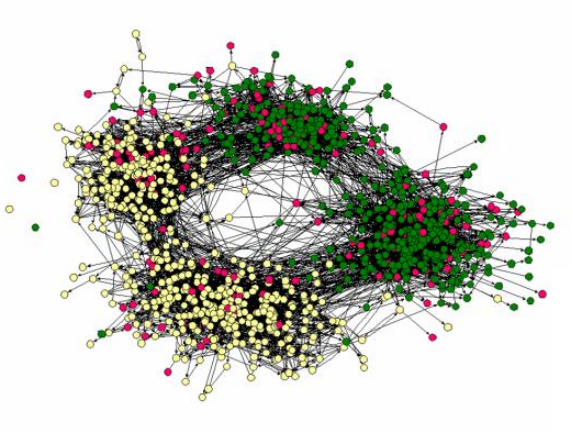
Big Graph Data Processing

Wook-Shin Han
POSTECH
Apr. 11, 2014

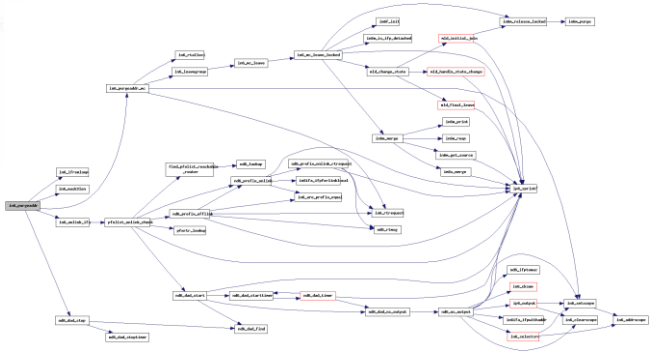
Outline

- **Big (Graph) Data**
- Big Graph Processing Techniques
- Conclusion

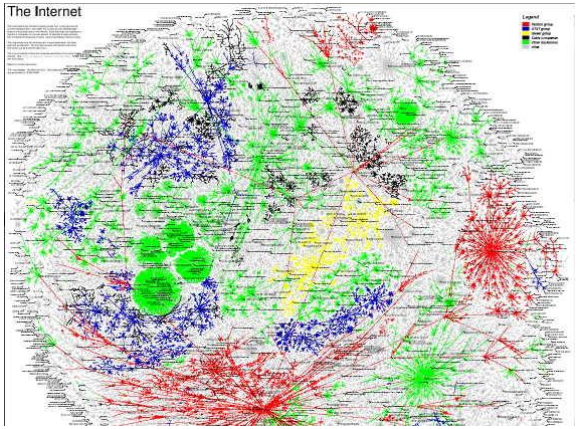
Welcome to Graph World



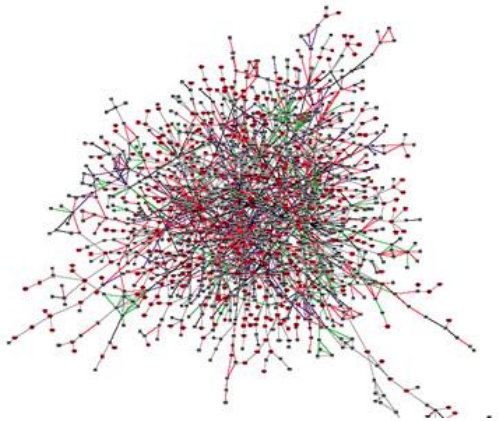
Friendship network



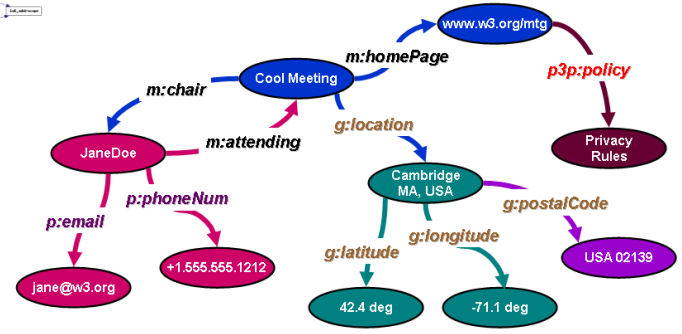
Call Graph



Internet map

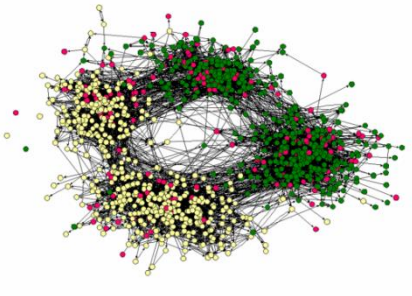


Protein interactions



Semantic web

Big Graphs in Real World



	# of nodes (vertices)	# of edges
Facebook	***1Billion***	**140 Billion**
Twitter	640 Million	10 Billion
LinkedIn	60 Million	0.9 Billion
Last.FM	40 Million	2 Billion
LiveJournal	25 Million	2 Billion
del.icio.us	5.3 Million	0.7 Billion

Outline

- My Background
- Big (Graph) Data
- **Big Graph Data Processing**
- Conclusion

Big Graph Processing

[VLDB10, SIGMOD11, VLDB13, SIGMOD13, KDD13, SIGMOD14]

iGraph v1.0

•Han, W., Lee, J., Duc, P., and Yu, J., "iGraph: A Framework for Comparisons of Disk-based Graph Indexing Techniques," In *VLDB* 2010. (invited to the VLDB Journal as **best of VLDB 2010 papers**)

•Han, W., Duc, P., Lee, J., Kasperovics, R., and Yu, J., "iGraph in Action: Performance Analysis of Disk-Based Graph Indexing Techniques," In *SIGMOD* 2011.

iGraph v2.0 + Turbo$_{ISO}$

•Lee, J., Han, W., Kasperovics, R., and Lee, J., "An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases," In *VLDB* 2013.

•Han, W., Lee, J., and Lee, J., "Turbo$_{ISO}$: Towards Ultra-Fast Subgraph Isomorphism Search in Graph Databases," In SIGMOD 2013.

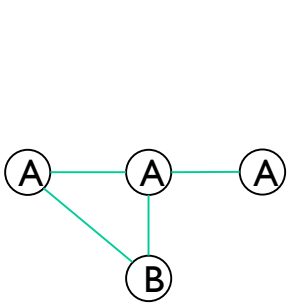
TurboGraph

•Han, W. et al., "TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC," In KDD 2013. (oral presentation)

