GLogS: Interactive Graph Pattern Matching Query at Large Scale

Longbin Lai[1*], Yufan Yang[2*], Zhibin Wang[3], Yuxuan Liu[2], Haotian Ma[2], SiJie Shen[1], Bingqing Lyu[1], Xiaoli Zhou[1], Wenyuan Yu[1], Zhengping Qian[1], Chen Tian[3], Sheng Zhong[3], Yeh-Ching Chung[2] and Jingren Zhou[1]

[1]Alibaba Group, China

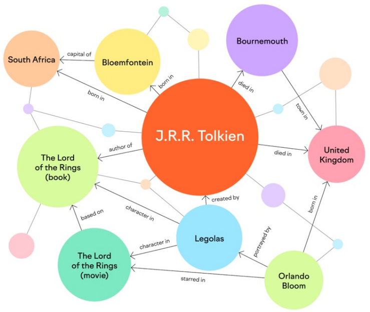
[2]The Chinese University of Hong Kong, Shenzhen

[3]Nanjing University

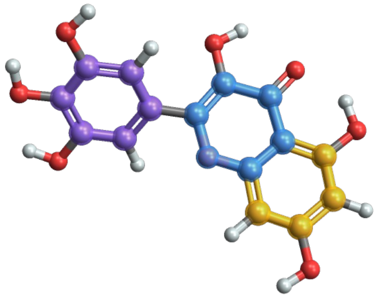
Outline

Graph and Graph Pattern Matching

Social Network Graph

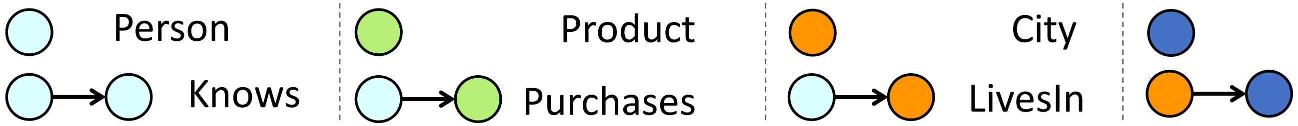


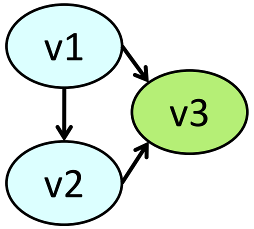
Knowledge Graph



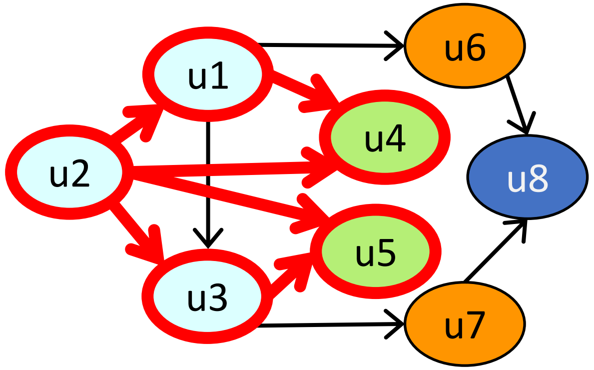
Molecular Graph

Graph is Prevalent!





(a) Pattern p



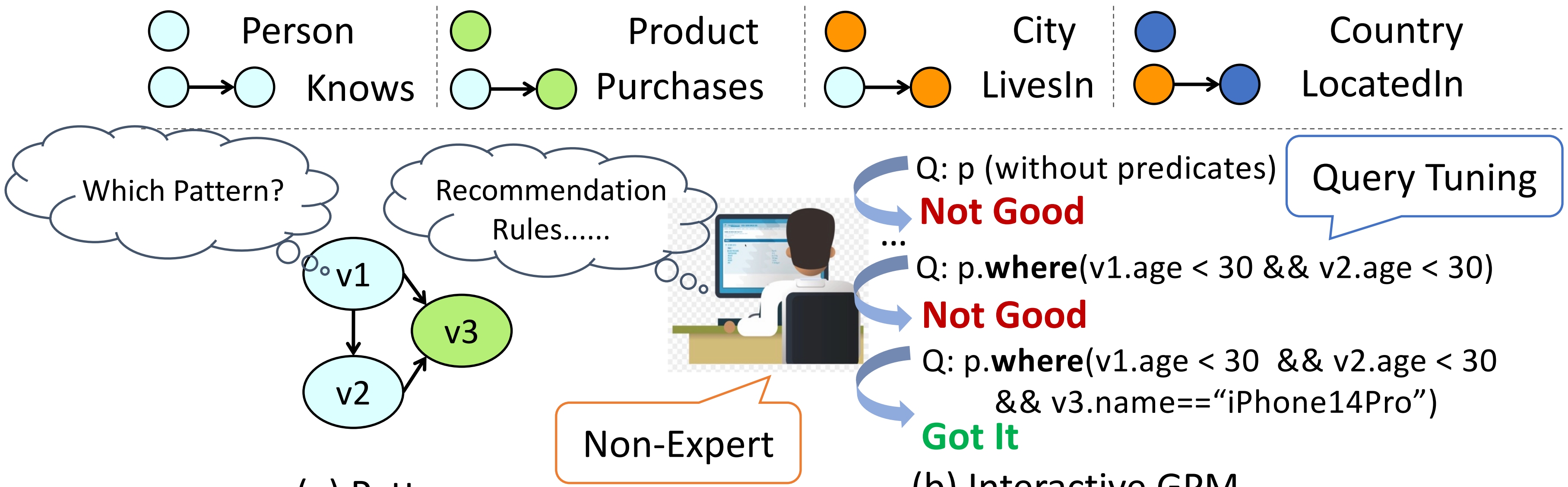
(b) Graph G

v1	v2	v3
u2	u1	u4
u2	u3	u5

(c) The mappings of p in G

Graph Pattern Matching

Interactive Graph Pattern Matching



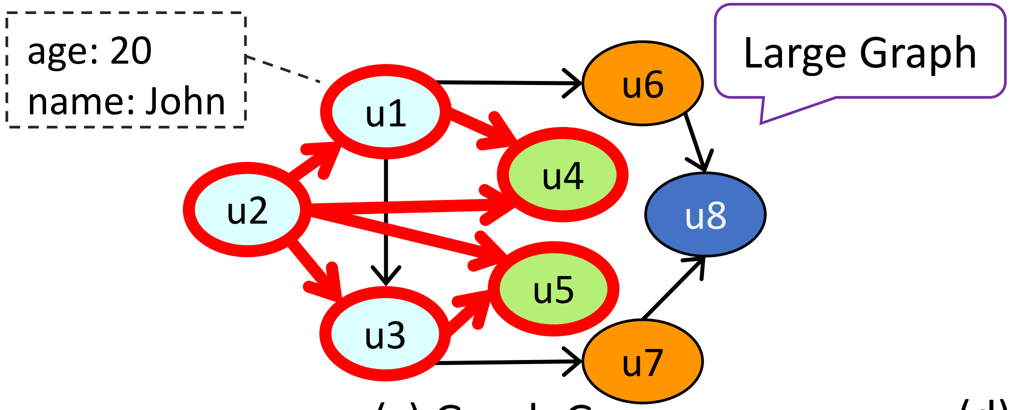
(a) Pattern p

(b) Interactive GPM

Usability

Performance

Scalability



(c) Graph G

v1	v2	v3
u2	u1	u4
u2	u3	u5

(d) The mappings of p in G

iGPM: Requirements and Features

Usability → Feature1: Declarative Language; Feature2: Automatic Optimization

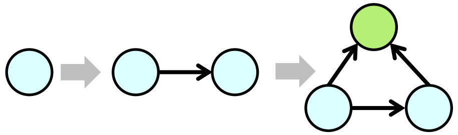
Performance → Feature2: Automatic Optimization; Feature3: Distributed Execution

Scalability → Feature3: Distributed Execution

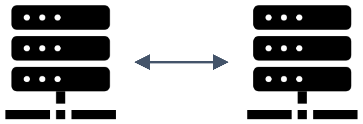
Feature1:
Declarative Language

```
p = g.V().match (
    as('v1').out('Knows').as('v2'),
    as('v1').out('Purchases').as('v3'),
    as('v2').out('Purchases').as('v3')
)
```

Feature2:
Automatic Optimization



Feature3:
Distributed Execution



Are there any existing solutions?

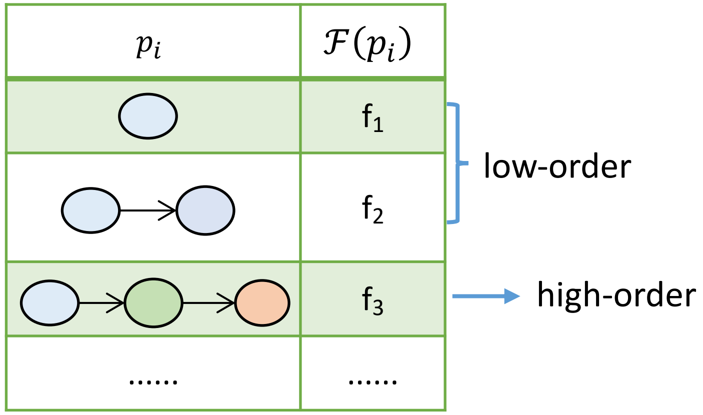
neo4j TigerGraph

iGPM: Existing Solutions and Challenges

neo4j

Feature1: Declarative Language

✓ Support Cypher



Feature2: Automatic Optimization

○ Limited Optimization

Cannot guarantee worst-case optimal
- Only support binary join operation
- large intermediate results size

Only use low-order statistics to estimate plan cost
- Inaccurate plan cost estimation

Bad execution plan!

Challenge1: Optimization
- Guarantee worst-case optimal
- Adopt high-order statistics

Feature3: Distributed Execution

✗ Single Machine Design

iGPM: Existing Solutions and Challenges

TigerGraph

Feature1:
Declarative Language

○ Support GSQL

Feature2:
Automatic Optimization

✕ Manually Query Tuning

Feature3:
Distributed Execution

✓ Distributed Design

However, it requires query pre-installation!

- Pre-install the queries before they can be executed in the TigerGraph
- Involves native code generation and compilation
  - 1~3 minutes per query, too long!

