RDF Data Management & SPARQL Query Processing

Cours développé par

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

Organisation

- 29 Octobre : RDF Stores 2CM + 1TP (début mini-projet)
- 19 Novembre: 1TP
- 26 Novembre: 2TP
- 3 Décembre: 2TP
- 10 Décembre: 2TP

Plan

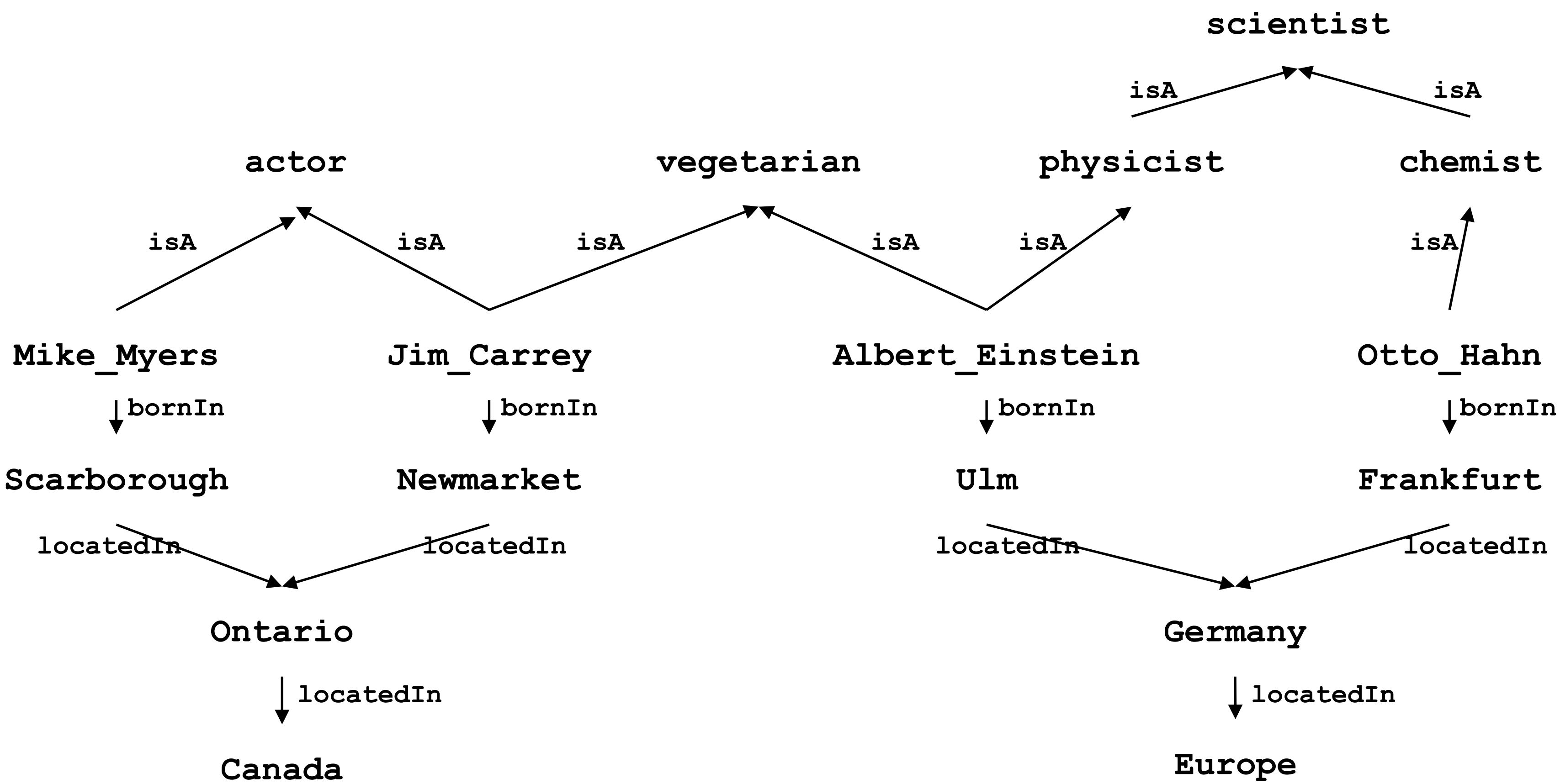
- **RDF and SPARQL**
- **RDF Row-stores**
- **RDF Column-stores**
- **RDF Graph-stores**

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 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 more than a Graph Database

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

property are also reources

```
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>
```

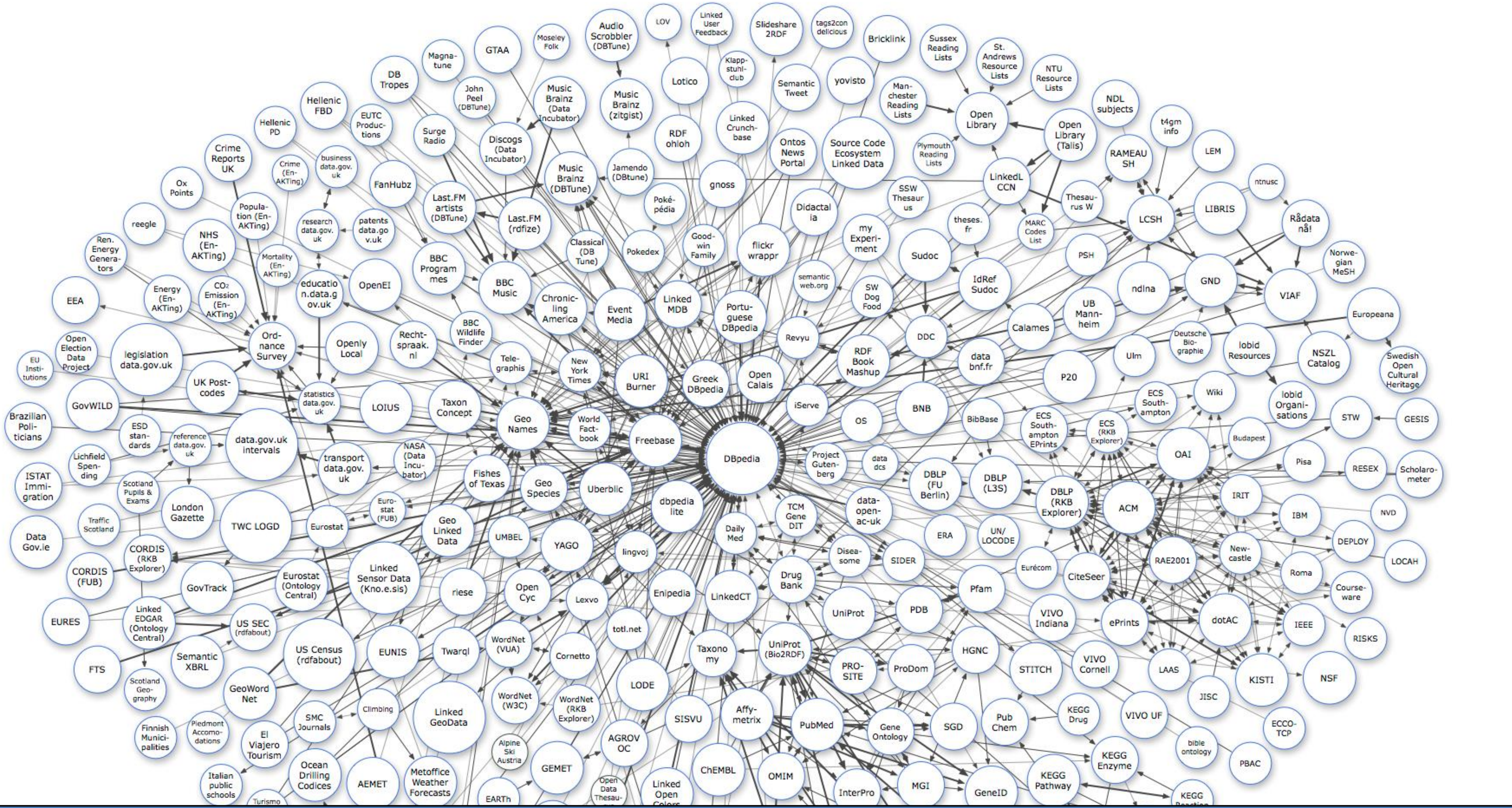
property are also resources

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

properties can be queried

Sources:
linkeddata.org
wikipedia.org

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**

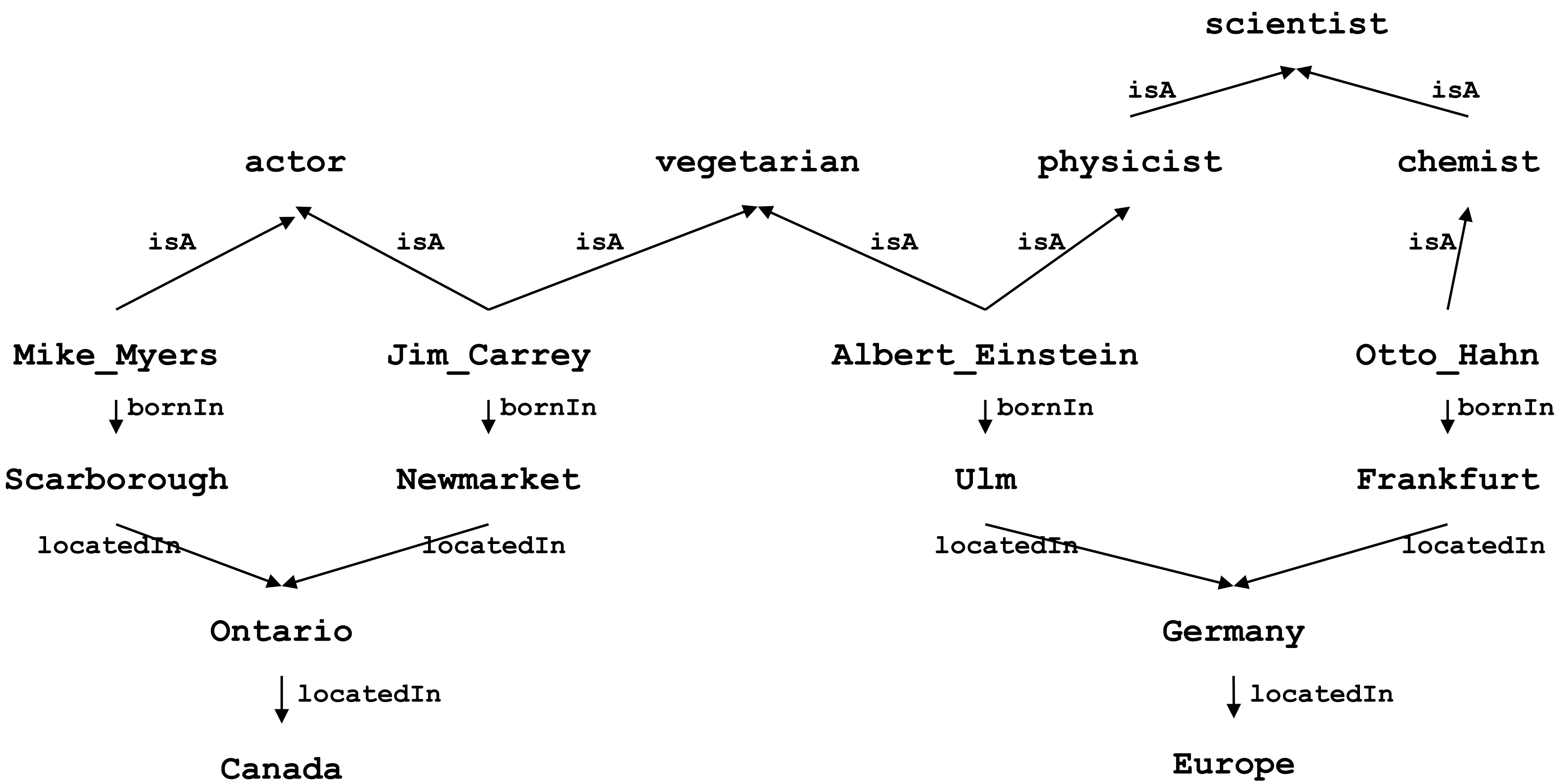
```
?x  wrote_book  ?y
```

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

SPARQL – Example

Example query:

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



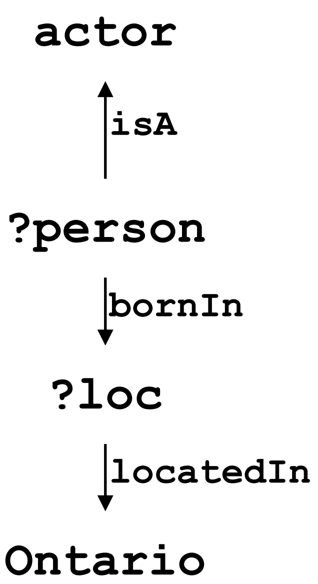
SPARQL – Example

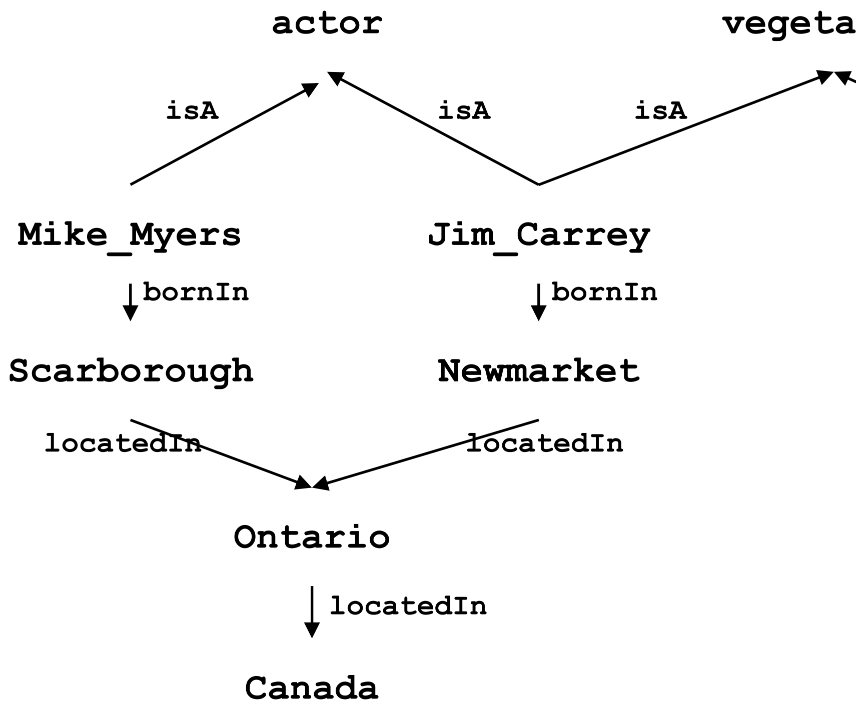
<u>Example query:</u>

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:





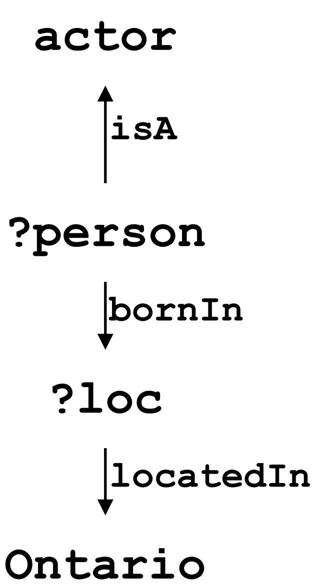
SPARQL – Example

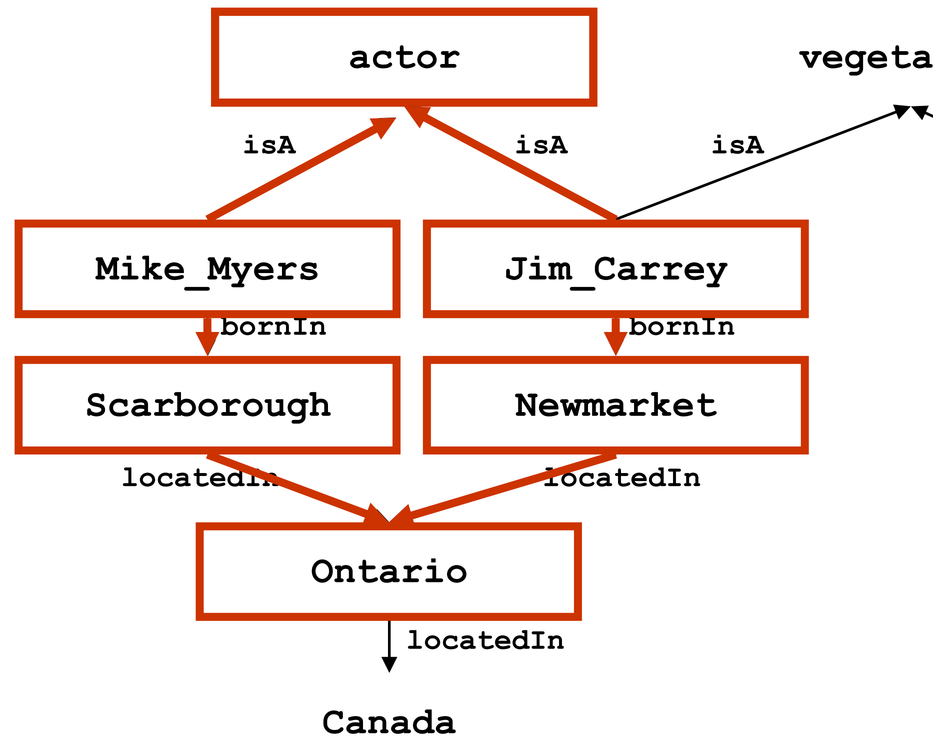
Example query:

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

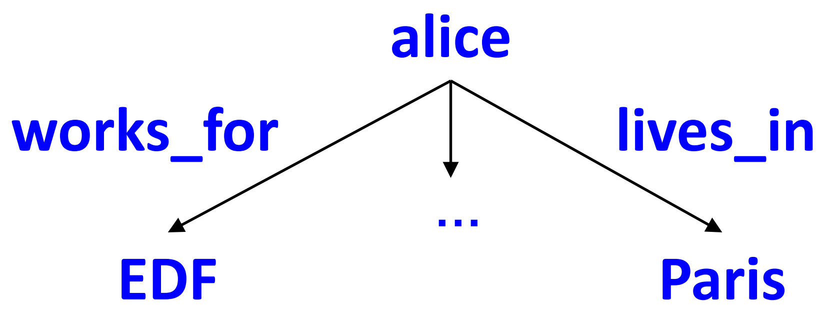
Classic relational database, storing relations by <u>rows</u>.

