A Reachability Index for Recursive Label-Concatenated Graph Queries

Chao Zhang[1,*], Angela Bonifati[1], Hugo Kapp[2], Vlad Ioan Haprian[2], and Jean-Pierre Lozi[3]

[1]Lyon 1 University, Lyon, France
[2]Oracle Labs, Zurich, Switzerland
[3]Inria, Paris, France

ICDE 2023

Université Claude Bernard Lyon 1

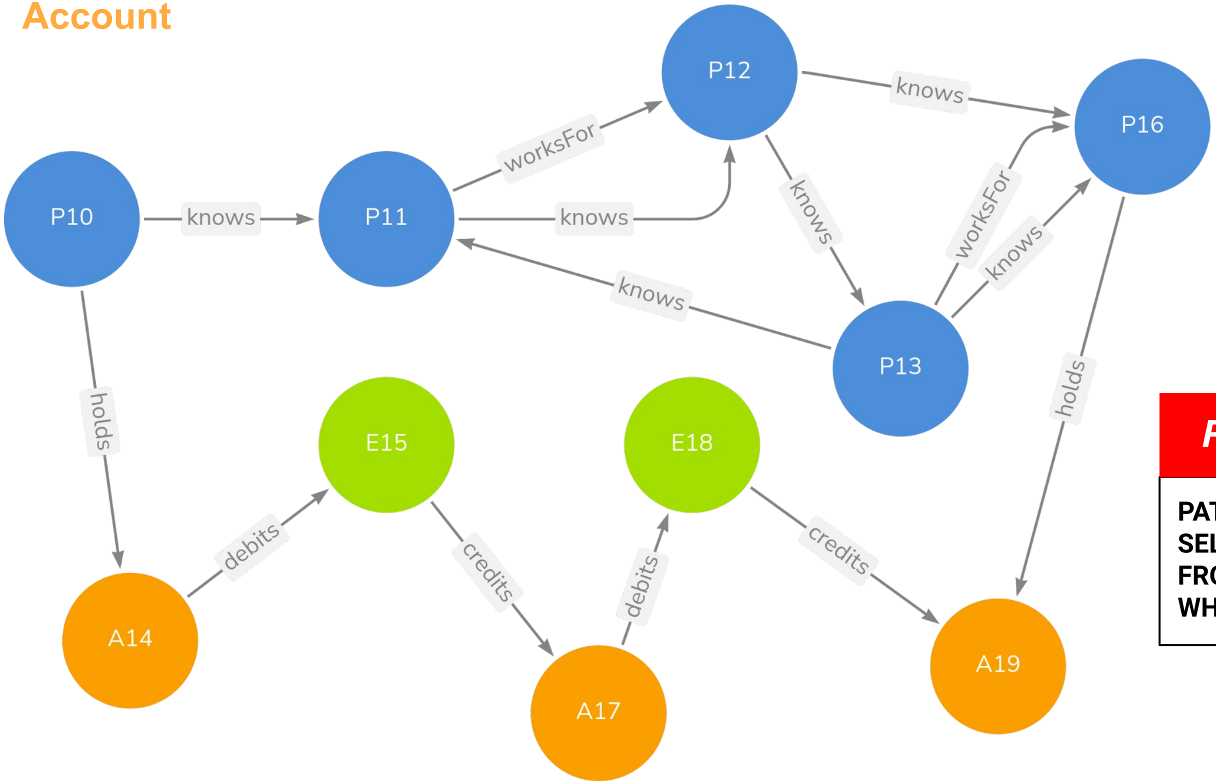
Oracle Labs PGX

Inria

*Currently at the University of Waterloo

An interleaved social and professional network

Person
External Entity
Account



Money laundering analysis:

Do accounts 14 and 19 have the repeated outside-inside money transferring pattern?

PGQL

```
PATH out_in AS () -[:debits-]> () -[:credits]-> ()
SELECT *
FROM MATH (s) -/:out_in+/-> (t)
WHERE ID(s) = 14 AND ID(t) = 19
```

RLC (recursive label-concatenated) queries

- Path constraints
  - A **concatenation** of edge labels under the Kleene plus, i.e., $(l_1, \ldots, l_k)^+$
- RLC query (s, t, L^+), $L = (l_1, \ldots, l_k)$, checks
  - whether there is a path from vertex s to vertex t, and
  - whether the edge label sequence of the path **matches the path constraint L^+**
- No constraint on path length
  - Paths selected by RLC query can have an arbitrary length
- Boolean query
  - Returns either True or False

RLC queries in mainstream graph processing systems

RLC queries **cannot** be expressed in

- Cypher of Neo4j (v4.3)
- GSQL of TigerGraph (v3.3)

RLC queries **can** be expressed in

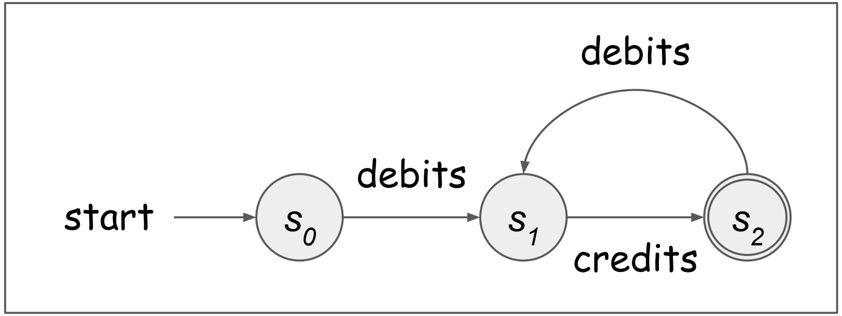
- SPARQL 1.1, supported by Virtuoso, Apache Jena, etc
- PGQL of Oracle PGX
- Gremlin, supported by TinkerPop-Enabled Graph System, e.g., Amazon Neptune

RLC queries can be expressed in **GQL** or **SQL/PGQ** [Deu22]

[Deu22] A. Deutsch et al. Graph Pattern Matching in GQL and SQL/PGQ. SIGMOD Conference 2022: 2246-2258

RLC query processing

- Path semantics
  - Simple paths: non-repeated vertices or edges
  - **Arbitrary paths**: vertices or edges can repeat
- Building an FA (Finite Automata) based on the path constraint
- Query processing: online traversal guided by an FA, e.g., BFS guided by an FA