→

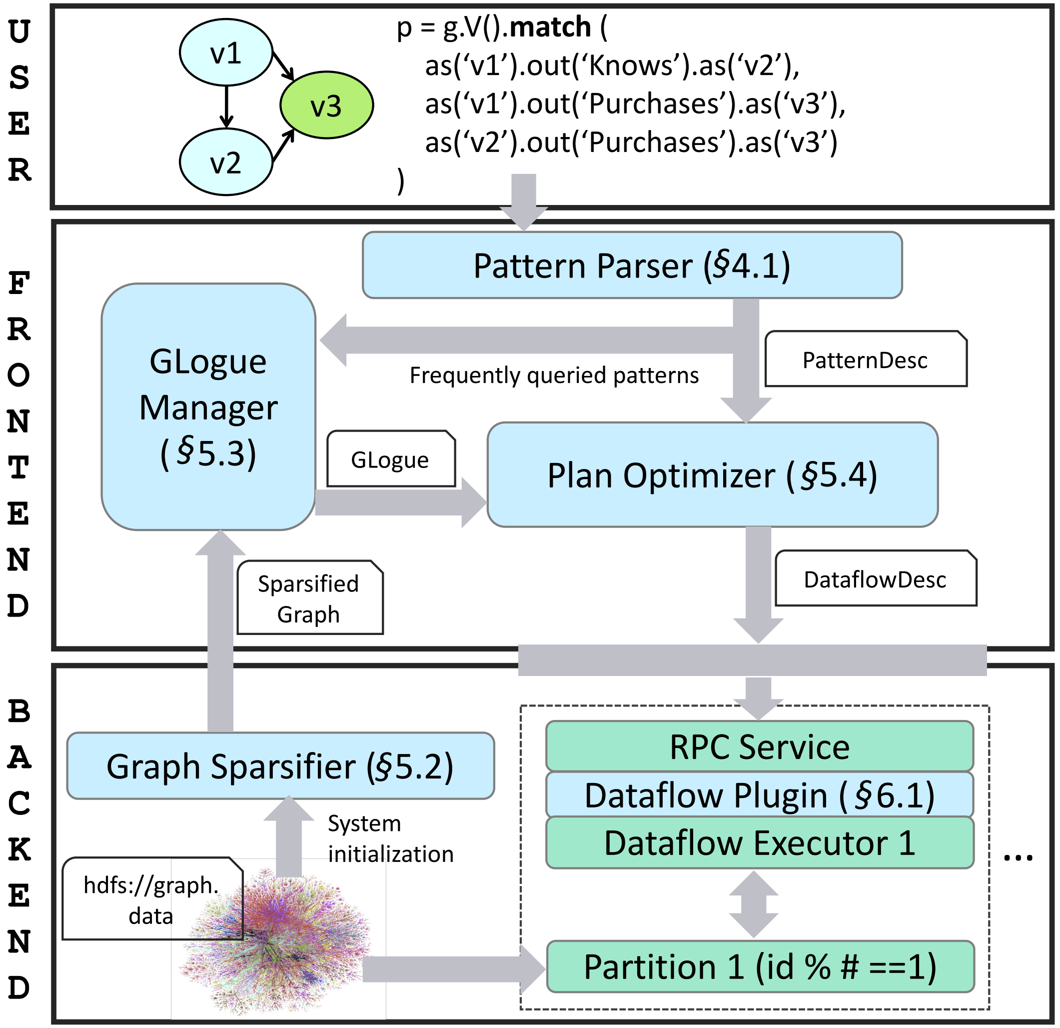
Challenge2: Compilation

- Timeliness
- Support interactive manner

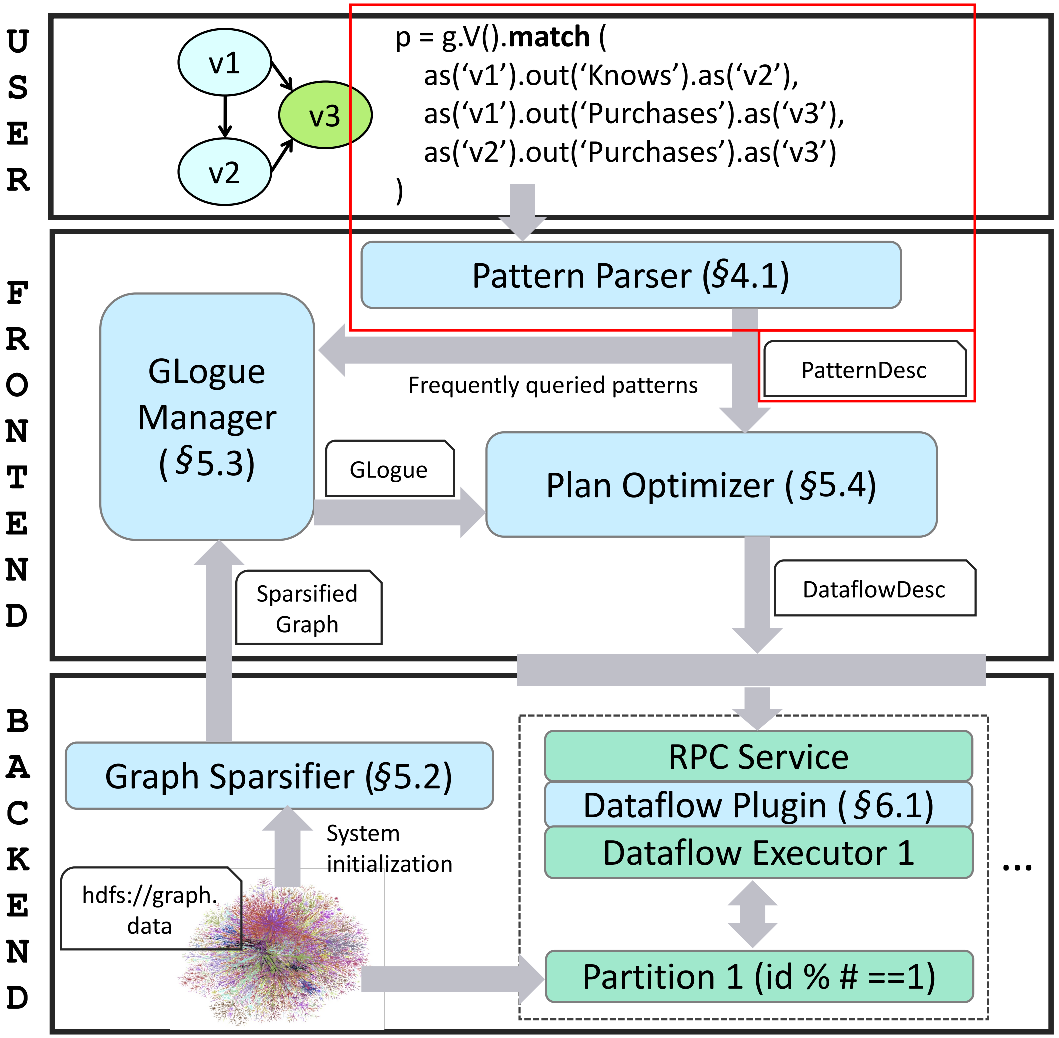
Outline

1. Background and Motivation
2. System Overview
3. Feature Highlights
4. Evaluation
5. Conclusion

System Overview



Frontend: Pattern Parser

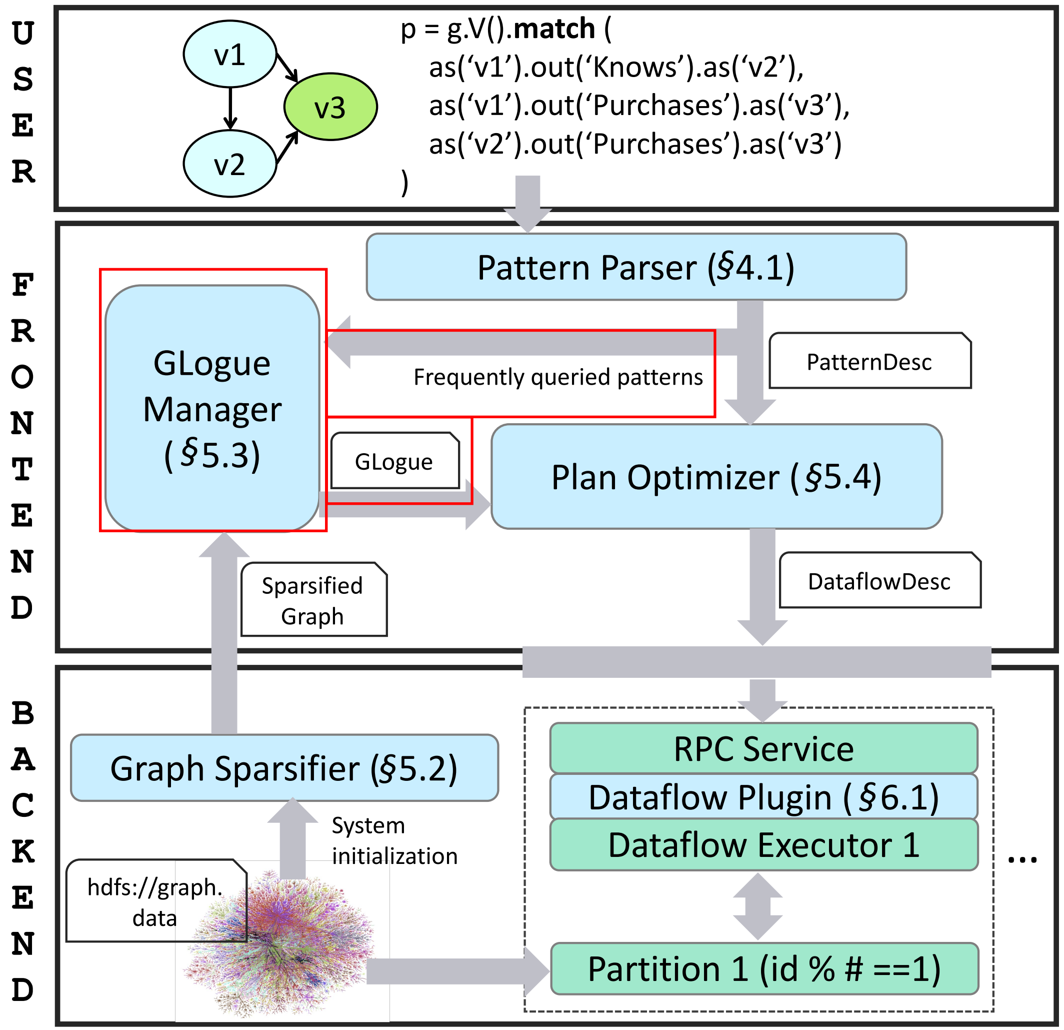


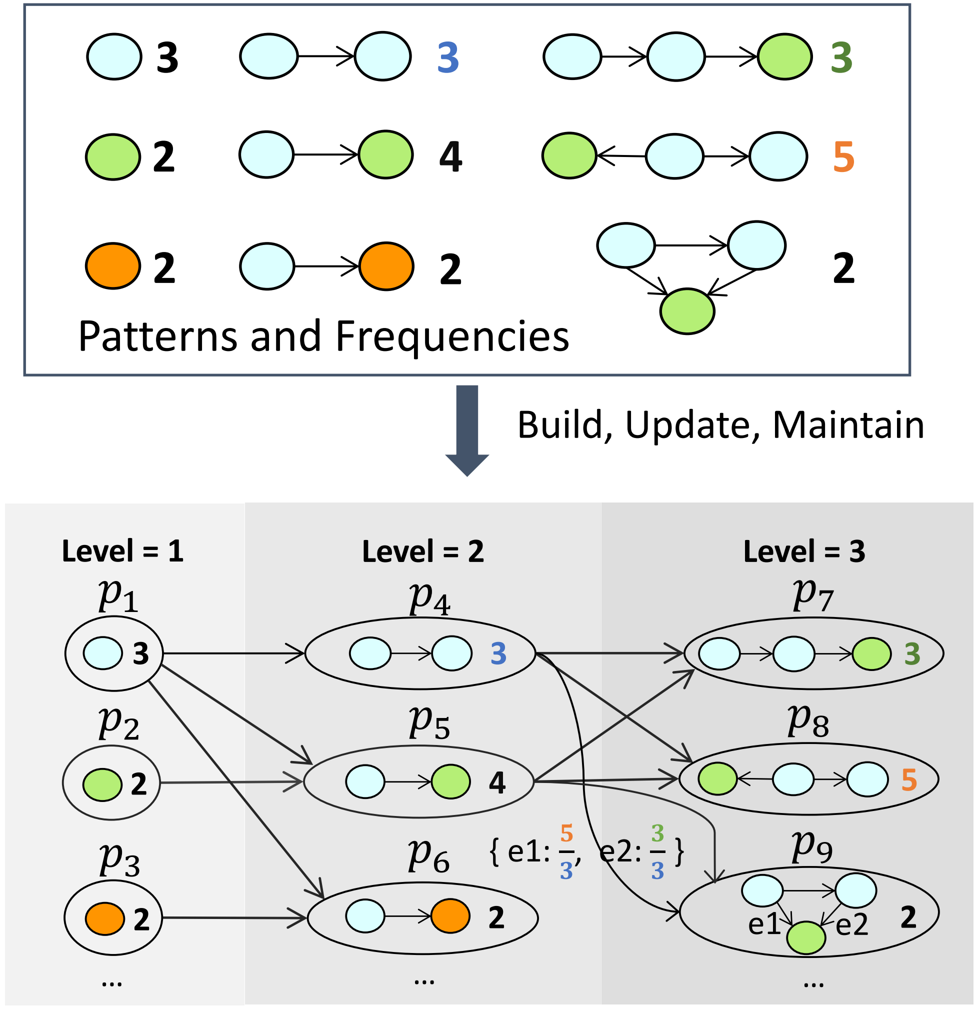
Gremlin	g.V().hasLabel('Person').**match**(as('v1').outE('Knows').as('e1').inV('Person').as('v2'), as('v1').outE('LivesIn').inV('City').as('v3'), as('v2').outE('LivesIn').inV('City').as('v3'))
Cypher	**MATCH** (v1: Person) -[e1:Knows]-> (v2: Person), (v1: Person) -[LivesIn]-> (v3: City), (v2: Person) -[LivesIn]-> (v3: City)

Parse

PatternDesc { sentences: [[
GetV('', 'v1', Person, None)],
GetE('v1', 'e1', Knows, Out)],
GetV('e1', 'v2', Person, Target)], ...] }

