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

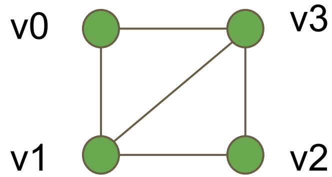
Outline

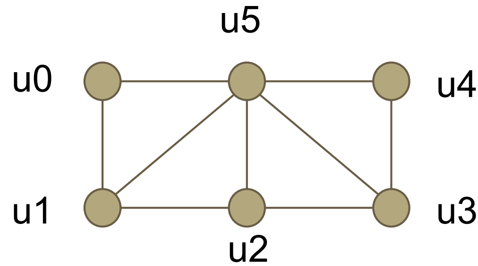
- Introduction
- Literature Survey
- Experiment Results & Observations
- A Practical Guide

Introduction

Subgraph Matching

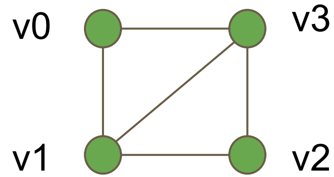
Given a pattern graph P and a data graph G (both are undirected, unlabelled simple graph), the problem is to find **all** subgraph instances (matches) g' in G, that are **isomorphic** to P.



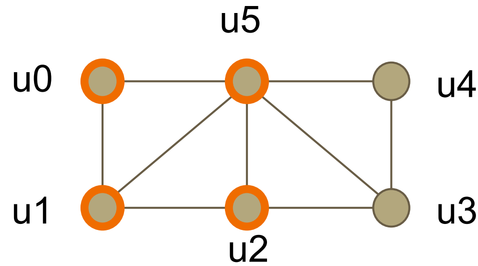


Subgraph Matching

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.



P

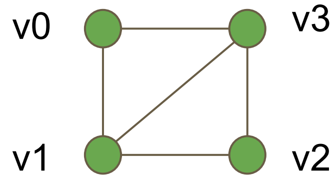


G

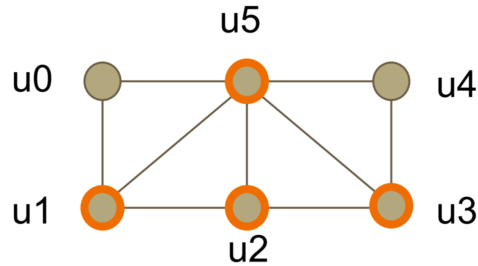
v0	v1	v2	v3
u0	u1	u2	u5

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.



P

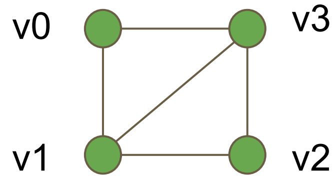


G

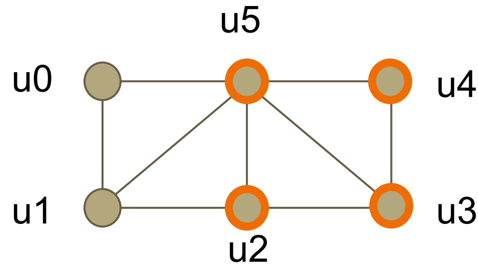
v0	v1	v2	v3
u0	u1	u2	u5
u1	u2	u3	u5

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.



P



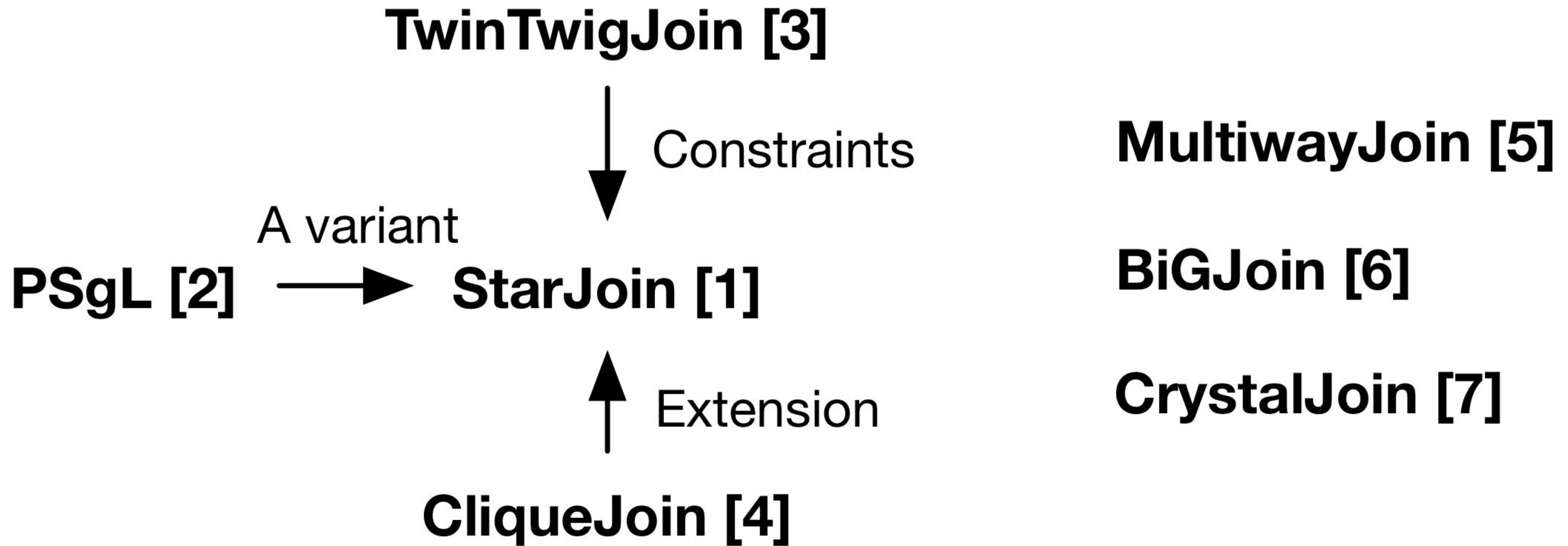
G

v0	v1	v2	v3
u0	u1	u2	u5
u1	u2	u3	u5
u2	u3	u4	u5

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



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?