(Postgres, Oracle, DB2, MySQL, …)

product	**country**	**sales**
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@works_for@EDF	**row2**	alice@livesIn@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

subject	**predicate**	**object**
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
WHEREpredicate = "lives_in"
  ANDobject != "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
```

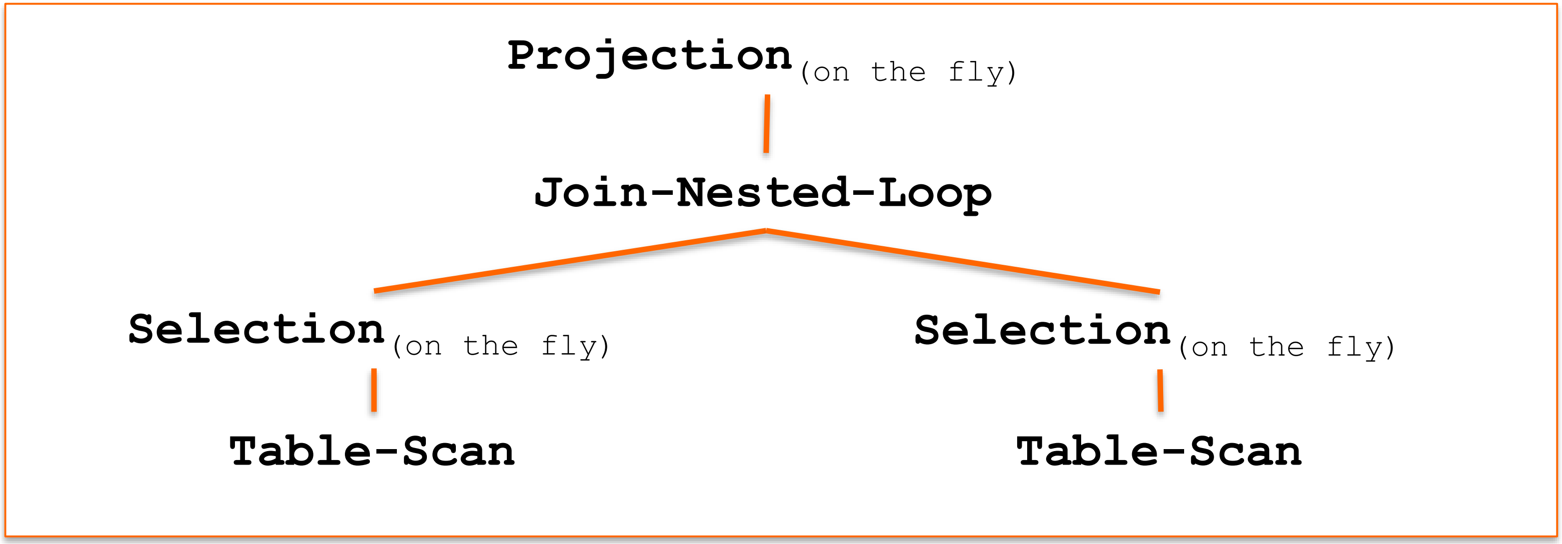
Projection$_{\texttt{L.subject, L.object, W.object}}$

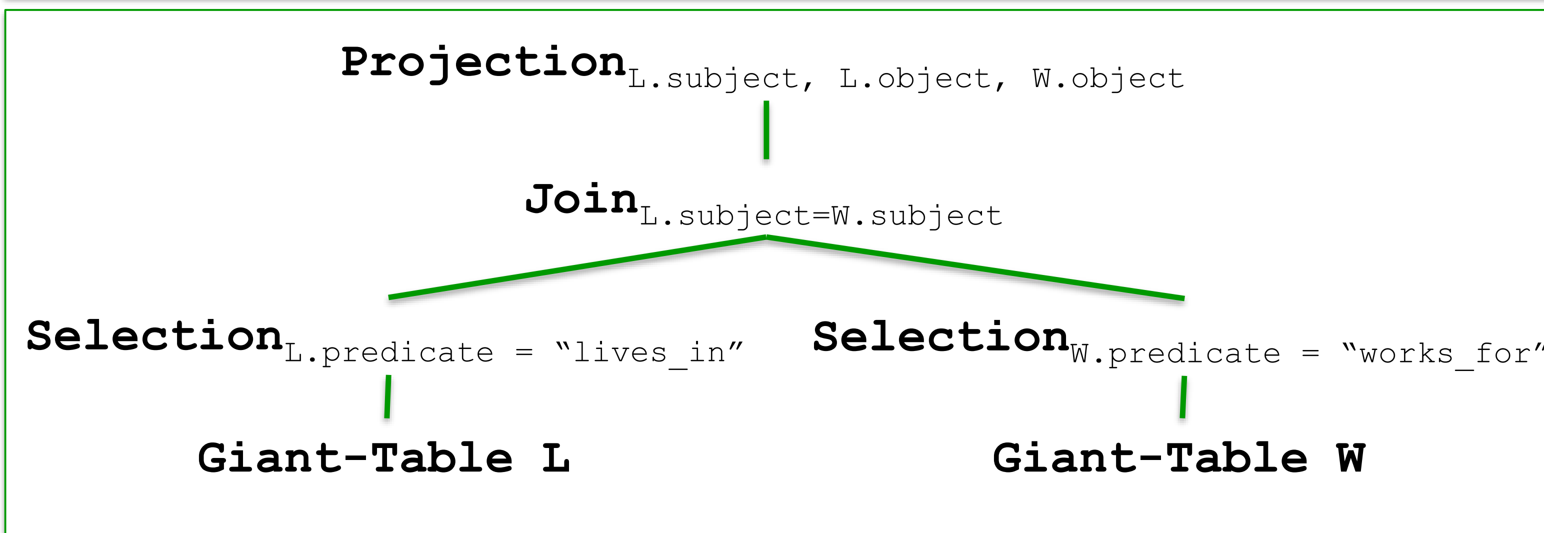
Join$_{\texttt{L.subject=W.subject}}$

Selection$_{\texttt{L.predicate = “lives\_in”}}$ **Selection**$_{\texttt{W.predicate = “works\_for”}}$

Giant-Table L **Giant-Table W**

From Relational Algebra to Physical Plan





From Relational Algebra to Physical Plan

Projection (on the fly)

Join-Nested-Loop

Selection (on the fly) **Selection** (on the fly)

Table-Scan **Table-Scan**

- Now, performance depend on the <u>row-store.</u>

row1	alice@works_for@EDF	**row2**	alice@livesIn@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

subject	**predicate**	**object**
alice	works_for	EDF
alice	lives_in	Paris
	…	

works_for

subject	**object**
alice	EDF
…	

lives_in

subject	**object**
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
WHERE  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

subject	object
alice	EDF
…	

lives_in

subject	object
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

BMW	*Los Angeles Airport*	*Alex Ferguson*	*John Belushi*	*Apple INC*	*Charlie Chaplin*

Clustered Properties

sport_man_property_cluster

subject	isAffiliatedTo	playsFor	hasGender	wasBornIn
Ferguson	Manchester	Manchester	Male	Scotland
Ronaldo	UNICEF	Inter	Male	Brazil
		…		

employee_property_cluster

subject	works_for	lives_in
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

subject	object
alice	EDF
alice	Paris
…	

lives_in

subject	object
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
```

subject	**works_for**	**lives_in**
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

subject	works_for	lives_in
alice	EDF	Paris
	…	

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**

- 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

subject	**isAffiliatedTo**	**playsFor**	**hasGender**	**wasBornIn**
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	**UNICEF**	Milan	**Male**	**Brazil**
Ronaldo	**UNICEF**	Barcelona	**Male**	**Brazil**
Ronaldo	**UNICEF**	RealMadrid	**Male**	**Brazil**
Ronaldo	**UNICEF**	Flamenco	**Male**	**Brazil**
		…		

- Note : by the way, the 4th 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

subject	predicate	object
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 inconveniences of the former method

class:Book

subject	title	author	year
ID1	XYZ	Joe Fox	2001
ID3	MNP	null	null
ID6	null	null	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 (~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

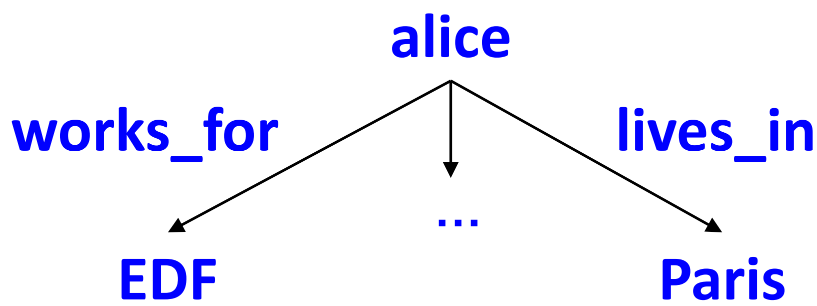
Map strings to unique integers (e.g., via hashing)

- 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



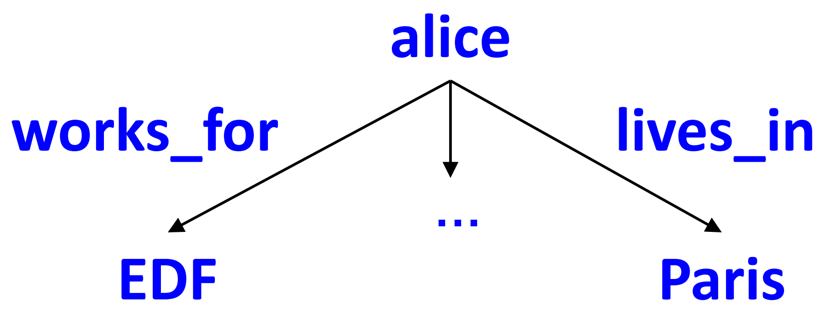
dictionary

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

Giant-Table

subject	pred.	object
1		

Dictionnary



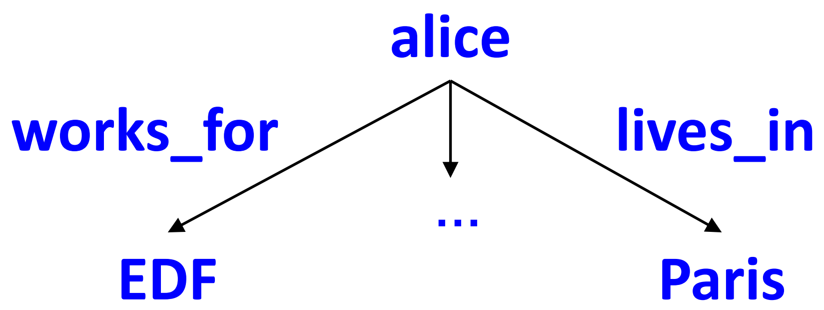
dictionary

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

Giant-Table

subject	pred.	object
1	2	

Dictionnary



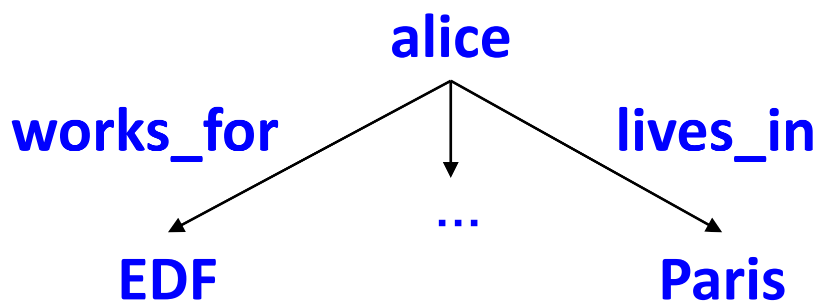
dictionary

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

Giant-Table

subject	pred.	object
1	2	4

Dictionnary



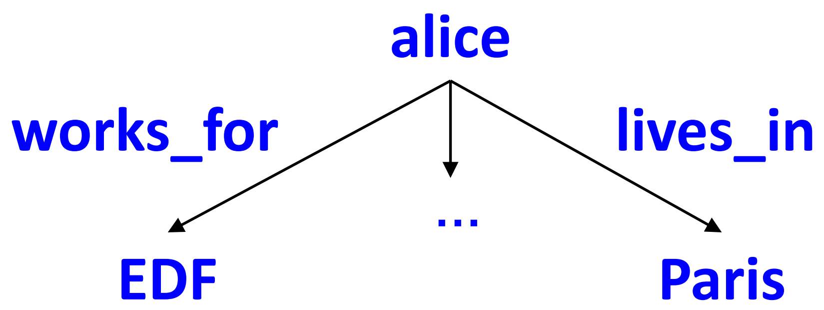
dictionary

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

Giant-Table

subject	pred.	object
1	2	4
1		

Dictionnary



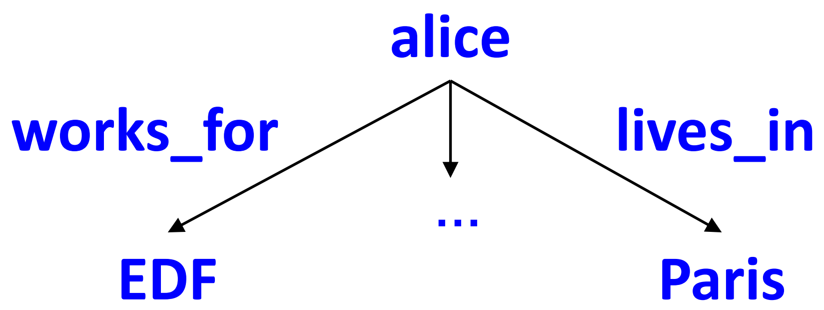
dictionary

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

Giant-Table

subject	pred.	object
1	2	4
1	3	

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

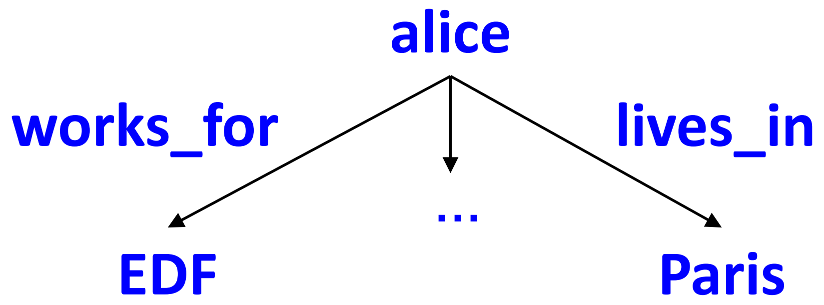
- Giant-Table model + ad-hoc implementation (no RDBMs as we have seen before)