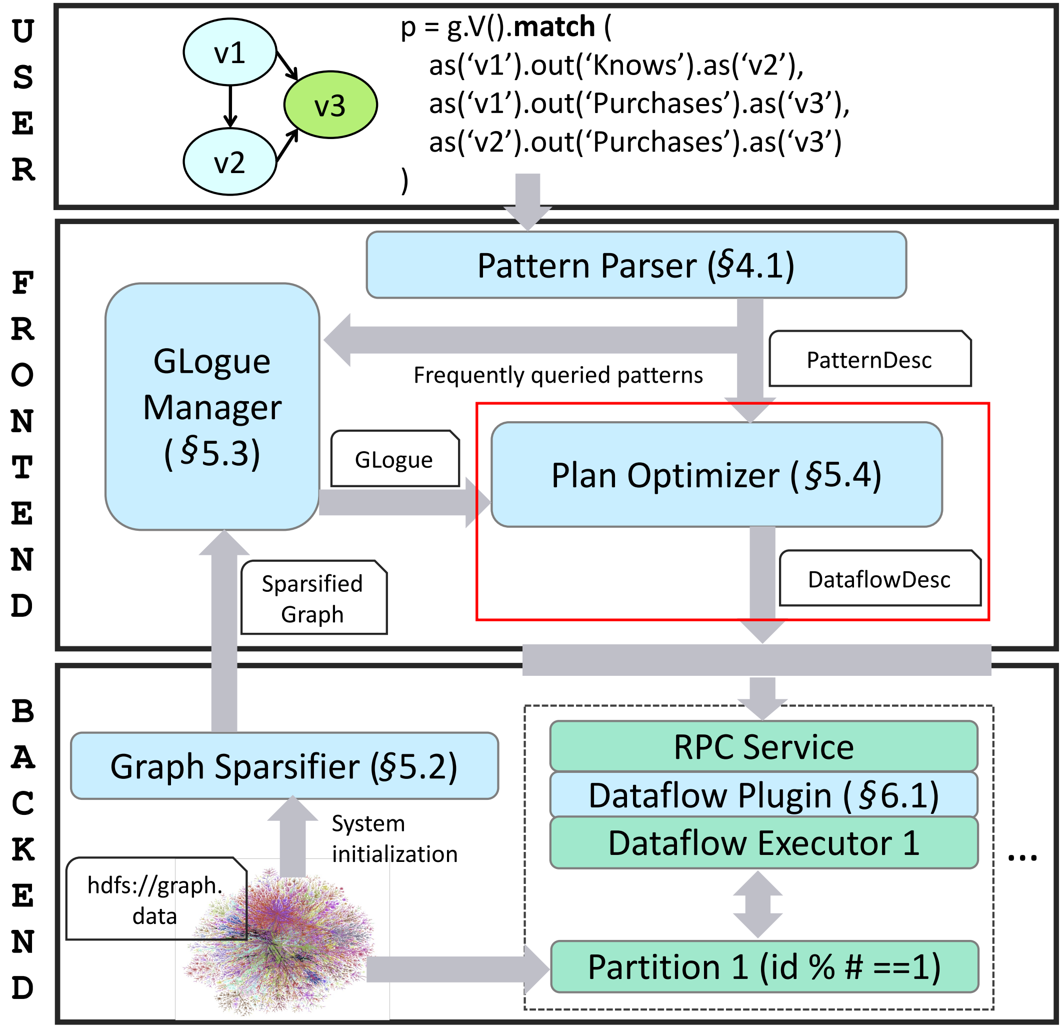
Frontend: GLogue Manager



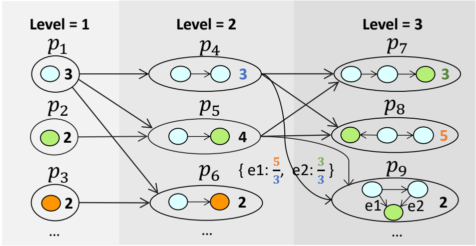


GLogue

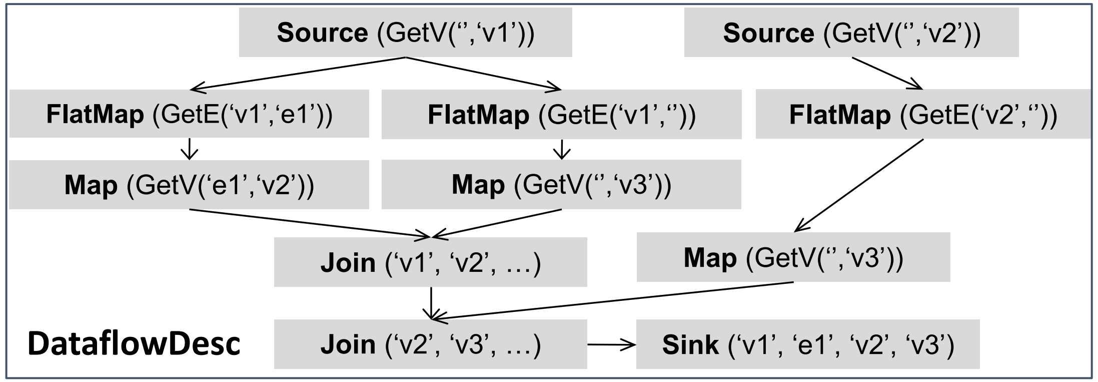
Frontend: Plan Optimizer



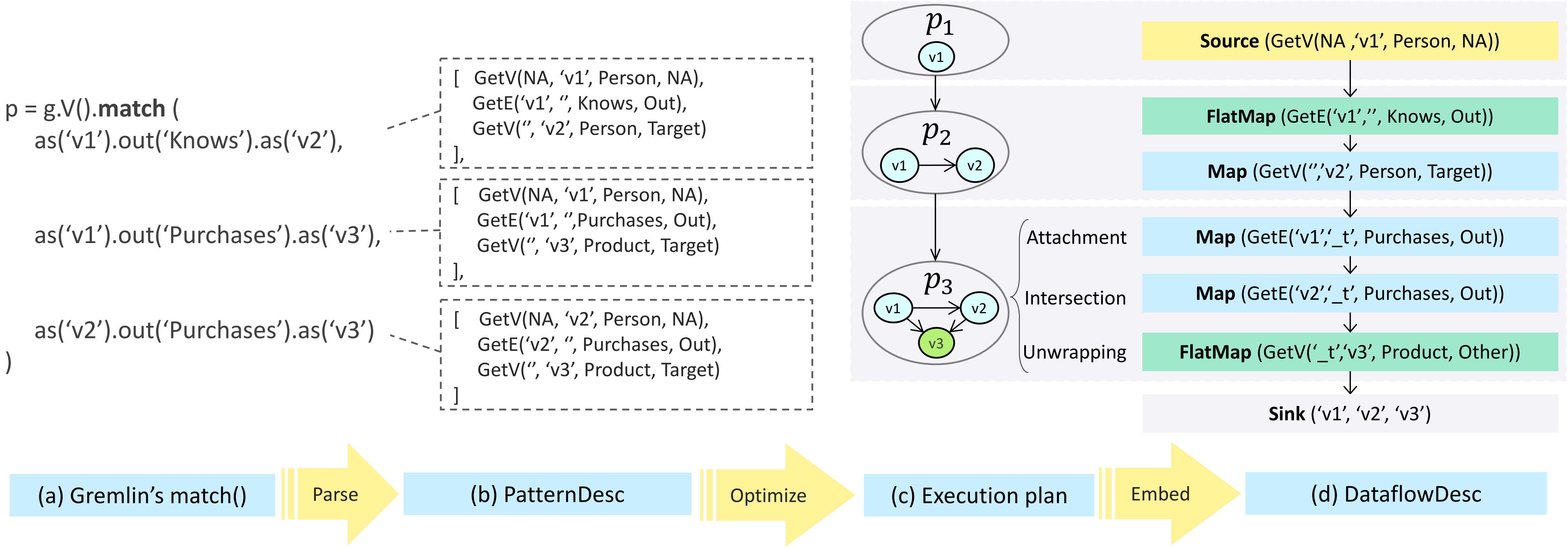
PatternDesc { sentences: [[
GetV('', 'v1', Person, None)],
GetE('v1', 'e1', Knows, Out)],
GetV('e1', 'v2', Person, Target)],
...] }



Plan Generation

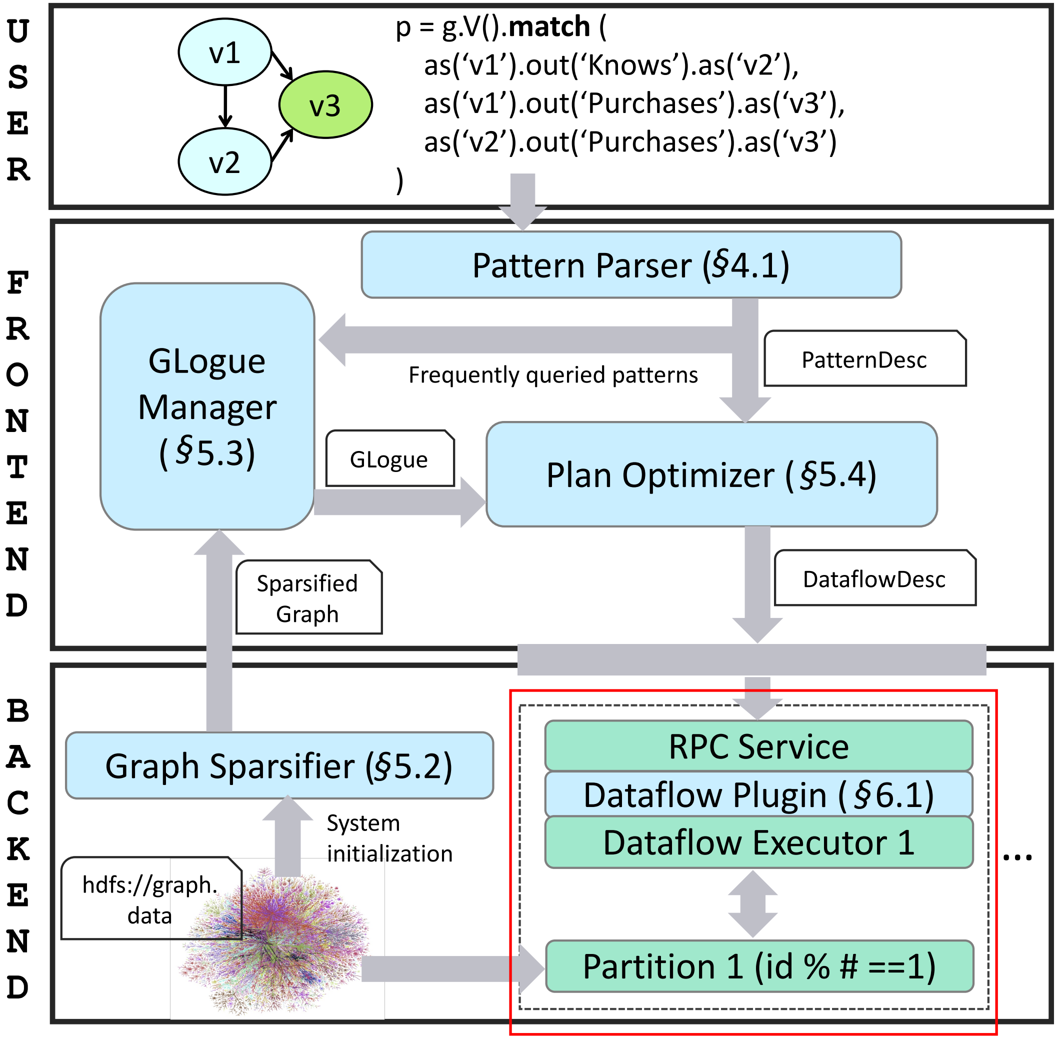


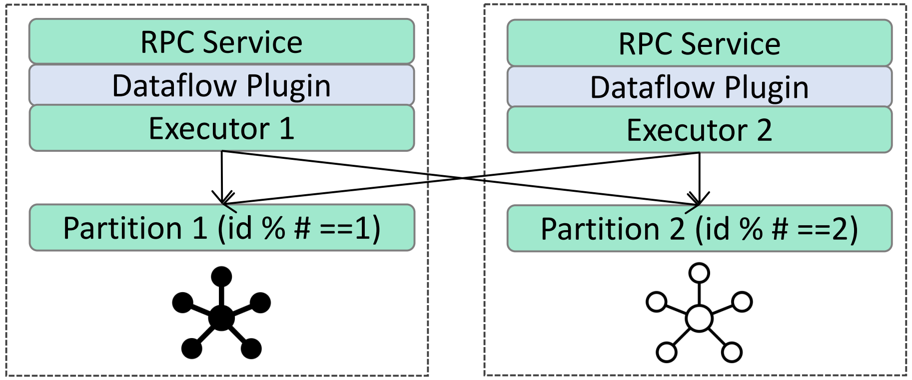
Frontend

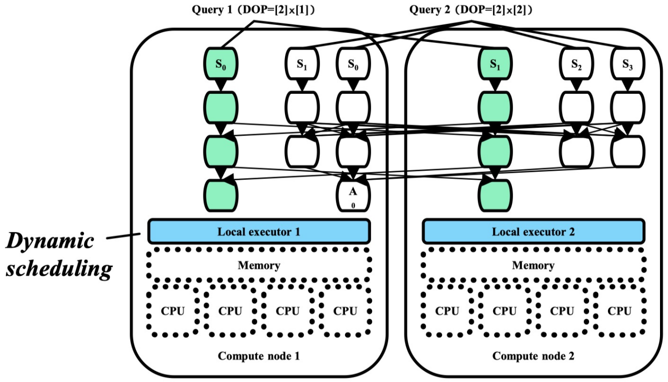


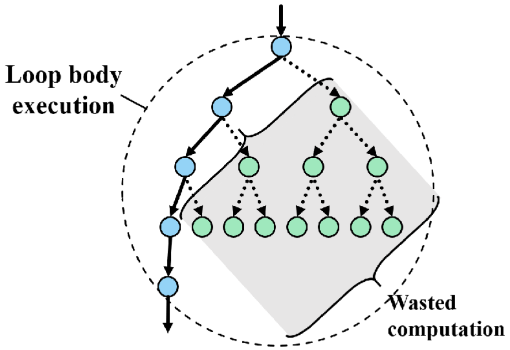
p = g.V().match (
as('v1').out('Knows').as('v2'),
as('v1').out('Purchases').as('v3'),
as('v2').out('Purchases').as('v3')
)
[GetV(NA, 'v1', Person, NA),
GetE('v1', '', Knows, Out),
GetV('', 'v2', Person, Target)
],
[GetV(NA, 'v1', Person, NA),
GetE('v1', '',Purchases, Out),
GetV('', 'v3', Product, Target)
],
[GetV(NA, 'v2', Person, NA),
GetE('v2', '', Purchases, Out),
GetV('', 'v3', Product, Target)
]
p_1
v1
p_2
v1
v2
p_3
v1
v2
v3
Attachment
Intersection
Unwrapping
Source (GetV(NA ,'v1', Person, NA))
FlatMap (GetE('v1','', Knows, Out))
Map (GetV('','v2', Person, Target))
Map (GetE('v1','_t', Purchases, Out))
Map (GetE('v2','_t', Purchases, Out))
FlatMap (GetV('_t','v3', Product, Other))
Sink ('v1', 'v2', 'v3')
(a) Gremlin's match()
Parse
(b) PatternDesc
Optimize
(c) Execution plan
Embed
(d) DataflowDesc

Backend: Distributed Dataflow Engine

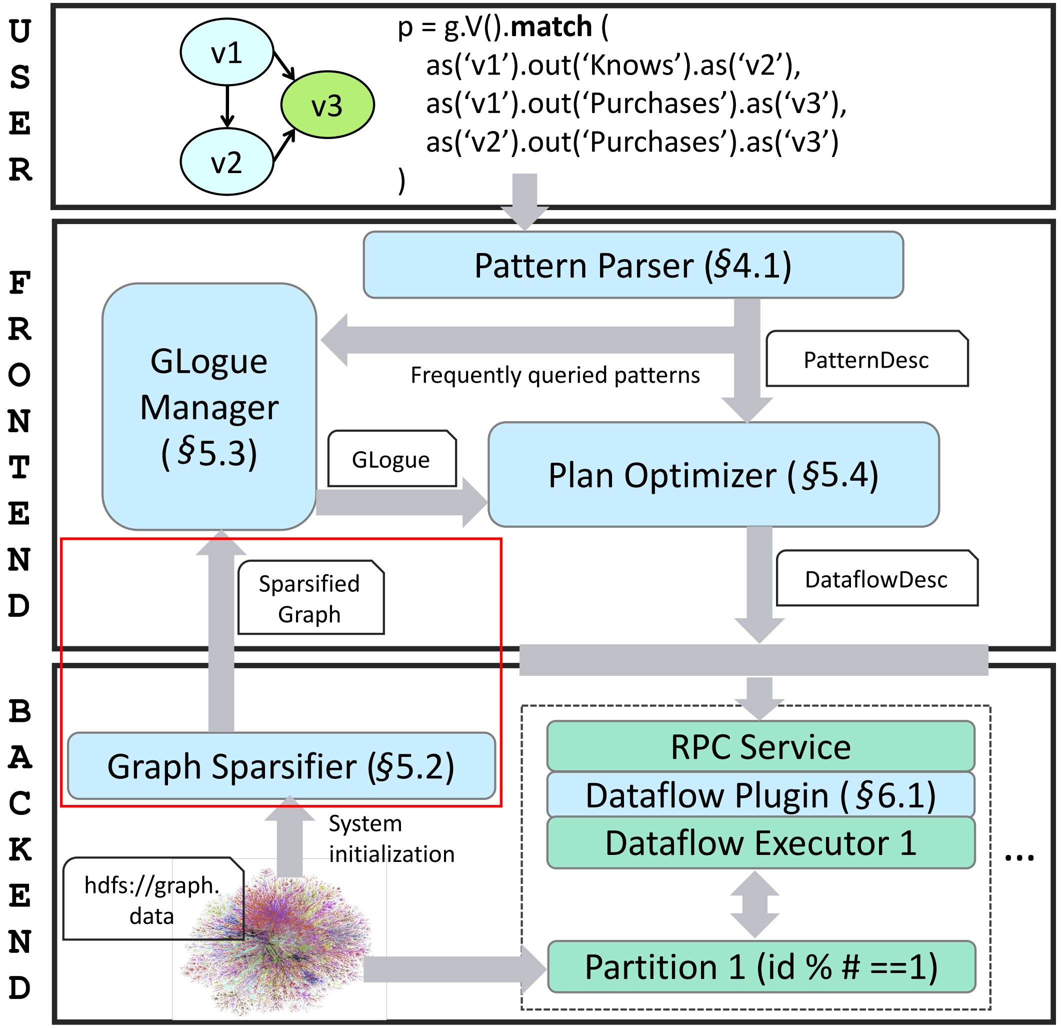


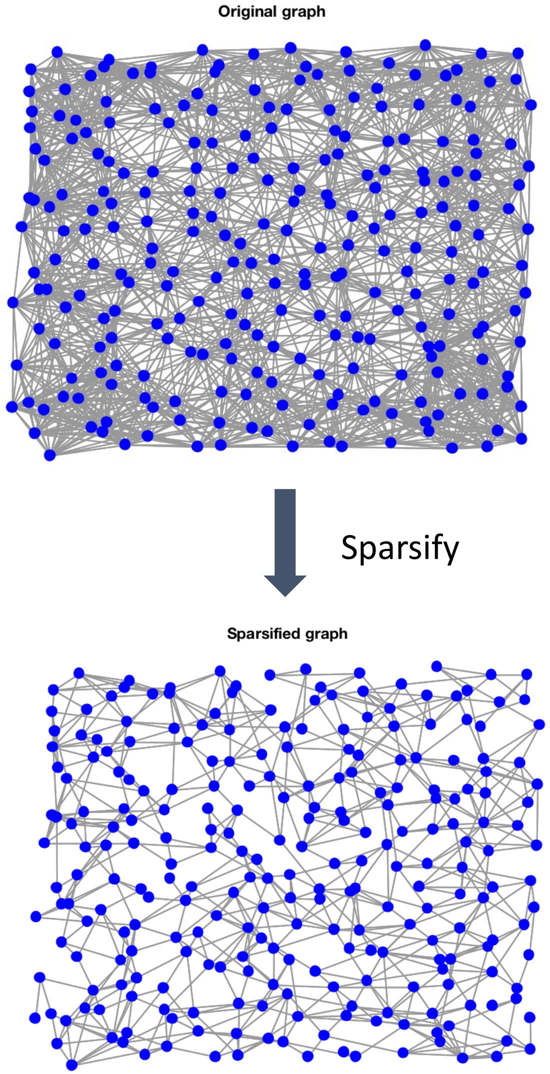






Backend: Graph Sparsifier

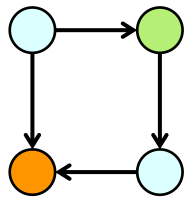




Outline

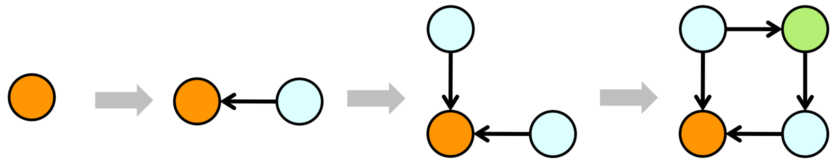
1. Background and Motivation
2. System Overview
3. **Feature Highlights**
4. Evaluation
5. Conclusion

