

# **RDF Data Management & SPARQL Query Processing**

**Federico Ulliana**  
UM, LIRMM, INRIA GraphIK

Slides collected from Martin Theobald, Katja Hose, Ralf Schenkel,  
and Stratos Idreos

# NOSQL Umbrella

(RDF and SPARQL are the only standardized languages)

- Key-value databases are systems are about as simple as databases get, being in essence variations on the theme of a persistent hash table. Current examples include MemcacheDB, Tokyo Cabinet, Redis and SimpleDB.
- Document databases are key-value stores that treat stored values as semi-structured data instead of as opaque blobs. Prominent examples at the moment include CouchDB, MongoDB and Riak.
- Wide-column databases tend to draw inspiration from Google's BigTable model. Open-source examples include Cassandra, HBase and Hypertable.
- Graph databases include generic solutions like Neo4j, InfoGrid and HyperGraphDB as well as all the numerous RDF-centric solutions out there: AllegroGraph, 4store, Virtuoso, and many, many others.

# Objectifs du cours

*Comprendre le fonctionnement des systèmes d'interrogation de données RDF*

*Implémenter un mini-moteur d'évaluation de requêtes qui incorpore les idées vues en cours*

*Conduire des expériences permettant d'analyser les performances du système réalisé*

# Plan

- **RDF and SPARQL**
- **RDF Row-stores**
- **RDF Column-stores**
- **RDF Graph-stores**

# RDF Triples

<Albert\_Einstein, isA, physicist>

<Albert\_Einstein, bornIn, Ulm>

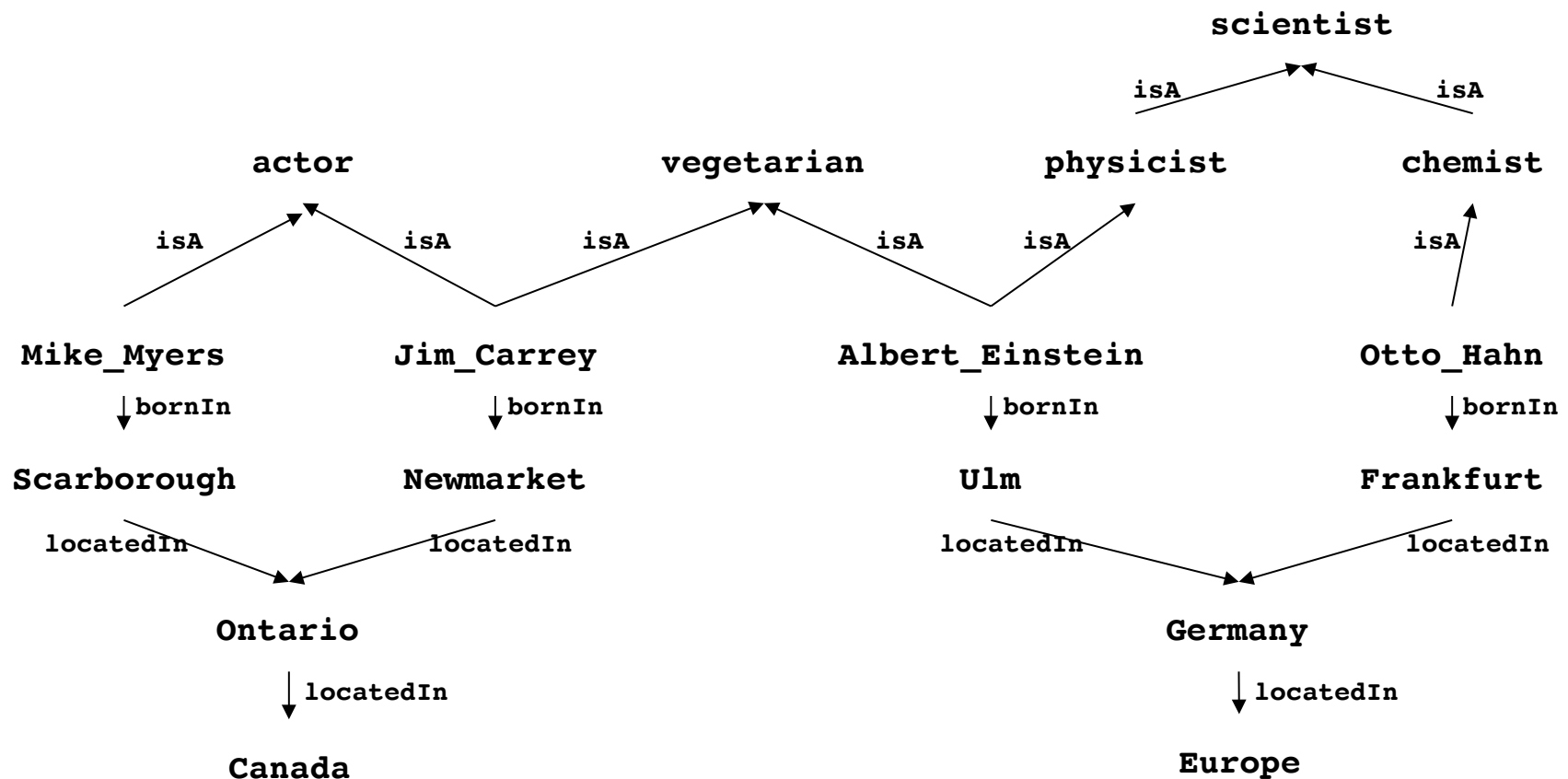
<Albert\_Einstein, isA, vegetarian> ...

# Graph Notation

<Albert\_Einstein, isA, physicist>

<Albert\_Einstein, bornIn, Ulm>

<Albert\_Einstein, isA, vegetarian> ...



# RDF is strictly more than a Graph Database

`<Albert_Einstein, isA, physicist>`

`<isA, rdfs:subPropertyOf, rdfs:subClassOf>`

`SELECT ?x ?y`

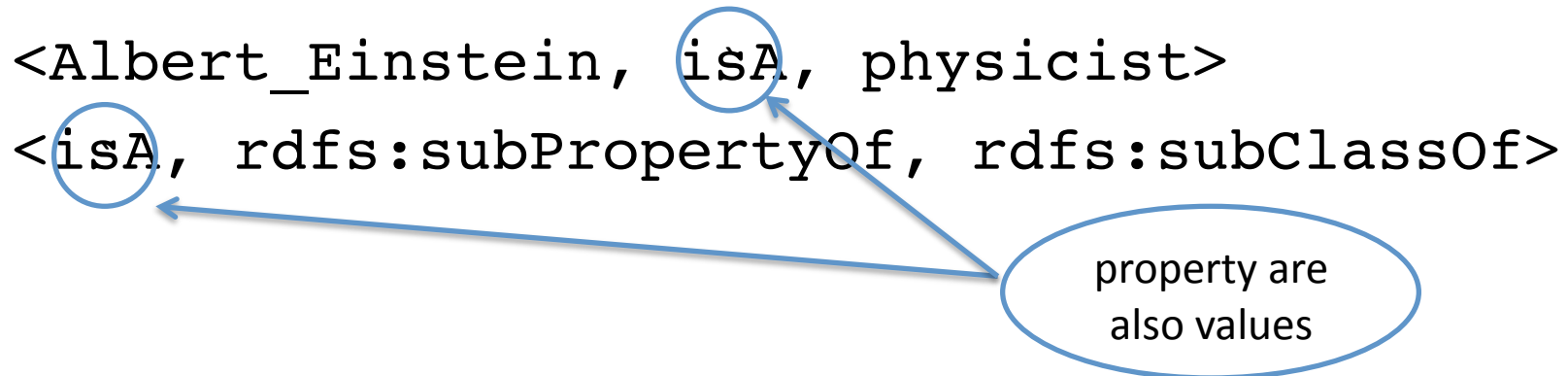
`WHERE {`

`?x ?p ?y .`

`?p rdfs:subPropertyOf, rdf:subClassOf}`

`}`

# RDF is strictly more than a Graph Database



```
SELECT ?x ?y  
WHERE {  
    ?x ?p ?y .  
    ?p rdfs:subPropertyOf, rdf:subClassOf}  
}
```



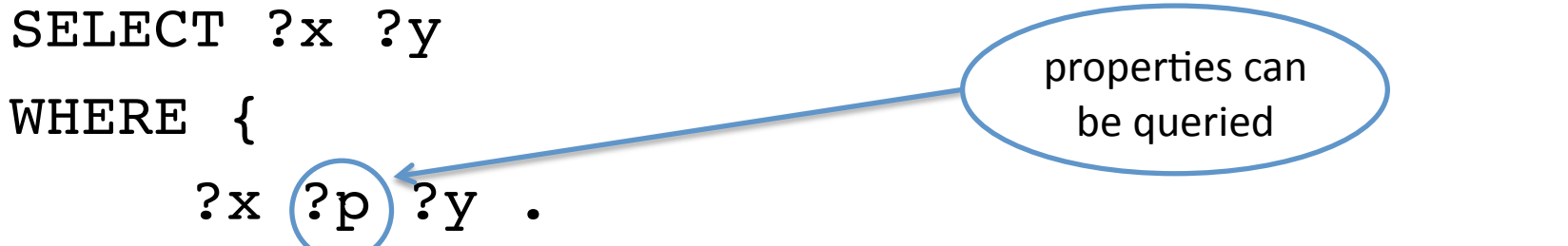
# RDF is strictly more than a Graph Database

`<Albert_Einstein, isa, physicist>`  
`<isa, rdfs:subPropertyOf, rdfs:subClassOf>`



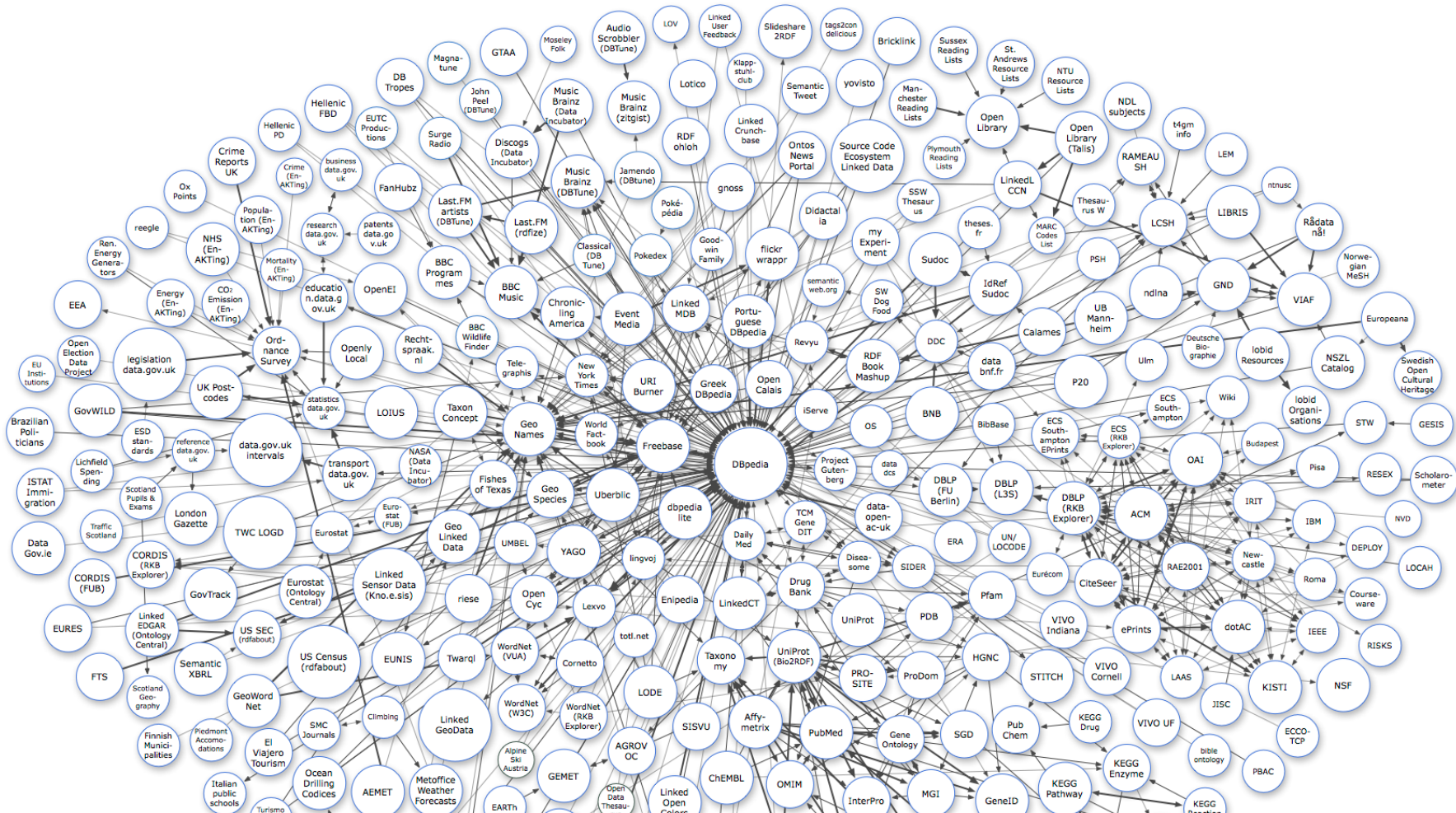
property are  
also values

```
SELECT ?x ?y
WHERE {
  ?x ?p ?y .
  ?p rdfs:subPropertyOf, rdf:subClassOf}
}
```



properties can  
be queried

Why caring about RDF data?  
Open Data are gaining momentum !



**More than 30 billion triples in more than 200 sources across the LOD cloud**  
**DBPedia: 3.4 million entities, 1 billion triples**

# Queries can be complex, too

```
SELECT DISTINCT ?a ?b ?lat ?long WHERE
{ ?a dbpedia:spouse ?b.
  ?a dbpedia:wikilink dbpediares:actor.
  ?b dbpedia:wikilink dbpediares:actor.
  ?a dbpedia:placeOfBirth ?c.
  ?b dbpedia:placeOfBirth ?c2.
  ?c owl:sameAs ?c2.
  ?c2 pos:lat ?lat.
  ?c2 pos:long ?long.
}
```

# SPARQL 1.0 / 1.1

- Query language for RDF suggested by the W3C
- SPARQL main building block: **triple patterns**

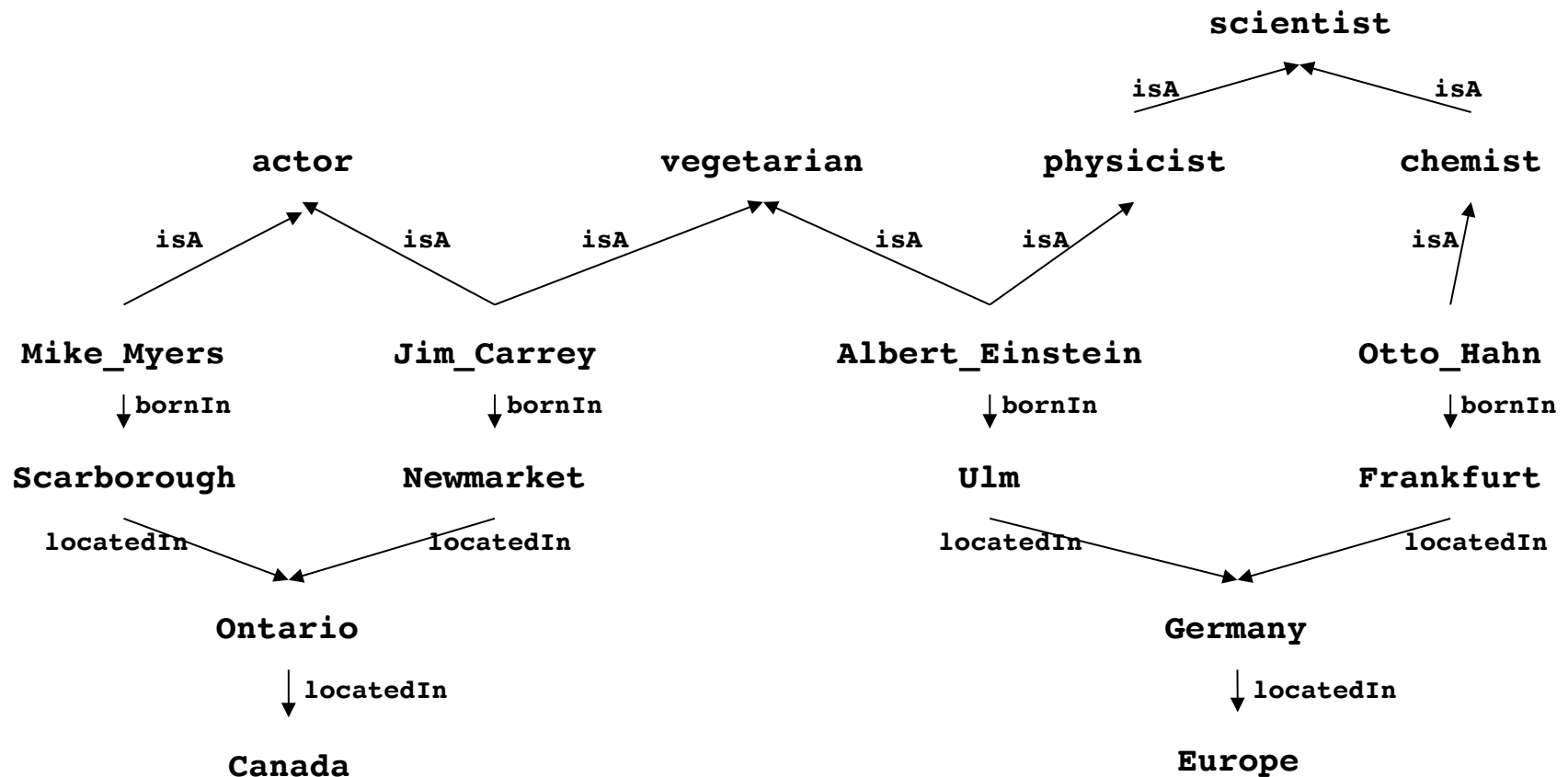
`?x wrote_book ?y`

- Like **select-project-join** for relational databases

# SPARQL – Example

Example query:

**Find all actors from Ontario** (that are in the knowledge base)



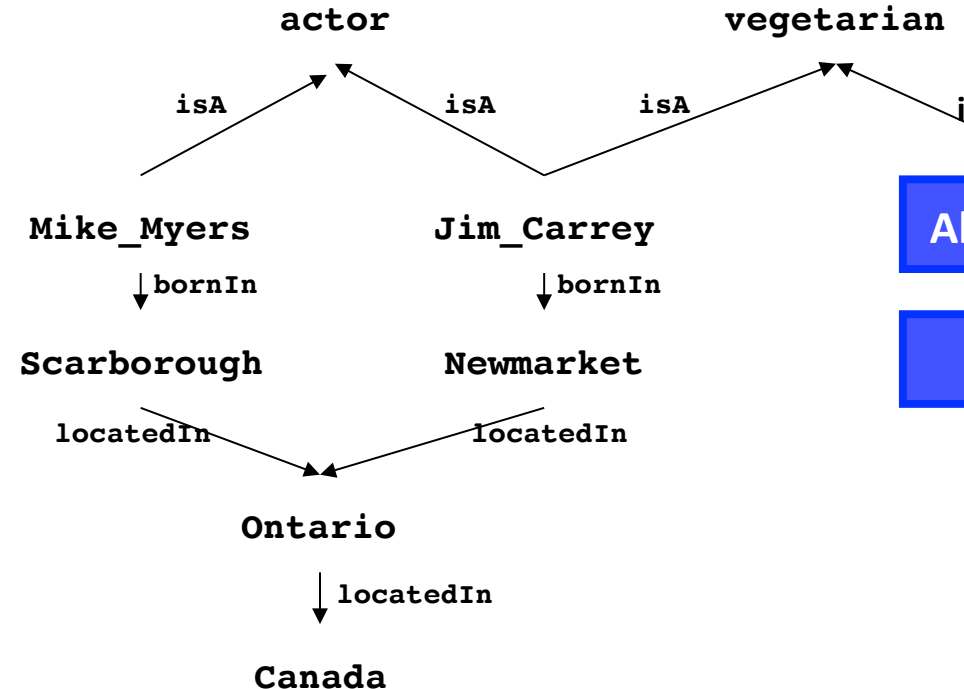
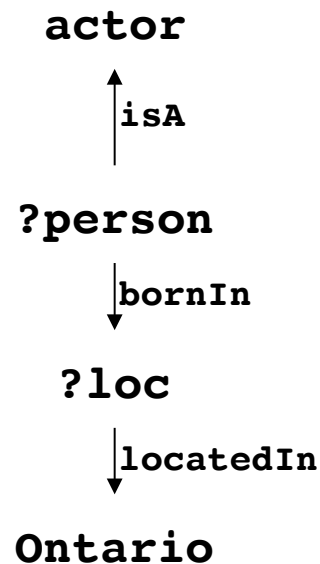
# SPARQL – Example

Example query:

**Find all actors from Ontario** (that are in the knowledge base)

```
SELECT ?person WHERE {  
  ?person isA actor.  
  ?person bornIn ?loc .  
  ?loc locatedIn Ontario .  
}
```

Find **subgraphs** of this form:



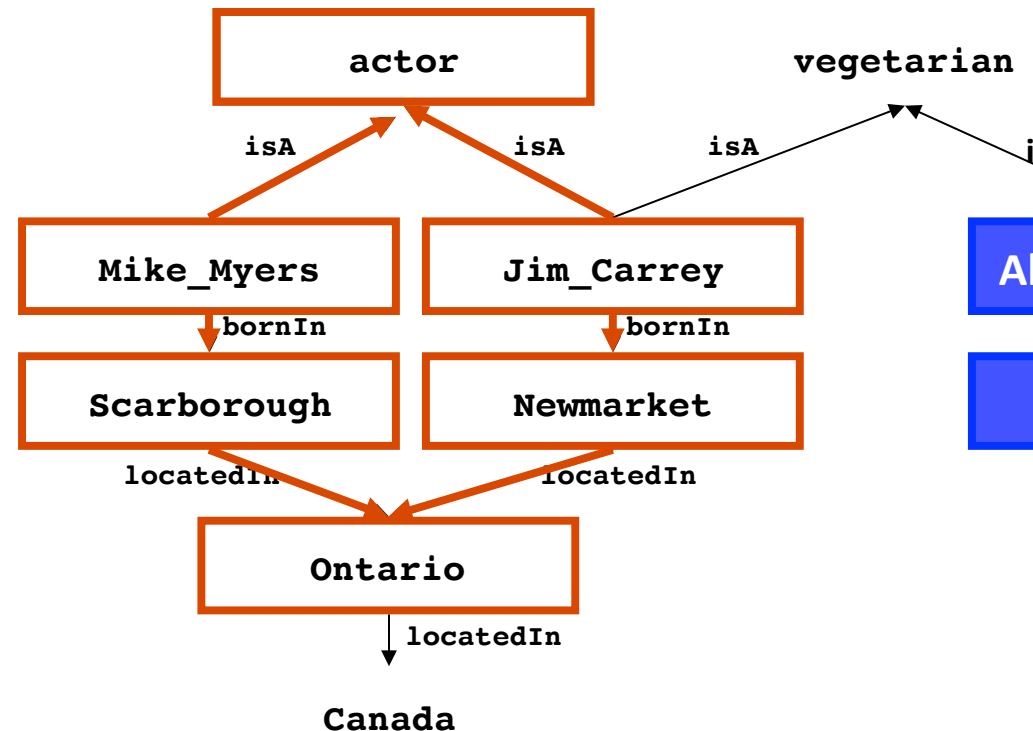
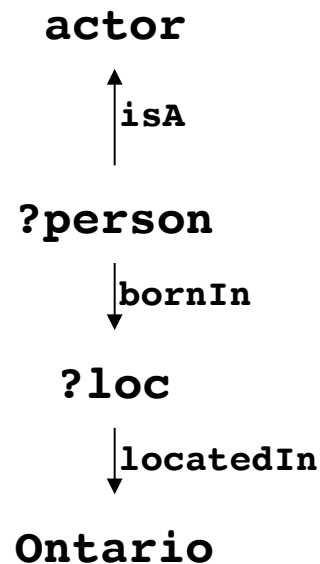
# SPARQL – Example

Example query:

**Find all actors from Ontario** (that are in the knowledge base)

```
SELECT ?person WHERE {  
  ?person isA actor.  
  ?person bornIn ?loc .  
  ?loc locatedIn Ontario .  
}
```

Find **subgraphs** of this form:



# Questions

- How to store RDF data ?
- How to query RDF data ?
- We will study three main approaches
  - Row-stores
  - Column-stores
  - Graph-stores



**ROW-STORES**

# Row-store

Classic relational database, storing relations by rows.  
(Postgres, Oracle, DB2, MySQL, ... )

<b>product</b>	<b>country</b>	<b>sales</b>
car	US	40K
bike	US	7K
	...	



