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

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

Social Network Graph

Knowledge Graph

Molecular Graph

Graph Pattern Matching

(a) Pattern p

(b) Graph G

(c) The mappings of p in G

Graph is Prevalent!

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u2</td>
<td>u1</td>
<td>u4</td>
</tr>
<tr>
<td>u3</td>
<td>u5</td>
<td>u7</td>
</tr>
<tr>
<td>u6</td>
<td>u7</td>
<td>u8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u2</td>
<td>u1</td>
<td>u4</td>
</tr>
<tr>
<td>u2</td>
<td>u3</td>
<td>u5</td>
</tr>
</tbody>
</table>
Interactive Graph Pattern Matching

- **Pattern p**
  - **Non-Expert**
  - **Query Tuning**
  - **Not Good**
  - **Got It**

- **Interactive GPM**

- **Graph G**

- **The mappings of p in G**

- **Usability**
- **Performance**
- **Scalability**

- **Large Graph**

- **Which Pattern?**

- **Recommendation Rules**

- **Q: p (without predicates)**
  - **Not Good**
  - **Q: p.where(v1.age < 30 && v2.age < 30)**
  - **Not Good**
  - **Q: p.where(v1.age < 30 && v2.age < 30 && v3.name==“iPhone14Pro”)**
  - **Got It**

- **Table:**
  - | v1 | v2 | v3 |
  - |------------------|
  - | u2 | u1 | u4 |
  - | u2 | u3 | u5 |
iGPM: Requirements and Features

Usability
Feature 1: Declarative Language

Performance
Feature 2: Automatic Optimization

Scalability
Feature 3: Distributed Execution

Are there any existing solutions?

neo4j  TigerGraph
iGPM: Existing Solutions and Challenges

**Feature 1:** Declarative Language
- Support Cypher

**Feature 2:** 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

**Feature 3:** Distributed Execution
- Single Machine Design
  - Bad execution plan!

**Challenge 1:** Optimization
- Guarantee worst-case optimal
- Adopt high-order statistics

<table>
<thead>
<tr>
<th>$p_i$</th>
<th>$F(p_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>$f_2$</td>
</tr>
<tr>
<td></td>
<td>$f_3$</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

low-order
high-order
iGPM: Existing Solutions and Challenges

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
System Overview

1. Pattern Parser (§4.1)

2. Plan Optimizer (§5.4)

3. GLogue Manager (§5.3)

4. Graph Sparsifier (§5.2)

5. RPC Service

6. Dataflow Plugin (§6.1)

7. Dataflow Executor 1

8. Partition 1 (id % # ==1)
Frontend: Pattern Parser

**Pattern Parser (§4.1)**

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

**Plan Optimizer (§5.4)**

**GLogue Manager (§5.3)**

**Graph Sparsifier (§5.2)**

**HDFS data**

**Dataflow Plugin (§6.1)**

**Dataflow Executor 1**

**Partition 1 (id % # == 1)**

**GLogue**

**Sparsified Graph**

**PatternDesc**

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

**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**

\[ MATCH (v1: Person) -[e1:Knows]-> (v2: Person),
    (v1: Person) -[LivesIn]-> (v3: City),
    (v2: Person) -[LivesIn]-> (v3: City) \]
Frontend: GLogue Manager

GLogue Manager

Pattern Parser (§4.1)

Frequently queried patterns

PatternDesc

Plan Optimizer (§5.4)

Sparsified Graph

DataflowDesc

Graph Sparsifier (§5.2)

hdfs://graph. data

System initialization

RPC Service

Dataflow Plugin (§6.1)

Dataflow Executor

Partition 1 (id % # ==1)

GLogue

patterns and frequencies

Build, Update, Maintain

Level = 1

Level = 2

Level = 3

GLogue
Frontend: Plan Optimizer

```
\[ p = g.V().match { 
  as('v1').out('Knows').as('v2'),
  as('v1').out('Purchases').as('v3'),
  as('v2').out('Purchases').as('v3') 
} \]
```

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

**Plan Generation**

```
Source (GetV('', 'v1')) → FlatMap (GetE('v1', 'e1')) → Map (GetV('e1', 'v2')) → Join ('v1', 'v2', ...)
Source (GetV('', 'v2')) → FlatMap (GetE('v1', '')) → FlatMap (GetE('v2', '')) → FlatMap (GetE('v1', 'e1')) → Map (GetV('e1', 'v2')) → Map (GetV('', 'v3')) → Join ('v2', 'v3', ...) → Sink ('v1', 'e1', 'v2', 'v3')
```

** Burgess, Albert; Shvachko, K.; Ch吴, H.; Falsafi, B.; Manimaran, G.; Papandreou, I.; Xin, N.; Ge, R.**


---

**Frontend:**
- **Plan Optimizer**
- **Pattern Parser**
- **Graph Sparsifier**
- **RPC Service**
- **Dataflow Plugin**
- **Dataflow Executor**

