RDFox — A Modern Materialisation-Based RDF System

Boris Motik

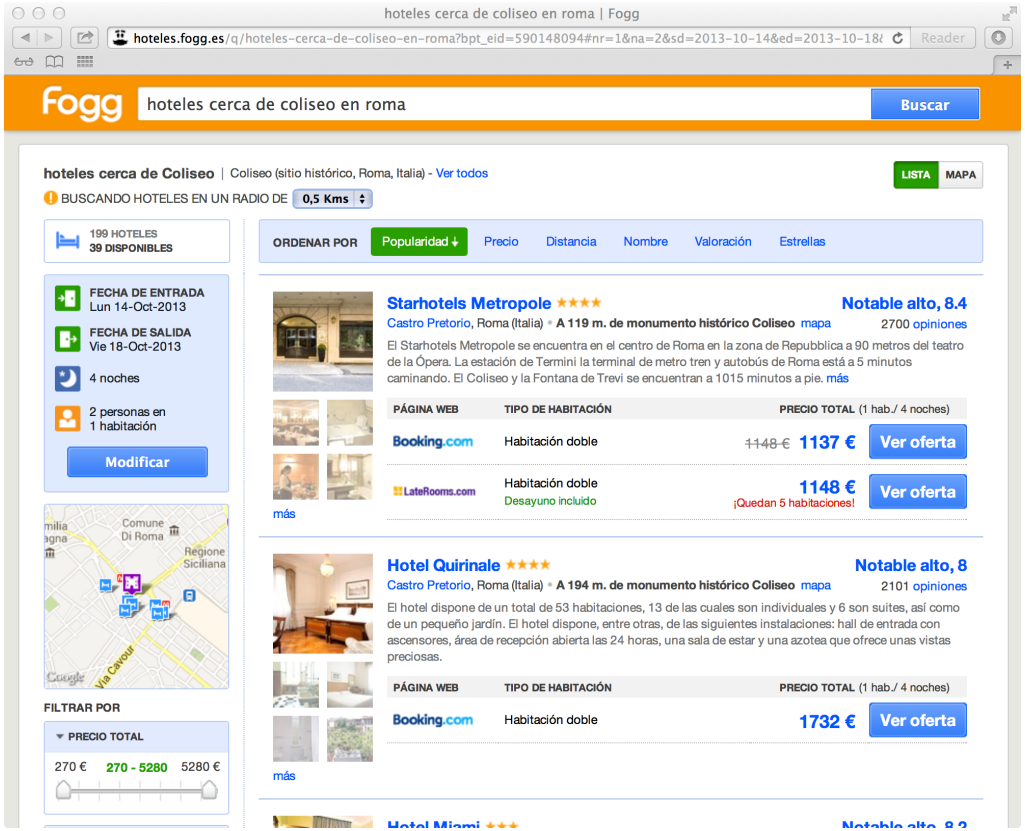
University of Oxford

November 9, 2015

Table of Contents

Table of Contents

Application: Search in Tourism (SkyScanner)



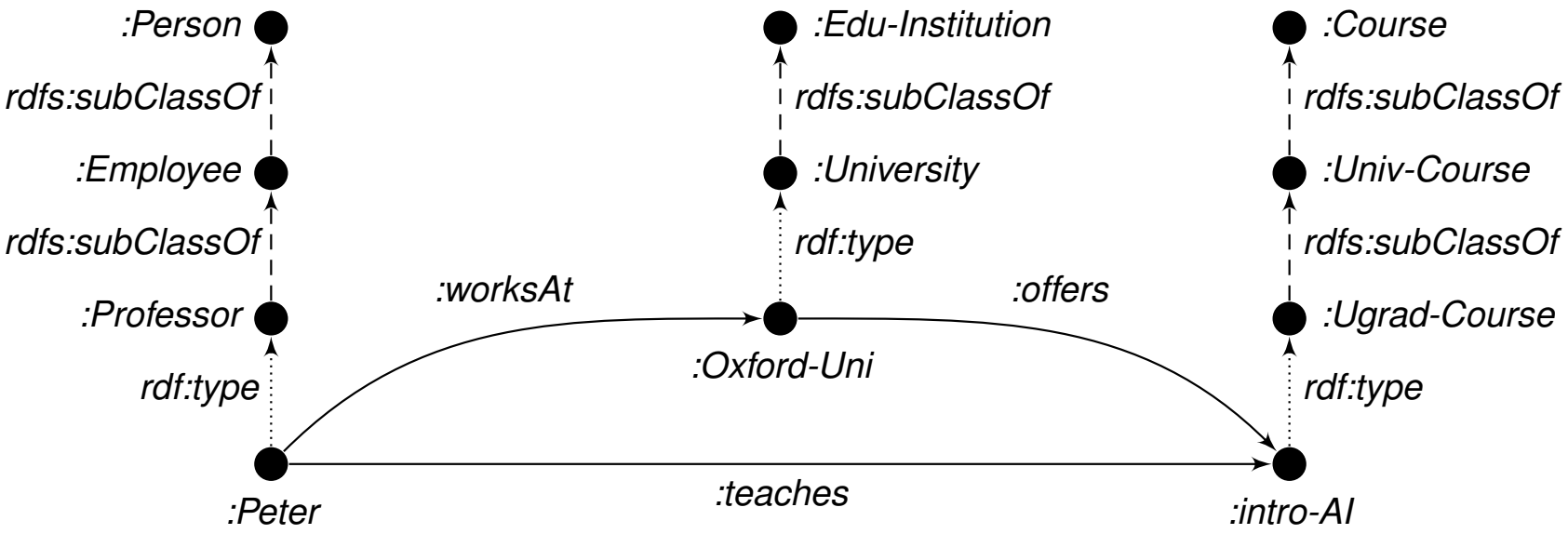
- Goal: search for hotels/flights/trips using natural language
- Need to represent large amounts of heterogeneous data
- Query for accommodation should include hotels, B&Bs, . . .

Application: Context-Aware Mobile Services (Samsung)

- Use sensors (WiFi, GPS, . . .) to identify the context
  - E.g., ‘at home’, ‘in a shop’, ‘with a friend’ . . .
- Adapt behaviour depending on the context
  - ‘If with a friend who has birthday, remind to congratulate’

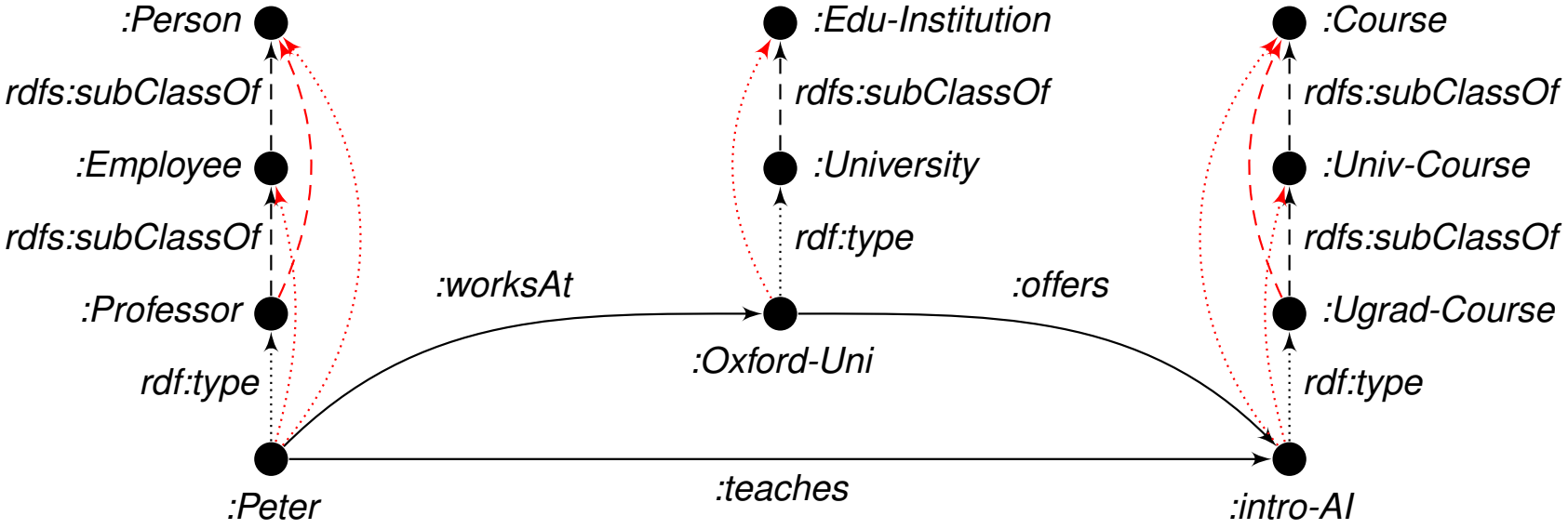
- Declaratively describe contexts and adaptations
  - E.g., ‘If can see home Wifi, then context is “at home”’
- Interpret all rules in real-time using reasoning
- Main benefit: declarative, rather than procedural

Representing Semistructured Data with RDF



- Nodes ⇒ resources; arcs ⇒ triples
- Similar to semantic networks and semistructured data models
- Special arcs (e.g., *rdf:type* and *rdfs:subClassOf*) provide structure to the data
- Inference ⇒ explicating information implicit in the data

Representing Semistructured Data with RDF



- Nodes ⇒ resources; arcs ⇒ triples
- Similar to semantic networks and semistructured data models
- Special arcs (e.g., *rdf:type* and *rdfs:subClassOf*) provide structure to the data
- Inference ⇒ explicating information implicit in the data

