Old Techniques for New Join Algorithms

Chris Aberger, Susan Tu,
Kunle Olukotun, and Christopher Ré
Stanford University

EMPTYHEADED

Love at First Query: Joins & RDF

Scalable Join Processing on Very Large RDF Graphs

Thomas Neumann
Max-Planck Institute for Informatics
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

Gerhard Weikum
Max-Planck Institute for Informatics
Saarbrücken, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

With the proliferation of the RDF data format, engines for RDF query processing are faced with very large graphs that contain hundreds of millions of RDF triples. This paper addresses the resulting scalability problems. Recent prior work along these lines has focused on indexing and other physical-design issues. The current paper focuses on join processing, as the fine-grained and schema-relaxed use of RDF often entails star- and chain-shaped join queries with many input streams from index scans.

We present two contributions for scalable join processing. First, we develop very light-weight methods for sideways information passing between separate joins at query run-time, to provide highly effective filters on the input streams of joins. Second, we improve previously proposed algorithms for join-order optimization by more accurate selectivity estimations for very large RDF graphs. Experimental studies with several RDF datasets, including the UniProt collection, demonstrate the performance gains of our approach, outperforming the previously fastest systems by more than an order of magnitude.

Categories and Subject Descriptors

H.2 [**Systems**]: Query processing

General Terms

Algorithms, Performance

1. INTRODUCTION

1.1 Background

The RDF data format, originally proposed for the Semantic Web, has gained popularity for building large-scale data collections. In particular, biologists and other life scientists like RDF because of its ease of use and flexibility [5, 37]. They can easily collect data without a schema-first database design phase – a paradigm known as "pay-as-you-go" dataspaces [16]. Moreover, it is easy to add metadata, annotations, and lineage information, and represent all of this uniformly together with the primary data. Querying provides certain degrees of schema agnosticity that relational systems do not offer: property names, the RDF counterpart of attribute names, can be left unspecified in queries. A second application area where RDF develops major momentum is knowledge-management communities such as DBpedia [9] or freebase [13] that aim to build large collections of facts about entities and their relations with RDF-based representations. Information extraction from Wikipedia and other Web sources (e.g., [2, 35, 40]) and social-tagging communities (e.g., [6, 12]) have a similar flavor.

RDF data collections can be seen as entity-relationship graphs with edges corresponding to *(subject, property, object) (SPO) triples* (with property often also referred to as predicate). For example, the fact that the French novelist (and Nobel prize winner) Jean-Marie Le Clezio has written the book "Desert" would be represented by the following four triples:

```
(Id1, hasName, Jean-Marie Le Clezio),
(Id1, hasNationality, French),
(Id1, hasWritten, Id2), (Id2, hasTitle Desert).
```

Querying such RDF data amounts to evaluating graph patterns, expressed in the SPARQL query language. For example, the query with the following three *triple patterns*

```
?x hasName ?y .  ?x hasNationality French .
?x hasProfession novelist
```

finds the names of all French novelists (where the dot denotes a conjunction). This example shows that the fine-grained modeling by triples and the pattern-oriented querying often entails manyway joins, at least conceptually. The example can be seen as a star (self-) join over a universal triples table. Queries that connect different entities via relations (e.g., authors and their books and the characters in these books) involve potentially long chain joins. Although it is conceivable to cluster RDF triples and map clusters onto relational-style tables with multiple properties (e.g., all persons together with their names, nationalities, professions), such approaches face major complexity regarding an appropriate physical design. Recent work [1, 25, 31, 38] has shown that column-store representations or using a single triples table for storage drastically simplify the physical-design problem and often provide superior performance.

1.2 Problem

Large join queries are an inherent characteristic of searching RDF data, posing a performance challenge already for

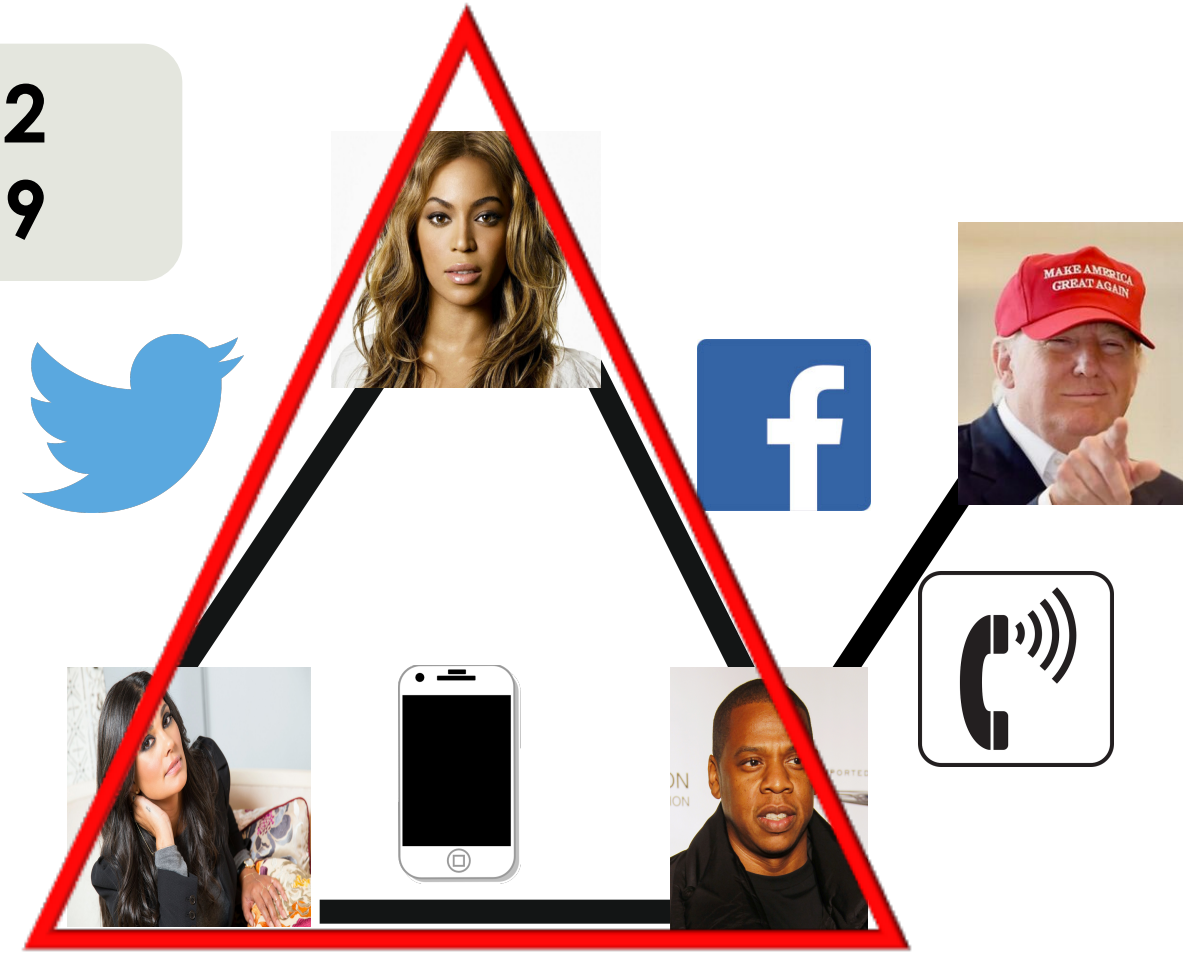
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

"Large join queries are an inherent characteristic of searching RDF data..."