GPM Execution Plan

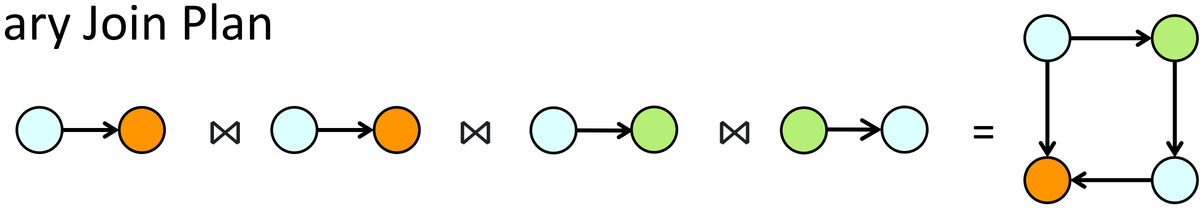


Target Pattern

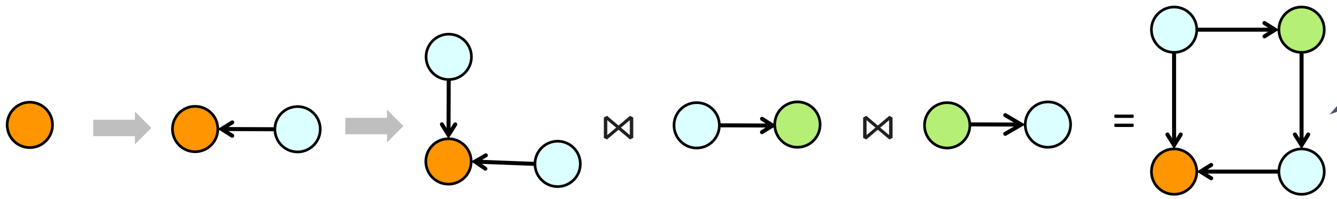
- Graph Pattern Matching Plan Space:
  - Worst-Case-Optimal Join Plan, based on Vertex Expansion



  - Binary Join Plan



  - Hybrid Plan



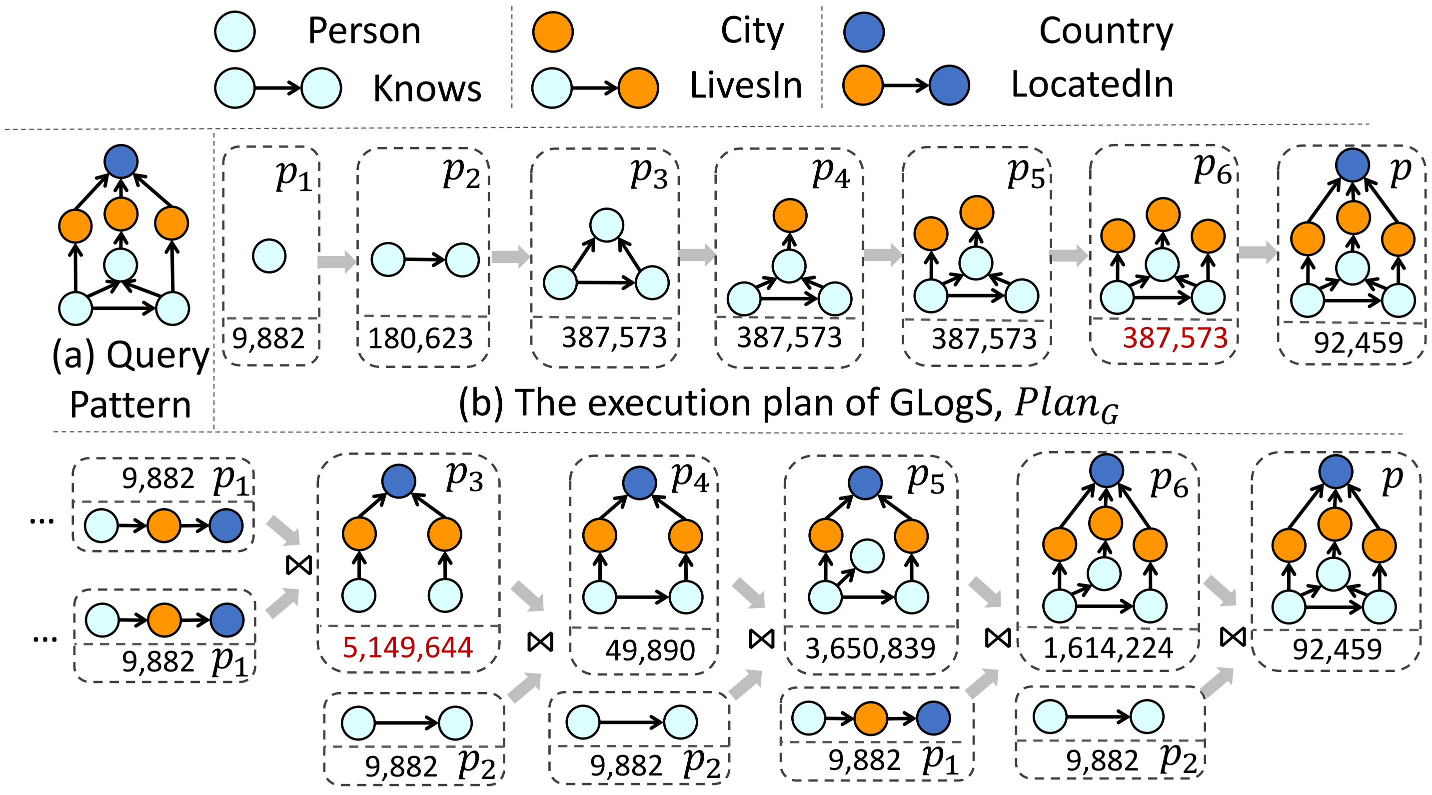
Best GPM execution plan lies in the hybrid plan space!

Vertex Expansion

⋈ Binary Join

GPM Execution Plan

- Key to GPM Execution Plan Optimization:
  - Minimizing Intermediate Results
- Comparison between the plans generated by GLogS and Neo4j for LDBC BI11 query separately



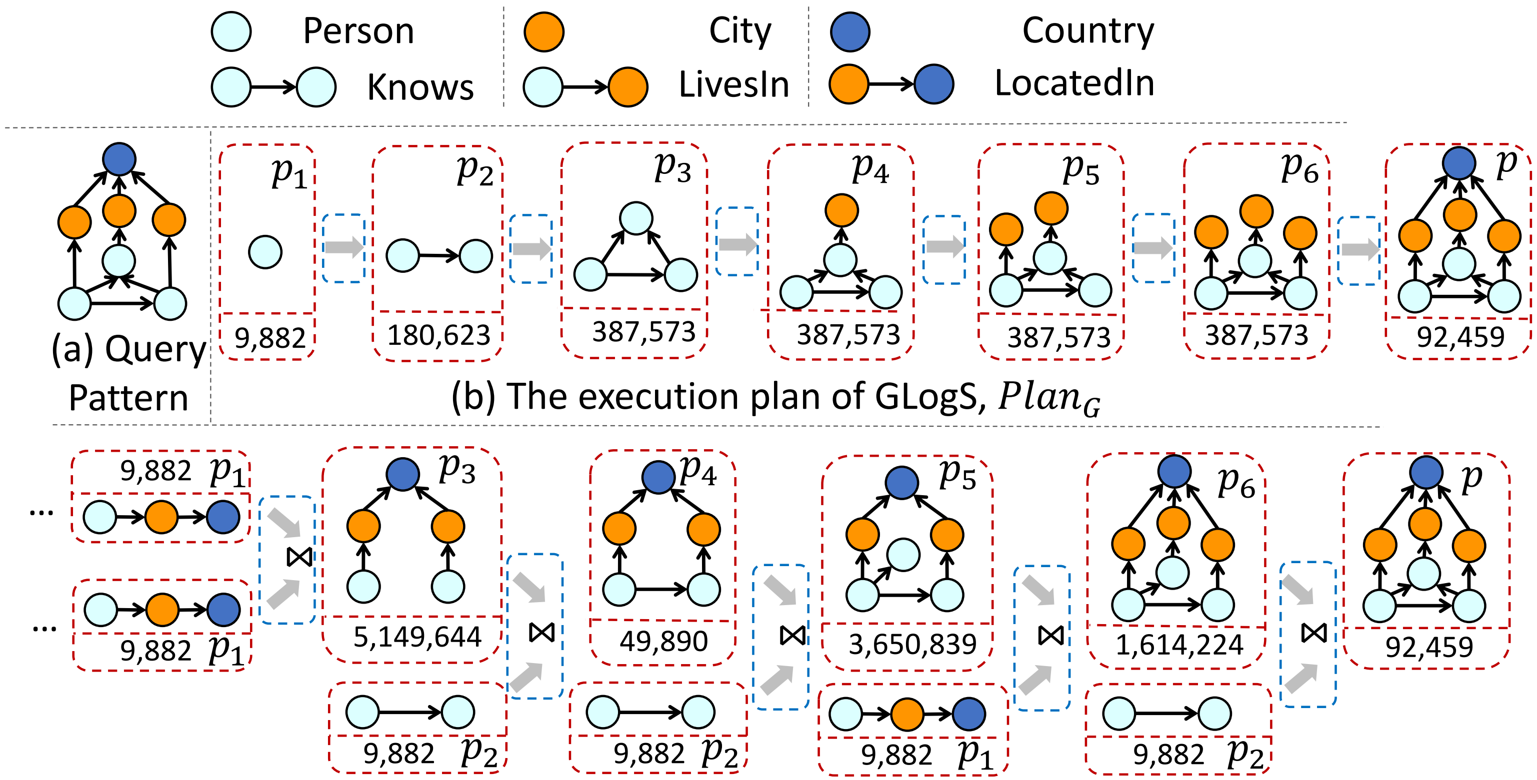
(b) The execution plan of GLogS, $Plan_G$

(c) The execution plan of Neo4j, $Plan_N$

Why is GLogS's plan better?
- Hybrid Execution plan
- High-order statistics

Cost Model for GPM Execution Plan

- Define a **Cost Model** to estimate the cost of every possible GPM execution plan
  - Cost of **accessing intermediate results** from the memory (either local or remote memory)
  - Cost of **operations** (vertex expansion, binary join)



(b) The execution plan of GLogS, $Plan_G$

(c) The execution plan of Neo4j, $Plan_N$

Cost Model for GPM Execution Plan

- $\text{Plan}(p) = (\Phi = \{p_1, p_2, \dots, p_n = p\}, \Gamma = [\tau_1, \tau_2, \dots, \tau_m])$
  - p_i: intermediate pattern
  - τ_i: operation on intermediate patterns, Join or Vertex Expansion
  - $\mathcal{F}(p_i)$: Frequency(count number) of the p_i in the graph.
  - $\text{Cost}(\text{Plan}(p)) = \sum_{p' \in \Phi} \mathcal{F}(p') + \sum_{\tau \in \Gamma} \text{Cost}(\tau)$
  - $\text{Cost}\left(\text{Join}(\{p_{s_1}, p_{s_2}\} \to p_t)\right) = \alpha_j \left(\mathcal{F}(p_{s_1}) + \mathcal{F}(p_{s_2})\right)$
  - $\text{Cost}(\text{Expand}(p_s \to p_t)) = \alpha_{ve} \sum_{f \in Q(p_s)} \sum_{i=1}^{k} \sigma_{e_i}(f) = \alpha_{ve} \mathcal{F}(p_s) \sum_{i=1}^{k} \overline{\sigma_{e_i}}$
- We find that $\boldsymbol{\mathcal{F}(p_i)}$ is the key to the entire cost model!