**Backend:**
- **GLogue Manager**
- **System Initialization**

**User Interface:**
- **Frequently queried patterns**

**Sparsified Graph**

**DataflowDesc**

**Partition 1 (id % # == 1)**

---

**Plan Generation Diagram**

```
Level 1: P1 → P2 → P3
Level 2: P4 → P5 → P6
Level 3: P7 → P8
```

**System Initialization**

```
hdfs://graph.data
```
p = g.V().match (as('v1').out('Knows').as('v2'),
as('v1').out('Purchases').as('v3'),
as('v2').out('Purchases').as('v3'))

\[
\begin{align*}
\text{Source} & \quad (\text{GetV}(\text{NA}, 'v1', \text{Person}, \text{NA})) \\
\text{FlatMap} & \quad (\text{GetE}('v1','_t', \text{Purchases}, \text{Out})) \\
\text{Map} & \quad (\text{GetV}('v2', 'Person, \text{Target})) \\
\text{Attachment} & \quad (\text{GetE}('v1','_t', \text{Purchases}, \text{Out})) \\
\text{Intersection} & \quad (\text{GetE}('v2','_t', \text{Purchases, Out})) \\
\text{Unwrapping} & \quad (\text{GetV}('_t', 'v3', \text{Product, Other})) \\
\text{FlatMap} & \quad (\text{GetV}('_t', 'v3', \text{Product, Other})) \\
\text{Sink} & \quad (\text{'v1', 'v2', 'v3'})
\end{align*}
\]
Backend: Distributed Dataflow Engine

Pattern Parser (§4.1)

GLogue Manager (§5.3)

Plan Optimizer (§5.4)

Graph Sparsifier (§5.2)

 RPC Service
 Dataflow Plugin (§6.1)
 Dataflow Executor 1

Partition 1 (id % # ==1)

Frequently queried patterns

PatternDesc

Sparsified Graph

DataflowDesc

hdfs://graph.data

System initialization

Dynamic scheduling

Loop body execution

Wasted computation

PatternDesc DataflowDesc

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

v1

v2

v3

v1

v2

v3

v1

v2

v3

RPC Service
Dataflow Plugin
Executor 1

Partition 1 (id % # ==1)

RPC Service
Dataflow Plugin
Executor 2

Partition 2 (id % # ==2)
## Backend: Graph Sparsifier

**Pattern Parser (§4.1)**

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

- `v1`
- `v2`
- `v3`

**Plan Optimizer (§5.4)**

**GLogue Manager (§5.3)**

**Graph Sparsifier (§5.2)**

**RPC Service**

**Dataflow Plugin (§6.1)**

**Dataflow Executor 1**

**Partition 1 (id % # == 1)**

**Sparsified Graph**

**hdfs://graph.data**

**System initialization**

**Sparsify**
GPM Execution Plan

- 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!
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

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)
Cost Model for GPM Execution Plan

• Plan($p$) = (Φ = {$p_1, p_2, ..., p_n = p$}, Γ = [$\tau_1, \tau_2, ..., \tau_m$])
  • $p_i$: intermediate pattern
  • $\tau_i$: operation on intermediate patterns, Join or Vertex Expansion
  • $F(p_i)$: Frequency(count number) of the $p_i$ in the graph.
  • Cost(Plan($p$)) = $\sum_{p' \in \Phi} F(p') + \sum_{\tau \in \Gamma} \text{Cost}(\tau)$
  • Cost\left(\text{Join($\{p_{s_1}, p_{s_2}\} \rightarrow p_t$)}\right) = $\alpha_j \left( F(p_{s_1}) + F(p_{s_2}) \right)$
  • Cost\left(\text{Expand($p_s \rightarrow p_t$)}\right) = $\alpha_{ve} \sum_{f \in \mathcal{Q}(p_s)} \sum_{i=1}^{k} \sigma_{e_i}(f) = \alpha_{ve} F(p_s) \sum_{i=1}^{k} \overline{\sigma_{e_i}}$

• We find that $F(p_i)$ 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?

