RDFox — A Modern Materialisation-Based RDF System

Boris Motik

University of Oxford

November 9, 2015
# Table of Contents

1. **Introduction**

2. **Parallel Materialisation in RDFox**

3. **Incremental Materialisation Maintenance**

4. **Conclusion**
# Table of Contents

1. **Introduction**

2. **Parallel Materialisation in RDFox**

3. **Incremental Materialisation Maintenance**

4. **Conclusion**
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

- 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

\[
\langle ?X, \text{rdfs:subClassOf}, ?Z \rangle \leftarrow \langle ?X, \text{rdfs:subClassOf}, ?Y \rangle \land \langle ?Y, \text{rdfs:subClassOf}, ?Z \rangle \\
\langle ?X, \text{rdfs:subPropertyOf}, ?Z \rangle \leftarrow \langle ?X, \text{rdfs:subPropertyOf}, ?Y \rangle \land \langle ?Y, \text{rdfs:subPropertyOf}, ?Z \rangle \\
\langle ?X, \text{rdf:type}, ?Z \rangle \leftarrow \langle ?X, \text{rdf:type}, ?Y \rangle \land \langle ?Y, \text{rdfs:subClassOf}, ?Z \rangle \\
\langle ?X, \text{rdf:type}, ?Z \rangle \leftarrow \langle ?X, ?W, ?Y \rangle \land \langle ?W, \text{rdfs:domain}, ?Z \rangle \\
\langle ?Y, \text{rdf:type}, ?Z \rangle \leftarrow \langle ?X, ?W, ?Y \rangle \land \langle ?W, \text{rdfs:range}, ?Z \rangle
\]
Goals

- Develop techniques for materialisation of datalog programs on RDF data
Introduction

**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 ⇒ natural to parallelise
    - Mid-range laptops have 4 cores, servers with 16 cores are routine

Goals of the RDFox system

- Implemented in the RDFox system
  - http://www.cs.ox.ac.uk/isg/tools/RDFox/
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 ⇒ natural to parallelise
    - Mid-range laptops have 4 cores, servers with 16 cores are routine
# Table of Contents

1. **Introduction**

2. **Parallel Materialisation in RDFox**

3. **Incremental Materialisation Maintenance**

4. **Conclusion**
**Existing Approaches to Parallel Materialisation**

- **Interquery parallelism**: run independent rules in parallel
  - Degree of parallelism limited by the number of independent rules
  - ⇒ 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 $(x \mod N = i)$
    - ⇒ Static partitioning may not distribute workload well due to data skew
    - ⇒ 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
    - ⇒ Hash tables keep being constantly recomputed
    - Sort-merge join requires constant data reordering

- **Goal**: distribute workload to threads evenly and with minimum overhead
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**

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:

\[ A(x) \land R(x, y) \rightarrow A(y) \]
**Solution Part I: Algorithm**

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

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**

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**

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: $A(b)$
Parallel Materialisation in RDFox

Solution Part I: Algorithm

\[ \begin{align*}
R(a,b) \\
R(a,c) \\
R(b,d) \\
R(b,e) \\
A(a) \\
R(c,f) \\
R(c,g) \\
A(b) \\
A(c)
\end{align*} \]

\[ \Rightarrow \]

\[ A(x) \land R(x, y) \rightarrow A(y) \]

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

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

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

Current subquery:  \( A(c) \)

\[ A(x) \land R(x, y) \rightarrow A(y) \]
Parallel Materialisation in RDFox

**Solution Part I: Algorithm**

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>R(a,c)</th>
<th>R(b,d)</th>
<th>R(b,e)</th>
<th>A(a)</th>
<th>R(c,f)</th>
<th>R(c,g)</th>
<th>A(b)</th>
<th>A(c)</th>
</tr>
</thead>
</table>

⇒

\[
A(x) \land R(x, y) \rightarrow A(y)
\]

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

\[ \Rightarrow A(x) \land R(x, y) \rightarrow A(y) \]

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: \( \text{R(b,y)} \)
**Solution Part I: Algorithm**

