A Structural Approach to Indexing Triples

François Picalausa[3], Yongming Luo[2], George Fletcher[2], **Jan Hidders**[1], and Stijn Vansummeren[3]

(1) TUD (Delft, NL)
(2) TU/e (Eindhoven, NL)
(3) ULB (Brussels, BE)

Introduction

- The challenge:
  - to speed up querying over huge RDF datasets
- Usually assumed to be large datasets with few updates, so we can relatively freely introduce extra indexes
  - Hexastore: [VLDB 2008, Weiss, Karras & Bernstein]
    - indexes on ***spo***, ***sop***, ***pso***, ***pos***, ***ops***, ***osp***
  - RDF3X [VLDB 2008, Neumann & Weikum]
    - also indexes on: ***s***, ***p***, ***o***, ***sp***, ***so***, ***ps***, ***po***, ***os***, ***op***
- Up to now fairly classical indexing techniques
  - **Recent Survey**: Storing and Indexing Massive RDF Datasets. Yongming Luo, Francois Picalausa, George H. L. Fletcher, Jan Hidders and Stijn Vansummeren. In: De Virgilio, R., et al. (eds.) Semantic Search over the Web, Data-Centric Systems and Applications, pp. 31–60. Springer, Heidelberg (2012).
- We focus on **structural indexes**,
  - a holistic type of indexing known from XML databases to speed up path expression evaluation

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1        :Type        :socialRelation .
    ?e1          ?rel1        ?e2 .
    ?rel2        :Type        :socialRelation .
    ?e2          ?rel2        ?e3 .
}
```

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

Sue	Larry
Joe	Sarah
Sue	Hiromi
John	Sarah

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1        :Type        :socialRelation .
    ?e1          ?rel1        ?e2 .
    ?rel2        :Type        :socialRelation .
    ?e2          ?rel2        ?e3 .
}
```

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1        :Type        :socialRelation .
    ?e1          ?rel1        ?e2 .
    ?rel2        :Type        :socialRelation .
    ?e2          ?rel2        ?e3 .
}
```

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1        :Type        :socialRelation .
    ?e1          ?rel1        ?e2 .
    ?rel2        :Type        :socialRelation .
    ?e2          ?rel2        ?e3 .
}
```

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1    :Type    :socialRelation .
    ?e1      ?rel1    ?e2 .
    ?rel2    :Type    :socialRelation .
    ?e2      ?rel2    ?e3 .
}
```

SPARQL Query Processing

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people who are indirectly related.

Sue	Larry
Joe	Sarah
Sue	Hiromi
John	Sarah

Adding join information

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

We mark all triples (s_1, p_1, o) such that their **object** *o* occurs as the **subject** of some other triple (o, p_2, o_2)

Using join information

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Find the people indirectly related.

```
SELECT ?e1 ?e3 WHERE {
    ?rel1        :Type       :socialRelation .
    ?e1          ?rel1       ?e2 .
    ?rel2        :Type       :socialRelation .
    ?e2          ?rel2       ?e3 .
}
```

Motivation

- Traditional relational SPARQL query engines fetch triples corresponding to individual triple patterns **independently**
- Rich history of introducing join information into query engines
  - Join Indexes: Precompute a single join (e.g. R.a = S.b)
  - Object Oriented indexes: Precompute join of single path in class hierarchy
  - Structural Indexes (for XML and RDF): Group **nodes** according to join similarity, fixed set of edge label
- By grouping together **triples** that can be joined in a "similar fashion", we can avoid fetching useless triples from disk.
  - How do we compute and store these groups?
  - How can we use them to process queries?

Table of Contents

- A Structural Index for Triples
- Building the Index
- Processing SPARQL queries

Table of Contents

- A Structural Index for Triples
- Building the Index
- Processing SPARQL queries

Real-World SPARQL Queries

Definition: The **equality type** of two triples $t = (t_1, t_2, t_3)$ and $u = (u_1, u_2, u_3)$ is the set $\mathrm{eqtp}(t, u) = \{ (i, j) \mid t_i = u_j, \text{ and } 1 \leq i, j \leq 3\}$ of positions where the triples share an equal value.