Cost Model for GPM Execution Plan

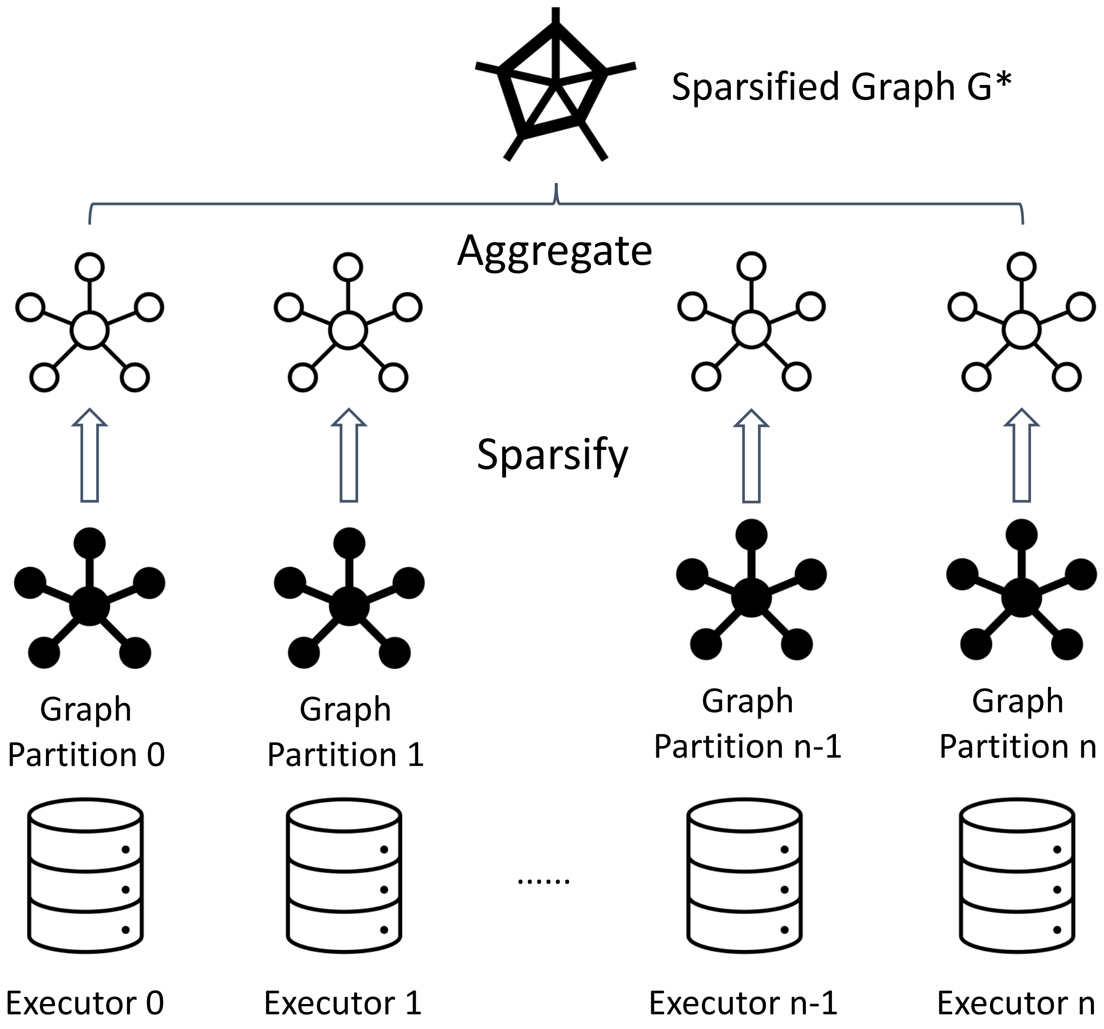
How to extract various of $\mathcal{F}(p_i)$ from graph efficiently? → Graph Sparsifier

What data structure should be used to organize all kinds of p_i, $\mathcal{F}(p_i)$ and τ_i? → GLogue

Based on p_i, $\mathcal{F}(p_i)$ and τ_i, how to generate optimized plan for a given query? → Plan Optimizer

Graph Sparsifier

- It's **cost-prohibitive** to compute $\mathcal{F}(p_i)$ **directly** from the original large graph datasets.
- Conduct sparsification on each partition of the graph, and then aggregate them to form the sparsified graph G$*$
- Use $\mathcal{F}_{G*}(p_i)$(with normalization) as an estimation of $\mathcal{F}_G(p_i)$ for cost evaluation



Graph Sparsifier: Stratified Sparsification

e	Knows	Purchases	LivesIn	LocatedIn
$\mathcal{F}(e)$	10000	50000	3000	100

Uniform Sparsification, rate 1%

e	Knows	Purchases	LivesIn	LocatedIn
$\mathcal{F}^*(e)$	≈100	≈500	≈30	$\boxed{≈1}$

Very likely to be 0!

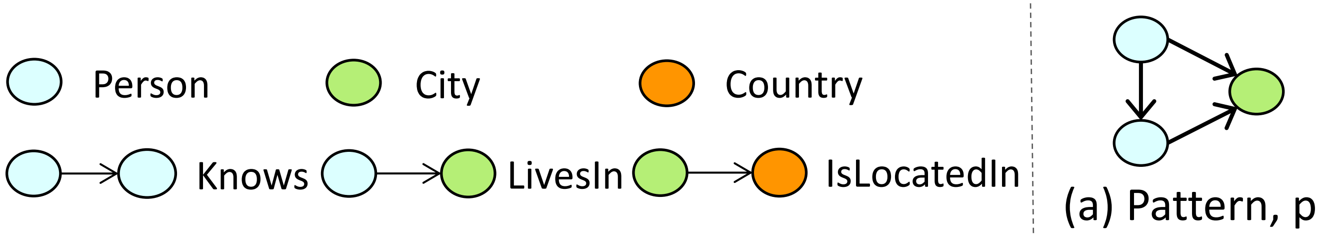
Pattern Vanishing!

Stratified Sparsification: rate $\min(1, \frac{M}{\sum \mathcal{F}(e)} * \frac{1}{\mathcal{F}(e)})$

e	Knows	Purchases	LivesIn	LocatedIn
rate	1%	0.2%	3%	100%
$\mathcal{F}^*(e)$	≈100	≈100	≈100	100

GLogue

- How to organize p_i, $\mathcal{F}(p_i)$ and τ_i?
  - Table-based catalog isn't good enough!



(a) Pattern, p

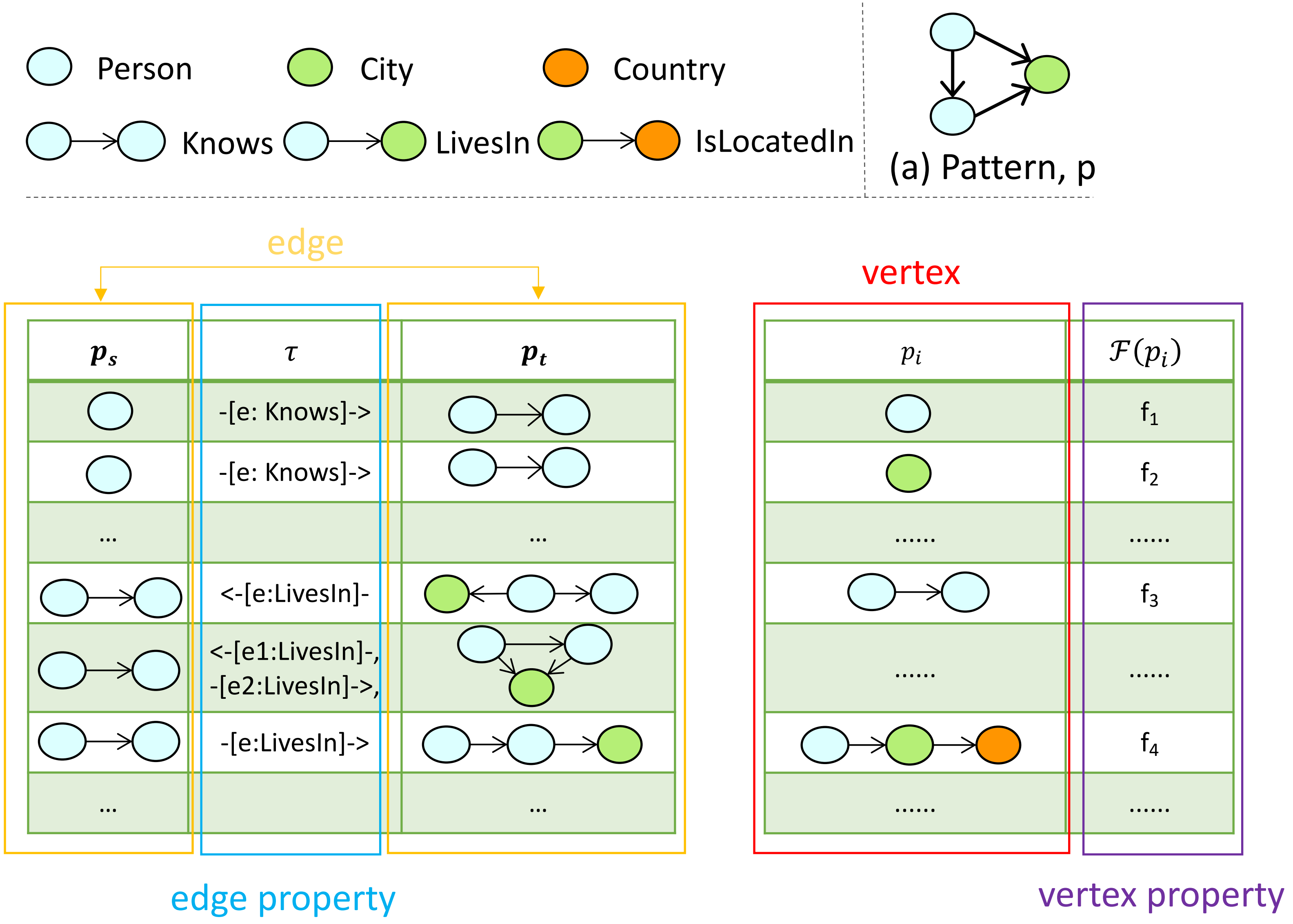
$\boldsymbol{p_s}$	τ	$\boldsymbol{p_t}$
	-[e: Knows]->	
	-[e: Knows]->	
…		…
	<-[e:LivesIn]-	
	<-[e1:LivesIn]-, -[e2:LivesIn]->,	
	-[e:LivesIn]->	
…		…

p_i	$\mathcal{F}(p_i)$
	f_1
	f_2
……	……
	f_3
……	……
	f_4
……	……

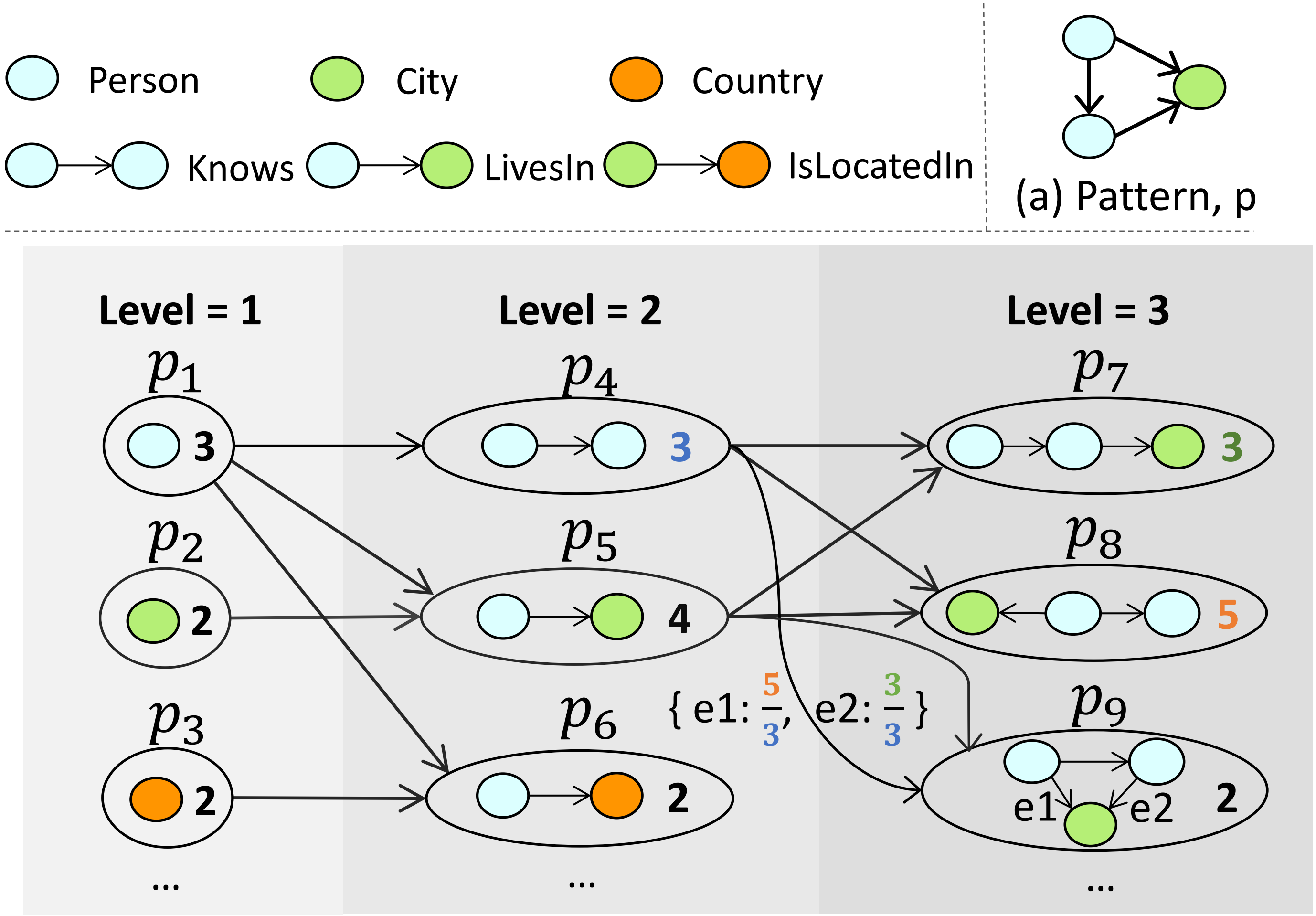
Many Redundancies!

Difficult for Plan Generation

GLogue


Person
City
Country
Knows
LivesIn
IsLocatedIn
(a) Pattern, p
edge
vertex
p_s
τ
p_t
-[e: Knows]->
-[e: Knows]->
<-[e:LivesIn]-
<-[e1:LivesIn]-,
-[e2:LivesIn]->,
-[e:LivesIn]->
p_i
$\mathcal{F}(p_i)$
f_1
f_2
f_3
f_4
edge property
vertex property

GLogue

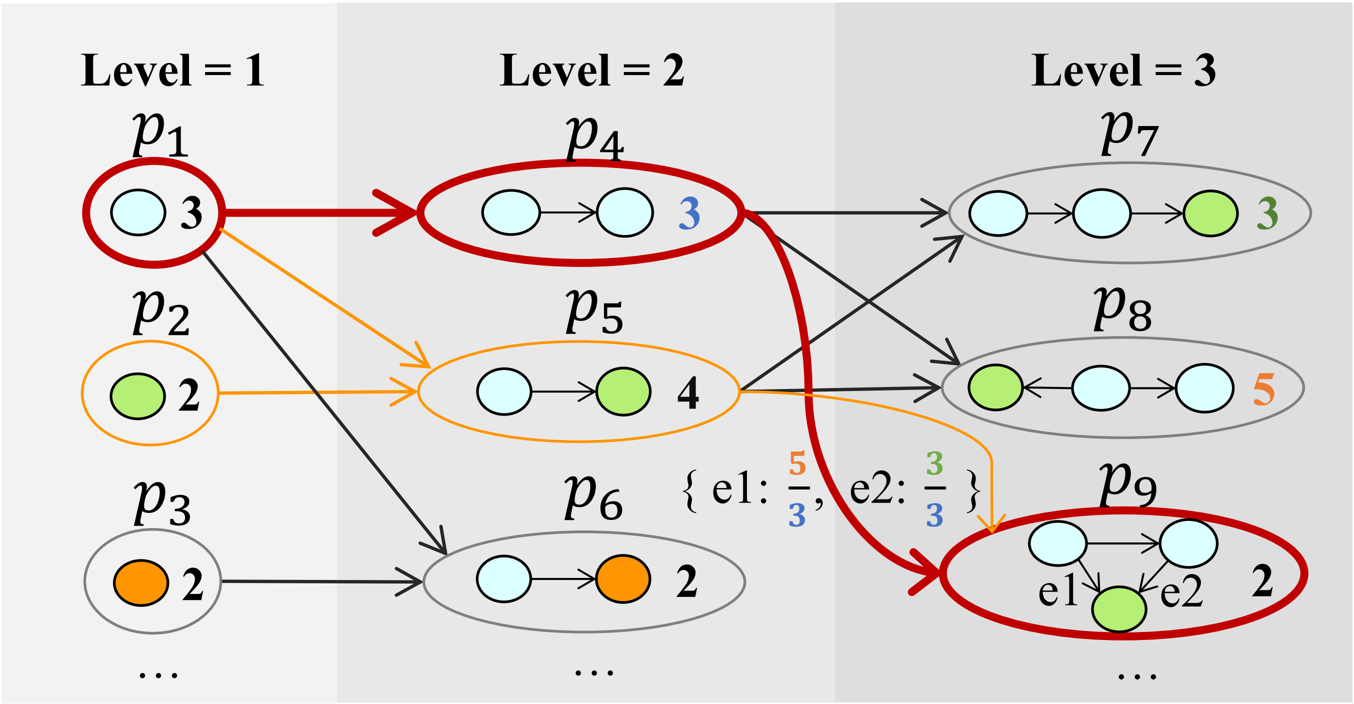


Algorithm 1: Constructing GLogue: (G^*, *Level*).