**row1**

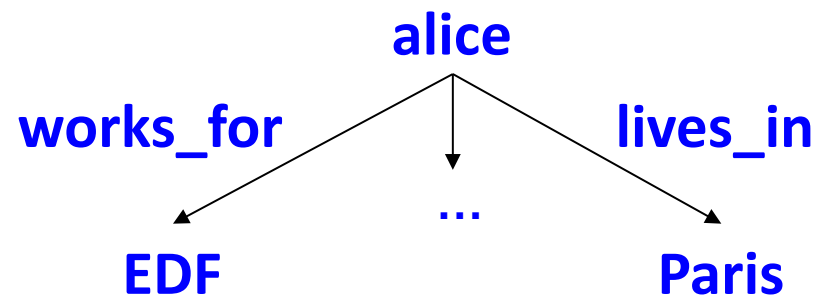
car@US@40K

**row2**

bike@US@7K

...

# RDF in a Relational Row-store



1 triple = 1 edge  
in the RDF Graph

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		



**row1**

alice@work  
s\_for@EDF

**row2**

alice@live  
sIn@Paris

...

# Giant-Table (Jena, HexaStore, RDF-3X)

1. Store triples in one **giant** 3-attribute table
2. Convert SPARQL to equivalent SQL
3. *Magic* : the database will do the rest

<b>subject</b>	<b>predicate</b>	<b>object</b>
alice	works_for	EDF
alice	lives_in	Paris
...		

**Giant**  
1 Billion  
Triples  
=  
1 Billion  
lines

# Conversion of SPARQL to SQL

triple pattern	→	FROM/WHERE
Shared variables	→	(self)JOIN conditions
Constants	→	WHERE conditions
FILTER conditions	→	WHERE conditions
OPTIONAL clauses	→	OUTER JOINS
UNION clauses	→	UNION expressions

# Conversion of Triple Patterns (with Constants)

**SPARQL** >

```
SELECT ?x WHERE {?x lives_in ?y}
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of Triple Patterns (with Constants)

**SPARQL** >

```
SELECT ?x WHERE {?x lives_in ?y}
```

**SQL** >

```
SELECT subject FROM Giant-Table  
WHERE predicate = "lives_in"
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of Shared Variables

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		



# Conversion of Shared Variables

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

**SQL** >

```
SELECT L.subject, L.object, W.object  
FROM Giant-Table as L, Giant-Table as W  
WHERE L.predicate = "lives_in"  
AND   W.predicate = "works_for"  
AND   L.subject = W.subject
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of FILTER conditions

**SPARQL** >

```
SELECT ?x WHERE { ?x lives_in ?y .  
                  FILTER(?y!="New York") }
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of FILTER conditions

**SPARQL** >

```
SELECT ?x WHERE { ?x lives_in ?y .  
                  FILTER(?y!="New York") }
```

**SQL** >

```
SELECT subject  
FROM Giant-Table  
WHERE predicate = "lives_in"  
      AND object != "New York"
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of UNION clauses

**SPARQL** >

```
SELECT ?x WHERE {  
    { ?x lives_in ?y }  
    UNION { ?x works_for ?z } }
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of UNION clauses

**SPARQL** >

```
SELECT ?x WHERE {  
    { ?x lives_in ?y }  
    UNION { ?x works_for ?z } }
```

**SQL** >

```
SELECT subject FROM Giant-Table  
WHERE predicate = "lives_in" )  
UNION  
SELECT subject FROM Giant-Table  
WHERE predicate = "works_for"
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of OPTIONAL clauses

**SPARQL** >

```
SELECT ?x WHERE { ?x lives_in ?y .  
                  OPTIONAL { ?x works_for ?z} }
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

# Conversion of OPTIONAL clauses

**SPARQL** >

```
SELECT ?x WHERE { ?x lives_in ?y .  
                  OPTIONAL { ?x works_for ?z} }
```

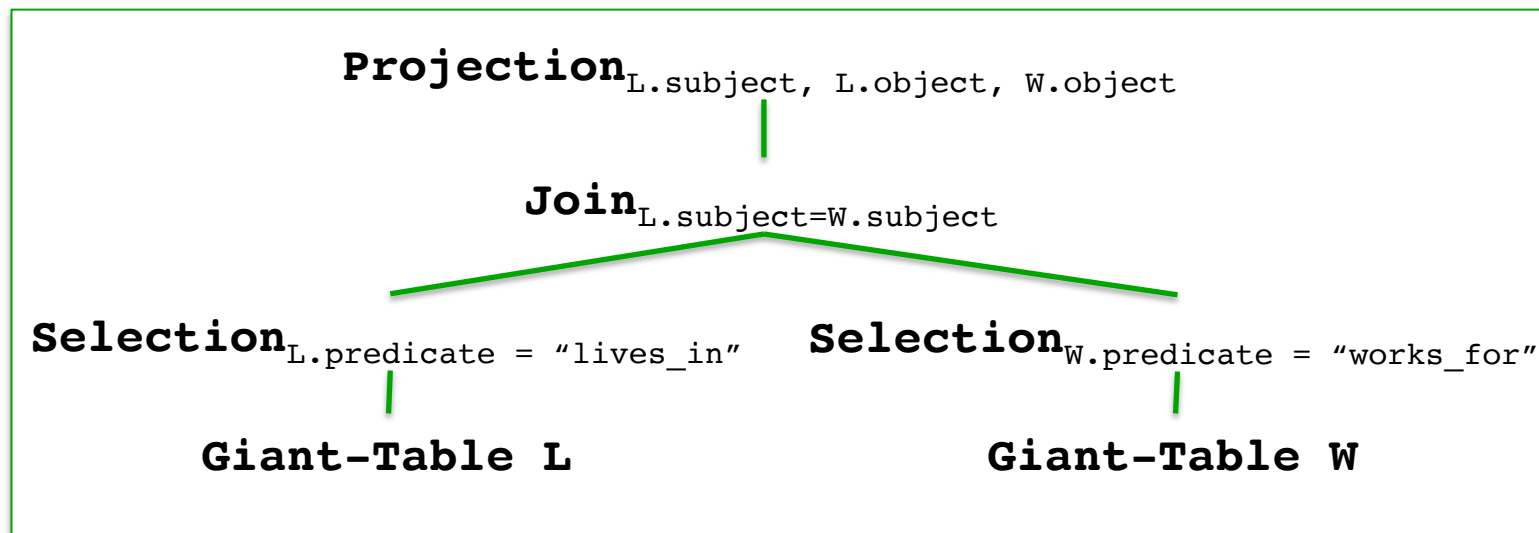
**SQL** >

```
( SELECT      subject      FROM Giant-Table  
  WHERE      predicate = "lives_in" ) L  
LEFT OUTER JOIN  
  ( SELECT      subject      FROM Giant-Table  
    WHERE      predicate = "works_for" ) W  
ON (L.subject = W.subject)
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

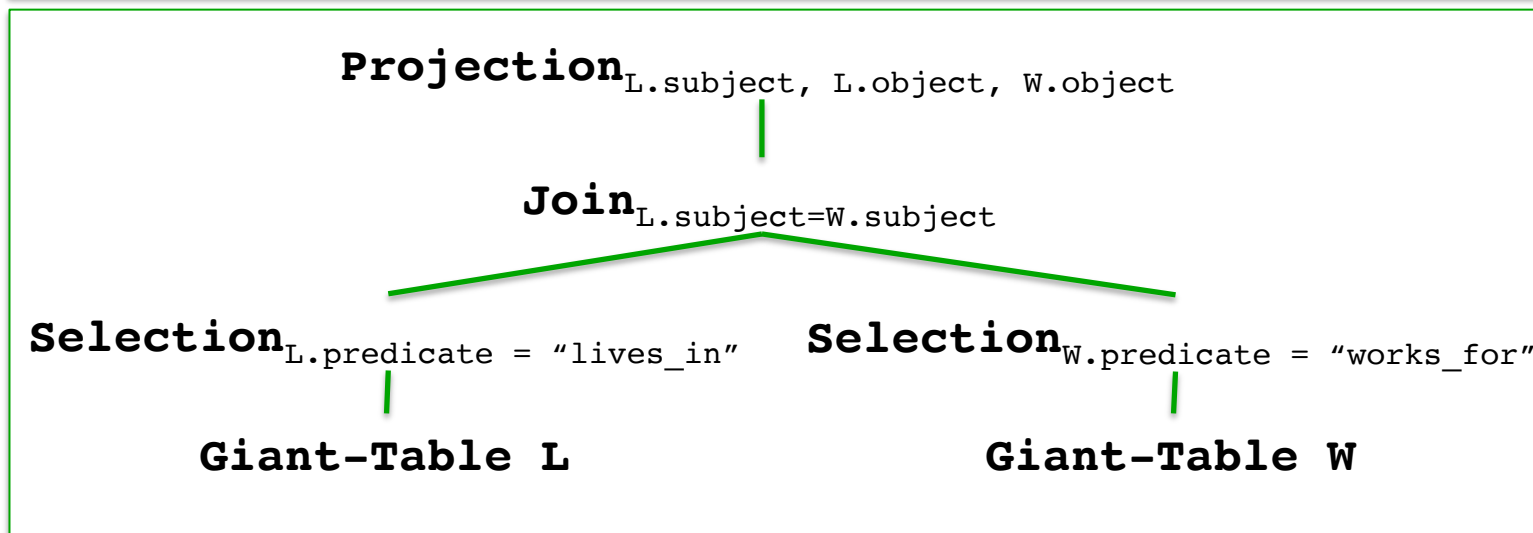
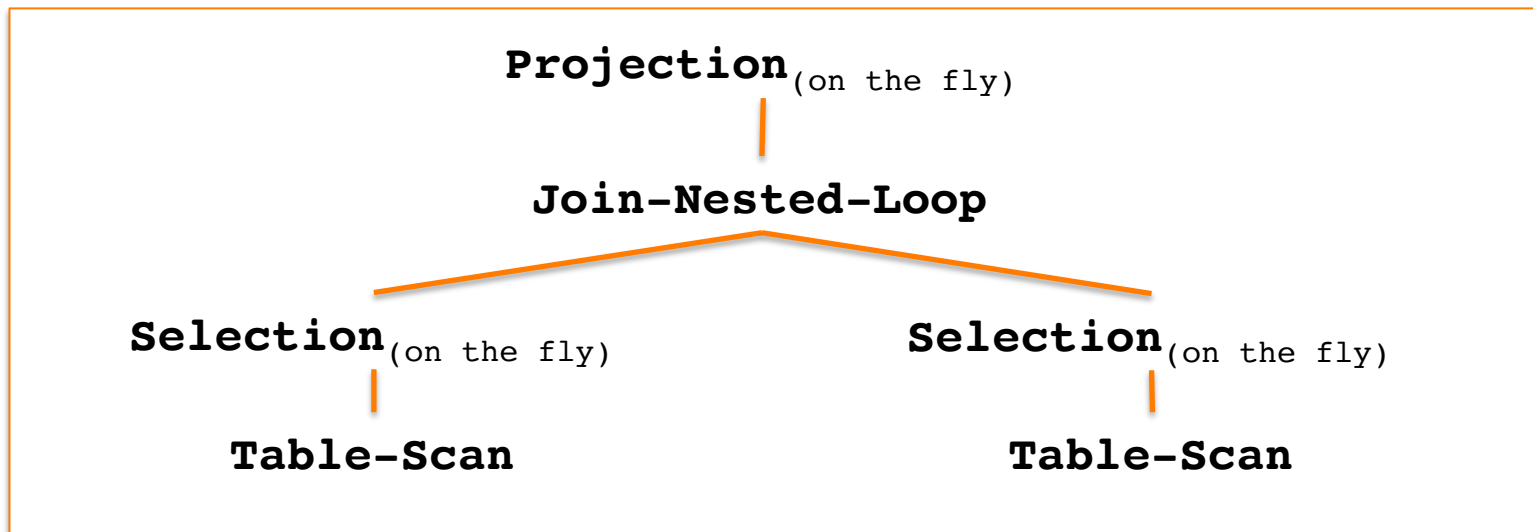
# From SQL to Relational Algebra

```
SELECT L.subject, L.object, W.object
FROM Giant-Table as L, Giant-Table as W
WHERE L.predicate = "lives_in"
AND    W.predicate = "works_for"
AND    L.subject = W.subject
```

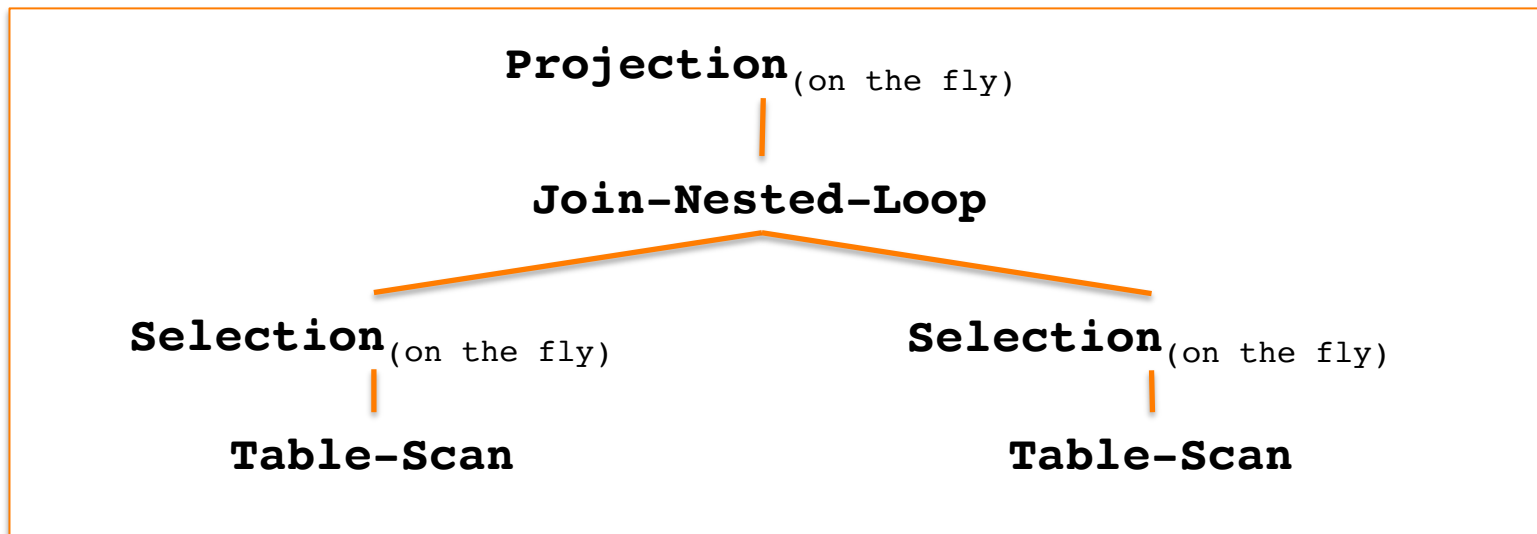




# From Relational Algebra to Physical Plan



# From Relational Algebra to Physical Plan



- Now, performances depend on the row-store.



<b>row1</b>	alice@work	<b>row2</b>	alice@live	...
	s_for@EDF		sIn@Paris	

Is that all?

# Is that all?

**Well, no.**

- Which other logical **schemas** can we use ?
- 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**?

# Is that all?

**Well, no.**

- Which other logical **schemas** can we use ?
- 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**?

**Existing databases need modifications:**

- flexible, extensible, generic storage not needed here
- cannot deal with multiple self-joins of a single table
- often generate bad execution plans

# **EXPLORING ALTERNATIVE RELATIONAL SCHEMAS**

# Problems with **Giant-Table**

- Too many joins, over a too large table.
- Alternative = many tables instead of one
- Property-Tables
- Clustered Property-Tables
- Property-Class

# Property-Tables

- A relational table for each single RDF property.

**Giant-Table**

<b>subject</b>	<b>predicate</b>	<b>object</b>
alice	works_for	EDF
alice	lives_in	Paris
...		

**works\_for**

<b>subject</b>	<b>object</b>
alice	EDF
...	

**lives\_in**

<b>subject</b>	<b>object</b>
alice	Paris
...	



# The former conversion ...

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

**SQL** >

```
SELECT L.subject, L.object, W.object  
FROM Giant-Table as L, Giant-Table as W  
WHERE L.predicate = "lives_in"  
AND   W.predicate = "works_for"  
AND   L.subject = W.subject
```

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris
...		

... now goes as follows

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

**works\_for**

subject	object
alice	EDF
...	

**lives\_in**

subject	object
alice	Paris
...	

... now goes as follows

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

**SQL** >

```
SELECT L.subject, L.object, W.object  
FROM lives_in as L, works_for as W  
AND L.subject = W.subject
```

**works\_for**

subject	object
alice	EDF
...	

**lives\_in**

subject	object
alice	Paris
...	

# Property-Tables

- Syntactically, we get smaller WHERE conditions
- Operationally, we avoid to self join a huge table
  - Keeping intermediary join result small is the key for efficiency in any database system

**works\_for**

<b>subject</b>	<b>object</b>
alice	EDF
...	

**lives\_in**

<b>subject</b>	<b>object</b>
alice	Paris
...	

# But properties can be correlated

YAGO: A Large Ontology from Wikipedia and WordNet

Group 1	Group 2	Group 3	Group 4	Group 5	Group 6
hasWebsite	isLocatedIn	hasGender	isCitizenOf	owns	created
isLocatedIn	isConnectedTo	isAffiliatedTo	wasBornIn	created	directed
owns	type	playsFor	livesIn	participatedIn	acted
created	subClassOf	wasBornIn	diedIn	locatedIn	influences

## **Examples**

<i>BMW</i>	<i>Los Angeles Airport</i>	<i>Alex Ferguson</i>	<i>Jhon Belushi</i>	<i>Apple INC</i>	<i>Charlie Chaplin</i>
------------	--------------------------------	----------------------	---------------------	------------------	----------------------------

# Clustered Properties

**sport\_man\_property\_cluster**

<b>subject</b>	<b>isAffiliatedTo</b>	<b>playsFor</b>	<b>hasGender</b>	<b>wasBornIn</b>
Ferguson	Manchester	Manchester	Male	Scotland
Ronaldo	UNICEF	Inter	Male	Brazil
...				

**employee\_property\_cluster**

<b>subject</b>	<b>works_for</b>	<b>lives_in</b>
alice	EDF	Paris
...		

# Recall the previous conversion ...

**SPARQL** >

```
SELECT * WHERE { ?x lives_in ?y.  
                  ?x works_for ?z }
```

**SQL** >

```
SELECT L.subject, L.object, W.object  
FROM lives_in as L, works_for as W  
AND L.subject = W.subject
```

**works\_for**

<b>subject</b>	<b>object</b>
alice	EDF
alice	Paris
...	

**lives\_in**

<b>subject</b>	<b>object</b>
alice	EDF
alice	Paris
...	

now it goes as follows

**SPARQL** >

```
SELECT * WHERE {  
    ?x lives_in ?y.  
    ?x works_for ?z }  
}
```

**SQL** >