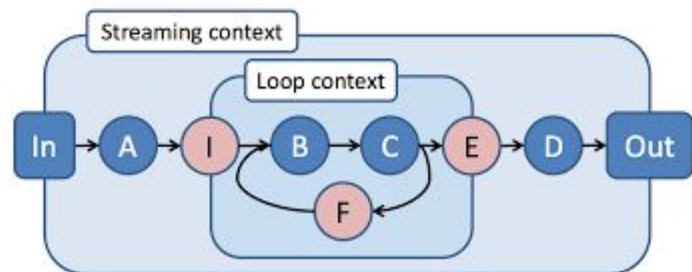
Algorithm	Strategy	System/Lang	Optimizations
StarJoin [1]	BinaryJoin	Trinity Memory / C++	None
PSgL [2]	BinaryJoin/Others	Giraph / Java	None
TwinTwigJoin [3]	BinaryJoin	Hadoop / Java	Compression
CliqueJoin [4]	BinaryJoin	Hadoop / Java	Triangle Indexing, Compression
MultiwayJoin [5]	Shares HypherCube	Myria / Java	N / A
BiGJoin [6]	WOptJoin	Timely / Rust	Batching, specific Triangle Indexing
CrystalJoin [7]	Others	Hadoop / Java	Compression

Our Contributions

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

Timely Dataflow System

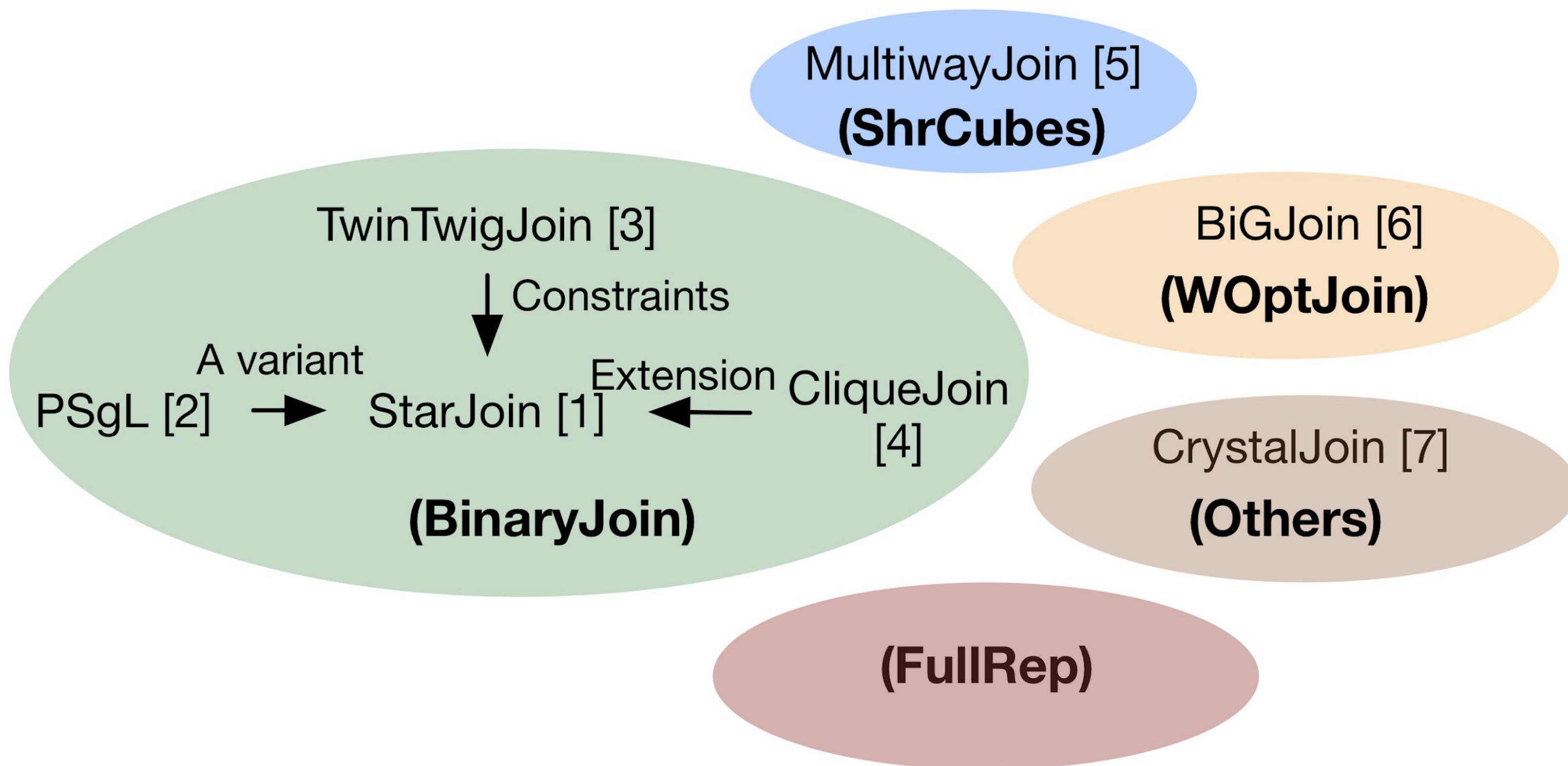
- A general-purpose data-parallel distributed dataflow system [10]
- Computation is abstracted as *dataflow graph*
  - DAG, but allowing loops in the loop context
  - Operators are vertices that define computing logics
  - Data flows are directed edges that chain operators