The Cool Clique

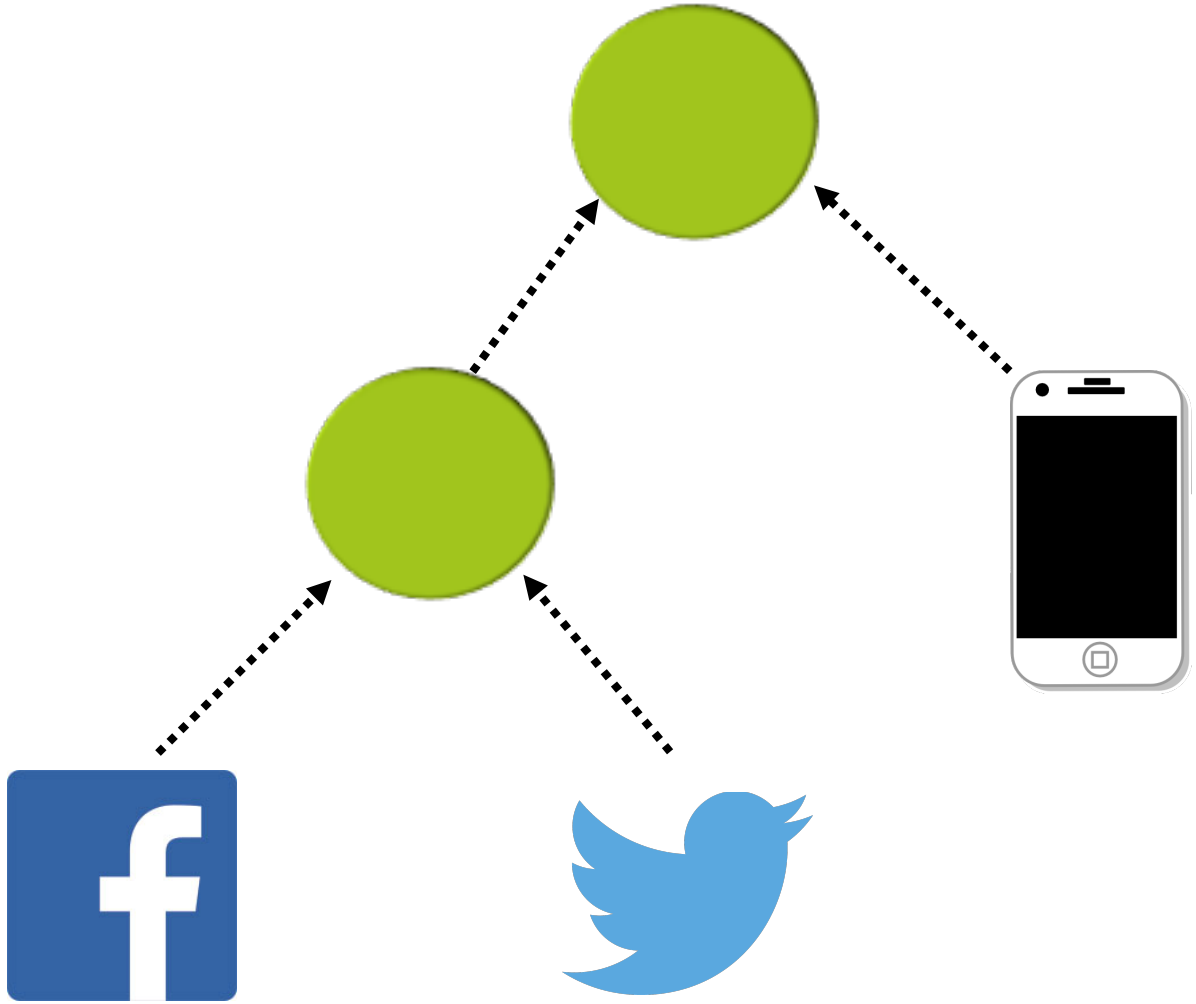
- Consider a social network ontology.

LUBM Query 2
LUBM Query 9



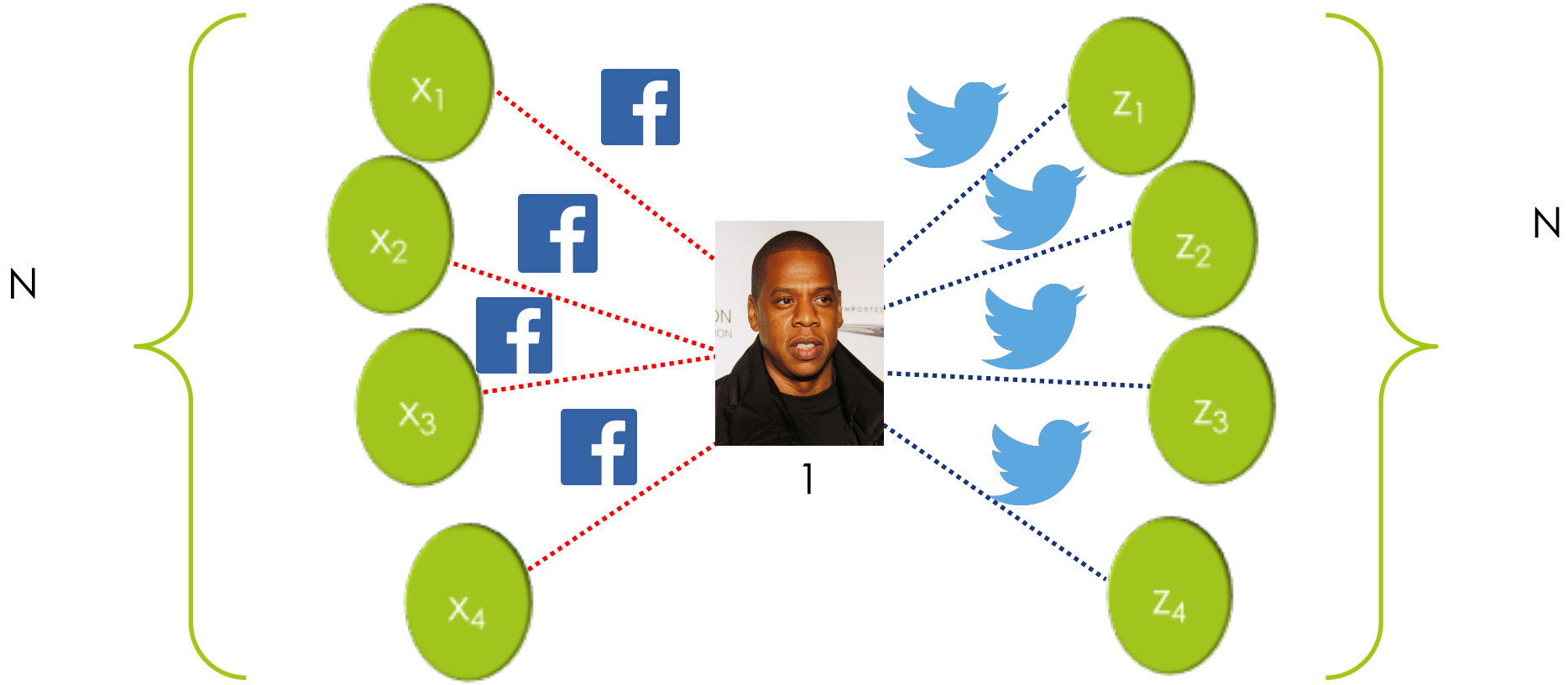
Joins Since System R: Pairwise

Facebook(x,y), Twitter(y,z), Text(x,z)



Pairwise Joins are Suboptimal

Facebook(x,y) , Twitter(y,z) , Text(x,z)



Panic: Best known bound is $O(N^{3/2})$ and any pairwise join plan takes $\Omega(N^2)$.