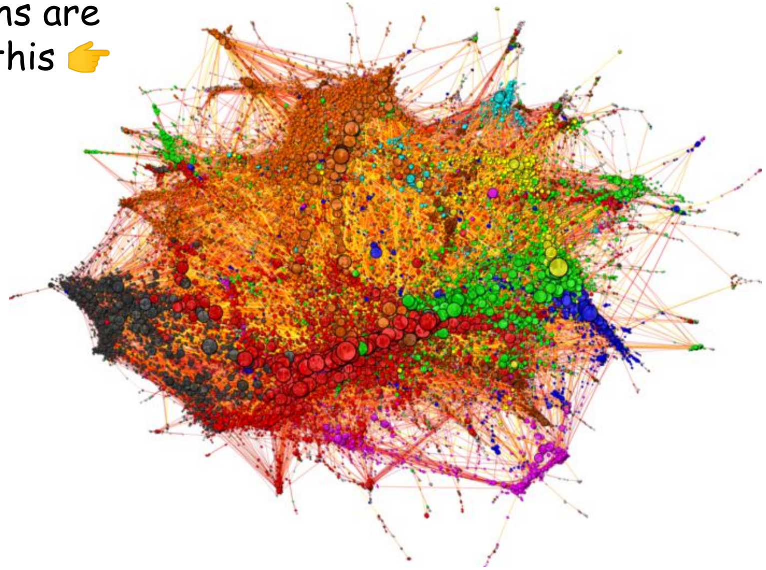


FA of (debits, credits)+

RLC query processing

- Problem
  - real-world graphs are large
- RLC queries often timed out [Bon19]
- **Building an index for online RLC queries**
  - Fast query processing
  - Efficient index computation and storage

Real-world graphs are like this 👉



Music recommendation graph: http://sixdegrees.hu/last.fm/interactive_map.html

[Bon19] A. Bonifati et al. Navigating the Maze of Wikidata Query Logs. WWW 2019: 127-138

Related works: reachability indexes

Plain reachability indexes

- **Queries**: checking the existence of a path only
- **Indexes**: Tree Cover, 2-Hop labeling, Dual labeling, TFL, TOL, GRAIL, BFL, etc

Label-constrained reachability indexes

- **Queries**: checking the existence of a path and whether the edge-label **set** of the path is a **subset** of a given edge-label **set**
- **Indexes**: Landmark Index, P2H, etc

Infeasible for RLC queries due to **missing the support for evaluating path constraints**

Infeasible for RLC queries due to **different path constraints,** i.e., set vs sequence

RLC Index

Index structure

Vertex v	Lout(v)	Lin(v)
…	…	…

Lout(v): recording reachability information **from** v

Lin(v): recording reachability information **to** v

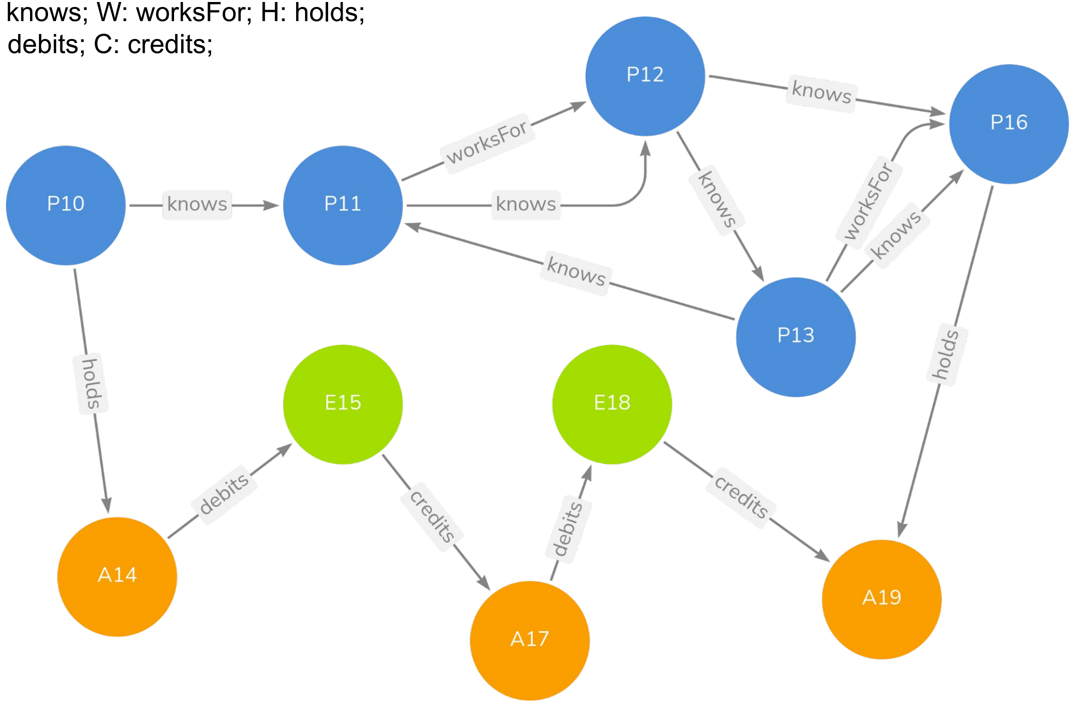
Schema of Lout(v) or Lin(v): (u, mr), where mr is a succinct path label sequence, defined later on

E.g., if (u, mr) is in Lout(v), then there is a path from v to u with the succinct path label sequence is mr

Algorithm 1: Query Algorithm.

procedure Query (s, t, L^+)
  if $\exists(t, L) \in \mathcal{L}_{out}(s)$ **or** $\exists(s, L) \in \mathcal{L}_{in}(t)$ **then**
    return *true*;
  for $(I', I'') \in \mathtt{mergeJoin}(\mathcal{L}_{out}(s), \mathcal{L}_{in}(t))$ **do**
    if $I'.mr = L$ **and** $I''.mr = L$ **then**
      return *true*;
  return *false*;

K: knows; W: worksFor; H: holds;
D: debits; C: credits;



Q(A14, A19, (debits, credits)+)

True, because of (14, DC) in $L_{in}(19)$

Q(P10, P16, (knows, worksFor)+)

True, because of (12, KW) in $L_{out}(10)$ and (12, KW) in $L_{in}(16)$

RLC Index

V	$L_{in}(v)$	$L_{out}(v)$
10	∅	(11, K), (12, KW), (14,H), (15, HD)
11	∅	(11, K)
12	(11, K), (11, W)	(11, K)
13	(11, WK), (11, K)	(11, K), (12, KW)
14	∅	∅
15	(14, D)	∅
16	(11, WK), (11, K), (12, KW), (13, W)	∅
17	(14, DC), (15, C)	∅
18	(15, CD), (17, D)	∅
19	(12, KH), (13, KH), (13, WH), (16, H), (14, DC), (17, DC), (18, C)	∅

