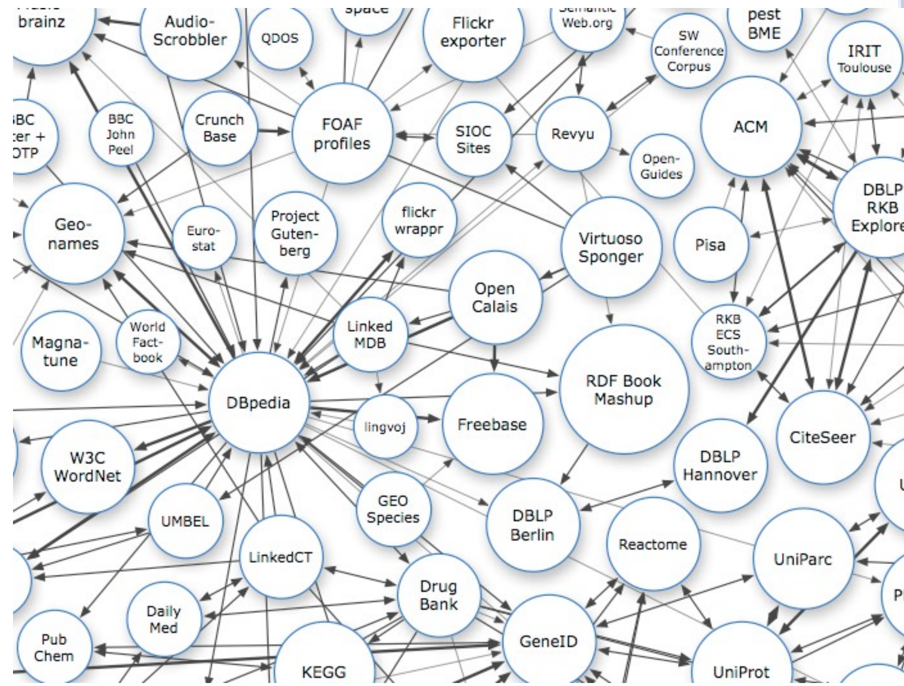
Towards a Social Semantic Space with Linked Data

Luis Daniel Ibáñez[1], Hala Skaf-Molli[1], Pascal Molli[1] and Olivier Corby[2]
[1]GDD – Université de Nantes
[2]Wimmics – INRIA Sophia-Antipolis

kolflow

Context - Task 5

- Goal: design and experiment a social semantic space where humans and smart agents can collaborate to produce knowledge understandable by humans and machines...
- The streams of knowledge being produced are continuous.
- Closest we have now: Linked Data.