Capturing Entailment Rules of RDFS in Datalog

$$\langle ?X, \textit{rdfs:subClassOf}, ?Z\rangle \leftarrow \langle ?X, \textit{rdfs:subClassOf}, ?Y\rangle \wedge \langle ?Y, \textit{rdfs:subClassOf}, ?Z\rangle$$
$$\langle ?X, \textit{rdfs:subPropertyOf}, ?Z\rangle \leftarrow \langle ?X, \textit{rdfs:subPropertyOf}, ?Y\rangle \wedge \langle ?Y, \textit{rdfs:subPropertyOf}, ?Z\rangle$$
$$\langle ?X, \textit{rdf:type}, ?Z\rangle \leftarrow \langle ?X, \textit{rdf:type}, ?Y\rangle \wedge \langle ?Y, \textit{rdfs:subClassOf}, ?Z\rangle$$
$$\langle ?X, ?W, ?Z\rangle \leftarrow \langle ?X, ?Y, ?Z\rangle \wedge \langle ?Y, \textit{rdfs:subPropertyOf}, ?W\rangle$$
$$\langle ?X, \textit{rdf:type}, ?Z\rangle \leftarrow \langle ?X, ?W, ?Y\rangle \wedge \langle ?W, \textit{rdfs:domain}, ?Z\rangle$$
$$\langle ?Y, \textit{rdf:type}, ?Z\rangle \leftarrow \langle ?X, ?W, ?Y\rangle \wedge \langle ?W, \textit{rdfs:range}, ?Z\rangle$$

Goals

- Develop techniques for materialisation of datalog programs on RDF data

Goals

- Develop techniques for materialisation of datalog programs on RDF data

- Current trends in databases and knowledge-based systems:
  - Price of RAM keeps falling
    - 128 GB is routine, systems with 1 TB are emerging
    - In-memory databases: SAP's HANA, Oracle's TimesTen, YarcData's Urika
  - Materialisation is computationally intensive $\Rightarrow$ natural to parallelise
    - Mid-range laptops have 4 cores, servers with 16 cores are routine

Goals

- Develop techniques for materialisation of datalog programs on RDF data in main-memory, multicore systems
  - Implemented in the RDFox system
  - `http://www.cs.ox.ac.uk/isg/tools/RDFox/`
- Current trends in databases and knowledge-based systems:
  - Price of RAM keeps falling
    - 128 GB is routine, systems with 1 TB are emerging
    - In-memory databases: SAP's HANA, Oracle's TimesTen, YarcData's Urika
  - Materialisation is computationally intensive $\Rightarrow$ natural to parallelise
    - Mid-range laptops have 4 cores, servers with 16 cores are routine

Table of Contents

Existing Approaches to Parallel Materialisation

- Interquery parallelism: run independent rules in parallel
  - Degree of parallelism limited by the number of independent rules
  - $\Rightarrow$ does not distribute workload to cores evenly
- Intraquery parallelism
  - Partition rule instantiations to N threads
    - E.g., constrain the body of rules evaluated by thread i to ($x \bmod N = i$)
    - $\Rightarrow$ Static partitioning may not distribute workload well due to data skew
    - $\Rightarrow$ Dynamic partitioning may incur an overhead due to load balancing
  - Parallelise join computation
    - Hash-partition data into blocks, compute the join for each block independently
    - $\Rightarrow$ Hash tables keep being constantly recomputed
    - Sort-merge join requires constant data reordering
- Goal: distribute workload to threads evenly and with minimum overhead

Interleaving Querying with Updates

- Efficient query evaluation requires indexes
  - Crucial for elimination of duplicate triples ⇒ ensures termination
  - Usually sorted (and clustered) to allow for merge joins
  - Hash indexes can also be used
  - Individual (i.e., not bulk) index updates are inefficient
- Materialisation interleaves . . .
  - . . . querying (during evaluation of rule bodies)
  - . . . updates (during updates of derived facts)
- ⇒ Data storage should support indexes and efficient parallel updates

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery:

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(a)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(a)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(b)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(b)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(a,y)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(c)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: A(c)

Solution Part I: Algorithm

$\Rightarrow$

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(b,y)

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)
A(f)
A(g)

$\Rightarrow$

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(c,y)

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)
A(f)
A(g)

$\Rightarrow$

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(d,y)

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)
A(f)
A(g)

$\Rightarrow$

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(e,y)

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)
A(f)
A(g)

$\Rightarrow$

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(f,y)

Solution Part I: Algorithm

R(a,b)
R(a,c)
R(b,d)
R(b,e)
A(a)
R(c,f)
R(c,g)
A(b)
A(c)
A(d)
A(e)
A(f)
⇒ A(g)

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

- For each fact:
  1. Match the fact to all body atoms to obtain subqueries
  2. Evaluate subqueries w.r.t. all previous facts
  3. Add results to the table

Current subquery: R(g,y)

Parallelising Computation

- Each thread extracts facts and evaluates subqueries independently
- The number of subqueries is determined by the number of facts
  - ensures in practice that threads are equally loaded
- Requires no thread synchronisation
- $\Rightarrow$ We partition rule instances dynamically and with little overhead

When Parallelisation Fails

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

$$R(a, b), \quad R(b, c), \quad R(c, d), \quad R(d, e), \quad A(a)$$

Parallelising Computation

- Each thread extracts facts and evaluates subqueries independently
- The number of subqueries is determined by the number of facts
  - ensures in practice that threads are equally loaded
- Requires no thread synchronisation
- $\Rightarrow$ We partition rule instances dynamically and with little overhead

When Parallelisation Fails

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

$$R(a, b), \quad R(b, c), \quad R(c, d), \quad R(d, e), \quad A(a), \quad A(b)$$

Parallelising Computation

- Each thread extracts facts and evaluates subqueries independently
- The number of subqueries is determined by the number of facts
  - ensures in practice that threads are equally loaded
- Requires no thread synchronisation
- $\Rightarrow$ We partition rule instances dynamically and with little overhead

When Parallelisation Fails

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

$$R(a, b), \quad R(b, c), \quad R(c, d), \quad R(d, e), \quad A(a), \quad A(b), \quad A(c)$$

Parallelising Computation

- Each thread extracts facts and evaluates subqueries independently
- The number of subqueries is determined by the number of facts
  - ensures in practice that threads are equally loaded
- Requires no thread synchronisation
- $\Rightarrow$ We partition rule instances dynamically and with little overhead

When Parallelisation Fails

$$A(x) \wedge R(x, y) \rightarrow A(y)$$

$$R(a, b), \quad R(b, c), \quad R(c, d), \quad R(d, e), \quad A(a), \quad A(b), \quad A(c), \quad A(d)$$

Parallelising Computation

- Each thread extracts facts and evaluates subqueries independently
- The number of subqueries is determined by the number of facts
  - ensures in practice that threads are equally loaded
- Requires no thread synchronisation
- $\Rightarrow$ We partition rule instances dynamically and with little overhead

When Parallelisation Fails

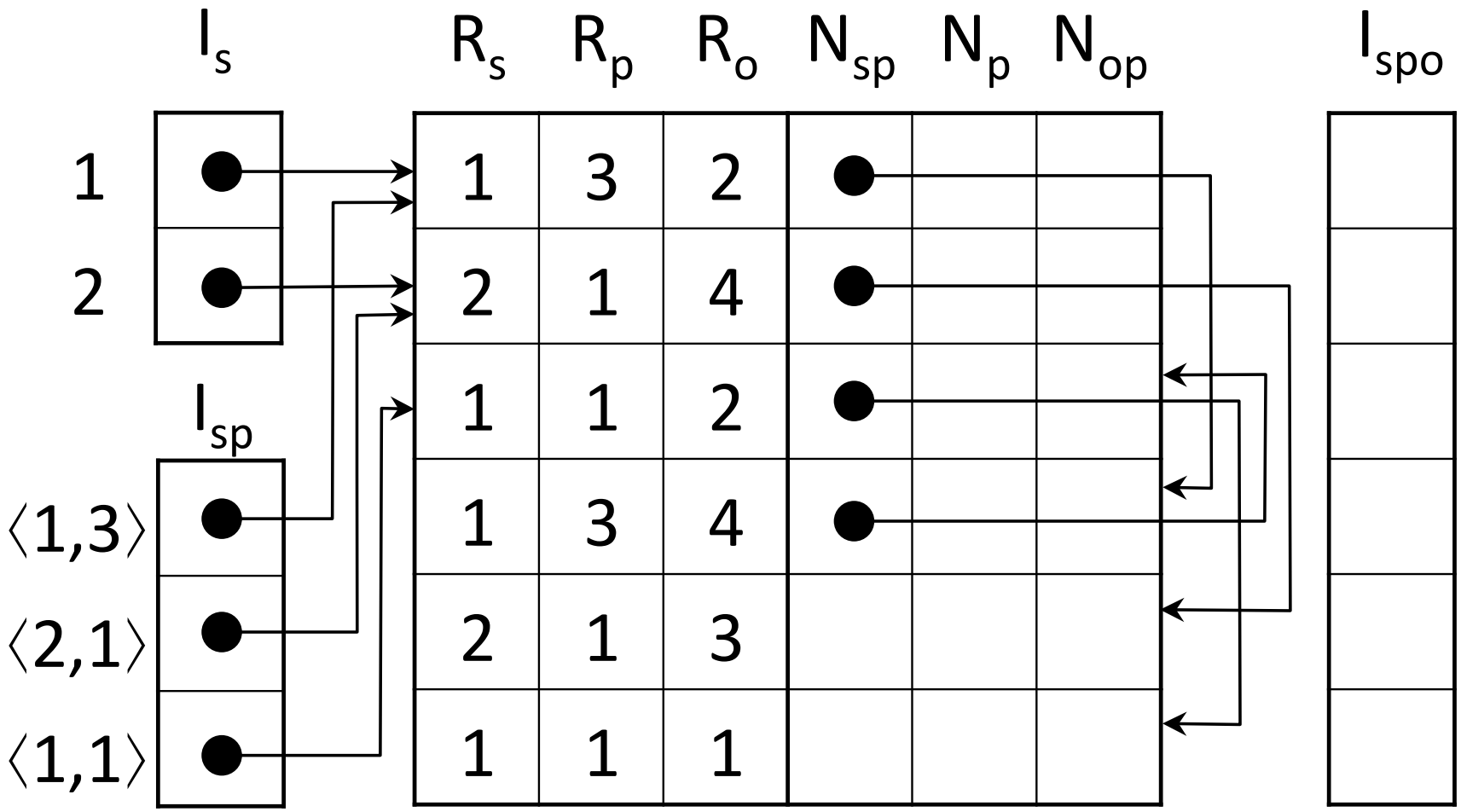
$$A(x) \wedge R(x, y) \rightarrow A(y)$$

$$R(a, b), \quad R(b, c), \quad R(c, d), \quad R(d, e), \quad A(a), \quad A(b), \quad A(c), \quad A(d), \quad A(e)$$