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>R(a,c)</th>
<th>R(b,d)</th>
<th>R(b,e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(a)</td>
<td>R(c,f)</td>
<td>R(c,g)</td>
<td></td>
</tr>
<tr>
<td>A(b)</td>
<td></td>
<td></td>
<td>A(e)</td>
</tr>
<tr>
<td>A(c)</td>
<td></td>
<td></td>
<td>A(f)</td>
</tr>
<tr>
<td>A(d)</td>
<td></td>
<td></td>
<td>A(g)</td>
</tr>
<tr>
<td>A(x) ∧ R(x, y) → A(y)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[
\begin{array}{|c|}
\hline
R(a,b) \\
R(a,c) \\
R(b,d) \\
R(b,e) \\
A(a) \\
R(c,f) \\
R(c,g) \\
A(b) \\
A(c) \\
A(d) \\
A(e) \\
A(f) \\
A(g) \\
\hline
\end{array}
\]

\[
A(x) \land R(x, y) \rightarrow A(y)
\]

- 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**

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: $R(e, y)$

\[
A(x) \land R(x, y) \rightarrow A(y)
\]
**Solution Part I: Algorithm**

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>R(a,c)</th>
<th>R(b,d)</th>
<th>R(b,e)</th>
<th>A(a)</th>
<th>R(c,f)</th>
<th>R(c,g)</th>
<th>A(b)</th>
<th>A(c)</th>
<th>A(d)</th>
<th>A(e)</th>
<th>A(f)</th>
<th>A(g)</th>
</tr>
</thead>
</table>

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**

<table>
<thead>
<tr>
<th>R(a,b)</th>
<th>R(a,c)</th>
<th>R(b,d)</th>
<th>R(b,e)</th>
<th>A(a)</th>
<th>R(c,f)</th>
<th>R(c,g)</th>
<th>A(b)</th>
<th>A(c)</th>
<th>A(d)</th>
<th>A(e)</th>
<th>A(f)</th>
<th>A(g)</th>
</tr>
</thead>
</table>

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)

\[ A(x) \land R(x, y) \rightarrow A(y) \]
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

\[
A(x) \land R(x, y) \rightarrow A(y)
\]

\[
R(a, b), \ R(b, c), \ R(c, d), \ R(d, e), \ A(a)
\]
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
- ⇒ We partition rule instances dynamically and with little overhead

WHEN PARALLELISATION FAILS

\[ A(x) \land R(x, y) \rightarrow A(y) \]
\[ R(a, b), \ R(b, c), \ R(c, d), \ R(d, e), \ A(a), \ A(b) \]
Parallel Materialisation in RDFox

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 \textit{dynamically} and with \textit{little overhead}

<table>
<thead>
<tr>
<th>WHEN PARALLELISATION FAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A(x) \land R(x, y) \rightarrow A(y) )</td>
</tr>
<tr>
<td>( R(a, b), R(b, c), R(c, d), R(d, e), A(a), A(b), A(c) )</td>
</tr>
</tbody>
</table>
Parallelising Computation

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

When Parallelisation Fails

\[
A(x) \land R(x, y) \rightarrow A(y)
\]

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

\[
A(x) \land R(x, y) \rightarrow A(y)
\]

$R(a, b), \ R(b, c), \ R(c, d), \ R(d, e), \ A(a), \ A(b), \ A(c), \ A(d), \ A(e)$
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)
Parallel Materialisation in RDFox

DATA STRUCTURE

<table>
<thead>
<tr>
<th>I_s</th>
<th>R_s</th>
<th>R_p</th>
<th>R_o</th>
<th>N_sp</th>
<th>N_p</th>
<th>N_op</th>
<th>I_spo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I_sp</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨1,3⟩</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨2,1⟩</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>⟨1,1⟩</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
‘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: $\text{CAS}(loc, exp, new)$
    - 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., $\text{CAS}$ 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

\[ T \begin{array}{c} 1 \\ 3 \\ 2 \end{array} \]

\[ T_{\text{new}}^1 \begin{array}{c} 1 \\ 3 \\ 6 \end{array} \]

\[ 1 \begin{array}{c} 3 \\ 4 \end{array} \]

\[ T_{\text{new}}^2 \begin{array}{c} 1 \\ 3 \\ 8 \end{array} \]
Lock-Free Insertion into Lists

- $T_{\text{next}} = T \cdot N_{sp}$
- $T_{\text{new}} \cdot N_{sp} = T_{\text{next}}$
- $T_{\text{next}} = T \cdot N_{sp}$
- $T_{\text{new}} \cdot N_{sp} = T_{\text{next}}$
**Lock-Free Insertion into Lists**

- $T_{\text{next}} = T \cdot N_{\text{sp}}$
- $T_{\text{new}} \cdot N_{\text{sp}} = T_{\text{next}}$
- $\text{CAS}(T \cdot N_{\text{sp}}, T_{\text{next}}, T_{\text{new}})$
- Succeeds!