```
SELECT *  
FROM employee_property_cluster  
WHERE lives_in IS NOT NULL  
AND works_for IS NOT NULL
```

<b>subject</b>	<b>works_for</b>	<b>lives_in</b>
alice	EDF	Paris
...		



# Clustered Property-Tables

- Syntactically, we get even less join conditions
- Operationally, we avoid even more joins when properties are within a cluster
  - but we may still have to join two clusters!!

`works_for_lives_in_cluster`

<b>subject</b>	<b>works_for</b>	<b>lives_in</b>
alice	EDF	Paris
	...	

# Correlations do not always hold

<code>&lt;http://yago-knowledge.org/resource/isAffiliatedTo&gt;</code>	<b>2.635.440</b>
<code>&lt;http://yago-knowledge.org/resource/playsFor&gt;</code>	<b>2.575.219</b>
<code>&lt;http://yago-knowledge.org/resource/hasGender</code>	<b>345.794</b>
<code>&lt;http://yago-knowledge.org/resource/wasBornIn&gt;</code>	<b>172.541</b>

- Only 172.541 resources have a value for all properties.
- Clustering properties may waste a lot of storage (**nulls**)

# Correlations do not always hold

<http://yago-knowledge.org/resource/isAffiliatedTo>	2.635.440
<http://yago-knowledge.org/resource/playsFor>	2.575.219
<http://yago-knowledge.org/resource/hasGender	345.794
<http://yago-knowledge.org/resource/wasBornIn>	172.541

## many\_properties\_cluster

subject	isAffiliatedTo	playsFor	hasGender	wasBornIn
				null
		null		null
		null		null
	null	null		null

# Attributes can have multiple values

**many\_properties\_cluster**

<b>subject</b>	<b>isAffiliatedTo</b>	<b>playsFor</b>	<b>hasGender</b>	<b>wasBornIn</b>
Ferguson	Manchester	Manchester	Male	Scotland
Ronaldo	UN	Inter	Male	Brazil
...				

# Attributes can have multiple values

many\_properties\_cluster

subject	isAffiliatedTo	playsFor	hasGender	wasBornIn
Ferguson	Manchester	Manchester	Male	Scotland
Ronaldo	UNICEF	Inter	Male	Brazil
Ronaldo	<b>UNICEF</b>	Milan	<b>Male</b>	<b>Brazil</b>
Ronaldo	<b>UNICEF</b>	Barcelona	<b>Male</b>	<b>Brazil</b>
Ronaldo	<b>UNICEF</b>	RealMadrid	<b>Male</b>	<b>Brazil</b>
Ronaldo	<b>UNICEF</b>	Flamenco	<b>Male</b>	<b>Brazil</b>
...				

- Note : by the way, the 4<sup>th</sup> normal form has been introduced exactly to avoid this.

# Leftover Triples

- Clustered Property Tables induce leftover triples
  - with none of the properties in a cluster
  - belonging to no class
  - extra joins between leftover-triples and clusters

**leftover-triples**

<b>subject</b>	<b>predicate</b>	<b>object</b>
alice	born_in	NY
EDF	located_in	Paris
...		

# The clustered-property table dilemma

- They are complex to design
  - If narrow: reduces nulls, increases unions/joins
  - If wide: reduces unions/joins, increases nulls

# Class-Property Tables

- A table contains all properties of the instances of a given class
- Has all inconvenients of the former method

`class:Book`

<b>subject</b>	<b>title</b>	<b>author</b>	<b>year</b>
ID1	XYZ	Joe Fox	2001
ID3	MNP	<b>null</b>	<b>null</b>
ID6	<b>null</b>	<b>null</b>	2004



# Property Tables: Pros and Cons

## Advantages:

- More in the spirit of existing relational systems
- Saves many self-joins over triple tables

## Disadvantages (mostly for clusters) :

- Potentially many NULL values
- Multi-value attributes problematic
- Schema changes very expensive

# **HEXASTORES : INDEXING IN RDF-3X**

# Hexastores

- RDF Systems introducing 6 indexing on triples
- SPO, PSO, OSP
  - to access data by subject, property, or object
- SOP, POS, OPS
  - to cover all permutations

# Indexes

- Indexes are data-structures that allow a fast ( $\sim$ constant time) access to the stored data
- One can simply add them to boost the performance of any relational schemas
  - warning: an index can be larger than a database!
    - the real question is : when to stop indexing ??
- We look at a more original approach (RDF3X) that completely eliminates the schema design.

## Preprocessing: build a Dictionary for Strings

Map strings to unique integers (e.g., via hashing)

- Regular size (4-8 bytes), much easier to handle
- Dictionary usually kept in main memory

`http://example.fr/Alice` → 1960

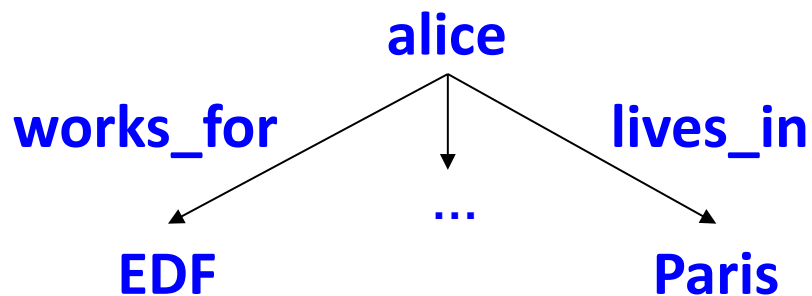
`http://example.fr/Bob` → 3795

`http://example.fr/Charles` → 4634

**If not build carefully, this dictionnary may break the original lexicographic sorting order**

**⇒ FILTER conditions may be more expensive!**

# Dictionnary



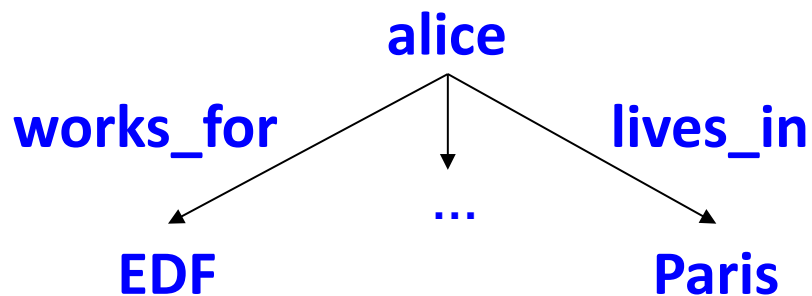
**dictionary**

<b>id</b>	<b>string</b>
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

**Giant-Table**

<b>subject</b>	<b>pred.</b>	<b>object</b>
1		

# Dictionnary



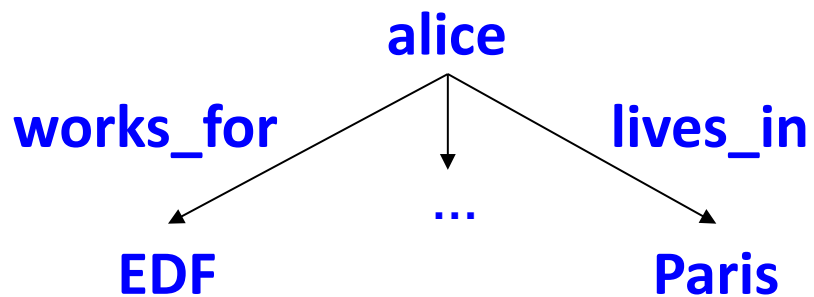
**dictionary**

<b>id</b>	<b>string</b>
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

**Giant-Table**

<b>subject</b>	<b>pred.</b>	<b>object</b>
1	2	

# Dictionnary



**dictionary**

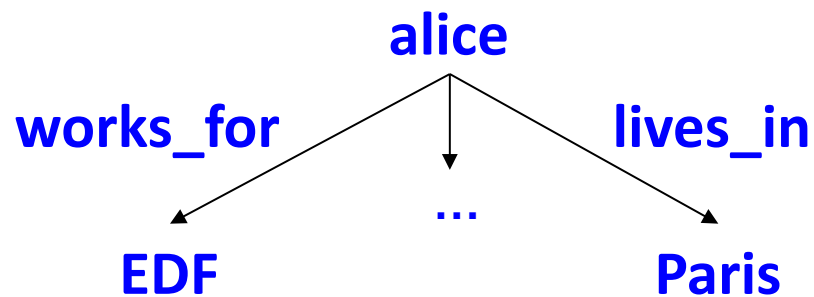
<b>id</b>	<b>string</b>
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

**Giant-Table**

<b>subject</b>	<b>pred.</b>	<b>object</b>
1	2	4



# Dictionary



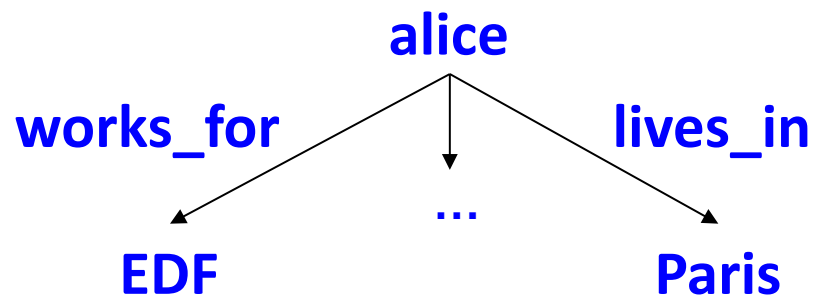
**dictionary**

<b>id</b>	<b>string</b>
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

**Giant-Table**

<b>subject</b>	<b>pred.</b>	<b>object</b>
1	2	4
1		

# Dictionary



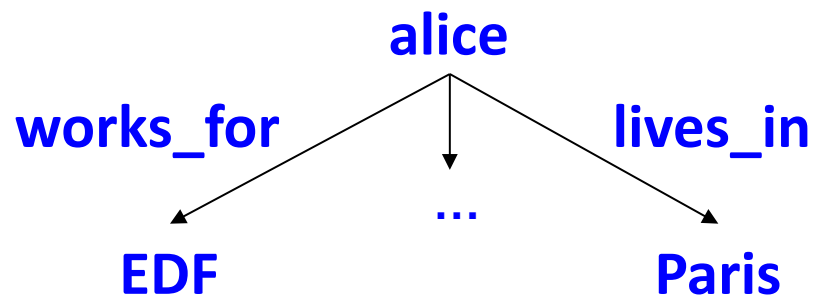
**dictionary**

<b>id</b>	<b>string</b>
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

**Giant-Table**

<b>subject</b>	<b>pred.</b>	<b>object</b>
1	2	4
1	3	

# Dictionnary



dictionary

id	string
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

Giant-Table

subject	pred.	object
1	2	4
1	3	5

up to 10x  
faster!!

# RDF3X Storage and Indexing

- Giant-Table model + ad-hoc implementation (no RDBMs as we have seen before)
- Ad-hoc implementation here means that the Giant-Table is actually fused with indexes (we will see this next)

What triple patterns are found in queries ?

( s   p   o )

( s   p   **?x** )

( s   **?x**   o   )

( **?x**   p   o   )

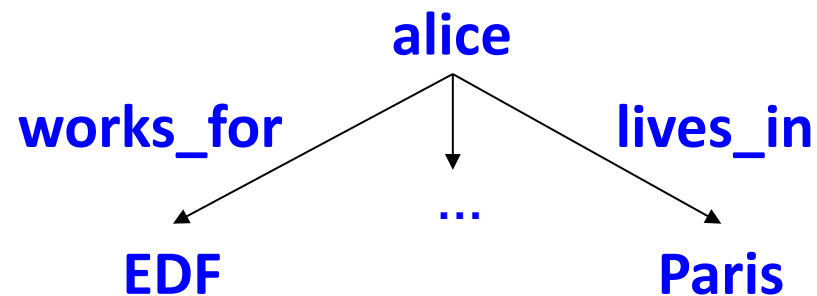
( s   **?x**   **?y** )

( **?x**   p   **?y** )

( **?x**   **?y**   o   )

( **?x**   **?y**   **?z** )

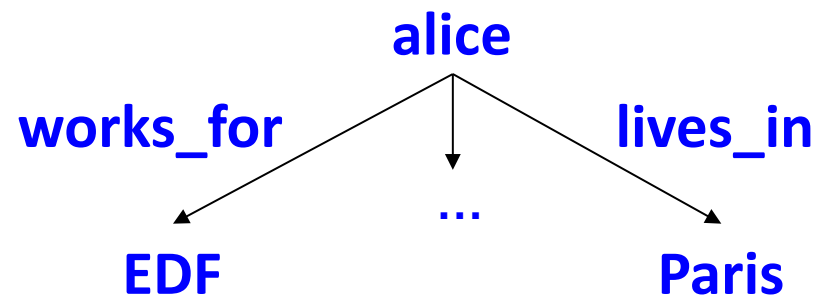
So we can store triples pattern-wise



**Giant-Table<SP0>**

<b>subject</b>	<b>predicate</b>	<b>object</b>
alice	works_for	EDF
alice	lives_in	Paris

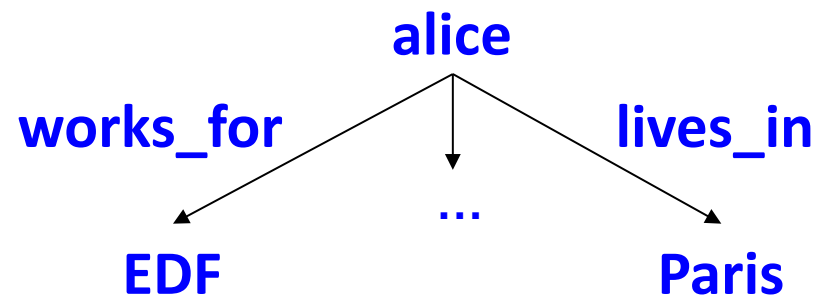
So we can store triples pattern-wise



**Giant-Table<SOP>**

<b>subject</b>	<b>object</b>	<b>predicate</b>
alice	EDF	works_for
alice	Paris	lives_in

So we can store triples pattern-wise

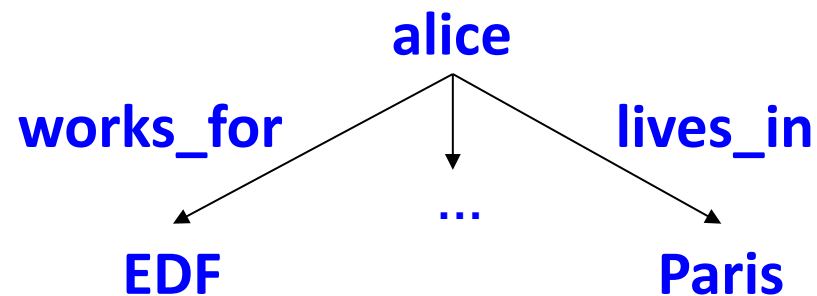


**Giant-Table<OPS>**

<b>object</b>	<b>predicate</b>	<b>subject</b>
EDF	works_for	alice
Paris	lives_in	alice



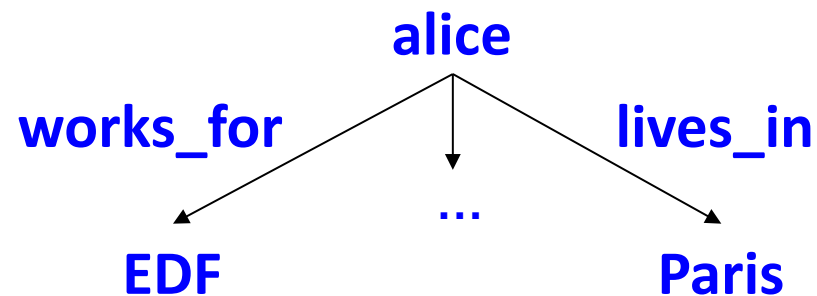
So we can store triples pattern-wise



**Giant-Table<POS>**

<b>predicate</b>	<b>object</b>	<b>subject</b>
works_for	EDF	alice
lives_in	Paris	alice

So we can store triples pattern-wise



**Giant-Table<PSO>**

<b>predicate</b>	<b>subject</b>	<b>object</b>
works_for	alice	EDF
lives_in	alice	Paris

# Why ? Because we deal with row-stores

Giant-Table<SPO>

subject	predicate	object
alice	works_for	EDF
alice	lives_in	Paris

- Easier to match (s p ?x) patterns if stored as



<b>row1</b>	alice@work	<b>row2</b>	alice@live	...
	s_for@EDF		sIn@Paris	

# Why ? Because we deal with row-stores

Giant-Table<POS>

<b>predicate</b>	<b>object</b>	<b>subject</b>
works_for	EDF	alice
lives_in	Paris	alice

- Easier to match ( ?x p o ) patterns if stored as

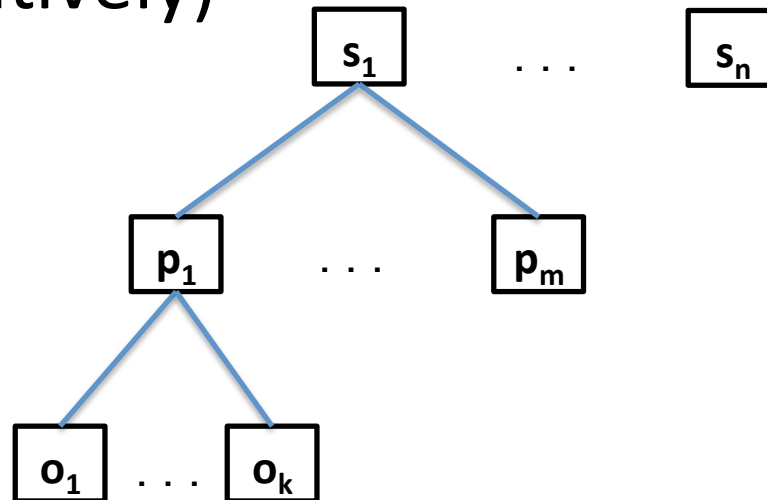
up to 3x  
faster!!



<b>row1</b>	works_for@ EDF@alice	<b>row2</b>	livesIn@Pa ris@alice	...
-------------	-------------------------	-------------	-------------------------	-----

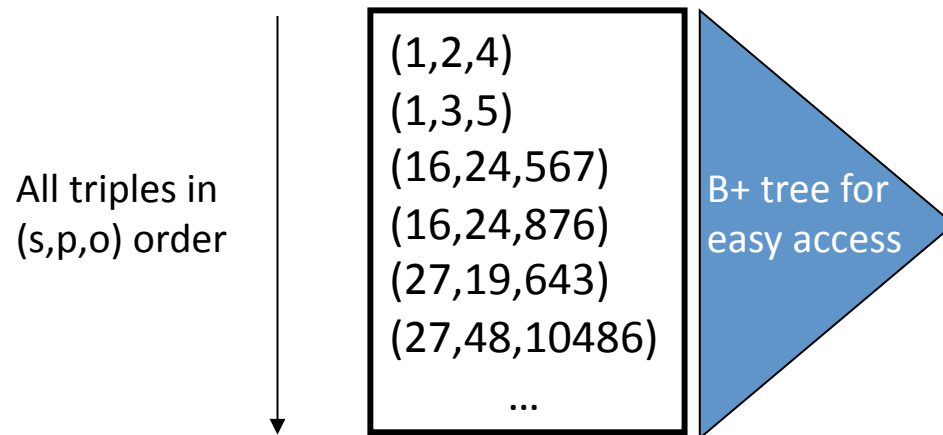
# RDF3X Storage and Indexing

- Similarly RDF3X create 6 indexes
  - SPO ; SOP ; PSO ; POS ; OSP ; OPS
- SPO (Intuitively)

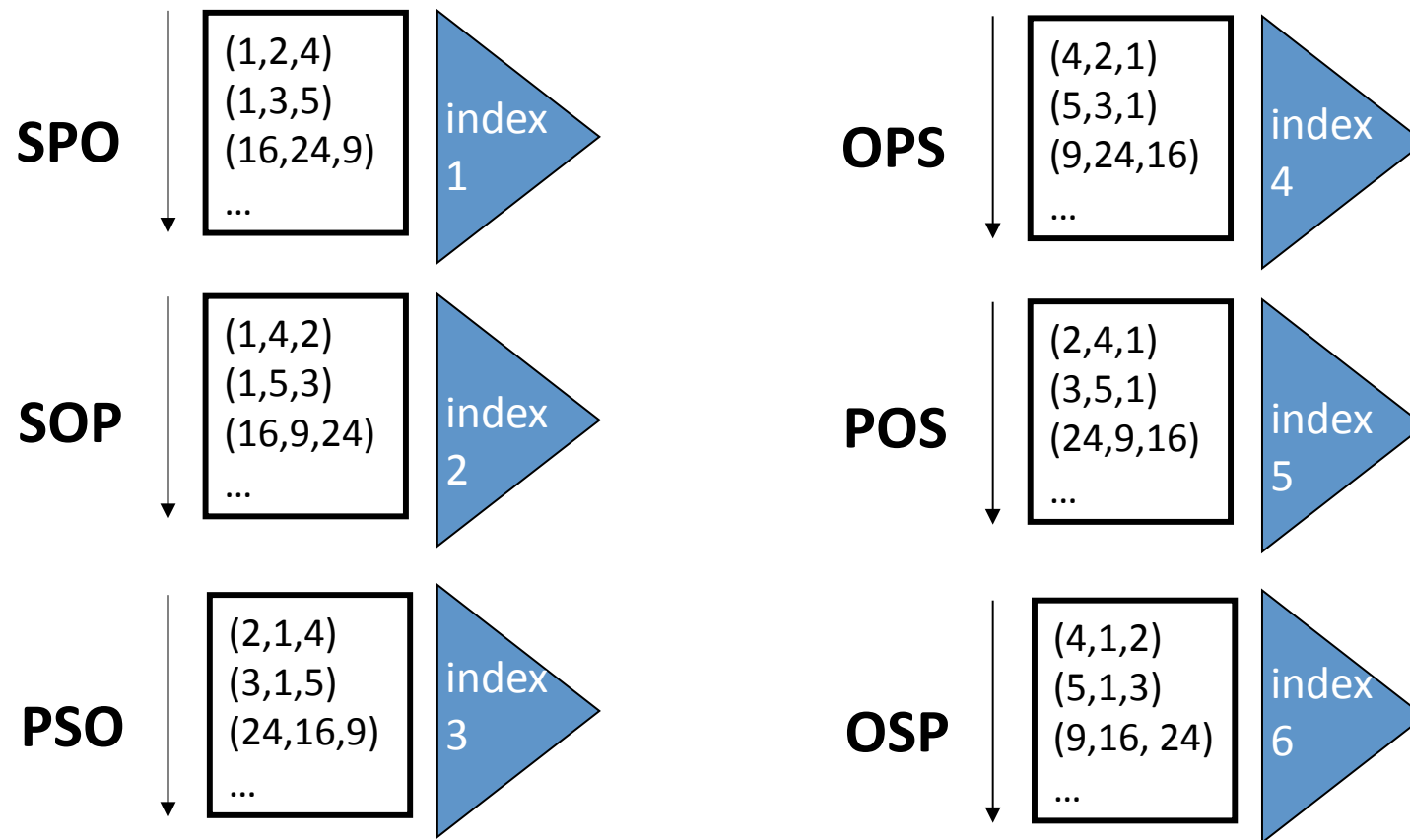


# RDF3X Storage and Indexing

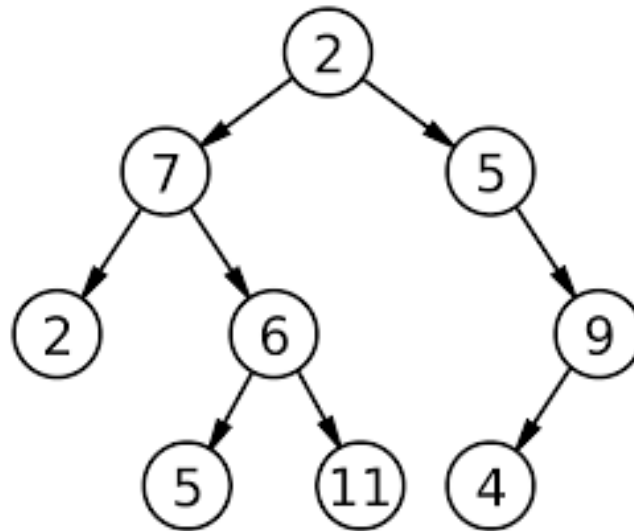
- Similarly RDF3X create 6 indexes
  - SPO ; SOP ; PSO ; POS ; OSP ; OPS
- SPO (in reality) : (B+)-trees over triples



# RDF3X 6 indexes (Hexastore)

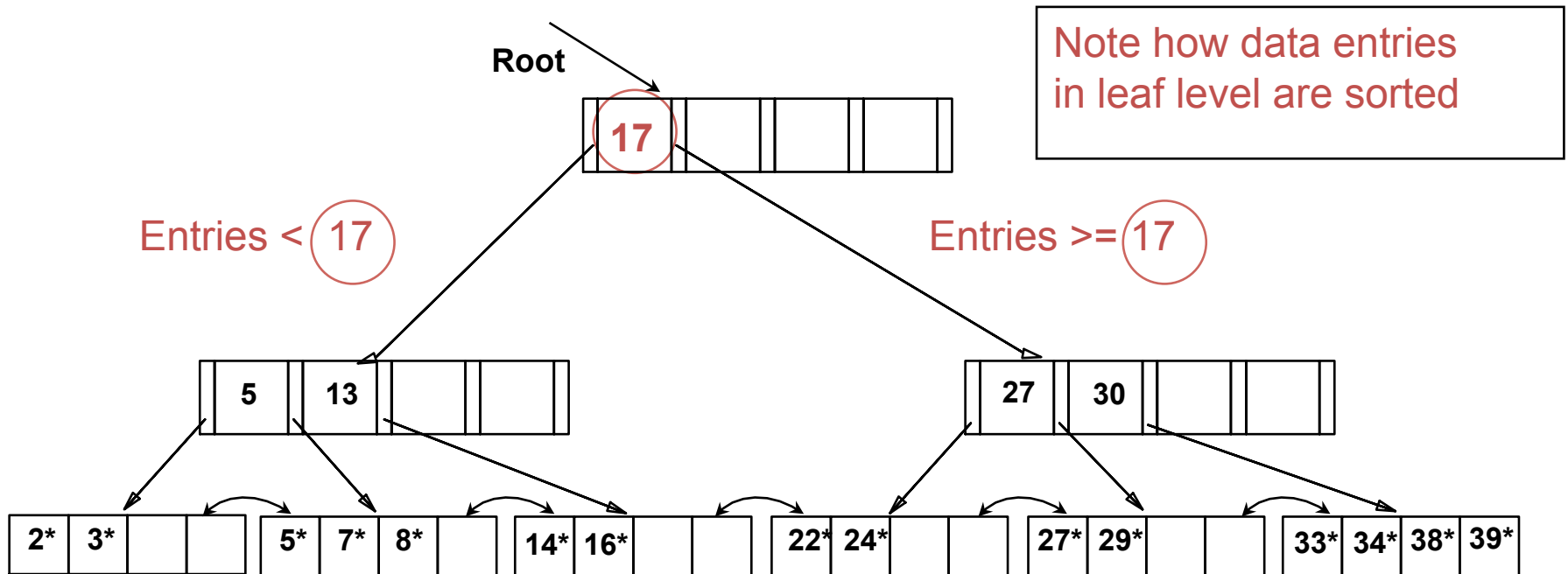


# Binary trees are not enough





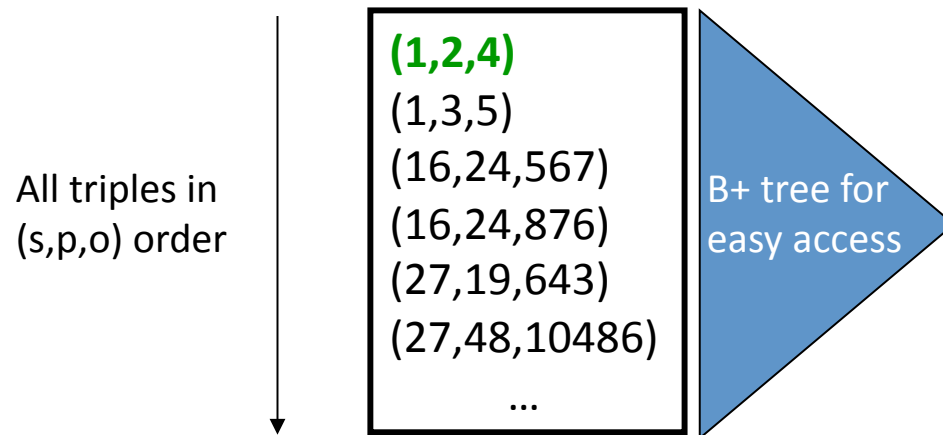
# B+ Tree



# RDF3X Query Processing

SELECT ?x where { alice, lives\_in, ?x }

- Lookup ids : **alice** → 1, **lives\_in** → 2
- Read results while prefix (1,2) matches: **(1,2,4)**



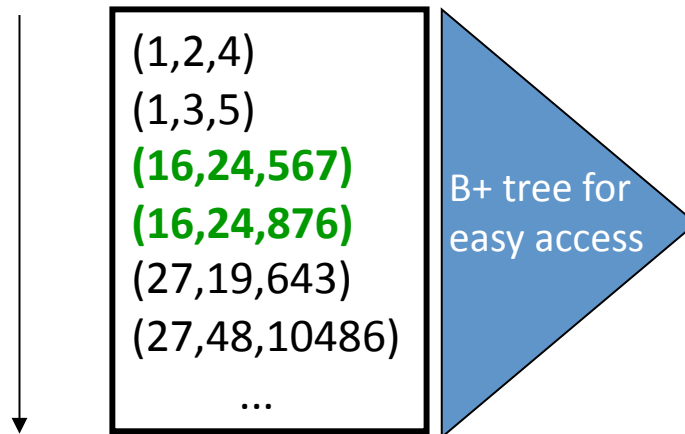
# RDF3X Query Processing

SELECT ?x where { Einstein, invented, ?x }

- Lookup ids **Einstein** → 16, **invented** → 24
- Read prefix matches (16,24): **(16,24,567)** **(16,24,876)**

already  
sorted

All triples in  
(s,p,o) order



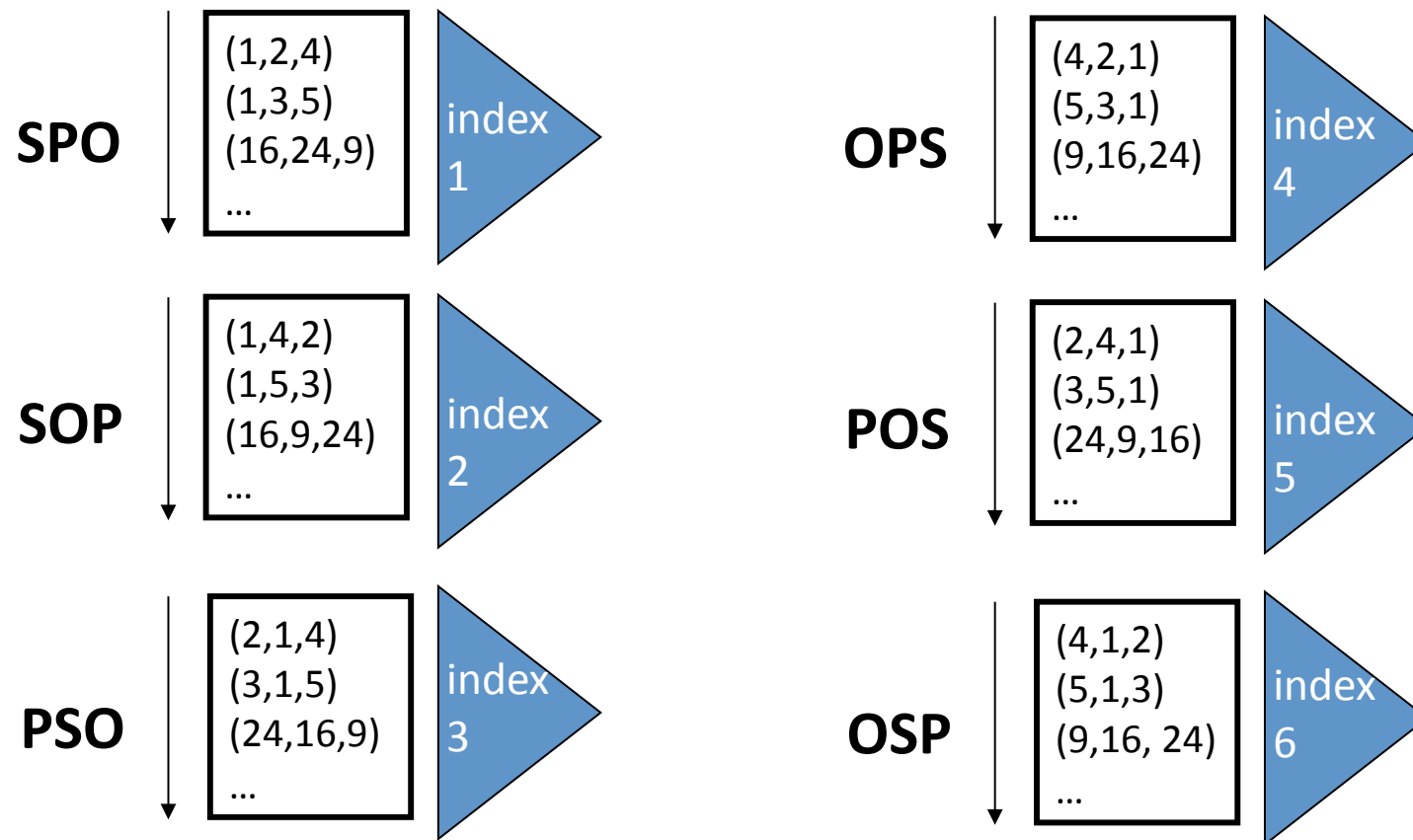
# RDF3X Storage and Indexing

Build clustered indexes for all **six permutations**

- SPO, POS, OSP to cover *all possible* triple patterns
- SOP, OPS, PSO to have *all sort orders* for patterns with two variables

**Triple table no longer needed, all triples in each index**

# How do the indexes work together ?

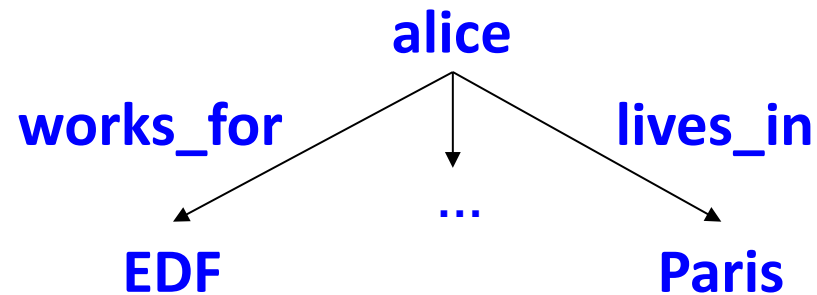


Now, how join two **(2)** triple patterns ?