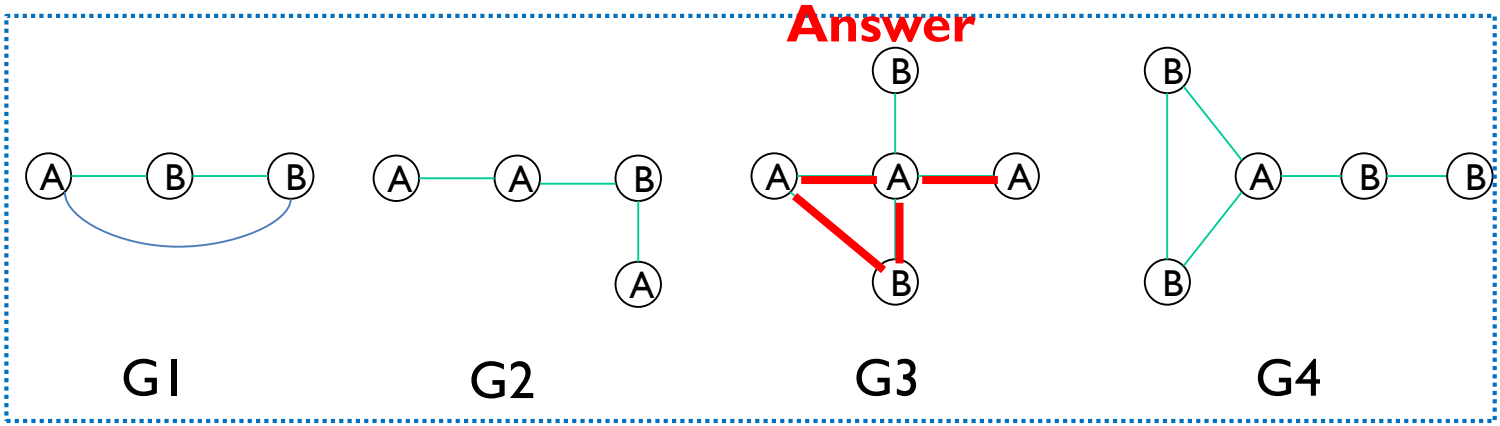
•Kim, J., Han, W., et al., "OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs," In SIGMOD 2014

Subgraph Isomorphism (Subgraph Matching)

- One of the most important graph queries
- Find all subgraphs that match a query graph

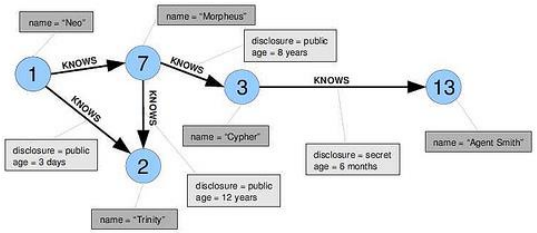


Query graph **Q**

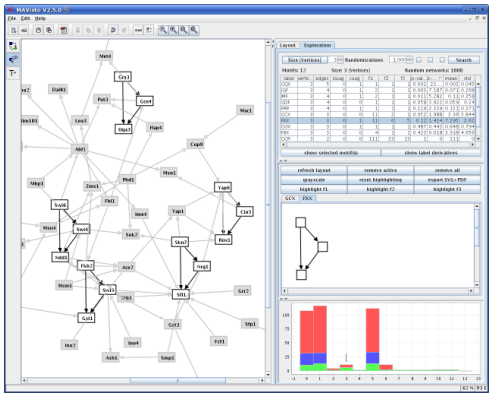


Graph database **D**

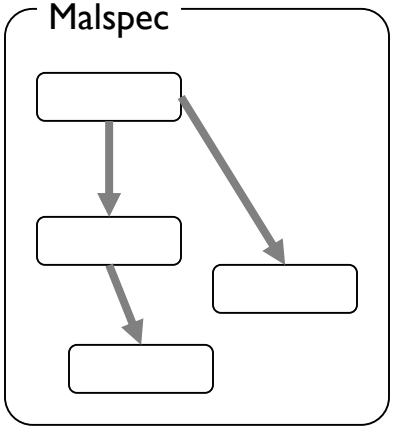
Many Important Applications



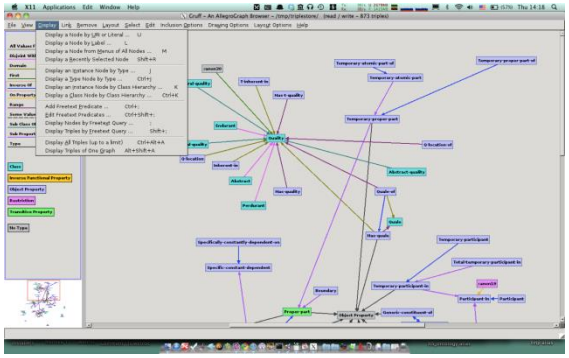
Social network search



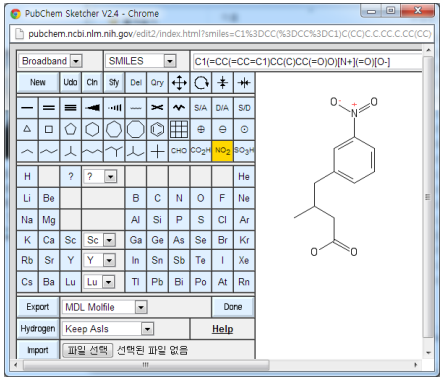
Motif search



Malware detection



RDF query processing



Chemical compound search

Problems in Existing Indexing Methods: Motivation for iGraph

- Serious problems in existing experiments
  - Compared indexes are implemented in different code bases
- Elapsed times reported can vary depending on implementation skills
- Number of disk I/Os not used.
- Small database ($\leq$ 20 Mbytes)
  - All files are cached in the OS file system cache

iGraph v1.0 [VLDB10, SIGMOD11]

- First common framework for disk-based graph indexes
- Supports both mining-based and non-mining based indexes
- Selected as a *best track paper* in VLDB 2010
- Open source: http://www.igraph.or.kr
  - Has been used in 26 countries

Universities using iGraph v1.0





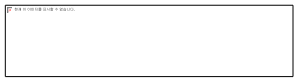












iGraph v2.0 [VLDB13]

- Focuses on the *subgraph isomorphism algorithm*
- The first generic framework that allows implementation of any subgraph isomorphism algorithm by extending this framework
- Provides in-depth analysis and comparison of the state-of-the art algorithms

$\text{Turbo}_{\text{ISO}}$ [SIGMOD13]

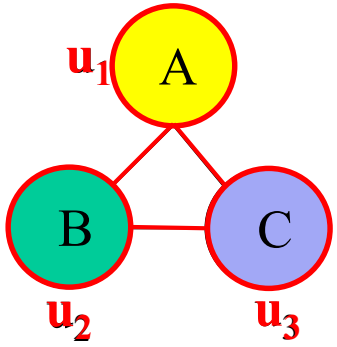
- A new subgraph isomorphism algorithm

Three things to remember

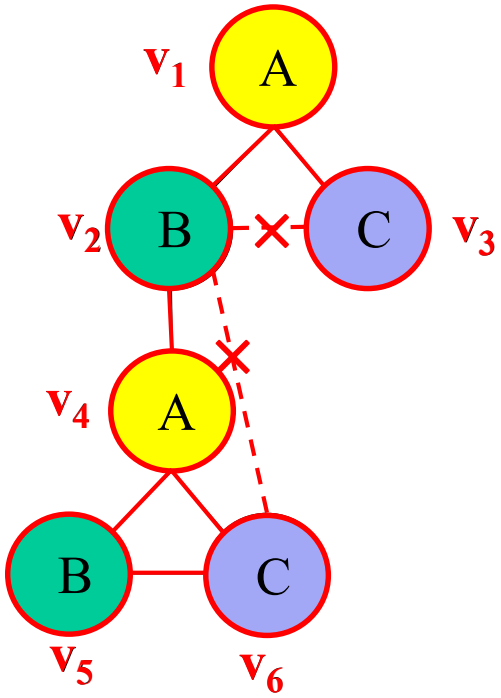
- Candidate region exploration
- Neighborhood equivalence class (NEC)
- Comb/Perm strategy

Review of Existing Subgraph Isomorphism Algorithm

- Backtracking algorithm
  - Find solutions by incrementing partial solutions or discarding them when they cannot be completed