- 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

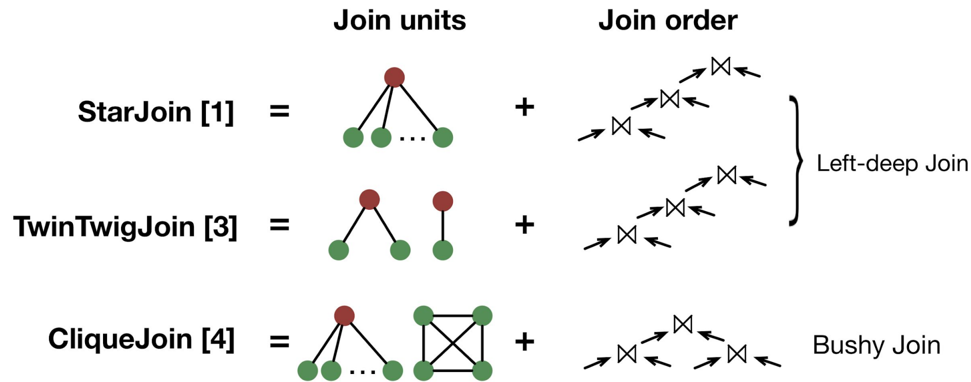
Literature Survey

Categorizing by Strategies



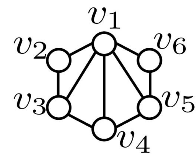
BinaryJoin Strategy

- Divide the pattern graph into a set of **join units** { p1, p2, ..., pk }
- Process k-1 binary joins following specific **join order**



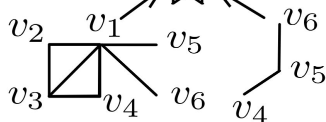
- We prove that CliqueJoin is *worst-case optimal* by showing that it can be expressed as **GenericJoin** proposed by Ngo et al. [8]