```
SELECT ?x where {  
    ?x, lives_in, Paris.  
    ?x, works_for, EDF  
}
```

- Naïve-way : evaluate one triple pattern at-a-time and then join (=intersect) the results

# Recall Dictionnary



dictionary

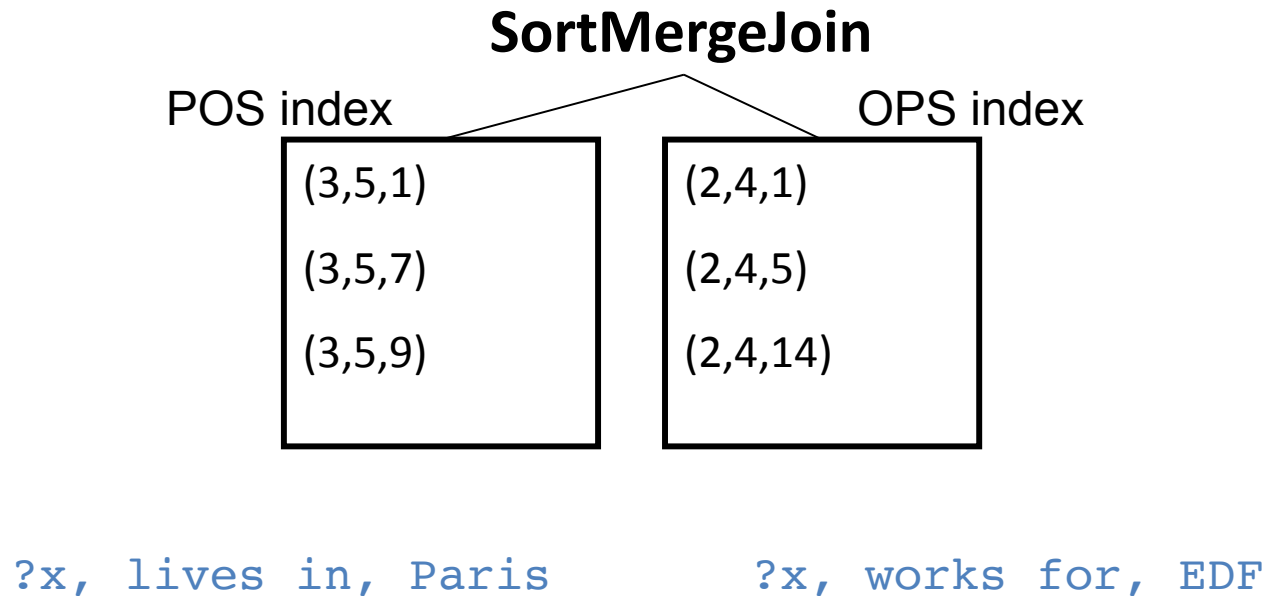
id	string
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

Giant-Table

subject	pred.	object
1	2	4
1	3	5

# RDF3X : Sort-Merge-join

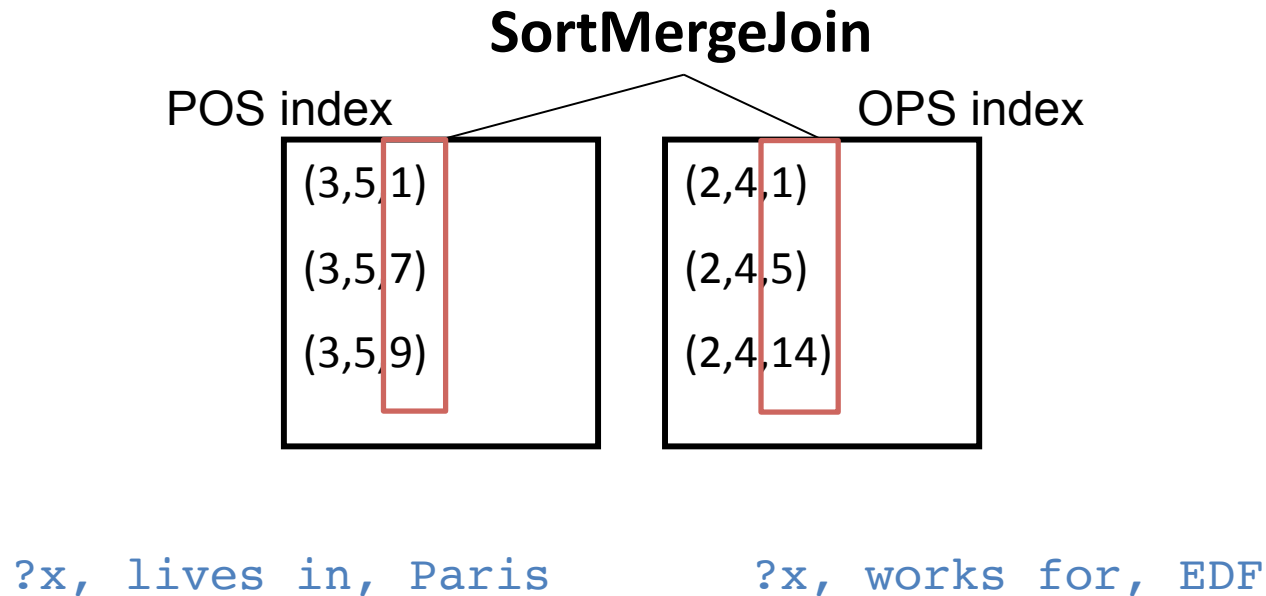
- For example, we decide to use POS & OPS index for 1<sup>st</sup> & 2<sup>nd</sup> pattern, respectively.
  - we will see next why we did this choice





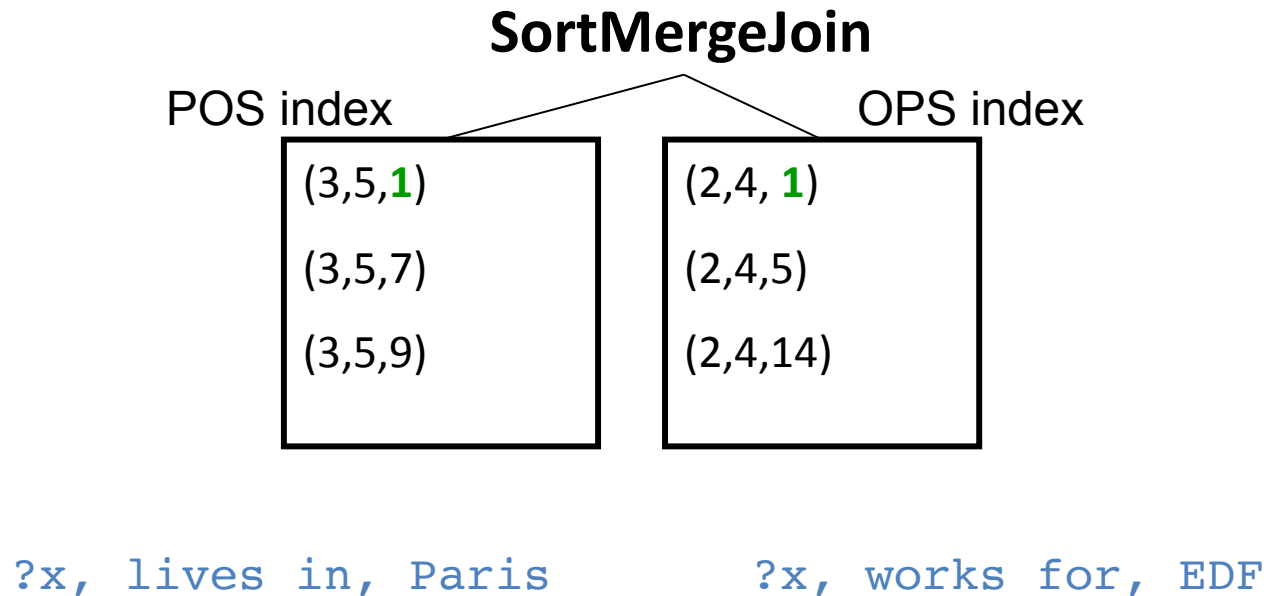
# RDF3X : Sort-Merge-join

- Scan both inputs: join matching values OR skip
- Idea : advance pointer with lower value



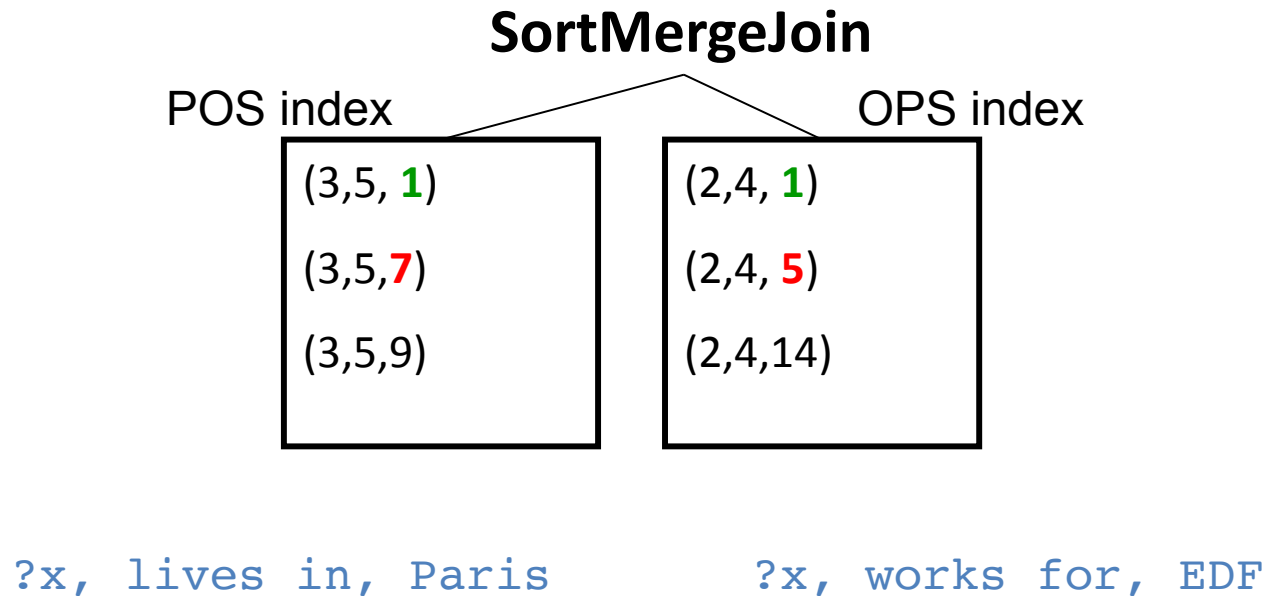
# RDF3X : Sort-Merge-join

- We access POS[3.5.**1**] and OPS[2.4.**1**]
  - we find **1** on both sides -> query result



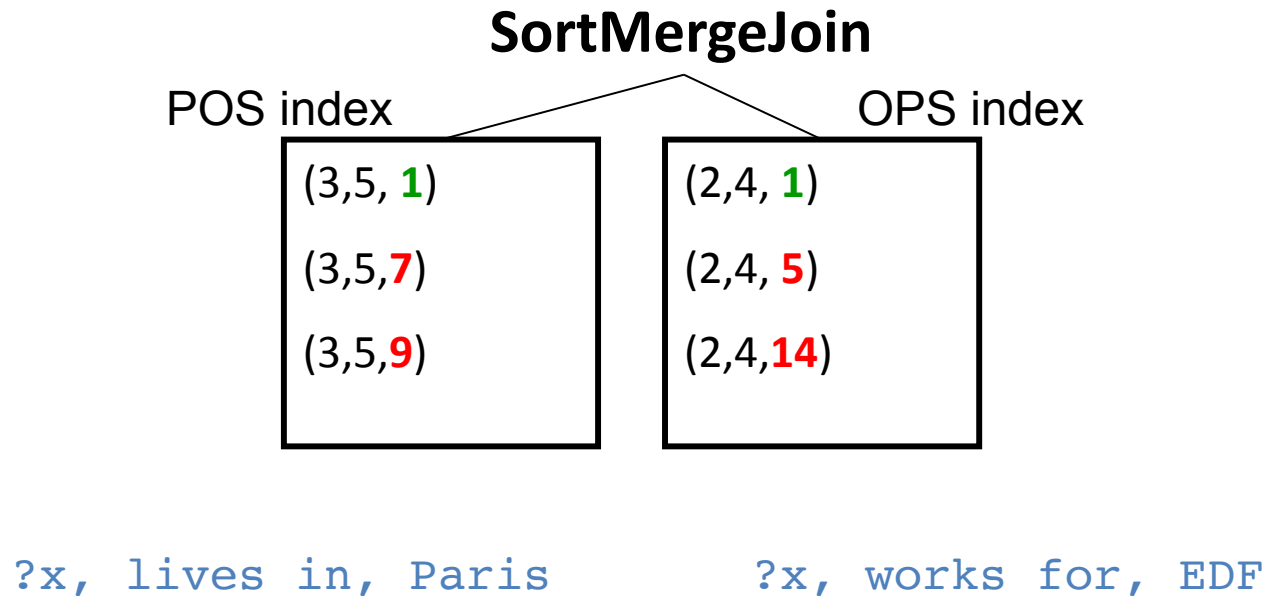
# RDF3X : Sort-Merge-join

- We access POS [3.5.**7**]
  - then we know that OPS[2.4.{**2**..**6**}] are not results



# RDF3X : Sort-Merge-join

- We access OPS[2.4.**14**] then we know that PSO[3.5.{**6**..**13**}] are not results



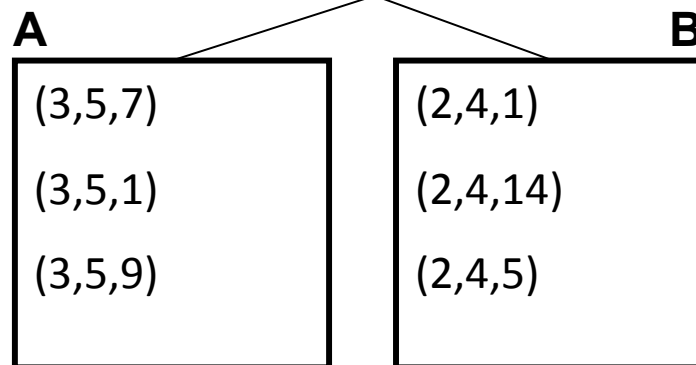
# For unsorted triples ? Nested Loop

for each element ***x*** of **A**

for each element ***y*** of **B**

compare ***x*** with ***y***

## Nested Loop Join



# RDF3X Query Evaluation

- How to join n-triple patterns  $t_1 \dots t_n$ ?