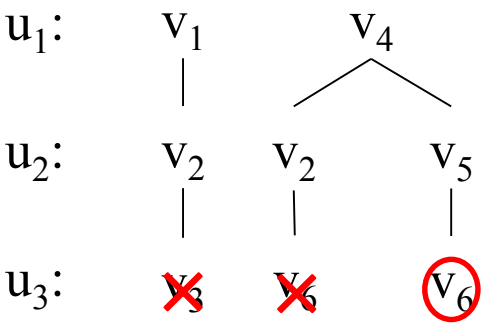


query graph: q



data graph: g

matching order: <u_1, u_2, u_3>



call tree *T*

Related Work

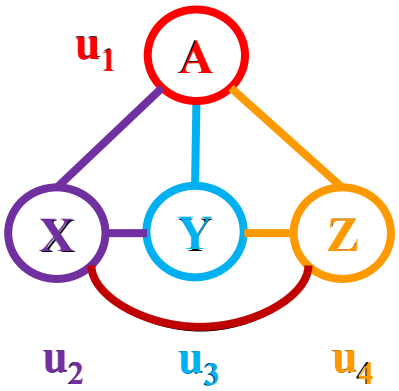
- Exact search
  - Non signature-based
    - Ullmann [JACM1976]

An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases [VLDB2013]

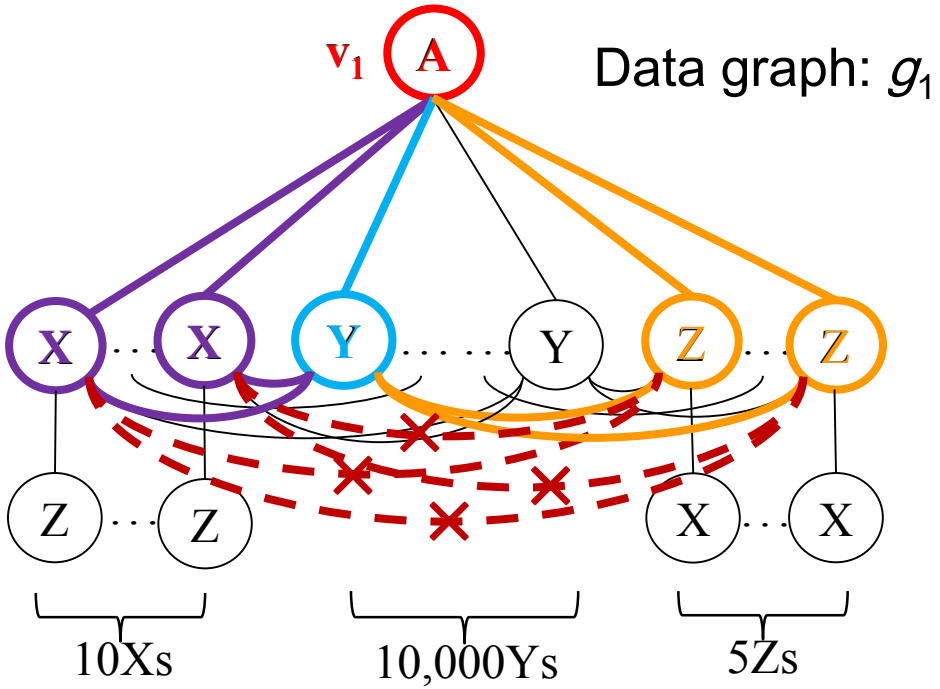
  - Signature-based
    - GraphQL [SIGMOD2008]
    - GADDI [EDBT2009]
    - Spath [VLDB2010]
- Similarity search
  - TALE [ICDE2008] NESS [SIGMOD2011], NEMA [VLDB2013], …

Importance of matching order

matching order O_1: <u_1, u_3, u_2 , u_4 >



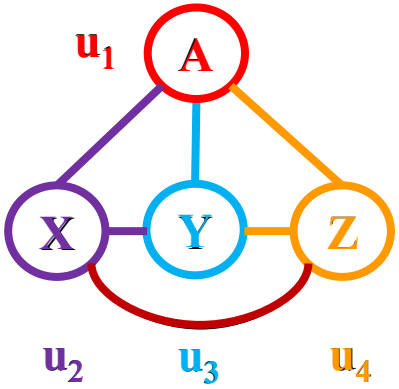
query graph: q



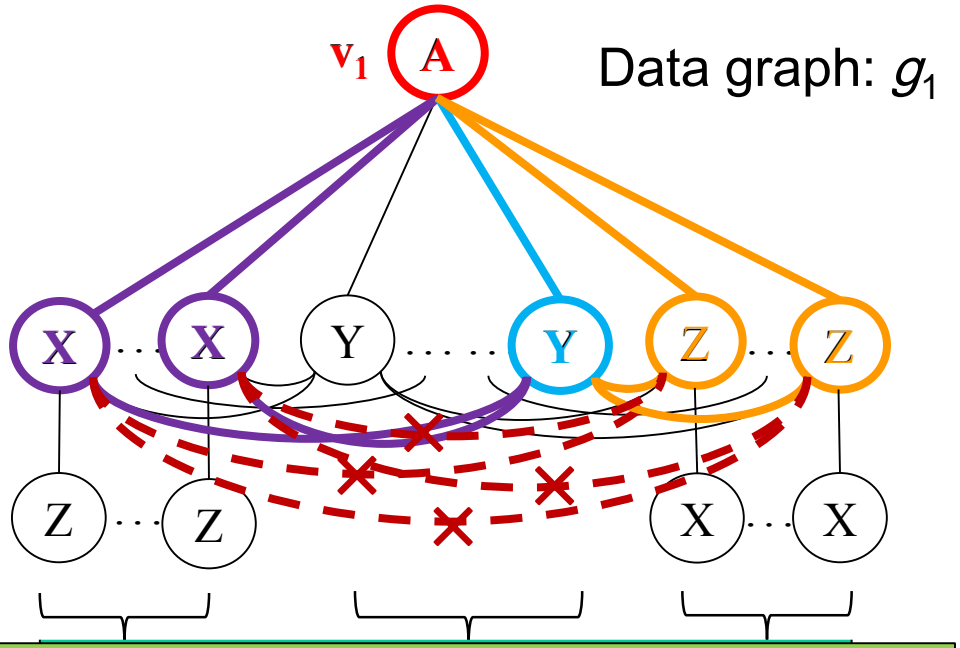
Data graph: g_1

Importance of matching order

matching order O_1: <u_1, u_3, u_2 , u_4 >



query graph: q



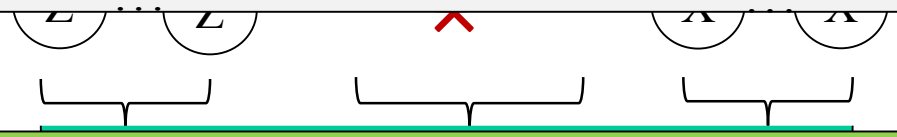
Data graph: g_1

Bad matching order O_1: needs 500,001 matches

Importance of matching order

matching order O_2: <u_1, u_4, u_2 , u_3 >

All subgraph isomorphism methods suffer from the notorious matching order problem!