Function `constructGlogue` *(Level)*: GLogue
  Initialize GLogue with $level = 1, 2$ precomputed ;
  for $3 \le i \le Level$ **do**
    Let $QSet$ maintain all p of i vertices in ascending order via $|E_p|$;
    for $p \in QSet$ **do**
      `updateFromPattern` (p, GLogue);
  return GLogue;

Function `updateFromPattern` *(p, GLogue)*
  GLogue.`addVertex` (p, $\mathscr{F}(p)$) ;
  for $p_s \subset p$ & $|V_{p_s}| = |V_p| - 1$ **do**
    if *not* GLogue.`contains`(p_s) **then**
      `updateFromPattern` (p_s, GLogue);
    for $e = (s, t) \in E_p \setminus E_{p_s}$ **do**
      let p'_s be the pattern with e adding to p_s;
      eMap.`insert`(e, $\frac{\mathscr{F}(p'_s)}{\mathscr{F}(p_s)}$);
    GLogue.`addEdge` ((p_s, p), eMap);
  for $p_{s_1}, p_{s_2} \subset p$ & $E_{p_{s_1}} \cup E_{p_{s_2}} = E_p$ **do**
    if *not* GLogue.`contains`(p_{s_1}) **then**
      `updateFromPattern` (p_{s_1}, GLogue);
    if *not* GLogue.`contains`(p_{s_2}) **then**
      `updateFromPattern` (p_{s_2}, GLogue);
    GLogue.`addEdge` ((p_{s_1}, p), ($p_{s_2}, \mathscr{F}(p_{s_2})$));

Plan Optimizer



Best Execution Plan

Potential Execution Plan

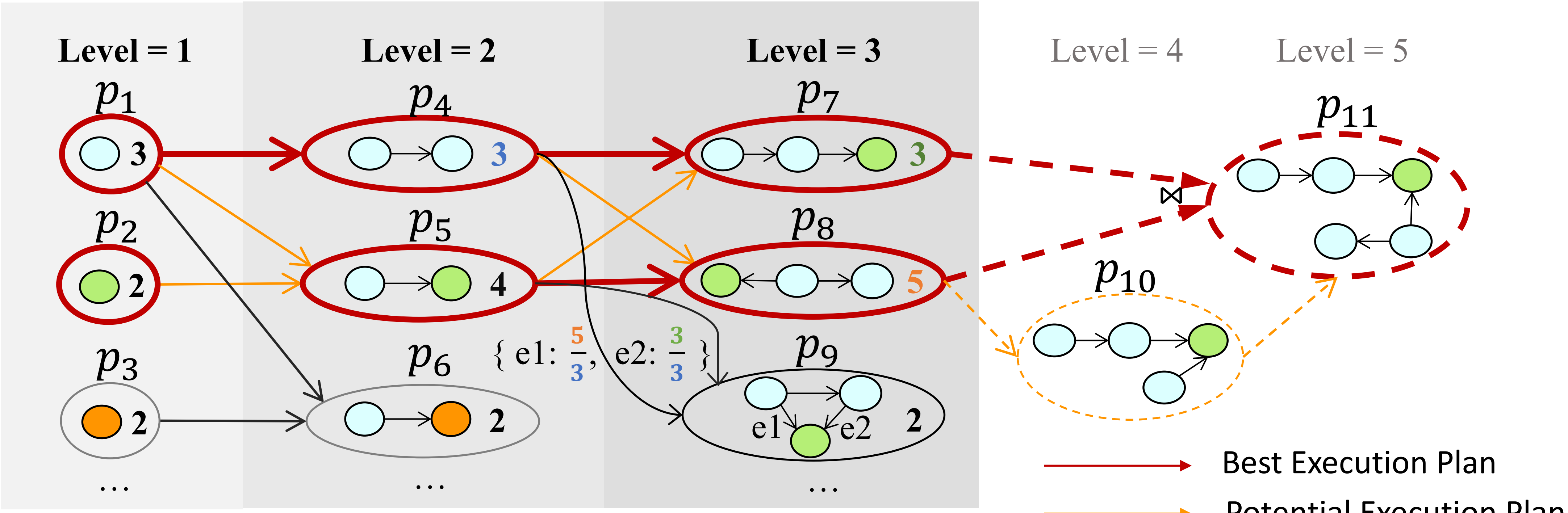
Algorithm 1: The Plan Optimizer.

Function `PlanOptimizer` (GLogue, PatternDesc)
  Construct a pattern p from the PatternDesc;
  Let $QSet$ organize all induced subgraphs of p by level;
  Initialize a `PlanMap` to record $\{p : (plan, cost)\}$ with patterns in level 1 and 2 pre-computed;
  for $3 \le level \le |V_p|$ **do**
    for $p \in QSet[level]$ **do**
      `searchPlan` $(p, \texttt{PlanMap}, \texttt{GLogue})$;
  return `PlanMap.get`(p);

Function `searchPlan` (p, `PlanMap`, `GLogue`)
  Initialize Plan(p) and Cost(Plan$(p)) \leftarrow \infty$;
  for edge $= (p_{s_1}, p) \in$ GLogue.getEdges(p) **do**
    $(plan1, cost1) \leftarrow$ `PlanMap.get`(p_{s_1});
    if edge *is a vertex extension* **then**
      Compute a new $plan'$ by merging $plan1$ and Expand$(p_{s_1} \to p)$;
    else if edge$\{(p_{s_2}, \mathscr{F}(p_{s_2}))\}$ *is binary join* **then**
      $(plan2, cost2) \leftarrow$ `PlanMap.get`(p_{s_2});
      Compute a new $plan'$ by merging $plan1$, $plan2$ and Join$(\{p_{s_1}, p_{s_2}\} \to p)$;
    Compute a new $cost'$ of $plan'$ by Equation 1;
    if $cost' <$ Cost(Plan(p)) **then**
      Update Plan(p) as $plan'$ and the cost as $cost'$;
  `PlanMap.insert`$(p, (\text{Plan}(p), \text{Cost}(\text{Plan}(p))))$;

Plan Optimizer

- If the pattern is not contained within the GLogue
  - Decompose the pattern step by step
  - Use this formula to estimate the frequencies of patterns not present in the GLogue.
  - $\mathcal{F}(p) = \mathrm{Avg}_{p_1,p_2} \frac{\mathcal{F}(p_1)\times\mathcal{F}(p_2)}{\mathcal{F}(p_1\cap p_2)}, p = p_1 \cup p_2$



Outline

Environment and Datasets

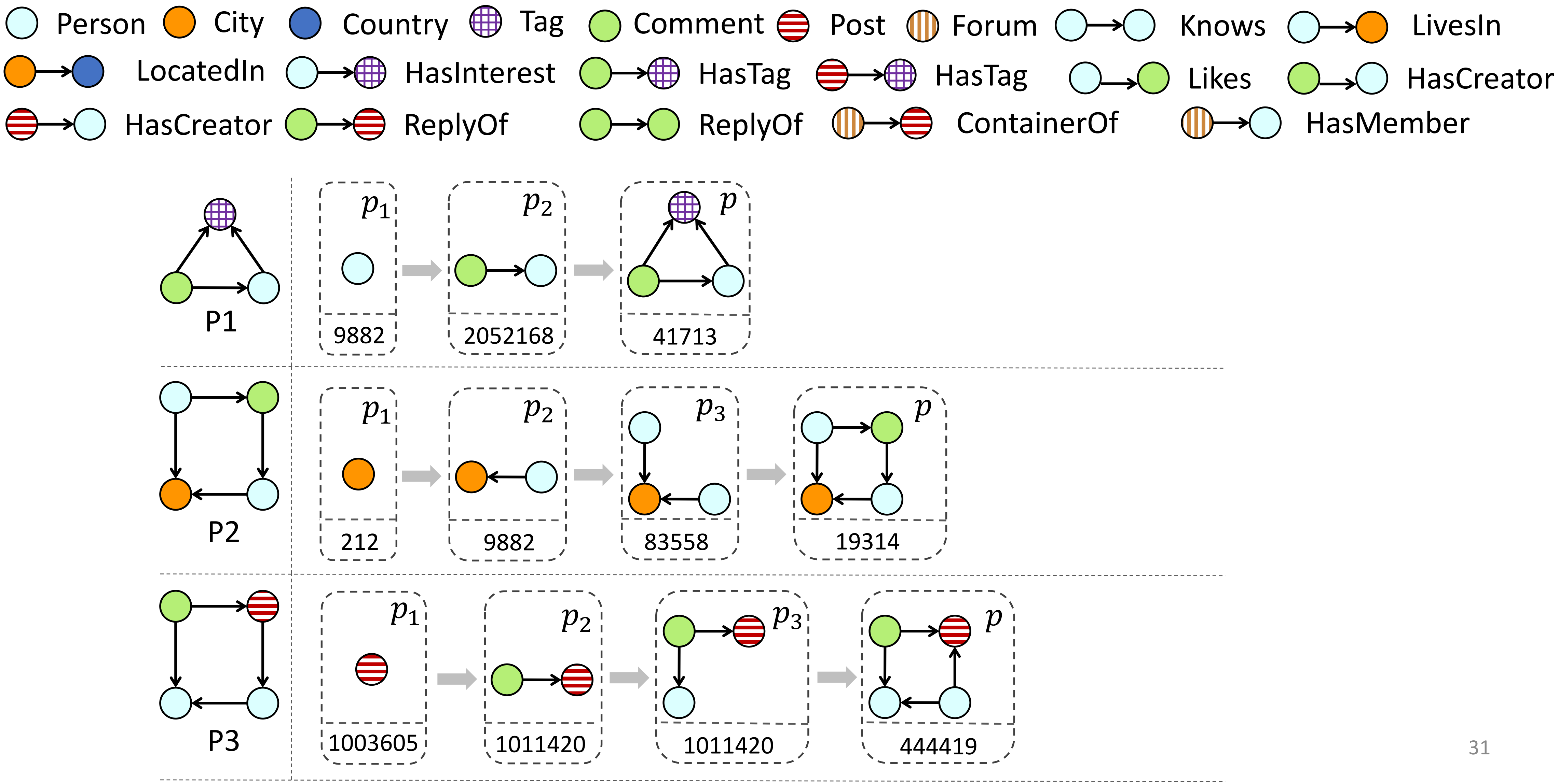
CPU	Memory	Disk	Network	Numbers
2*24-core Intel(R) Xeon(R) Platinum 8163 CPUs at 2.50GHz	512GB	2TB PCIE SSD	EDR 25Gbps InfiniBand network Full Bisection Bandwidth	1 frontend server Up to 16 backend servers

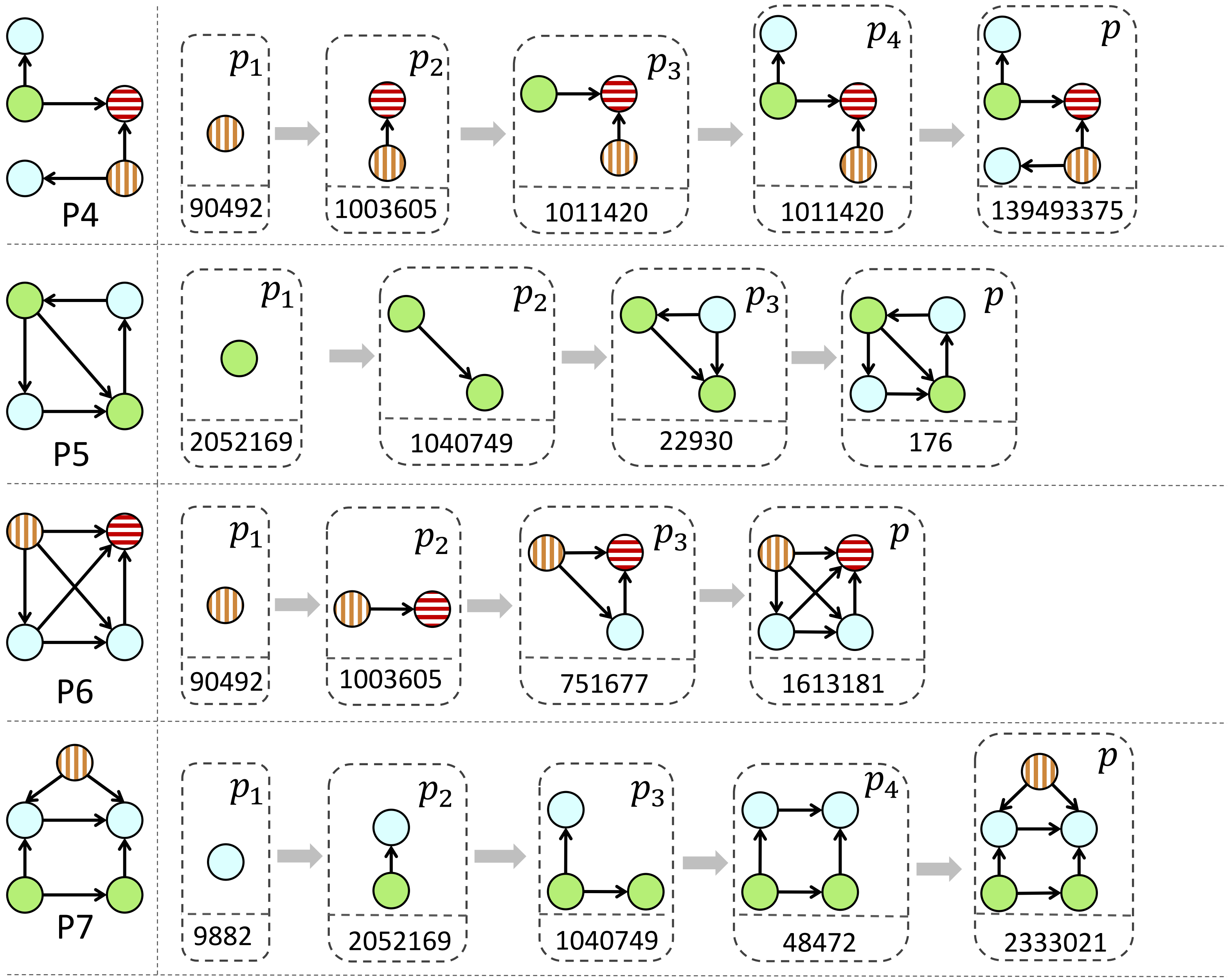
Table: Configurations for servers used in the experiments