```
SELECT ?x where {  ?x, lives_in, ?y      . t1
                   ?x, works_for, ?z     . t2
                   ?y, isLocatedIn, "US" . t3
                   ?z, isLocatedIn, "US" . t4
                   }
```

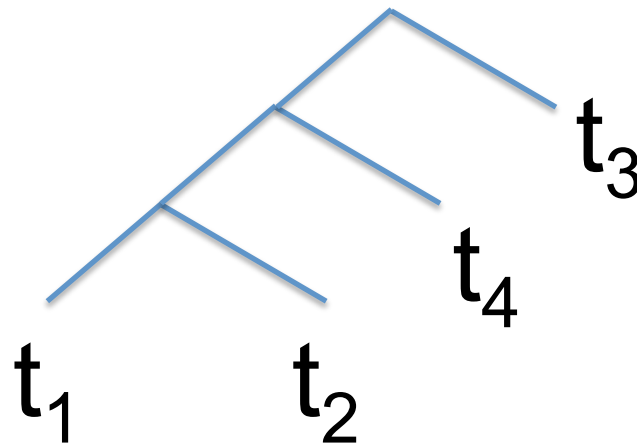
Classical relational problem: chose a left-deep join order, like

$$( (t_1 \text{ Join } t_2) \text{ Join } t_4) \text{ Join } t_3)$$

# RDF3X Query Evaluation

Classical problem: chose a left-deep plan, like

$((t_1 \text{ Join } t_2) \text{ Join } t_4) \text{ Join } t_3$



# Scenario 1 : no triples using property lives\_in

- then

( (t<sub>1</sub> Join t<sub>2</sub>) Join t<sub>4</sub>) Join t<sub>3</sub>)

is optimal as it immediately discovers that the query has no answer

```
SELECT ?x where {  
    ?x, lives_in, ?y      .      t1  
    ?x, works_for, ?      .      t2  
    ?y, isLocatedIn, "US" .      t3  
    ?z, isLocatedIn, "US"      t4  
}
```



## Scenario 2: no triples using property isLocatedIn

- then

$( (t_1 \text{ Join } t_2) \text{ Join } t_4) \text{ Join } t_3$

is **not** optimal as it computes  $(t_1 \text{ Join } t_2)$  before understanding that the query is empty

SELECT ?x where {

?x, lives_in, ?y	.	t <sub>1</sub>
?x, works_for, ?z	.	t <sub>2</sub>
?y, isLocatedIn, "US"	.	t <sub>3</sub>
?z, isLocatedIn, "US"	.	t <sub>4</sub>
}		

Scenario 3 : `isLocatedIn` is very rare and `lives_in` is more frequent than `works_for`

- then

$(t_4 \text{ Join } t_2) \text{ Join } (t_3 \text{ Join } t_1)$

is optimal (in average) as it is likely to keep intermediate results low (but this plan is not left deep!!!)

`SELECT ?x where {`

```
?x, lives_in, ?y      .      t1
?x, works_for, ?z     .      t2
?y, isLocatedIn, "US" .      t3
?z, isLocatedIn, "US"  .      t4
}
```

# Selectivity Estimates

- How do we know which property, or object or subject is more frequent in a query ?

# Selectivity Estimates

- Example : take the index OPS and (pre-)compute for how many values holds

`?x isLocatedIn "US"`

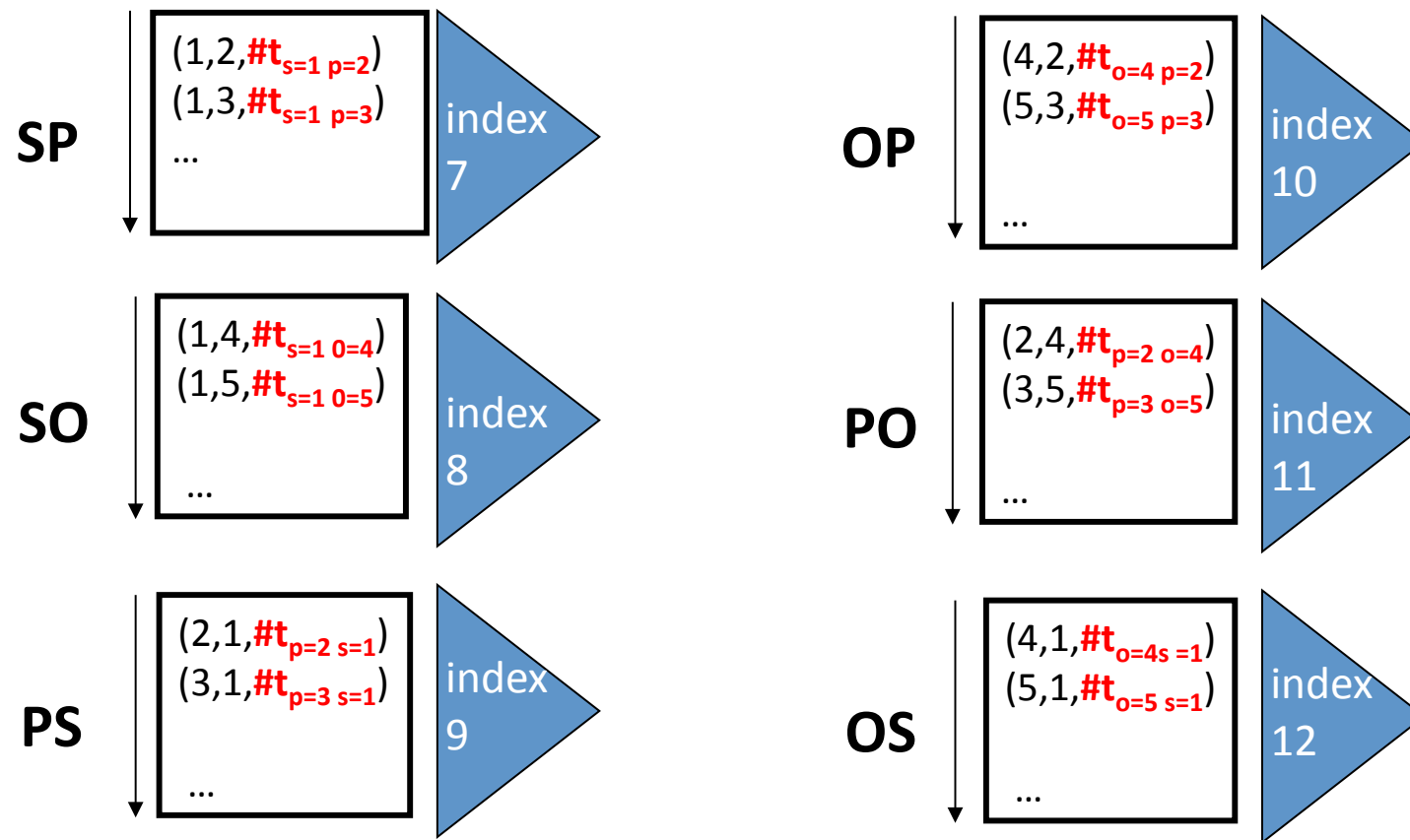
- ***selectivity*** = *#occurrences\_pattern / #triples*
- can be for example 1.000 out of 100.000 (1%)

# Selectivity Estimates

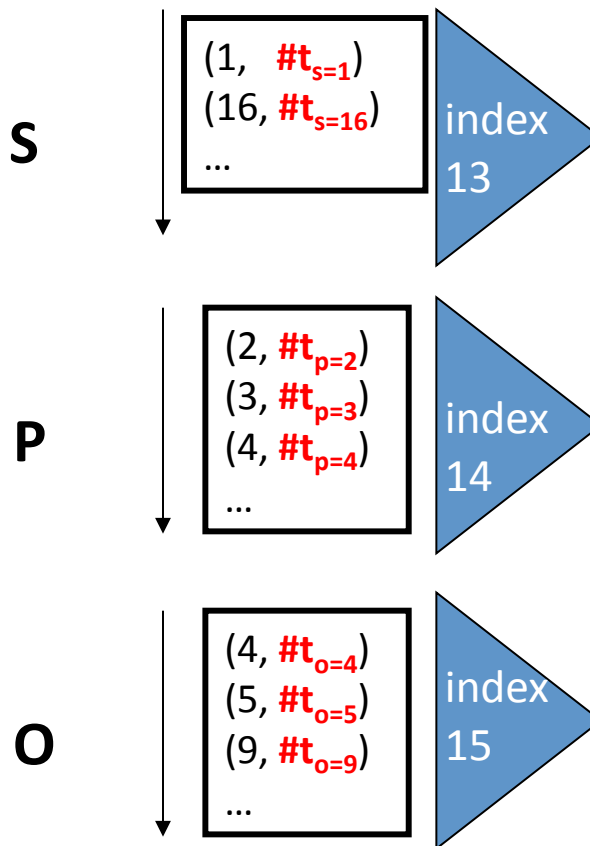
- Goal : skip non-matching triples asap
- Choose OPS instead of POS whenever the object “O” is more selective than property “P”
- Example (extreme) : social graph with 1 property
  - goal : match P only when O is what we are looking for
  - O can be a user or a value. So : match O first, then P.

# RDF3X binary index with counters

$\#t_{s=n \ p=m}$  = **number** of triples with subject **n** and property **m**



# RDF3X unary indexes with counters



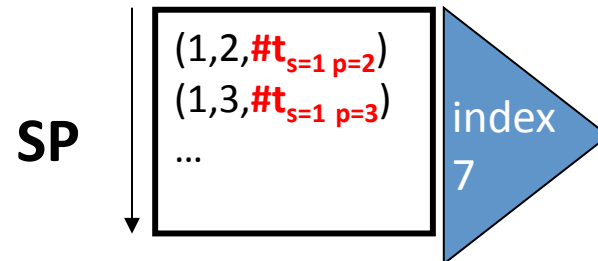
# In reality

- Binary and unary indexes can be fused with the other indexes

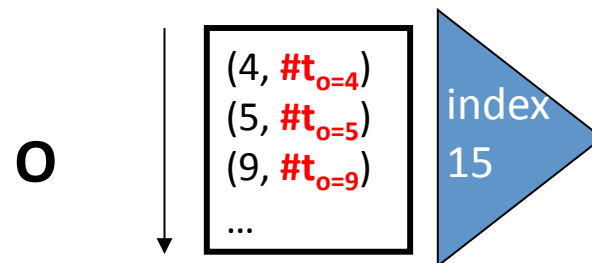


# Selectivity Estimation

- selectivity of  $t = (\text{Obama}, \text{knows}, -)$



- selectivity of  $t = (-, -, \text{"US"})$

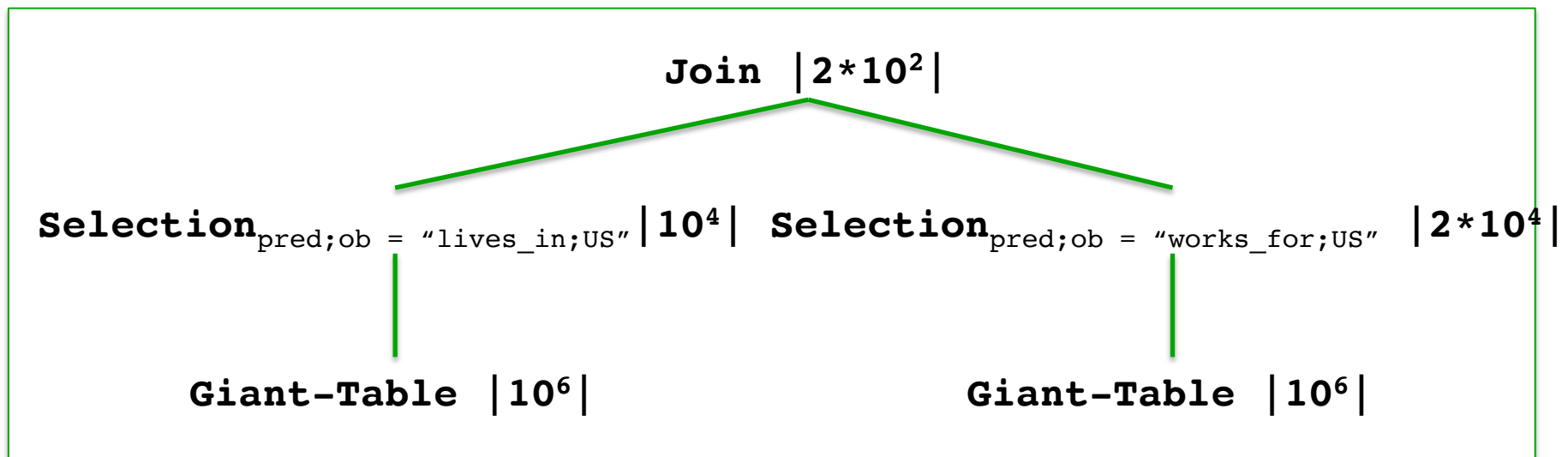


Core ingredients of a good query optimizer are  
selectivity estimators for triple patterns (index scans) and joins

# How to use selectivity estimation?

?x, lives\_in, US .    ?x, works\_for, US

- We want to estimate the **size** of the result. Assume :
  - $10^6$  triples in the database
  - Selectivity of the first pattern is 1%
  - Selectivity of the second pattern is 2%



# Selectivity Estimation (1)

- The size of  $t1 \text{ Join } t2$ , denoted  $|t1 \text{ Join } t2|$ , is *estimated* as

$$|t1 \text{ Join}_{\{\}} t2| \cong |t1| * |t2| = \text{Cart}(t1, t2)$$

where  $|t1| = |DB| * \text{selectivity}(t1)$  (same for  $|t2|$ )

– if  $t1$  and  $t2$  have no common variables

- eg. ?x lives\_in US and ?z works\_for IBM

## Selectivity Estimation (2)

- The size of  $t1 \text{ Join } t2$ , denoted  $|t1 \text{ Join } t2|$ , is ***estimated*** as

independence  
hypothesis!!

$$|t1 \text{ Join}_{?x} t2| = |DB| * \text{selectivity}(t1) * \text{selectivity}(t2)$$

– if  $t1$  and  $t2$  have a common variable

- eg.  $?x$  `lives_in` US and  $?x$  `works_for` IBM

# Selectivity Estimation (3)

- The size of  $Q_1 \text{ Join } Q_2$  is *estimated* as

$$| Q_1 \text{ Join}_{?x} Q_2 | \sim |Q_2| * 1 = |Q_2|$$

- if  $Q_1$  and  $Q_2$  are two (possibly) complex sub-queries with common variables and  $?x$  is a key for  $Q_1$
- Since  $?x$  is a key for  $Q_1$  then one line of  $|Q_2|$  is joined with at most one line of  $|Q_1|$

# Selectivity Estimation (3)

- The size of  $Q_1 \text{ Join } Q_2$  is *estimated* as

$$| Q_1 \text{ Join } Q_2 | \sim \min( |Q_1| , |Q_2| )$$

- if  $Q_1$  and  $Q_2$  are two (possibly) complex sub-queries with common variables
- in reality the formula is a bit more complex

# Selectivity Estimation (3)

If is not a key  $?x$  for  $Q_1$  or  $Q_2$

$$| Q_1 \text{ Join}_{?x} Q_2 | \quad \sim \quad |Q_2| * |Q_1| / V(?x, Q_1)$$

$V(?x, Q_1)$  is the number of distinct values for  $?x$  on  $Q_1$   
note that  $V(?x, Q_1) = |Q_1|$  only if  $?x$  is a key

# Selectivity Estimation (3)

If is not a key  $?x$  for  $Q_1$  or  $Q_2$

how many tuples with same value

$$|Q_1 \text{ Join}_{?x} Q_2| \sim |Q_2| * |Q_1| / V(?x, Q_1)$$


$V(?x, Q_1)$  is the number of distinct values for  $?x$  on  $Q_1$   
note that  $V(?x, Q_1) = |Q_1|$  only if  $?x$  is a key



# Selectivity Estimation (3)

If is not a key  $?x$  for  $Q_1$  or  $Q_2$

how many tuples with same value

$$|Q_1 \text{ Join}_{?x} Q_2| \sim |Q_2| * \underbrace{|Q_1| / V(?x, Q_1)}_{\text{more than 1 tuple}}$$

$V(?x, Q_1)$  is the number of distinct values for  $?x$  on  $Q_1$   
note that  $V(?x, Q_1) = |Q_1|$  only if  $?x$  is a key

# Selectivity Estimation (3)

If is not a key  $?x$  for  $Q_1$  or  $Q_2$  , more precisely

$$| Q_1 \text{ Join}_{?x} Q_2 | \sim \min \left\{ \begin{array}{l} |Q_2| * |Q_1| / V(?x, Q_1) \\ |Q_1| * |Q_2| / V(?x, Q_2) \end{array} \right.$$

$V(?x, Q_1)$  is the number of distinct values for  $?x$  on  $Q_1$   
note that  $V(?x, Q_1) = |Q_1|$  only if  $?x$  is a key

# Selectivity Estimation (3)

If is not a key  $?x$  for  $Q_1$  or  $Q_2$  , finally

$$| Q_1 \text{ Join}_{?x} Q_2 |$$

- if impossible (or too expensive) to compute  $V(?x, Q_1)$  take for instance  $V(?x, Q_1) = |Q_1|$ 
  - this means that  $| Q_1 \text{ Join}_{?x} Q_2 | \sim \min(|Q_1|, |Q_2|)$
- if more than one variable, take the min among all variables,  $|Q_1| * |Q_2| / V(?x, Q_2)$

Why off-the-shelf relational database estimation  
over a **Giant-Table** cannot work well ?

$|t1 \text{ Join } t2|$  = applies formula for queries as just seen

instead of



independence  
hypothesis!!

$|t1 \text{ Join } t2| = |DB| * \text{selectivity}(t1) * \text{selectivity}(t2)$

# Back to our query



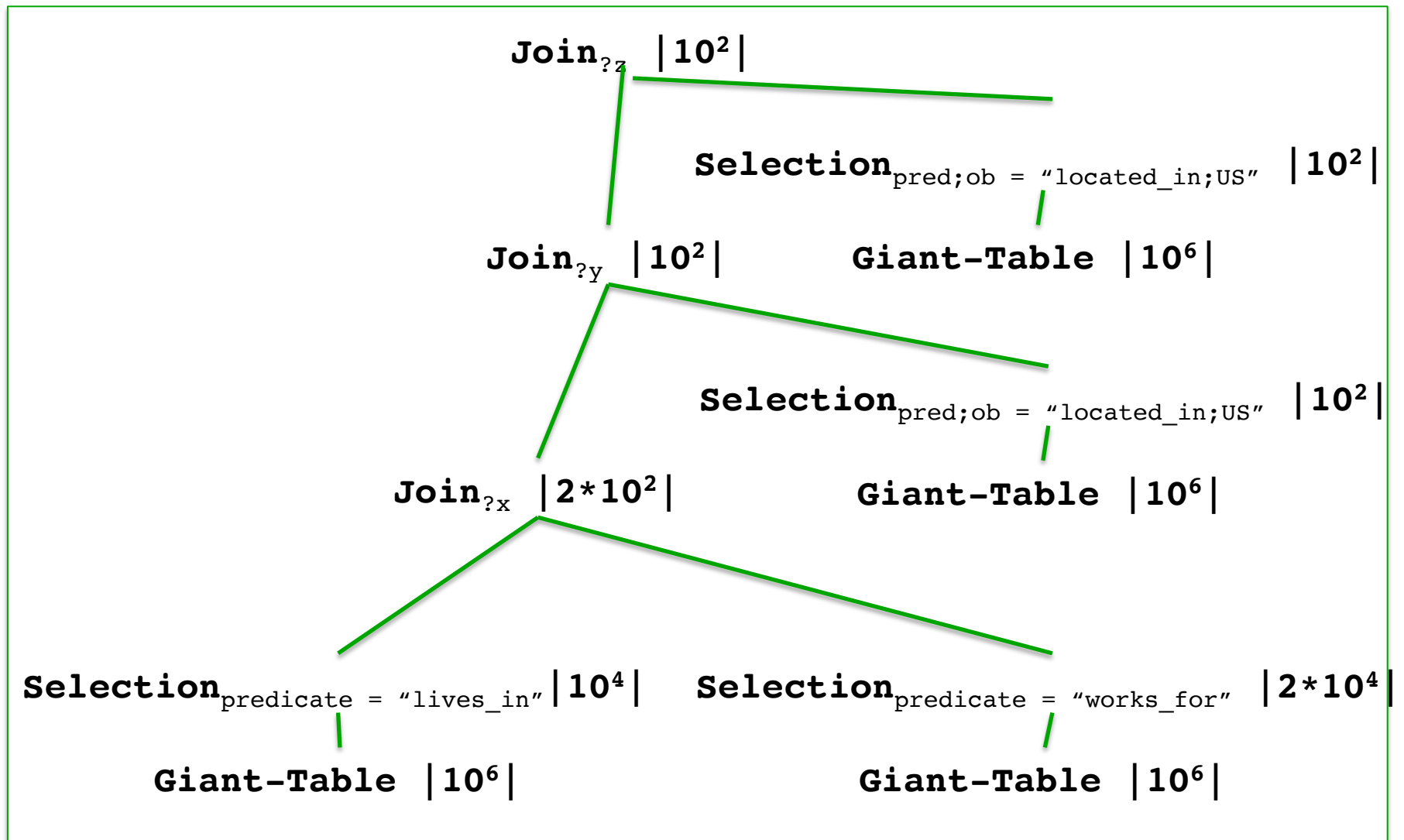
triple  
pattern  
selectivity

```
SELECT ?x where {  
    ?x, lives_in, ?y      . t1 10-2  
    ?x, works_for, ?z     . t2 2*10-2  
    ?y, isLocatedIn, "US" . t3 10-4  
    ?z, isLocatedIn, "US" } t4
```

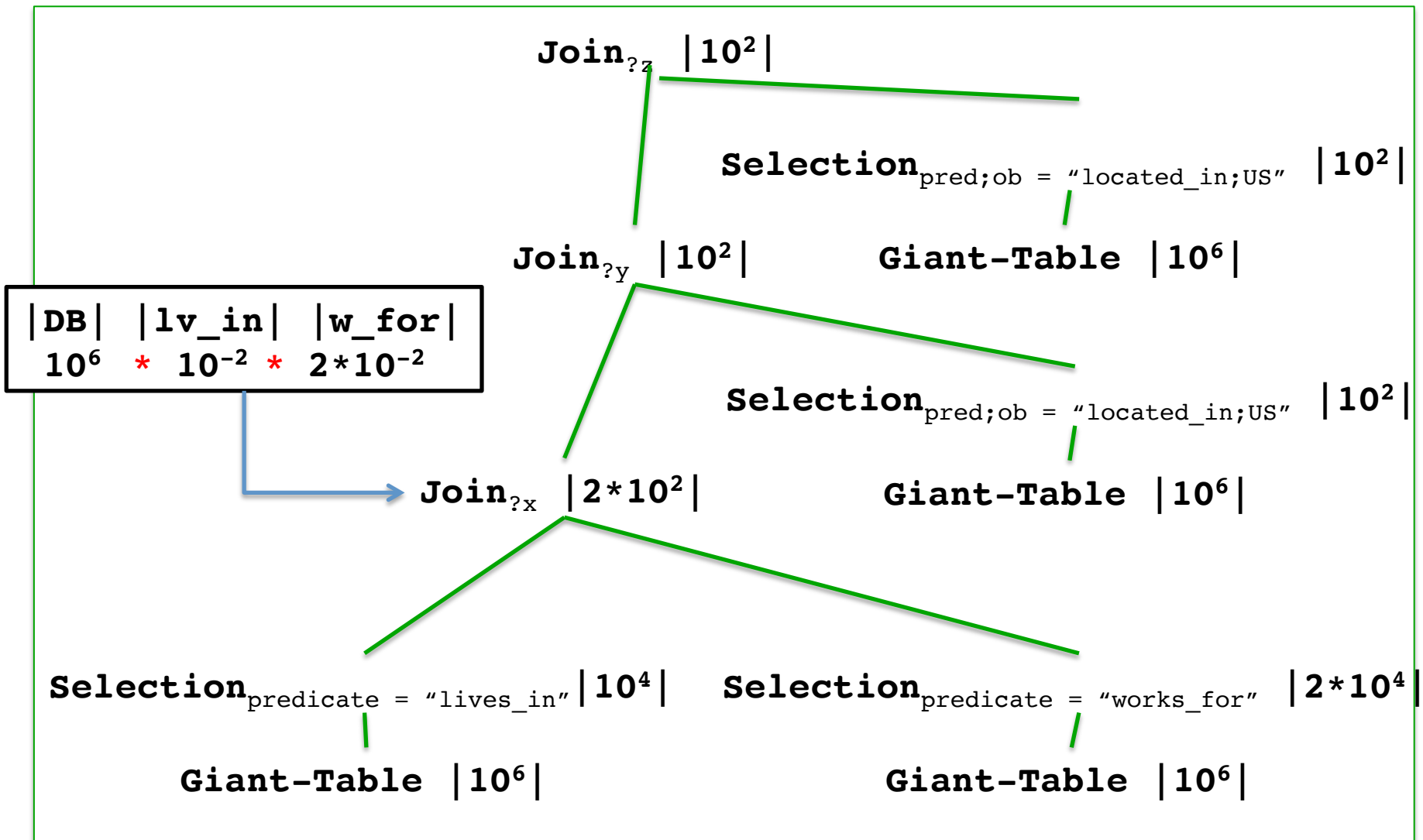
$(t_1 \text{ Join } t_2) \text{ Join } t_3 \text{ Join } t_4$

- Which plan is better ?
  - smaller the intermediary results --> faster the joins

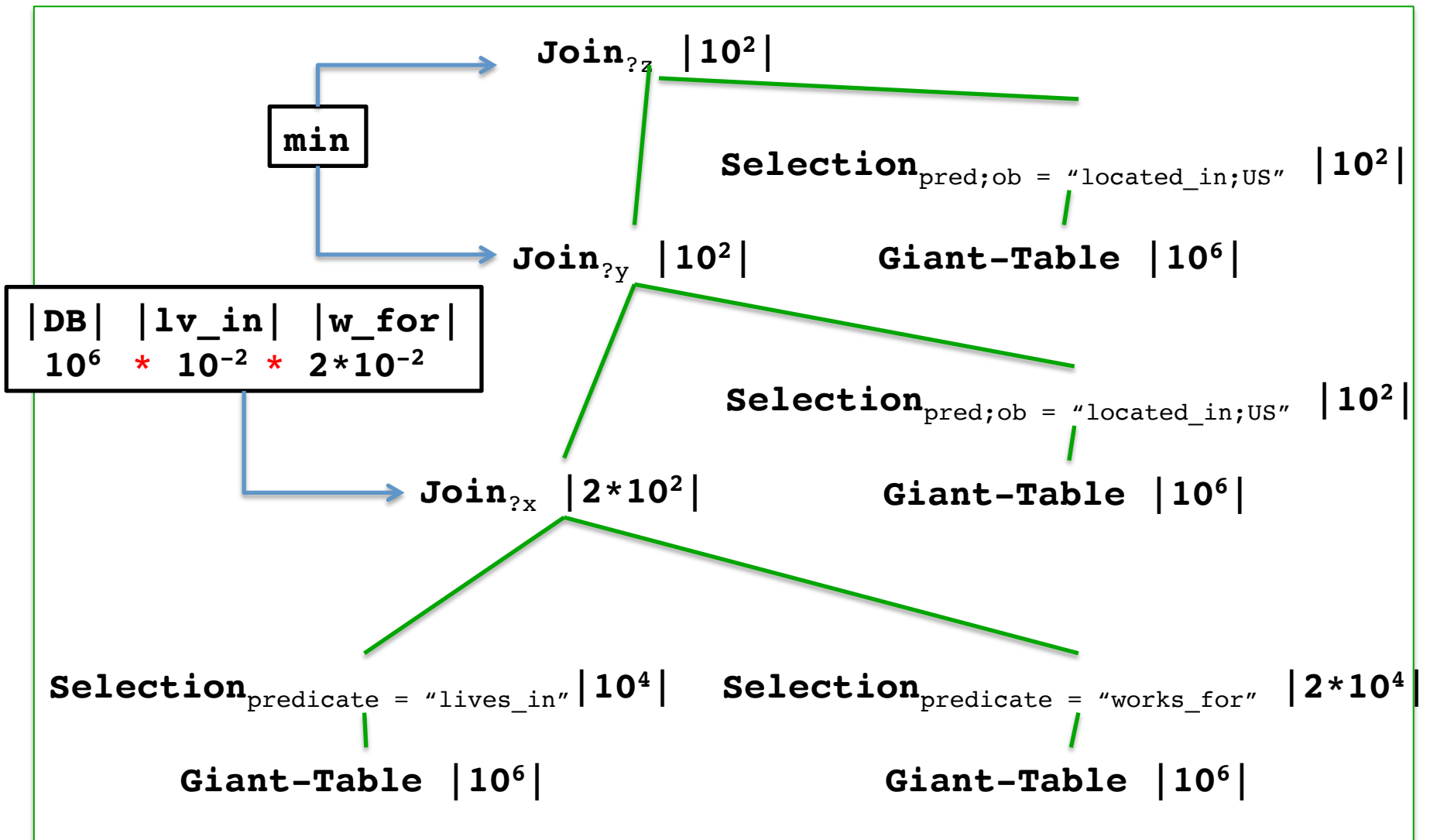
# Plan 1



# Plan 1



# Plan 1





# Back to our query



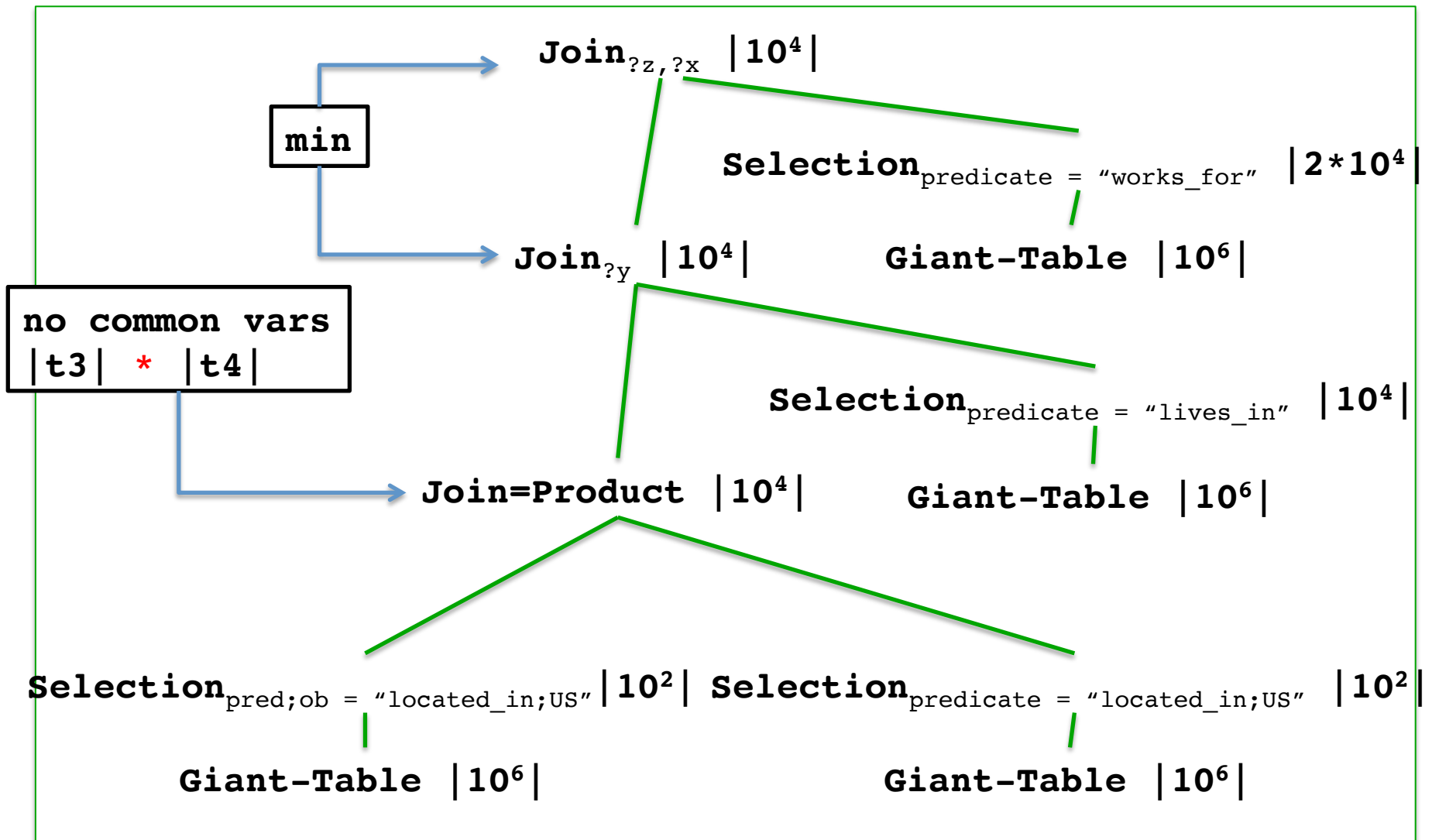
triple  
pattern  
selectivity