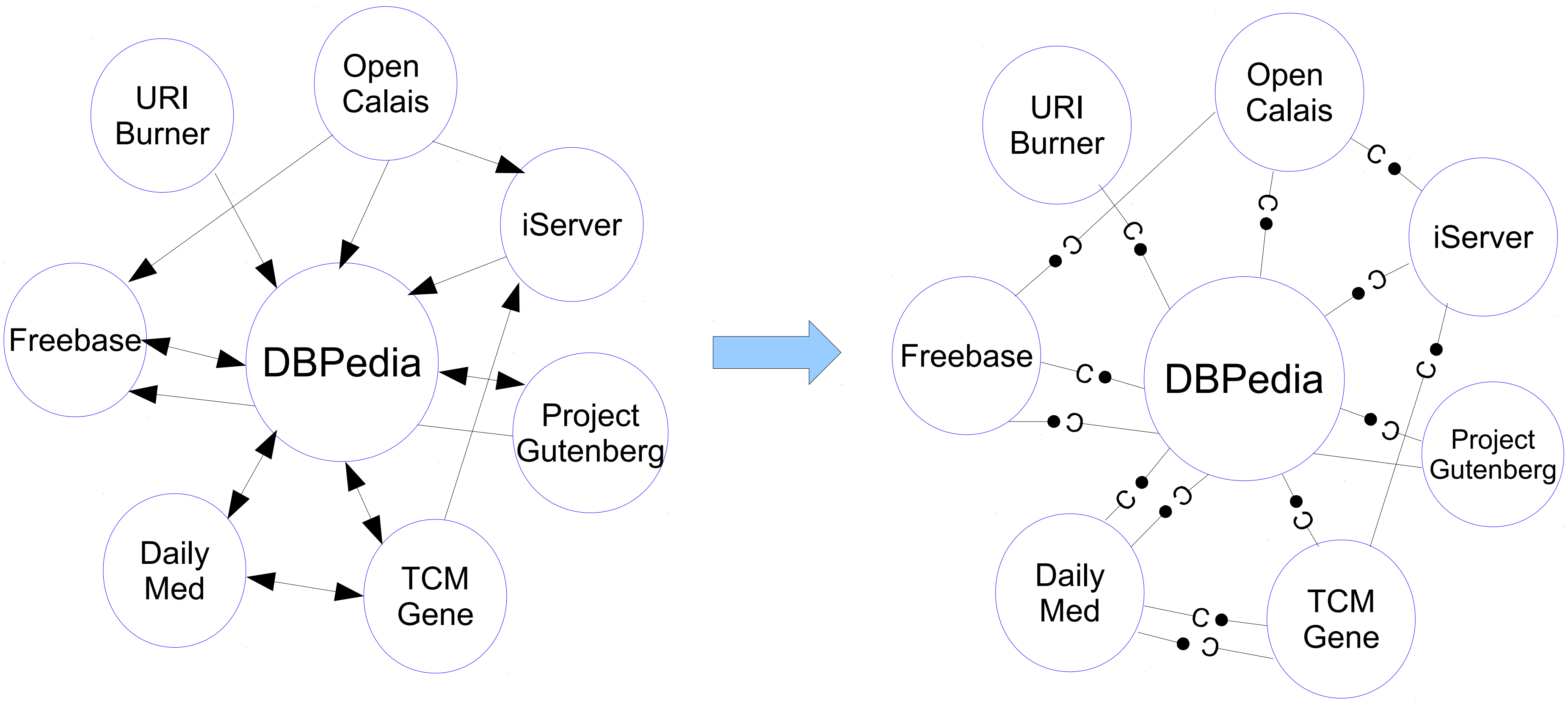


Context – Linked Data as SSS

- Linked Data has semantics, knowledge, humans and machines.
- But is just datasets publishing and interlinking. It is not editable.
  - No edition → no collaboration → no SSS
- How to allow collaborative editing of datasets?
  - Right now, copying is the only general solution

Context – DBPedia went Live

- Continuous stream the inserted and deleted triples...



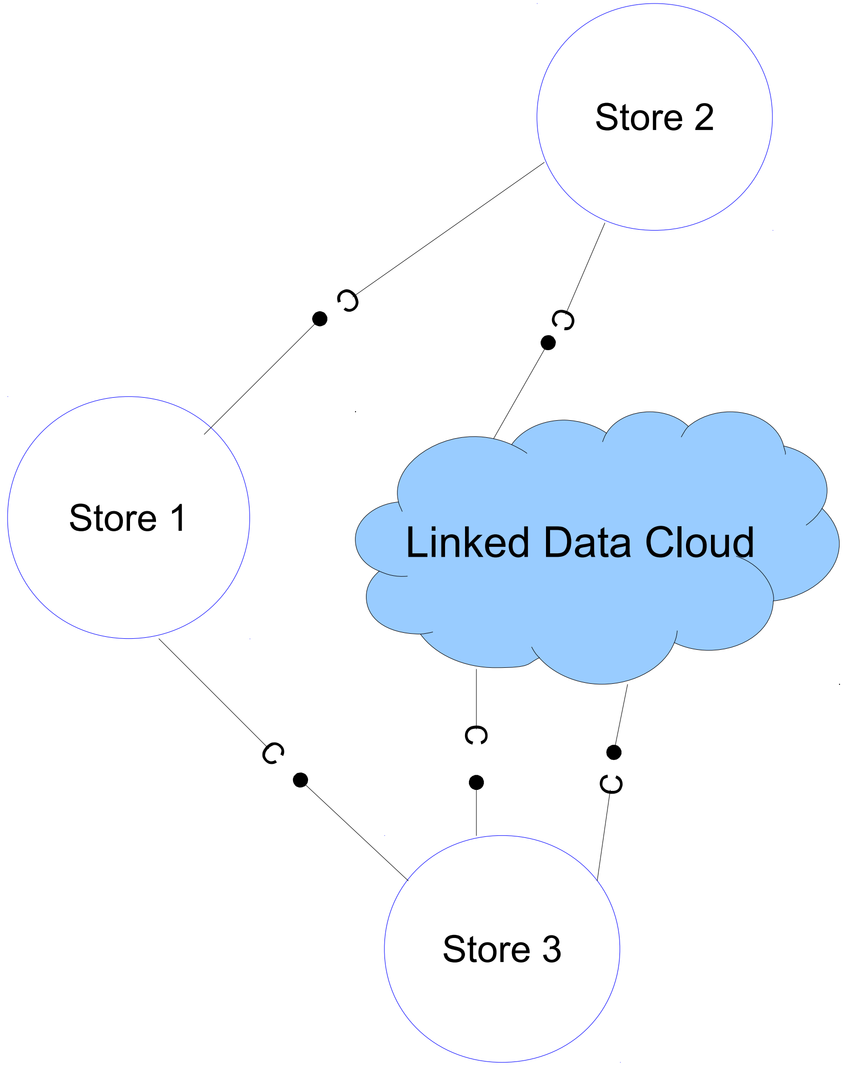
- Linking extends to follow/pull your changes....
- If I follow your changes and you follow mine, we can improve both datasets → Collaboration!