- 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)
```

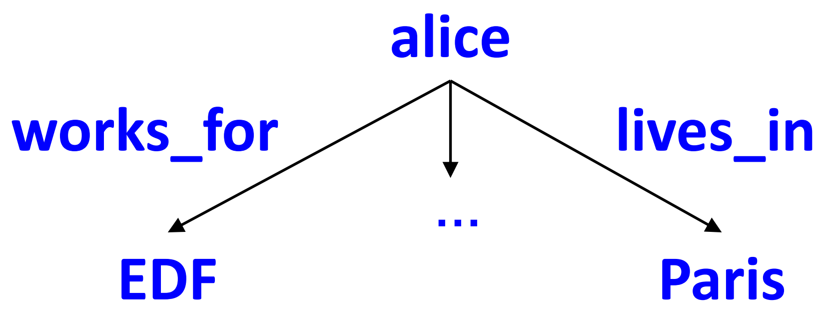
So we can store triples pattern-wise



`Giant-Table<SPO>`

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

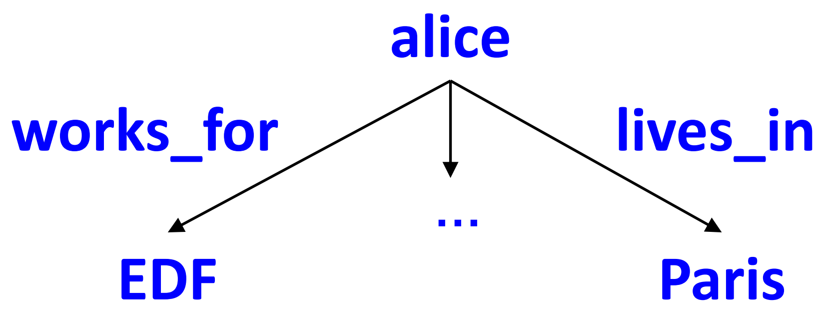
So we can store triples pattern-wise



Giant-Table<SOP>

subject	object	predicate
alice	EDF	works_for
alice	Paris	lives_in

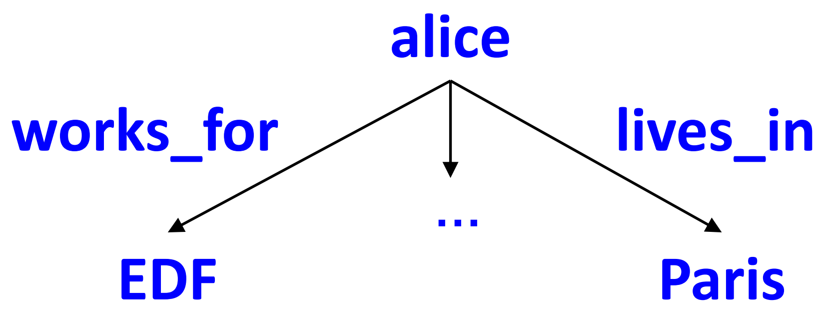
So we can store triples pattern-wise



Giant-Table<OPS>

object	predicate	subject
EDF	works_for	alice
Paris	lives_in	alice

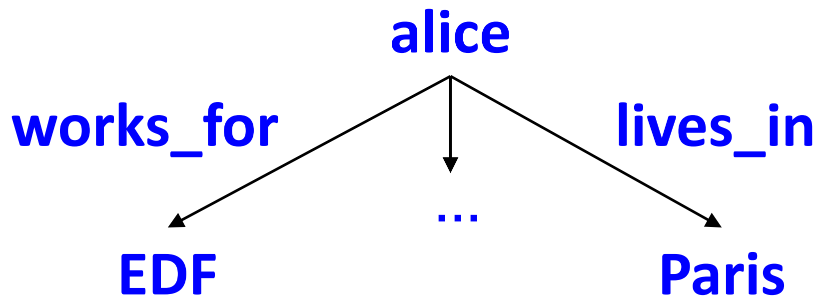
So we can store triples pattern-wise



Giant-Table<POS>

predicate	object	subject
works_for	EDF	alice
lives_in	Paris	alice

So we can store triples pattern-wise



Giant-Table<PSO>

predicate	subject	object
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

row1	alice@works_for@EDF	**row2**	alice@livesIn@Paris	…

Why ? Because we deal with row-stores

Giant-Table<POS>

predicate	object	subject
works_for	EDF	alice
lives_in	Paris	alice

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

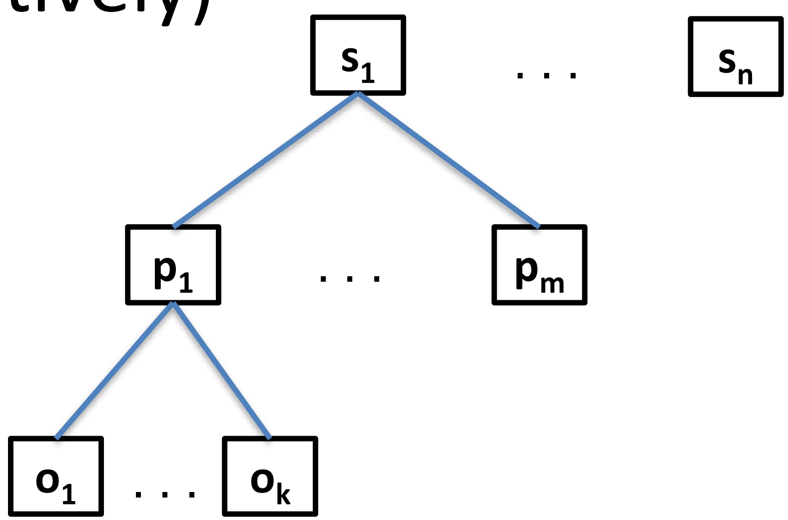
up to 3x faster!!

row1	works_for@EDF@alice	**row2**	livesIn@Paris@alice	…

RDF3X Storage and Indexing

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

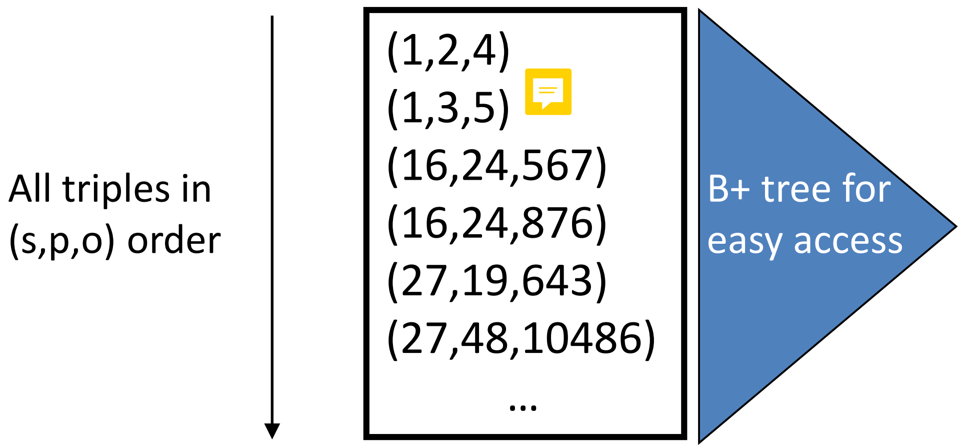
- SPO (Intuitively)



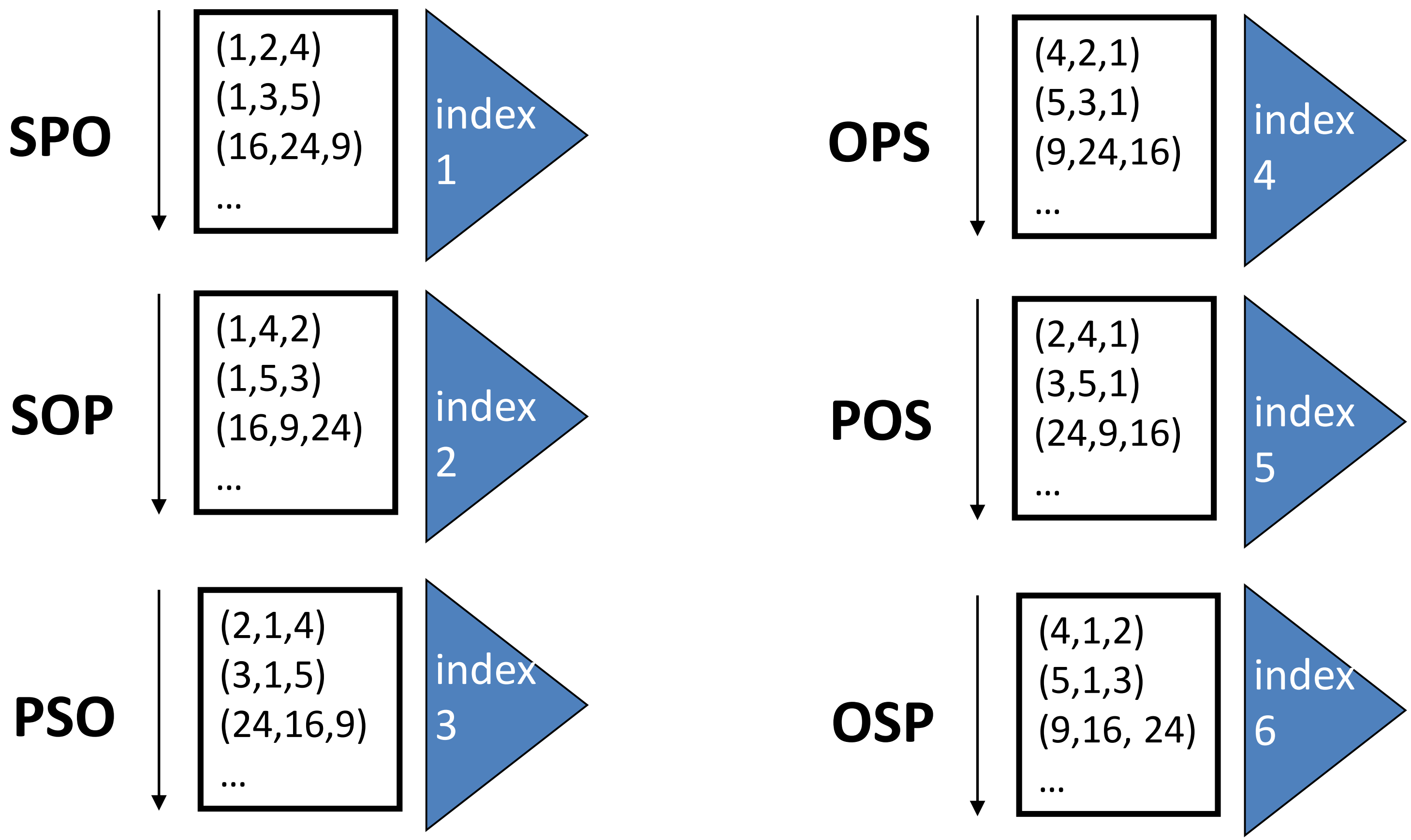
RDF3X Storage and Indexing

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

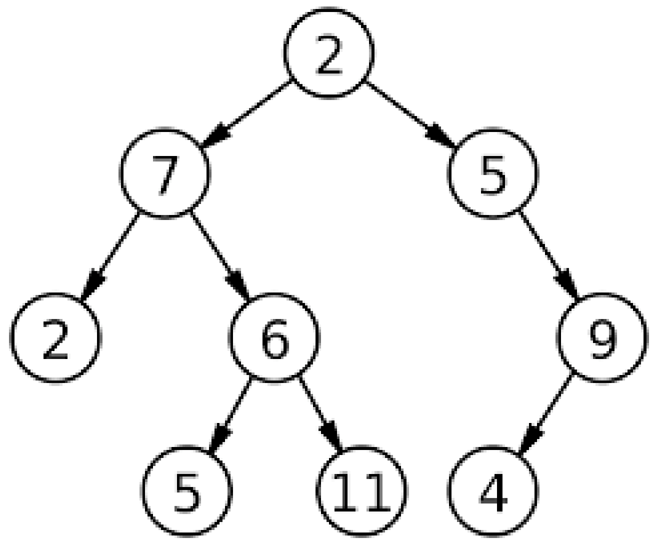
- SPO (in reality) : (B+)-trees over triples



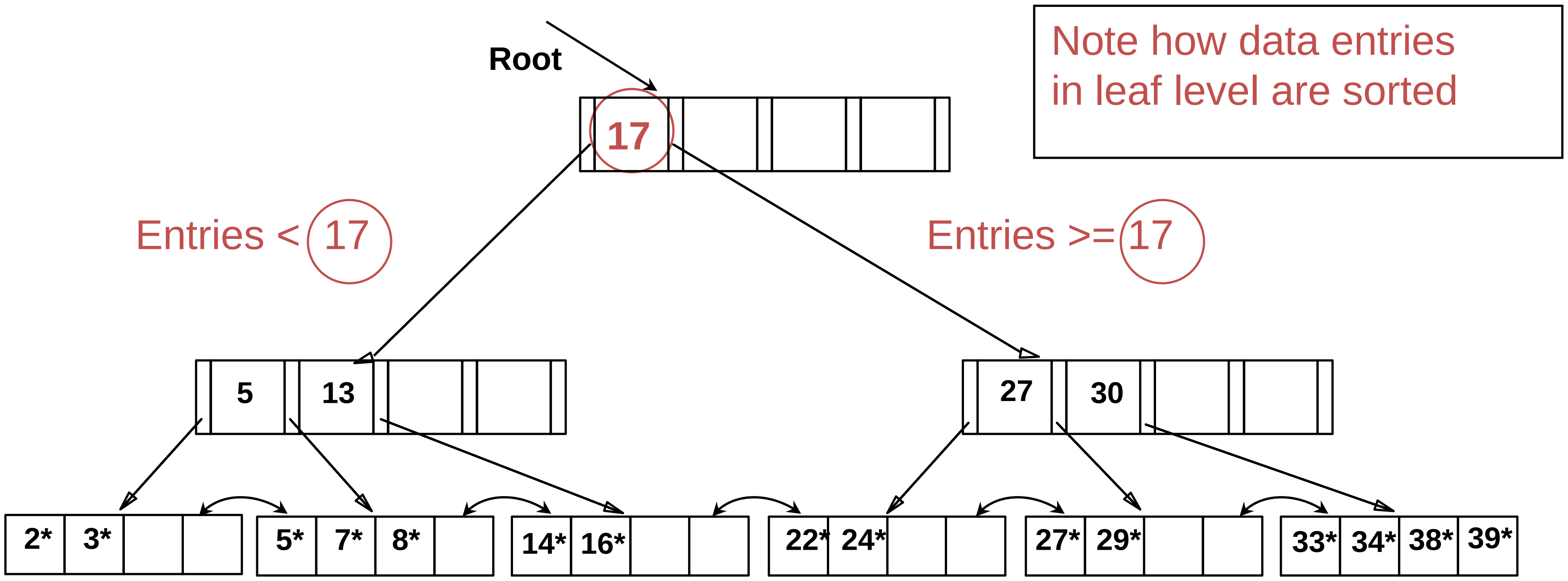
RDF3X 6 indexes (Hexastore)



Binary trees are not enough



B+ Tree

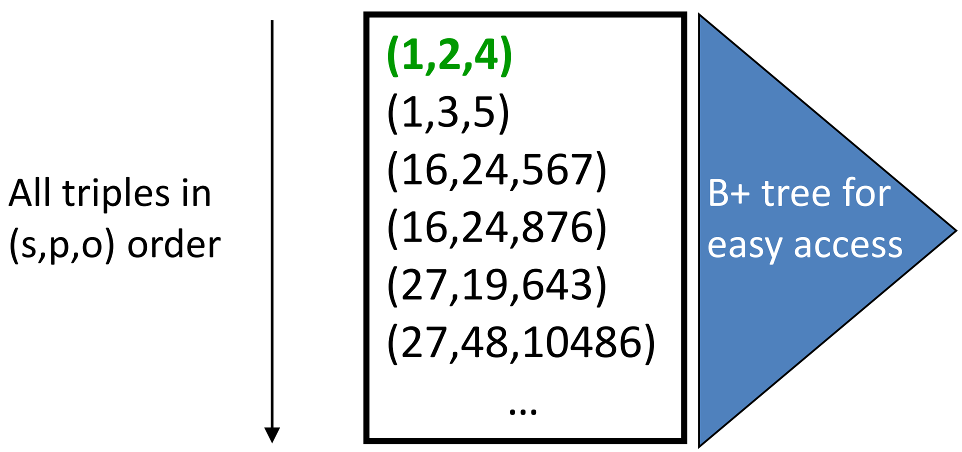


Root
17
Entries < 17
Entries >= 17
Note how data entries in leaf level are sorted
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*

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)**



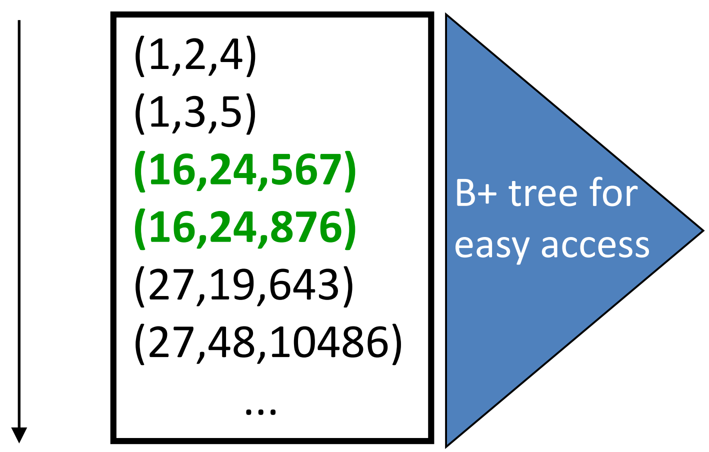
RDF3X Query Processing

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

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

already sorted

All triples in (s,p,o) order



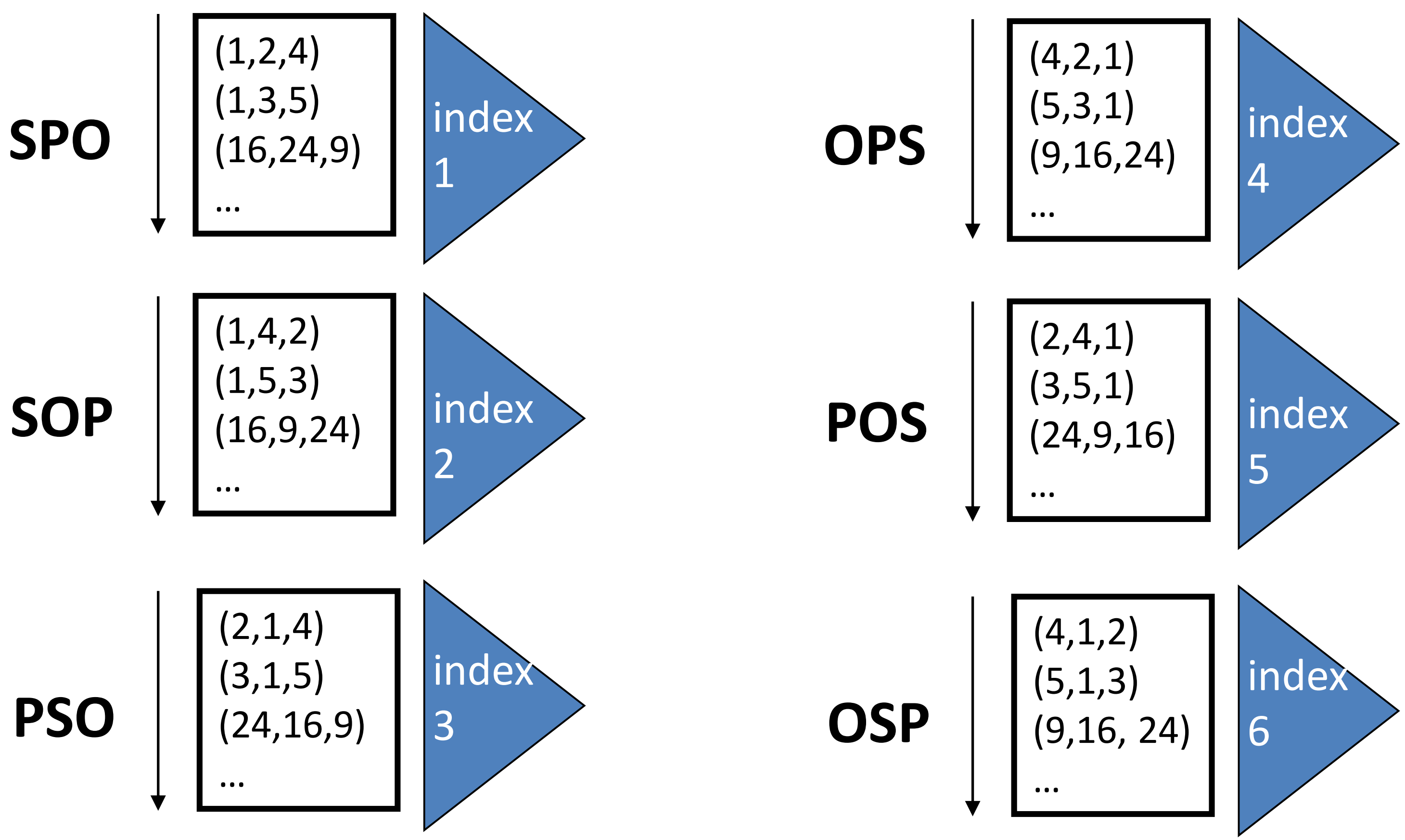
RDF3X Storage and Indexing

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 ?

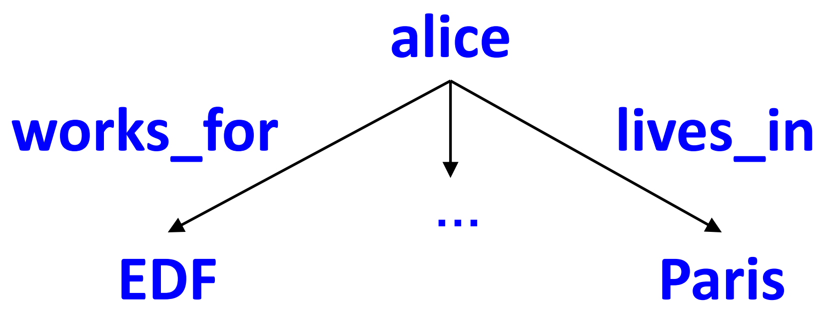


Now, how join two (**2**) triple patterns ?