Challenge

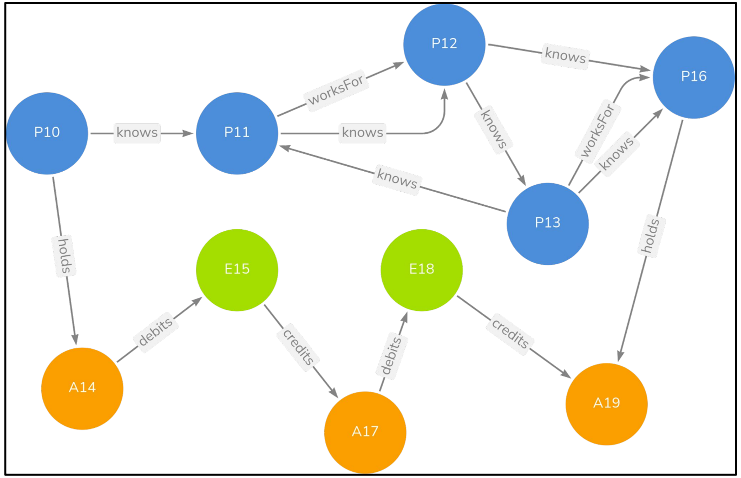
how to build RLC Index

Challenge C1: infinite edge-label sequences

Challenge C2: efficient indexing algorithm

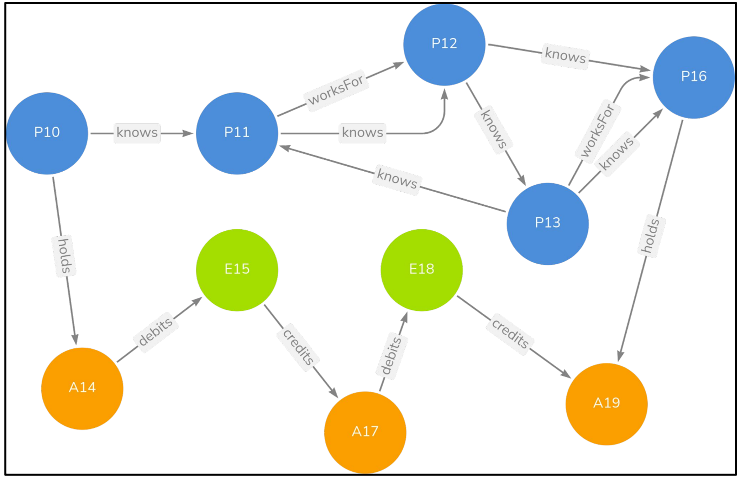
Indexing edge-label sequence

- Edge-label sequence is **necessary**
- Example
  - The sequence (debits, credits, debits, credits) is necessary for query (A14, A19, (debits, credits)+)
- Succinct representation: MR (minimum repeat)
- Example
  - MR((debits, credits, debits, credits)) = (debits, credits)



Indexing edge-label sequence

- Question: how many MRs from P11 to P13
  - Infinite due to the cycle with P11, P12, and P13
- Observation of real-world RLC queries
  - The length of recursive concatenation is bounded, i.e., $(l_1, \ldots, l_k)^+$, where k is bounded
- Question: given k ≤ 2, for P11 and P13
  - how many MRs of length up to 2
  - how to compute all the MRs



KBS (kernel-based search)

- Intuition: generating **kernels** on the fly

  guiding the computation of MRs using kernels

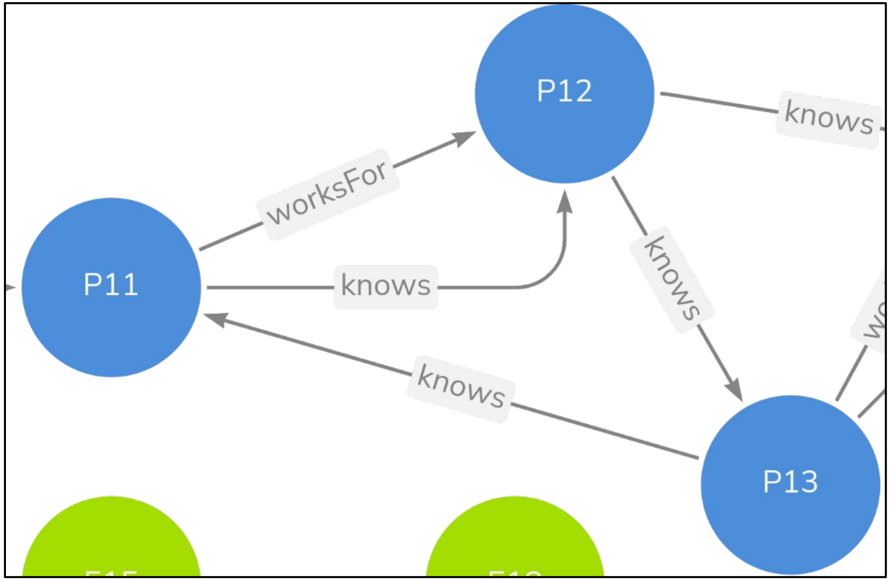
- Example of kernel
  - Label sequence: (l_1, l_2, l_1, l_2, l_1)
  - Kernel: (l_1, l_2)
- KBS: two-phase search
  - Kernel search
  - Kernel BFS

Kernel

Definition 3: If a label sequence L can be represented as $L = (L')^h \circ L''$, where $h \geq 2$, $L' \neq \epsilon$ and $MR(L') = L'$, and L'' is ϵ or a proper prefix of L', then L has the *kernel* L' and the *tail* L''.

Running example: KBS

Given k ≤ 2, computing all the MRs from P11 to P13



Paths:

(P11, worksFor, P12, knows, P13)

(P11, knows, P12, knows, P13)

~~(P11, worksFor, P12, knows, P13, knows, P11, worksFor, **P12**, knows, P13)~~

~~(P11, worksFor, P12, knows, P13, knows, P11, knows, **P12**, knows, P13)~~

~~(P11, knows, P12, knows, P13, knows, P11, worksFor, **P12**, knows, P13)~~

(P11, knows, P12, knows, P13, knows, P11, knows, **P12**, knows, P13)

Foundations of KBS

Sufficient and necessary conditions

Our conditions are correct

The results are complete

Theorem 1: Given a path p from u to v and a positive integer k, p has a non-empty k-MR if and only if one of the following conditions is satisfied,

- Case 1: $|p| \leq k$. $MR(\Lambda(p))$ is the k-MR of p;
- Case 2: $k < |p| \leq 2k$. If $|MR(\Lambda(p))| \leq k$, $MR(\Lambda(p))$ is the k-MR of p;
- Case 3: $|p| > 2k$. Let x be the intermediate vertex on p, s.t. $|p(u,x)| = 2k$. If $\Lambda(p(u,x))$ has a kernel L' and a tail L'', and $MR(L'' \circ \Lambda(p(x,v))) = L'$, then L' is the k-MR of p.