Live Linked Data

- A social network for Linked Data Participants based on a "follow your change" relationship.
- Makes Linked Data Editable
  - Thus, collaborative.
  - From Linked Data 1.0 to 2.0
- Data at each node is fresh.

Live Linked Data

- When participants start updating datasets:
- We don't know who consumes from who...
- Can get my own updates, multiple updates, conflicts...
- What consistency criteria? And how to ensure it?



Live Linked Data

- Allow temporal divergence between replicas $\longrightarrow$ The consistency is eventual.
- Each linked data node:
  - Executes SPARQL Update queries locally.
  - Publishes these operations in “Live Streams”
  - Other nodes consume and re-execute them.
- The system is correct if Convergence, Causality and Intention hold.

SU-Set

payload set S
    initial $\emptyset$
query *lookup* (triple t) : boolean b
    let $b = (\exists u : (t,u) \in S)$
update *insert* (set<triple> T)
    $atSource(T)$
        let $\alpha = unique()$
    $downstream(T, \alpha)$
        let $R = \{(t,\alpha) : t \in T\}$
        $S := S \cup R$
update *delete* (set<triple> T)
    $atSource(T)$
        let $R = \emptyset$
        foreach t in T:
            let $Q = \{(t,u) \,|\, (\exists u : \,|\, (t,u) \in S)\}$
            $R := R \cup Q$
    $downstream(R)$
        // *Causal Reception*
        pre All add(t,u) delivered
        $S := S \setminus R$

Abstract operation is Sparql Update

Same id for all triples in-serted together saves communication

Delete all pairs associa-ted to each triple.
Can be expensive.

What is the price to pay?

- Time Overhead :
  - Adding an id to each element is linear.
  - Selection and lookup is not affected by many pairs with the same triple.
- Round and # of messages Overhead :
  - Convergence after one round, one message per operation $\rightarrow$ Optimal

Validation -Price in space

32 by

Two UUIDs, 16 bytes each

(UUID1, UUID2)

Site identifier

Vector clock

Communication Cost

- DBPedia Live generates one file with triples inserted and one with triples deleted approximately each 10 seconds.
- No pattern operations → No overhead here.
- Many more insertions than deletions
  - Insertions are cheap, they only need one id.
- Many triples per insertion
  - More triples inserted at a time is cheaper.

Communication Cost in DBPedia Live

7 days of streaming
No concurrent insertions

Operation	# of Triples	Communication (MB) No ids	1 id per triple	1 id per operation
21957 Inserts	21762190	3403,4	4469,89	3404,6
21957 Deletes	1755888	238,46	324,5	324,5
	Overhead		31,64%	2,39%

- Under this change rate and insert/delete ratio the overhead is acceptable. Broader complexity analysis submitted to special issue of IJMSO

So far we have...

- A CRDT for RDF-Graph updated with SPARQL 1.1
  - Allows to synchronize semantic stores with eventual consistency.
- Biggest prices to pay are in communication :
  - ID overhead.
  - Causal delivery maintenance overhead.
- Theoretical estimates of this overhead.

Current Work - Implementation

- Test Cases based on SPARQL 1.1 specification developed and coded in JUnit.