t:	Sue	Manages	Joe
u:	Joe	Manages	Larry

$$\mathrm{eqtp}(t, u) = \{(3,1), (2,2)\}$$

Structural Index

Definition: A structural index is an edge labeled graph (V,E), where

The nodes V are a partition of the RDF dataset

The edges E are labeled by the equality types between triples in nodes

Subject	Predicate	Object
Sue	Manages	Joe
Joe	Manages	Larry
Larry	Manages	Sarah
Sue	FriendOf	John
John	FriendOf	Hiromi
Hiromi	FriendOf	Sarah

Subject	Predicate	Object
Sue	Type	CEO
Manages	Type	socialRelation
FriendOf	Type	socialRelation
Likes	Type	socialRelation

Structural Index

Definition: A structural index is an edge labeled graph (V,E), where

The nodes V are a partition of the RDF dataset

The edges E are labeled by the equality types between triples in nodes

1		
Sue	FriendOf	John
Sue	Manages	Joe

2		
John	FriendOf	Hiromi
Joe	Manages	Larry

3		
Hiromi	FriendOf	Sarah
Larry	Manages	Sarah

4		
Sue	Type	CEO

5		
Manages	Type	socialRelation
FriendOf	Type	socialRelation

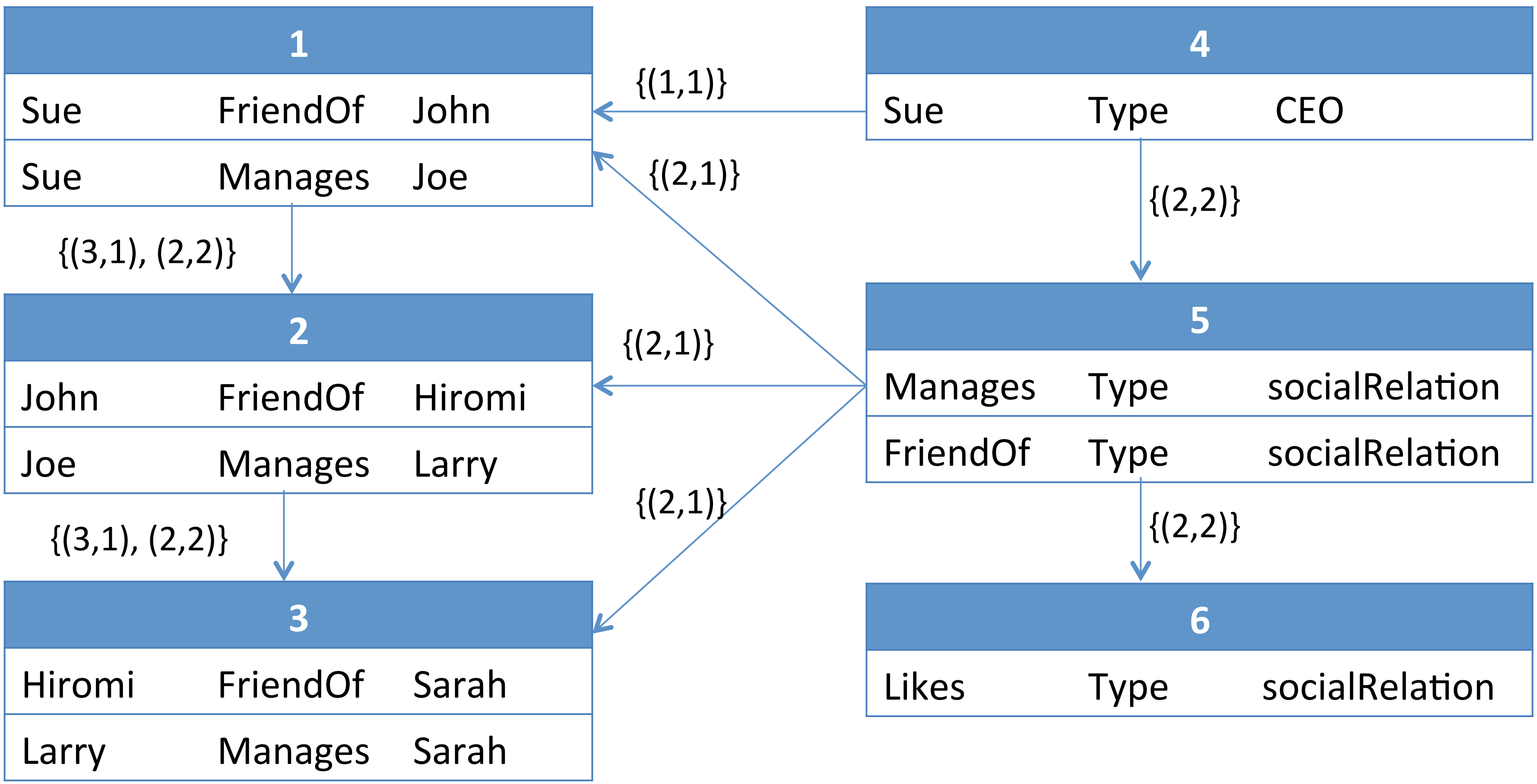
6		
Likes	Type	socialRelation

Structural Index

Definition: A structural index is an edge labeled graph (V,E), where

The nodes V are a partition of the RDF dataset

The edges E are labeled by the equality types between triples in nodes

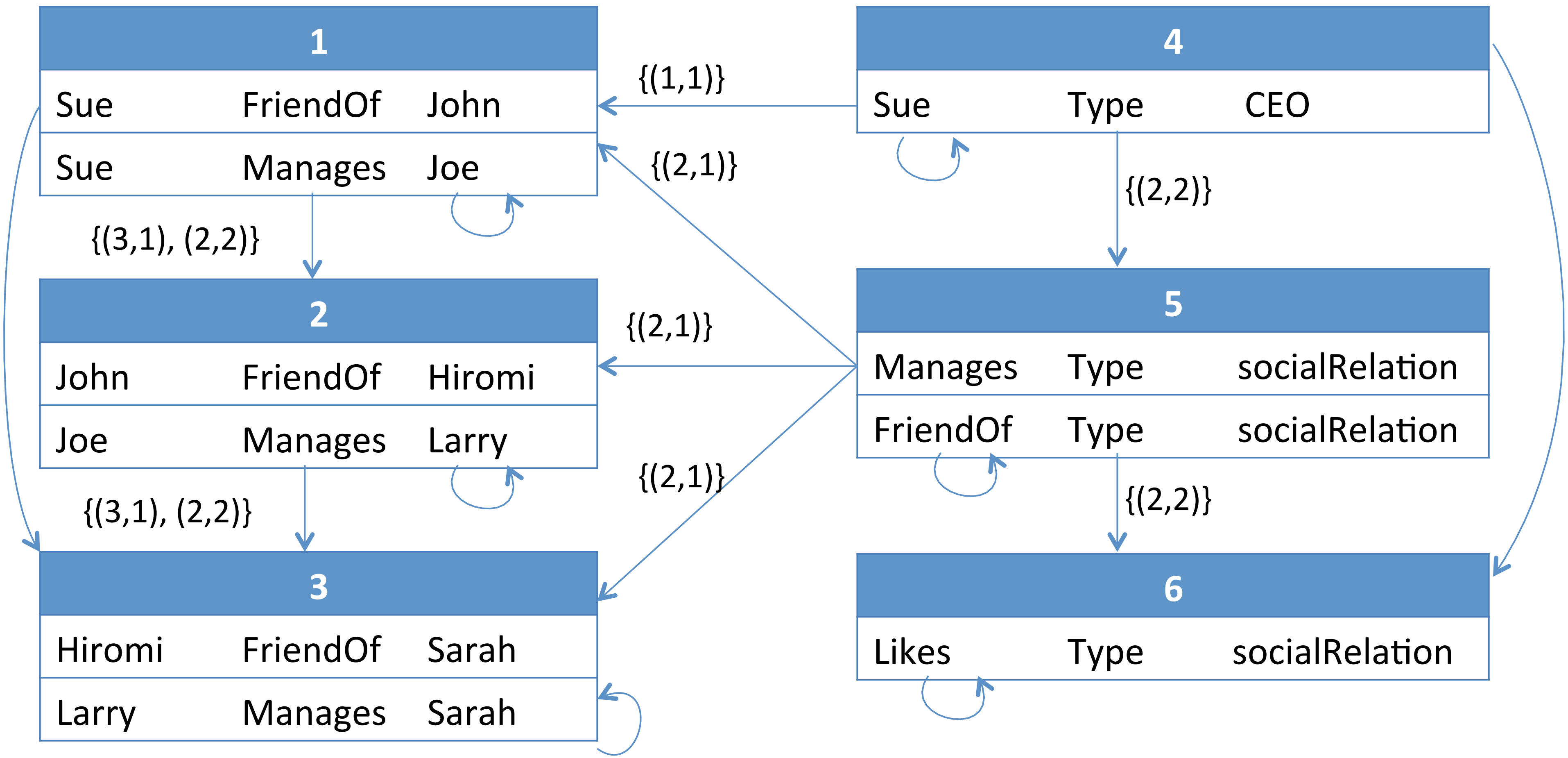


Structural Index

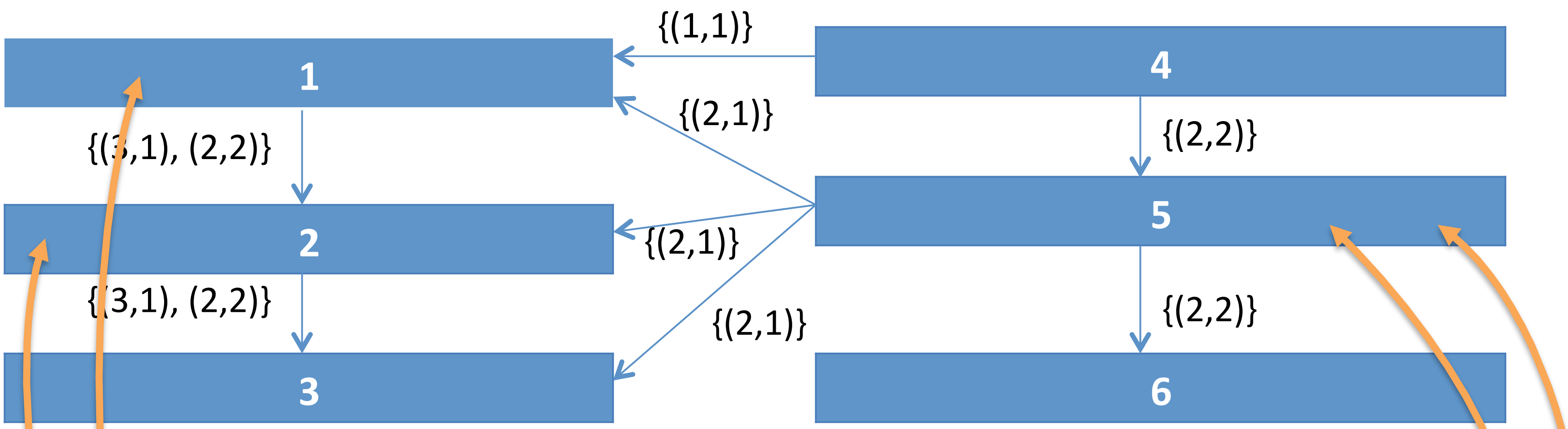
Definition: A structural index is an edge labeled graph (V,E), where

The nodes V are a partition of the RDF dataset

The edges E are labeled by the equality types between triples in nodes



Querying the Index




```
SELECT ?e1 ?e3 WHERE {
    ?rel1        Type        socialRelation .
    ?e1          ?rel1       ?e2 .
    ?rel2        Type        socialRelation .
    ?e2          ?rel2       ?e3 .