```
SELECT ?x where {  
    ?x, lives_in, ?y      . t1 10-2  
    ?x, works_for, ?z     . t2 2*10-2  
    ?y, isLocatedIn, "US" . t3 10-4  
    ?z, isLocatedIn, "US" } t4
```

$(t_4 \text{ Join } t_3) \text{ Join } t_2 \text{ Join } t_1$

- Which plan is better ?
  - smaller the intermediary results --> faster the joins

# Plan 2



# Back to our query



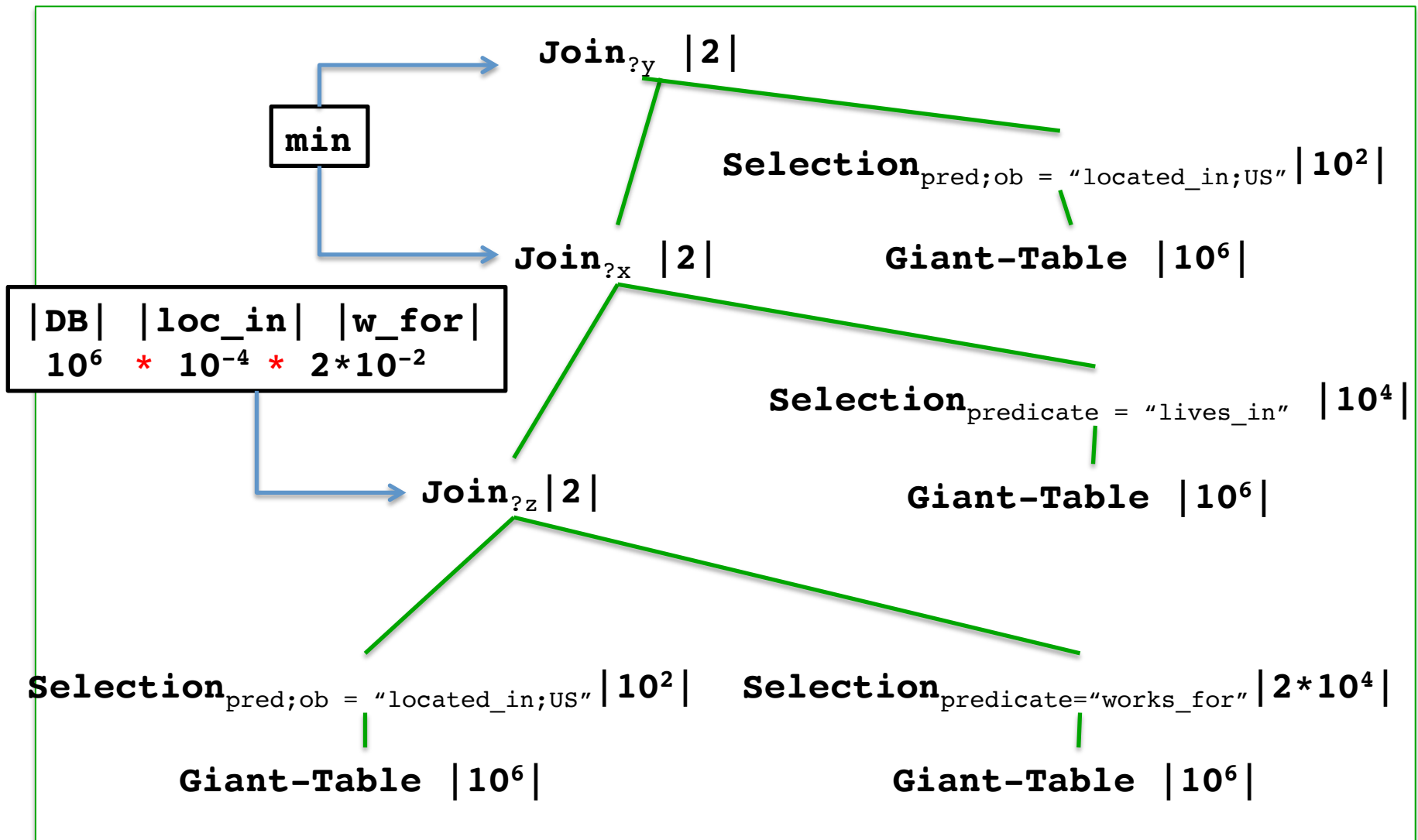
triple  
pattern  
selectivity

```
SELECT ?x where {  
    ?x, lives_in, ?y      . t1 10-2  
    ?x, works_for, ?z     . t2 2*10-2  
    ?y, isLocatedIn, "US" . t3 10-4  
    ?z, isLocatedIn, "US" } t4
```

$(t_4 \text{ Join } t_2) \text{ Join } t_1 \text{ Join } t_3$

- Which plan is better ?
  - smaller the intermediary results --> faster the joins

# Plan 3



# Back to our query



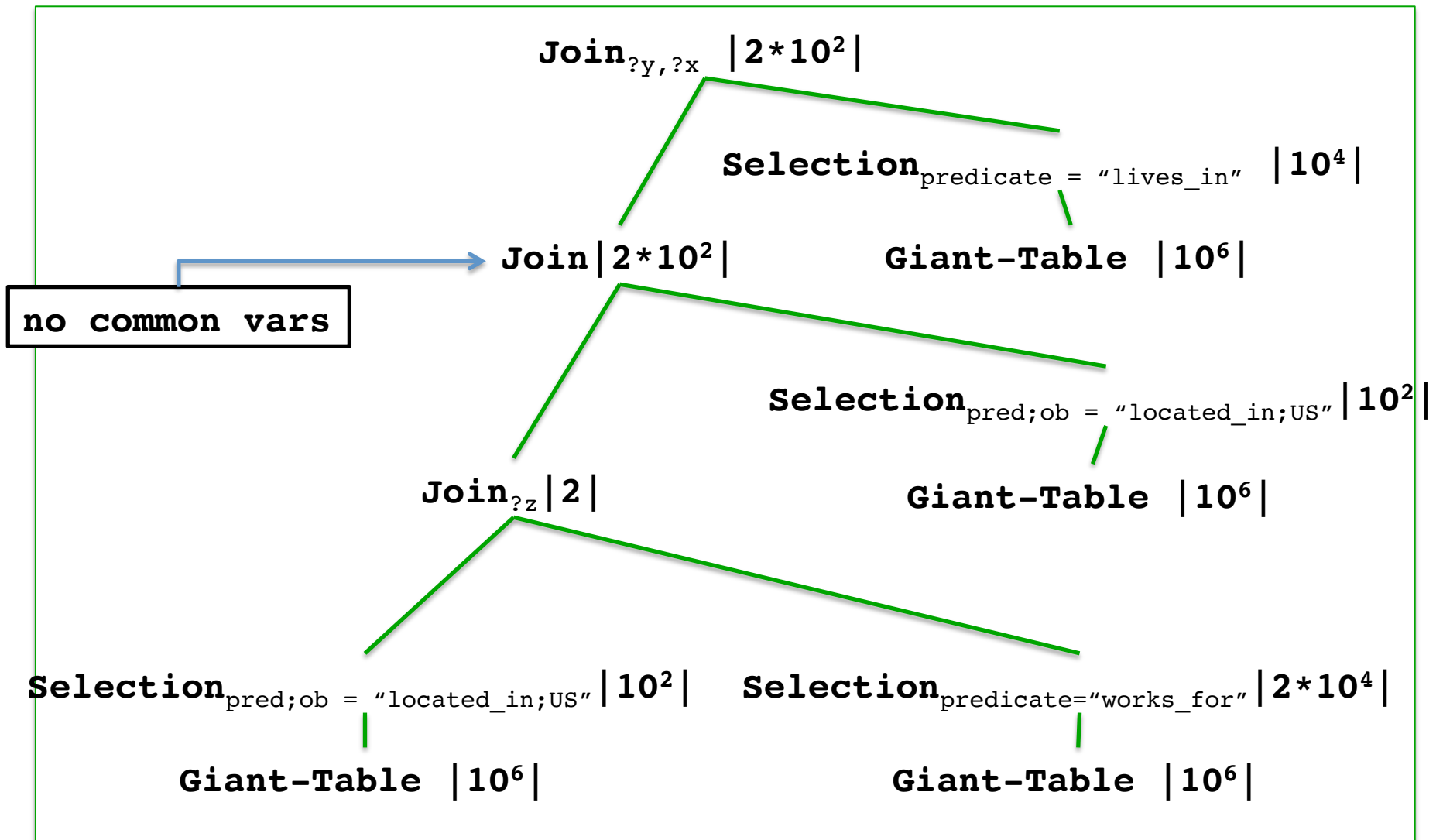
triple  
pattern  
selectivity