Solution Part II: Indexing RDF Data in Main Memory

- Indexing requirement: retrieve matches to atoms $\langle t_1, t_2, t_3 \rangle$ with t_i terms
  - Eight different binding patterns in total
- Add each triple into three singly-linked lists
  - *sp*-list collects triples with the same subject grouped by predicate
  - *po*-list collects triples with the same predicate grouped by object
  - *os*-list collects triples with the same object grouped by subject
- Index entry points into these lists
  - I_s maps each s to the first triple of the relevant *sp*-list
  - I_{sp} maps each $\langle s, p \rangle$ to the first triple of the relevant *sp*-list containing p
  - I_p and I_{po}, and I_o and I_{os} are analogous
  - I_{spo} indexes triples by all components $\Rightarrow$ used for duplicate elimination
- Optimisation: binding pattern $\langle s, ?Y, o \rangle$ is rare
  - Use *sp*-, *op*-, and *p*-list
  - Only six indexes: I_s, I_{sp}, I_o, I_{op}, I_p, I_{spo}
  - To answer $\langle s, ?X, o \rangle$, use I_s or I_o (depending on size)

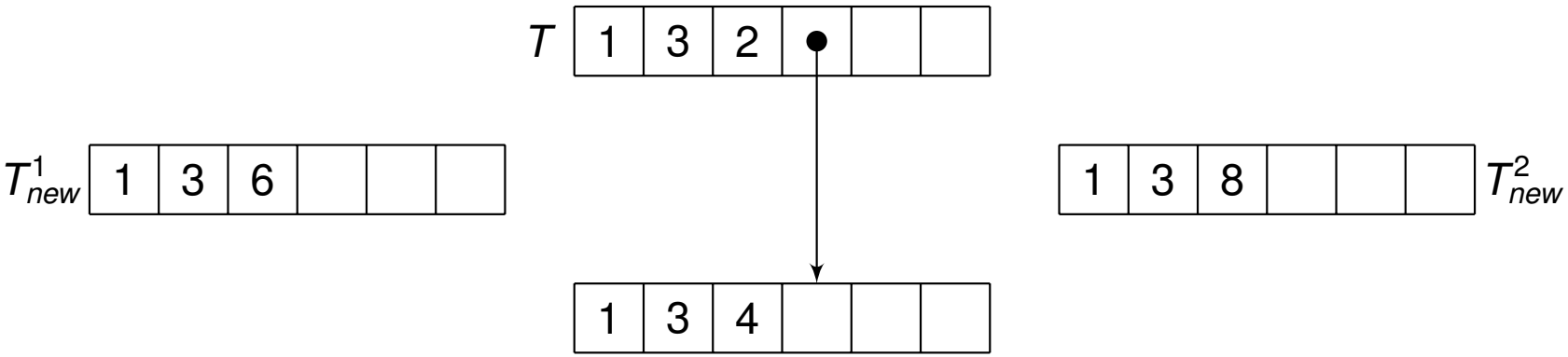
Data Structure



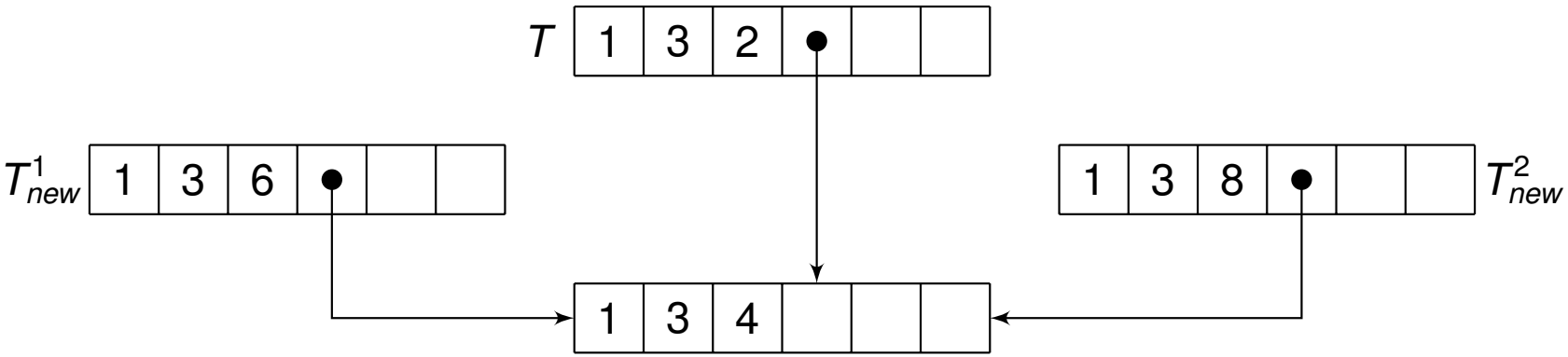
'Mostly' Lock-Free Updates

- Lock-based programming
  - Main benefit: simplicity, easy to ensure linearisability
  - Main problem: susceptible to thread scheduling
    - A thread acquires a lock and goes to sleep $\Rightarrow$ block progress of all other threads
    - Can happen due to swapping, causes priority inversion
- Lock-free programming
  - At all time, at least one thread makes progress
  - Commonly implemented using compare-and-set: CAS(loc, exp, new)
    - Load the value stored of location loc into temporary variable old
    - Store new into location loc if $old = exp$
    - Hardware ensures atomicity
  - A thread can wait indefinitely (e.g., CAS may keep failing)
  - (Unlike wait-free programming: each thread progresses after a fixed amount of time)
- Complete lock-freedom can be costly $\Rightarrow$ we resort to locks occasionally
  - 'Mostly' lock-free

Lock-Free Insertion into Lists

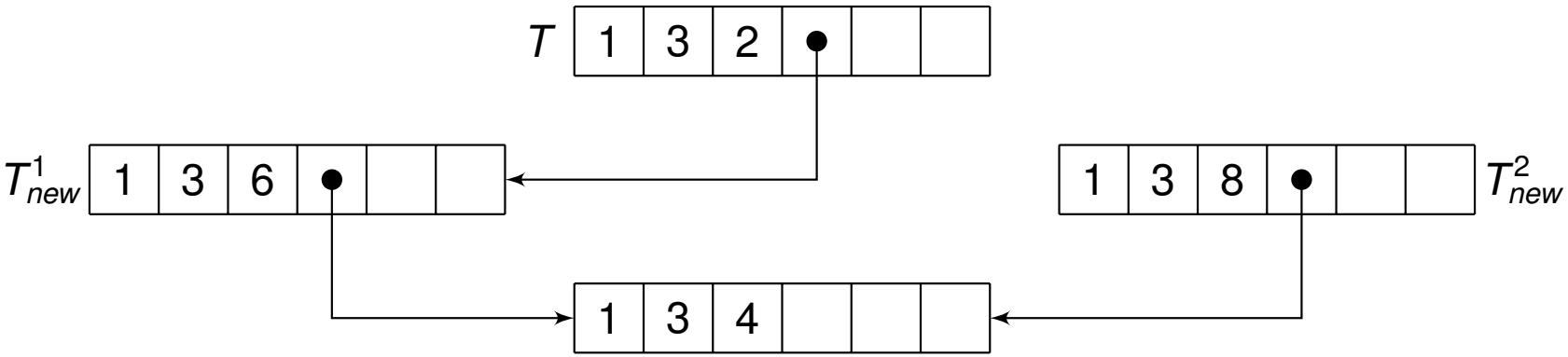


Lock-Free Insertion into Lists



- $T^1_{next} = T.N_{sp}$
- $T^1_{new}.N_{sp} = T^1_{next}$
- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$

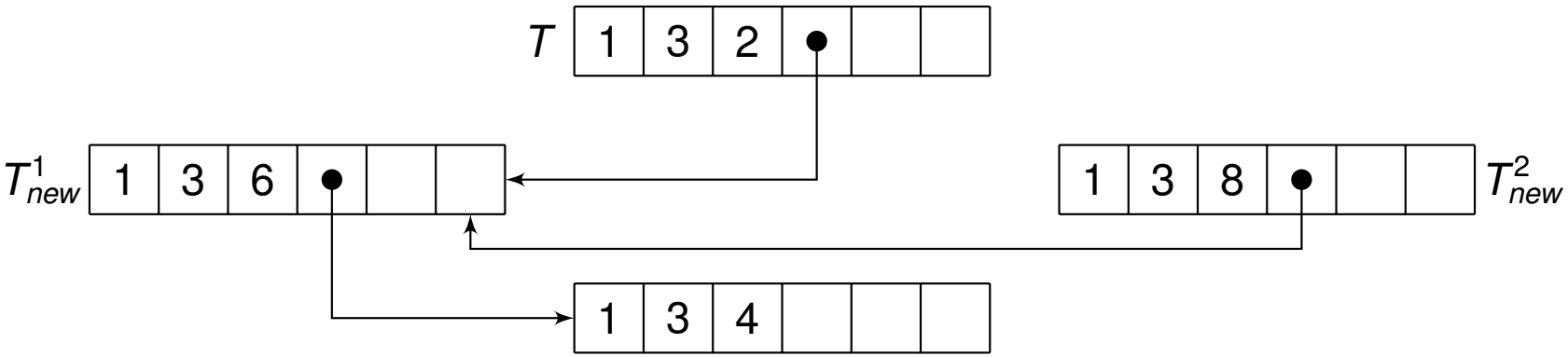
Lock-Free Insertion into Lists



- $T^1_{next} = T.N_{sp}$
- $T^1_{new}.N_{sp} = T^1_{next}$
- CAS($T.N_{sp}$, T^1_{next}, T^1_{new})
- Succeeds!

- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$
- CAS($T.N_{sp}$, T^2_{next}, T^2_{new})
- Fails so we repeat the process

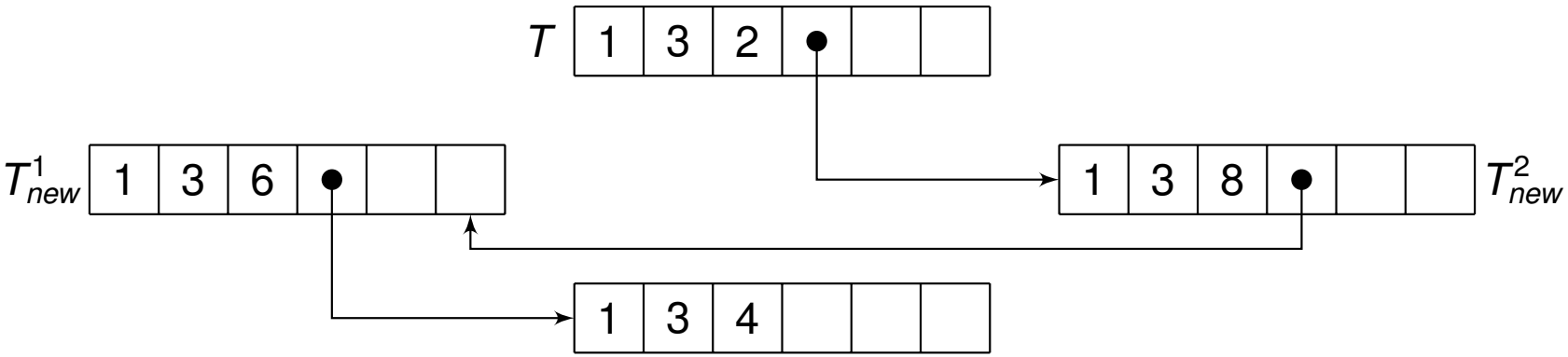
Lock-Free Insertion into Lists



- $T^1_{next} = T.N_{sp}$
- $T^1_{new}.N_{sp} = T^1_{next}$
- CAS($T.N_{sp}$, T^1_{next}, T^1_{new})
- Succeeds!

- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$
- CAS($T.N_{sp}$, T^2_{next}, T^2_{new})
- Fails so we repeat the process
- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$

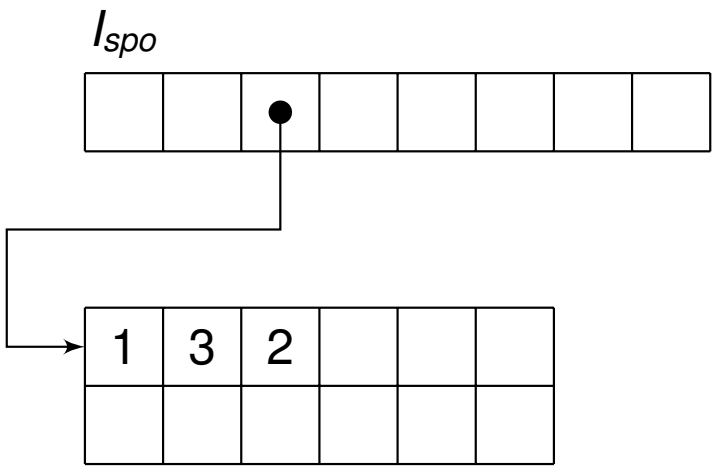
Lock-Free Insertion into Lists



- $T^1_{next} = T.N_{sp}$
- $T^1_{new}.N_{sp} = T^1_{next}$
- CAS($T.N_{sp}$, T^1_{next}, T^1_{new})
- Succeeds!

- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$
- CAS($T.N_{sp}$, T^2_{next}, T^2_{new})
- Fails so we repeat the process
- $T^2_{next} = T.N_{sp}$
- $T^2_{new}.N_{sp} = T^2_{next}$
- CAS($T.N_{sp}$, T^2_{next}, T^2_{new})
- Succeeds!

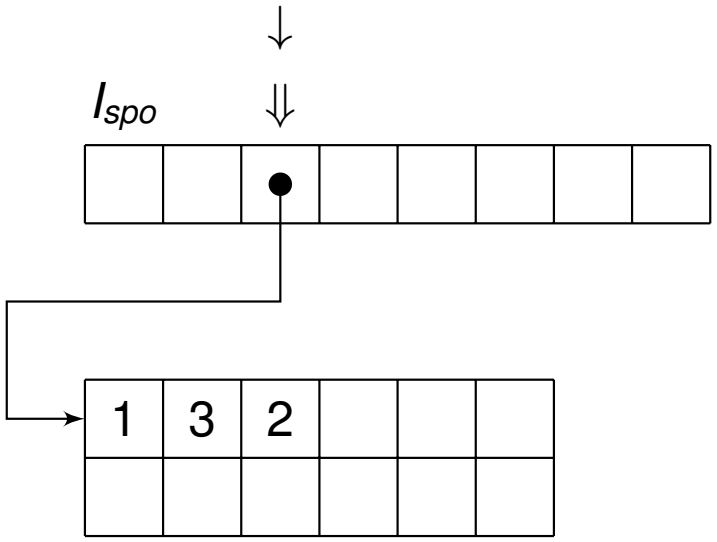
'Mostly' Lock-Free Insertion into I_{spo}



Add $\langle 1, 3, 6 \rangle$:

Add $\langle 1, 3, 6 \rangle$:

'Mostly' Lock-Free Insertion into I_{spo}



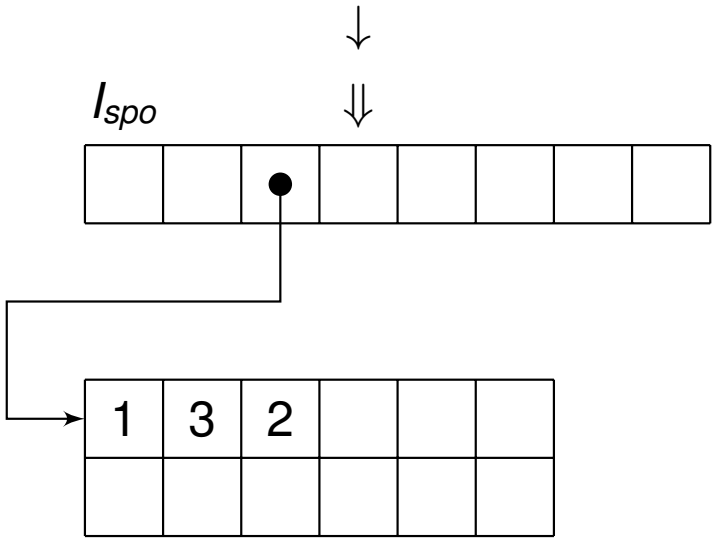
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$

'Mostly' Lock-Free Insertion into I_{spo}



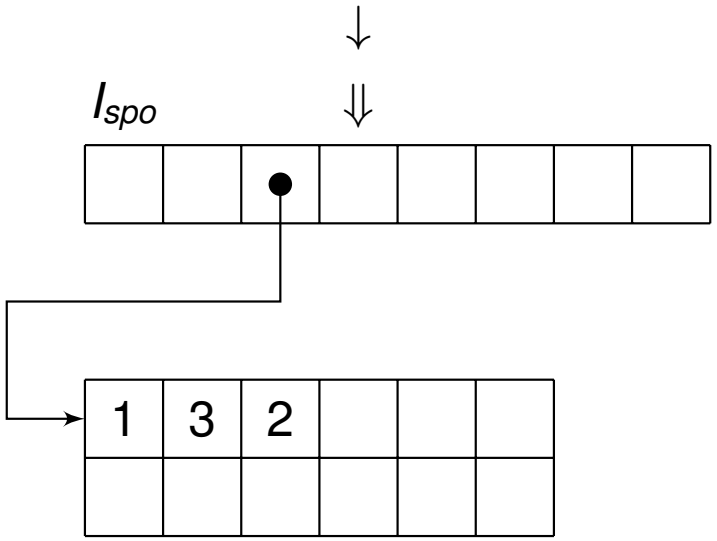
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance

'Mostly' Lock-Free Insertion into I_{spo}



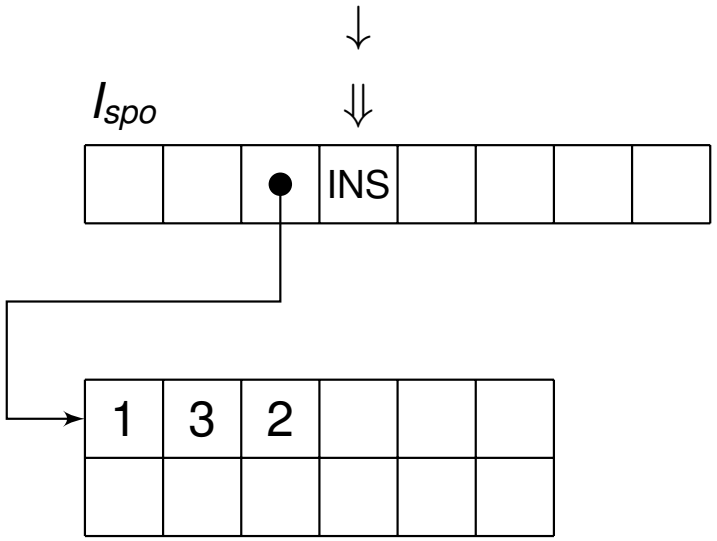
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS

