Distributed Subgraph Matching on Timely Dataflow

Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, Jingren Zhou
Outline

- Introduction
- Literature Survey
- Experiment Results & Observations
- A Practical Guide
Introduction
Subgraph Matching

Given a pattern graph $P$ and a data graph $G$ (both are undirected, unlabelled simple graph), the problem is to find all subgraph instances (matches) $g'$ in $G$ that are isomorphic to $P$. 
Subgraph Matching

Given a pattern graph $P$ and a data graph $G$ (both are undirected, unlabelled simple graph), the problem is to find all subgraph instances (matches) $g'$ in $G$, that are isomorphic to $P$. 
Subgraph Matching

Given a pattern graph $P$ and a data graph $G$ (both are undirected, unlabelled simple graph), the problem is to find all subgraph instances (matches) $g'$ in $G$, that are isomorphic to $P$. 
Subgraph Matching

Given a pattern graph $P$ and a data graph $G$ (both are undirected, unlabelled simple graph), the problem is to find all subgraph instances (matches) $g'$ in $G$, that are isomorphic to $P$. 

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0</td>
<td>u1</td>
<td>u2</td>
<td>u3</td>
<td>u4</td>
</tr>
<tr>
<td>u1</td>
<td>u2</td>
<td>u3</td>
<td>u4</td>
<td>u5</td>
</tr>
</tbody>
</table>

In the example, $u2$ and $u3$ are matched with $v2$ and $v3$ respectively.
Distributed Subgraph Matching

- Distributed Solutions for **performance** and **scalability**
  - Computational Intractability: Subgraph Isomorphism is NPC
  - Graphs are now easily in billion-scale

- Join-based Algorithms
  - Subgraph Matching can be naturally expressed using joins
  - Join operation can be easily distributed
  - Many systems natively support join operations
A Thriving Literature


Extension

CliqueJoin [4]

MultiwayJoin [5]
BiGJoin [6]
CrystalJoin [7]
What algorithm performs the best?

- Every new paper claims better performance. But?
  - Different languages based on different systems (system cost ignored)
  - *Hardcoded* optimizations for each query
  - Existing implementations intertwine *Strategies* and *Optimizations*
## What algorithm performs the best?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Strategy</th>
<th>System/Lang</th>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>StarJoin [1]</td>
<td>BinaryJoin</td>
<td>Trinity Memory / C++</td>
<td>None</td>
</tr>
<tr>
<td>PSgL [2]</td>
<td>BinaryJoin/Others</td>
<td>Giraph / Java</td>
<td>None</td>
</tr>
<tr>
<td>MultiwayJoin [5]</td>
<td>Shares HypherCube</td>
<td>Myria / Java</td>
<td>N / A</td>
</tr>
<tr>
<td>CrystalJoin [7]</td>
<td>Others</td>
<td>Hadoop / Java</td>
<td>Compression</td>
</tr>
</tbody>
</table>
**Our Contributions**

<table>
<thead>
<tr>
<th>A Common System</th>
<th>All Optimizations</th>
<th>In-depth Experiments</th>
<th>A practical guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>A benchmarking platform based on Timely dataflow system for distributed subgraph matching.</td>
<td>Three general-purpose optimizations - Batching, TrIndexing and Compression - to apply to all strategies while possible</td>
<td>A complete variations of data graphs, query graphs, strategies and optimizations</td>
<td>A practical guide for distributed subgraph matching based on empirical analysis covering the perspectives of join strategies, optimizations and join plans.</td>
</tr>
</tbody>
</table>
Timely Dataflow System

- A general-purpose data-parallel distributed dataflow system [10]
- Computation is abstracted as dataflow graph
  - DAG, but allowing loops in the loop context
  - Operators are vertices that define computing logics
  - Data flows are directed edges that chain operators

- Reasons of using Timely dataflow
  - Small system cost [11]: the impact of system can be reduced to minimum
  - Low-level primitive operators: flexible enough to implement all benchmarking algorithms
Literature Survey
Categorizing by Strategies

- TwinTwigJoin [3]
- StarJoin [1]
- CliqueJoin [4]
- CrystalJoin [7] (Others)
- MultiwayJoin [5] (ShrCubes)
- PSgL [2] (BinaryJoin)

A variant

Constraints

Extension
BinaryJoin Strategy

- Divide the pattern graph into a set of join units \{ p_1, p_2, ..., p_k \}
- Process k-1 binary joins following specific join order

We prove that CliqueJoin is worst-case optimal by showing that it can be expressed as GenericJoin proposed by Ngo et al. [8]
StarJoin Algorithm