StarJoin Algorithm

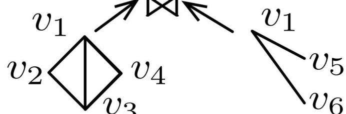


Round 4

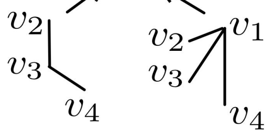
Round 3



Round 2

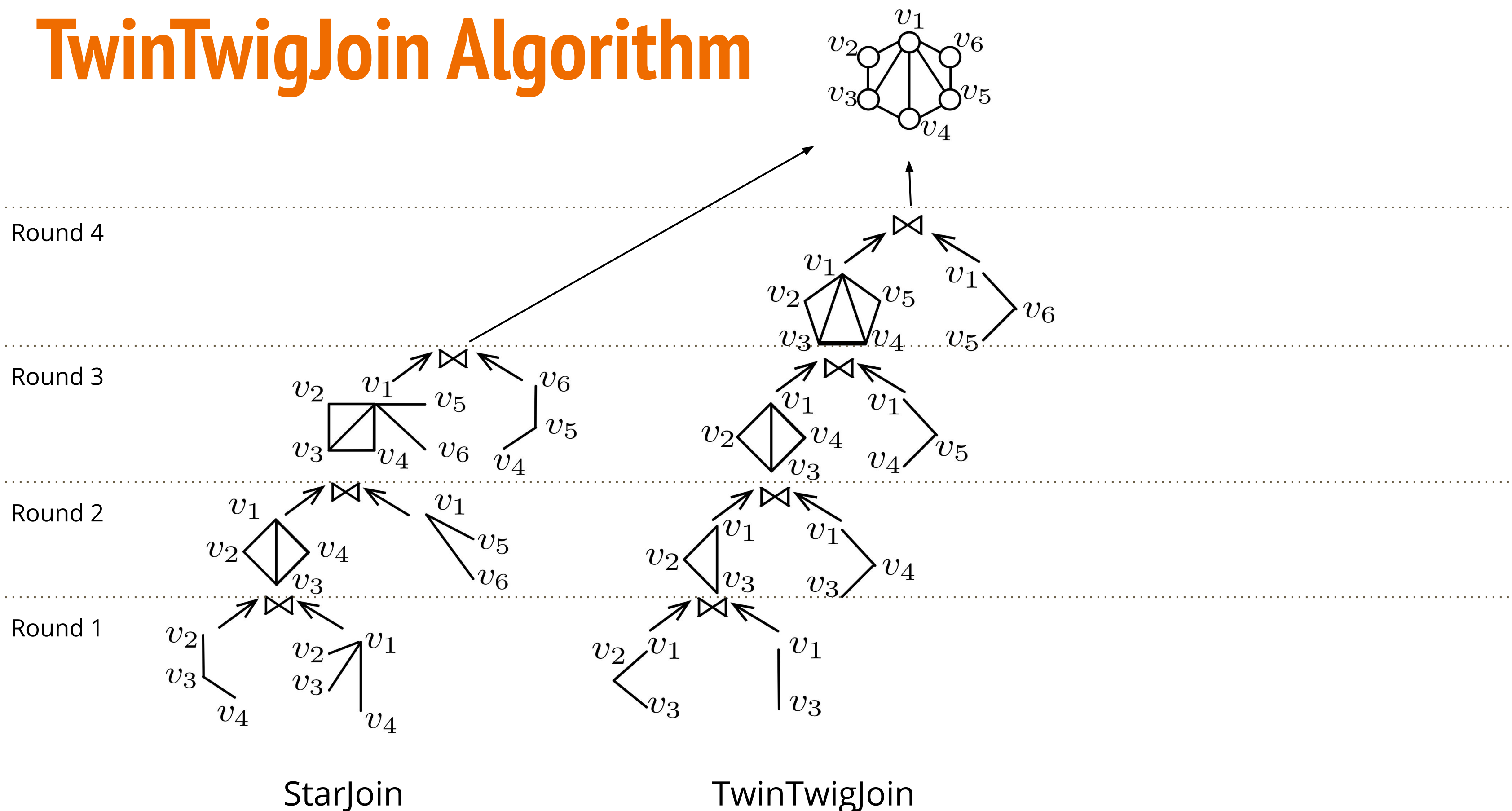


Round 1

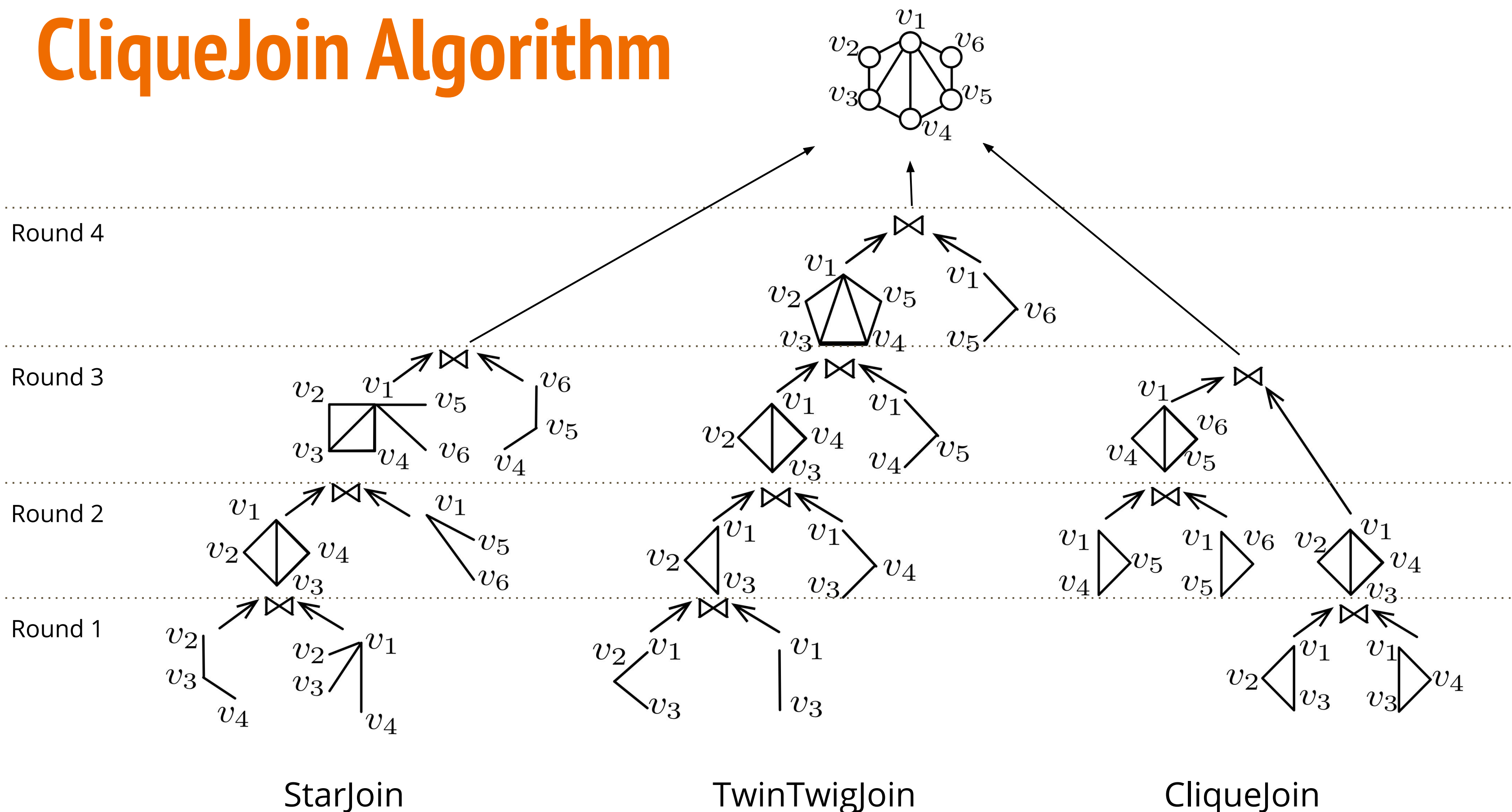


StarJoin

TwinTwigJoin Algorithm



CliqueJoin Algorithm

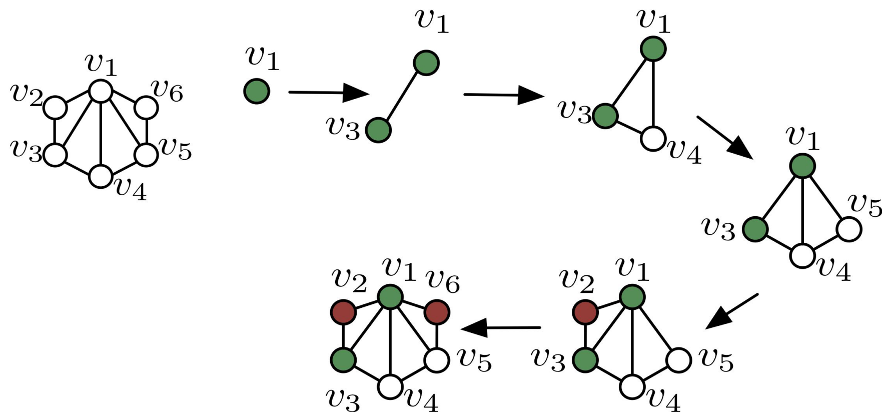


WOptJoin

- BinaryJoin: “growing by graphs (i.e. join units)”
- WOptJoin: “growing by vertices” [6]
  - Given a vertex order {v1, v2, ..., vn}
  - Start by matching v1, then {v1, v2} 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)”
- WOptJoin: “growing by vertices” [6]
  - Given a vertex order {v1, v2, …, vn}
  - Start by matching v1, then {v1, v2} and so on until constructing the final results
  - BiGJoin follows this strategy, and is implemented on Timely dataflow

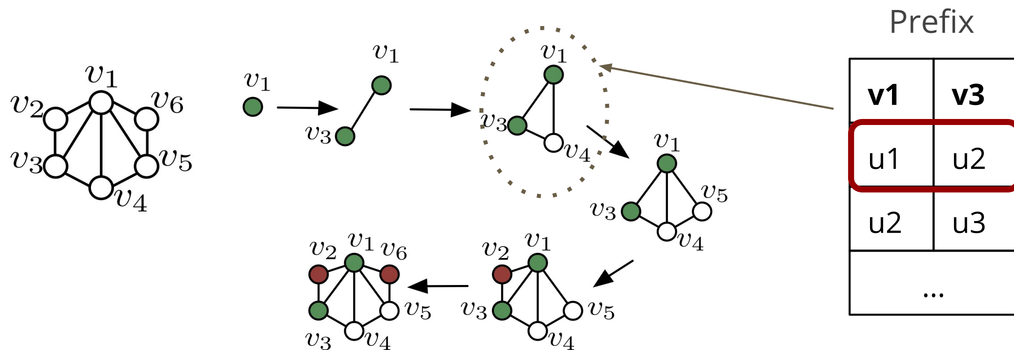


BiGJoin Algorithm

- 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



Prefix

v1	v3
u1	u2
u2	u3
…	

$$N(u_1) = \{u_2, u_3, u_4, u_5\}$$
$$N(u_2) = \{u_1, u_3, u_4\}$$

Count: $(u_1, 4), (u_2, 3)$ *// count # neighbors*

Propose: $\{u_1, u_2\}, N(u_2) = \{u_1, u_3, u_4\}$ *// Propose on the one with smallest number of neighbors*

Intersect: $\{u_1, u_2\}, \{u_1, u_3, u_4\} \cap N(u_1) = \{u_3, u_4\}$ *// Intersect with the other vertices' neighbors*