```
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

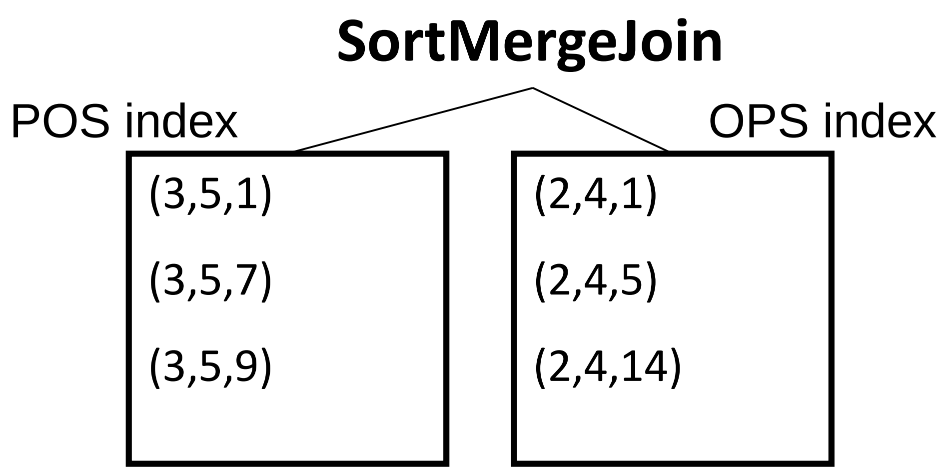
id	string
1	alice
2	works_for
3	lives_in
4	EDF
5	Paris

Giant-Table

subject	pred.	object
1	2	4
1	3	5

RDF3X : Sort-Merge-join

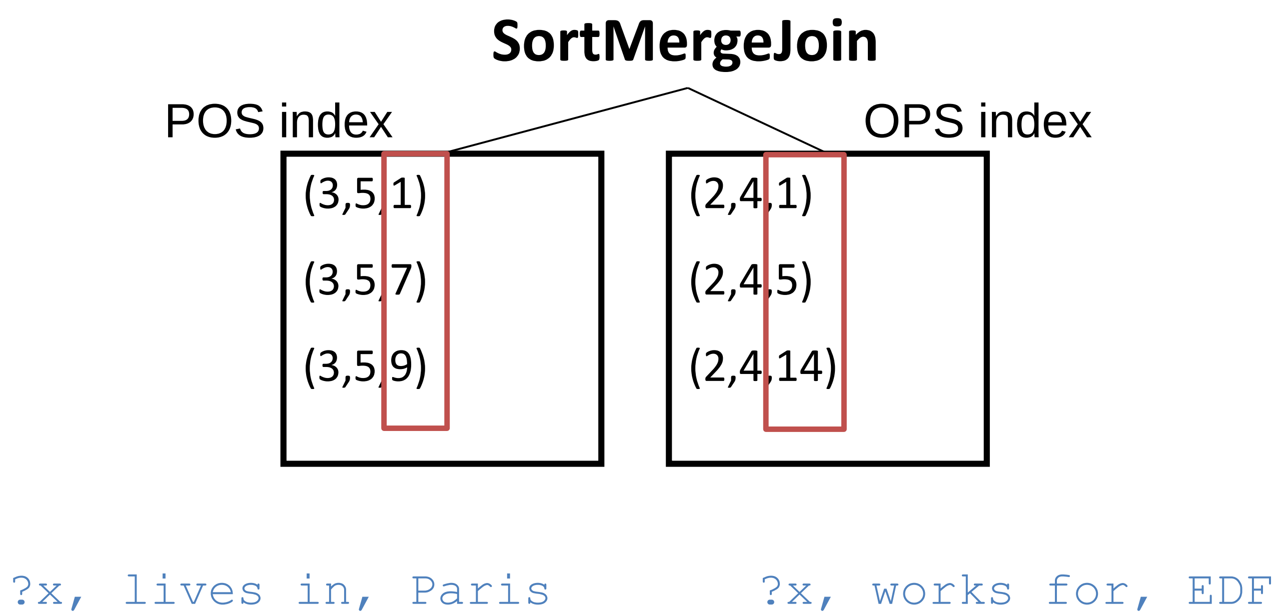
- For example, we decide to use POS & OPS index for 1^{st} & 2^{nd} pattern, respectively.
  – we will see next why we did this choice



`?x, lives_in, Paris`   `?x, works_for, EDF`

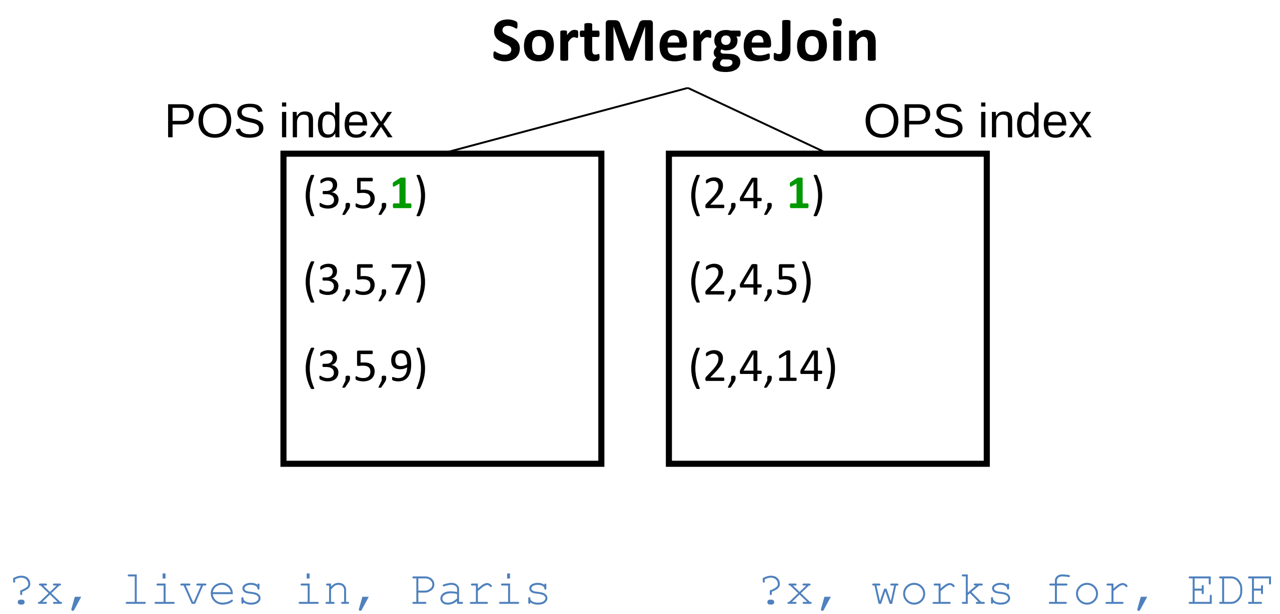
RDF3X : Sort-Merge-join

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



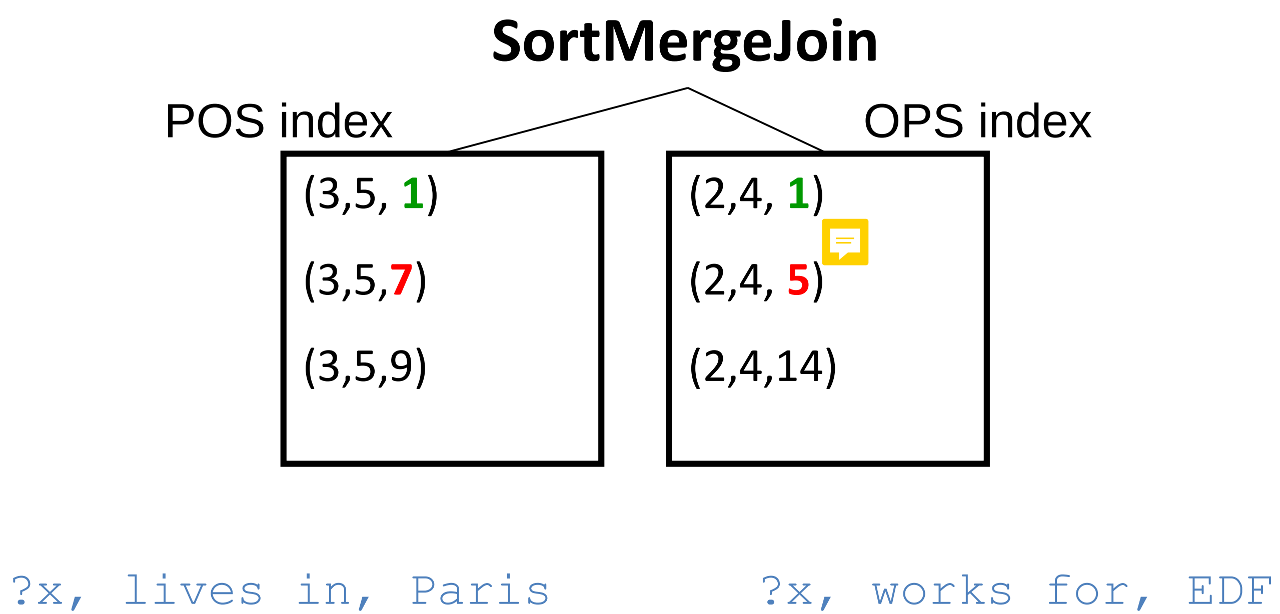
RDF3X : Sort-Merge-join

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



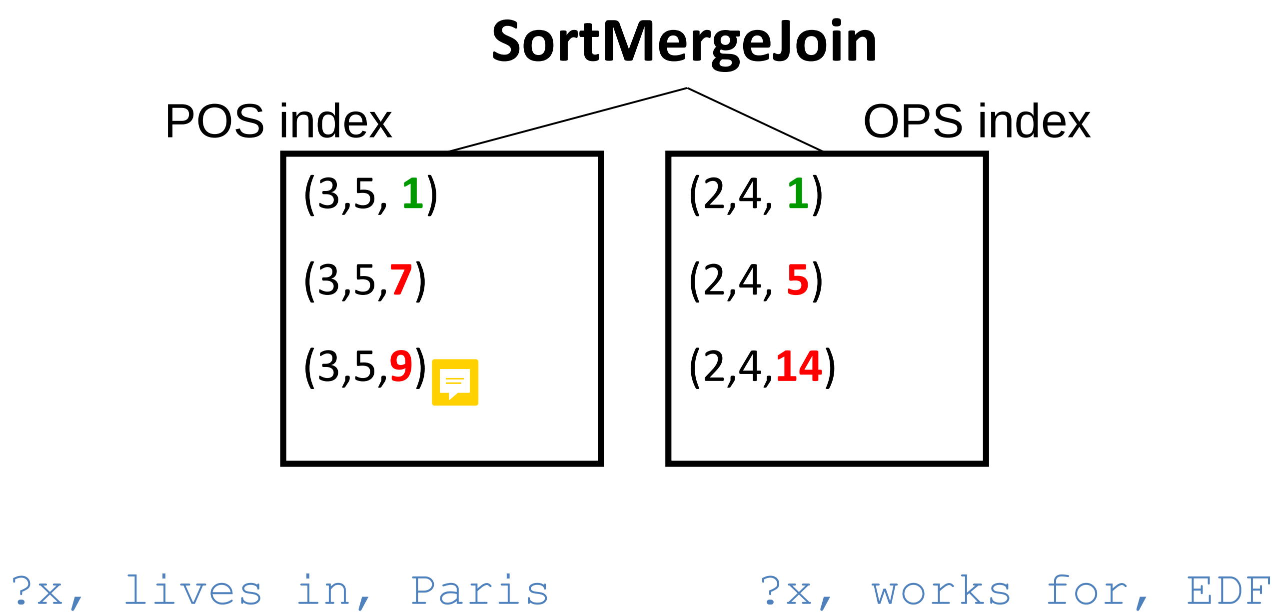
RDF3X : Sort-Merge-join

- We access POS [3.5.7]
  - then we know that OPS[2.4.{2..6}] are not results



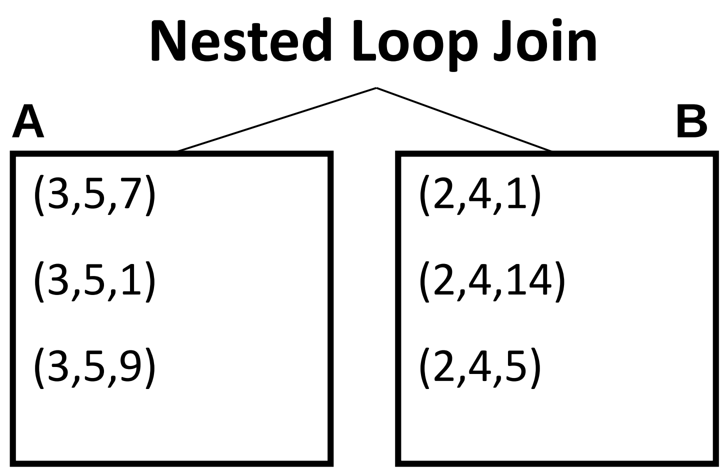
RDF3X : Sort-Merge-join

- 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
```



RDF3X Query Evaluation

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

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

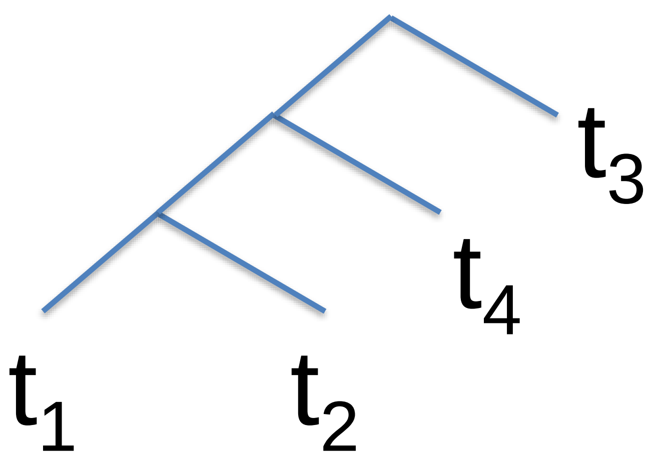
Classical relational problem: choose a left-deep join order, like

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

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

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

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 Join t_2) before understanding that the query is empty

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

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

- then

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

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
                    }
```

COLUMN-STORES

Column-store

Relational database, storing relations as <u>columns</u>.