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

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

Graph Sparsifier

GLogue

Plan Optimizer
Graph Sparsifier

- It’s **cost-prohibitive** to compute $F(p_i)$ **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 $F_{G^*}(p_i)$ (with normalization) as an estimation of $F_G(p_i)$ for cost evaluation
Graph Sparsifier: Stratified Sparsification

<table>
<thead>
<tr>
<th>$e$</th>
<th>Knobs</th>
<th>Purchases</th>
<th>LivesIn</th>
<th>LocatedIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{F}(e)$</td>
<td>10000</td>
<td>50000</td>
<td>3000</td>
<td>100</td>
</tr>
</tbody>
</table>

Uniform Sparsification, rate 1%

<table>
<thead>
<tr>
<th>$e$</th>
<th>Knobs</th>
<th>Purchases</th>
<th>LivesIn</th>
<th>LocatedIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{F}^*(e)$</td>
<td>$\approx100$</td>
<td>$\approx500$</td>
<td>$\approx30$</td>
<td>$\approx1$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$e$</th>
<th>Knobs</th>
<th>Purchases</th>
<th>LivesIn</th>
<th>LocatedIn</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate</td>
<td>1%</td>
<td>0.2%</td>
<td>3%</td>
<td>100%</td>
</tr>
<tr>
<td>$\mathcal{F}^*(e)$</td>
<td>$\approx100$</td>
<td>$\approx100$</td>
<td>$\approx100$</td>
<td>100</td>
</tr>
</tbody>
</table>

Very likely to be 0!

Pattern Vanishing!
How to organize $p_i$, $\mathcal{F}(p_i)$ and $\tau_i$?

Table-based catalog isn’t good enough!

<table>
<thead>
<tr>
<th>$p_s$</th>
<th>$\tau$</th>
<th>$p_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌</td>
<td>-[e: Knows]-&gt;</td>
<td>🔴</td>
</tr>
<tr>
<td>❌</td>
<td>-[e: Knows]-&gt;</td>
<td>🔴</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>🔴</td>
<td>-[e:LivesIn]-</td>
<td>🔴</td>
</tr>
<tr>
<td>🔴</td>
<td>-[e1:LivesIn],</td>
<td>🔴</td>
</tr>
<tr>
<td></td>
<td>-[e2:LivesIn]-&gt;</td>
<td>🔴</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

Many Redundancies!

Difficult for Plan Generation
**GLogue**

\[ \text{Person} \rightarrow \text{City} \rightarrow \text{Country} \rightarrow \text{IsLocatedIn} \]

(a) Pattern, \( p \)

<table>
<thead>
<tr>
<th>( p_s )</th>
<th>( \tau )</th>
<th>( p_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Person}</td>
<td>\text{Knows} \rightarrow</td>
<td>\text{City}</td>
</tr>
<tr>
<td>\text{City}</td>
<td>\text{Knows} \rightarrow</td>
<td>\text{Country}</td>
</tr>
<tr>
<td>\text{Country}</td>
<td>\text{LivesIn} \rightarrow</td>
<td>\text{IsLocatedIn}</td>
</tr>
</tbody>
</table>

\( \text{edge property} \)

<table>
<thead>
<tr>
<th>( p_i )</th>
<th>( \mathcal{F}(p_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Person}</td>
<td>( f_1 )</td>
</tr>
<tr>
<td>\text{City}</td>
<td>( f_2 )</td>
</tr>
<tr>
<td>\text{Country}</td>
<td>( f_3 )</td>
</tr>
<tr>
<td>\text{IsLocatedIn}</td>
<td>( f_4 )</td>
</tr>
</tbody>
</table>

\( \text{vertex property} \)
Algorithm 1: Constructing GLogue: \((G^*, \text{Level})\).

1. \textbf{Function} constructGLogue \((\text{Level})\): GLogue
2. \textbf{Initialize} GLogue with \(\text{level} = 1, 2\) precomputed;
3. \textbf{for} \(3 \leq \text{Level} \leq \text{Level}\) \textbf{do}
4. \hspace{0.5cm} \textbf{Let} QSet maintain all \(i\) vertices in ascending order via \(|E_p|\);
5. \hspace{0.5cm} \textbf{for} \(p \in \text{QSet}\) \textbf{do}
6. \hspace{1.5cm} \textbf{updateFromPattern} \((p, \text{GLogue})\);
7. \textbf{return} GLogue;

8. \textbf{Function} updateFromPattern \((p, \text{GLogue})\)
9. \hspace{0.5cm} GLogue.addVertex \((p, \mathcal{F}(p))\);
10. \hspace{0.5cm} \textbf{for} \(p_i \in p\) \& \(|V_{p_i}| \neq |V_p| - 1\) \textbf{do}
11. \hspace{1.5cm} \textbf{if} not GLogue.contains \((p_i)\) \textbf{then}
12. \hspace{2cm} \textbf{updateFromPattern} \((p_i, \text{GLogue})\);
13. \hspace{1.5cm} \textbf{for} \(e = (s, t) \in E_p \setminus E_{p_i}\) \textbf{do}
14. \hspace{2cm} \textbf{let} \(p_i'\) \textbf{be the pattern with} \(e\) \textbf{adding to} \(p_i\);
15. \hspace{2cm} \textbf{eMap.insert} \((e, \mathcal{F}(p_i'))\);
16. \hspace{2cm} GLogue.addEdge \(\{(p, p_i), \text{eMap}\}\);
17. \hspace{0.5cm} \textbf{for} \(p_i, p_{i+1} \in p\) \& \(E_{p_i} \cup E_{p_{i+1}} = E_p\) \textbf{do}
18. \hspace{1.5cm} \textbf{if} not GLogue.contains \((p_{i+1})\) \textbf{then}
19. \hspace{2cm} \textbf{updateFromPattern} \((p_{i+1}, \text{GLogue})\);
20. \hspace{1.5cm} \textbf{if} not GLogue.contains \((p_{i+2})\) \textbf{then}
21. \hspace{2cm} \textbf{updateFromPattern} \((p_{i+2}, \text{GLogue})\);
22. \hspace{1.5cm} GLogue.addEdge \(\{(p_i, p_{i+1}), \mathcal{F}(p_i), \mathcal{F}(p_{i+1})\}\);
Plan Optimizer