- $T_{\text{next}} = T \cdot N_{\text{sp}}$
- $T_{\text{new}} \cdot N_{\text{sp}} = T_{\text{next}}$
- $\text{CAS}(T \cdot N_{\text{sp}}, T_{\text{next}}, T_{\text{new}})$
- Fails so we repeat the process
Lock-Free Insertion into Lists

\[ T_{\text{new}}^{1} \rightarrow 1 \ 3 \ 6 \rightarrow T \quad 1 \ 3 \ 2 \rightarrow T_{\text{new}}^{2} \]

- \( T_{\text{next}}^{1} = T.N_{sp} \)
- \( T_{\text{new}}^{1} \cdot N_{sp} = T_{\text{next}}^{1} \)
- \( \text{CAS}(T.N_{sp}, T_{\text{next}}^{1}, T_{\text{new}}^{1}) \)
- Succeeds!

- \( T_{\text{next}}^{2} = T.N_{sp} \)
- \( T_{\text{new}}^{2} \cdot N_{sp} = T_{\text{next}}^{2} \)
- \( \text{CAS}(T.N_{sp}, T_{\text{next}}^{2}, T_{\text{new}}^{2}) \)
- Fails so we repeat the process
- \( T_{\text{next}}^{2} = T.N_{sp} \)
- \( T_{\text{new}}^{2} \cdot N_{sp} = T_{\text{next}}^{2} \)
**Lock-Free Insertion into Lists**

- \( T_{\text{next}} = T.N_{sp} \)
- \( T_{\text{new}}.N_{sp} = T_{\text{next}} \)
- \( \text{CAS}(T.N_{sp}, T_{\text{next}}, T_{\text{new}}) \)
- Succeeds!

\[
\begin{array}{c}
T \quad 1 \quad 3 \quad 2 \\
T_{\text{new}} \quad 1 \quad 3 \quad 6 \\
T_{\text{new}} \quad 1 \quad 3 \quad 4 \\
\end{array}
\]

- \( T_{\text{next}} = T.N_{sp} \)
- \( T_{\text{new}}.N_{sp} = T_{\text{next}} \)
- \( \text{CAS}(T.N_{sp}, T_{\text{next}}, T_{\text{new}}) \)
- Fails so we repeat the process

\[
\begin{array}{c}
T \quad 1 \quad 3 \quad 8 \\
T_{\text{new}} \quad 1 \quad 3 \quad 4 \\
\end{array}
\]

- \( T_{\text{next}} = T.N_{sp} \)
- \( T_{\text{new}}.N_{sp} = T_{\text{next}} \)
- \( \text{CAS}(T.N_{sp}, T_{\text{next}}, T_{\text{new}}) \)
- Succeeds!
‘Mostly’ Lock-Free Insertion into $I_{spo}$

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

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

Not equal, so advance

Empty, so lock bucket using CAS

Failure, so repeat

Bucket locked, so wait

Equal, so terminate
‘Mostly’ Lock-Free Insertion into $I_{spo}$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$

Add $\langle 1, 3, 6 \rangle$:
- Determine index $\downarrow$
‘Mostly’ Lock-Free Insertion into $l_{spo}$

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 $l_{spo}$

Add $\langle 1, 3, 6 \rangle$:
- Determine index ↓
- Not equal, so advance
- Empty, so lock bucket using CAS

Add $\langle 1, 3, 6 \rangle$:
- Determine index ↓
- 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 ↓
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!

Add $\langle 1, 3, 6 \rangle$:
- Determine index ↓
- 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 ↓
- Not equal, so advance
- Empty, so lock bucket using CAS
- Success!

Add \( \langle 1, 3, 6 \rangle \):
- Determine index ↓
- Not equal, so advance
- Empty, so lock bucket using CAS
- Failure, so repeat
- Bucket locked, so wait
‘Mostly’ Lock-Free Insertion into $l_{spo}$

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
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 $l_{sp}$, $l_{op}$, and $l_{p}$ for common $\langle s, p \rangle$, $\langle o, p \rangle$, and $p$

To address 1, extract triples from the triple table in blocks
To address 2, reserve space for new triples in blocks.