DBs have been asymptotically suboptimal for the last 4 decades…

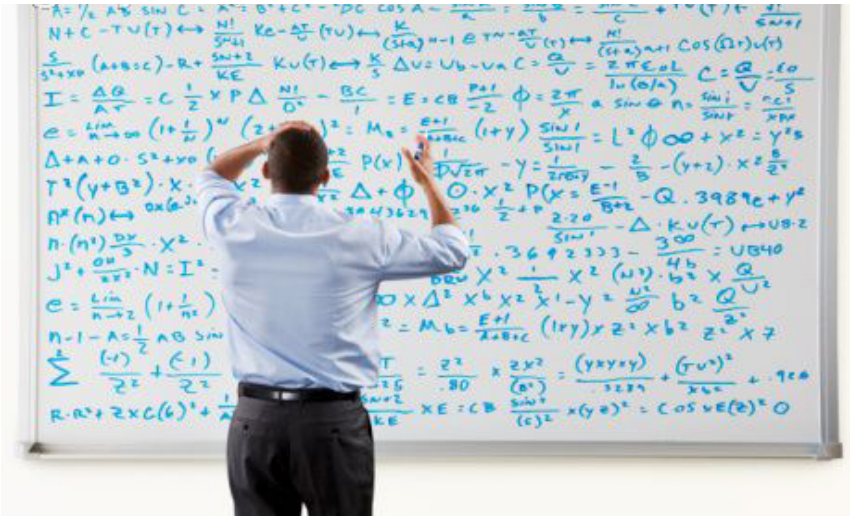
Ngo, Porat, Ré, and Rudra *(PODS 2012)*

1st algorithm for joins with optimal worst-case runtime

Instead of computing joins over **relations** in a **pairwise** manner, compute them over **attributes** in a **multiway** fashion.

Algorithm: Only Map and Set Intersection.

In theory, theory and practice are the same. Lessons from the past.



Coppersmith–Winograd algorithm for matrix multiplication $O(N^{2.375477})$

Hardware don't care: Naive $O(N^3)$ still works best in practice!

EMPTYHEADED

We built an engine.

Problem Statement and Objective

Problem:

(1) Join techniques are old and suboptimal.
(2) RDF queries are usually complex joins.

Goals:

(1) Can WC-Optimal joins **outperform** specialized RDF engines *when they have a theoretical advantage*?
(2) Can a WC-Optimal design **compete with** specialized RDF engines *when there is no theoretical advantage*?

Rethinking Join Processing

Part 1. EmptyHeaded's Architecture
- RDF Data in EmptyHeaded
- Demystifying WC-Optimal joins
- EmptyHeaded's Query Compiler

Part 2. Optimizations for RDF
- Pushing Down Selections
- Pipelining
- Index Layout

EMPTYHEADED

EMPTYHEADED

The Engine: Core Concepts.

Vertical Partitioning: RDF Data in a DB

Subject	Property	Object
Christopher Ré	teaches	Database
Christopher Ré	type	Assistant Prof.
Kunle Olukotun	type	Full Prof.
Chris Aberger	type	Graduate Student

teaches(s,o)

Subject	Object
Christopher Ré	Database
Kunle Olukotun	Architecture

type(s,o)

Subject	Object
Christopher Ré	Assistant Prof.
Kunle Olukotun	Full Prof.
Chris Aberger	Graduate Student

Key Idea: RDF data now 2-attribute relations!

Rethinking Join Processing

Part 1. EmptyHeaded's Architecture
- RDF Data in EmptyHeaded
- **Demystifying WC-Optimal joins**
- EmptyHeaded's Query Compiler

Part 2. Optimizations for RDF
- Pushing Down Selections
- Pipelining
- Index Layout

EMPTYHEADED

Demystifying the WC-Optimal Algo.

Ex: R(x,y),S(y,z),T(x,z).

```
for x in R[] ∩ T[]
  for y in R[x] ∩ S[]
    for z in S[y] ∩ T[x]
      out ← out ∪ (x,y,z)
```

Rethinking Join Processing

Part 1. EmptyHeaded's Architecture
- RDF Data in EmptyHeaded
- Demystifying WC-Optimal joins
- **EmptyHeaded's Query Compiler**

Part 2. Optimizations for RDF
- Pushing Down Selections
- Pipelining
- Index Layout

EMPTYHEADED

Query Plans for WC-Optimal Joins

Generalized hypertree decompositions (GHDs) yield even better runtimes.

- A generalization of tree decompositions.
- See Puttagunta et al. [PODS '16]

Key Idea: This is our analog of relational algebra to represent logical query plans.

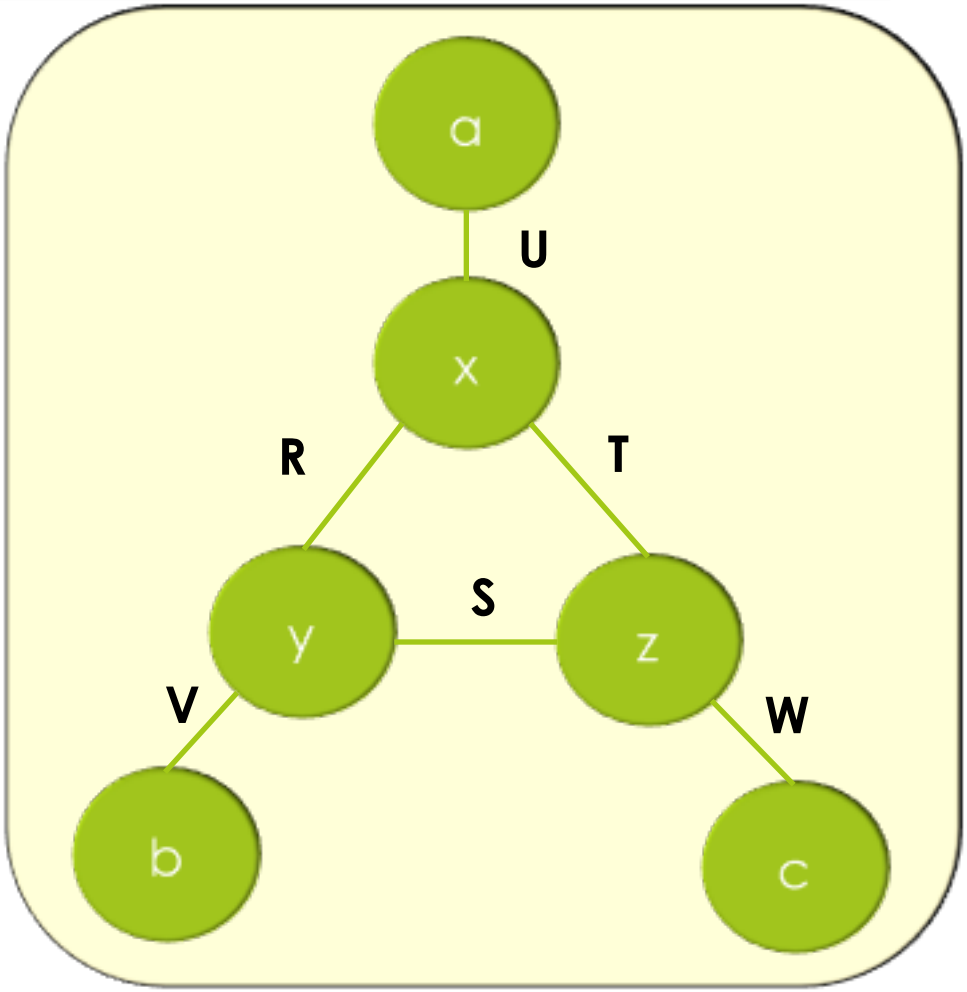
Enables: Classic query optimizations like early aggregation and pushing down selections