Next Prefix: $\{u_1, u_2, u_3\}, \{u_1, u_2, u_4\}$ *// Flatmap to get the next partial results w.r.t Prefix* $\{u_1, u_2\}$

Shares of Hypercube

- Given a pattern graph of n vertices, the searching space forms an n-dimensional hypercube
  - $\underbrace{V \times V \times \cdots \times V}_{n}$

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

- MultiwayJoin Algorithm (details in the paper)

Optimizations

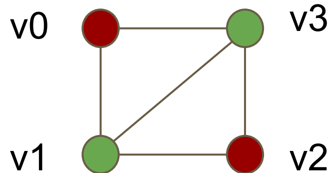
- Three general-purpose optimizations
  - Batching
  - Triangle Indexing
  - Compression (Factorization)
- Methodologies
  - We apply all optimizations to both BinaryJoin and WOptJoin strategies
  - Focus on strategy-level comparison in order to see what cause the performance gains, strategies or optimizations
  - 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)



Vertices in MVC

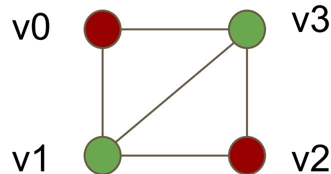
Vertices to compress

$$v_1 \rightarrow u_1, v_3 \rightarrow u_2$$
$$N(u_1) = \{u_2, u_3, \ldots, u_{1003}\}$$
$$N(u_2) = \{u_1, u_3, \ldots, u_{1003}\}$$

$$Match(v_0, v_2) = N(u_1) \cap N(u_2) = \{u_3, u_4, \ldots, u_{1003}\}$$

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)



Vertices in MVC

Vertices to compress

$$v_1 \rightarrow u_1, v_3 \rightarrow u_2$$
$$N(u_1) = \{u_2, u_3, \ldots, u_{1003}\}$$
$$N(u_2) = \{u_1, u_3, \ldots, u_{1003}\}$$
$$Match(v_0, v_2) = N(u_1) \cap N(u_2) = \{u_3, u_4, \ldots, u_{1003}\}$$