'Mostly' Lock-Free Insertion into I_{spo}



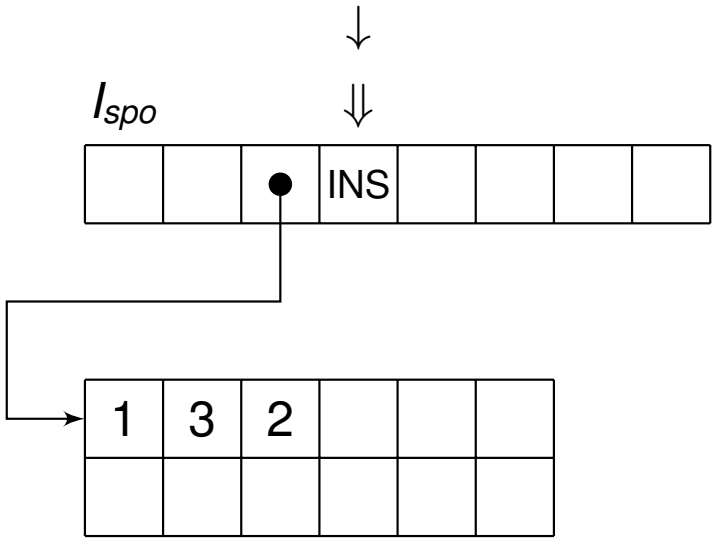
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat

'Mostly' Lock-Free Insertion into I_{spo}



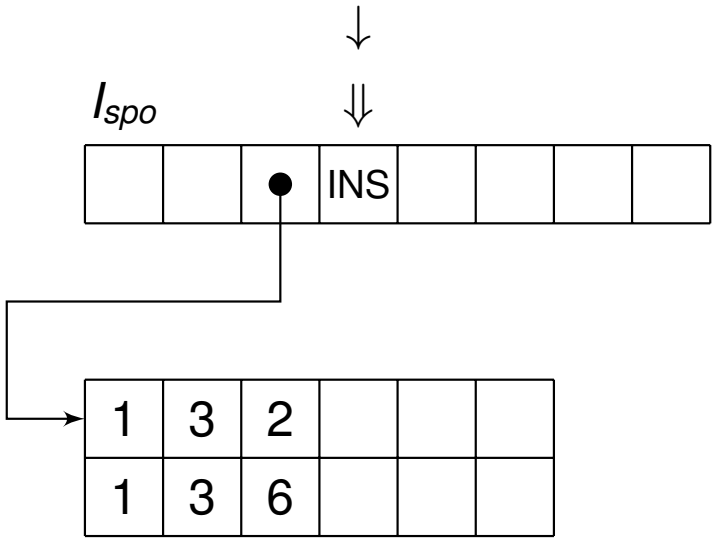
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat
- Bucket locked, so wait

'Mostly' Lock-Free Insertion into I_{spo}



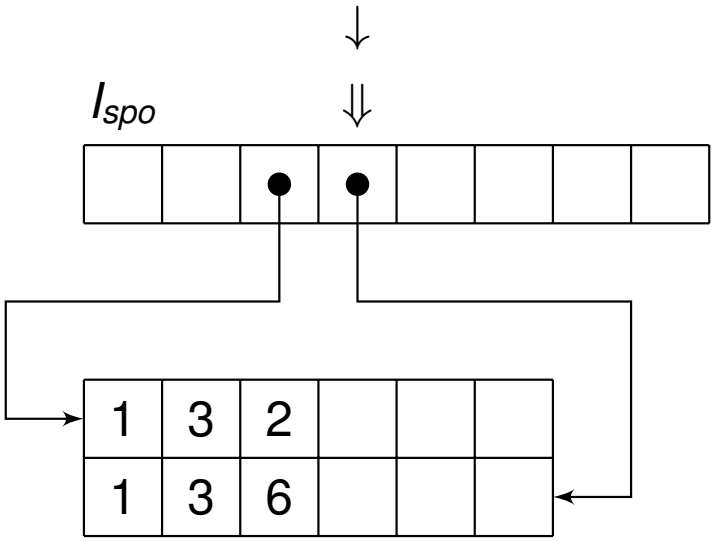
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!
- Allocate triple

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat
- Bucket locked, so wait

'Mostly' Lock-Free Insertion into I_{spo}



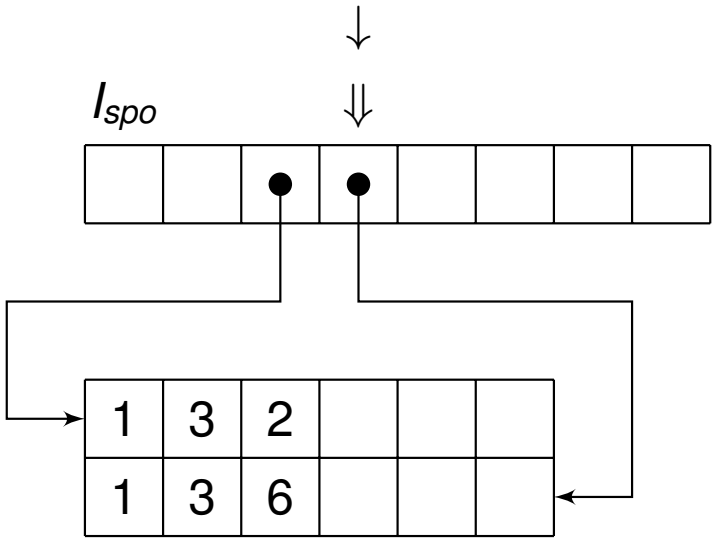
Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!
- Allocate triple
- Update bucket (unlock) and terminate

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat
- Bucket locked, so wait

'Mostly' Lock-Free Insertion into I_{spo}



Add $\langle 1, 3, 6 \rangle$:

- Determine index $\Downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!
- Allocate triple
- Update bucket (unlock) and terminate

Add $\langle 1, 3, 6 \rangle$:

- Determine index $\downarrow$
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat
- Bucket locked, so wait
- Equal, so terminate

Reducing Thread Interference (I)

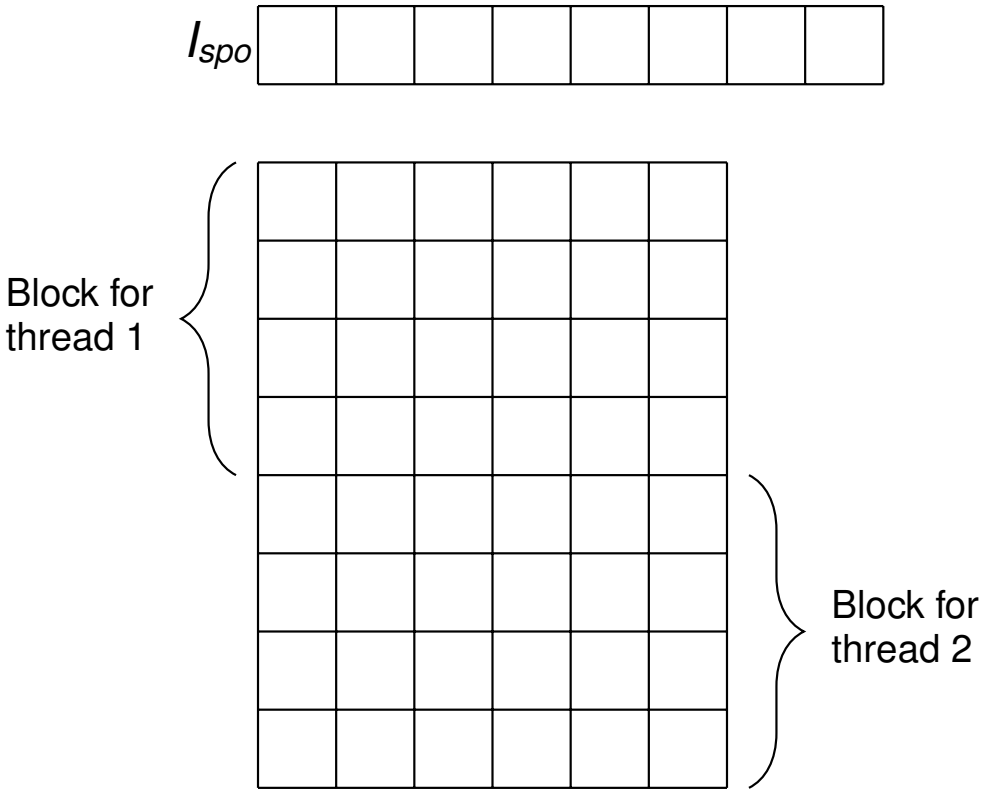
- A write by core A to a location cached by core B invalidates B's cache
- Major parallelisation bottleneck $\Rightarrow$ should avoid writing to shared locations

- Several such bottlenecks:
  1. The pointer to the next triple to process
  2. The pointer to the end of the list of triples
  3. Entries of I_{sp}, I_{op}, and I_p for common $\langle s, p \rangle$, $\langle o, p \rangle$, and p

- To address 1, extract triples from the triple table in blocks

Reducing Thread Interference (II)