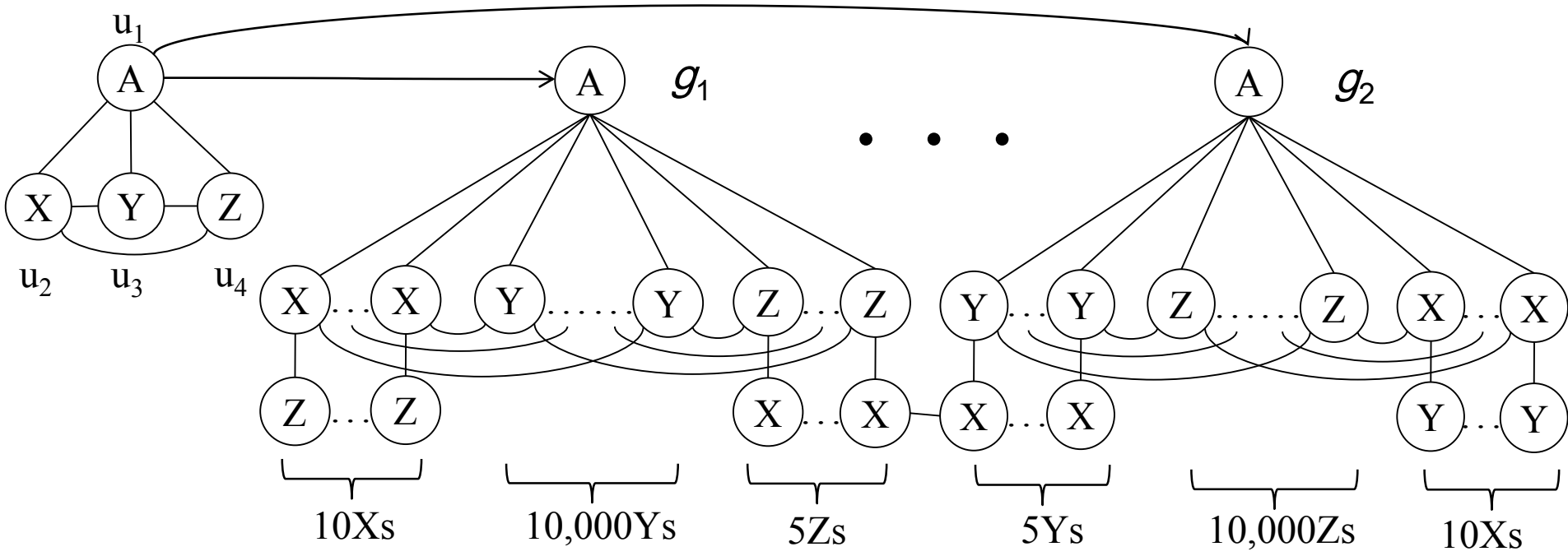


Good matching order O_2: needs only 51 matches

Motivation 1: One good matching order is not enough!

O_1(= <u_1, u_3, u_2, u_4, >): **500,001** matches for g_1 + **51** matches for g_2

O_2(= <u_1, u_4, u_2, u_3 >): **51** matches for g_1 + **500,001** matches for g_2