```
SELECT ?x where {  
    ?x, lives_in, ?y      . t1 10-2  
    ?x, works_for, ?z     . t2 2*10-2  
    ?y, isLocatedIn, "US" . t3 10-4  
    ?z, isLocatedIn, "US" } t4
```

$(t_4 \text{ Join } t_2) \text{ Join } t_3 \text{ Join } t_1$

- Which plan is better ?
  - smaller the intermediary results --> faster the joins

# Plan 4



# Summing Up

- Choosing the join ordering between triple patterns is crucial for efficient query evaluation
- This heavily relies on selectivity estimators for triple patterns (index scans) and joins

# RDF3X : What we skipped

- Compression Scheme
- Selectivity estimation of correlated properties via sampling
- Query optimization for very complex queries based on dynamic programming



# **COLUMN-STORES**

# Column-store

Relational database, storing relations as columns.

(C-Store, MonetDB and VectorWise, Ingres, IBM&MS Analytics)

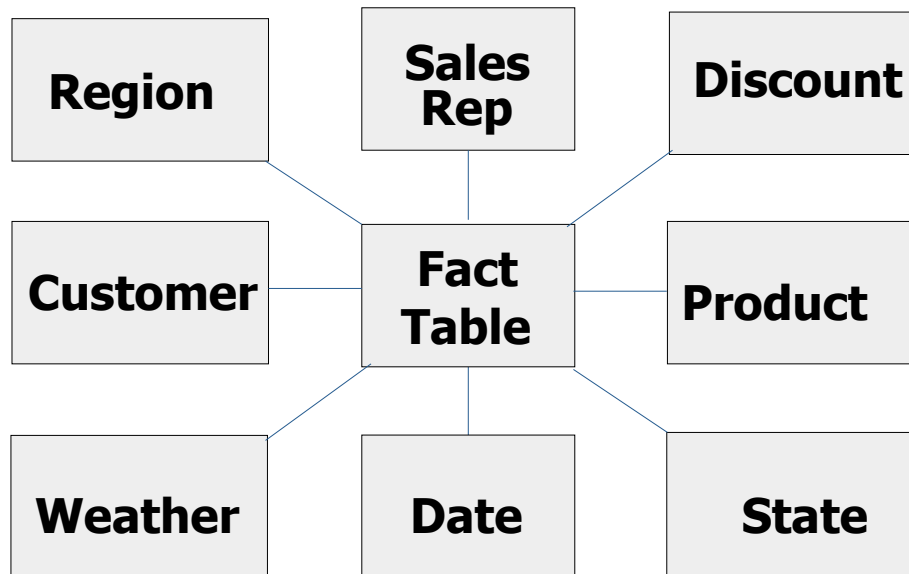
product	country	sales
car	US	40K
bike	US	7K
	...	



**col1** car@bike **col2** US@US **col3** 40K@7K .

# Today, any Relational Datawarehouse is a Column-Store

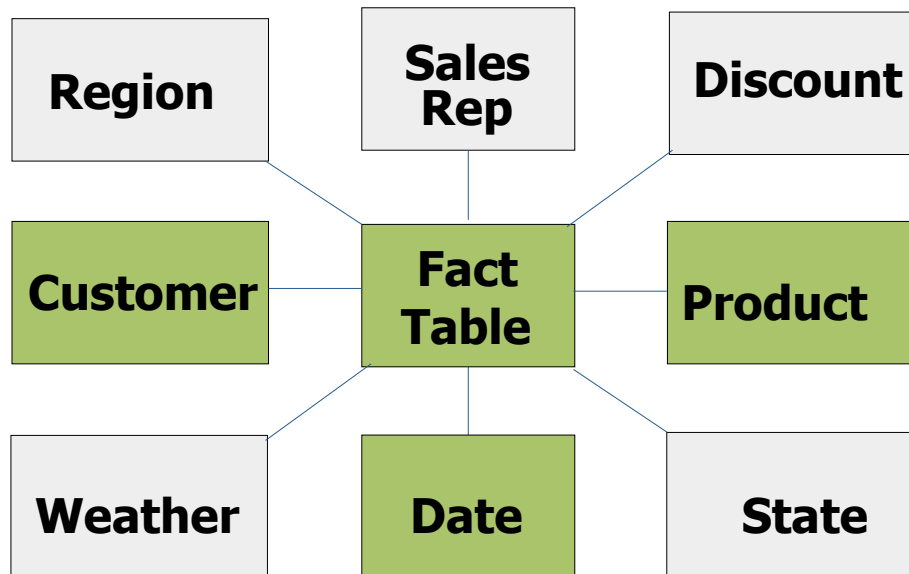
- «Do not call them column-stores, just modern DB systems.»  
Stratos Idreos



- Tables in a DW can have more than ten dimensions

# Today, any Relational Datawarehouse is a Column-Store

- «Do not call them column-stores, just modern DB systems.»  
Stratos Idreos



- Analytical query just uses a few of them

# Rows become crowded



**row1** id1@customer1@  
region1@sales\_  
rep1@discount1  
@product1  
@weather1@date  
1@state1...

**row2** id2@customer2@  
region2@sales\_  
rep2@discount2  
@product2  
@weather2@date  
2@state2

...

# Worst case : visit whole row

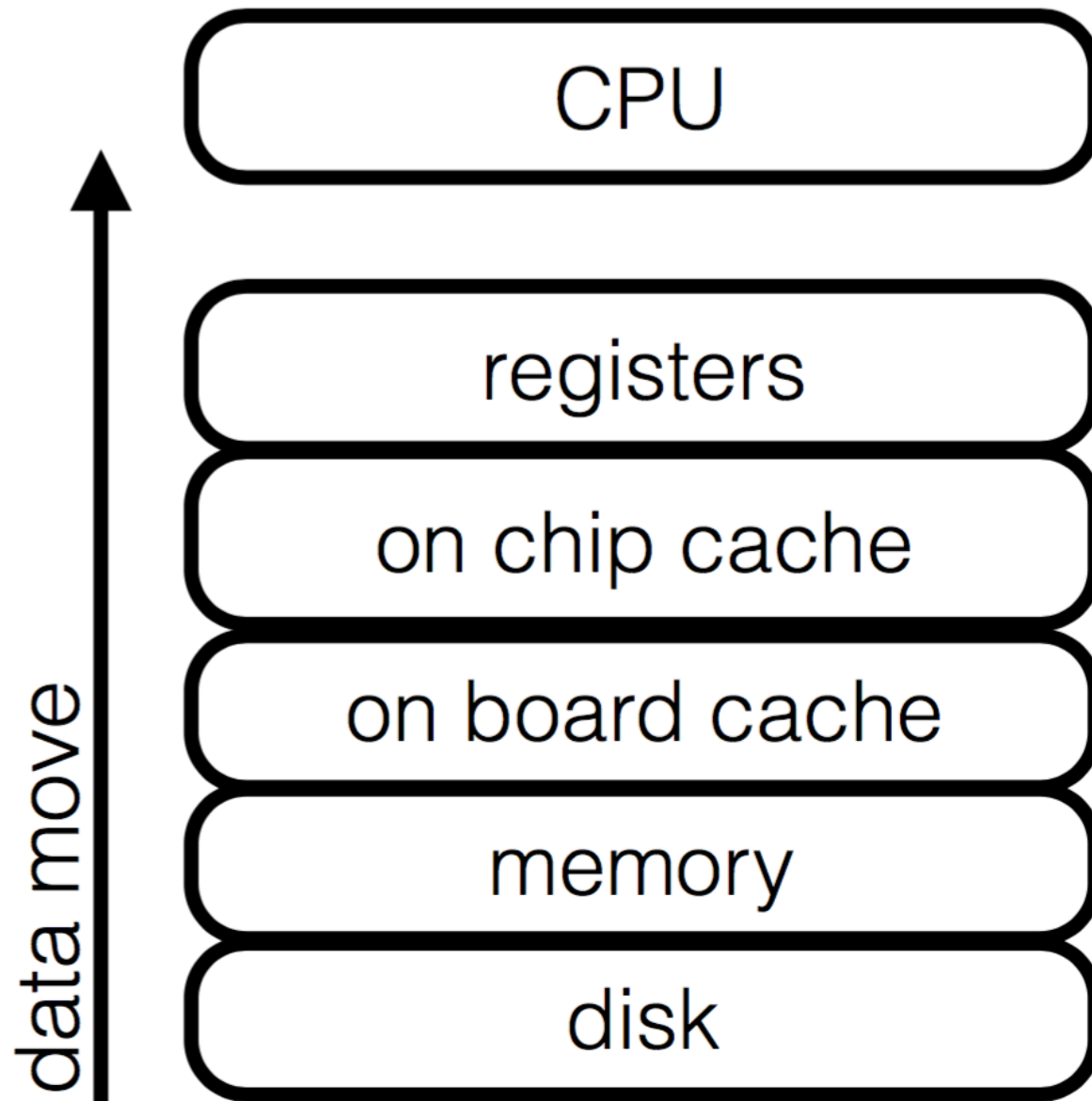


**row1** id1@customer1@  
region1@sales\_  
rep1@discount1  
@product1  
@weather1@date  
1@state1...

**row2** id2@customer2@  
region2@sales\_  
rep2@discount2  
@product2  
@weather2@date  
2@state2

...

- **Not only a reading-problem : entire sets of rows have to be loaded into memory from disk!**
  - This wastes bandwidth



# Rows are stored in pages

**row1** id1@customer1@  
region1@sales\_  
rep1@discount1  
@product1  
@weather1@date  
1@state1...

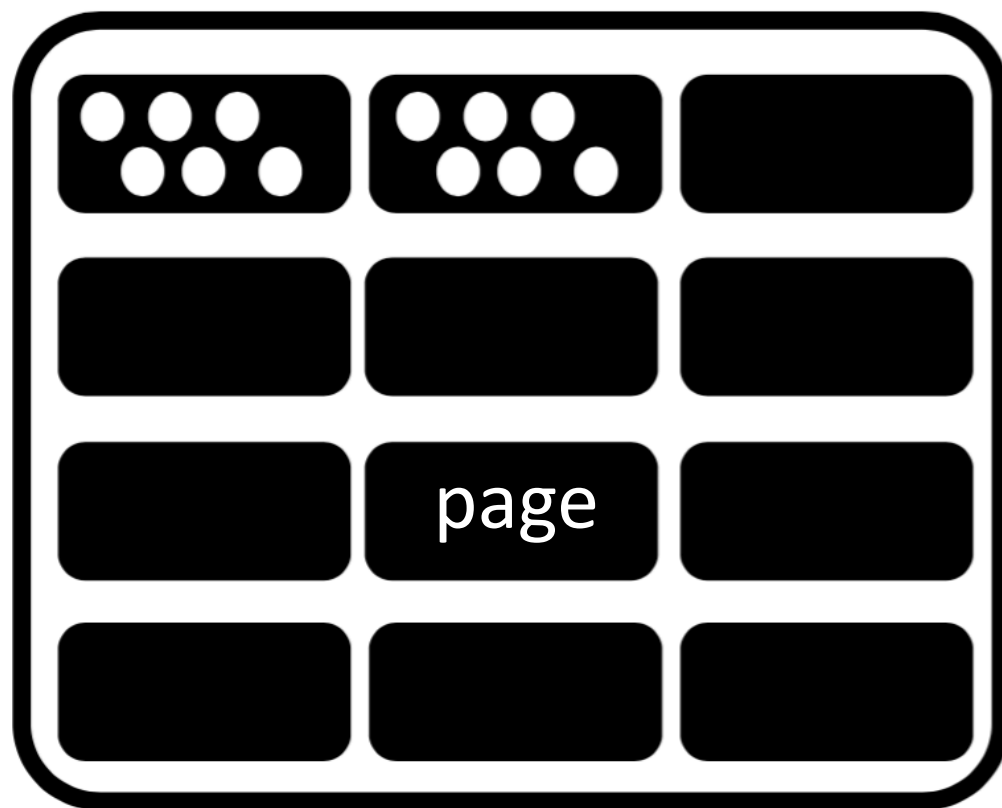
**row2** id2@customer2@  
region2@sales\_  
rep2@discount2  
@product2  
@weather2@date  
2@state2

...

page

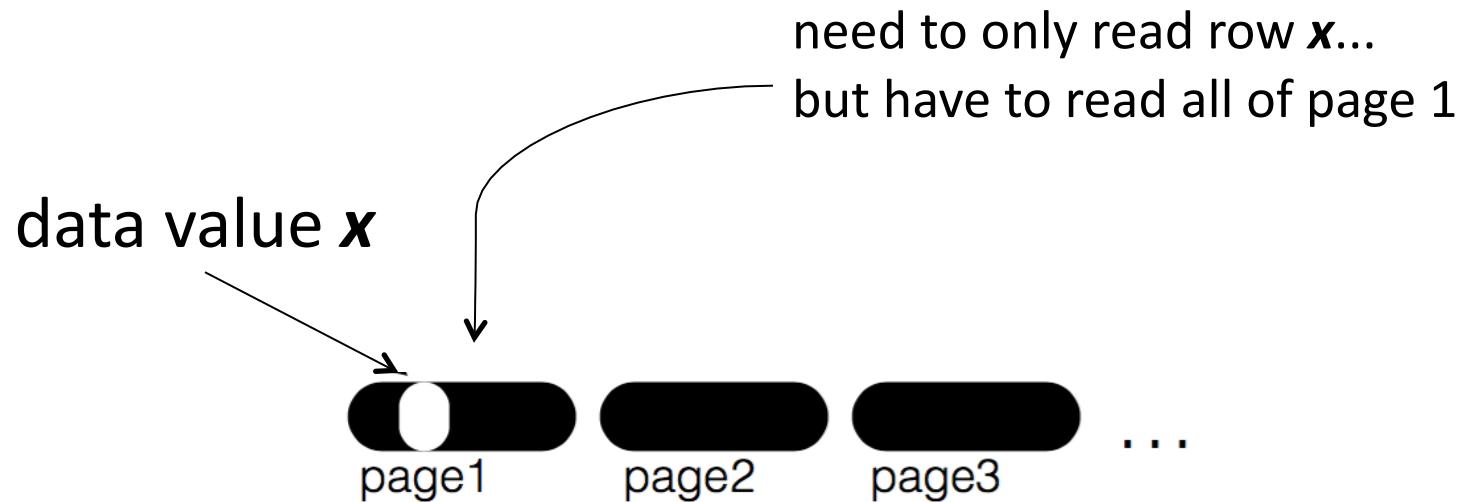


# Pages are stored within files



file

# Constraint : Pages are read as a whole!




# Better to « pay as you go »!



<b>col1</b>	<b>id1@id2</b>	<b>col2</b>	<b>customer1@customer2</b>	<b>col3</b>	region1@region2
<b>col4</b>	sales_rep1@sales_rep2	<b>col5</b>	discount1@discount2@	<b>col6</b>	<b>product1@product12</b>
<b>col7</b>	weather1@weather2	<b>col8</b>	<b>date1@date2</b>	<b>col9</b>	state1@state2

# Store columns on separate pages!



<b>col1</b>	<b>id1@id2</b>	<b>col2</b>	<b>customer1@customer2</b>	<b>col3</b>	region1@region2
<b>col4</b>	sales_rep1@sales_rep2	<b>col5</b>	discount1@discount2	<b>col6</b>	<b>product1@product12</b>
<b>col7</b>	weather1@weather2	<b>col8</b>	<b>date1@date2</b>	<b>col9</b>	state1@state2

col1 id1@id2

col2 custome  
r1@cust  
omer2

col3 region  
1@regi  
on2

col4 sales\_r  
ep1@sal  
es\_rep2

col5 discount  
1@discou  
nt2@

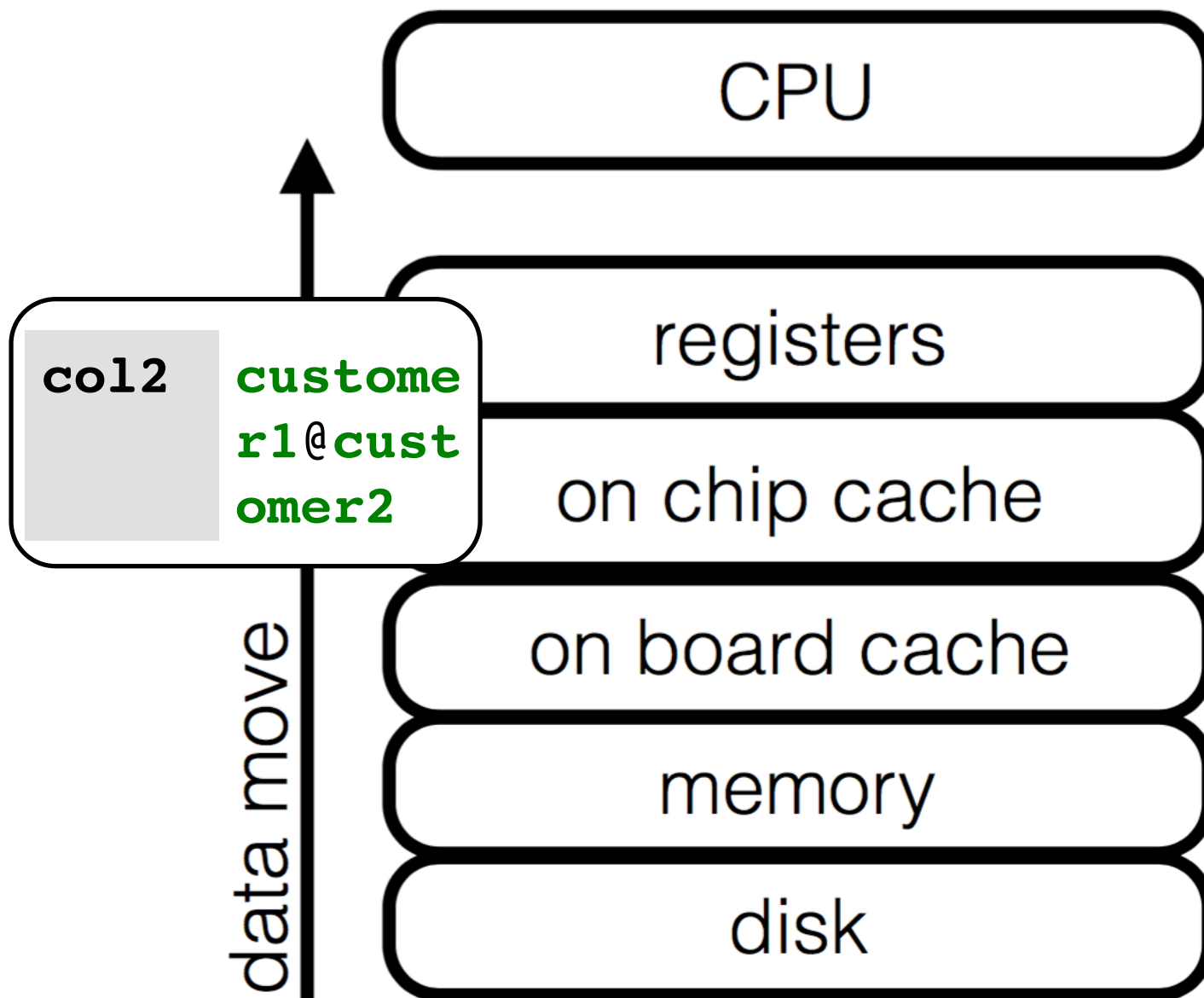
col6 produc  
t@1pro  
duct12

col7 weather  
1@weath  
er2

col8 date1@  
date2

col9 state1  
@state  
2

file



# Column Stores

The state of the art solution for

1. analytical
2. read-mostly queries
3. on wide-relations
4. supporting only batch-updates

## But wait...

- This does not mean that columnstores will automatically work well for RDF data..
  - for example, RDF property-tables are not wide, cluster-tables are wide however
- Let's see...



# Logical Model for RDF ColumnStores : Property-Tables

- A relational table for each single RDF property.

Giant-Table	subject	predicate	object
	alice	works_for	EDF
	alice	lives_in	Paris
	...		

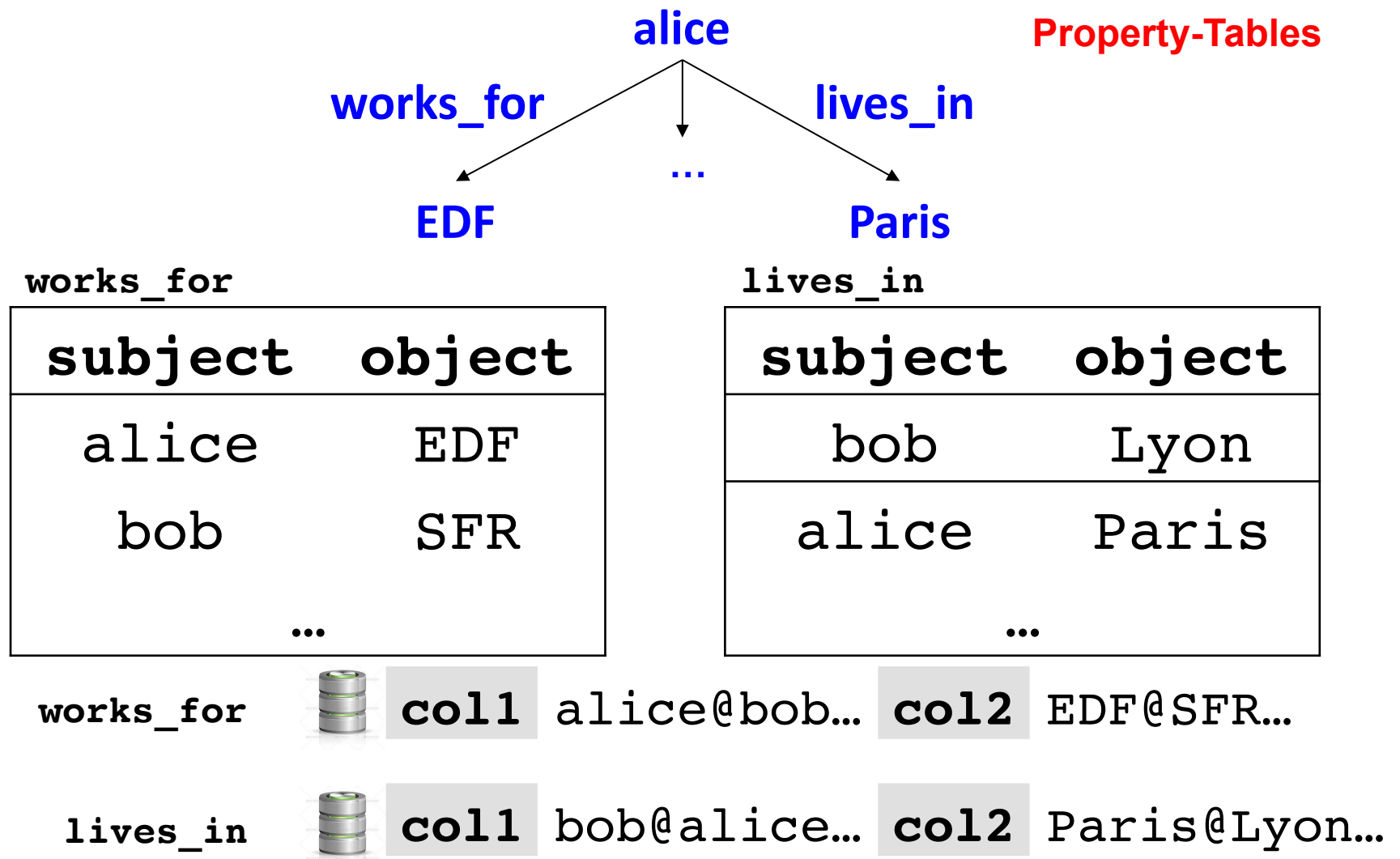
works\_for

subject	object
alice	EDF
...	

lives\_in

subject	object
alice	Paris
...	

# RDF in a Column-store (MonetDB, C-Store, Virtuoso)



# Property-Tables (Virtuso, C-Store)

1. Store triples in property tables
  2. Convert SPARQL to equivalent SQL
  3. *Magic* : the database will do the rest
- What is the consequence of employing column stores ?

# RDF Column Stores (Virtouso,C-Store)

## 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

## Neat advantage : column-projection

- Fast queries if only subject or object of a triple are accessed, not both

```
SQL> SELECT subject FROM lives_in
```

`lives_in`  **col1** alice@... **col2** Paris@...

- All results can be found in the first column

# Advantages of using column stores

- Allows for a very compact representation
- Exploits merge joins
- A column contains “homogeneous” values, where many values repeat and thus compression is more effective



**col1**

car@bike

**col2**

US@US

**col3**

40K@7K .

# Disadvantages of using column stores

- Need to recombine columns if subject and object are accessed



**col1** car@bike **col2** US@US **col3** 40K@7K .

- Inefficient for triple patterns with predicate variable  
<alice ?p bob>
- Big question : when to reconstruct a tuple ?  
(alap)

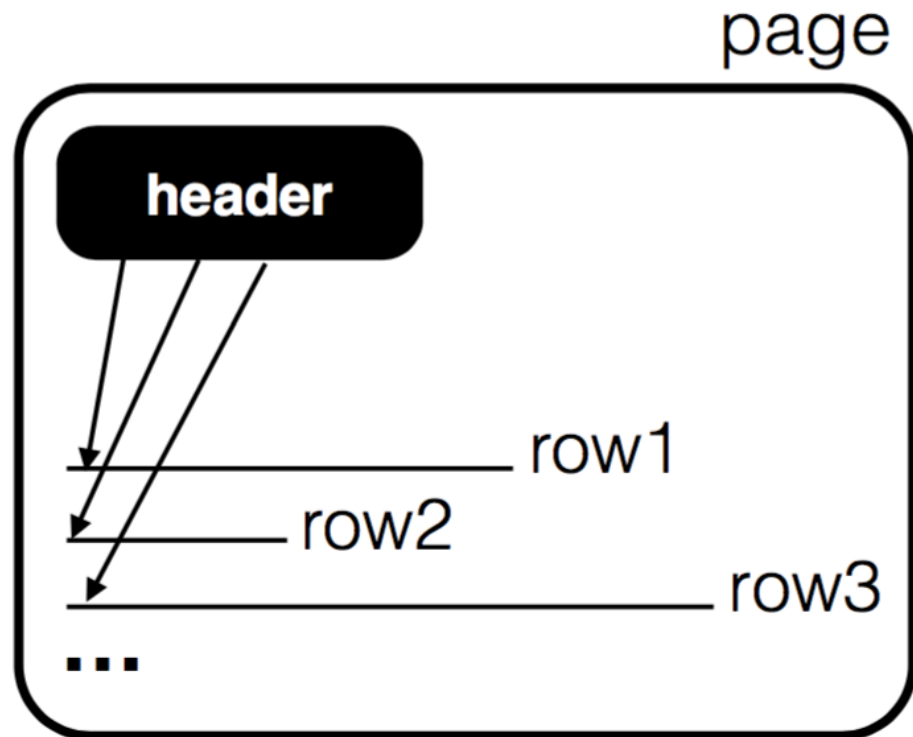
# Advantages of Column Stores

1. Tuple Headers Stored Separately
2. Optimizations for fixed-length tuples
3. Column-oriented data compression
4. Carefully optimized merge-join code

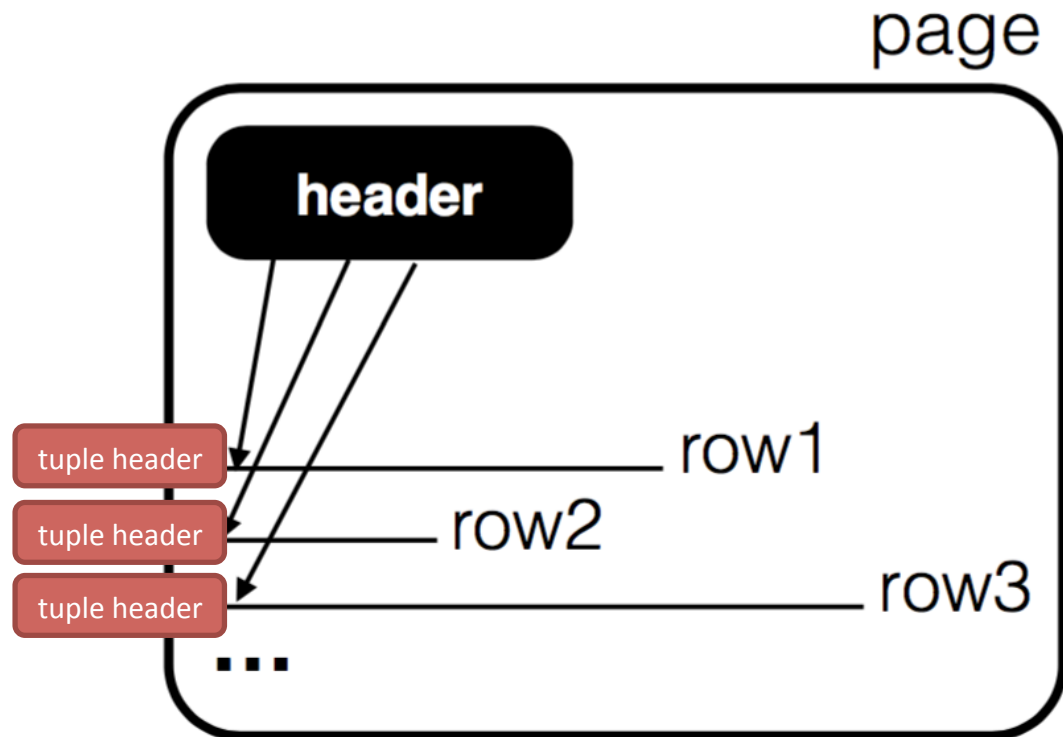


**TUPLE HEADERS**

# Page headers in Rowstores



# Tuple headers in Rowstores



| tuple header |    >>    | row<sub>of size 2</sub> |

# Tuple headers

- Metadata **at the beginning of the row**
  - Insert transaction timestamp
  - number of attributes in tuple (useless for RDF)
  - NULL flags
- Postgres: 27 byte tuple header + 8 bytes two-column tables

| header |      >>      | row<sub>of size 2</sub> |

# Postgres Tuple Header

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	int16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data
null bitmap		optional	
object id		optional	

# Postgres Tuple Header

- Some information on visibility of a tuple for current transaction snapshot or newer version (needed for snapshot isolation algorithm)
  - t\_xmin TransactionId 4 bytes insert XID stamp
  - t\_xmax TransactionId 4 bytes delete XID stamp
  - t\_cid CommandId 4 bytes insert CID stamp (actual a UNION struct)
  - t\_ctid ItemPointerData 6 bytes current TID of this or newer row version
- How long is this row? Is it variable length? Does it have NULLs?
  - t\_natts int16 2 bytes number of attributes
  - t\_infomask uint16 2 bytes various flag bits
    - e.g. HAS\_NULL | HASVARWIDTH | HASOID | locks(!)
- t\_hoff uint8 1 byte /\* sizeof header incl. bitmap, padding \*/
- null bitmap (optional)
- object id(optional)

# Tuple headers in Columnstores

- Puts header information in separate columns and can selectively ignore it
  - #attributes is always two
  - in some cases, NULL values are totally avoided
- Column-store effective tuple width : ~8 bytes
- Reading a tuple takes 4-5 times less time than Postgres (27 + 8bytes), so does a simple table scan.

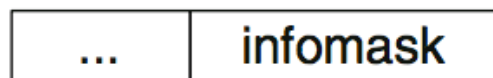
# Rowstore vs Columnstore Headers

**Postgres Header (27 bytes)**

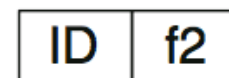
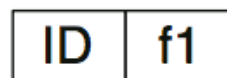


**Tuple Data**

**Column Store Header**



**Tuple Data**





# **OPTIMIZATIONS FOR FIXED-LENGTH TUPLES**

# Optimizations for variable-length tuples

- 1 attribute **variable length** => the whole tuple is
- Common case: row-stores designed for this