Round 1

Round 2

Round 3

Round 4
TwinTwigJoin Algorithm

Round 1

Round 2

Round 3

Round 4

StarJoin

TwinTwigJoin
CliqueJoin Algorithm

Round 1

Round 2

Round 3

Round 4

StarJoin

TwinTwigJoin

CliqueJoin
WOptJoin

- BinaryJoin: “growing by graphs (i.e. join units)”
  - Given a vertex order \( \{v_1, v_2, \ldots, v_n\} \)
  - Start by matching \( v_1 \), then \( \{v_1, v_2\} \) and so on until constructing the final results
  - BiGJoin follows this strategy, and is implemented on Timely dataflow
WOptJoin

- BinaryJoin: “growing by graphs (join units)”
  - Given a vertex order \{v_1, v_2, ..., v_n\}
  - Start by matching \(v_1\), then \(\{v_1, v_2\}\) and so on until constructing the final results
  - BiGJoin follows this strategy, and is implemented on Timely dataflow
BiGJoin Algorithm

- Based on Ngo’s worst-case optimal join algorithm [8]
- Concepts:
  - Prefix: the current partial results
  - Prefix*: the projection of Prefix on the vertices that are connecting current vertex in the pattern graph
- Three operators on Timely Dataflow
  - Count: Count # neighbors of each vertex in the prefix*
  - Propose: Attach the neighbors that are the smallest among the prefix*’s vertices
  - Intersect: Process set intersection among the neighbors of all associated vertices
BiGJoin Algorithm

Prefix = Prefix*
as v4 connecting both v1 and v3

Count: (u₁, 4), (u₂, 3)  // count # neighbors
Propose: \{u₁, u₂\}, N(u₂) = \{u₁, u₃, u₄\}  // Propose on the one with smallest number of neighbors
Intersect: \{u₁, u₂\}, \{u₁, u₃, u₄\} \cap N(u₁) = \{u₃, u₄\}  // Intersect with the other vertices’ neighbors
Next Prefix: \{u₁, u₂, u₃\}, \{u₁, u₂, u₄\}  // Flatmap to get the next partial results w.r.t Prefix \{u₁, u₂\}
Shares of Hypercube

- Given a pattern graph of n vertices, the searching space forms an n-dimensional hypercube
  \[ V \times V \times \cdots \times V \]

- The idea of sharing
  - Divide V into b shares \( \{V_1, V_2, \ldots, V_n\} \), where \( b = \sqrt[\# \text{machines}]{n} \)
  - The machine indexed by \( \{x_1, x_2, \ldots, x_n\} \) where \( 1 \leq x_i \leq b \), handles the share of
    \[ V_{x_1} \times V_{x_2} \times \cdots \times V_{x_n} \]

- MultiwayJoin Algorithm (details in the paper)
Optimizations

- Three general-purpose optimizations
  - Batching
  - Triangle Indexing
  - Compression (Factorization)

- Methodologies
  - We apply all optimizations to both BinaryJoin and WOptJoin strategies
  - Focus on strategy-level comparison in order to see what cause the performance gains, strategies or optimizations
  - Hand-written optimizations are excluded
Details of Compression

- Originally proposed by Qiao et al. [7]
- Intuition
  - Subgraph enumeration can generate enormous (intermediate) results
  - Some vertices can be compressed as they are not needed in future computation
    - Heuristics by [7]: the vertices that do not belong to the minimum vertex cover (MVC)
Details of Compression

- Originally proposed by Qiao et al. [7]
- Intuition
  - Subgraph enumeration can generate enormous (intermediate) results
  - Some vertices can be compressed as they are not needed in future computation
    - Heuristics by [7]: the vertices that do **not** belong to the minimum vertex cover (MVC)

\[
\begin{align*}
  v_1 &\rightarrow u_1, v_3 &\rightarrow u_2 \\
  N(u_1) &= \{u_2, u_3, \ldots, u_{1003}\} \\
  N(u_2) &= \{u_1, u_3, \ldots, u_{1003}\} \\
  \text{Match}(v_0, v_2) &= N(u_1) \cap N(u_2) = \{u_3, u_4, \ldots, u_{1003}\}
\end{align*}
\]

- Vertices in MVC
- Vertices to compress
Details of Compression

- Originally proposed by Qiao et al. [7]
- Intuition
  - Subgraph enumeration can generate enormous (intermediate) results
  - Some vertices can be compressed as they are not needed in future computation
    - Heuristics by [7]: the vertices that do **not** belong to the minimum vertex cover (MVC)

```
v1  →  u_1, v_3  →  u_2
N(u_1) = \{u_2, u_3, \ldots, u_{1003}\}
N(u_2) = \{u_1, u_3, \ldots, u_{1003}\}
Match(v_0, v_2) = N(u_1) \cap N(u_2) = \{u_3, u_4, \ldots, u_{1003}\}
```