O_1: 500,001 matches

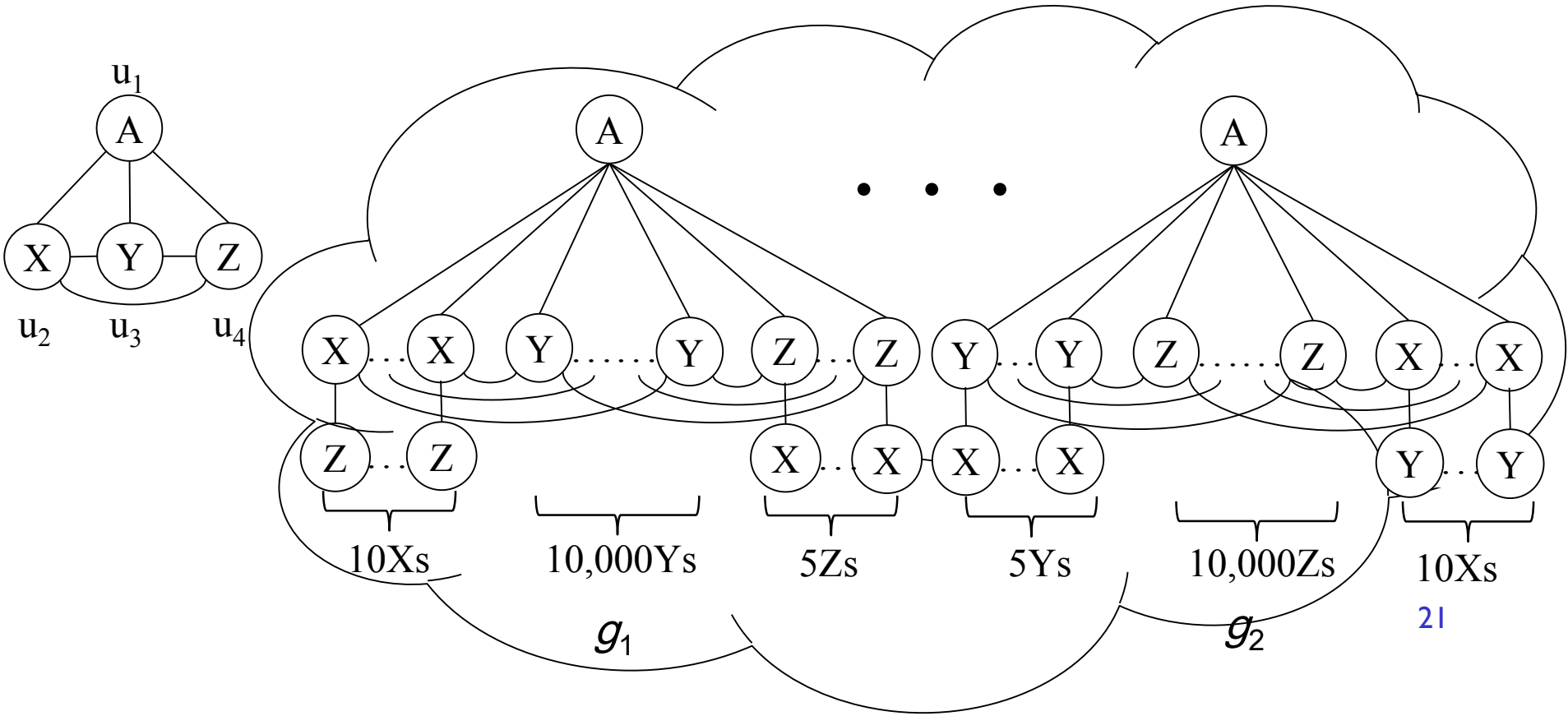
O_2: 51 matches

O_1: 51 matches

O_2: 500,001 matches

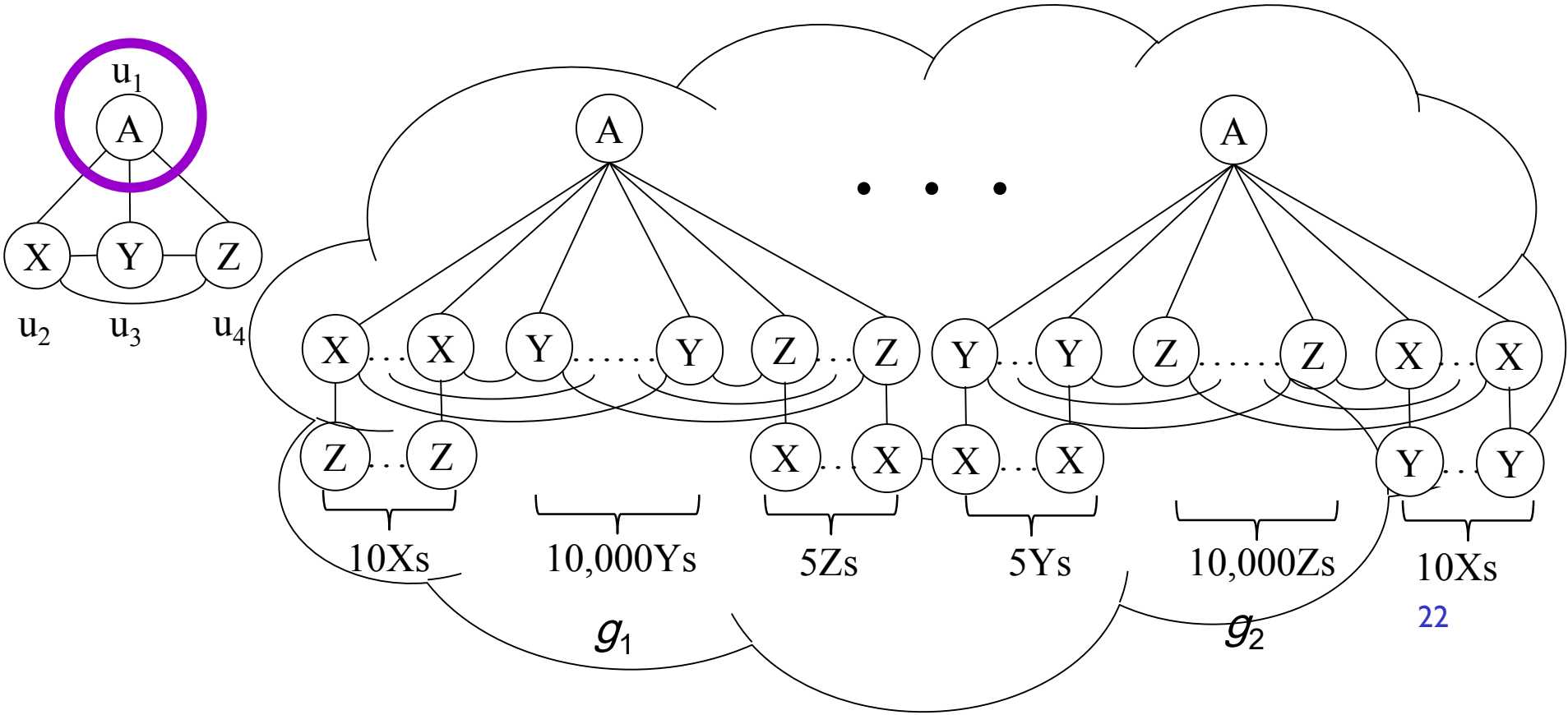
Key Idea 1: Candidate Region Exploration

- Solves the *notorious matching order problem* in graph databases



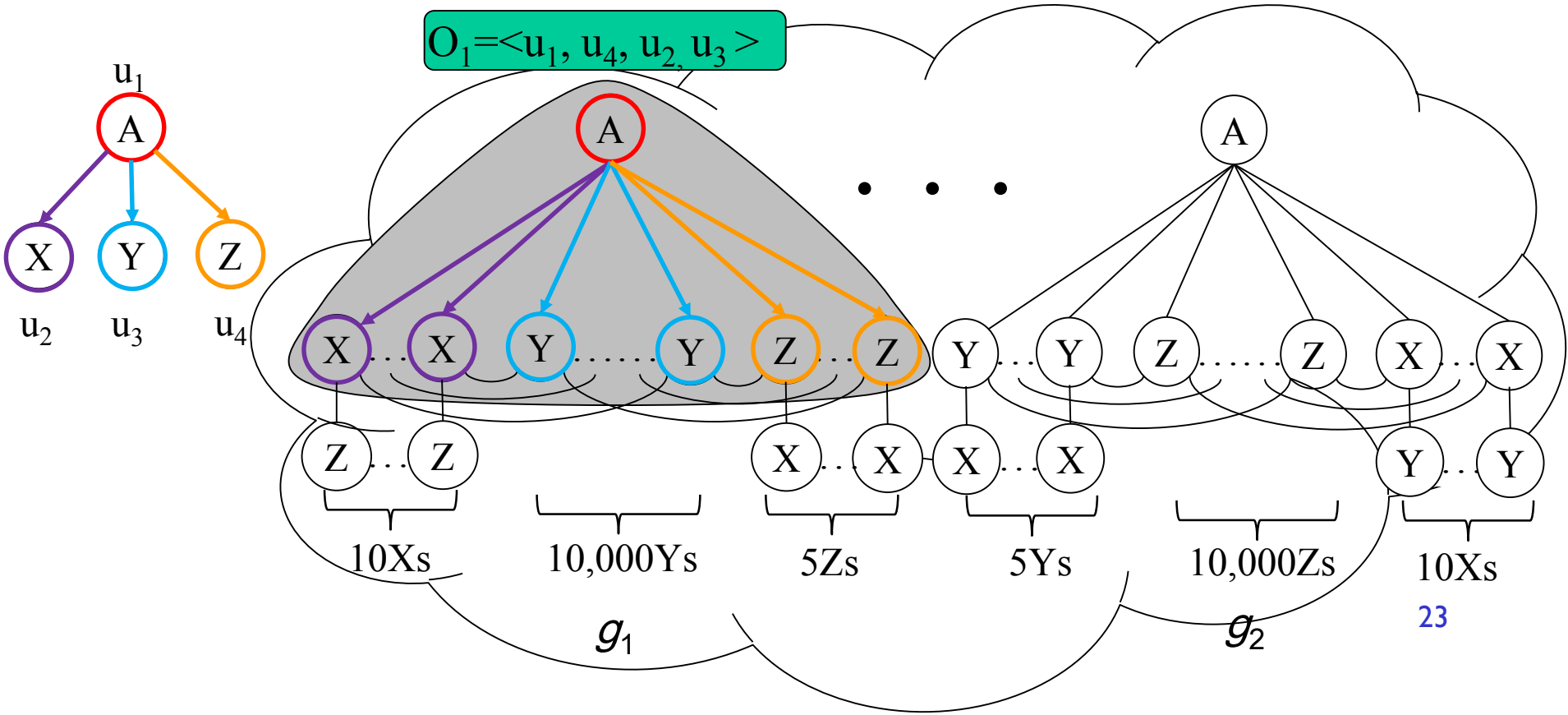
Key Idea 1: Candidate Region Exploration