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

- Block for thread 1
- Block for thread 2

Triple insertion fully lock free!
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!
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!
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!
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!
For each thread $i$, keep local indexes $I_{sp}^i$, $I_s^i$, $I_{op}^i$, $I_o^i$, $I_p^i$

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

<table>
<thead>
<tr>
<th></th>
<th>Seq. imp.</th>
<th>Par. imp.</th>
<th>Memory</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>B/t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Claros_L</td>
<td>48</td>
<td>89</td>
<td>1138</td>
<td>25026</td>
</tr>
<tr>
<td>Claros_LE</td>
<td>2062</td>
<td>60</td>
<td>4218</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>1138</td>
<td>165</td>
<td>25026</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>883</td>
<td>58</td>
<td>1174</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>896</td>
<td>94</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14293</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBpedia_L</td>
<td>274</td>
<td>69</td>
<td>5844</td>
<td>15499</td>
</tr>
<tr>
<td>DBpedia_LE</td>
<td>143</td>
<td>67</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>15499</td>
<td>51</td>
<td>354</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>5968</td>
<td>174</td>
<td>4492</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>1735</td>
<td>171</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14855</td>
<td>164</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUBM_U 01K</td>
<td>332</td>
<td>74</td>
<td>7736</td>
<td>6136</td>
</tr>
<tr>
<td>UOBM_L 01K</td>
<td>71</td>
<td>65</td>
<td>948</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>41</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td></td>
<td>7947</td>
<td>187</td>
<td>5606</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>2409</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>316</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>11311</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10892</td>
<td>101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4778</td>
<td>38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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)

<table>
<thead>
<tr>
<th>Threads</th>
<th>LUBM-50K</th>
<th>Claros</th>
<th>DBpedia</th>
</tr>
</thead>
<tbody>
<tr>
<td>import</td>
<td>sec</td>
<td>speedup</td>
<td>sec</td>
</tr>
<tr>
<td>1</td>
<td>6.8k</td>
<td>—</td>
<td>168</td>
</tr>
<tr>
<td>16</td>
<td>27.0k</td>
<td>1.0x</td>
<td>10.0k</td>
</tr>
<tr>
<td>32</td>
<td>1.7k</td>
<td>15.7x</td>
<td>906.0</td>
</tr>
<tr>
<td>48</td>
<td>1.1k</td>
<td>24.0x</td>
<td>583.3</td>
</tr>
<tr>
<td>64</td>
<td>920.7</td>
<td>29.3x</td>
<td>450.8</td>
</tr>
<tr>
<td>80</td>
<td>721.2</td>
<td>37.4x</td>
<td>374.9</td>
</tr>
<tr>
<td>96</td>
<td>523.6</td>
<td>51.5x</td>
<td>384.1</td>
</tr>
<tr>
<td>112</td>
<td>442.4</td>
<td>60.9x</td>
<td>364.3</td>
</tr>
<tr>
<td>128</td>
<td>400.6</td>
<td>67.3x</td>
<td>331.4</td>
</tr>
<tr>
<td>256</td>
<td>387.4</td>
<td>69.6x</td>
<td>225.7</td>
</tr>
<tr>
<td>384</td>
<td>—</td>
<td>—</td>
<td>226.1</td>
</tr>
<tr>
<td>512</td>
<td>—</td>
<td>—</td>
<td>189.1</td>
</tr>
<tr>
<td>640</td>
<td>—</td>
<td>—</td>
<td>153.5</td>
</tr>
<tr>
<td>768</td>
<td>—</td>
<td>—</td>
<td>140.5</td>
</tr>
<tr>
<td>896</td>
<td>—</td>
<td>—</td>
<td>130.4</td>
</tr>
<tr>
<td>1024</td>
<td>—</td>
<td>—</td>
<td>127.0</td>
</tr>
<tr>
<td>size</td>
<td>B/trp</td>
<td>Triples</td>
<td>B/trp</td>
</tr>
<tr>
<td>aft imp</td>
<td>124.1</td>
<td>6.7G</td>
<td>80.5</td>
</tr>
<tr>
<td>aft mat</td>
<td>101.0</td>
<td>9.2G</td>
<td>36.9</td>
</tr>
<tr>
<td>import rate</td>
<td>1.0M</td>
<td>112k</td>
<td>120k</td>
</tr>
<tr>
<td>mat. rate</td>
<td>6.1M</td>
<td>4.2M</td>
<td>4.0M</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction

2. Parallel Materialisation in RDFox

3. Incremental Materialisation Maintenance

4. Conclusion
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**

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

| \(A(a)\) | 1 |
| \(B(a)\) | 1 |
**Basic Counting (Nonrecursive Variant)**

### Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A(a))</td>
<td>1</td>
</tr>
<tr>
<td>(B(a))</td>
<td>1</td>
</tr>
<tr>
<td>(C_0(a))</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[ C_0(x) \leftarrow A(x) \quad C_0(x) \leftarrow B(x) \quad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \quad C_0(x) \leftarrow C_n(x) \]
Basic Counting (Nonrecursive Variant)

Example

\[
\begin{align*}
C_0(x) &\leftarrow A(x) \\
C_0(x) &\leftarrow B(x) \\
C_i(x) &\leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) &\leftarrow C_n(x)
\end{align*}
\]

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A(a)</td>
<td>1</td>
</tr>
<tr>
<td>B(a)</td>
<td>1</td>
</tr>
<tr>
<td>C_0(a)</td>
<td>2</td>
</tr>
</tbody>
</table>
**Basic Counting (Nonrecursive Variant)**

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

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

<table>
<thead>
<tr>
<th>A(a)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(a)</td>
<td>1</td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td>2</td>
</tr>
<tr>
<td>$C_1(a)$</td>
<td>1</td>
</tr>
</tbody>
</table>
**Basic Counting (Nonrecursive Variant)**

**Example**

<table>
<thead>
<tr>
<th>A(a)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(a)</td>
<td>1</td>
</tr>
<tr>
<td>C_0(a)</td>
<td>2</td>
</tr>
<tr>
<td>C_1(a)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Associate with each fact a counter initialised to zero
- Increment the counter after each derivation
**Basic Counting (Nonrecursive Variant)**

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0(x) \leftarrow A(x)$</td>
</tr>
</tbody>
</table>

- 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**

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A(a)</td>
<td>1</td>
</tr>
<tr>
<td>B(a)</td>
<td>1</td>
</tr>
<tr>
<td>C_0(a)</td>
<td>3</td>
</tr>
<tr>
<td>C_1(a)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>C_n(a)</td>
<td>1</td>
</tr>
</tbody>
</table>
**Basic Counting (Nonrecursive Variant)**

**Example**

\[
\begin{align*}
C_0(x) &\leftarrow A(x) \\
C_0(x) &\leftarrow B(x) \\
C_i(x) &\leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) &\leftarrow C_n(x)
\end{align*}
\]

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

| \(A(a)\) | 0 |
| \(B(a)\) | 1 |
| \(C_0(a)\) | 3 |
| \(C_1(a)\) | 1 |
| ... |  |
| \(C_n(a)\) | 1 |
Basic Counting (Nonrecursive Variant)

Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0(x)$ ← $A(x)$</td>
<td>$C_0(x)$ ← $B(x)$</td>
<td>$C_i(x)$ ← $C_{i-1}(x)$ for $1 \leq i \leq n$</td>
<td>$C_0(x)$ ← $C_n(x)$</td>
</tr>
</tbody>
</table>

- 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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(a)$</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B(a)$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_1(a)$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_n(a)$</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Counting (Nonrecursive Variant)

Example

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

- 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

| A(a) | 0 |
| B(a) | 0 |
| C₀(a) | 1 |
| C₁(a) | 1 |
| ... |
| Cₙ(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**

\[
\begin{array}{llll}
C_0(x) & \leftarrow & A(x) & \\
C_0(x) & \leftarrow & B(x) & \\
C_i(x) & \leftarrow & C_{i-1}(x) & \text{for } 1 \leq i \leq n \\
C_0(x) & \leftarrow & C_n(x) & \\
\end{array}
\]

- 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
Incremental Materialisation Maintenance

COUNTING AND RECURSION

**Example**

| \(A(a)\) | 1 |
| \(B(a)\) | 1 |

\[
\begin{align*}
\text{Example} & : \\
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]
COUNTING AND RECURSION

**Example**

<table>
<thead>
<tr>
<th></th>
<th>$C_0(x)$ ← $A(x)$</th>
<th>$C_0(x)$ ← $B(x)$</th>
<th>$C_i(x)$ ← $C_{i-1}(x)$ for $1 \leq i \leq n$</th>
<th>$C_0(x)$ ← $C_n(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(a)$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B(a)$</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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

