A",happy:"Yes!",age:55}	Node[1]{name:"B"}	:BLOCKS[1]{}
Node[0]{name:"A",happy:"Yes!",age:55}	Node[1]{name:"B"}	:KNOWS[0]{}

MATCH (n)

RETURN n.age // returns null if no age

MATCH (a { name: "A" })

RETURN a.age > 30, "I'm a literal",(a)-->()

Edge creation (ability to return new graphs)

Other Cypher operations

- Booleans:
  MATCH (n)
  WHERE n.name = 'Peter' XOR (n.age < 30 AND n.name = "Tobias") OR NOT (n.name = "Tobias" OR n.name="Peter")
  RETURN n
- Optional matching
- Returned data can be: ordered, truncated, aggregated
- Unwind: unfolds a collection into a set
  UNWIND[1,2,3] AS x **RETURN** x // three results
- Indexes: **CREATE INDEX ON** :Person(name)
- **EXPLAIN** to get the query plan
- **PROFILE** to measure the effort

Richer path specification in SPARQL

- RDF: W3C standard for semantic Web data (graphs)
  - Nodes are labeled with URIs or constants
  - Edges are labeled with URIs
- SPARQL: query language for RDF data

- SPARQL 1.1 provides rich property path descriptions (think regular expressions: http://www.w3.org/TR/sparql11-query/#propertypaths)

```
{ :book1 dc:title|rdfs:label ?displayString }
{ ?x foaf:mbox <mailto:alice@example> .
  ?x foaf:knows/foaf:name ?name . }
{ ?x foaf:knows/^foaf:knows ?y . FILTER(?x != ?y) }
{ ?ancestor (ex:motherOf|ex:fatherOf)+ ?me}
```

Graph stores: summary

- Graph databases repeatedly "attempted" but not fully "solved" yet
- Very convenient data model, natural representation
- Typically no strict schema
- No standard query language
- Semantic graphs are a particular case (RDF and SPARQL *are* standards)
- Most powerful tools around: *distributed* graph stores (Pregel, Spark GraphX)
  – Extra dimension: graph partitioning
  – Less effort on query language; in progress

References

[AQH+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener. "The Lorel Query Language for Semistructured Data", International Journal on Digital Libraries, 1997

[GW97] R.Goldman and J.Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases", VLDB 1997

[KBN+02] R.Kaushik, P.Bohannon, J.Naughton and H.Korth. "Covering Indexes for Branching Path Queries", SIGMOD 2002

[MW97] J.McHugh and J.Widom. "Query Optimization for Semistructured Data", tech. report, 1997

[MWA+98] J.McHugh, J.Widom, S.Abiteboul, Q.Luo and A.Rajaraman. "Indexing Semistructured Data", tech. report, 1998

[MW99] J.McHugh and J.Widom. "Query Optimization for XML", VLDB 1999

[MS99] T.Milo and D.Suciu. "Index Structures for Path Expressions", ICDT 1999

[PGW95] Y.Papakonstantinou, H.Garcia-Molina and J.Widom. "Object Exchange Across Heterogeneous Information Sources", ICDE 1995