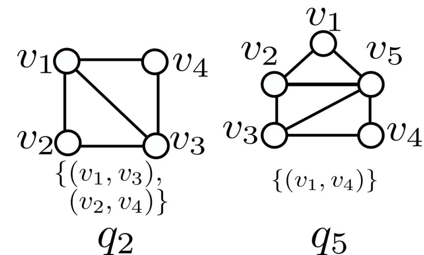
Decompose into $\binom{1000}{2}$ results

Experiment Results & Observations

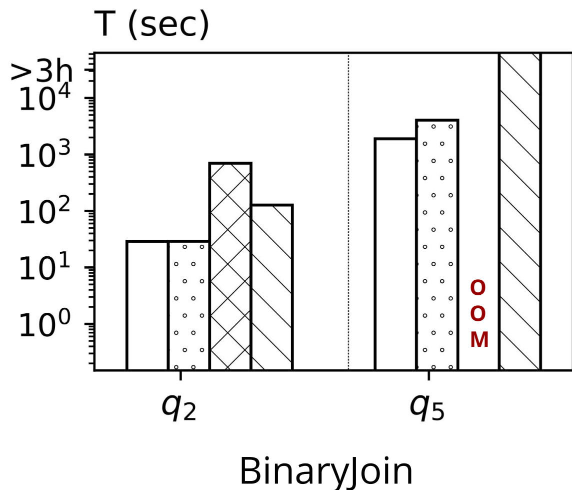
Experiment Settings

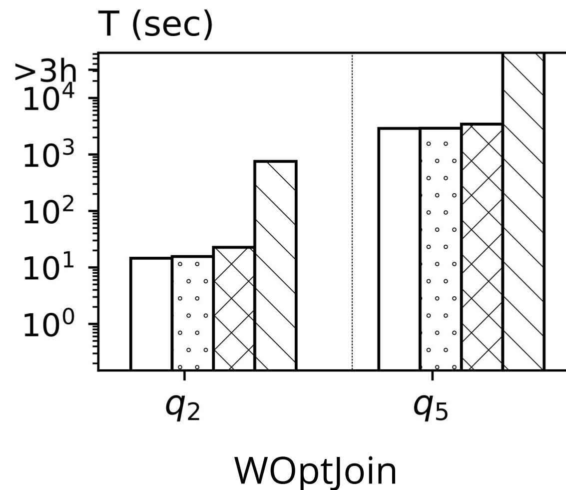
- **Local Cluster**: 10 machines connected via one 10GBps switch and one 1GBps switch. Each machine has 4 cores and 64GB memory
- Metrics
  - T: The slowest worker's wall clock time.
    - 3 hours maximum, **OT** if running out of time
    - Tp, *computation time*: timing all computation-related functions,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