Key Insight: Creates an execution DAG.

GHDs by Example

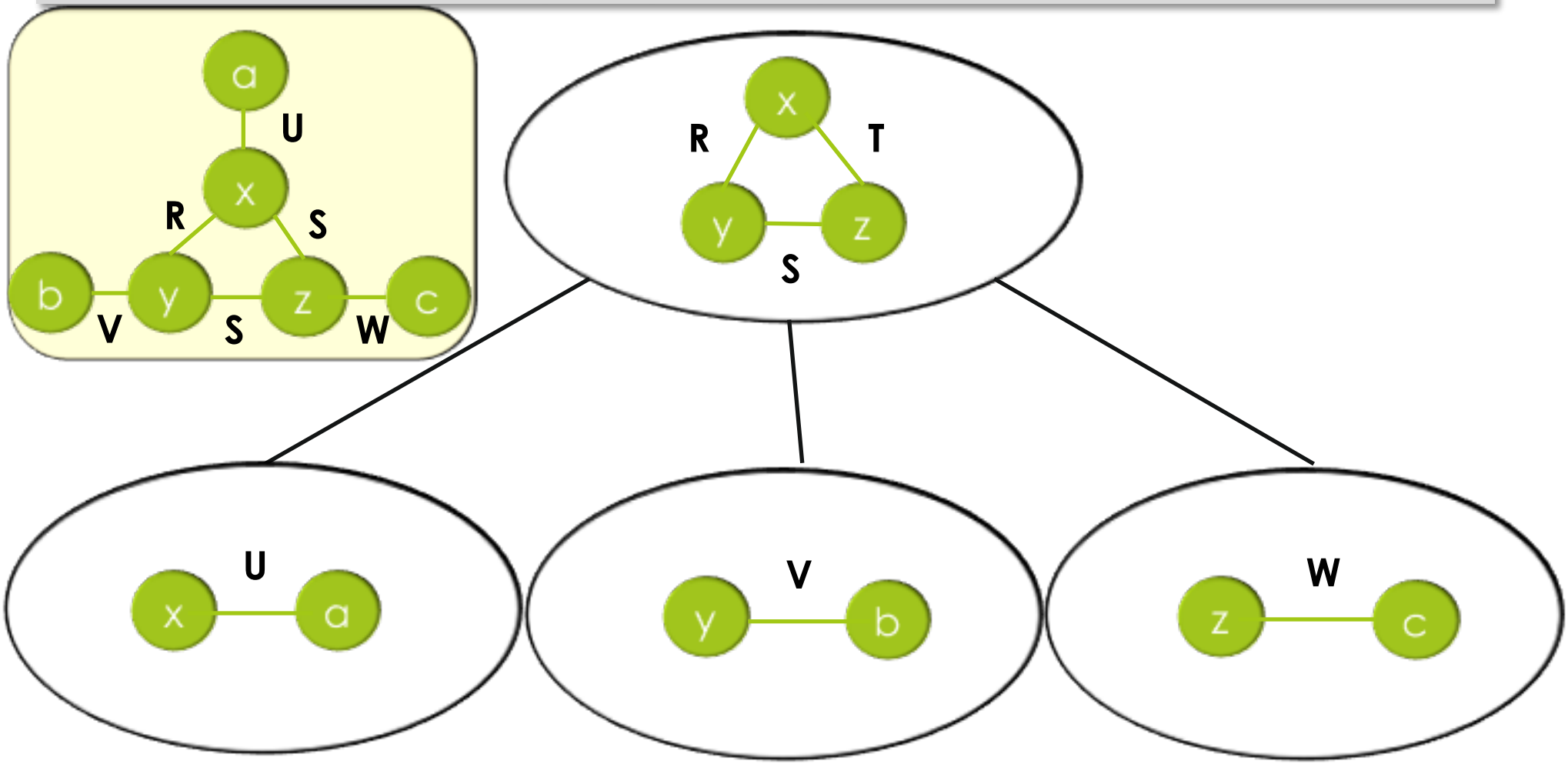
LUBM 2,9: *R(x,y),S(y,z),T(x,z),*
U(x,a=1),V(y,b=2),W(z,c=3)

Intuition:
(1) Add nodes for pieces of graph to process together.
(2) Add edges for data dependencies.
The resulting execution tree is a GHD.