}
```

Table of Contents

Real-World SPARQL Queries

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Real-World SPARQL Queries

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

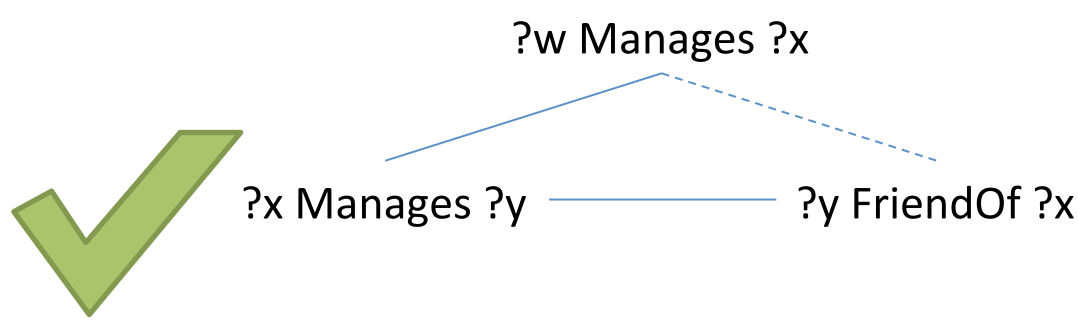
[Picalausa, Vansummeren – in SWIM2011]

A query Q is acyclic if it has a join forest.

A join forest for Q is a forest F whose set of nodes are the triple patterns of the query.
For any pair of triple patterns p and q in Q that have a variable in common:

1. p and q belong to the same connected component of F
2. All variables common to p and q occur in every triple pattern on the path in F from p to q

?w	Manages	?x .
?x	Manages	?y .
?y	FriendOf	?w .



Real-World SPARQL Queries

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

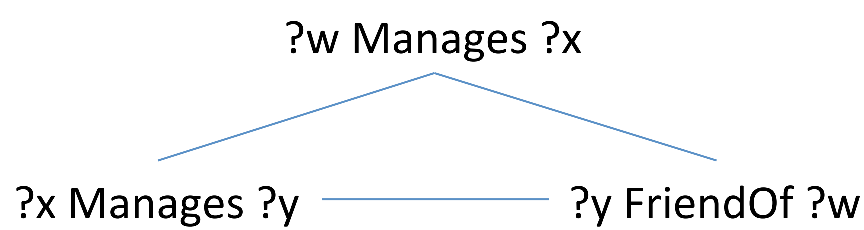
[Picalausa, Vansummeren – in SWIM2011]

A query Q is acyclic if it has a join forest.

A join forest for Q is a forest F whose set of nodes are the triple patterns of the query.
For any pair of triple patterns p and q in Q that have a variable in common:

1. p and q belong to the same connected component of F
2. All variables common to p and q occur in every triple pattern on the path in F from p to q

```
?w   Manages   ?x .
?x   Manages   ?y .
?y   FriendOf  ?w .
```



Structural Characterization

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Definition: A BGP query is **pure** if it contains only variables.

Structural Characterization

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Theorem: Given two triples t, and u, the following are equivalent:

- t is in Q(D) if and only if u is in Q(D), for every *pure acyclic BGP* Q
- t is *similar* to u

[Fletcher, Hidders, Vansummeren, Luo, Picalausa, De Bra — DBPL 2011]

Consider a RDF dataset D. A triple t of D **simulates** a triple u of D guardedly if for every triple t' of D, there exists some triple u' of D such that eqtp(t,t') ⊆ eqtp(u,u') and t' simulates u'.

Structural Characterization

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Theorem: Given two triples t, and u, the following are equivalent:
- t is in Q(D) if and only if u is in Q(D), for every *pure acyclic BGP* Q
- t is *similar* to u

[Fletcher, Hidders, Vansummeren, Luo, Picalausa, De Bra — DBPL 2011]

Consider a RDF dataset D. A triple t of D **simulates** a triple u of D guardedly if for every triple t' of D, there exists some triple u' of D such that eqtp(t,t') ⊆ eqtp(u,u') and t' simulates u'.

(Sue, FriendOf, John) → (John, FriendOf, Hiromi) → (FriendOf, Type, relation)

{(3,1), (2,2)} {(2,1)}

(Joe, Manages, Larry) → (Lary, Manages, Sarah) → (Manages, Type, relation)

Structural Characterization

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Theorem: Given two triples t, and u, the following are equivalent:

- t is in Q(D) if and only if u is in Q(D), for every *pure acyclic BGP* Q
- t is *similar* to u

[Fletcher, Hidders, Vansummeren, Luo, Picalausa, De Bra — DBPL 2011]

Consider a RDF dataset D. A triple t of D **simulates** a triple u of D guardedly if for every triple t' of D, there exists some triple u' of D such that eqtp(t,t') ⊆ eqtp(u,u') and t' simulates u'.

(Sue, FriendOf, John) → (John, FriendOf, Hiromi) → (FriendOf, Type, relation)

simulates {(3,1), (2,2)} **simulates** {(2,1)} **simulates**

(Joe, Manages, Larry) → (Lary, Manages, Sarah) → (Manages, Type, relation)

Structural Characterization

Fact: Most queries posed in practice only use basic graph pattern (BGP).
99% of real-world BGP queries are found to be **acyclic**.