Effects of Optimizations



AllOpt w.o. Batching w.o. Trindexing w.o. Compression



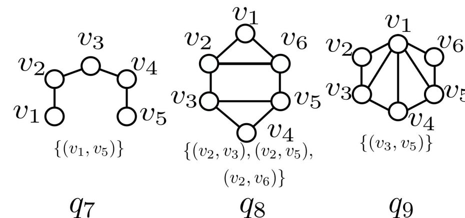


LJ dataset: 4.85M vertices, 43.37M edges

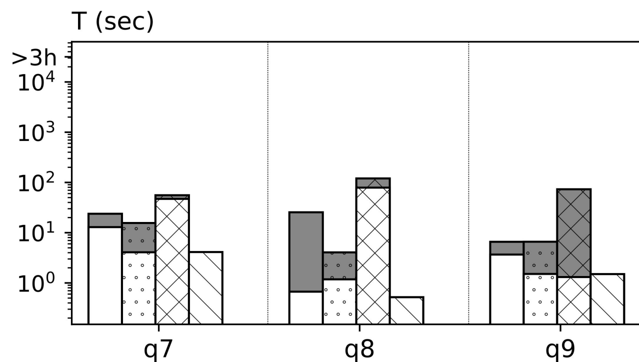
Observations

- 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

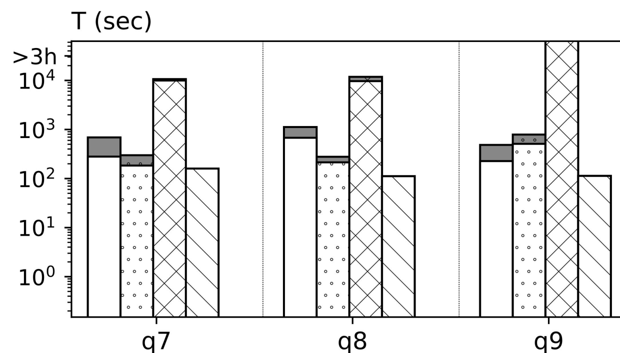
Challenging Queries



BinJoin WOptjoin ShrCube FullRep



USRoad



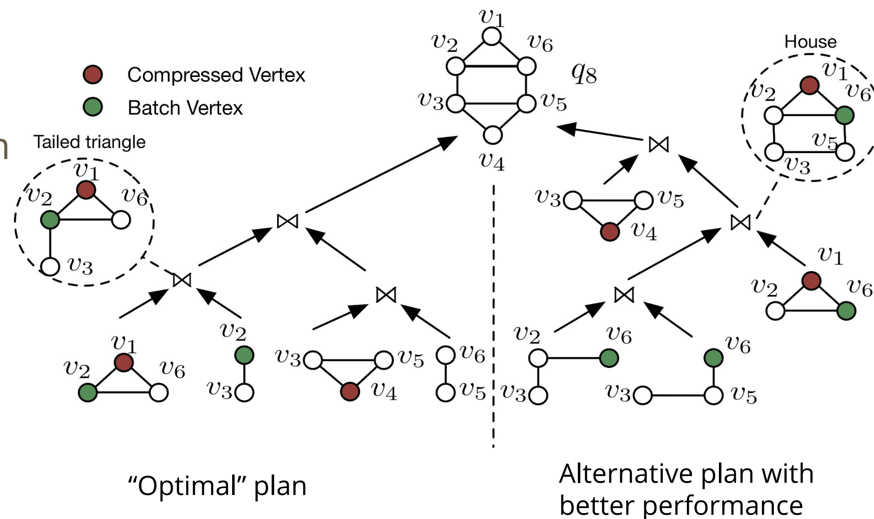
Google

USRoad dataset: 23.95M vertices, 28.85M edges

Google dataset: 0.86M vertices, 4.32M edges

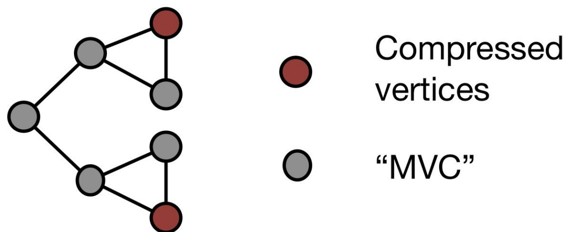
Observations

- 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

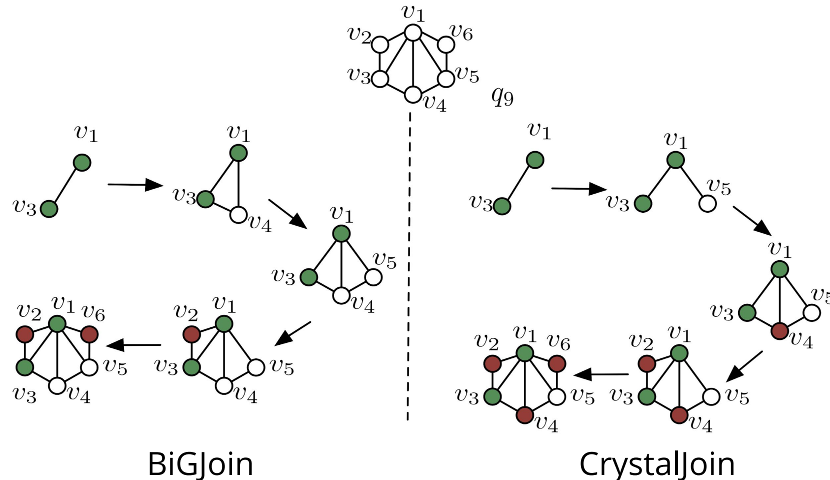
- The heuristics of Crystaljoin
  - MVC-first + compress the remaining
  - It guarantees the best compression [7], but prioritizing computing MVC can be costly
  - e.g.
    - Note that we use connected “MVC” [9] instead of actual MVC
    - The “MVC”-first plan is very expensive as “MVC” is a costly 5-path



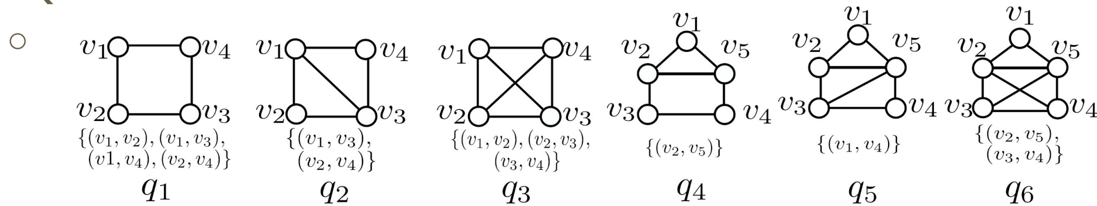
  - When it produces strictly larger compression

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
  - 

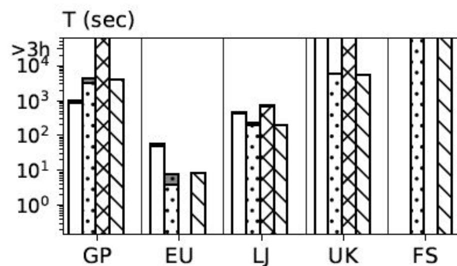
- 5 Datasets
  - Varieties of types: Web Graph, social networks and road networks
  - Varieties of sizes: 12M edges ~ 1806M edges
  - Varieties of densities (avg degree): 4 ~ 218
- 4 Strategies
  - BinaryJoin, WOptJoin, Shares of HyperCube (SHRCube), FullRep