Insert (ground triples)

SPARQL Operation:

PREFIX dc: <http://purl.org/dc/elements/1.1/>

INSERT DATA

{ <http://example/book1> dc:title "A new book" ;

dc:creator "A.N.Other" .}

Local Data Before:

@prefix dc: <http://purl.org/dc/elements/1.1/> .

@prefix ns: <http://example.org/ns#> .

[id-1-1,<http://example/book1>, ns:price , 42]

Local Data After:

@prefix dc: <http://purl.org/dc/elements/1.1/> .

@prefix ns: <http://example.org/ns#> .

[id-1-1,<http://example/book1>, ns:price , 42]

[id-1-2,<http://example/book1>, dc:title , “A new book”]

[id-1-2,<http://example/book1>, dc:creator , “A.N.Other”]

1) Two concurrent inserts

Site1 arrival: {id-1-1 -> (

[<http://example/president25>, foaf:givenName ,"William"]

[<http://example/president25> ,foaf:familyName, "McKinley"])}

Site2 arrival: {id-2-1 -> (

[<http://example/president25>, foaf:givenName ,"Will"]

[<http://example/president25> ,foaf:familyName, "McKinley"])}

Data Before:

[id-3-10, <http://example/president25>, foaf:givenName ,"William"]

[id-3-40, <http://example/president25> ,foaf:familyName, "McKinley"]

Data After:

[id-3-10, <http://example/president25>, foaf:givenName ,"William"]

[id-2-1, <http://example/president25>, foaf:givenName ,"Will"]

[id-1-1, <http://example/president25>, foaf:givenName ,"William"]

[id-3-40, <http://example/president25> ,foaf:familyName, "McKinley"]

[id-1-1, <http://example/president25> ,foaf:familyName, "McKinley"]

[id-2-1, <http://example/president25> ,foaf:familyName, "McKinley"]

Implementation

- Alpha Version of SU-Set implemented into Corese
    - Interface tagger, for ID assignation
    - Interface Listener, to log operations, maintain list of neighbors, broadcast, pull, or whatever we want.
- The work on a full version is proposed for Luis' eventual stay at Sophia.

Implementation - Causality

- Vector clocks are traffic cheap, but they require global knowledge of network's membership
  - Too high when members change often =(
- As DBPedia Live publishes an operation log, an AntiEntropy[1] scheme is more suitable.
  - No membership.
  - ID smaller (no vector).
  - Needs more communication.
  - Need to recalculate theoretical overhead.

[1] A. Demers et. al. Epidemic Algorithms for Replicated Database Maintenance. Xerox Palo Alto TechReport 1989, based on earlier version in ACM PDC 1987

Proposed Validation

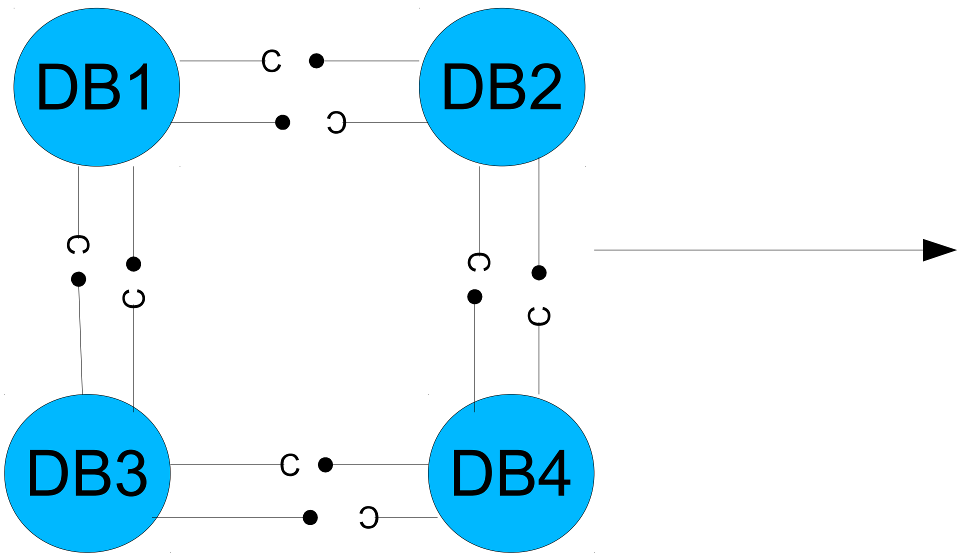
- Is Live Linked Data feasible with real datastores?
- DBPediaLive is by far the worst case in combination of size and change rate...
  - If a LLD full of DBPedias work, normal one should work...