[Picalausa, Vansummeren – in SWIM2011]

Theorem: Given two triples t, and u, the following are equivalent:

- t is in Q(D) if and only if u is in Q(D), for every *pure acyclic BGP* Q
- t is *similar* to u

[Fletcher, Hidders, Vansummeren, Luo, Picalausa, De Bra — DBPL 2011]

A triple t of D is **similar** to a triple u of D, denoted t ≃ u, if t simulates u and u simulates t.

(Sue, FriendOf, John) → (John, FriendOf, Hiromi) → (FriendOf, Type, relation)

similar {(3,1), (2,2)} similar {(2,1)} similar

(Joe, Manages, Larry) → (Lary, Manages, Sarah) → (Manages, Type, relation)

Structural Index

1		
Sue	FriendOf	John
Sue	Manages	Joe

2		
John	FriendOf	Hiromi
Joe	Manages	Larry

3		
Hiromi	FriendOf	Sarah
Larry	Manages	Sarah

4		
Sue	Type	CEO

5		
Manages	Type	socialRelation
FriendOf	Type	socialRelation

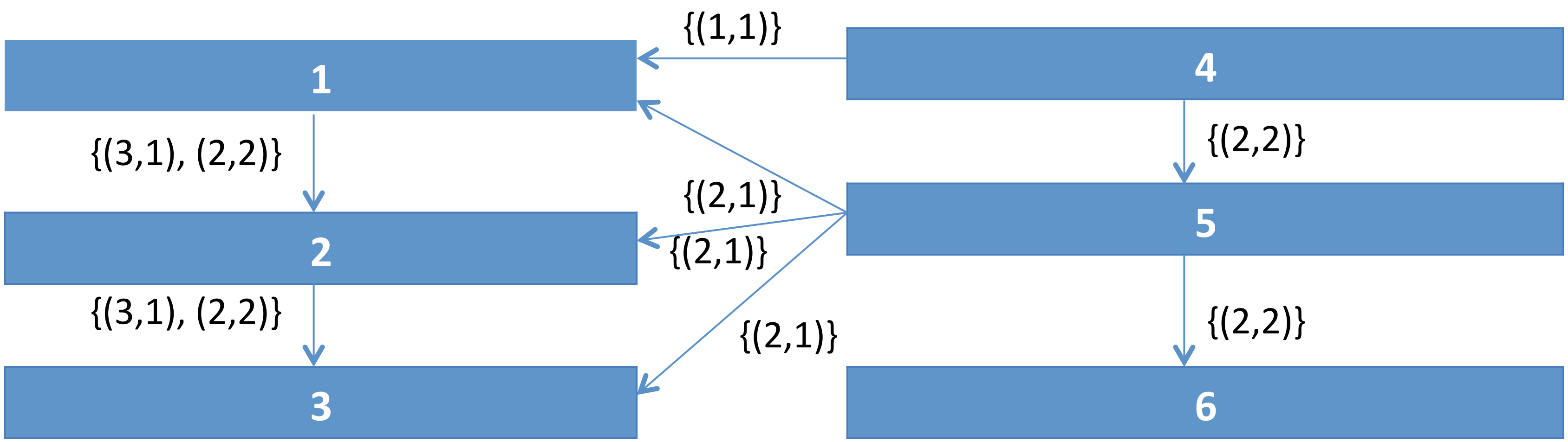
6		
Likes	Type	socialRelation

Table of Contents

- A Structural Index for Triples
- Building the Index
- Processing SPARQL queries