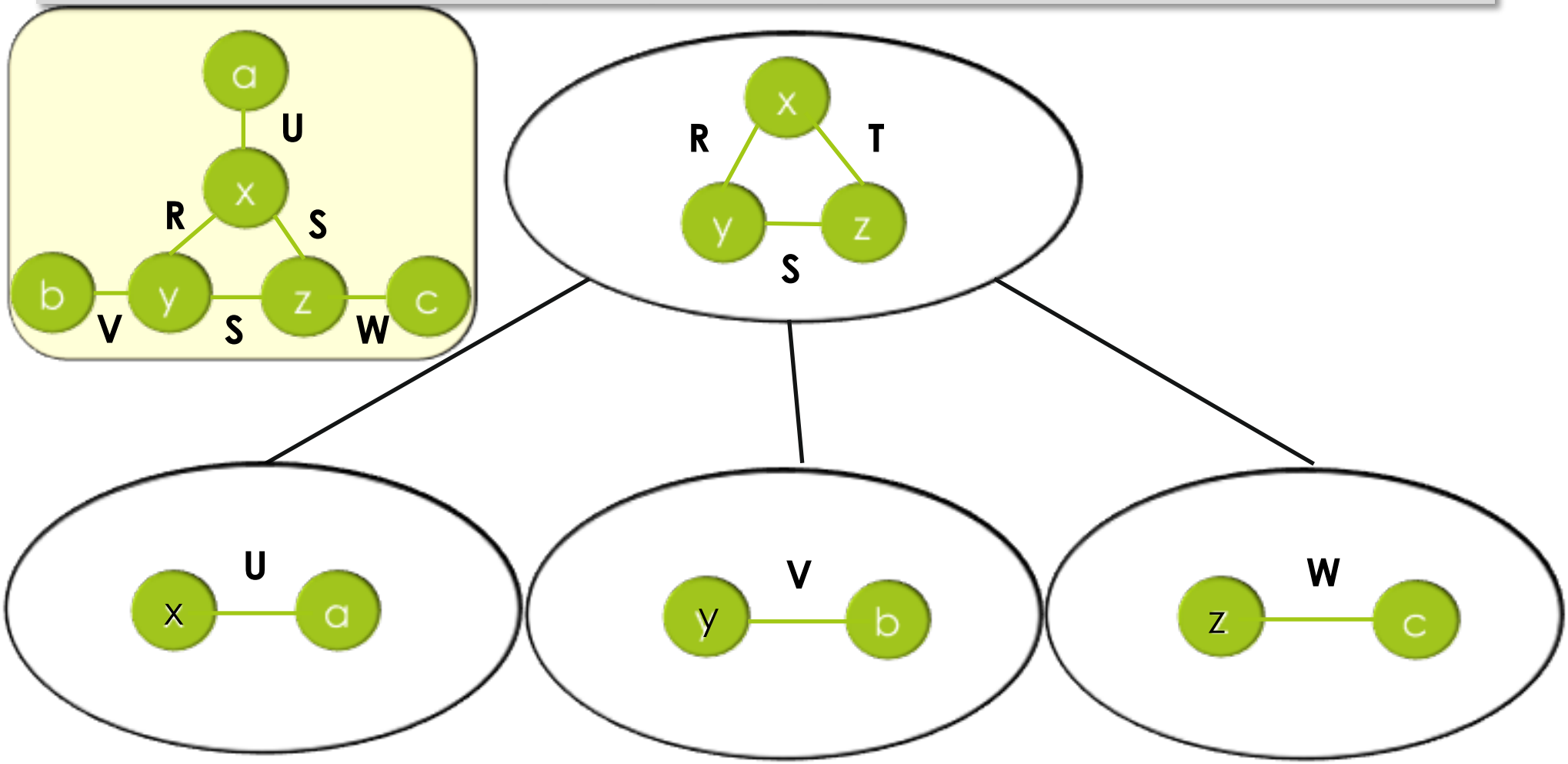
GHDs by Example

LUBM 2,9: *R(x,y),S(y,z),T(x,z),*
U(x,a=0),V(y,b=1),W(z,c=2)



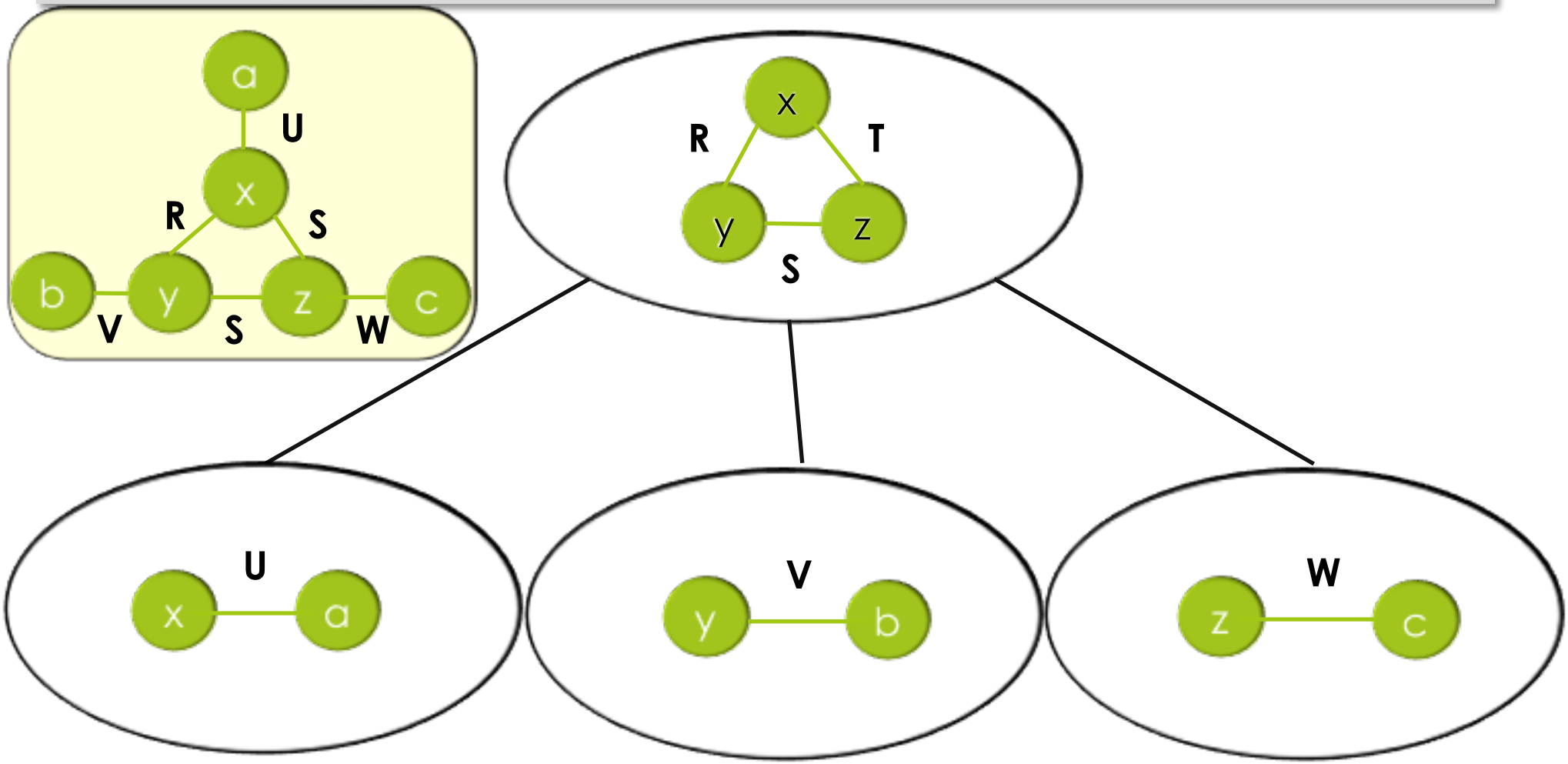
GHDs by Example

LUBM 2,9: *R(x,y),S(y,z),T(x,z),*
U(x,a=0),V(y,b=1),W(z,c=2)



GHDs by Example

LUBM 2,9: *R(x,y),S(y,z),T(x,z),*
U(x,a=0),V(y,b=1),W(z,c=2)