The proof is included in the full version on arXiv

Lazy KBS vs eager KBS

Lazy KBS: computing kernels when the length of edge label sequence is **2k**

Getting **valid** kernels first, and then using the kernel to guide the search

Eager KBS: computing kernels when the length of edge label sequence is **k**

Getting **valid** and **invalid** kernels first, and then using both of them to guide the search

Eager KBS is more efficient than lazy KBS, because the depth of its traversal is shorter

Eager KBS is also correct, because searches guided by invalid kernels will not reach target

Challenge

how to have RLC index

Challenge C1: infinite edge-label sequence

Challenge C2: efficient indexing algorithm

Indexing algorithm

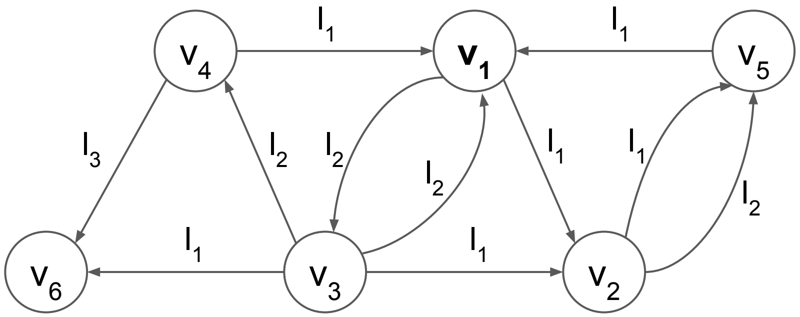
- Performing backward and forward KBS from each vertex
- Each KBS contains two phases
  - kernel search
  - kernel BFS
- During kernel search
  - computing and inserting MRs for each traversal step
- During kernel BFS
  - inserting MRs only if they are same as kernels

Algorithm 2: Indexing Algorithm.

```
procedure kernelBasedSearch(v, k)
    for (L, vSet) ∈ backwardKernelSearch(v, k) do
        backwardKernelBFS(v, vSet, L);
    for (L, vSet) ∈ forwardKernelSearch(v, k) do
        forwardKernelBFS(v, vSet, L);

procedure backwardKernelSearch(v, k)
    q ← an empty queue of (vertex, label sequence);
    q.enqueue(v, ϵ);
    map ← a map of (kernel candidates, vertex set);
    while q is not empty do
        (x, seq) ← q.dequeue();
        for incoming edge e(y, x) to x do
            seq' ← λ(e(y, x)) ∘ seq; L ← MR(seq');
            insert (y, v, L);
            map.get(L).add(x);
            if |seq'| < k then
                q.enqueue(y, seq');
    return map;

procedure insert(s, t, L)
    if aid(t) > aid(s) or Query(s, t, L+) then
        return false;
    else
        add (t, L) into L_out(s);
        return true;

procedure backwardKernelBFS(v, vSet, L)
    q ← an empty queue of (vertex, integer);
    for x ∈ vSet do
        mark x as visited by state 1, q.enqueue(x, |L|);
    while q is not empty do
        (x, i) ← q.dequeue(), i ← i − 1;
        if i = 0 then  i = |L| ;
        label l ← L.get(i);
        for incoming edge e(y, x) to x do
            if l ≠ λ(e(y, x)) or y was visited by state i then
                continue;
            if  i = 1 and insert(y, v, L) then
                continue;
            q.enqueue(y, i); mark y visited by state i;
```

RLC indexing with k = 2

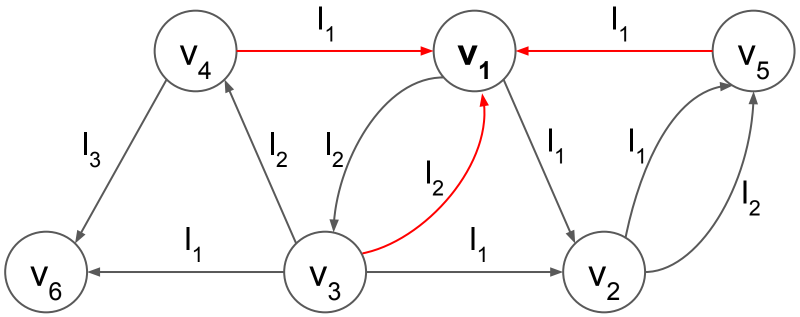
Backward KBS from v_1
⇨ Kernel search
Kernel BFS



V	Lout(v)	Lin(v)
v_1		
v_2		
v_3		
v_4		
v_5		
v_6		

RLC indexing with k = 2

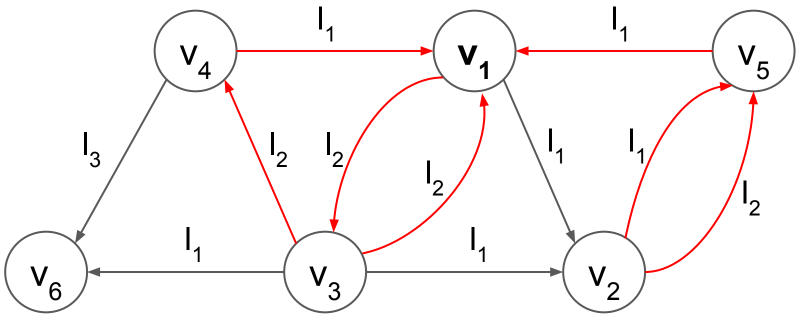
Backward KBS from v_1
Kernel search
Kernel BFS



V	Lout(v)	Lin(v)
v_1		
v_2		
v_3	(v_1, l_2)	
v_4	(v_1, l_1)	
v_5	(v_1, l_1)	
v_6		

RLC indexing with k = 2

Backward KBS from v_1
Kernel search
Kernel BFS



Kernel search terminates

Three kernels have been generated

1. (l_1) with a set of frontier vertices $\{v_4, v_5, v_2\}$
2. (l_2) with a set of frontier vertices $\{v_3, v_1\}$
3. (l_2, l_1) with a set of frontier vertices $\{v_3, v_2\}$

V	Lout(v)	Lin(v)
v_1	(v_1, l_2)	
v_2	$(v_1, l_1), (v_1, (l_2,l_1))$	
v_3	$(v_1, l_2), (v_1, (l_2,l_1))$	
v_4	(v_1, l_1)	
v_5	(v_1, l_1)	
v_6		

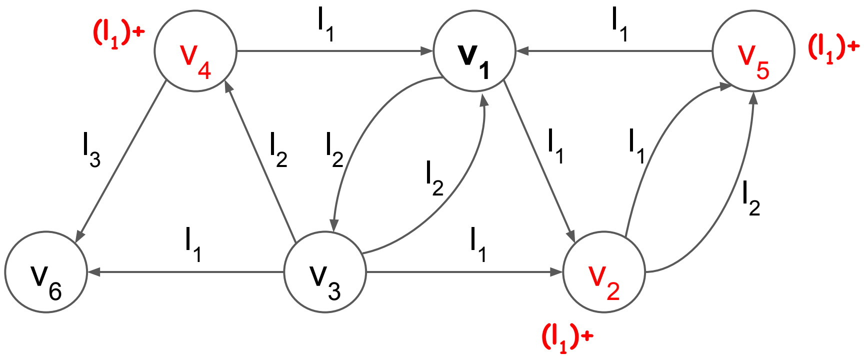
RLC indexing with k = 2

Backward KBS from v_1

Kernel search

⇨ Kernel BFS

The first kernel $(l_1)+$ with a set of frontier vertices $\{v_4, v_5, v_2\}$



V	Lout(v)	Lin(v)
v_1	(v_1, l_2)	
v_2	$(v_1, l_1), (v_1, (l_2,l_1))$	
v_3	$(v_1, l_2), (v_1, (l_2,l_1))$	
v_4	(v_1, l_1)	
v_5	(v_1, l_1)	
v_6		