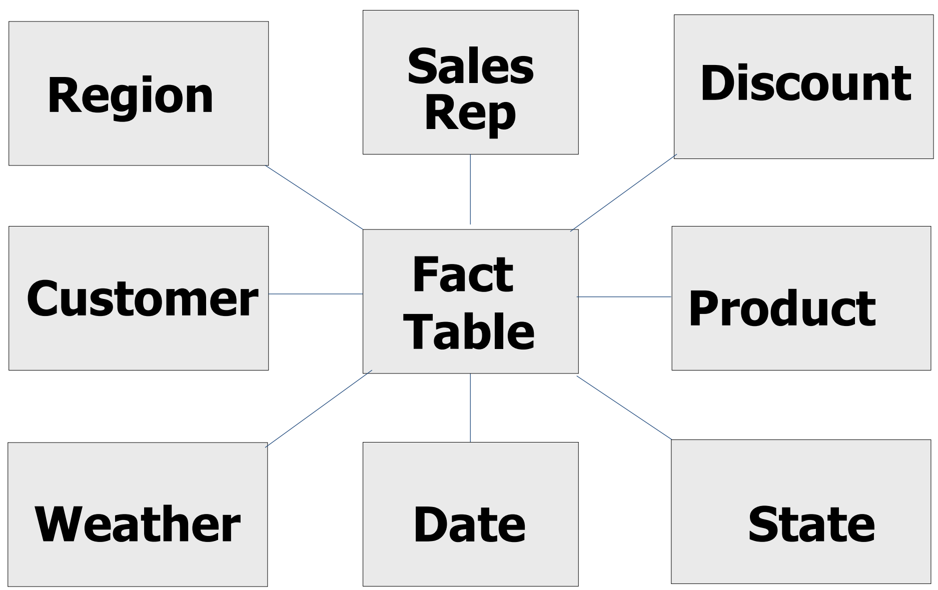
(C-Store, MonetDB and VectorWise, Ingres, IBM&MS Analytics)

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

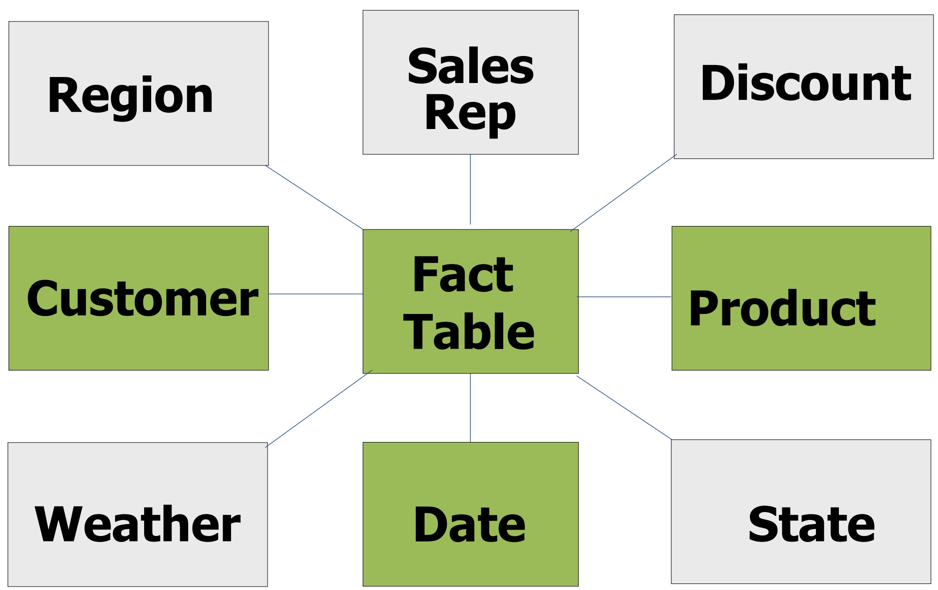
- *«Do not call them column-stores, just modern DB systems.»*
  Stratos Idreos



- 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

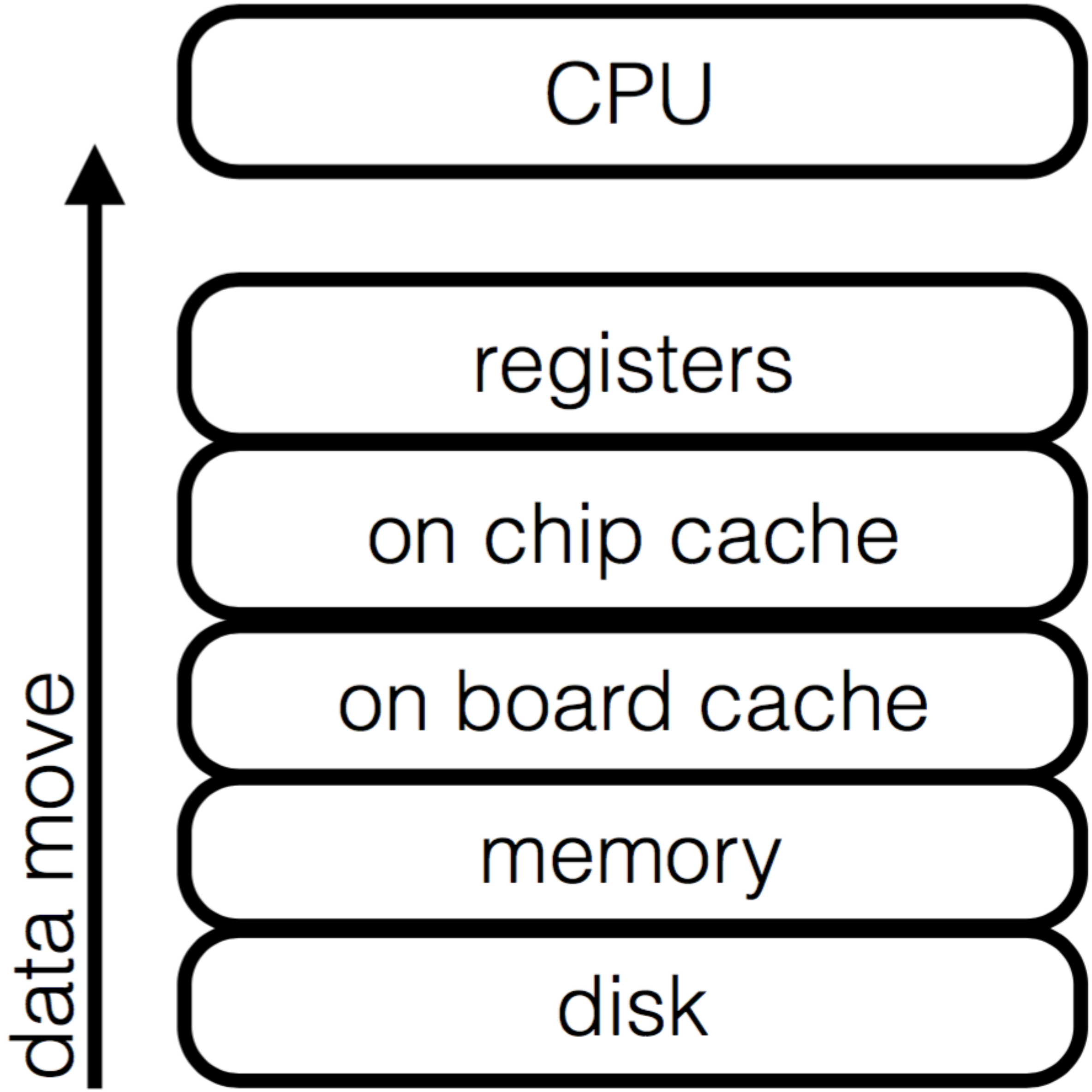
Rows become crowded

row1	id1@customer1@region1@sales_rep1@discount1@product1@weather1@date1@state1…	**row2**	id2@customer2@region2@sales_rep2@discount2@product2@weather2@date2@state2	…

Worst case : visit whole row

row1	**id1**@**customer1**@region1@sales_rep1@discount1@**product1**@weather1@**date1**@state1…	**row2**	**id2**@**customer2**@region2@sales_rep2@discount2@**product2**@weather2@**date2**@state2	…

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

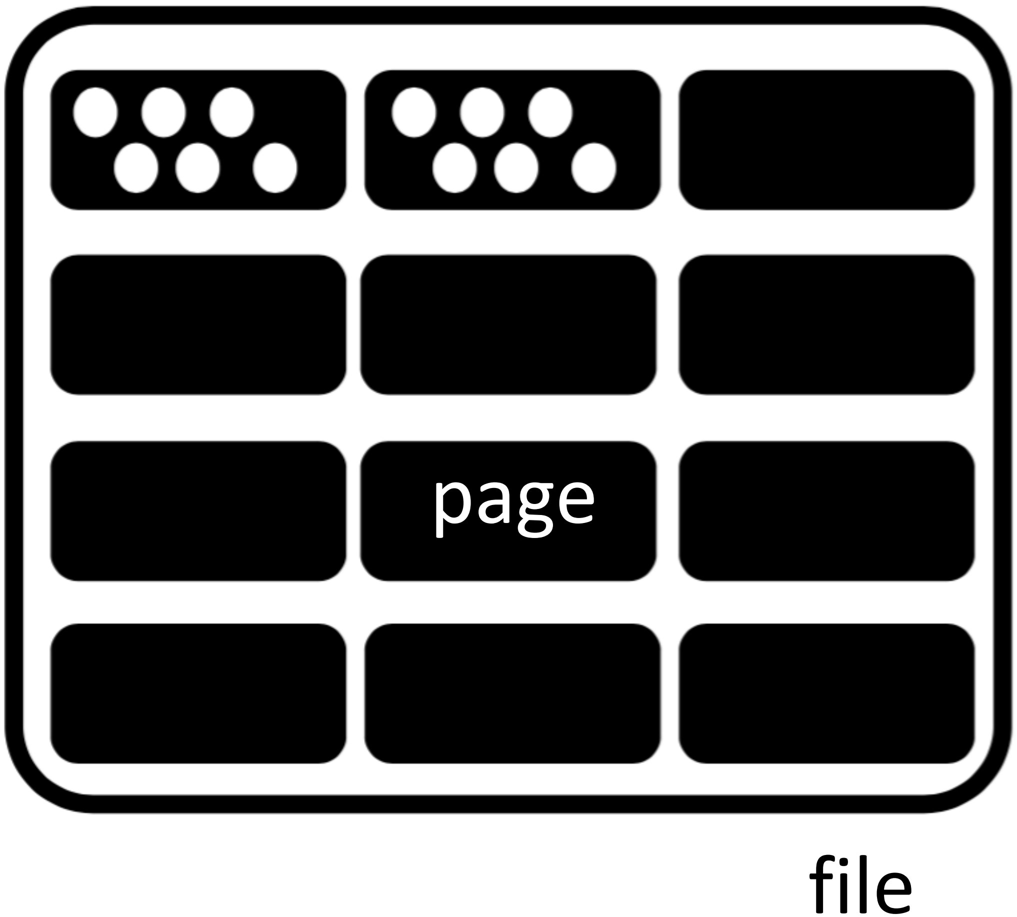

CPU
registers
on chip cache
on board cache
memory
disk
data move

Rows are stored in pages

row1	**id1**@**customer1**@region1@sales_rep1@discount1@**product1**@weather1@**date1**@state1…	**row2**	**id2**@**customer2**@region2@sales_rep2@discount2@**product2**@weather2@**date2**@state2	…

page

Pages are stored within files

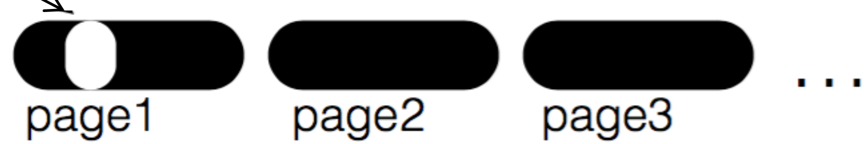


Constraint : Pages are read as a whole!

need to only read row ***x***...