**Example**

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C_0(a)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Example**

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

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

- Associate with each fact an array of counters, one per iteration
**COUNTING AND RECURSION**

**EXAMPLE**

<table>
<thead>
<tr>
<th>$A(a)$</th>
<th>$B(a)$</th>
<th>$C_0(a)$</th>
<th>$C_1(a)$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th></th>
<th>$A(a)$</th>
<th>$B(a)$</th>
<th>$C_0(a)$</th>
<th>$C_1(a)$</th>
<th>$\ldots$</th>
<th>$C_n(a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Associate with each fact an array of counters, one per iteration.
**Counting and Recursion**

**Example**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(a)$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$B(a)$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$C_1(a)$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_n(a)$</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

- Associate with each fact an array of counters, one per iteration.
**Counting and Recursion**

**Example**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A(a))</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(B(a))</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(C_0(a))</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(C_1(a))</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C_n(a))</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

Example

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

\[
\begin{array}{c|cc}
A(a) & 0 & 0 \\
B(a) & 0 & 1 \\
C_0(a) & 0 & 1 \\
C_1(a) & 1 & . . . \\
C_n(a) & 1 & \end{array}
\]

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

<table>
<thead>
<tr>
<th>$A(a)$</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B(a)$</td>
<td>0</td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td>0</td>
</tr>
<tr>
<td>$C_1(a)$</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$C_n(a)$</td>
<td>1</td>
</tr>
</tbody>
</table>

1. Associate with each fact an array of counters, one per iteration
2. Delete $A(a)$ and $B(a)$ by undoing derivations
### Example

<table>
<thead>
<tr>
<th></th>
<th>$A(a)$</th>
<th>$B(a)$</th>
<th>$C_0(a)$</th>
<th>$C_1(a)$</th>
<th>$\ldots$</th>
<th>$C_n(a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(a)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B(a)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_0(a)$</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_1(a)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\ldots$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_n(a)$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[ C_0(x) \leftarrow A(x) \quad C_0(x) \leftarrow B(x) \quad C_i(x) \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \quad C_0(x) \leftarrow C_n(x) \]
COUNTING AND RECURSION

**Example**

\[
\begin{align*}
C_0(x) &\leftarrow A(x) & C_0(x) &\leftarrow B(x) & C_i(x) &\leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n & C_0(x) &\leftarrow C_n(x) \\
0 & & 0 & & 0 & & 0 & & 0 & & 0 \\
A(a) & & 0 & & 0 & & 0 & & 0 & & 0 \\
B(a) & & 0 & & 0 & & 0 & & 0 & & 0 \\
C_0(a) & & 0 & & 0 & & 0 & & 0 & & 0 \\
C_1(a) & & 0 & & 0 & & 0 & & 0 & & 0 \\
\ldots & & \ldots & & \ldots & & \ldots & & \ldots & & \ldots \\
C_n(a) & & 0 & & 0 & & 0 & & 0 & & 0 \\
\end{align*}
\]

- 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**

\[
\begin{align*}
C_0(x) &\leftarrow A(x) \\
C_0(x) &\leftarrow B(x) \\
C_i(x) &\leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) &\leftarrow C_n(x)
\end{align*}
\]

\[
\begin{align*}
A(a) \\
B(a)
\end{align*}
\]
**THE DRed Algorithm at a Glance**

**Example**

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

- Materialise initial facts

| $A(a)$ | $B(a)$ | $C_0(a)$ | $C_1(a)$ | $\ldots$ | $C_n(a)$ |
THE DRed Algorithm at a Glance

**Example**

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

<table>
<thead>
<tr>
<th>$A(a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B(a)$</td>
</tr>
<tr>
<td>$C_0(a)$</td>
</tr>
<tr>
<td>$C_1(a)$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$C_n(a)$</td>
</tr>
</tbody>
</table>
THE DRed ALGORITHM AT A GLANCE

Example

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

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

\[
\begin{align*}
A(a) \\
B(a) \\
C_0(a) \\
C_1(a) \\
\cdots \\
C_n(a)
\end{align*}
\]

\[
\begin{align*}
C_0(x)^D & \leftarrow A(x)^D \\
C_0(x)^D & \leftarrow B(x)^D \\
C_i(x)^D & \leftarrow C_{i-1}(x)^D \text{ for } 1 \leq i \leq n \\
C_0(x)^D & \leftarrow C_n(x)^D
\end{align*}
\]
THE DRed ALGORITHM AT A GLANCE