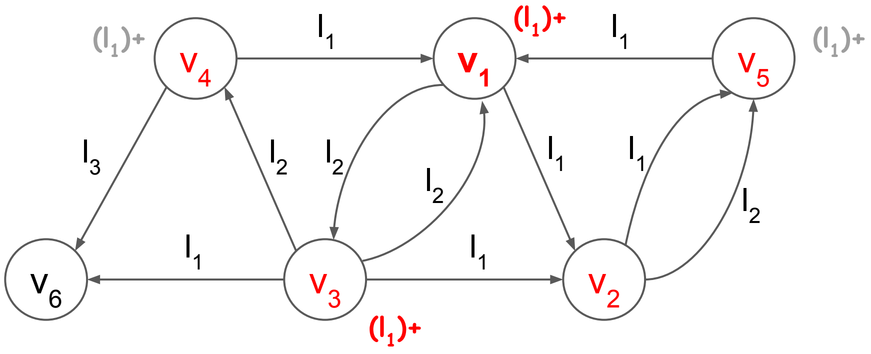
RLC indexing with k = 2

Backward KBS from v_1

Kernel search

Kernel BFS

The first kernel $(l_1)+$ with a set of frontier vertices $\{v_4, v_5, v_2\}$



The BFS at v_4 terminates because the incoming edge label l_2 is an invalid state for kernel $(l_1)+$

The BFS at v_5 terminates because the only incoming neighbour with label l_1, i.e., v_2, has already been visited

V	Lout(v)	Lin(v)
v_1	(v_1, l_2), (v_1, l_1)	
v_2	(v_1, l_1), $(v_1, (l_2, l_1))$	
v_3	(v_1, l_2), $(v_1, (l_2, l_1))$, (v_1, l_1)	
v_4	(v_1, l_1)	
v_5	(v_1, l_1)	
v_6		

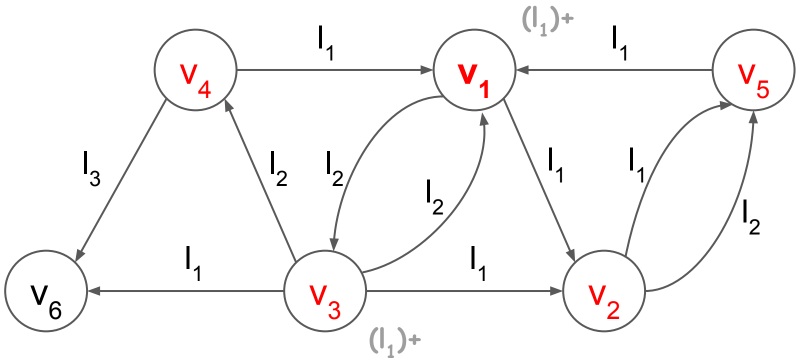
RLC indexing with k = 2

Backward KBS from v_1

Kernel search

Kernel BFS

The first kernel $(l_1)+$ with a set of frontier vertices $\{v_4, v_5, v_2\}$



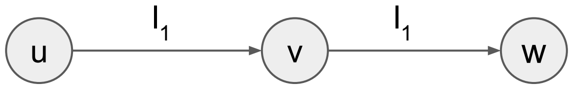
The BFS at v_1 terminates because incoming neighbours with l_1, i.e., v_4 and v_5, have already been visited

The BFS at v_3 terminates because there is not incoming edges with label l_1

The kernel BFS with $(l_1)+$ terminates

V	Lout(v)	Lin(v)
v_1	$(v_1, l_2), (v_1, l_1)$	
v_2	$(v_1, l_1), (v_1, (l_2,l_1))$	
v_3	$(v_1, l_2), (v_1, (l_2,l_1)),$ (v_1, l_1)	
v_4	(v_1, l_1)	
v_5	(v_1, l_1)	
v_6		

Accessing order



- Accessing order:
  - The order of vertices, in which the indexing algorithm is performed
- Intuition:
  - Starting from the "middle"
- Example:
  - Less index entries with the order (v, u, w)
- Strategy:
  - Sorting in (out-degree(v) + 1) x (in-degree(v) + 1)

The case with (u, w, v)

Vertex v	Lout(v)	Lin(v)
u	(w, l_1)	∅
v	(w, l_1)	(u, l_1)
w	∅	(u, l_1)

The case with (v, u, w)

Vertex v	Lout(v)	Lin(v)
u	(v, l_1)	∅
v	∅	∅
w	∅	(v, l_1)

Pruning rules

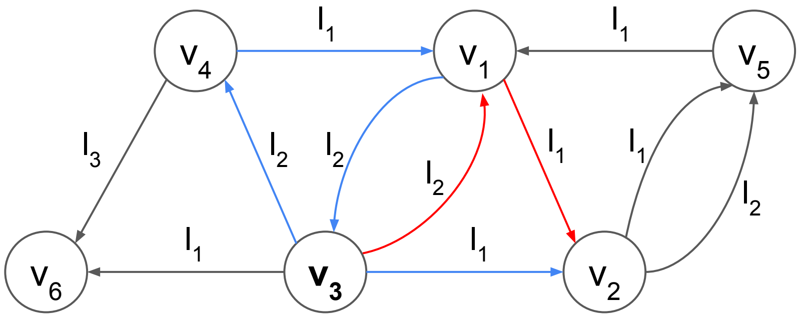
- Efficient indexing:
  - When to skip index entries
  - When to terminate the KBS from a vertex early
- Intuition:
  - Concatenating edge-label sequences of sub-paths as much as possible
- Three pruning rules for efficient indexing
  - PR1 and PR2: skipping redundant entries
  - PR3: terminating the search early

Pruning Rules

- ***PR1:*** *If the k-MR of an index entry that needs to be recorded can be acquired from the current snapshot of the* $\mathtt{RLC}$ *index, then the index entry can be skipped.*
- ***PR2:*** *If vertex* v_i *is visited by the backward KBS performed from vertex* $v_{i'}$ *s.t.* $aid(v_{i'}) > aid(v_i)$*, then the corresponding index entry can be skipped.*
- ***PR3:*** *If vertex* v_i *is visited by the kernel-BFS phase of a backward KBS performed from vertex* $v_{i'}$*, and PR1 (or PR2) is triggered, then vertex* v_i *and all the vertices in* $in(v_i)$ *are skipped.*

Pruning Rules

- ***PR1:*** *If the k-MR of an index entry that needs to be recorded can be acquired from the current snapshot of the* RLC *index, then the index entry can be skipped.*



The forward KBS from v_3 can visit v_2, such that it tries to creat $(v_3, (l_2, l_1))$ in $Lin(v_2)$

However, there already exists $(v_1, (l_2, l_1))$ in both $Lout(v_3)$ and $Lin(v_2)$, such that the index entry that needs to be inserted can be pruned

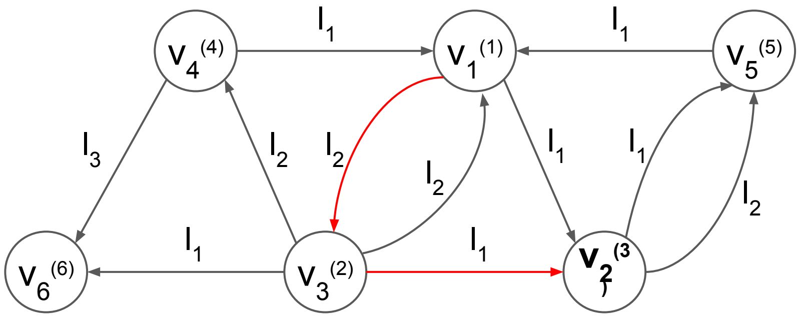
The snapshot of the RLC index after performing KBS from v_1