but have to read all of page 1

data value ***x***

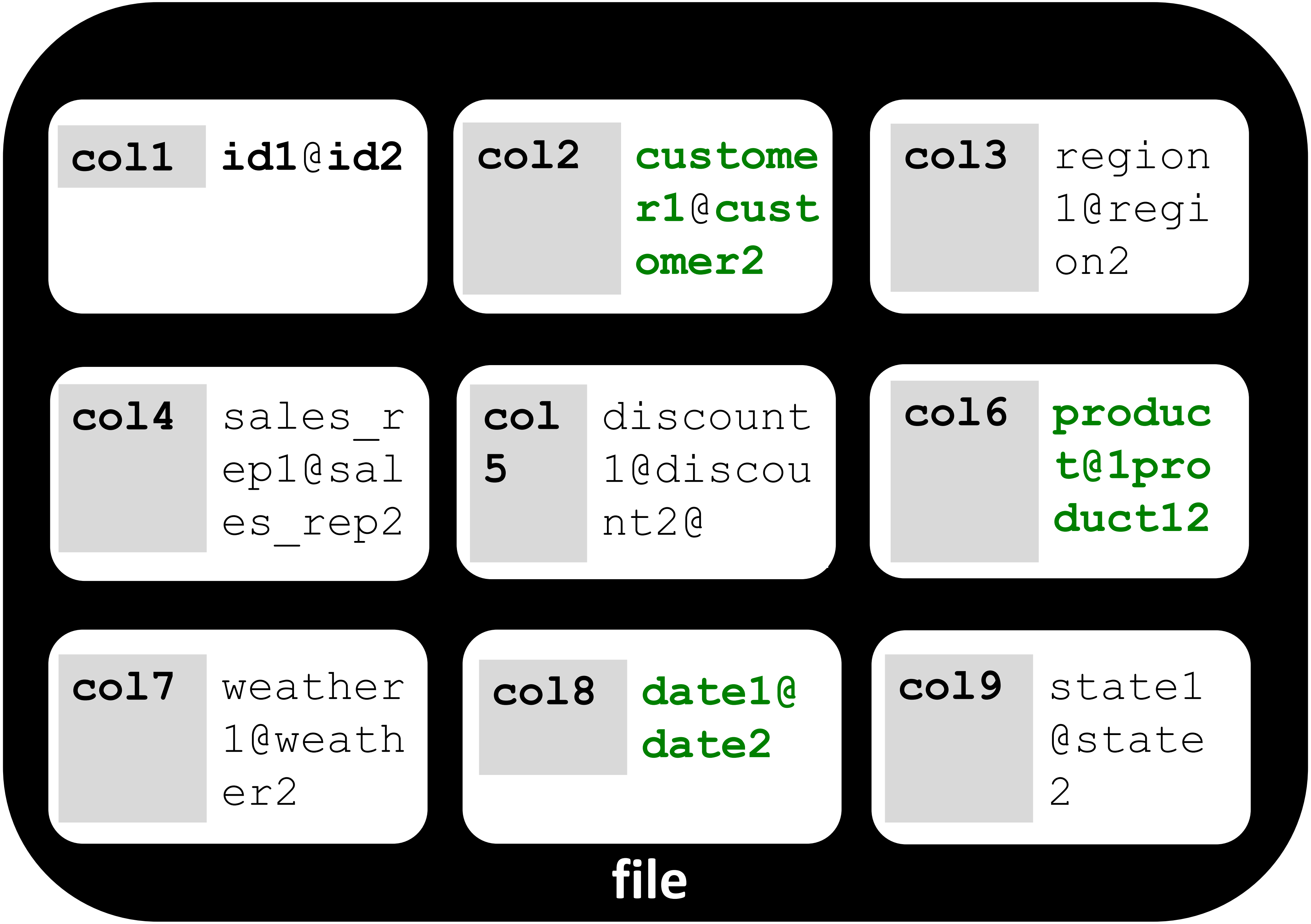


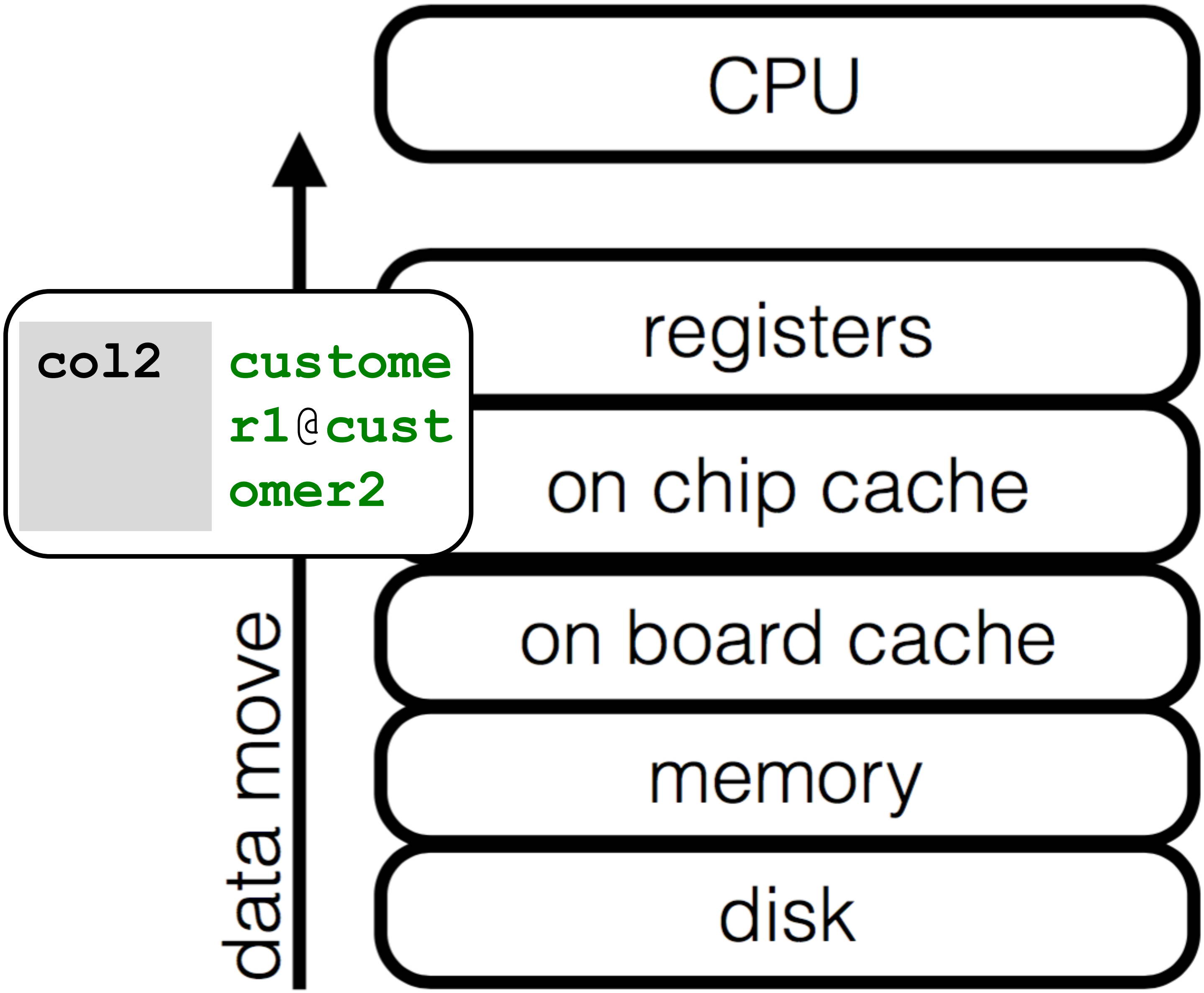
Better to « pay as you go »!

col1	**id1@id2**	**col2**	**customer1@customer2**	**col3**	region1@region2
col4	sales_rep1@sales_rep2	**col5**	discount1@discount2@	**col6**	**product@1product12**
col7	weather1@weather2	**col8**	**date1@date2**	**col9**	state1@state2

Store columns on separate pages!

col1	**id1@id2**	**col2**	**customer1@customer2**	**col3**	region1@region2
col4	sales_rep1@sales_rep2	**col5**	discount1@discount2@	**col6**	**product@1product12**
col7	weather1@weather2	**col8**	**date1@date2**	**col9**	state1@state2


col1
id1@id2
col2
customer1@customer2
col3
region1@region2
col4
sales_rep1@sales_rep2
col5
discount1@discount2@
col6
product@1product12
col7
weather1@weather2
col8
date1@date2
col9
state1@state2
file


CPU
registers
on chip cache
on board cache
memory
disk
data move
col2
custome
r1@cust
omer2

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
	…	

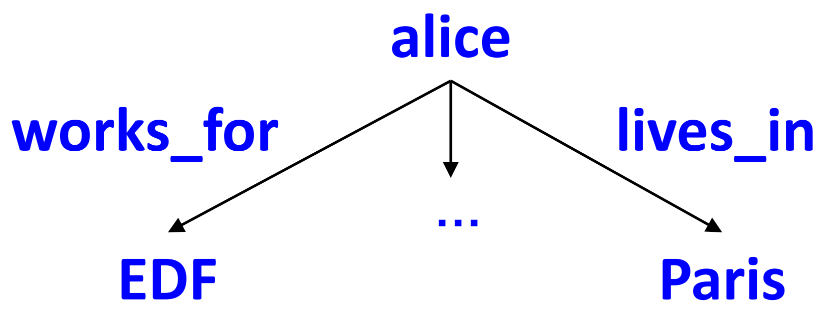
works_for

subject	object
alice	EDF
…	

lives_in

subject	object
alice	Paris
…	

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



Property-Tables

works_for

subject	object
alice	EDF
bob	SFR
...	

lives_in

subject	object
bob	Lyon
alice	Paris
...	

works_for	col1	alice@bob...	col2	EDF@SFR...
lives_in	col1	bob@alice...	col2	Paris@Lyon...

Neat advantage : column-projection

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

```
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>

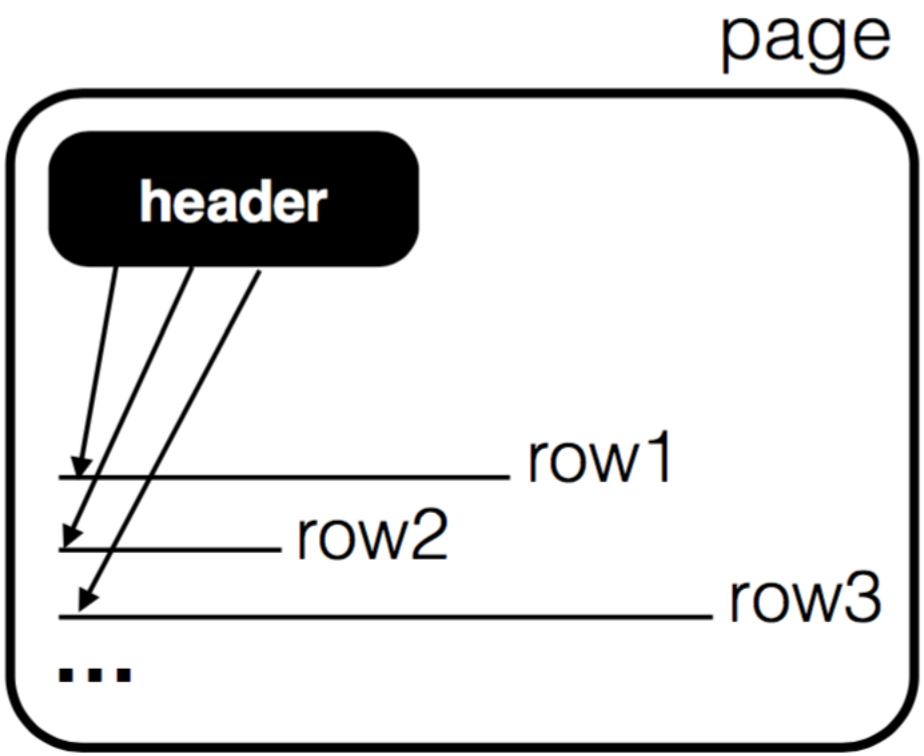
- Big question : when to reconstruct a tuple ? (alap)

Advantages of Column Stores

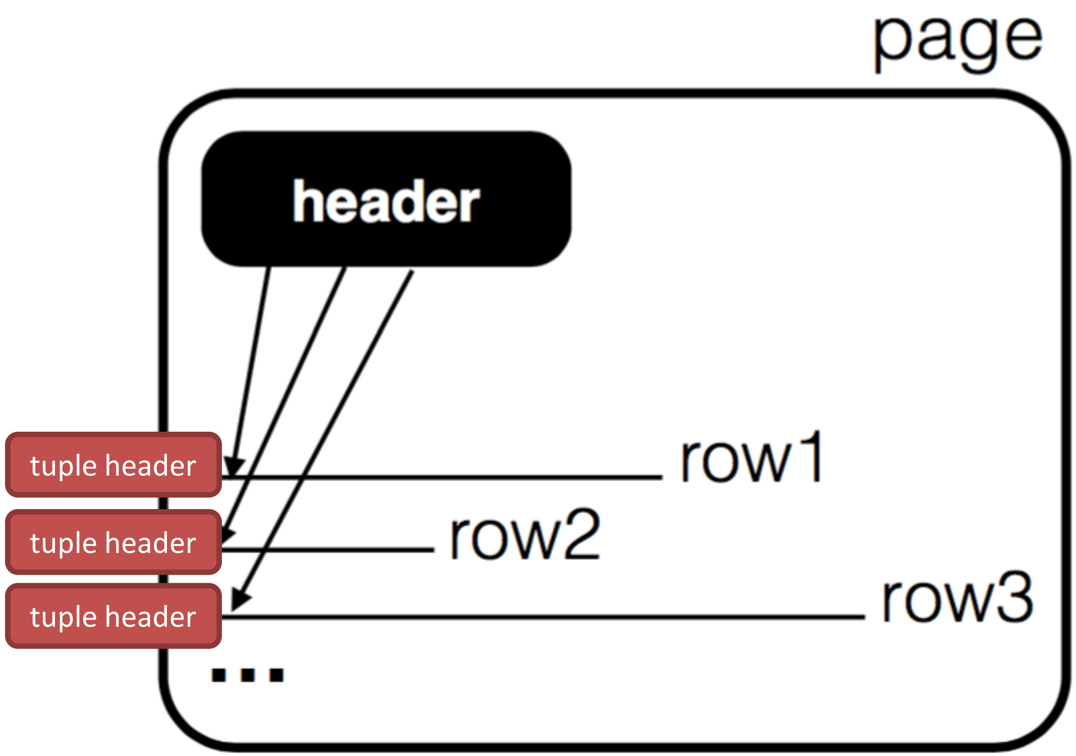
1. Tuple Headers Stored Separately
2. Optimizations for fixed-length tuples
3. Column-oriented data compression
4. Carefully optimized merge-join code

TUPLE HEADERS

Page headers in Rowstores



Tuple headers in Rowstores



$$|\text{ tuple header }| \gg |\text{row}_{\text{of size 2}}|$$

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

$$| \text{header} | \gg |\text{row}_{\text{of size 2}}|$$

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	

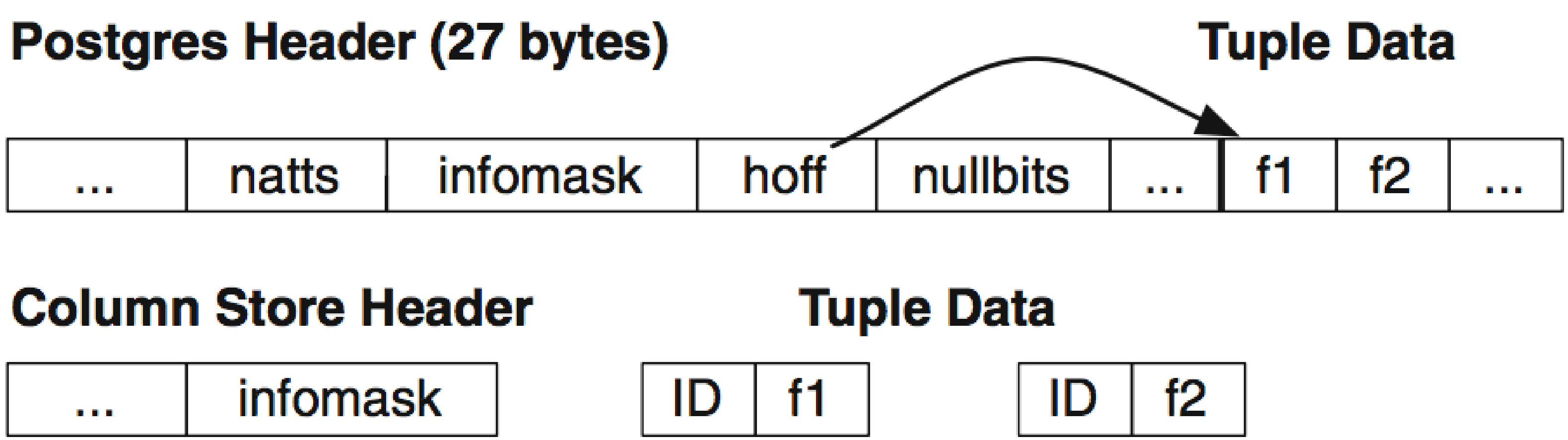
Postgres Tuple Header

- 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



OPTIMIZATIONS FOR FIXED-LENGTH TUPLES

Optimizations for variable-length tuples

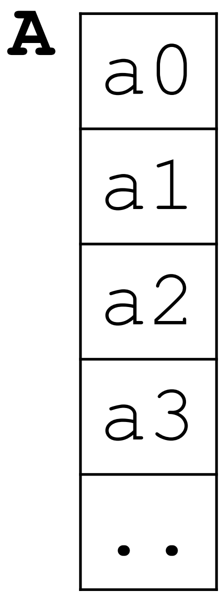
- 1 attribute variable length => the whole tuple is
- Common case: row-stores designed for this