Decompose into \( \binom{1000}{2} \) results
Experiment Results & Observations
Experiment Settings

- **Local Cluster**: 10 machines connected via one 10GBps switch and one 1GBps switch. Each machine has 4 cores and 64GB memory
- **Metrics**
  - T: The slowest worker’s wall clock time.
    - 3 hours maximum, **OT** if running out of time
    - Tp, *computation time*: timing all computation-related functions, and take the slowest among the workers
    - Tc, *communication time*: Tc = T - Tp
Effects of Optimizations

![Diagram with bar charts showing the effects of different optimizations on query processing time for BinaryJoin and WOptJoin. The diagrams compare the performance of AllOpt, w.o. Batching, w.o. Trindexing, and w.o. Compression.]

LJ dataset: 4.85M vertices, 43.37M edges
Observations

- **Batching**
  - Batching greatly reduces memory consumption, but barely affects performance.

- **Triangle Indexing**
  - By average it takes 5 times more storage to index triangles on the studied datasets.
  - It has **critical** impact for BinaryJoin.
  - It is effective for WOptJoin when the network is slow (1GBps), but less so when it is fast.

- **Compression**
  - Compression may introduce more cost than gains on very-sparse graphs like road network.

- **All optimizations are applied for BinarayJoin and WOptJoin in the following**
Challenging Queries

USRoad dataset: 23.95M vertices, 28.85M edges

Google dataset: 0.86M vertices, 4.32M edges
Observations

- The cost-based “optimal” join plan given by CliqueJoin [4] does not always render the best performance
  - e.g., “Tailed triangle” (TR) vs “House” (H)
  - In theory, TR has lower estimated cost [4], and lower worst-case bound [8] than H
  - In practice, TR is as costly as H, and joining two TRs in the “optimal” plan makes it worse
Observations

● The heuristics of Crystaljoin
  ○ MVC-first + compress the remaining
  ○ It guarantees the best compression [7], but prioritizing computing MVC can be costly
  ○ e.g.
    ■ Note that we use connected “MVC” [9] instead of actual MVC
    ■ The “MVC”-first plan is very expensive as “MVC” is a costly 5-path
  ○ When it produces strictly larger compression
Observations

- The case that Crystaljoin indeed performs better
  - When it produces strictly larger compression
  - e.g.
    - Crystaljoin’s plan now compresses three vertices
    - BiGJoin (when applying compression), can only compress two vertices
All-around Comparisons

- **6 Queries**
  - $v_1$, $v_2$, $v_3$, $v_4$
  - $q_1$: $\{(v_1,v_2),(v_1,v_3),(v_2,v_4)\}$
  - $q_2$: $\{(v_1,v_3),(v_2,v_4)\}$
  - $q_3$: $\{(v_1,v_2),(v_2,v_3),(v_3,v_4)\}$
  - $q_4$: $\{(v_1,v_3),(v_2,v_5)\}$
  - $q_5$: $\{(v_2,v_5),(v_3,v_4)\}$
  - $q_6$: $\{(v_1,v_4)\}$

- **5 Datasets**
  - Varieties of types: Web Graph, social networks and road networks
  - Varieties of sizes: 12M edges ~ 1806M edges
  - Varieties of densities (avg degree): 4 ~ 218

- **4 Strategies**
  - BinaryJoin, WOptJoin, Shares of HyperCube (SHRCube), FullRep
All-around Comparisons

Tc: shadowed fillings of the bars
Tp: white fillings of the bars
Observations

- FullRep typically outperforms the other strategies
- Computation time $T_p$ dominates in most cases
  - Observed in the 10Gbps network
  - Communication time dominates in the slower network (1Gbps)
  - The distributed subgraph matching tends to be computation-intensive
A Practical Guide
Input: G, Q

Does the machine have sufficient memory to hold G

Y

FullRep

N

Does the machine have sufficient memory to hold 60% of G

Y

Apply triangle partition

N

Apply normal hash partition

If Q is a clique or a star

Y

BinJoin

N

Does BinJoin plan render less cost (via cost estimation or worst-case bound)

Y

BinJoin

N

WOptJoin

Note: Do not apply compression when G is very sparse (avg_deg < 5)
Q & A

Working on open-sourcing, bins available for verifying the results:
References

1. Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. PVLDB, 5(9), 2012.
2. Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In SIGMOD'14, pages 625-636.