V	Lout(v)	Lin(v)
v_1	$(v_1, l_2), (v_1, l_1), (v_1, (l_2, l_1))$	∅
v_2	$(v_1, l_1), (v_1, (l_2, l_1))$	$(v_1, l_1), (v_1, (l_2, l_1))$
v_3	$(v_1, l_2), (v_1, (l_2, l_1)), (v_1, l_1)$	$(v_1, l_2), (v_1, (l_1, l_2))$
v_4	(v_1, l_1)	(v_1, l_2)
v_5	(v_1, l_1)	$(v_1, (l_1, l_2)), (v_1, l_1),$
v_6		$(v_1, (l_2, l_1))$

Pruning Rules

- ***PR2:*** *If vertex v_i is visited by the backward KBS performed from vertex $v_{i'}$ s.t. $aid(v_{i'}) > aid(v_i)$, then the corresponding index entry can be skipped.*

aid: accessing ID, e.g., $aid(v_3) = 2$



The backward KBS from v_2 can visit v_1, such that it tries to creat $(v_2, (l_2, l_1))$ in $Lout(v_1)$

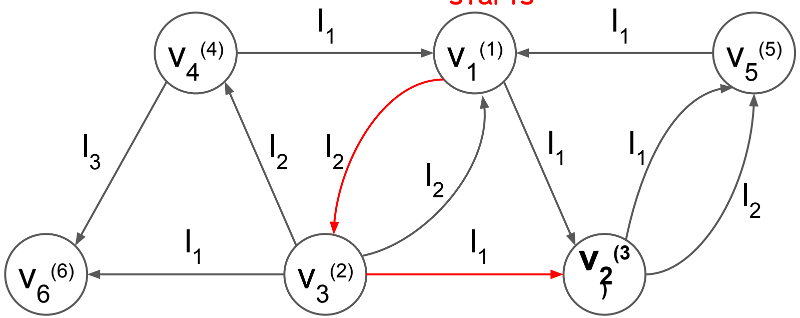
However, $aid(v_2) > aid(v_1)$, such that the index entry that needs to be inserted can be pruned

The snapshot of the RLC index after performing KBS from v_1 and v_3

V	Lout(v)	Lin(v)
v_1	$(v_1, l_2), (v_1, l_1), (v_1, (l_2, l_1))$	$\varnothing$
v_2	$(v_1, l_1), (v_1, (l_2, l_1))$	$(v_1, l_1), (v_1, (l_2, l_1))$
v_3	$(v_1, l_2), (v_1, (l_2, l_1)), (v_1, l_1), (v_3, (l_1, l_2))$	$(v_1, l_2), (v_1, (l_1, l_2))$
v_4	$(v_1, l_1), (v_3, (l_1, l_2))$	(v_1, l_2)
v_5	$(v_1, l_1), (v_3, (l_1, l_2))$	$(v_1, (l_1, l_2)), (v_1, l_1), (v_3, (l_1, l_2))$
v_6	$\varnothing$	$(v_1, (l_2, l_1)), (v_3, l_1), (v_3, (l_2, l_3))$

Pruning Rules

- ***PR3:*** *If vertex v_i is visited by the kernel-BFS phase of a backward KBS performed from vertex $v_{i'}$, and PR1 (or PR2) is triggered, then vertex v_i and all the vertices in $in(v_i)$ are skipped.*



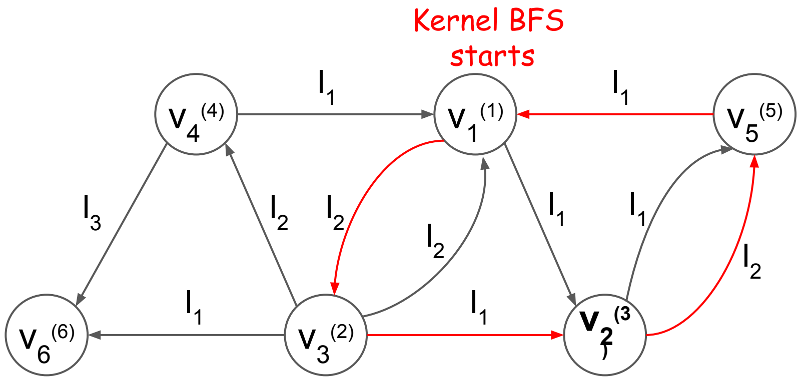
The backward KBS from v_2 is transformed from kernel search to kernel BFS guided by $(l_2, l_1)^+$ after visiting v_1

The snapshot of the RLC index after performing KBS from v_1 and v_3

V	Lout(v)	Lin(v)
v_1	(v_1, l_2), (v_1, l_1), $(v_1, (l_2, l_1))$	∅
v_2	(v_1, l_1), $(v_1, (l_2, l_1))$	(v_1, l_1), $(v_1, (l_2, l_1))$
v_3	(v_1, l_2), $(v_1, (l_2, l_1))$, (v_1, l_1), $(v_3, (l_1, l_2))$	(v_1, l_2), $(v_1, (l_1, l_2))$
v_4	(v_1, l_1), $(v_3, (l_1, l_2))$	(v_1, l_2)
v_5	(v_1, l_1), $(v_3, (l_1, l_2))$	$(v_1, (l_1, l_2))$, (v_1, l_1), $(v_3, (l_1, l_2))$
v_6	∅	$(v_1, (l_2, l_1))$, (v_3, l_1), $(v_3, (l_2, l_3))$

Pruning Rules

- ***PR3:*** *If vertex v_i is visited by the kernel-BFS phase of a backward KBS performed from vertex $v_{i'}$, and PR1 (or PR2) is triggered, then vertex v_i and all the vertices in $in(v_i)$ are skipped.*



The backward KBS from v_2 is transformed from kernel search to kernel BFS guided by $(l_2, l_1)^+$ after visiting v_1

When the kernel BFS visits v_2, it tries to creat $(v_2, (l_2, l_1))$ in Lout(v_2)

However, there exists $(v_1, (l_2, l_1))$ in both Lout(v_2) and Lin(v_2), i.e., PR1 can be triggered

Then, the kernel BFS can terminate

The snapshot of the RLC index after performing KBS from v_1 and v_3

V	Lout(v)	Lin(v)
v_1	(v_1, l_2), (v_1, l_1), $(v_1, (l_2, l_1))$	∅
v_2	(v_1, l_1), $(v_1, (l_2, l_1))$	(v_1, l_1), $(v_1, (l_2, l_1))$
v_3	(v_1, l_2), $(v_1, (l_2, l_1))$, (v_1, l_1), $(v_3, (l_1, l_2))$	(v_1, l_2), $(v_1, (l_1, l_2))$
v_4	(v_1, l_1), $(v_3, (l_1, l_2))$	(v_1, l_2)
v_5	(v_1, l_1), $(v_3, (l_1, l_2))$	$(v_1, (l_1, l_2))$, (v_1, l_1), $(v_3, (l_1, l_2))$
v_6	∅	$(v_1, (l_2, l_1))$, (v_3, l_1), $(v_3, (l_2, l_3))$

Foundations of the indexing algorithm

No redundant index entries

Theorem 2: With pruning rules, the `RLC` index is condensed.