- To address 2, reserve space for new triples in blocks

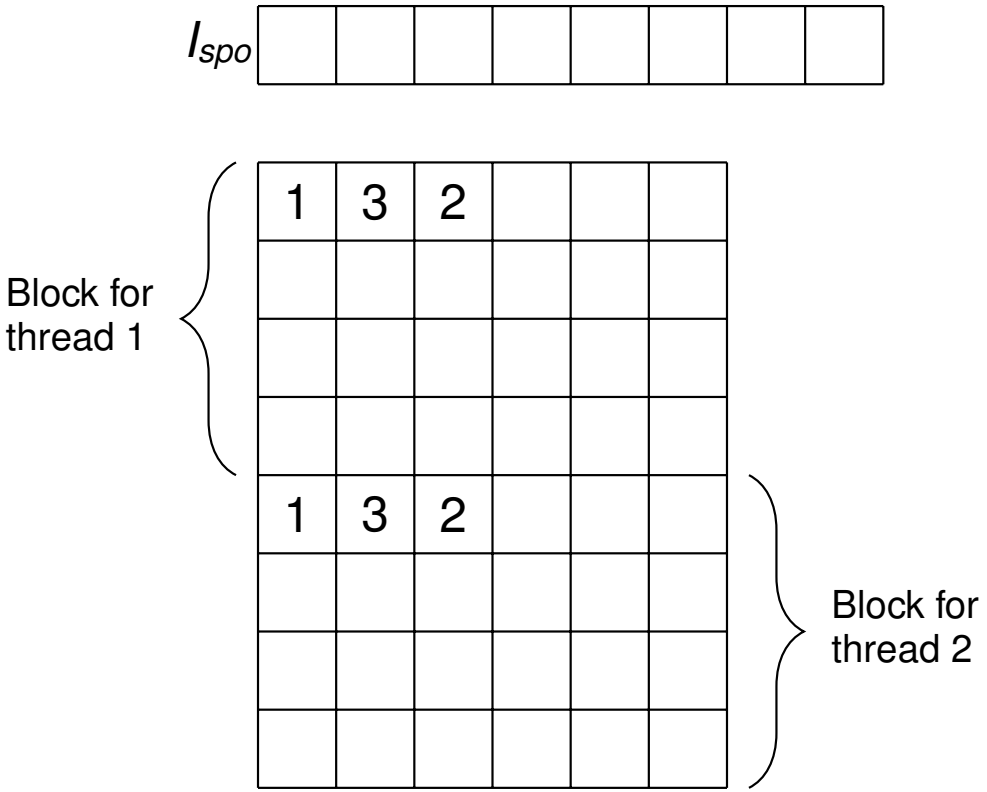


Add $\langle 1, 3, 2 \rangle$ on both threads:

- Triple insertion fully lock free!

Reducing Thread Interference (II)

- To address 2, reserve space for new triples in blocks



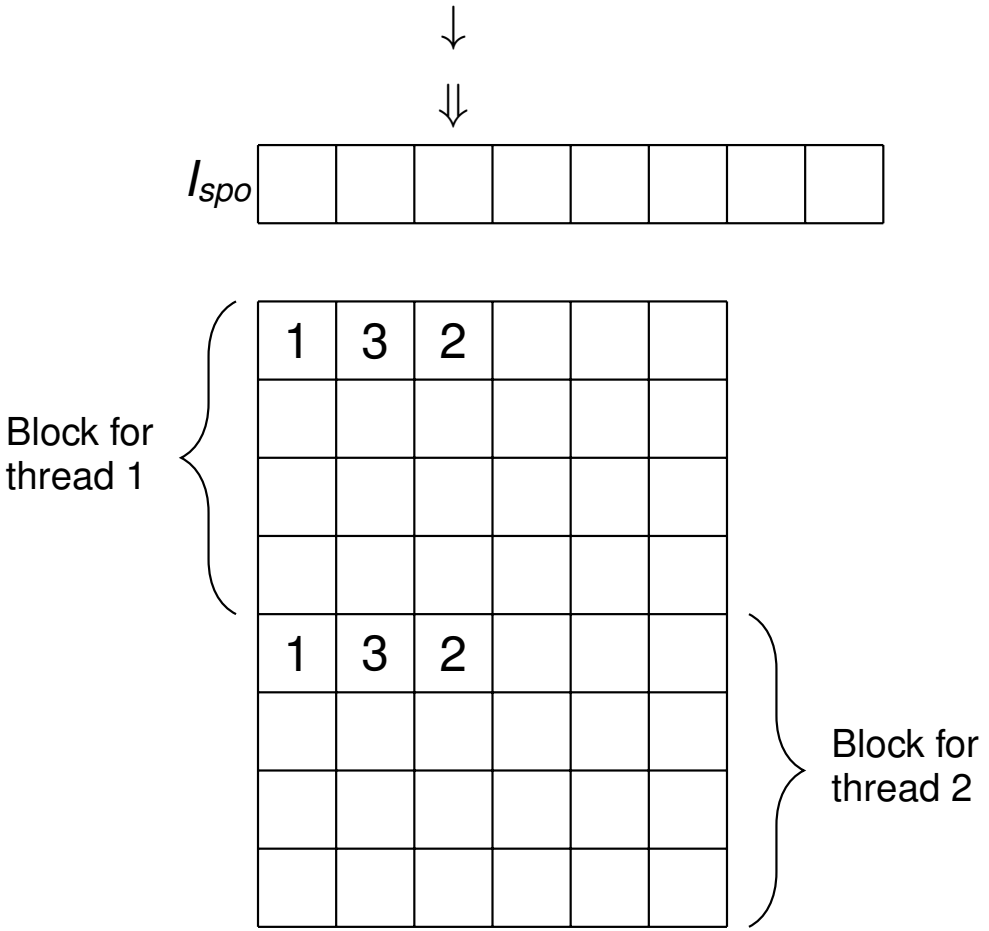
Add $\langle 1, 3, 2 \rangle$ on both threads:

- Add $\langle 1, 3, 2 \rangle$ to local windows

- Triple insertion fully lock free!

Reducing Thread Interference (II)

- To address 2, reserve space for new triples in blocks



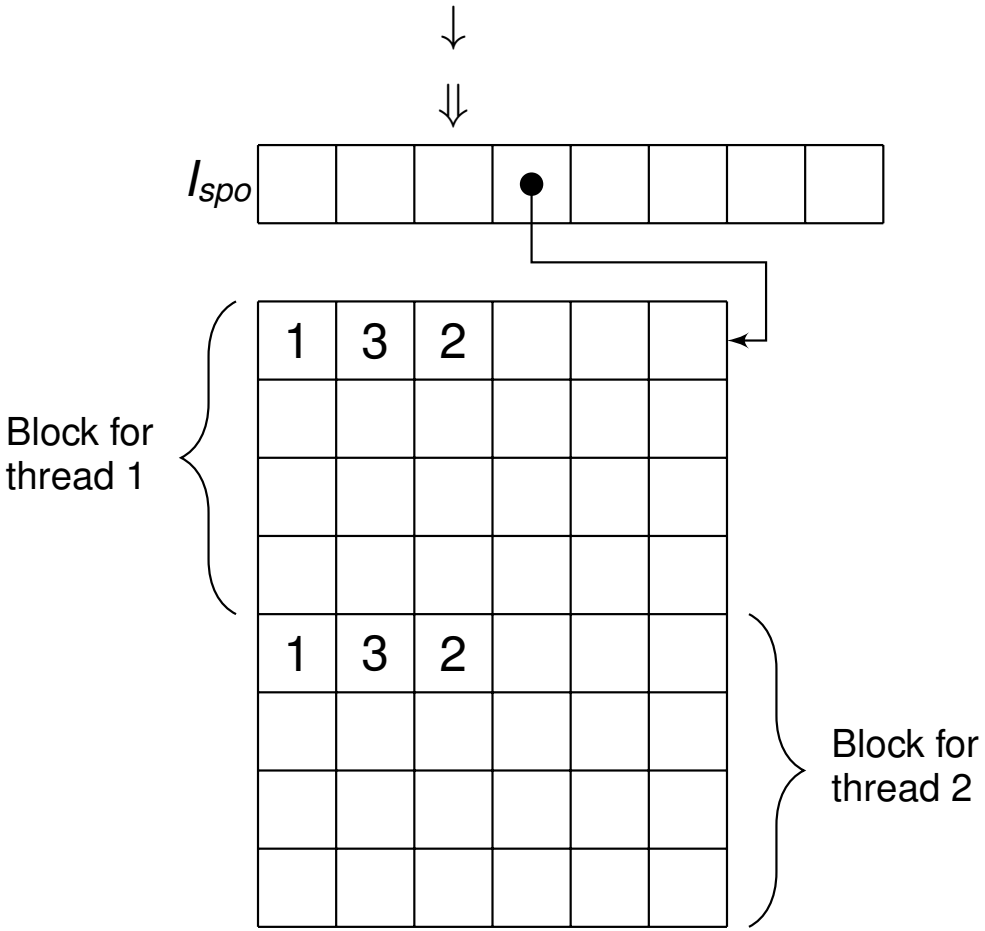
Add $\langle 1, 3, 2 \rangle$ on both threads:

- Add $\langle 1, 3, 2 \rangle$ to local windows
- Compute indexes $\Downarrow$ and $\downarrow$

- Triple insertion fully lock free!

Reducing Thread Interference (II)

- To address 2, reserve space for new triples in blocks



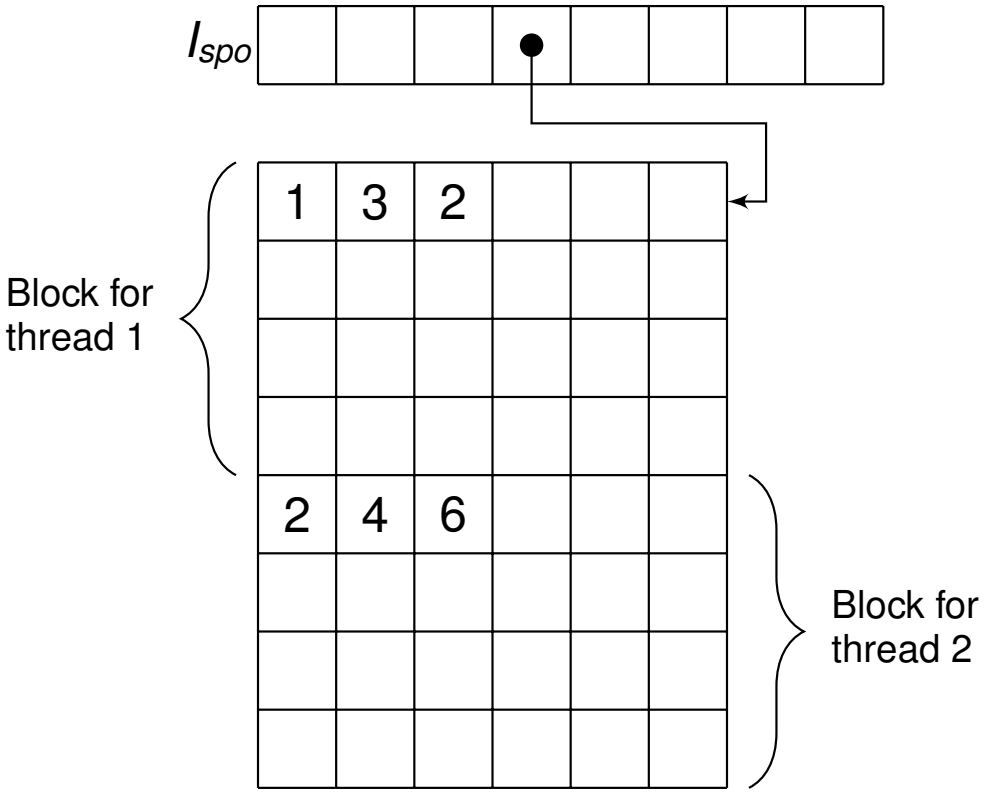
Add $\langle 1, 3, 2 \rangle$ on both threads:

- Add $\langle 1, 3, 2 \rangle$ to local windows
- Compute indexes $\Downarrow$ and $\downarrow$
- CAS into bucket $\Rightarrow$ only one thread succeeds

- Triple insertion fully lock free!