**Example**

\[
\begin{align*}
C_0(x) & \leftarrow A(x) \\
C_0(x) & \leftarrow B(x) \\
C_i(x) & \leftarrow C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
C_0(x) & \leftarrow C_n(x)
\end{align*}
\]

- Materialise initial facts
- Delete \( A(a) \) using DRed:
  1. Delete all facts with a derivation from \( A(a) \)
     \[
     \begin{align*}
     C_0(x)^D & \leftarrow A(x)^D \\
     C_0(x)^D & \leftarrow B(x)^D \\
     C_i(x)^D & \leftarrow C_{i-1}(x)^D \text{ for } 1 \leq i \leq n \\
     C_0(x)^D & \leftarrow C_n(x)^D
     \end{align*}
     \]
  2. Rederive facts that have an alternative derivation
     \[
     \begin{align*}
     C_0(x) & \leftarrow C_0(x)^D \land A(x) \\
     C_0(x) & \leftarrow C_0(x)^D \land B(x) \\
     C_i(x) & \leftarrow C_i(x)^D \land C_{i-1}(x) \text{ for } 1 \leq i \leq n \\
     C_0(x) & \leftarrow C_0(x)^D \land C_n(x)
     \end{align*}
     \]
In RDF, a fact often has many alternative derivations
⇒ Many facts get deleted in the first step

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

<table>
<thead>
<tr>
<th>$A(a)$</th>
<th>$B(a)$</th>
<th>$C_0(a)$</th>
<th>$C_1(a)$</th>
<th>...</th>
<th>$C_n(a)$</th>
</tr>
</thead>
</table>

1. Is $A(a)$ derivable in any other way?
2. No ⇒ delete
3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
4. Apply the rules to $C_0(a)$ 'backwards' ⇒ 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
In RDF, a fact often has many alternative derivations
⇒ Many facts get deleted in the first step

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

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

- $A(a)$
- $B(a)$
- $C_0(a)$
- $C_1(a)$
- ...$C_n(a)$
Incremental Materialisation Maintenance

**IMPROVEMENT: THE FORWARD/BACKWARD/FORWARD ALGORITHM**

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

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

<table>
<thead>
<tr>
<th>A(a)</th>
<th>B(a)</th>
<th>C₀(a)</th>
<th>C₁(a)</th>
<th>...</th>
<th>Cₙ(a)</th>
</tr>
</thead>
</table>

#### Delete A(a) using B/F:

1. Is A(a) derivable in any other way?
In RDF, a fact often has many alternative derivations
⇒ Many facts get deleted in the first step

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

Delete \(A(a)\) using B/F:

1. Is \(A(a)\) derivable in any other way?
2. No \(\Rightarrow\) delete
In RDF, a fact often has many alternative derivations

⇒ Many facts get deleted in the first step

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

- Delete $A(a)$ using B/F:
  1. Is $A(a)$ derivable in any other way?
  2. No ⇒ 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
- ⇒ Many facts get deleted in the first step

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

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

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

1. Is $A(a)$ derivable in any other way?
2. No ⇒ delete
3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
4. Apply the rules to $C_0(a)$ ‘backwards’ ⇒ by $C_0(x) \leftarrow B(x)$, we get $B(a)$
In RDF, a fact often has many alternative derivations
⇒ Many facts get deleted in the first step

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

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

1. Is $A(a)$ derivable in any other way?
2. No ⇒ delete
3. As in DRed, identify $C_0(a)$ as derivable from $A(a)$
4. Apply the rules to $C_0(a)$ ‘backwards’ ⇒ by $C_0(x) \leftarrow B(x)$, we get $B(a)$
5. $B(a)$ is explicit so it is derivable
In RDF, a fact often has many alternative derivations

⇒ Many facts get deleted in the first step

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

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
In RDF, a fact often has many alternative derivations

⇒ Many facts get deleted in the first step

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

Delete \( A(a) \) using B/F:

1. Is \( A(a) \) derivable in any other way?
2. No ⇒ delete
3. As in DRed, identify \( C_0(a) \) as derivable from \( A(a) \)
4. Apply the rules to \( C_0(a) \) ‘backwards’ ⇒ 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

1. Introduction

2. Parallel Materialisation in RDFox

3. Incremental Materialisation Maintenance

4. Conclusion
Research Directions

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