Structural Index Storage

Ideally, the structural index is sufficiently small to be kept in main memory

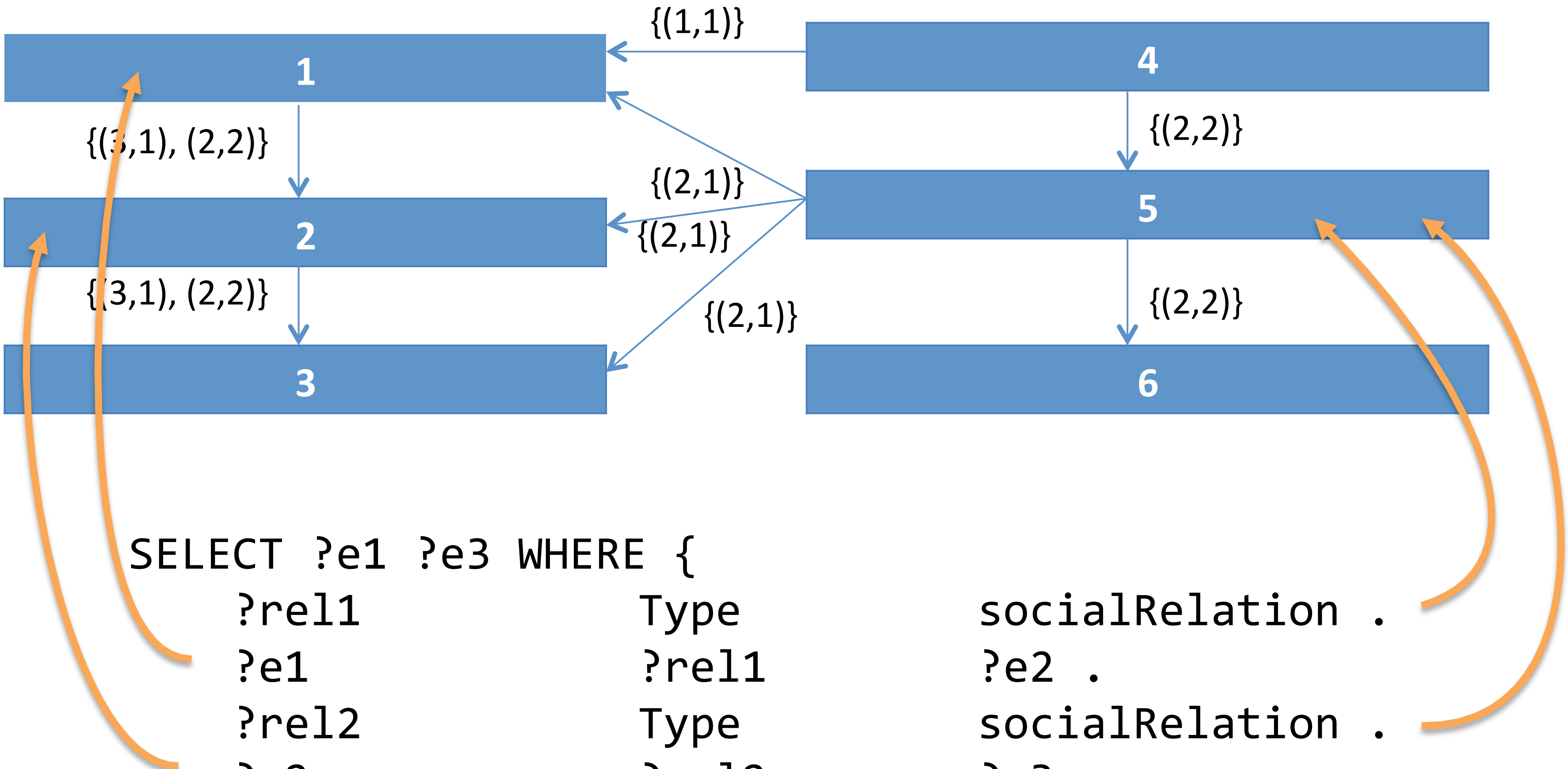


Each triple (subject, predicate, object) are stored as a quad (subject, predicate, object, partition)

Subject	Predicate	Object	Partition
Sue	Manages	Joe	1
Joe	Manages	Larry	2
Larry	Manages	Sarah	3
Sue	FriendOf	John	1

Querying the Index

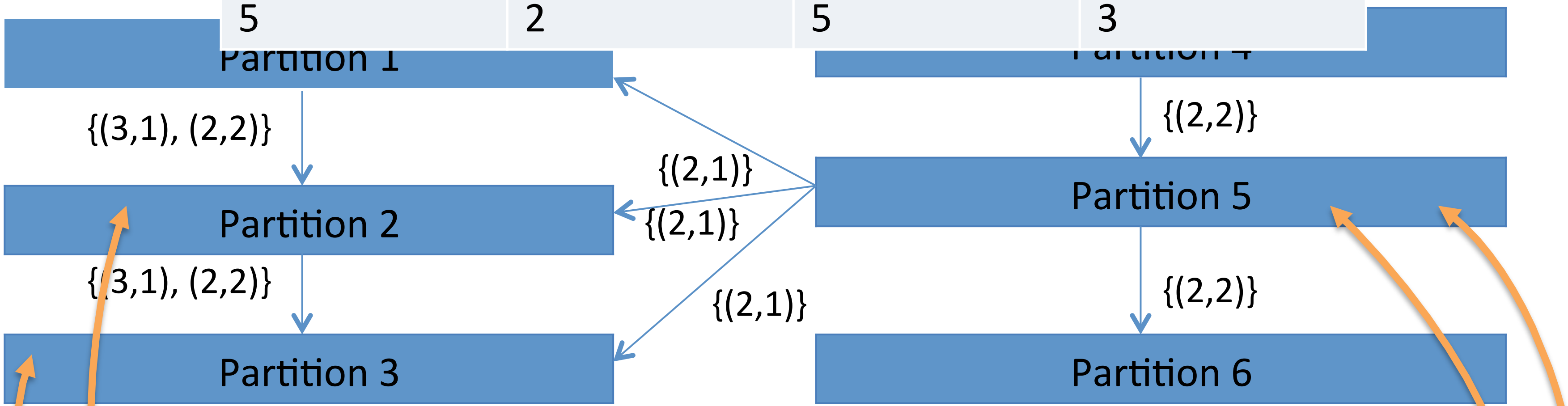
Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2




```
SELECT ?e1 ?e3 WHERE {
    ?rel1           Type          socialRelation .
    ?e1             ?rel1         ?e2 .
    ?rel2           Type          socialRelation .
    ?e2             ?rel2         ?e3 .
}
```

Querying the Index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3




```
SELECT ?e1 ?e3 WHERE {
    ?rel1        Type        socialRelation .
    ?e1          ?rel1       ?e2 .
    ?rel2        Type        socialRelation .
    ?e2          ?rel2       ?e3 .