Correct and complete index

Theorem 3: Given an edge-labeled graph G and the `RLC` index of G with a positive integer k built by Algorithm 2, there exists a path from vertex s to vertex t in G which satisfies a label constraint $L^+, |L| \leq k$, if and only if one of the following condition is satisfied

(1) $\exists(x, L) \in \mathcal{L}_{out}(s)$ and $\exists(x, L) \in \mathcal{L}_{in}(t)$;
(2) $\exists(t, L) \in \mathcal{L}_{out}(s)$, or $\exists(s, L) \in \mathcal{L}_{in}(t)$.

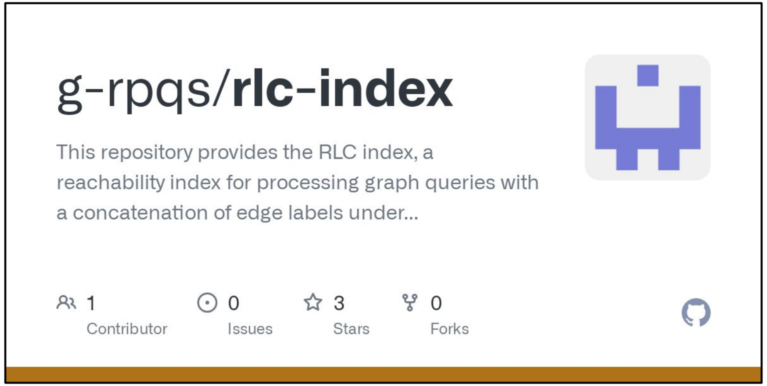
The proofs are included in the full version on arXiv

Experimental setup

- Baselines
  - ETC (extended transitive closure): for every pairs of vertices, recording all the k-MRs
  - Online traversal: BFS and Bidirectional BFS
- 13 highly dense real-world graphs
- Workloads
  - 1000 true-queries and 1000 false-queries
- Parameter k
  - We start with k = 2, which is the real-world case
  - Then, we analyze the cases of k = 2, 3, 4
- Implementation: Java 11
- Setting
  - 8 VCPUs of 2.4GHz; 128GB main memory
  - Heap size of JVM: 120GB

TABLE III
OVERVIEW OF REAL-WORLD GRAPHS.

Dataset	$\|V\|$	$\|E\|$	$\|\mathbb{L}\|$	Synthetic Labels	Loop Count	Triangle Count
Advogato (AD)	6K	51K	3		4K	98K
Soc-Epinions (EP)	75K	508K	8	√	0	1.6M
Twitter-ICWSM (TW)	465K	834K	8	√	0	38K
Web-NotreDame (WN)	325K	1.4M	8	√	27K	8.9M
Web-Stanford (WS)	281K	2M	8	√	0	11M
Web-Google (WG)	875K	5M	8	√	0	13M
Wiki-Talk (WT)	2.3M	5M	8	√	0	9M
Web-BerkStan (WB)	685K	7M	8	√	0	64M
Wiki-hyperlink (WH)	1.7M	28.5M	8	√	4K	52M
Pokec (PR)	1.6M	30.6M	8	√	0	32M
StackOverflow (SO)	2.6M	63.4M	3		15M	114M
LiveJournal (LJ)	4.8M	68.9M	50	√	0	285M
Wiki-link-fr (WF)	3.3M	123.7M	25	√	19K	30B



https://github.com/g-rpqs/rlc-index

Indexing performance

- Building ETC is not feasible
  - Building ETC timed out in 24 hours or run out of memory except for the AD graph
- Four-orders-of-magnitude improvement
- Effectiveness of the pruning rules
  - Although SO requires more indexing time than LJ and WF, index size of the former is smaller than those of the latter

TABLE IV
INDEXING TIME (IT) AND INDEX SIZE (IS).

Dataset	RLC Index		ETC	
	IT (s)	IS (MB)	IT (s)	IS (MB)
AD	0.7	1.9	2216.1	2798.7
EP	22.6	29.3	-	-
TW	8.1	93.5	-	-
WN	33.1	122.6	-	-
WS	53.5	173.9	-	-
WG	101.3	403.6	-	-
WT	812.9	607.1	-	-
WB	167.1	474.2	-	-
WH	3707.2	1319.1	-	-
PR	3104.1	1212.6	-	-
SO	57072.5	844.2	-	-
LJ	18240.9	6248.1	-	-
WF	51338.7	6467.9	-	-

Query performance

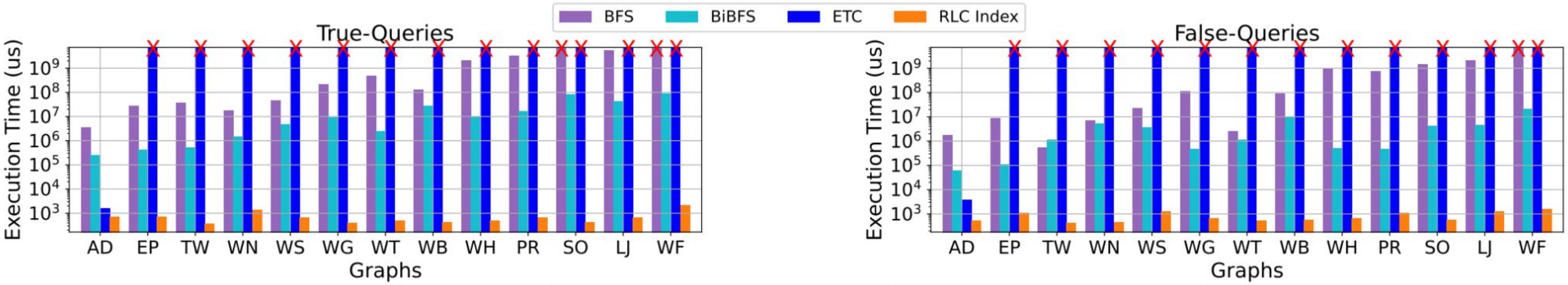


Fig. 3. Execution time of 1000 true-queries or 1000 false-queries on real-world graphs.