```
row1 idcustomer1@firstname1@lastname1@...
```

- Tuples located/iterated via pointers in the page header (instead of address offset calculation)

# Optimizations for fixed-length tuples

- In columnstores, with a dictionary encoding, fixed length val. are stored/accessed as **arrays**

<b>A</b>	a0
	a1
	a2
	a3
	..

Positional lookup

$$a(i) = A + i * \text{width}(A)$$

# Optimizations for fixed-length tuples

- Every RDF property table has two columns

works_for	
subject	object
alice	EDF
...	

- Store each on disk in a page (eg, 64K)
  - in SW-Store, they found this suboptimal for queries accessing both values

# Hybrid Storage (SW-Store)

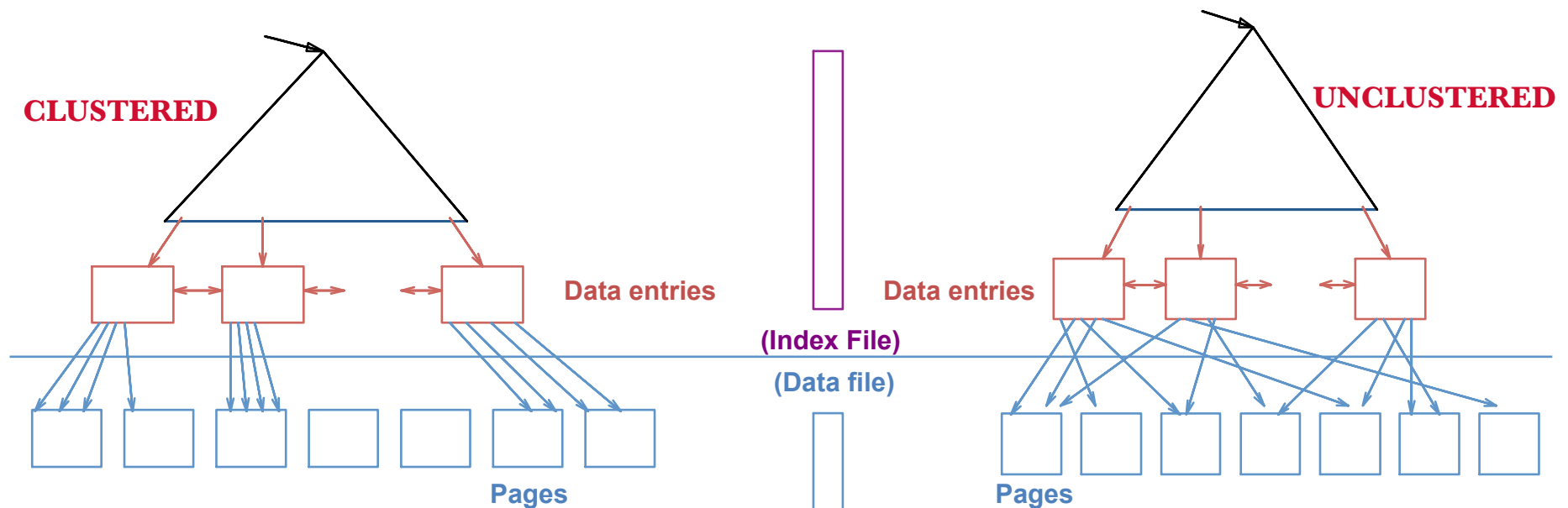
- A page slot contains data from both columns

<b>P</b>	s0	o0
	s1	o1
	s2	o2
	s3	o3
	...	...

$$o(i) = P + i * (\text{width}(S) + \text{width}(O)) + \text{width}(S)$$

# +Index on hybrid Column (SW-Store)

- clustered B+ tree index on subject
- unclustered B+ tree index on object.



Clustered/unclustered

- Clustered = records close in index are close in data
- Unclustered = records close in index may be far in data

# More optimizations : Id-Table

- Maintain a **single-column** table that contains all triple subjects (much better if ordered)

Id-Table	$id_{\text{Obama}}$
	$id_{\text{Alice}}$
	$id_{\text{Paris}}$
	..

# Id-Table

- If property **P** is NOT multivalued (eg. birthday) we can avoid to store subject id

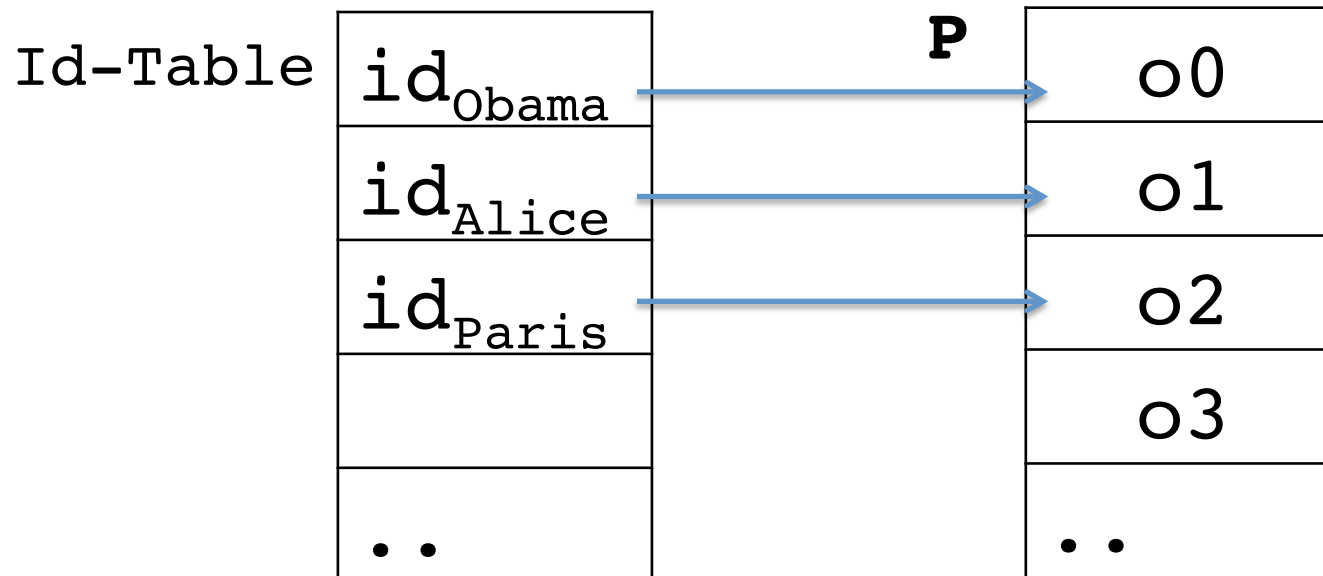
Id-Table	$id_{Obama}$
	$id_{Alice}$
	$id_{Paris}$
	• •

<b>P</b>	o0
	o1
	o2
	o3
	• •



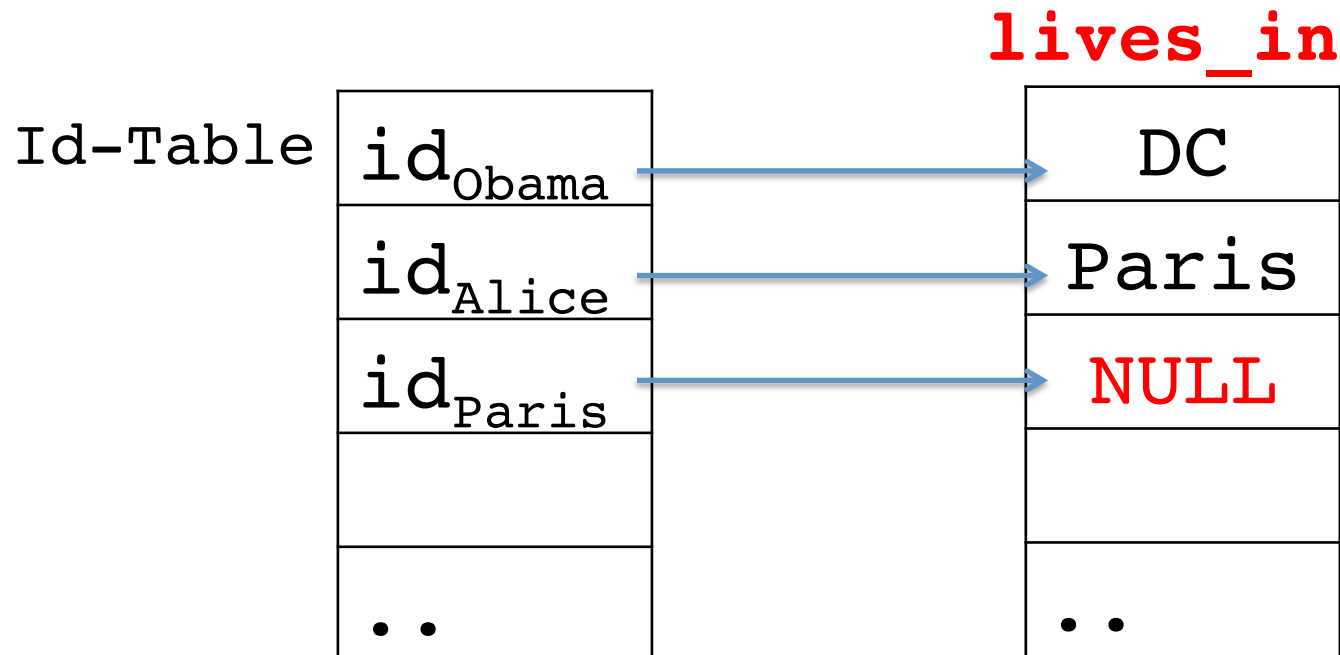
# Id-Table

- If property **P** is NOT multivalued (eg. birthday) we can avoid to store subject id



# Nulls are back!

- If property **P** is NOT multivalued (eg. birthday) we can avoid to store subject id



# Id-Table : Dealing with Nulls

- Goal : remove nulls from the column
- There is not a single optimal solution
- The point is to find a tradeoff between the **size** and **manegeability** of the compressed column
  - this depends only on the **data-distribution**

# It's matter of « density »

D. Abadi [Column-Stores For Wide and Sparse Data](#)

Dense



Not dense nor sparse



Sparse



## Case 1: «dense» data and long null sequences

**Title**

t1
t2
t3
t4
NULL

**Language**

NULL
FR
EN
NULL
NULL

## Case 1: Low overhead for «dense» data

- Store indexes of non-null values

**Title**

t1
t2
t3
t4
NULL

**Title**

t1
t2
t3
t4

**Range [ 1-4 ]**

## Case 1: Low overhead for «dense» data

- Store indexes of non-null elements

**Language**

NULL
FR
EN
NULL
NULL

**Language**

FR
EN

**Range[2-3]**

## Case 1: Low overhead for «dense» data

- Store indexes of non-null elements
- If data is dense, then

$$|\text{range information}| \ll |\text{nulls}|$$



## Case 2: data not «dense» nor «sparse»

- Store a bitmap index (0=NULL)

**Copyright**

2001
NULL
1985
NULL
1995
2004

## Case 2: data not «dense» nor «sparse»

- Store a bitmap index (0=NULL)

Copyright Bit:101011

2001
NULL
1985
NULL
1995
2004

## Case 2: data not «dense» nor «sparse»

- Store a bitmap index (0=NULL)

Copyright Bit:101011

- Overhead = 1bit per value

2001
NULL
1985
NULL
1995
2004

## Case 3 : sparse data

- Store the list of non-null ids

**Author**

Hugo
NULL
NULL
NULL
NULL

**Artist**

NULL
Dylan
NULL
NULL
NULL

## Case 3 : sparse data

- Store the list of non-null ids

**Author**

Hugo
NULL
NULL
NULL
NULL

**Author**

List:1

Hugo
------

## Case 3 : sparse data

- Store the list of non-null ids

**Artist**

NULL
Dylan
NULL
NULL
NULL

**Artist**

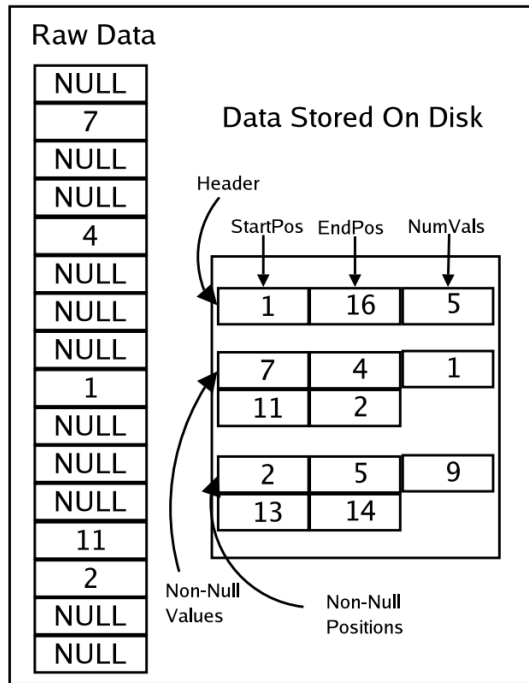
List:2

Dylan

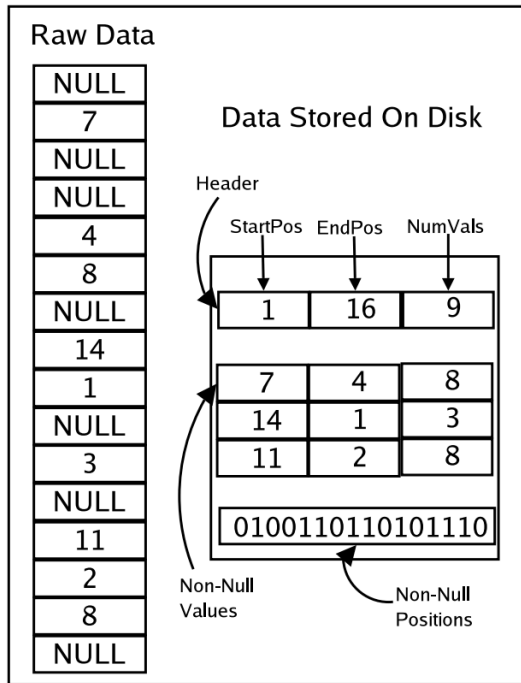
## Case 3 : sparse data

- Store the list of non-null ids
- If data is sparse

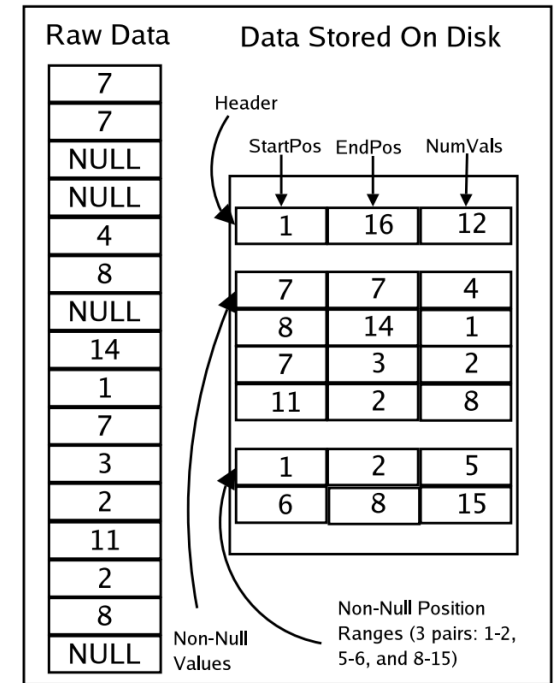
$$| \text{List} | \ll | \text{nulls} |$$



(a)



(b)



(c)

**Figure 1: Positions represented using a list (a), a bit-string (b), and as ranges (c) for sparse columns**



# Back to Query Evaluation

SELECT ?y ?z where {

    ?x, author, Hugo

    ?x, title, ?y

    ?x copyright ?z

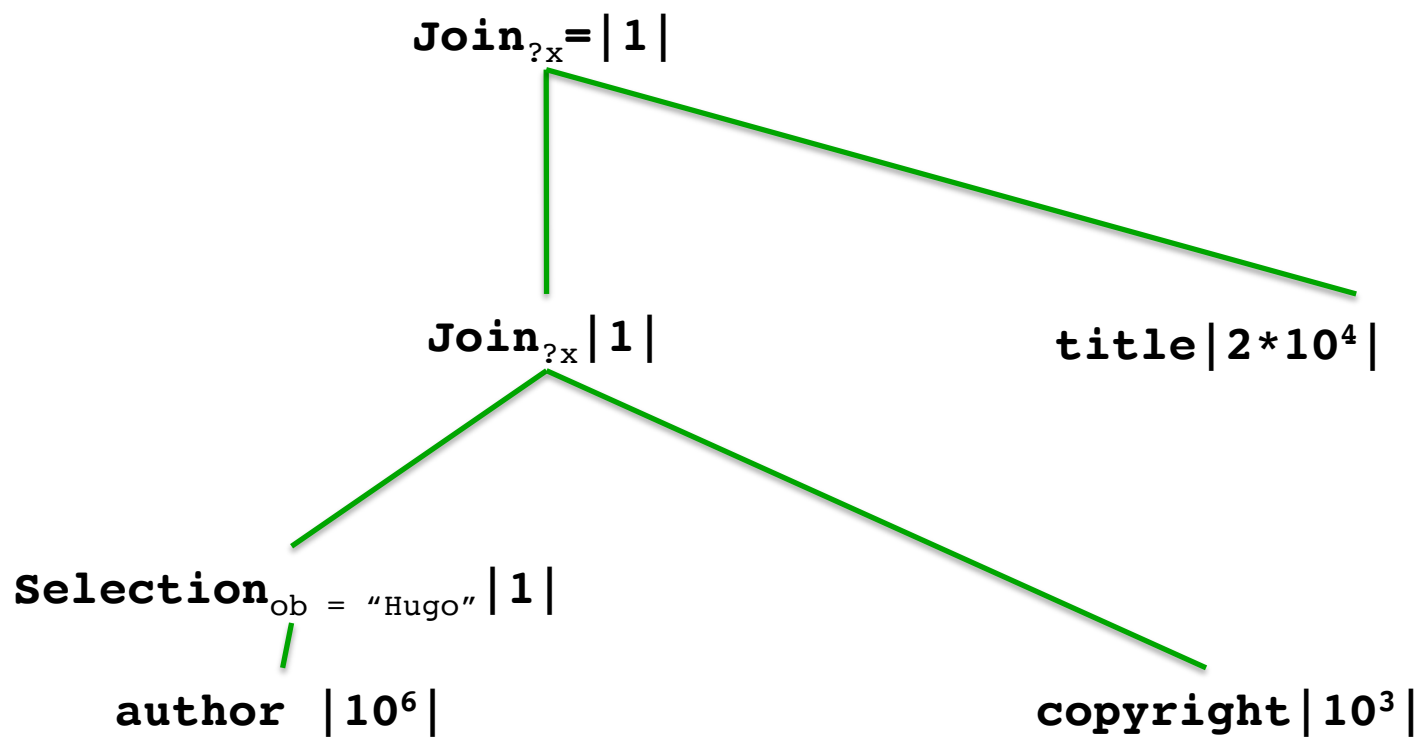
}

- $t_1$   $10^{-6}$
- $t_2$   $2 * 10^{-2}$
- $t_3$   $10^{-3}$

triple  
pattern  
selectivity

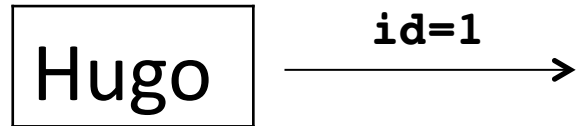
$(t_1 \text{ Join } t_3) \text{ Join } t_2$

# Plan



**Author**

**List:1**



**Author**

List:1

Hugo

id=1

**Copyright**

Bit:110011

2001

1985

NULL

NULL

1995

2004

**Author**

List:1

Hugo

**Copyright**

Bit:110011

2001
1985
NULL
NULL
1995
2004

**Title**

Range[1-5]

t1
t2
t3
t4

id=1

id=1

# Query

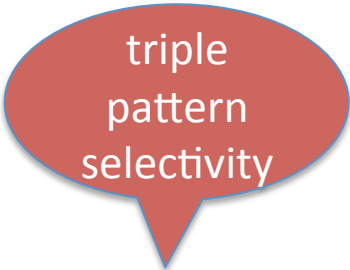
SELECT ?y ?z where {

?x, author, Hugo

?x, title, ?y

?x copyright ?z

}

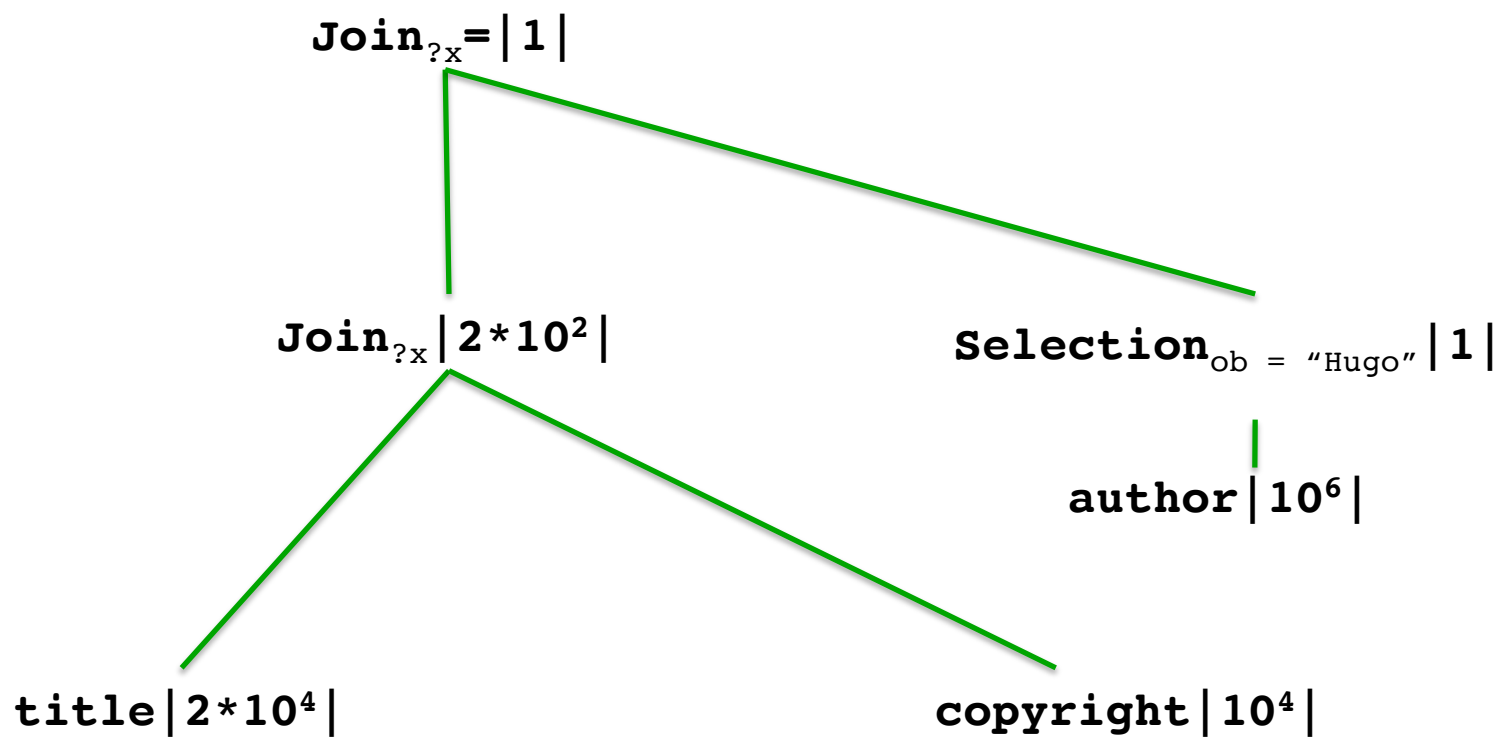


triple  
pattern  
selectivity

- $t_1$   $10^{-6}$
- $t_2$   $2 * 10^{-2}$
- $t_3$   $10^{-3}$

$(t_2 \text{ Join } t_3) \text{ Join } t_1$

# Plan



**Title**  
Range[1-7]

t1
t2
t3
t4
t5
t6
t7

**Copyright**  
Bit:110011

2001
1985
NULL
NULL
1995
2004

**Author**  
List:1

Hugo
------

↙ ↘  
id=  
1,2,3,  
4,5,6,  
7,...

↙ ↘  
id=  
1,~~2,3,~~  
4,5,6,  
7,...

↘  
id=1



# **COLUMN-ORIENTED DATA COMPRESSION**

# Column-oriented data compression

- Since each attribute is stored separately (even within a slot), it can be compressed separately using the best algorithm
  - for example, the subject ID column, a monotonically increasing array of integers
  - data from the same domain tend to show locality
- It is often possible also to operate directly on compressed representations
  - Bandwidth requirements are reduced when transferring compressed data

**SO FAR SO GOOD. NOW WHO  
WINS?**

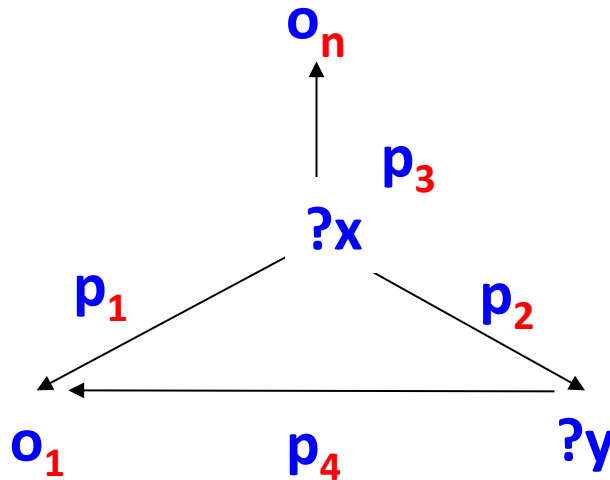
# There is no single winner

	<i>RDF-3x</i>	<i>VOS [6.1]</i>	<i>VOS [7.1]</i>	<i>MonetDB</i>	<i>4Store</i>
% of queries for which tested system is fastest	20.9%	0.0%	22.6%	56.5%	0.0%
Total workload execution time (hours)	27.1	20.9	20.8	38.6	72.2
Mean (per query) execution time (seconds)	7.8	6.0	6.0	11.1	20.8

Table 1.1: Summary of results over WatDiv 100M RDF triples, 12500 SPARQL queries.

# Mini-Projet : Moteur RDF (Partie 1)

- Implémenter un moteur de requêtes pour données RDF utilisant l'une des approches vues en cours :
  - Hexastore, Columnstore, Graphstore



# Mini-Projet : Moteur RDF (Partie 2)

- Évaluer les performances du système réalisé
  - comparer avec 1 autre système + Jena
- Fournir un système fonctionnel à ses collègues (dans les délais) fait partie de l'évaluation