}
```

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): $((\text{Partition5} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Partition5} \bowtie (\text{Partition2} \cup \text{Partition3})))$

(M3): $((\text{Pattern1} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Pattern3} \bowtie (\text{Partition2} \cup \text{Partition3})))$

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): **((Partition1 ⋈ Partition5) ⋈ (Partition2 ⋈ Partition5))**
∪ ((Partition2 ⋈ Partition5) ⋈ (Partition3 ⋈ Partition5))

(M2): ((Partition5 ⋈ (Partition1 ∪ Partition2)) ⋈
(Partition5 ⋈ (Partition2 ∪ Partition3)))

(M3): ((Pattern1 ⋈ (Partition1 ∪ Partition2)) ⋈
(Pattern3 ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): ((Partition1 ⋈ Partition5) ⋈ (Partition2 ⋈ Partition5))
∪ **((Partition2 ⋈ Partition5) ⋈ (Partition3 ⋈ Partition5))**

(M2): ((Partition5 ⋈ (Partition1 ∪ Partition2)) ⋈
(Partition5 ⋈ (Partition2 ∪ Partition3)))

(M3): ((Pattern1 ⋈ (Partition1 ∪ Partition2)) ⋈
(Pattern3 ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup\ ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): ((**Partition5** ⋈ (Partition1 ∪ Partition2)) ⋈
(Partition5 ⋈ (Partition2 ∪ Partition3)))

(M3): ((Pattern1 ⋈ (Partition1 ∪ Partition2)) ⋈
(Pattern3 ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): ((Partition1 ⋈ Partition5) ⋈ (Partition2 ⋈ Partition5))
∪ ((Partition2 ⋈ Partition5) ⋈ (Partition3 ⋈ Partition5))

(M2): ((Partition5 ⋈ (**Partition1** ∪ **Partition2**)) ⋈
(Partition5 ⋈ (Partition2 ∪ Partition3)))

(M3): ((Pattern1 ⋈ (Partition1 ∪ Partition2)) ⋈
(Pattern3 ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup\ ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): $((\text{Partition5} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\textbf{Partition5} \bowtie (\text{Partition2} \cup \text{Partition3})))$

(M3): $((\text{Pattern1} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Pattern3} \bowtie (\text{Partition2} \cup \text{Partition3})))$

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((Partition1 \bowtie Partition5) \bowtie (Partition2 \bowtie Partition5))$
$\cup ((Partition2 \bowtie Partition5) \bowtie (Partition3 \bowtie Partition5))$

(M2): $((Partition5 \bowtie (Partition1 \cup Partition2)) \bowtie$
$(Partition5 \bowtie (\mathbf{Partition2} \cup \mathbf{Partition3})))$

(M3): $((Pattern1 \bowtie (Partition1 \cup Partition2)) \bowtie$
$(Pattern3 \bowtie (Partition2 \cup Partition3)))$

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
~~5~~	1	5	2
~~5~~	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup\ ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): $((\text{Partition5} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Partition5} \bowtie (\text{Partition2} \cup \text{Partition3})))$

(M3): $((\textbf{Pattern1} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Pattern3} \bowtie (\text{Partition2} \cup \text{Partition3})))$

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup\ ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): $((\text{Partition5} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Partition5} \bowtie (\text{Partition2} \cup \text{Partition3})))$

(M3): ((Pattern1 ⋈ (**Partition1** ∪ **Partition2**)) ⋈
(Pattern3 ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): ((Partition1 ⋈ Partition5) ⋈ (Partition2 ⋈ Partition5))
∪ ((Partition2 ⋈ Partition5) ⋈ (Partition3 ⋈ Partition5))

(M2): ((Partition5 ⋈ (Partition1 ∪ Partition2)) ⋈
(Partition5 ⋈ (Partition2 ∪ Partition3)))

(M3): ((Pattern1 ⋈ (Partition1 ∪ Partition2)) ⋈
(**Pattern3** ⋈ (Partition2 ∪ Partition3)))

Only use partitions when query optimizer deems useful

Query Processing Strategies

Input: The SPARQL query

All embeddings of the query into the structural index

Pattern 1	Pattern 2	Pattern 3	Pattern 4
5	1	5	2
5	2	5	3

Output: A physical query plan

(M1): $((\text{Partition1} \bowtie \text{Partition5}) \bowtie (\text{Partition2} \bowtie \text{Partition5}))$
$\cup ((\text{Partition2} \bowtie \text{Partition5}) \bowtie (\text{Partition3} \bowtie \text{Partition5}))$

(M2): $((\text{Partition5} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Partition5} \bowtie (\text{Partition2} \cup \text{Partition3})))$

(M3): $((\text{Pattern1} \bowtie (\text{Partition1} \cup \text{Partition2})) \bowtie$
$(\text{Pattern3} \bowtie (\mathbf{Partition2} \cup \mathbf{Partition3})))$

Only use partitions when query optimizer deems useful

Empirical Evaluation

How do the different processing strategies compare?
Can traditional query processors benefit from this additional index?