Executing 1000 queries using the RLC index takes **1 ms** except the WF graph that is **2 ms**

Up to six-orders-of-magnitude improvement over BFS

Up to four-orders-of-magnitude improvement over bidirectional BFS

Impact of k

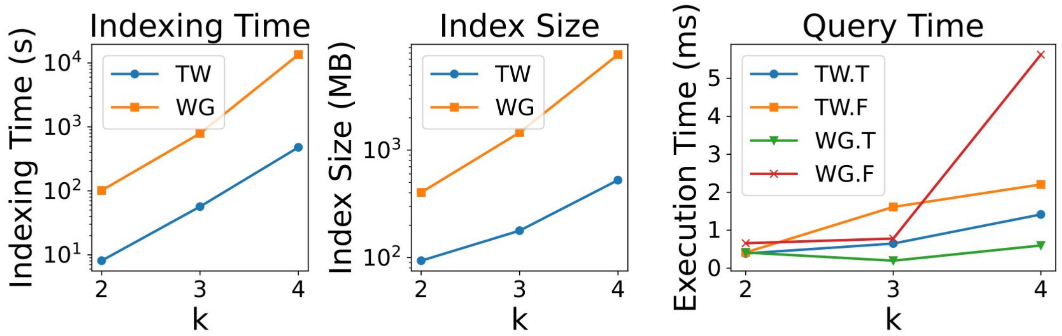


Fig. 4. RLC index performance with different recursive k values.

Both indexing time and index size will increase when the value k increases

Query time also increases a bit due to the large index size

The number of path-constraints or kernels will exponentially grow as the increase of k

Impact of graph characteristics

BA-graphs:
Barabási–Albert model

ER-graphs:
Erdős–Rényi model

Degree-distribution:
- BA-graphs: **skew**
- ER-graphs: **uniform**

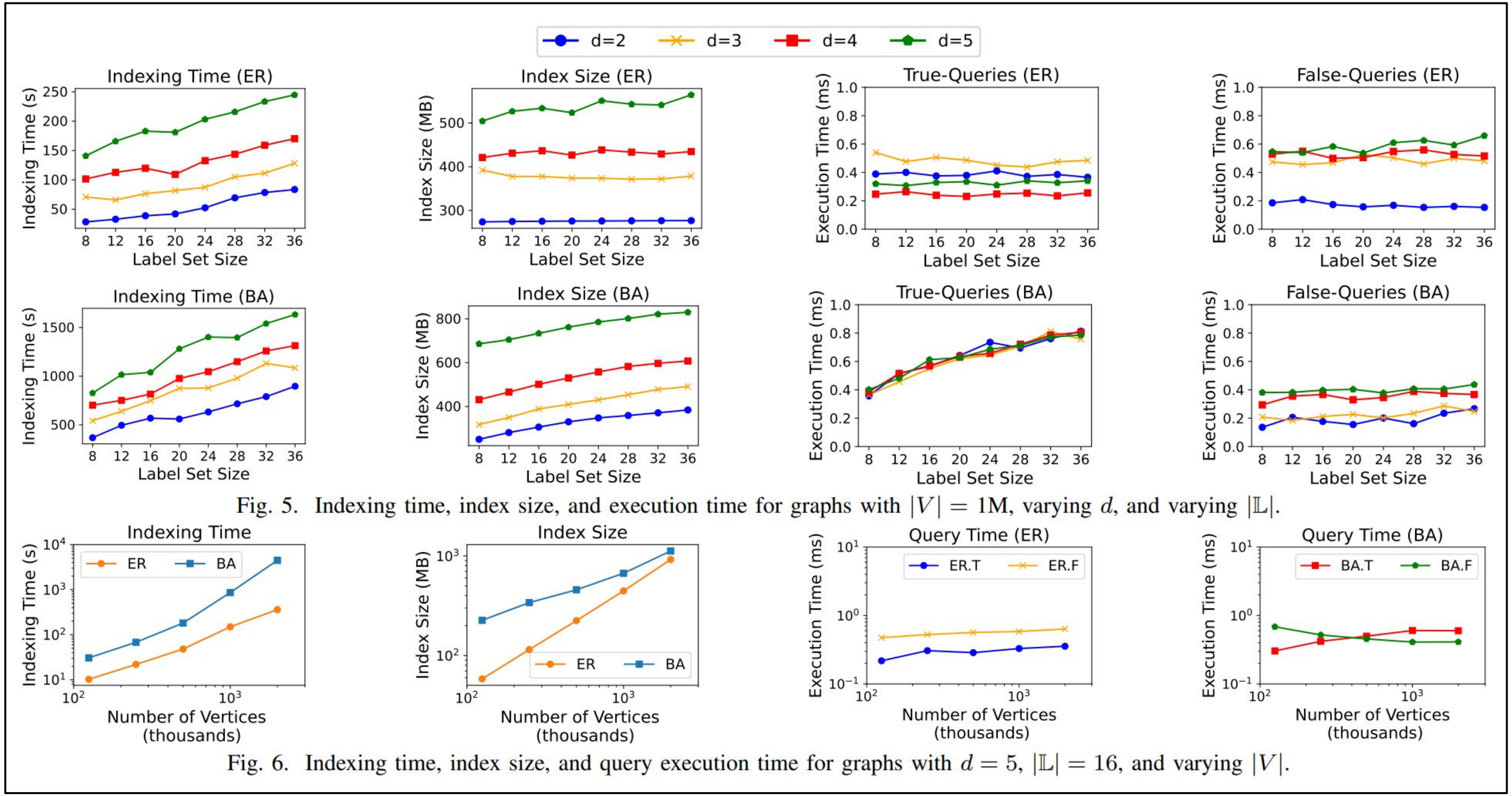


Fig. 5. Indexing time, index size, and execution time for graphs with $|V| = 1M$, varying d, and varying $|\mathbb{L}|$.

Fig. 6. Indexing time, index size, and query execution time for graphs with $d = 5$, $|\mathbb{L}| = 16$, and varying $|V|$.

Comparison with existing systems

- Systems
  - Commercial and open-sourced systems
  - Virtuoso (v7.2.6.3233), Sys1, and Sys2
- Dataset: the WN graph
  - |V|: 325K
  - |E|: 1.4M
- RLC index built with k = 3
  - 5.9 minutes
  - 821 megabytes

TABLE V
SPEED-UPS (SU) AND WORKLOAD SIZE BREAK-EVEN POINTS (BEP) OF THE RLC INDEX OVER GRAPH ENGINES.

Sys.	RLC Query						Extended Query	
	Q1		Q2		Q3		Q4	
	SU	BEP	SU	BEP	SU	BEP	SU	BEP
Sys1	1200x	84100	10400x	34000	18400x	9400	34000x	300
Sys2	3000x	34900	202000x	1700	1300000x	130	104000x	98
Virtuoso	597x	180000	4900x	71700	38100000x	5	-	-

Q1: $(a)^+$

Q2: $(a, b)^+$

Q3: $(a, b, c)^+$

Q4: a^+b^+

BEP indicates when the RLC index should be built

The RLC index built for Q3 can also significantly improve the execution time of Q1, Q2, and Q4 as well

Conclusion

- RLC queries
  - Reachability queries with a path constraint based on the Kleene plus over a concatenation of edge labels
- RLC index
  - Evaluating RLC queries through path concatenation
- Indexing algorithm
  - Backward and forward kernel-based search with pruning rules
- Experimental evaluation
  - RLC index can significantly improve query processing while reduce offline indexing overhead

Thank you and Q&A