Reducing Thread Interference (II)

- To address 2, reserve space for new triples in blocks



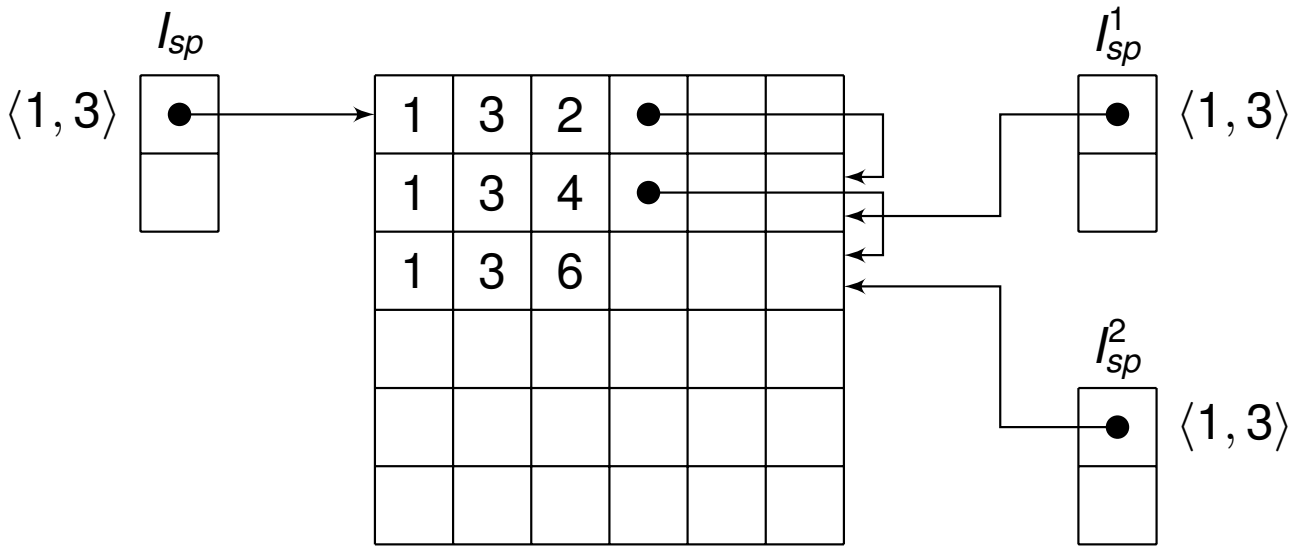
Add $\langle 1, 3, 2 \rangle$ on both threads:

- Add $\langle 1, 3, 2 \rangle$ to local windows
- Compute indexes $\Downarrow$ and $\downarrow$
- CAS into bucket $\Rightarrow$ only one thread succeeds
- Failing thread reuses space for later triples

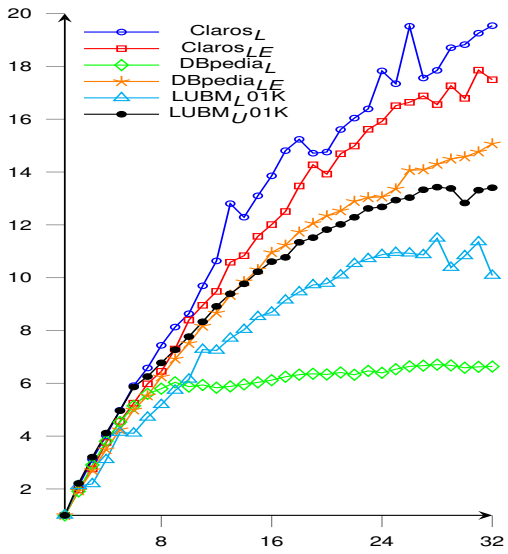
- Triple insertion fully lock free!

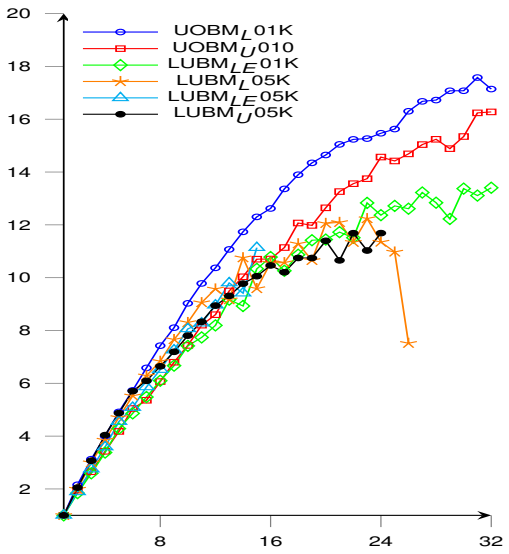
Reducing Thread Interference (III)

- For each thread i, keep local indexes I^i_{sp}, I^i_s, I^i_{op}, I^i_o, I^i_p
- Index entries provide thread i with private insertion points



Evaluation: Parallelisation Overhead and Speedup





- Small concurrency overhead; parallelisation pays off already with two threads
- Speedup of up to 13x with 16 physical cores
- Increases to 19x with 32 virtual cores

Evaluation: Comparison with the State of the Art

	SysX				PG-VP				MDB-VP				PG-TT		MDB-TT		OWLIM-Lite			
	Import		Materialise		Import		Materialise		Import		Materialise		Import		Import		Import		Materialise	
	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t	T	B/t
Claros$_L$	48	89	2062	60	1138	165	25026	97	883	58	Mem		1174	217	896	94	300	54	14293	28
Claros$_{LE}$			4218	44			Mem				Mem								Time	
DBpedia$_L$	274	69	143	67	5844	148	Mem		15499	51	354	49	5968	174	4492	92	1735	171	14855	164
DBpedia$_{LE}$			9538	49			Mem				Mem								Time	
LUBM$_L$01K			71	65			948	127			27	41							316	36
LUBM$_{LE}$ 01K	332	74	765	54	7736	144	15632	112	6136	56	Mem		7947	187	5606	104	2409	40	Time	
LUBM$_U$01K			113	67			Time				138	44							Time	
UOBM$_U$010	5	68	2501	43	116	154	Time		96	50	Mem		120	203	96	85	32	69	11311	24
UOBM$_L$01K	632	64	467	63	13864	119	6708	107	11063	41	358	33	14901	176	10892	101	4778	38	Time	

- DBRDF: an implementation using RDBMS and known techniques
  - PostgreSQL (row store) or MonetDB (column store) as underlying engine
  - vertical partitioning (VP) or triple table (TT) storage model
- OWLIM-Lite: a commercial RDF store by OntoText

Evaluation: Oracle's SPARC T5 (128/1024 cores, 4 TB)

	LUBM-50K		Claros		DBpedia	
Threads	sec	speedup	sec	speedup	sec	speedup
import	6.8k	—	168	—	952	—
1	27.0k	1.0x	10.0k	1.0x	31.2k	1.0x
16	1.7k	15.7x	906.0	11.0x	3.0k	10.4x
32	1.1k	24.0x	583.3	17.1x	1.8k	17.5x
48	920.7	29.3x	450.8	22.2x	2.0k	16.0x
64	721.2	37.4x	374.9	26.7x	1.2k	25.8x
80	523.6	51.5x	384.1	26.0x	1.2k	26.7x
96	442.4	60.9x	364.3	27.4x	825	37.8x
112	400.6	67.3x	331.4	30.2x	1.3k	24.3x
128	387.4	69.6x	225.7	44.3x	697.9	44.7x
256	—	—	226.1	44.2x	684.0	45.7x
384	—	—	189.1	52.9x	546.2	57.2x
512	—	—	153.5	65.1x	431.8	72.3x
640	—	—	140.5	71.2x	393.4	79.4x
768	—	—	130.4	76.7x	366.2	85.3x
896	—	—	127.0	78.8x	364.9	86.6x
1024	—	—	124.9	80.1x	358.8	87.0x
size	B/trp	Triples	B/trp	Triples	B/trp	Triples
aft imp	124.1	6.7G	80.5	18.8M	58.4	112.7M
aft mat	101.0	9.2G	36.9	539.2M	39.0	1.5G
import rate	1.0M		112k		120k	
mat. rate	6.1M		4.2M		4.0M	

Table of Contents

Why Incremental Reasoning?

- Common application scenario: continuous small changes in input data
  - Similar to stream reasoning!
- Materialisation can be expensive ⇒ starting from scratch is unacceptable!
- Incremental maintenance: minimise work needed to update materialisation
- State of the art (from the 90s):
  - the Counting algorithm
    - Basic variant applicable only to nonrecursive programs!
    - Extension to recursive programs rather complex
  - the Delete/Rederive (DRed) algorithm
    - Works for nonrecursive rules too

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

$A(a)$	1
$B(a)$	1

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	1

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	2

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	2
$C_1(a)$	1

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	2
$C_1(a)$	1
...	