All-around Comparisons

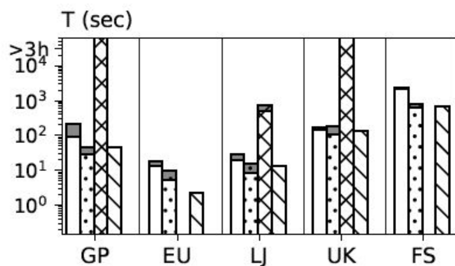
Tc: shadowed fillings of the bars

Tp: white fillings of the bars

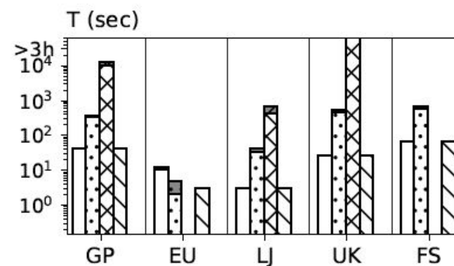
BinJoin WOptjoin ShrCube FullRep



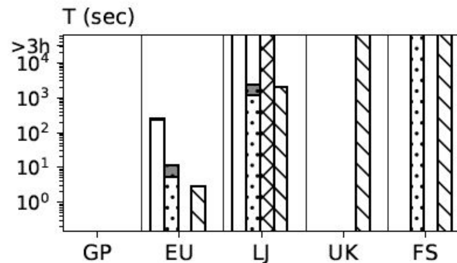
(a) q_1



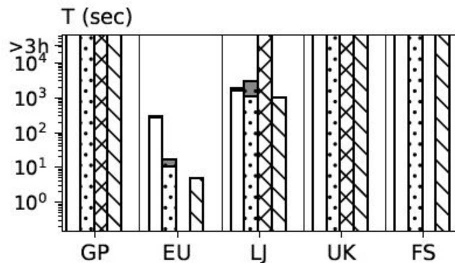
(b) q_2



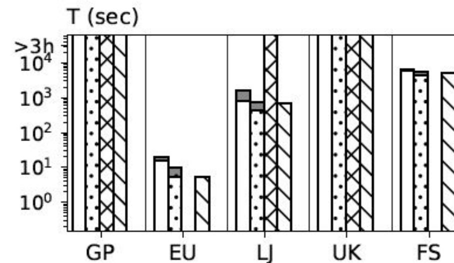
(c) q_3



(d) q_4



(e) q_5

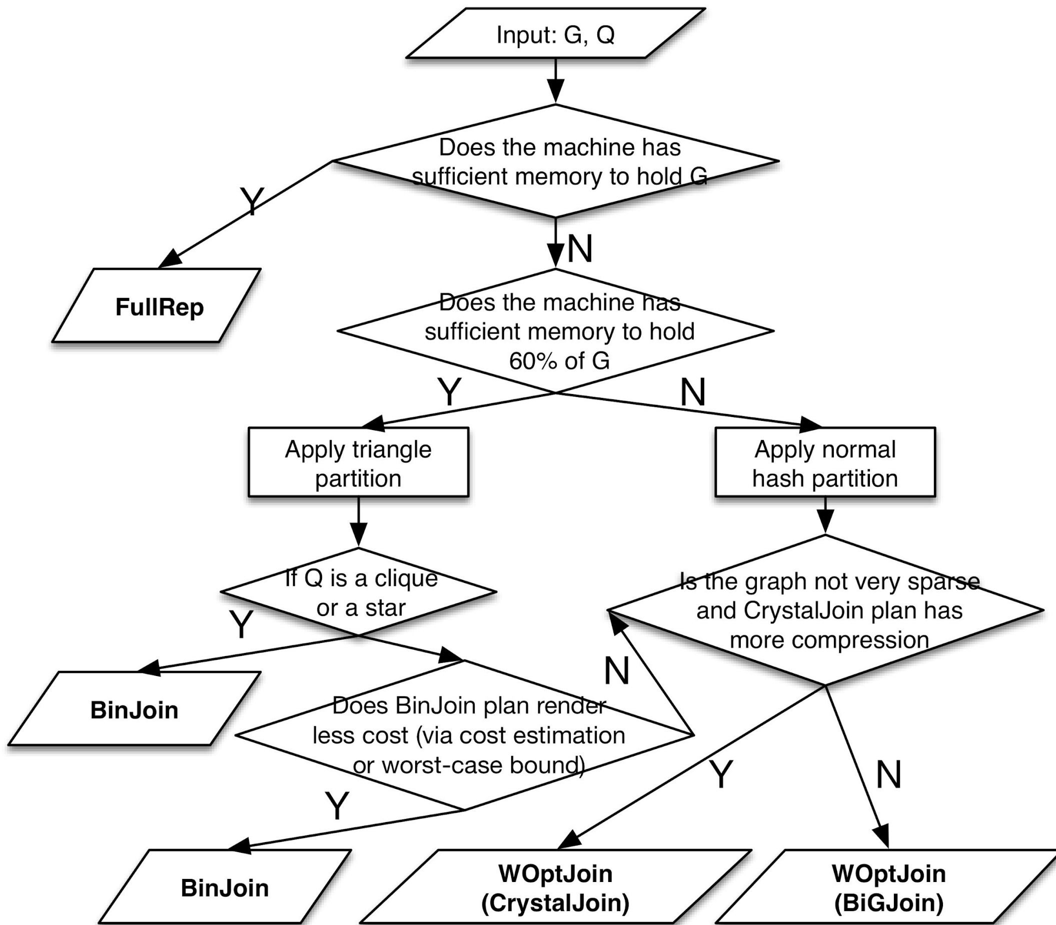


(f) q_6

Observations

- FullRep typically outperforms the other strategies
- Computation time Tp 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



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.
3. L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. PVLDB, 8(10), 2015.
4. L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. PVLDB, 10(3), 2016.
5. F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In Proc. of ICDE, 2013.
6. K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. PVLDB, 11(6), 2018.
7. M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: On compression and computation. PVLDB, 11(2), 2017.
8. H. Q. Ngo, E. Porat, C. Re, and A. Rudra. Worst-case optimal join algorithms. J. ACM, 65(3), 2018.
9. H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. SIGMOD '16, pages 1231{1245, 2016.
10. D.G. Murray, F.McSherry, R. Isaacs, M.Isard, P.Barham, and M.Abadi, Naiad: A Timely Dataflow System. SOSP 13.
11. F.McSherry, M.Isard, D.G.Murray, Scalability! But at what COST? HotOS 2015.