```
row1 idcustomer1@fistname1@lastname1@...
```

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

Optimizations for fixed-length tuples

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



Positional lookup

`a(i) = A + i*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

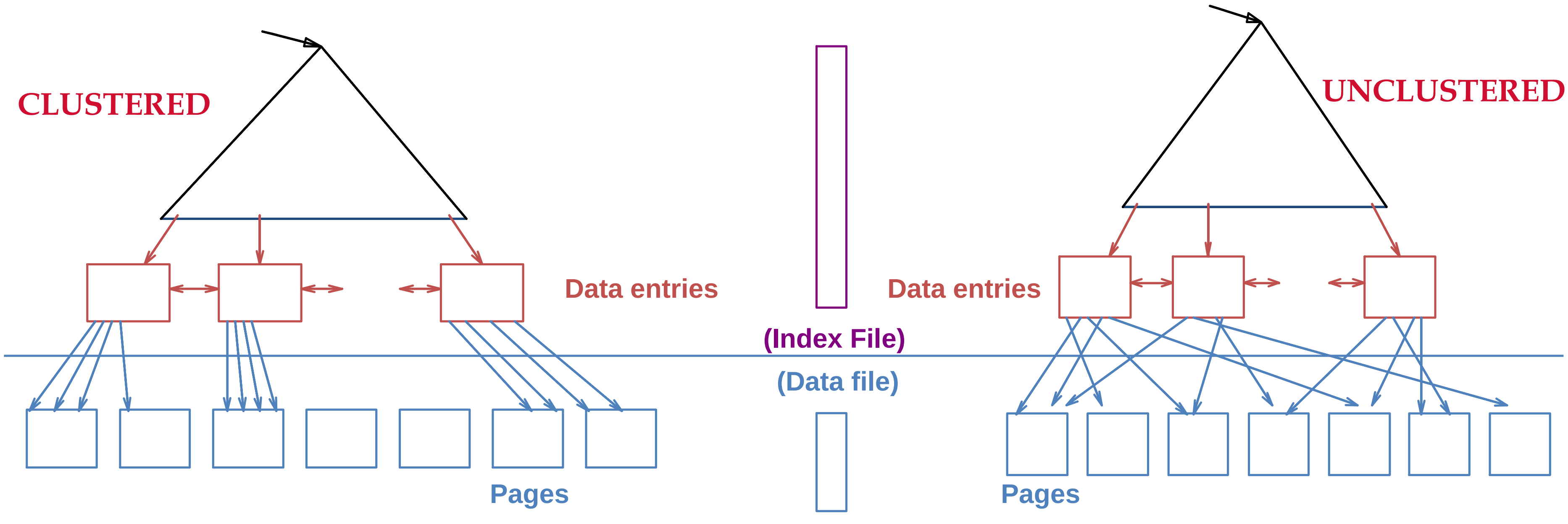
P

s0	o0
s1	o1
s2	o2
s3	o3
.. ..	

`o(i) = P + i *(width(S) + width(O)) + 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_{Obama}
id_{Alice}
id_{Paris}
..

Id-Table

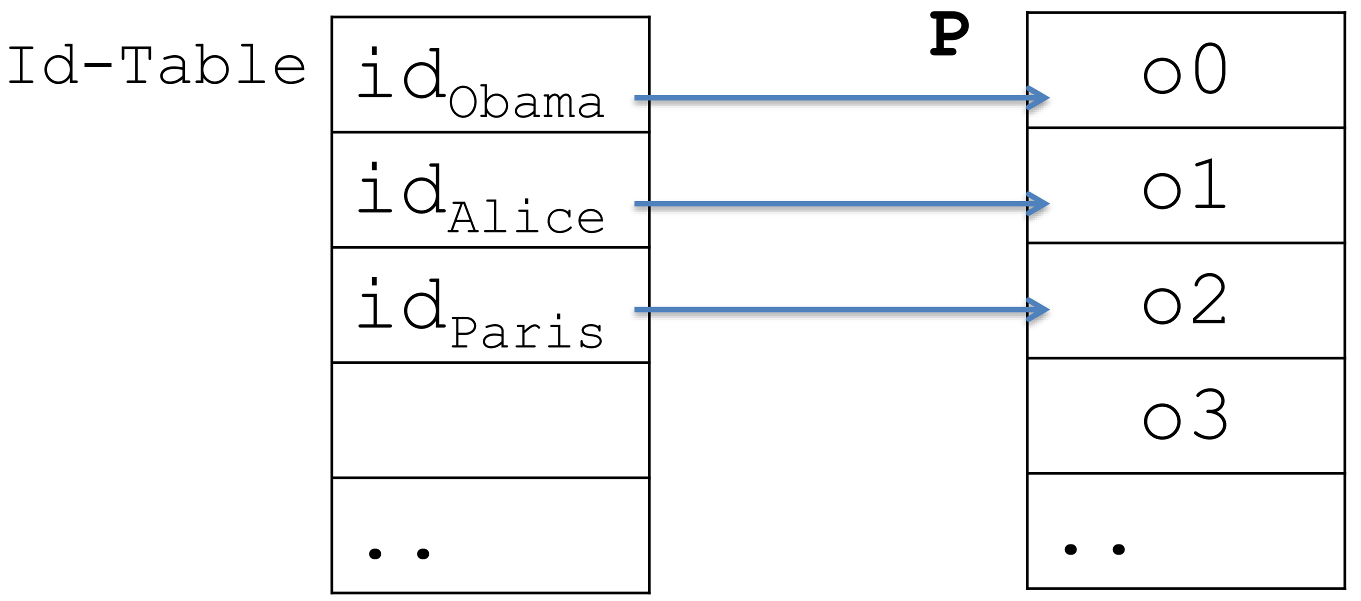
- If property **P** is <u>NOT</u> multivalued (eg. birthday) we can avoid to store subject id

Id-Table
id_{Obama}
id_{Alice}
id_{Paris}
..

P
o0
o1
o2
o3
..

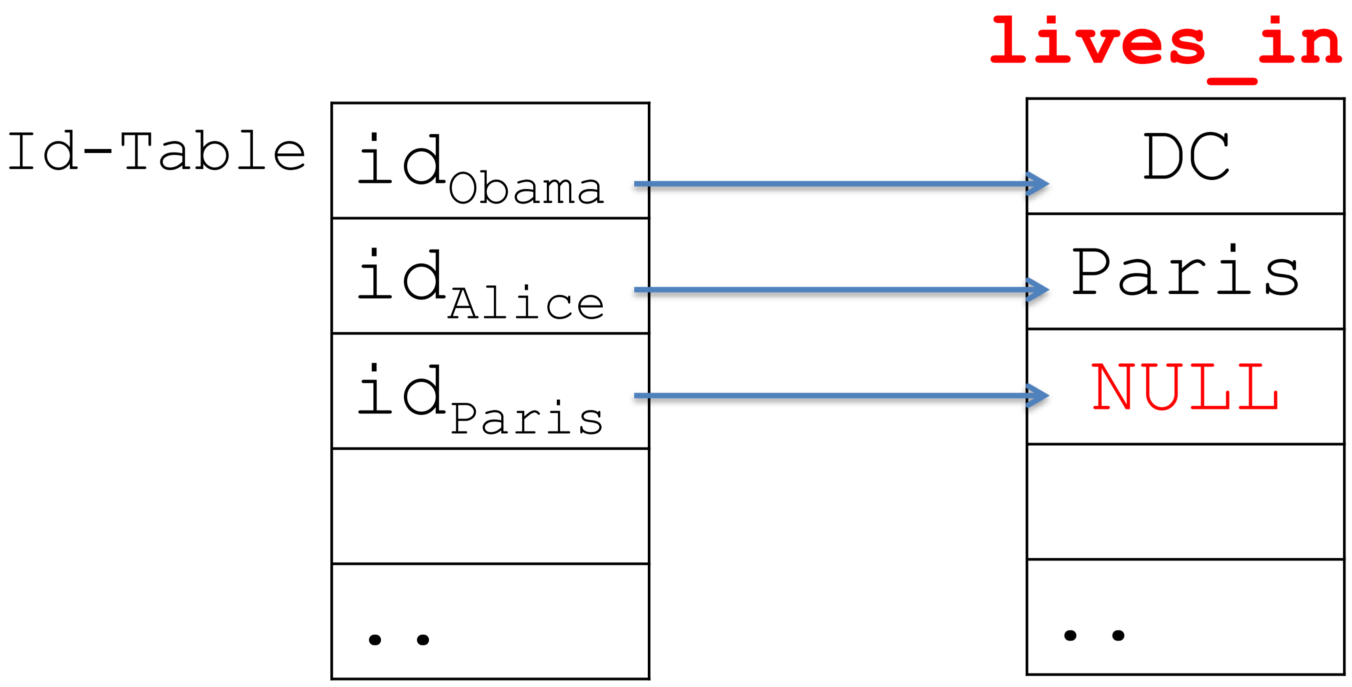
Id-Table

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



Nulls are back!

- If property **P** is <u>NOT</u> 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 Range[1-4]

t1
t2
t3
t4

Case 1: Low overhead for «dense» data

- Store indexes of non-null elements

Language
NULL
FR
EN
NULL
NUL

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

Case 2: data not «dense» nor «sparse»

- Store a bitmap index (0=NULL)

Copyright Bit:101011

2001
NULL
1985
NULL
1995

Case 2: data not «dense» nor «sparse»

- Store a bitmap index (0=NULL)

Copyright Bit:101011

2001
NULL
1985
NULL
1995

- Overhead = 1bit per value

Case 3 : sparse data

- Store the list of non-null ids

Author

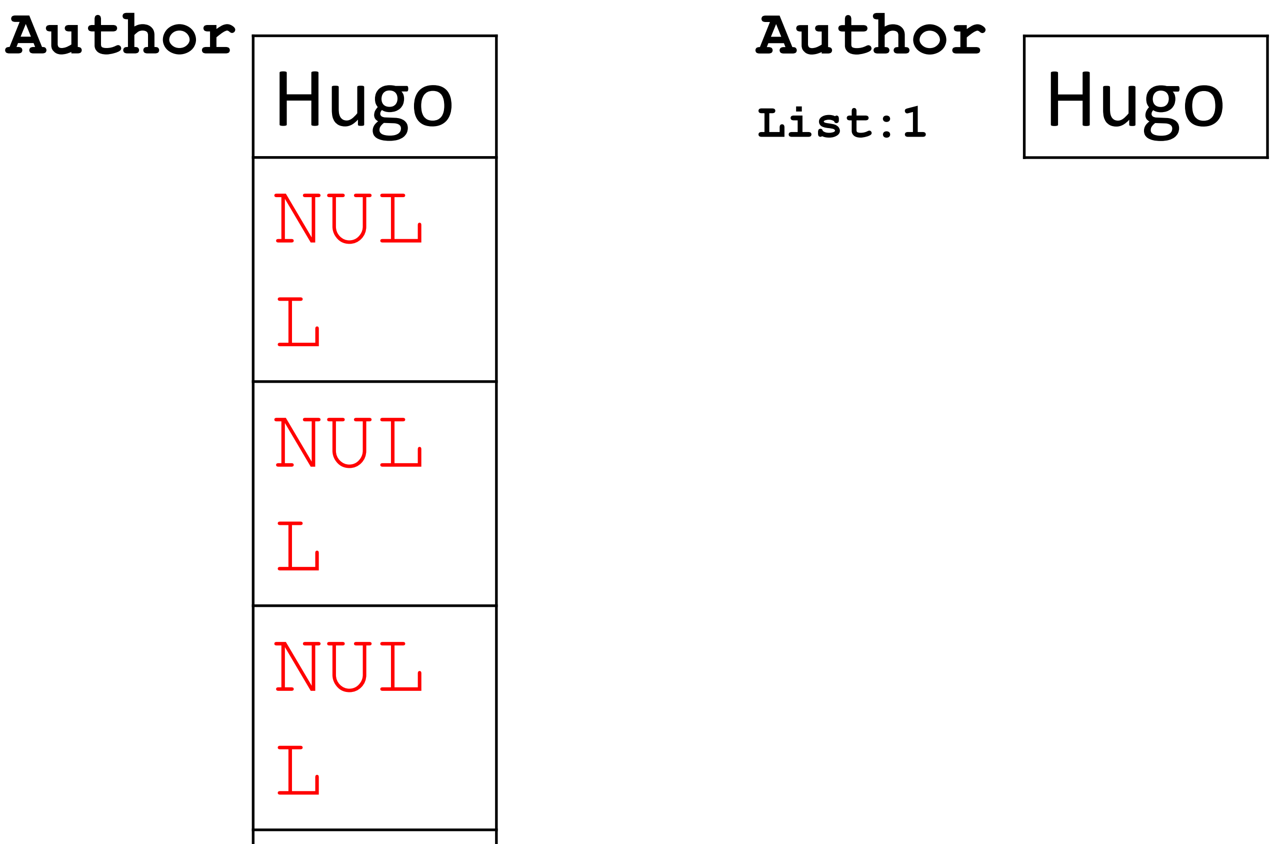
Hugo
NULL
NULL
NULL

Artist

NULL
Dylan
NULL
NULL

Case 3 : sparse data

- Store the list of non-null ids



Case 3 : sparse data

- Store the list of non-null ids

Artist

NULL
Dylan
NULL
NULL

Artist

List:2

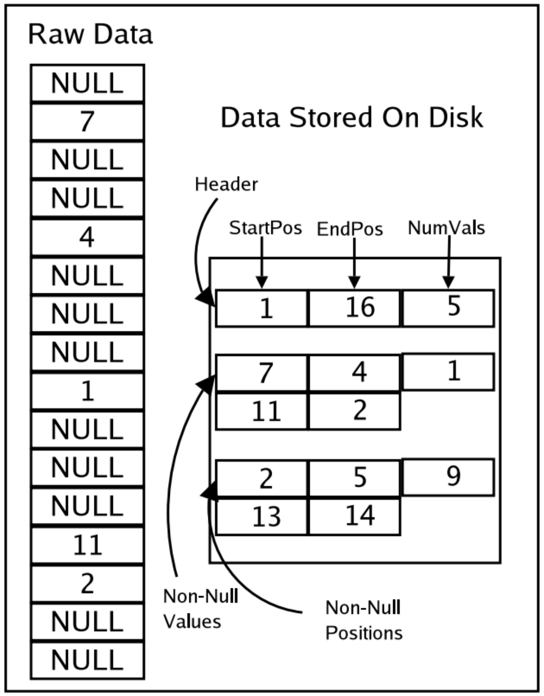
Dylan

Case 3 : sparse data

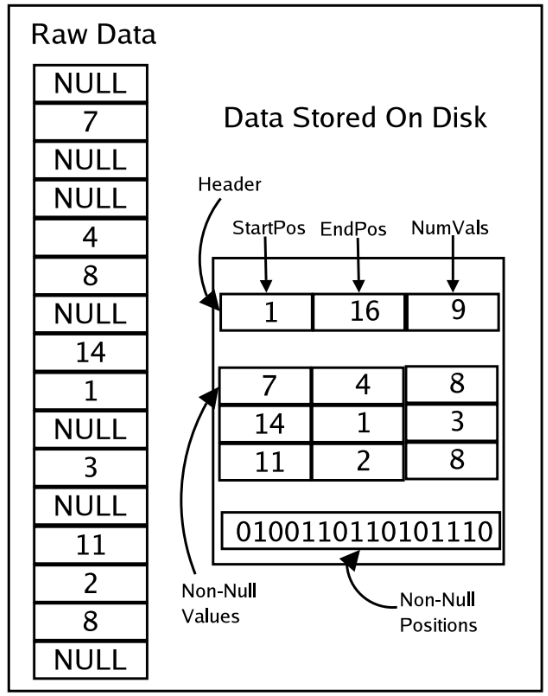
- Store the list of non-null ids

- If data is sparse

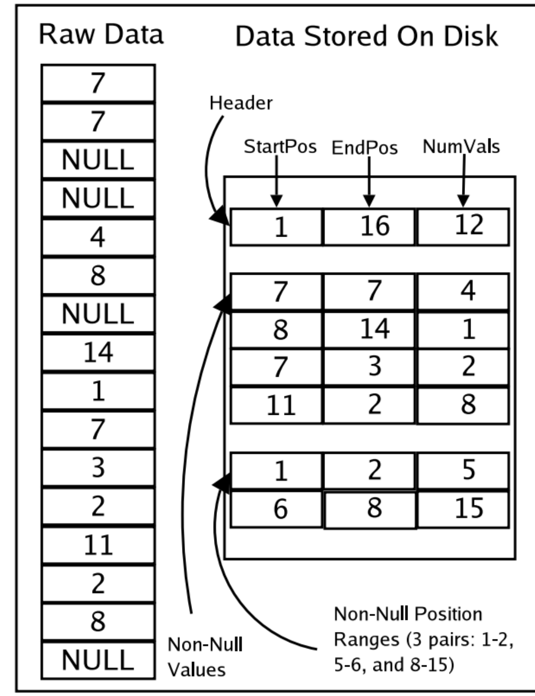
$$|List| << |nulls|$$



(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

triple pattern selectivity