- Solves the *notorious matching order problem* in graph databases



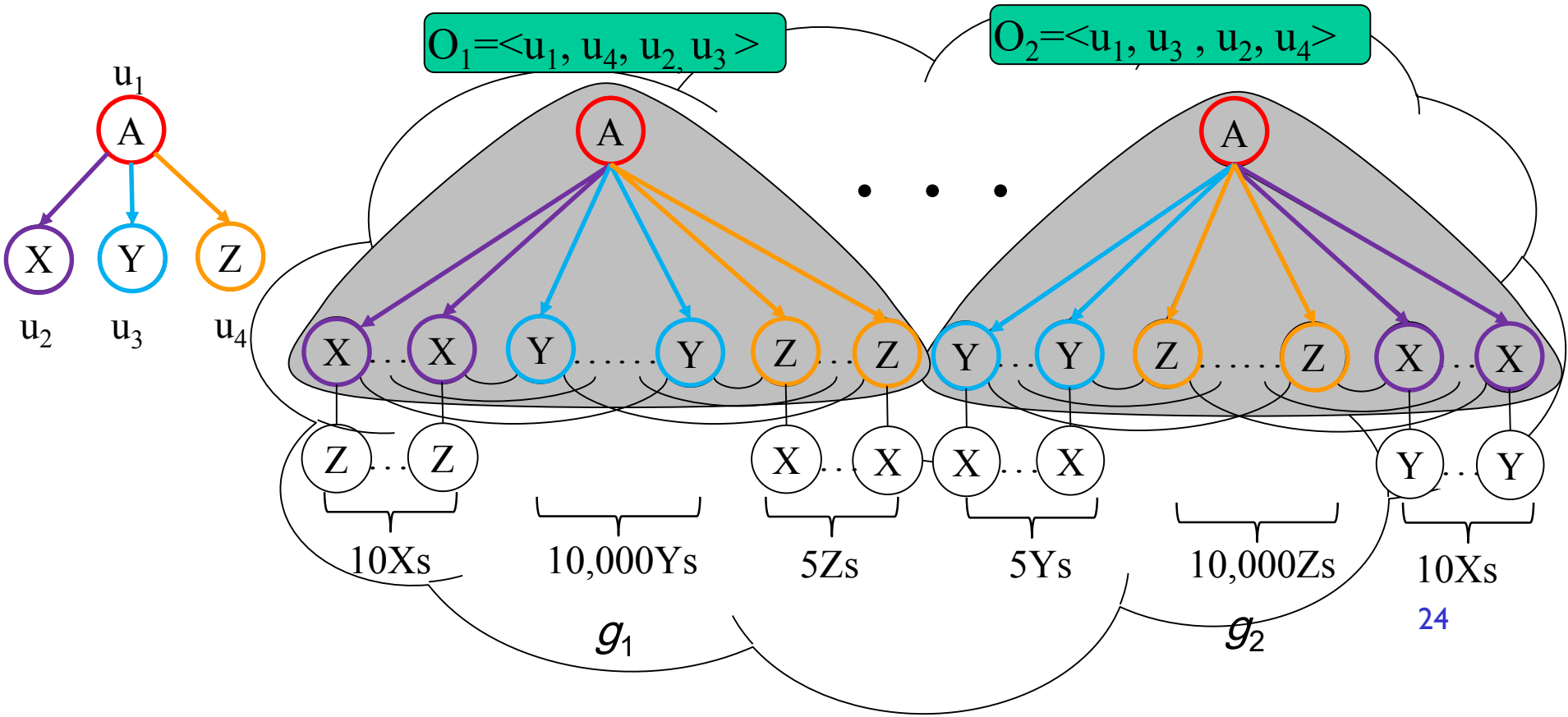
Key Idea 1: Candidate Region Exploration

- Solves the *notorious matching order problem* in graph databases

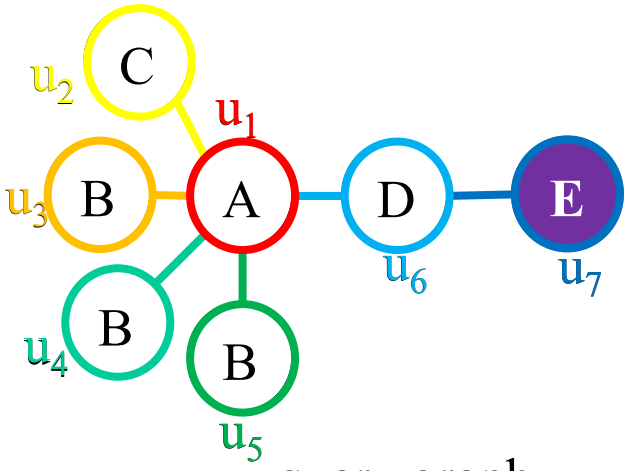


Key Idea 1: Candidate Region Exploration

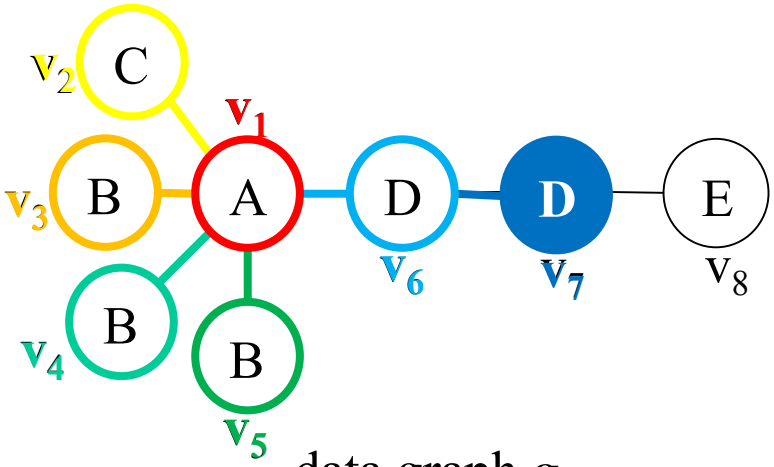
- Solves the *notorious matching order problem* in graph databases