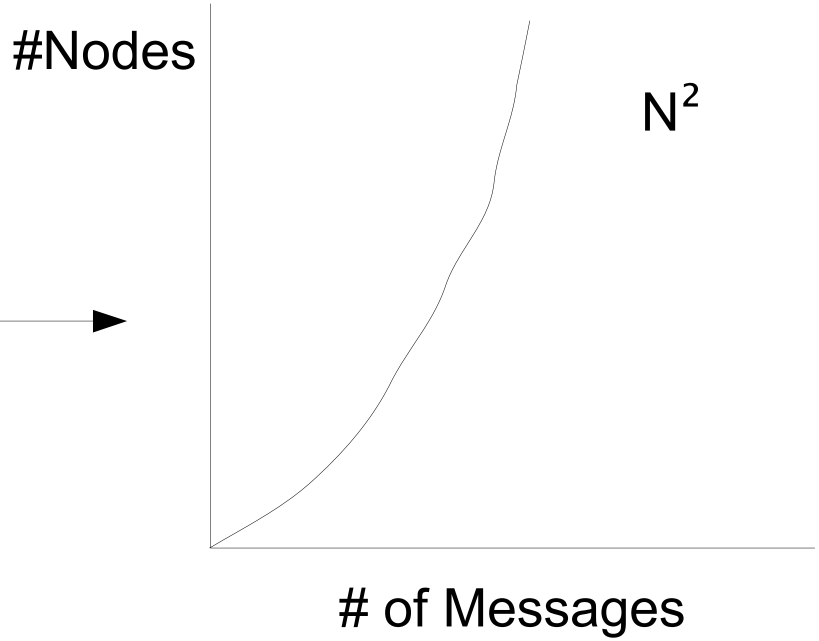
Experimentation – Local Overhead

- Does the new ID breaks or slows down the semantic store?
- Run Berlin (the most used) and DBPedia SPARQL (the one associated to DBPedia) benchmarks over Corese with and without SU-Set. Extra variables are :
  - Probability of concurrent insertion of the same element (Duplicates)
  - Maximum number of duplicates per triple (Number of Nodes)
- Does our “acceptable” time/space overhead computation holds?
- At which values we make the system explode?

Experiments - Communication

- Does our theoretical computation of overhead holds?
- Worst case analysis :
    - Everybody consumes from everybody.
    - All nodes update concurrently.





Experiments - Communication

- What happens in topologies closer to reality?:
  - Social networks (Reciprocal links, scale-free properties)
  - Twitter-like (Non-reciprocal, scale-free properties except for most popular nodes)
  - The Linked Open Data Graph (LOD) graph itself.
- Assumptions:
  - No failures
  - No ontology conflicts (another task for that)

Experiments - Variables

- Number of Semantic Stores (SS)
- Number of triples at each SS ← DBPedia's 1Billion.
- Size of unique id ← Depends on Causality protocol.
- Change rate and ratio of inserted/deleted triples for each node ← DBPedia figures (0.0006% per minute and 12:1), but interesting to measure (DYLDO initiative).
    - Our previous seven days of measuring are enough?

Experiments - Variables

- Operation “chunk” size ← Bigger chunk less overhead, but less freshness, we take DBPedia values (Avg. 170kb).
- Duplicate percentage and maximum number.
- Measure traffic and number of messages :
  - Does our theoretical estimates hold?
  - How much is due to causal delivery maintenance? - We will need to recompute that. It is the heaviest part? We will need something else?

Future Work

- Can we construct a CRDT for RDF without the costly causal delivery (or a weaker condition)?
- Can we implement the pattern operations to reduce traffic?
- Or can we prove we can't?
- How about querying? (Sync & Search?)

Experimentation – Sync & Search

- Goal : Compare Sync & Search with Warehousing and Distributed Searching (FedX) → Who is better in which case ?
- New parameter: Number of queries per time unit.
- We expect
  - Sync & Search better when there are many queries and moderate change rate.
  - Warehouse better when size of stores is manageable and low change rate.
  - Distributed search better when query number low.

Experimentation – Sync & Search