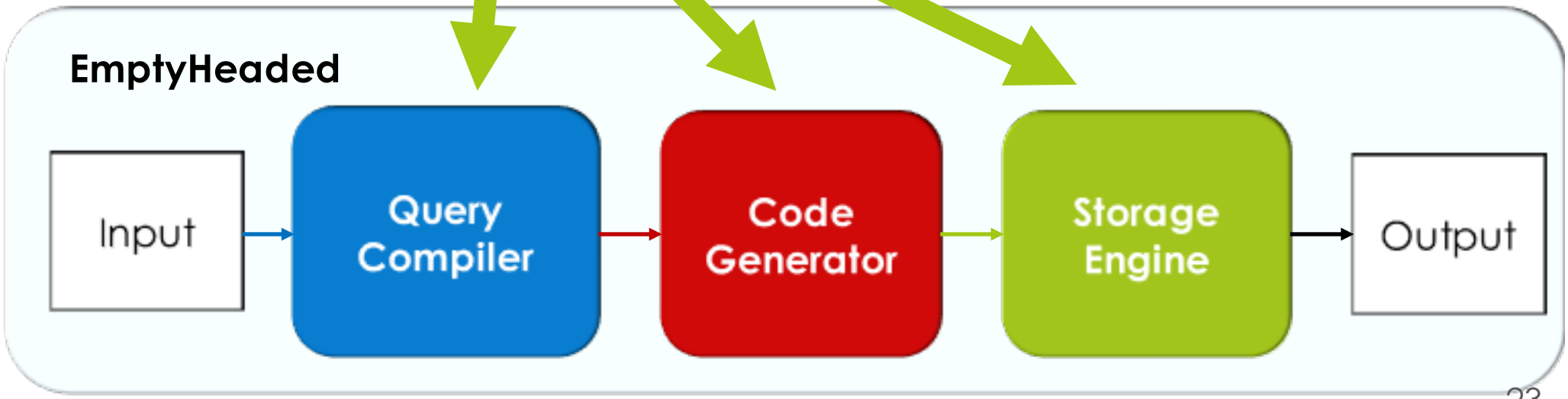
EMPTYHEADED

The Engine + RDF Optimizations.

RDF Processing in the EmptyHeaded Architecture

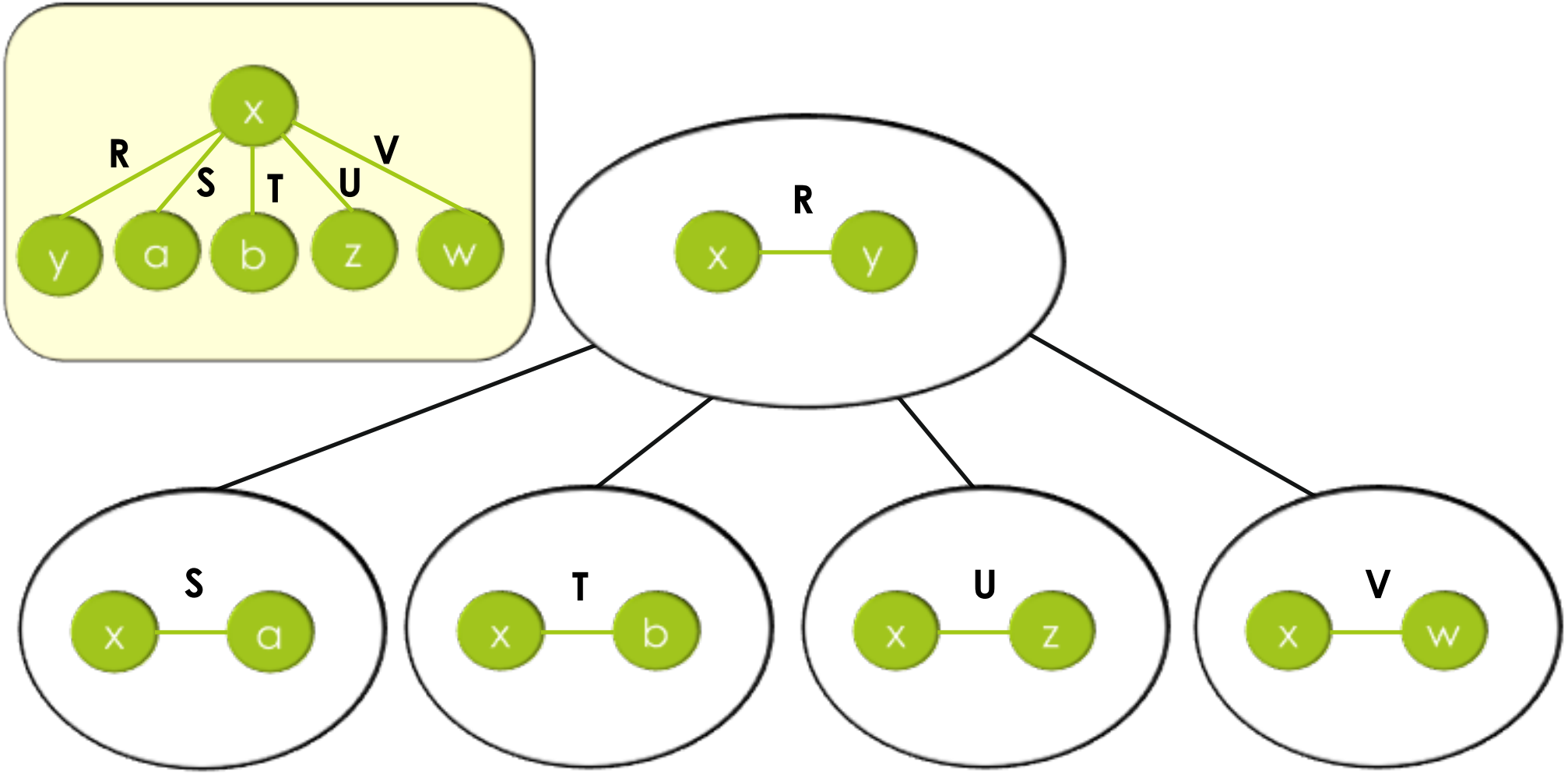
3 Classic Optimizations

- Pushing Down Selections: Q4
- Pipelining: Q8
- Index Layout: Q14



Pushing Down Selections

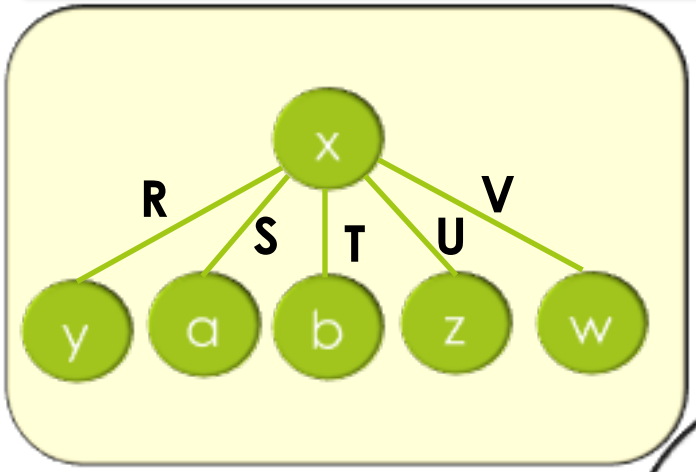
LUBM 4: *R(x,y),S(x,a=0),T(x,b=1),U(x,z),V(x,w)*



High Selectivity!