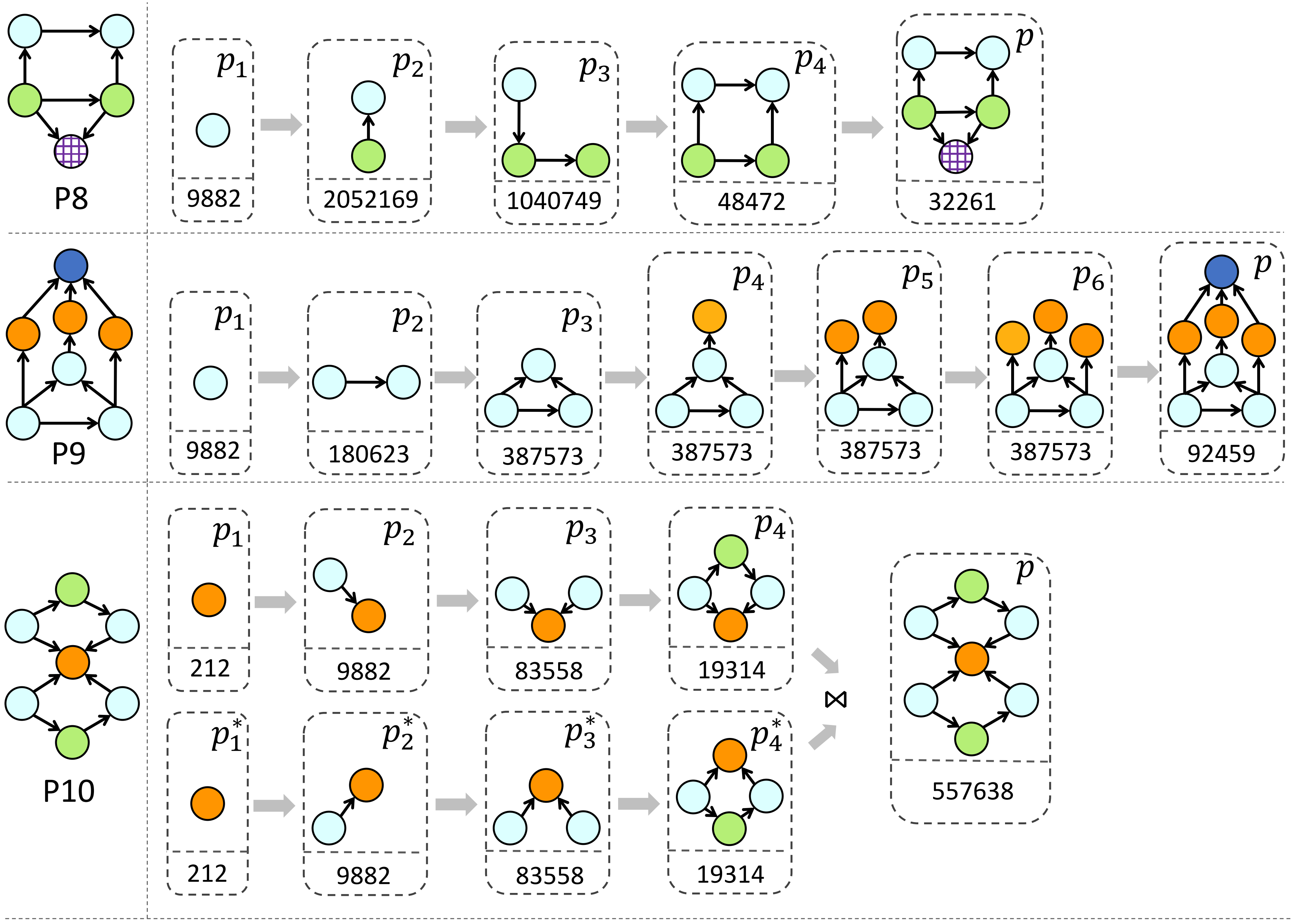
Graph	|V|	|E|	Size	γ
G_1	3M	17M	1.5GB	100%
G_{30}	89M	541M	40GB	1%
G_{100}	283M	1,754M	156GB	0.1%
G_{300}	817M	5,269M	597GB	0.1%
G_{1000}	2,687M	17,789M	1,960GB	0.03%

Table: LDBC graphs used in the experiments

Pattern and Plan




P4
p_1
90492
p_2
1003605
p_3
1011420
p_4
1011420
p
139493375
P5
p_1
2052169
p_2
1040749
p_3
22930
p
176
P6
p_1
90492
p_2
1003605
p_3
751677
p
1613181
P7
p_1
9882
p_2
2052169
p_3
1040749
p_4
48472
p
2333021


P8
p1
9882
p2
2052169
p3
1040749
p4
48472
p
32261
P9
p1
9882
p2
180623
p3
387573
p4
387573
p5
387573
p6
387573
p
92459
P10
p1
212
p2
9882
p3
83558
p4
19314
p1*
212
p2*
9882
p3*
83558
p4*
19314
⋈
p
557638

Experiment1: V.S. Neo4j and TigerGraph

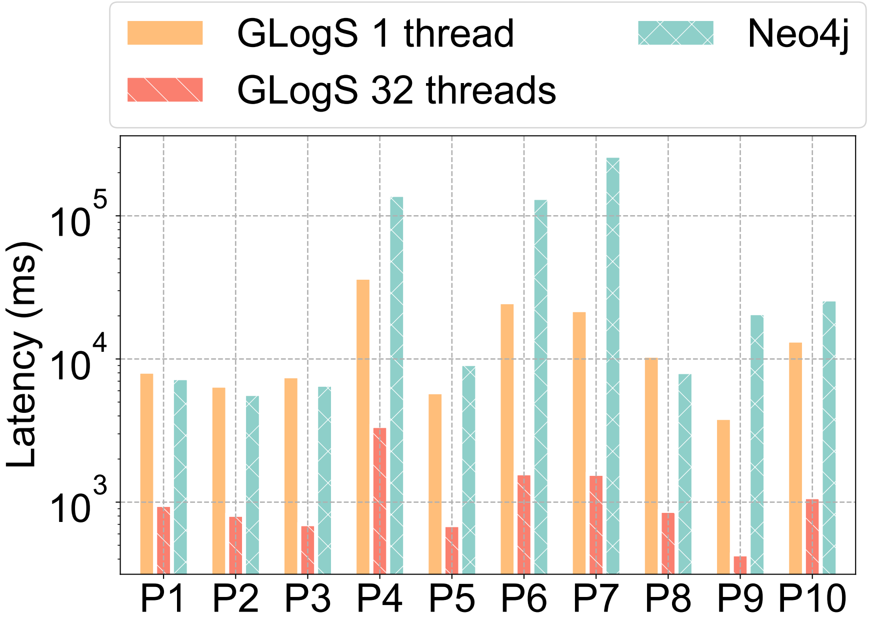


Figure: GLogS V.S. Neo4j

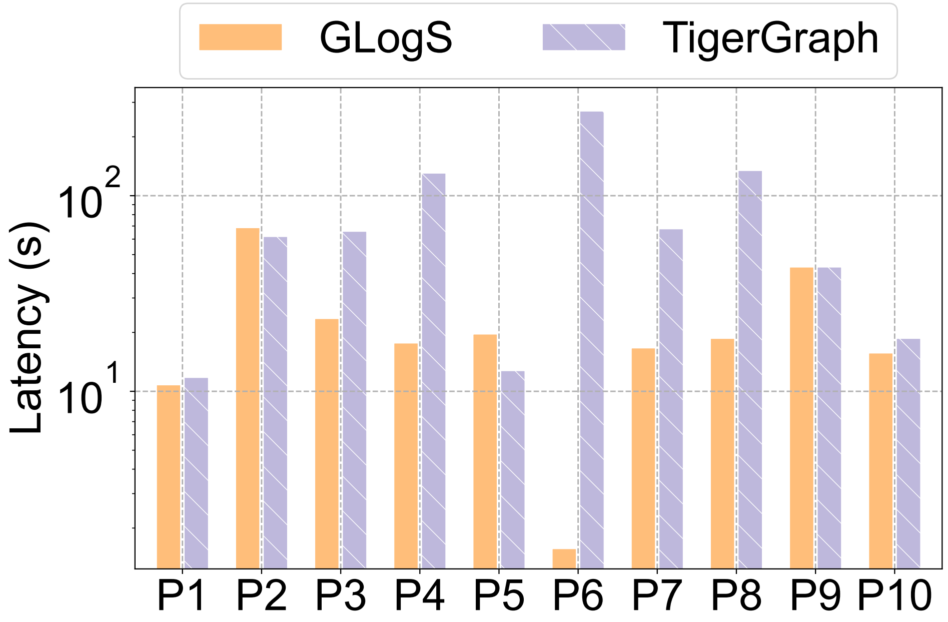


Figure: GLogS V.S. TigerGraph

GLogs achieves 51× and 57% speedup compared with Neo4j and TigerGraph, respectively.

Experiment2: High-order Statistics

	Level = 2	Level = 3	Level = 4
Slow-down(%)	966	245	243
Generation Time(s)	6	55	1664
Memory Usage(GB)	2	3	105
# Patterns	34	248	4164

Table: The effectiveness of high-order statistics

Experiment3: Sparsification

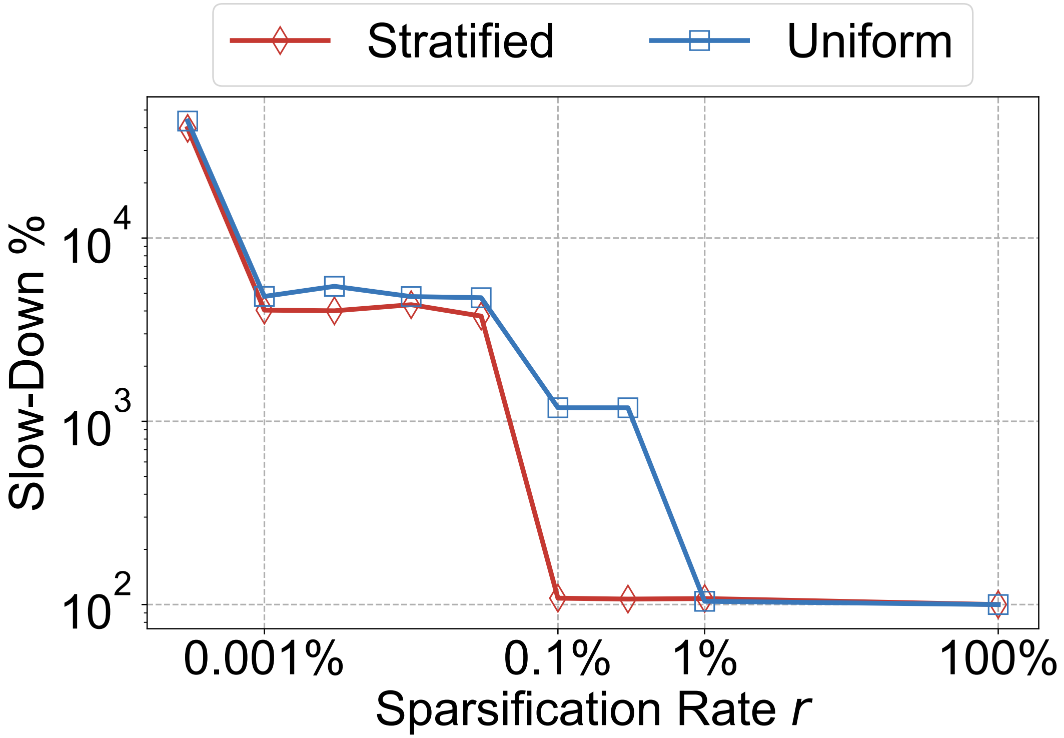


Figure: Stratified V.S. Uniform Sparsification

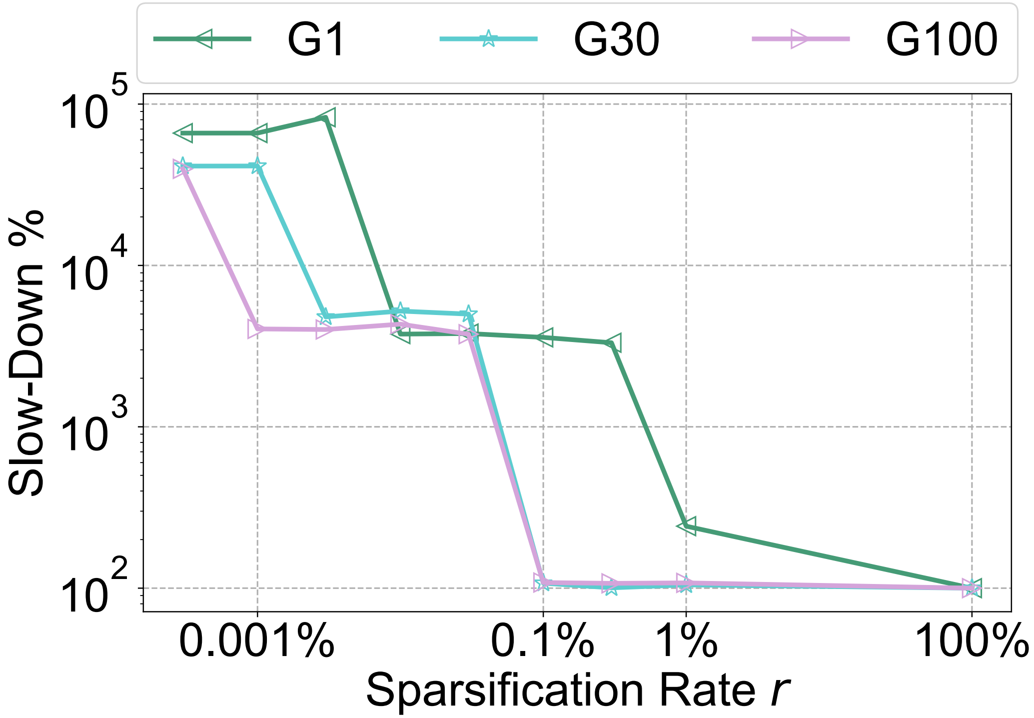


Figure: Sparsification of graphs with different scale

Uniform sparsification requires a higher rate to achieve same performance as stratified sparsification.
Small graph requires a higher rate to have a good plan.