```
SELECT ?y ?z where {
        ?x, author,  Hugo        .
        ?x, title, ?y            .
        ?x copyright ?z          .
        }
```

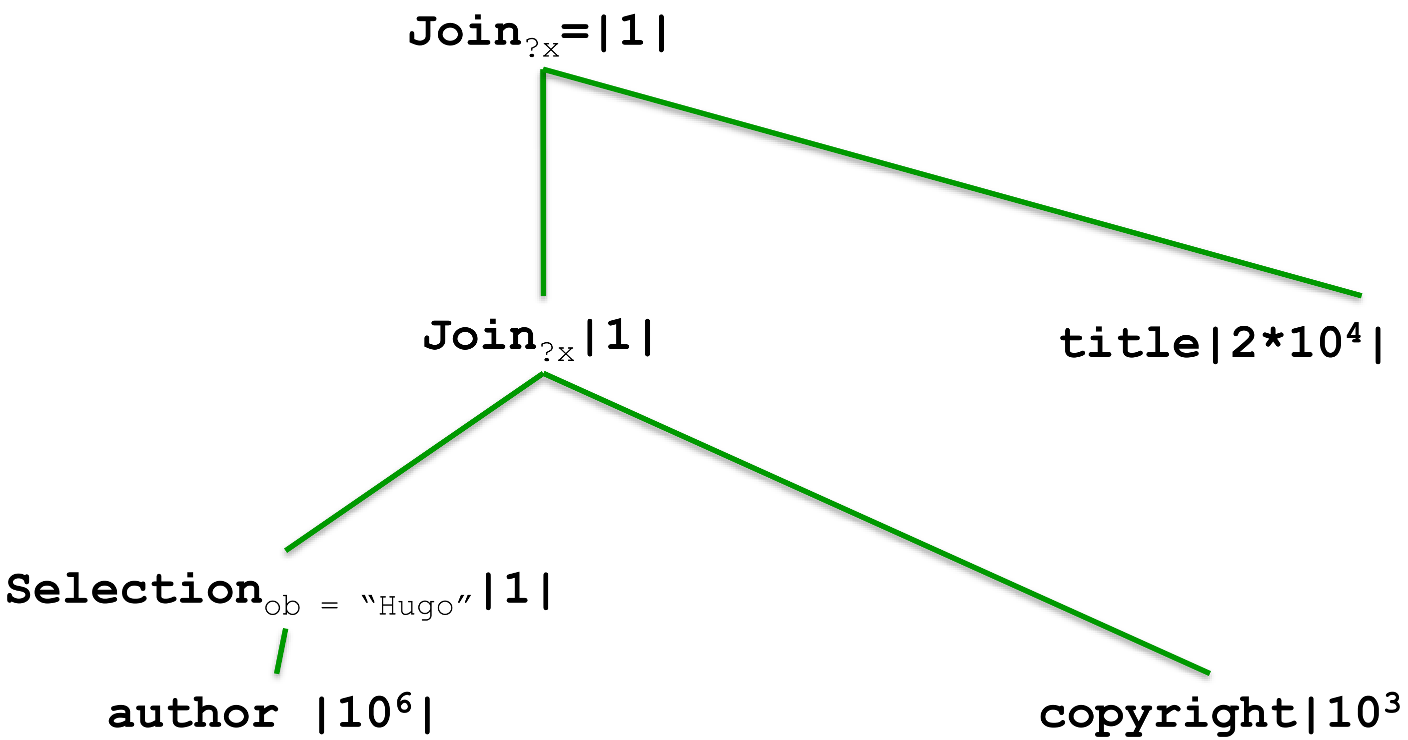
t_1 $\mathbf{10^{-6}}$

t_2 $\mathbf{2*10^{-2}}$

t_3 $\mathbf{10^{-3}}$

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

Plan

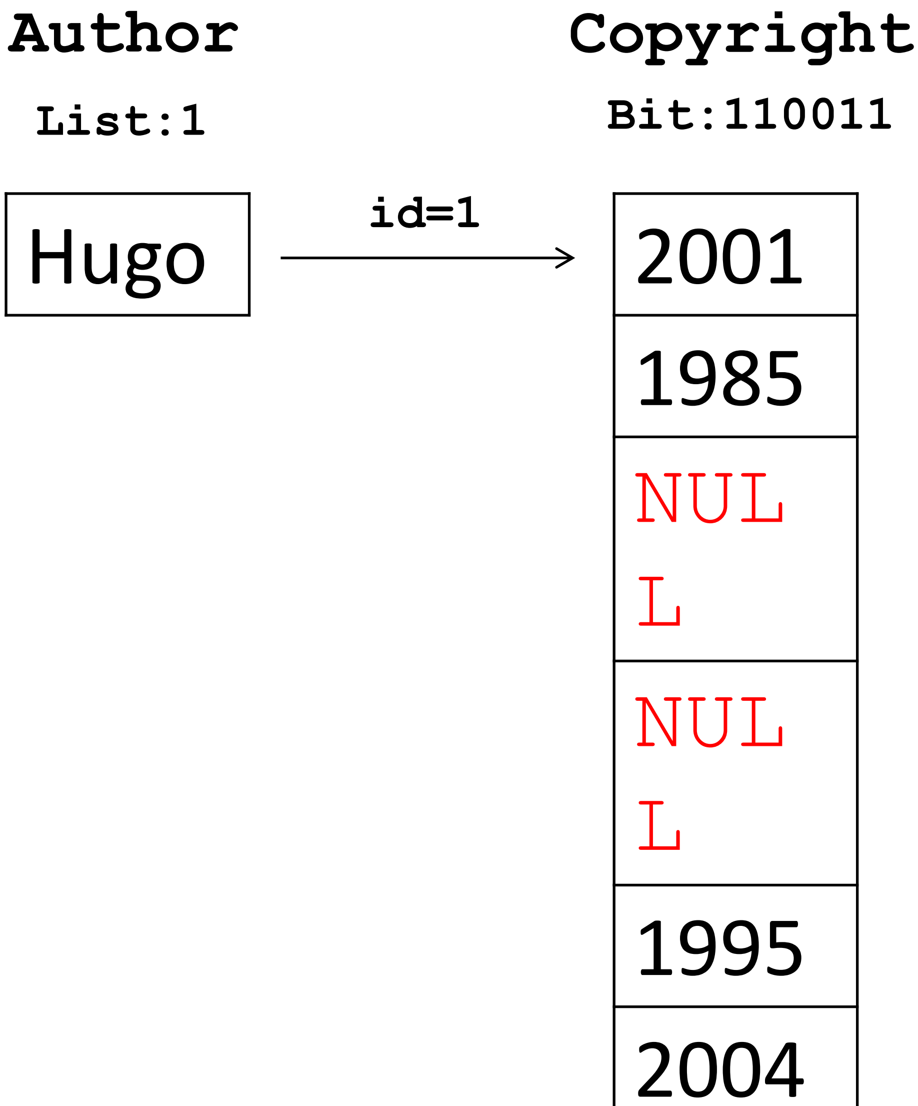


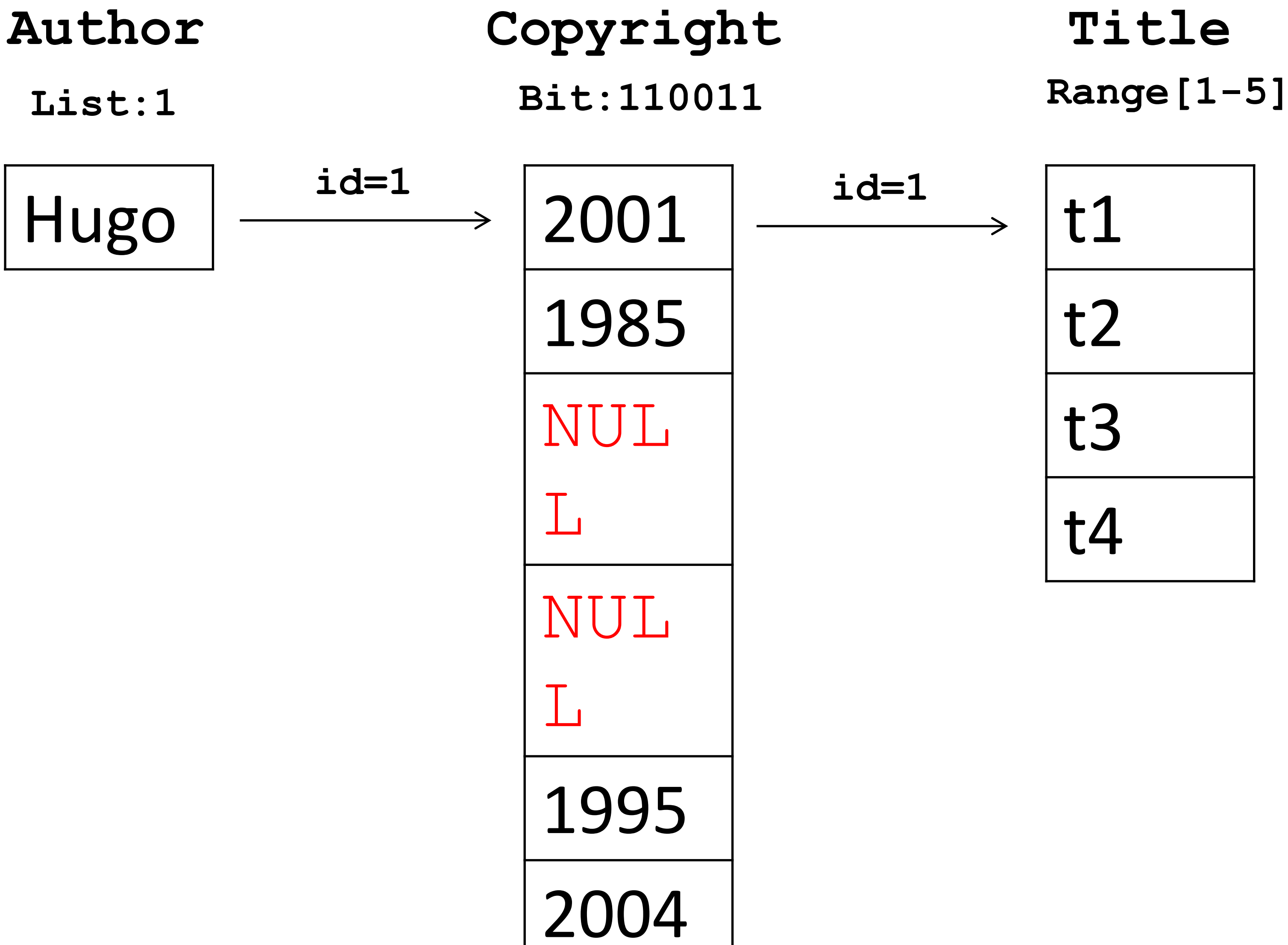
Join?x=|1|
Join?x|1|
title|2*10⁴|
Selectionob = "Hugo"|1|
author |10⁶|
copyright|10³|

Author

List:1

Hugo $\xrightarrow{\texttt{id=1}}$


Author
List:1
Hugo
id=1
Copyright
Bit:110011
2001
1985
NULL
NULL
1995
2004


Author
List:1
Hugo
id=1
Copyright
Bit:110011
2001
1985
NULL
NULL
1995
2004
id=1
Title
Range[1-5]
t1
t2
t3
t4

Query

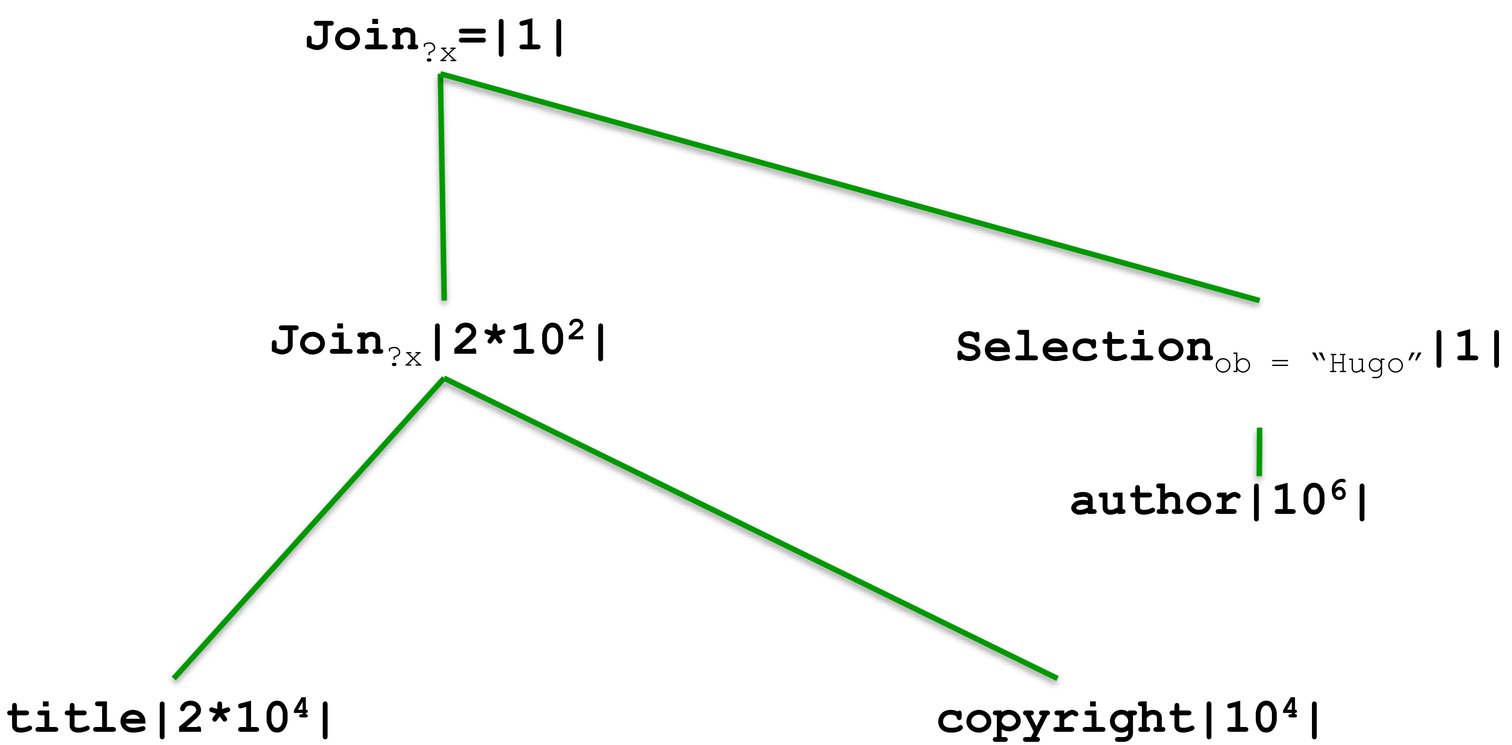
triple pattern selectivity

```
SELECT ?y ?z where {
            ?x, author,  Hugo          .
            ?x, title, ?y              .
            ?x copyright ?z            .
            }
```

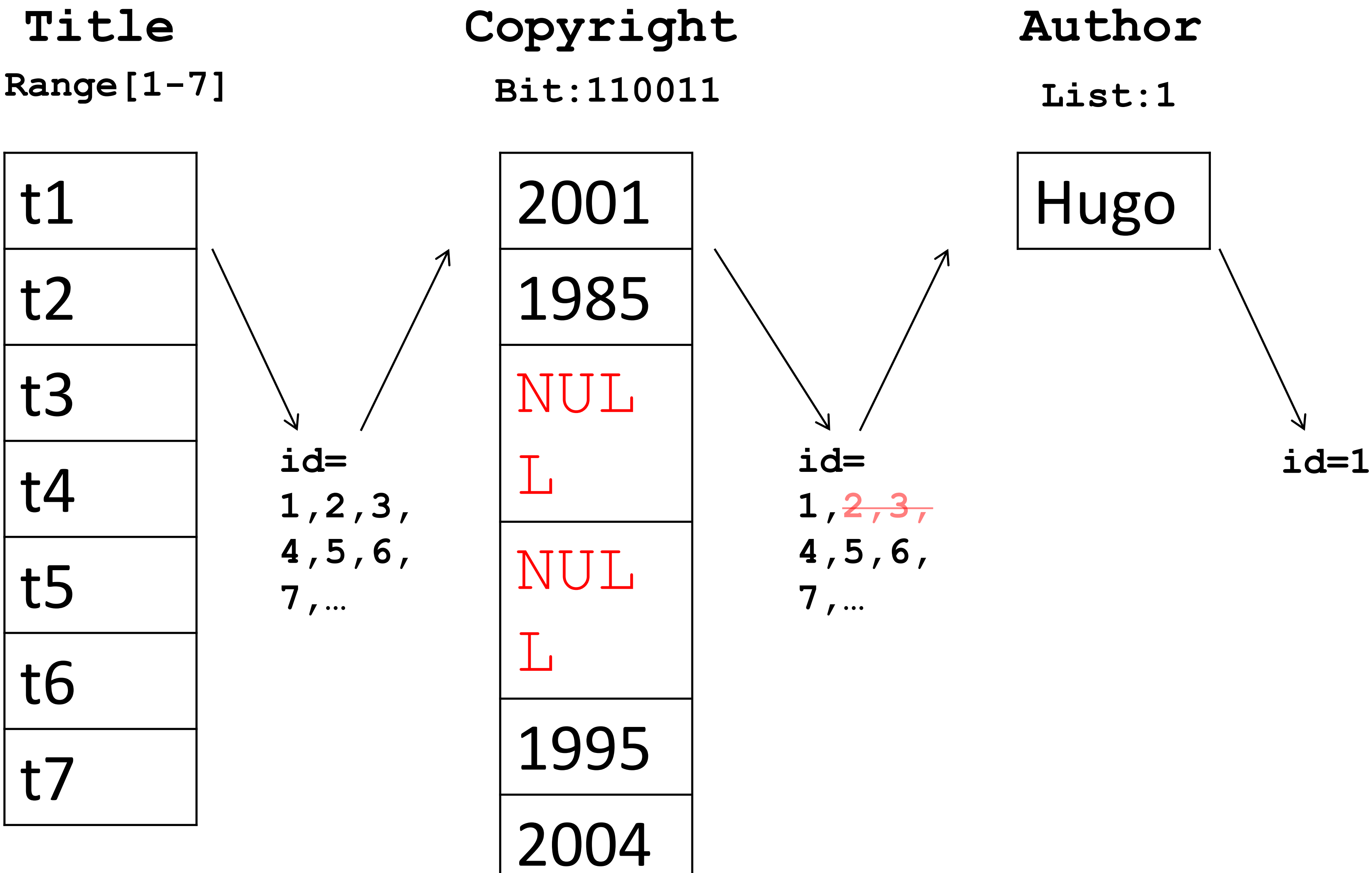
?x, author, Hugo	.	t_1	10^{-6}
?x, title, ?y	.	t_2	$2*10^{-2}$
?x copyright ?z	.	t_3	10^{-3}

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

Plan



Join?x=|1|
Join?x|2*10²|
Selectionob = "Hugo"|1|
author|10⁶|
title|2*10⁴|
copyright|10⁴|


Title
Range[1-7]
t1
t2
t3
t4
t5
t6
t7
id=
1,2,3,
4,5,6,
7,…
Copyright
Bit:110011
2001
1985
NUL
L
NUL
L
1995
2004
id=
1,~~2,3,~~
4,5,6,
7,…
Author
List:1
Hugo
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?

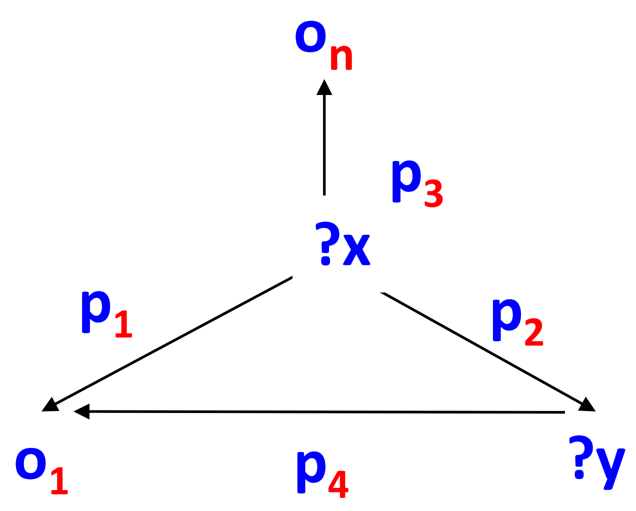
There is no single winner

	RDF-3x	VOS [6.1]	VOS [7.1]	MonetDB	4Store
% 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