Pushing Down Selections

LUBM 4: *R(x,y1),S(x,a=0),T(x,b=1),U(x,z),V(x,w)*



Speedup: 69.94x

Rethinking Join Processing

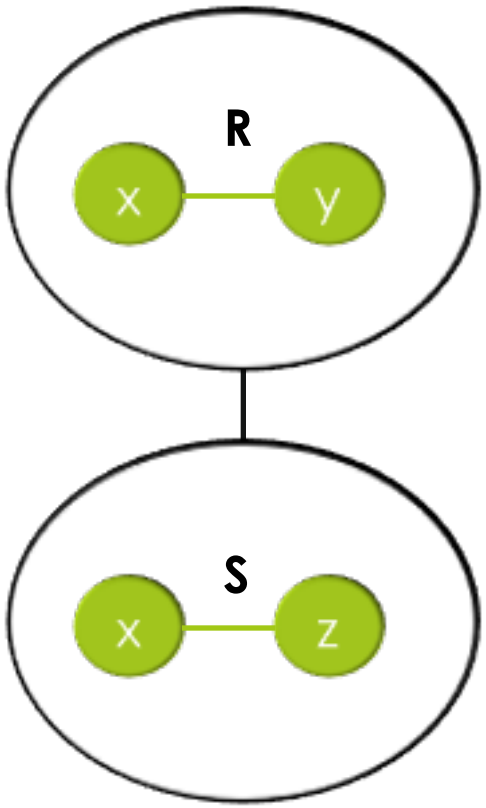
Part 1. EmptyHeaded's Architecture
- RDF Data in EmptyHeaded
- Demystifying WC-Optimal joins
- EmptyHeaded's Query Compiler

Part 2. Optimizations for RDF
- Pushing Down Selections
- **Pipelining**
- Index Layout

EMPTYHEADED

Pipelining

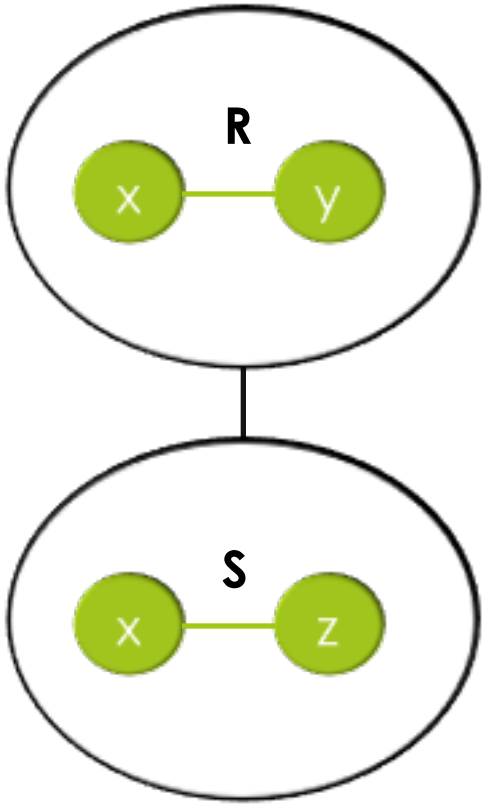
LUBM 8: Given relations R(x,y),S(x,z)
select x,y,z **from** R,S **where** R.x = S.x




```
for x in S[]
    for z in S[x]
        out1 ← out1 ∪ (x,x)
for x in out1[] ∩ R[]
    for y in R[]
        out2 ← out2 ∪ (x,y)
for x in out2[]
    for y in out2[x]
        for z in out1[x]
            out ← out ∪ (x,y,z)
```

Pipelining

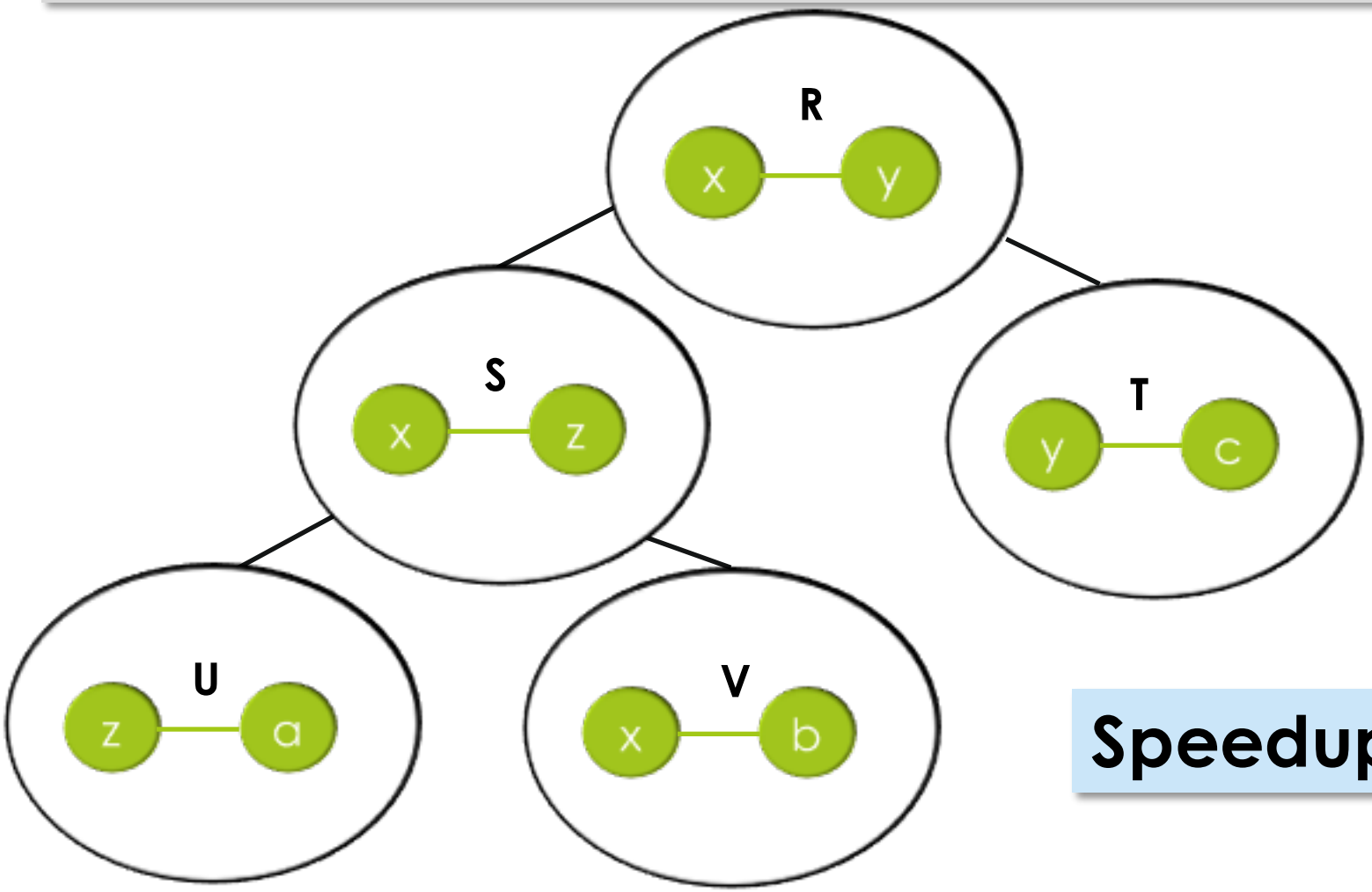
LUBM 8: Given relations R(x,y),S(x,z)
select x,y,z **from** R,S **where** R.x = S.x




```
for x in S[]
    for z in S[x]
        out1 ← out1 ∪ (x,x)
for x in out1[] ∩ R[]
    for y in S[]
        for z in out1[x]
            out ← out ∪ (x,y)
```