Experiment4: Scalability

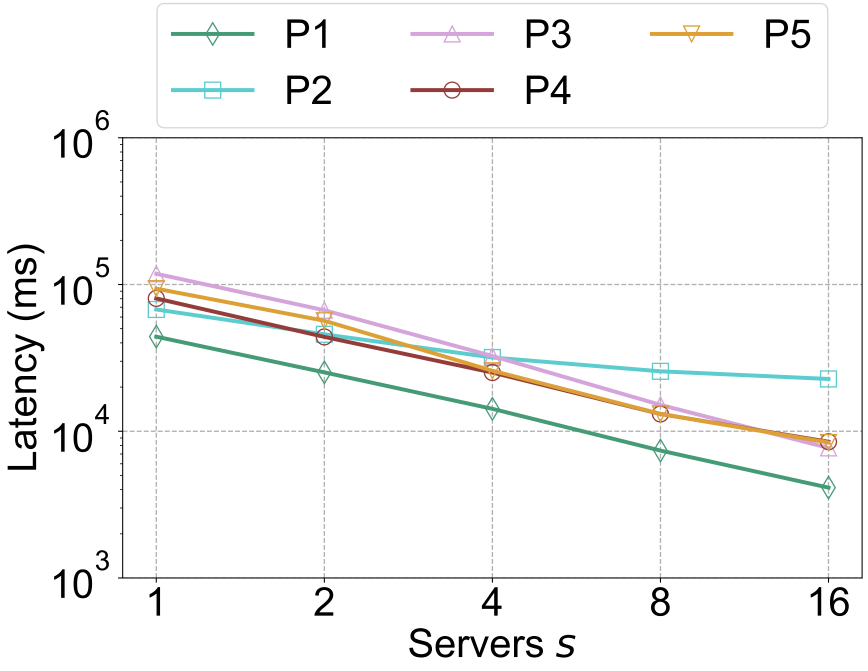


Figure: Scale-out experiments for group1 queries

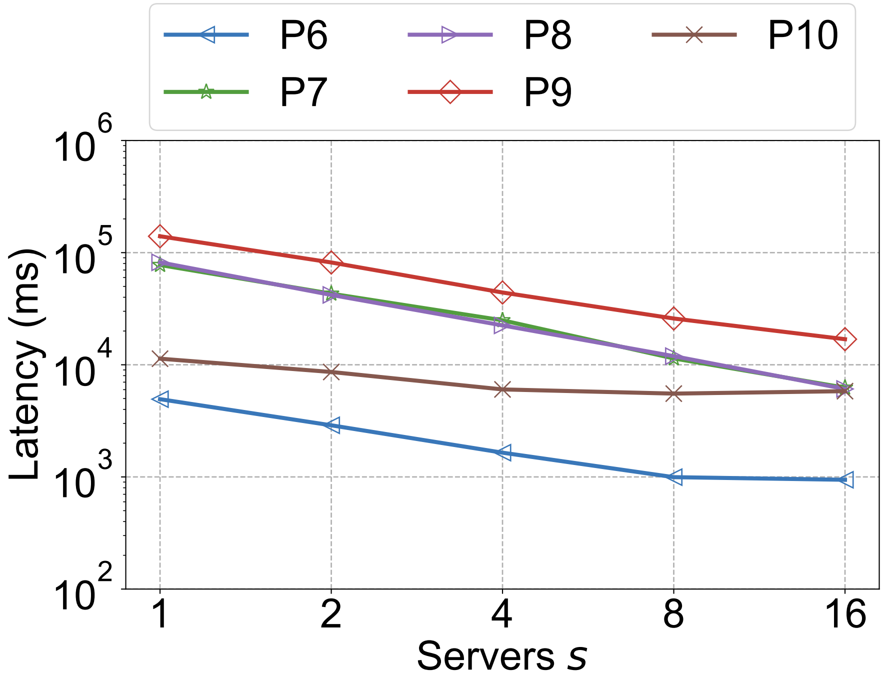


Figure: Scale-out experiments for group2 queries

Most queries scale well, with up to 15× (average 6×) performance gain from one machine to 16.

Experiment4: Scalability

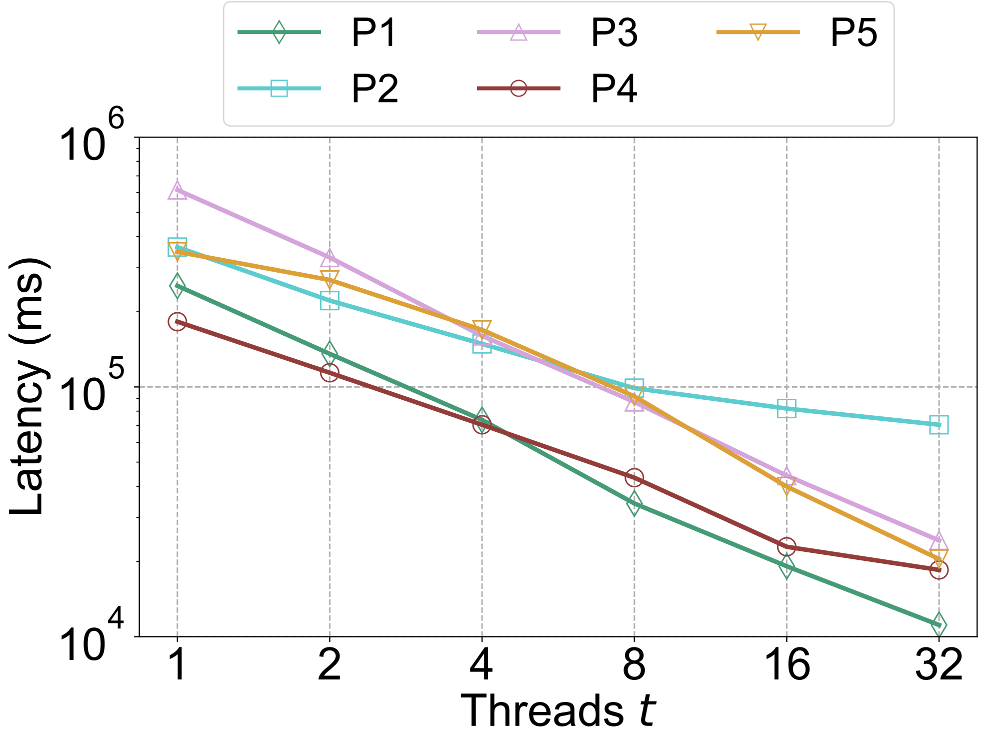


Figure: Scale-up experiments for group1 queries

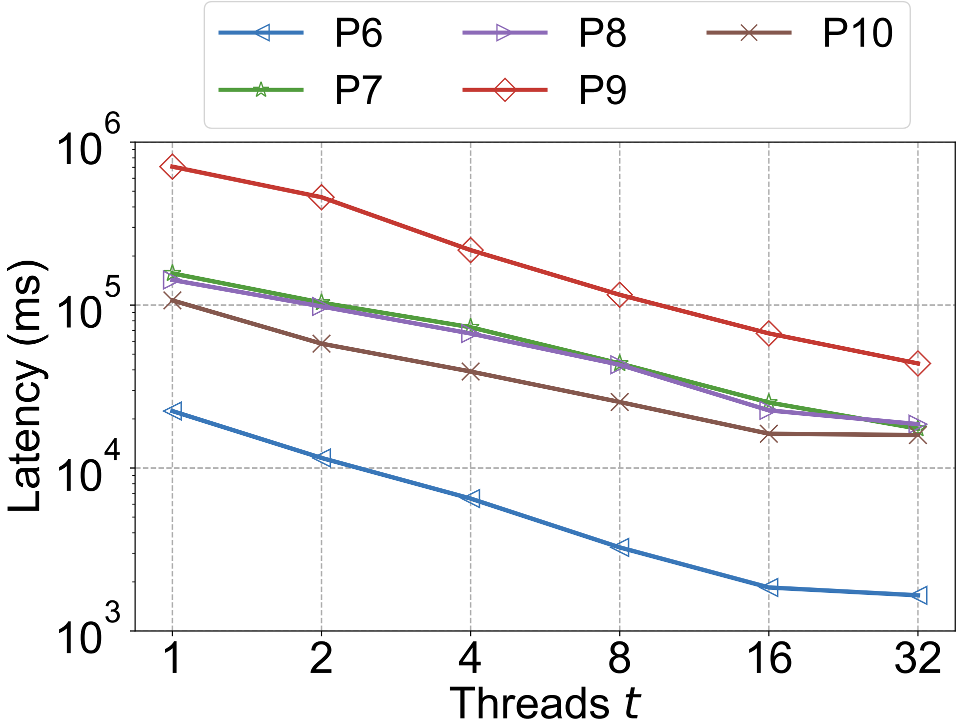


Figure: Scale-up experiments for group2 queries

Up to 23× (average of 10×) when increasing the number of threads from 1 to 32
The scalability of p_2 and p_{10} is insignificant due to a skewed workload.

Experiment4: Scalability

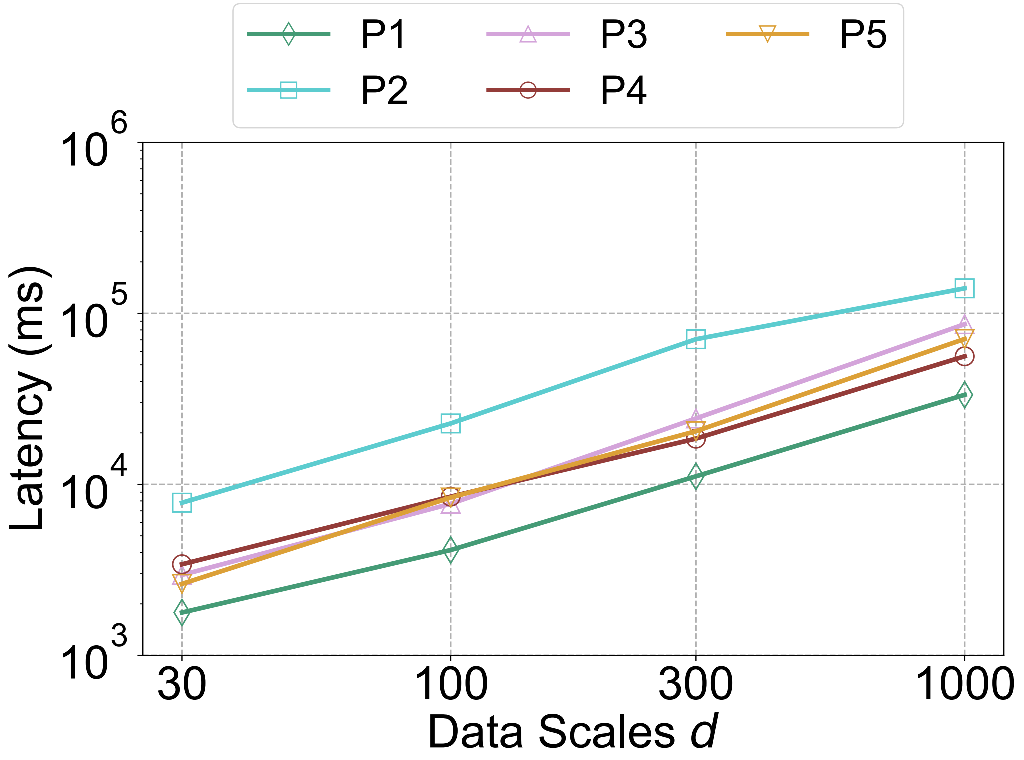


Figure: Data-scale experiments for group1 queries

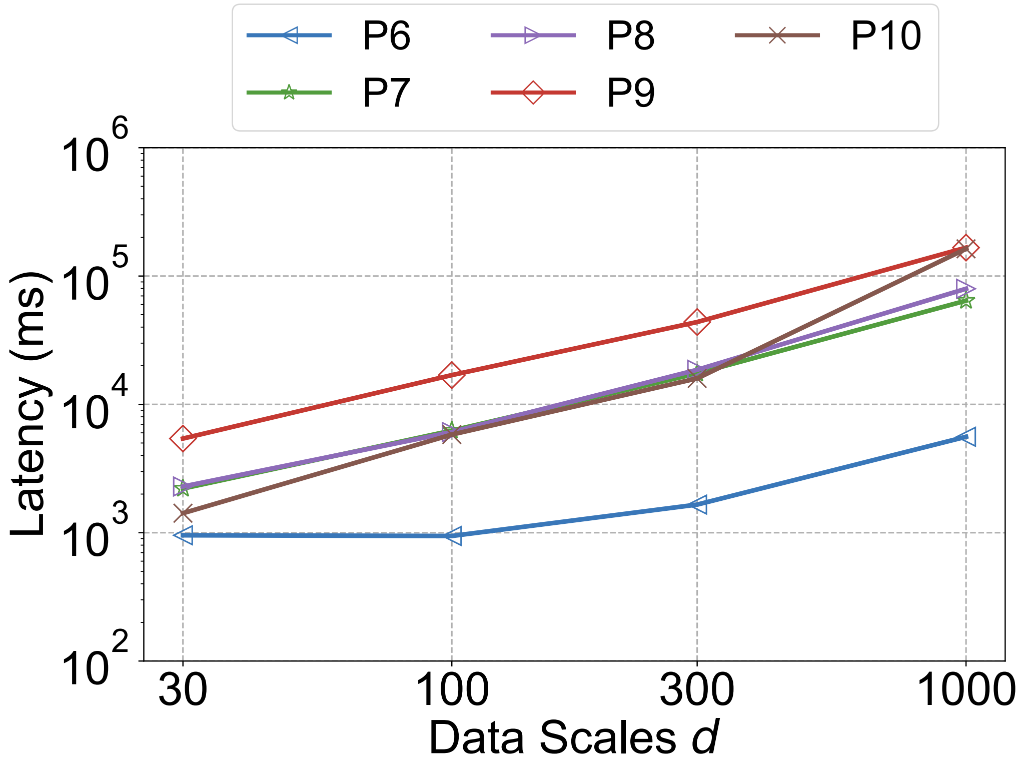


Figure: Data-scale experiments for group2 queries

As the graphs become larger, most queries demonstrate an almost linear trend in performance degradation.
For p_6, its performance degrades by 3× from G_{30} to G_{1000}, due to its short duration and limited graph exploration.
For p_{10}, its performance degrades by 100× from G_{30} to G_{1000}, due to the large cost of join operation.

Outline

1. Background and Motivation
2. System Overview
3. Feature Highlights
4. Evaluation
5. Conclusion

Conclusion

- GLogS system solves iGPM, meeting the requirements of:
  - usability
  - performance
  - scalability
- Allows user interactively submit GPM queries
- Supports automatic optimization for arbitrary GPM queries:
  - adopt high-order statistics
  - guarantee worst-case optimal
- Proposes a novel graph-based structure GLogue:
  - maintain the high-order statistics of the graph
- Capable of deployed in a large cluster to handle real-life large graphs

Related Work

- GPC: A Pattern Calculus for Property Graphs
  - https://dl.acm.org/doi/10.1145/3584372.3588662
- GQL Standard
  - https://www.gqlstandards.org/

GPM Queries are gradually being standardized and becoming increasingly important!

GLogS is continuously evolving with the graph community and standards!

Scan to visit GraphScope Github Repo:
https://github.com/alibaba/GraphScope

THANK YOU