Algorithm 1: The Plan Optimizer.

```
1 Function PlanOptimizer (GLogue, PatternDesc)
2     Construct a pattern p from the PatternDesc;
3     Let QSet organize all induced subgraphs of p by level;
4     Initialize a PlanMap to record \{ p : (plan, cost) \} with patterns in level 1 and 2 pre-computed;
5     for 3 ≤ level ≤ |V_p| do
6         for p ∈ QSet[level] do
7             searchPlan (p, PlanMap, GLogue);
8     return PlanMap.get(p);
9 Function searchPlan (p, PlanMap, GLogue)
10     Initialize Plan(p) and Cost(Plan(p)) ← ∞;
11     for edge = (p_1, p) ∈ GLogue.getEdges(p) do
12         (plan1, cost1) ← PlanMap.get(p_1);
13         if edge is a vertex extension then
14             Compute a new plan' by merging plan1 and Expand(p_1 → p);
15             else if edge \{ (p_2, F(p_2)) \} is binary join then
16                 (plan2, cost2) ← PlanMap.get(p_2);
17                 Compute a new plan' by merging plan1, plan2 and Join\{ (p_1, p_2) → p \};
18             Compute a new cost' of plan' by Equation 1;
19             if cost' < Cost(Plan(p)) then
20                 Update Plan(p) as plan' and the cost as cost';
21     PlanMap.insert(p, (Plan(p), Cost(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.
  • $\mathcal{F}(p) = \text{Avg}_{p_1,p_2} \frac{\mathcal{F}(p_1) \times \mathcal{F}(p_2)}{\mathcal{F}(p_1 \cap p_2)}$, $p = p_1 \cup p_2$
Environment and Datasets

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

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

P1

P2

P3

Person City Country Tag
LocatedIn HasInterest HasTag
HasCreator ReplyOf
Comment Post Forum
Knows Likes HasCreator
LivesIn ContainerOf
HasMember

\[ p_1 \] 9882

\[ p_2 \] 2052168

\[ p \] 41713

\[ p_1 \] 212

\[ p_2 \] 9882

\[ p_3 \] 83558

\[ p \] 19314

\[ p_1 \] 1003605

\[ p_2 \] 1011420

\[ p_3 \] 1011420

\[ p \] 444419
Experiment 1: V.S. Neo4j and TigerGraph

GLogs achieves 51× and 57% speedup compared with Neo4j and TigerGraph, respectively.
## Experiment 2: High-order Statistics

<table>
<thead>
<tr>
<th></th>
<th>Level = 2</th>
<th>Level = 3</th>
<th>Level = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow-down(%)</td>
<td>966</td>
<td>245</td>
<td>243</td>
</tr>
<tr>
<td>Generation Time(s)</td>
<td>6</td>
<td>55</td>
<td>1664</td>
</tr>
<tr>
<td>Memory Usage(GB)</td>
<td>2</td>
<td>3</td>
<td>105</td>
</tr>
<tr>
<td># Patterns</td>
<td>34</td>
<td>248</td>
<td>4164</td>
</tr>
</tbody>
</table>

Table: The effectiveness of high-order statistics
Experiment 3: Sparsification

Uniform sparsification requires a higher rate to achieve the same performance as stratified sparsification. Small graphs require a higher rate to have a good plan.
Experiment 4: Scalability

Most queries scale well, with up to $15 \times$ (average $6 \times$) performance gain from one machine to 16.
Experiment 4: Scalability

Up to $23 \times$ (average of $10 \times$) 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.
Experiment 4: Scalability

As the graphs become larger, most queries demonstrate an almost linear trend in performance degradation. For $p_6$, its performance degrades by $3 \times$ from $G_{30}$ to $G_{1000}$, due to its short duration and limited graph exploration. For $p_{10}$, its performance degrades by $100 \times$ 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!
THANK YOU