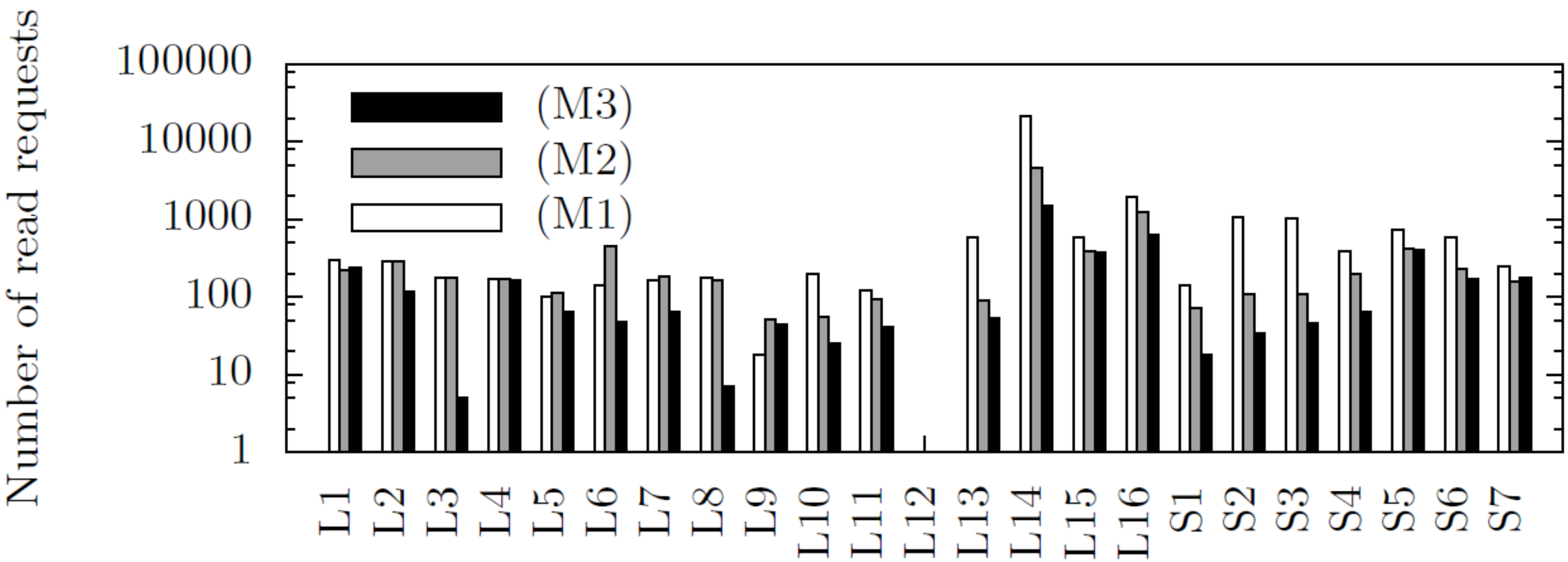
SAINT-DB: modification of RDF-3X with structural indexes

Datasets:

- **LUBM**: Synthetically generated dataset of 2 million triples
- **Southampton**: Real-world dataset of 4 million triples

All results given in number of **disk page reads**

Comparison of the different strategies



Comparison with RDF-3X

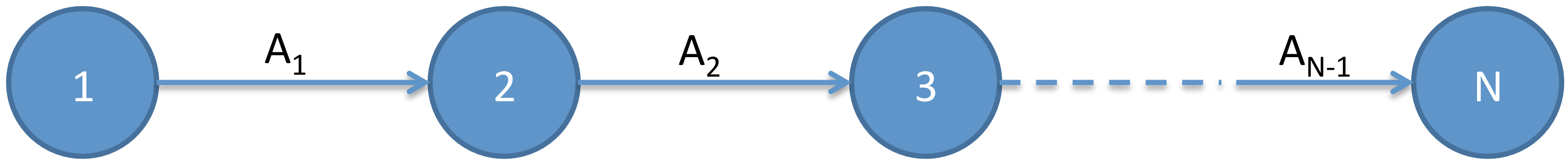
- C1: Single triple pattern
  (Sue, Manages ?y)
- C2: Highly selective triple patterns in the query
  (?x, Type, CEO) (?x, Manages, John)
- C3:Queries with multiple triple patterns, non selective

Processing strategy: M3

	C1			**C2**				**C3**				
	L2	*L3*	*L4*	*L9*	*S1*	*S2*	*S4*	*L1*	*L5*	*L6*	*L7*	*L8*
SAINT-DB	116	5	163	18	18	36	64	238	39	47	38	7
RDF-3X	89	5	123	12	16	35	53	194	132	39	268	7
Speed-up	0.77	1.00	0.75	0.67	0.89	0.97	0.83	0.82	3.38	0.83	7.05	1.00

	C3										
	L10	*L11*	*L12*	*L13*	*L14*	*L15*	*L16*	*S3*	*S5*	*S6*	*S7*
SAINT-DB	25	41	0	53	1519	352	288	48	410	173	175
RDF-3X	21	30	281	109	2668	2178	1224	33	424	316	236
Speed-up	0.84	0.73	∞	2.06	1.76	6.19	4.25	0.69	1.03	1.83	1.35

Comparison with RDF-3X – Best Case



1000 chains are generated for each N = 3..5

Queries are chains of triple patterns of the form

$(?x_1, ?y_1, ?x_2)\ (?x_2, ?y_2, ?x_3), \ldots (?x_n, ?y_n, ?x_{n+1})$ $n = 4..7$

	4	5	6	7
SAINT-DB	306	350	393	438
RDF-3X	3864	4799	5734	6669
Speed-up	12.63	13.71	14.59	15.23

Conclusion

- We introduced a triple-based structural index for RDF
- This index is tied to practical fragments of SPARQL
- Our initial empirical study shows that the approach is profitable

Future Work

- Alternate Structures for storing the index and dataset
- More optimized query processing strategies
- Efficient external memory and/or distributed computation of the indexes
- Extension to richer fragments of SPARQL

Thank you!