Pipelining

LUBM 8: Given relations R(x,y),S(x,z)
select x,y,z **from** R,S **where** R.x = S.x



Speedup: 4.67x

Rethinking Join Processing

Part 1. EmptyHeaded's Architecture
- RDF Data in EmptyHeaded
- Demystifying WC-Optimal joins
- EmptyHeaded's Query Compiler

Part 2. Optimizations for RDF
- Pushing Down Selections
- Pipelining
- **Index Layout**

EMPTYHEADED

Data Layout: Trie Representation

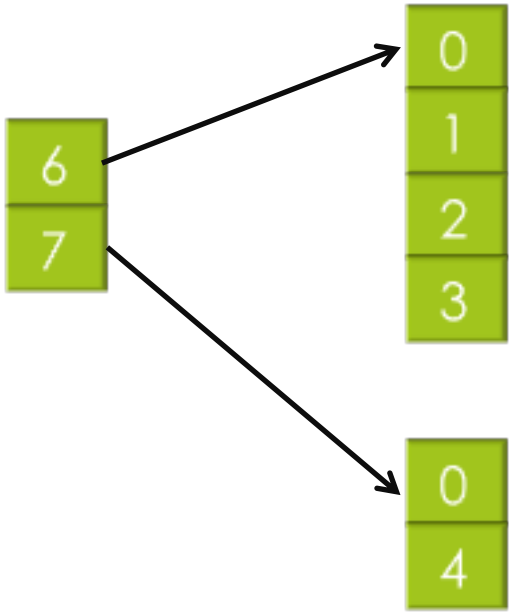
teaches

Subject	Object
C. Ré	Prog.
C. Ré	DB
C. Ré	ML
C. Ré	Theory
K. Olukotun	Prog.
K. Olukotun	Arch.

teaches

S	O
6	0
6	1
6	2
6	3
7	0
7	4

Trie
teaches(S,O)



Dictionary Encoded ID's for each node

Sets could be
dense, sparse, or in between

Two types of Set Representation

Unsigned Integer (uint)

- Standard CSR representation
- Works well with sparse data
  - 4 comparisons per cycle using SSE intrinsic
    - Next processor generations even faster!

SIGMOD '16 contains more types…

Bitvector

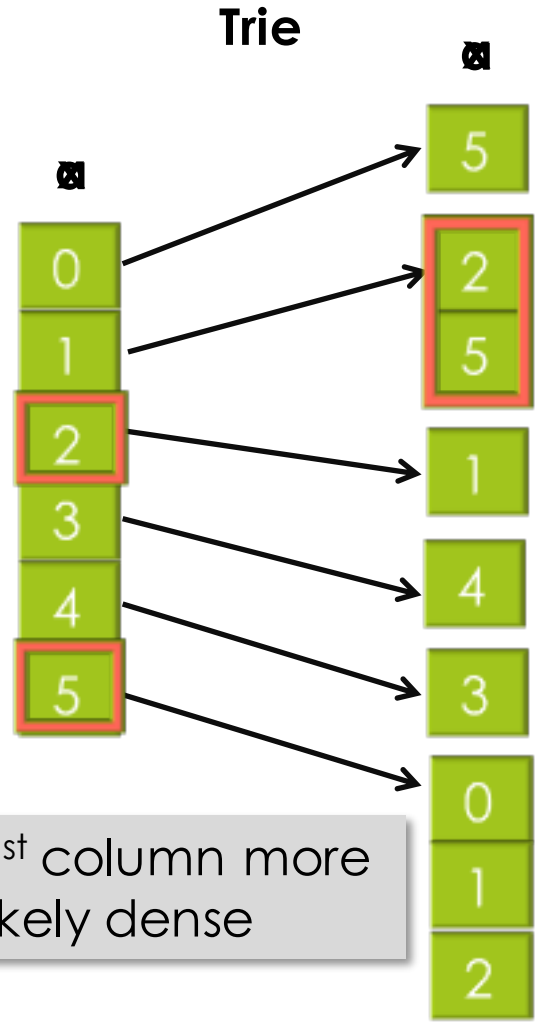
- Works well with dense data
  - Can provide compression or increase memory usage
  - 256 comparisons per cycle
    - Intersections to use wide AVX AND instructions
    - Next processor generation 2x faster

Index Layout

LUBM 14: ***select*** *x* ***from*** *R* ***where*** *a =‘1’*

- **Simple selection query**
  - **Consider attribute ordering [x,a]**
  - **Consider attribute ordering [a,x]**

Insight: Pushing down selections within a GHD node
Layout: 7.92x
Order: 234x



EMPTYHEADED

Experiments.

LUBM Benchmark

		Best	EmptyHeaded	TripleBit	RDF3X	MonetDB	LogicBlox
Theoretical Advantage	Q2	973.95 ms	**1x**	2.38x	1.92x	8.79x	1.52x
	Q9	581.37 ms	**1x**	3.53x	6.63x	24.29x	1.35x
No Theoretical Advantage	Q13	0.87 ms	**1x**	48.90x	35.49x	86.18x	102.77x
	Q14	3.00 ms	1.90x	54.47x	**1x**	313x	325x
	Q7	6.00 ms	3.18x	8.53x	**1x**	573x	6.52x
	Q8	78.50 ms	9.83x (3.04x)	**1x**	3.07x	207x	5.03x

Q2 and Q9 are the bottleneck queries!

A General Engine

- Graph Queries
  - Triangle: **5.17x**
  - PageRank: **7.45x**
  - SSSP: **9.47x**

Speedup to PowerGraph.

- Linear Algebra Kernels
  - Sparse Matrix: **1.01x**
  - S. Matrix D. Vector: **0.58x**
  - D. Matrix D. Vector: **0.88x**
  - D. Matrix D. Matrix: **0.75x**

Speedup to Intel MKL.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

Conclusion

- Worst-case optimal join algorithms actually work in practice; speedups for RDF
- EmptyHeaded engine
  - **Outperforms** RDF3X and TripleBit by **2-6x** with theoretical advantage (cyclic queries).
  - **Competes** within an order of magnitude on acyclic queries and sometimes outperforms!

Theory translates to empirical advantages!

cf. Aberger et al. - SIGMOD '16

caberger@stanford.edu
www.stanford.edu/~caberger
https://github.com/HazyResearch/EmptyHeaded

EMPTYHEADED