Motivation 2: Useless Permutations

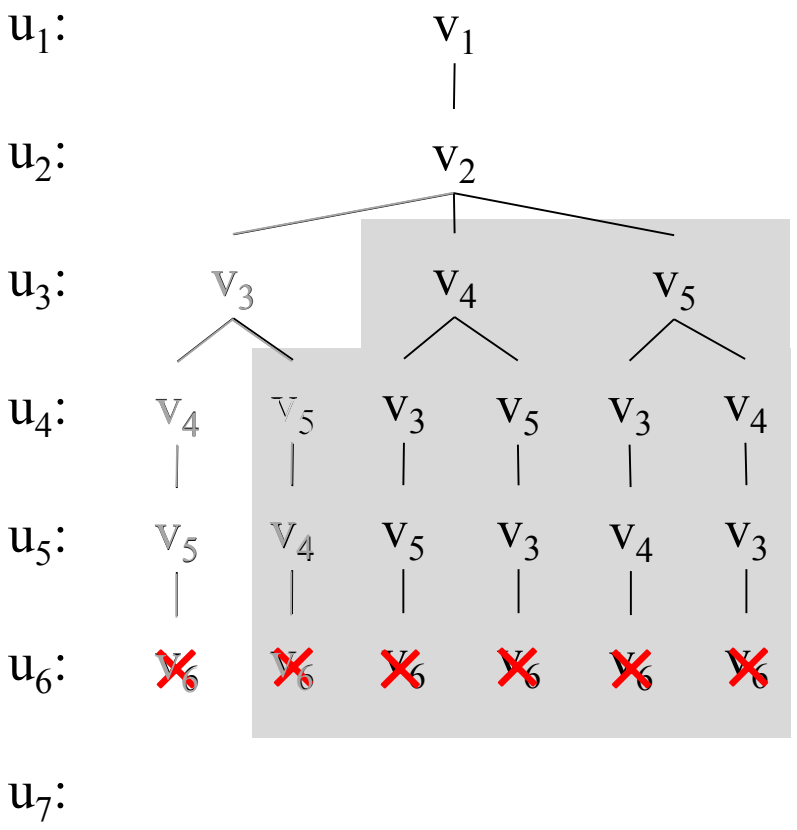


query graph *q*



data graph *g*

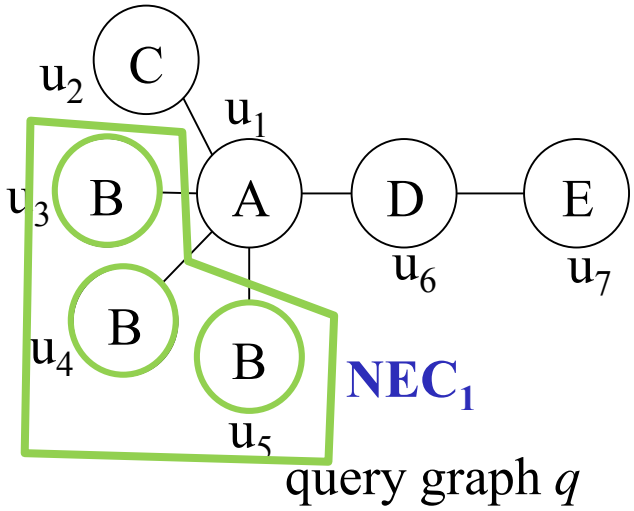
matching order: <u_1, u_2, ..., u_7>



call tree *T*

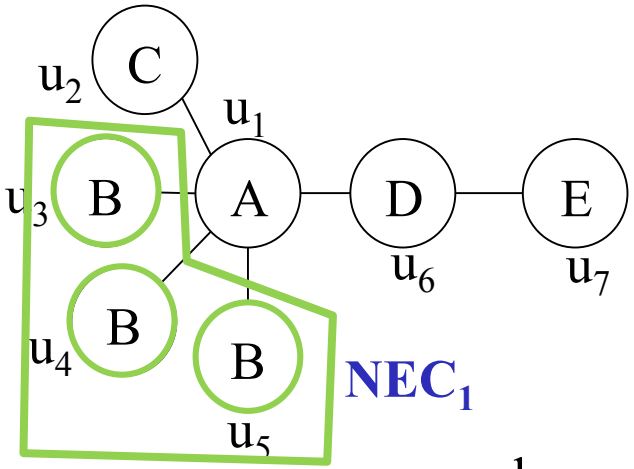
Neighborhood Equivalence Class (NEC)

- Each query vertex in the same NEC has
  - the same label
  - the same *adjacent* query vertices

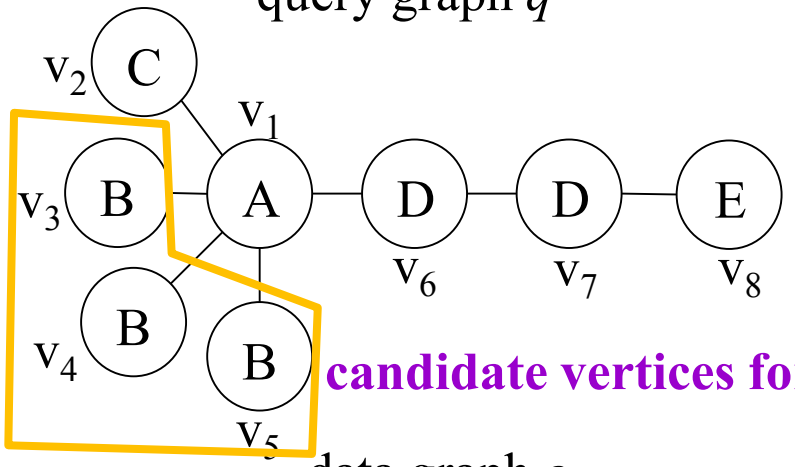


Key Idea 2: Comb/Perm strategy with NEC

- Avoid useless permutations

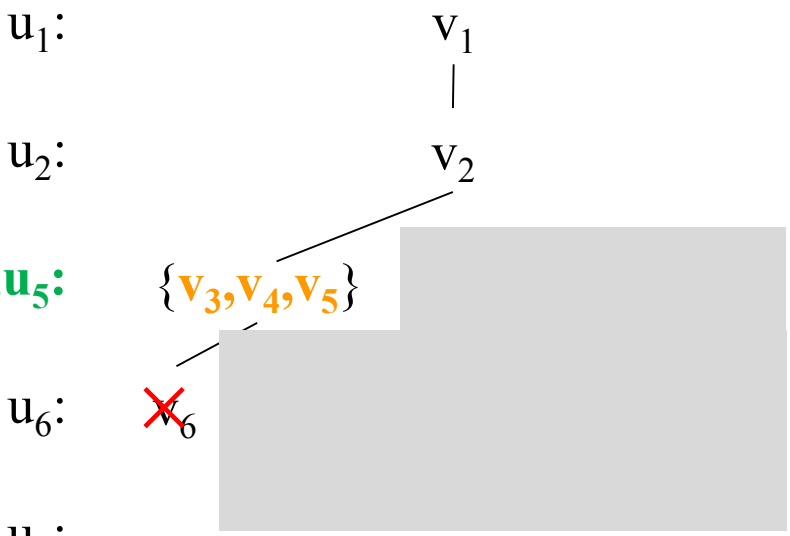


query graph *q*



data graph *g*

matching order: <u_1, u_2, {u_3,u_4,u_5},u_6, u_7>

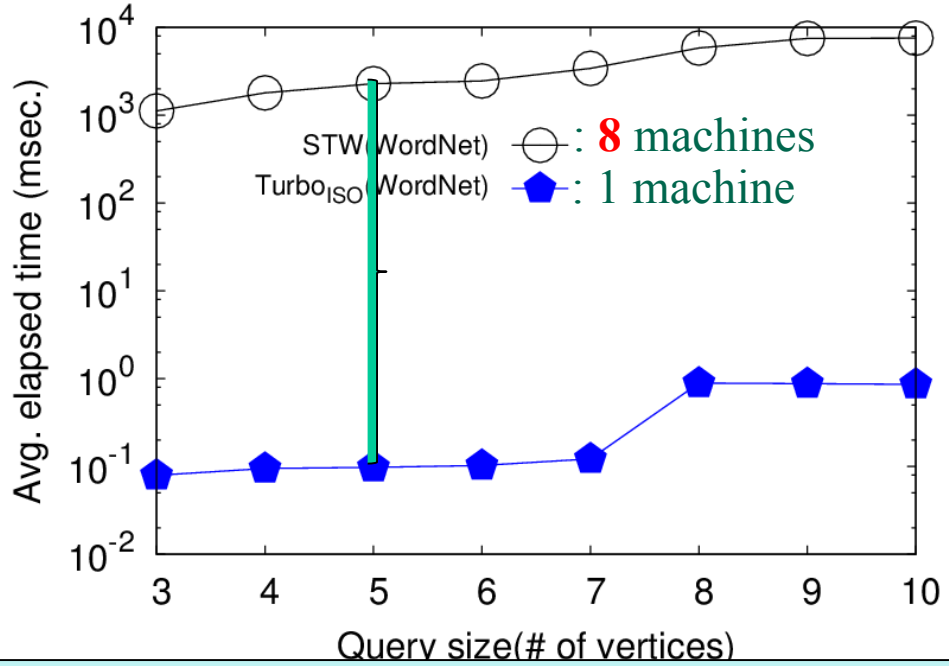


Contributions of Turbo$_{ISO}$

- Up to four orders of magnitude performance improvement over the state of the art method*
- Candidate region exploration
  - Provides good and robust matching order
  - Completely solves the notorious matching order problem
- Neighborhood equivalence class and Comb/Perm strategy
  - Avoid useless permutations

Comparison with STW

- Comparison with STW*
- Graph: WordNet



28,096 times faster than the state of the art!