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	2
$C_1(a)$	1
...	
$C_n(a)$	1

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation

$A(a)$	1
$B(a)$	1
$C_0(a)$	3
$C_1(a)$	1
...	
$C_n(a)$	1

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0
$B(a)$	1
$C_0(a)$	3
$C_1(a)$	1
...	
$C_n(a)$	1

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
- Delete $A(a)$:
  - Decrease its counter

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0
$B(a)$	1
$C_0(a)$	2
$C_1(a)$	1
...	
$C_n(a)$	1

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
- Delete $A(a)$:
  - Decrease its counter
  - The counter of $A(a)$ reaches zero, so propagate deletion

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0
~~$B(a)$~~	0
$C_0(a)$	2
$C_1(a)$	1
...	
$C_n(a)$	1

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
- Delete $A(a)$:
  - Decrease its counter
  - The counter of $A(a)$ reaches zero, so propagate deletion
- Problem of this variant: delete $B(a)$
  - Decrease its counter

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0
~~$B(a)$~~	0
$C_0(a)$	1
$C_1(a)$	1
...	
$C_n(a)$	1

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
- Delete $A(a)$:
  - Decrease its counter
  - The counter of $A(a)$ reaches zero, so propagate deletion
- Problem of this variant: delete $B(a)$
  - Decrease its counter
  - The counter of $B(a)$ reaches zero, so propagate deletion

Basic Counting (Nonrecursive Variant)

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0
~~$B(a)$~~	0
$C_0(a)$	1
$C_1(a)$	1
...	
$C_n(a)$	1

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
- Delete $A(a)$:
  - Decrease its counter
  - The counter of $A(a)$ reaches zero, so propagate deletion
- Problem of this variant: delete $B(a)$
  - Decrease its counter
  - The counter of $B(a)$ reaches zero, so propagate deletion
  - However, $C_0(a)$ still has a cyclic derivation from $C_n(a)$
  - $\Rightarrow$ The algorithm does not delete $C_0(a), \ldots, C_n(a)$!
  - Reference counting is not a general garbage collection method

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

$A(a)$	1
$B(a)$	1

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

$A(a)$	1	
$B(a)$	1	
$C_0(a)$		1

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

$A(a)$	1	
$B(a)$	1	
$C_0(a)$		2

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

$A(a)$	1		
$B(a)$	1		
$C_0(a)$		2	
$C_1(a)$			1

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

$A(a)$	1		
$B(a)$	1		
$C_0(a)$		2	
$C_1(a)$			1
...			

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

$A(a)$	1			
$B(a)$	1			
$C_0(a)$		2		
$C_1(a)$			1	
...				
$C_n(a)$				1

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

$A(a)$	1				
$B(a)$	1				
$C_0(a)$		2			1
$C_1(a)$			1		
...					
$C_n(a)$				1	

- Associate with each fact an array of counters, one per iteration

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0			
~~$B(a)$~~	0			
$C_0(a)$		2		1
$C_1(a)$			1	
...				
$C_n(a)$				1

- Associate with each fact an array of counters, one per iteration
- Delete $A(a)$ and $B(a)$ by undoing derivations

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0				
~~$B(a)$~~	0				
$C_0(a)$		0			1
$C_1(a)$			1		
...					
$C_n(a)$				1	

- Associate with each fact an array of counters, one per iteration
- Delete $A(a)$ and $B(a)$ by undoing derivations

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0			
~~$B(a)$~~	0			
$C_0(a)$		0		1
$C_1(a)$			0	
...				
$C_n(a)$				1

- Associate with each fact an array of counters, one per iteration
- Delete $A(a)$ and $B(a)$ by undoing derivations

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0			
~~$B(a)$~~	0			
$C_0(a)$		0		1
$C_1(a)$			0	
...				
$C_n(a)$				0

- Associate with each fact an array of counters, one per iteration
- Delete $A(a)$ and $B(a)$ by undoing derivations

Counting and Recursion

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

~~$A(a)$~~	0				
~~$B(a)$~~	0				
$C_0(a)$		0			0
$C_1(a)$			0		
...					
$C_n(a)$				0	

- Associate with each fact an array of counters, one per iteration
- Delete $A(a)$ and $B(a)$ by undoing derivations

The DRed Algorithm at a Glance

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

$A(a)$
$B(a)$

The DRed Algorithm at a Glance

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Materialise initial facts

$A(a)$
$B(a)$
$C_0(a)$
$C_1(a)$
$\ldots$
$C_n(a)$

The DRed Algorithm at a Glance

Example

$C_0(x) \leftarrow A(x)$ $\quad C_0(x) \leftarrow B(x)$ $\quad C_i(x) \leftarrow C_{i-1}(x)$ for $1 \leq i \leq n$ $\quad C_0(x) \leftarrow C_n(x)$

- Materialise initial facts
- Delete $A(a)$ using DRed:

~~$A(a)$~~
$B(a)$
$C_0(a)$
$C_1(a)$
...
$C_n(a)$

The DRed Algorithm at a Glance

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

- Materialise initial facts
- Delete $A(a)$ using DRed:
  1. Delete all facts with a derivation from $A(a)$

$$
\begin{aligned}
C_0(x)^D &\leftarrow A(x)^D \\
C_0(x)^D &\leftarrow B(x)^D \\
C_i(x)^D &\leftarrow C_{i-1}(x)^D \text{ for } 1 \leq i \leq n \\
C_0(x)^D &\leftarrow C_n(x)^D
\end{aligned}
$$

$\cancel{A(a)}$
$B(a)$
$\cancel{C_0(a)}$
$\cancel{C_1(a)}$
$\ldots$
$\cancel{C_n(a)}$

The DRed Algorithm at a Glance

Example

$C_0(x) \leftarrow A(x) \qquad C_0(x) \leftarrow B(x) \qquad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \qquad C_0(x) \leftarrow C_n(x)$

- Materialise initial facts
- Delete $A(a)$ using DRed:
  1. Delete all facts with a derivation from $A(a)$

$$
\begin{aligned}
C_0(x)^D &\leftarrow A(x)^D \\
C_0(x)^D &\leftarrow B(x)^D \\
C_i(x)^D &\leftarrow C_{i-1}(x)^D \text{ for } 1 \leq i \leq n \\
C_0(x)^D &\leftarrow C_n(x)^D
\end{aligned}
$$

  2. Rederive facts that have an alternative derivation

$$
\begin{aligned}
C_0(x) &\leftarrow C_0(x)^D \wedge A(x) \\
C_0(x) &\leftarrow C_0(x)^D \wedge B(x) \\
C_i(x) &\leftarrow C_i(x)^D \wedge C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) &\leftarrow C_0(x)^D \wedge C_n(x)
\end{aligned}
$$

$\cancel{A(a)}$
$B(a)$
$C_0(a)$
$C_1(a)$
$\ldots$
$C_n(a)$

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$A(a)$
$B(a)$
$C_0(a)$
$C_1(a)$
$\ldots$
$C_n(a)$

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$A(a)$
$B(a)$
$C_0(a)$
$C_1(a)$
$\ldots$
$C_n(a)$

- Delete $A(a)$ using B/F:

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$A(a)$
$B(a)$
$C_0(a)$
$C_1(a)$
...
$C_n(a)$

?

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$$\boxed{\begin{array}{l} \cancel{A(a)} \\ B(a) \\ C_0(a) \\ C_1(a) \\ \dots \\ C_n(a) \end{array}} \times$$

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$\cancel{A(a)}$	$\times$
$B(a)$	
$C_0(a)$	?
$C_1(a)$	
$\ldots$	
$C_n(a)$	

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete
  3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

$\cancel{A(a)}$	$\times$
$B(a)$	?
$C_0(a)$	?
$C_1(a)$	
$\ldots$	
$C_n(a)$	

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete
  3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
  4. Apply the rules to $C_0(a)$ 'backwards' $\Rightarrow$ by $C_0(x) \leftarrow B(x)$, we get $B(a)$

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

~~$A(a)$~~	$\times$
$B(a)$	$\checkmark$
$C_0(a)$	?
$C_1(a)$	
$\dots$	
$C_n(a)$	

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete
  3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
  4. Apply the rules to $C_0(a)$ 'backwards' $\Rightarrow$ by $C_0(x) \leftarrow B(x)$, we get $B(a)$
  5. $B(a)$ is explicit so it is derivable

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

~~$A(a)$~~	$\times$
$B(a)$	$\checkmark$
$C_0(a)$	$\checkmark$
$C_1(a)$	
$\dots$	
$C_n(a)$	

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete
  3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
  4. Apply the rules to $C_0(a)$ 'backwards' $\Rightarrow$ by $C_0(x) \leftarrow B(x)$, we get $B(a)$
  5. $B(a)$ is explicit so it is derivable
  6. So $C_0(a)$ is derivable too

Improvement: The Forward/Backward/Forward Algorithm

- In RDF, a fact often has many alternative derivations
- $\Rightarrow$ Many facts get deleted in the first step

- The Backward/Forward (B/F) algorithm: look for alternatives immediately

~~$A(a)$~~	$\times$
$B(a)$	$\checkmark$
$C_0(a)$	$\checkmark$
$C_1(a)$	
$\dots$	
$C_n(a)$	

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No $\Rightarrow$ delete
  3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
  4. Apply the rules to $C_0(a)$ 'backwards' $\Rightarrow$ by $C_0(x) \leftarrow B(x)$, we get $B(a)$
  5. $B(a)$ is explicit so it is derivable
  6. So $C_0(a)$ is derivable too
  7. Stop propagation and terminate

Table of Contents

Research Directions

- Investigate potential for data compression
- Improve join cardinality estimation
- Improve query planning