*Su... DB 2012. De...

Best Characteristics of Turbo$_{ISO}$

- Ultrafast and Robust performance
  - Online analytics is possible!
- Parameter-free
- Little index maintenance cost
  - Supports very large *evolving* graphs!
- Easily parallelizable

TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC [KDD13]

Motivation

Distributed System approach

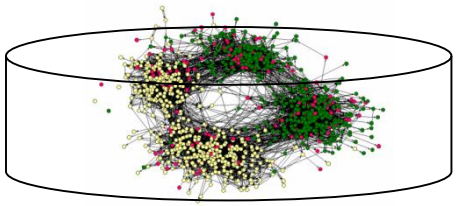
Gbase [KDD'11,VLDBJ'12]

GraphLab [VLDB'12]

Pregel [SIGMOD'10]

Single machine approach

GraphChi [OSDI'12]



DBMS approach

VERY SLOW for mining???

Can we exploit nice concepts in DBMSs **without losing performance?**

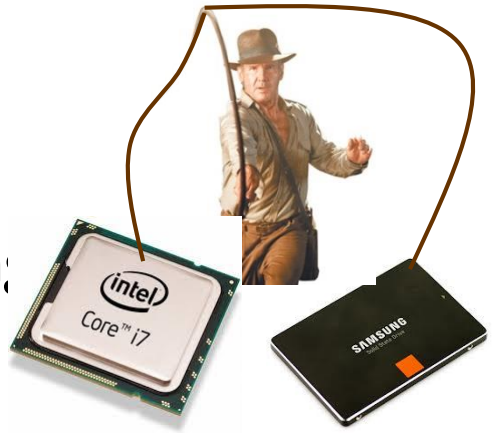
Comparison with other engines

Query	Input graph	Vertices/Edges	Representative perfomance result	GraphChi [3] on a single PC	TurboGraph on a single PC
Pagerank (5 iterations)	Twitter	41M/1.46B	Spark[1](50 machines), 8.1 min	19.35 min	2.74 min
Pagerank (1 iteration)	Twitter	41M/1.46B	Gbase[2](100 machines), 13.5 min	6.42 min	0.62 min
Pagerank (1 iteration)	YahooWeb	1.41B/6.63B	Gbase[2](100 machines), 13.0 min	20.90 min	3.71 min

[1] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," KDD 2012.
[2] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos, "GBASE: An Efficient Analysis Platform for Large Graphs," VLDB Journal, 2012.
[3] A. Kyrola, G. Blelloch, C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," OSDI 2012.

Why is TurboGraph Ultra-fast?

- **Fast Graph Storage Engine**
- **Full parallelism**
  - Multi-core parallelism
  - Flash SSD IO parallelism
    - Reading 400~500 Mbytes/sec from commodity SSDs
    - 97K IOPS (High-performance Random Read)
- **Full overlap**
  - CPU processing and I/O processing
  - I/O latency can be hidden!



Three things to remember

- Efficient disk/in-memory graph storage
- *Pin-and-slide* model
- Handling general vectors (see the paper)

Challenges for Graph Storage

- Adjacency list vs. adjacency matrix
- Two types of graph operations in disk-based graphs
  - Graph traversal (unique in graphs)
  - Bitmap operations during computation

Disk-based representation in TurboGraph

- Slotted page of 1 Mbyte size
  - Page contains records corresponding to adjacency lists
  - RID consists of a page ID and a slot number
- Vertex IDs or RIDs in adjacency list
  - Vertex ID approach
    - Good for bitmap operation
    - Bad for graph traversal
      - requires *a potentially LARGE mapping table!*
  - RID approach
    - Good for graph traversal
    - *Seems to be bad* for bitmap operation??

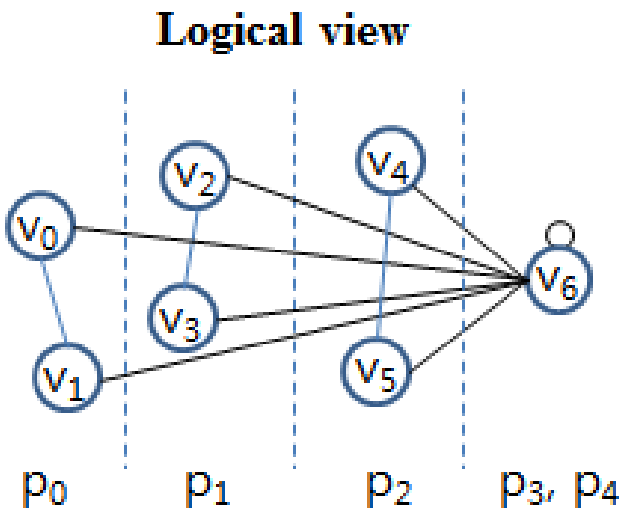
RID (mapping) table

- Each entry corresponds to a page (not a single RID)
  - Size is very small
- Each entry stores the starting vertex ID in the page
- Translation of RID (pageID, slotNo) to vertex ID
  - RIDTable[pageID].startVertex + slotNo
  - Can be **done in O(1)**

In-memory Data structures

- Buffer pool
- Mapping table from page ID to frame ID
  - Hash-table based mapping incurs significant performance overhead for graph traversal!
  - TurboGraph uses a page table approach!
- A data structure for handling large adjacency list (see paper)

Example



Memory

buffer pool

p2	p1	
0	1	2

page table

p0	-1
p1	1
p2	0
p3	-1
p4	-1

RID Table

v0	-1	-1
v2	-1	-1
v4	-1	-1
v6	0	2
v6	-1	-1

LOPL

0	p3
1	p4

Disk

slotted page list

p0			p1			p2			p3			p4	
2	r1	r6	2	r3	r6	2	r5	r6	7	r0	r1	r5	r6
2	r0	r6	2	r2	r6	2	r4	r6	r2	r3	r4		
	v1	v0		v3	v2		v5	v4		v6			v6

Core operations in buffer pool

- PINPAGE(pid)/UNPINPAGE(pid)
  - support large adjacency lists
- PINCOMPUTEUNPIN(pid, RIDList, uo)
  - Prepins an available frame
  - Issues an asynchronous I/O request to the FlashSSD
  - On completion of the I/O, a callback thread processes the vertices in the RIDList by invoking the user-defined function uo.Compute
  - After processing all vertices in RIDList, unpin the page

Supported Query Power: Matrix-vector multiplication

- G = (V,E), X (column vector)
- $M(G)_i$: *i*-th column vector of G
- Column view: $Y = \sum_{i=1}^{|V|} M(G)_i \times X_i$
  - Applications can define their own multiplication and summation semantics (the user-defined function Compute can generalize both)
  - $M(G)_i$ is represented as the adjacency list of v_i
- We can restrict the computation to just a subset of vertices $v_{I[1]} \sim v_{I[k]}$

Column-view of matrix-vector multiplication in TurboGraph

Algorithm 1 Matrix-Vector-Multiplication($G = (V, E)$, X, I, Y)

1: **for** $i = 1$ to $|I|$ **do**
2: $\quad Compute(v_{I[i]}.adj, X_{I[i]}, Y)$
3: **end for**

Pin-and-Slide Model

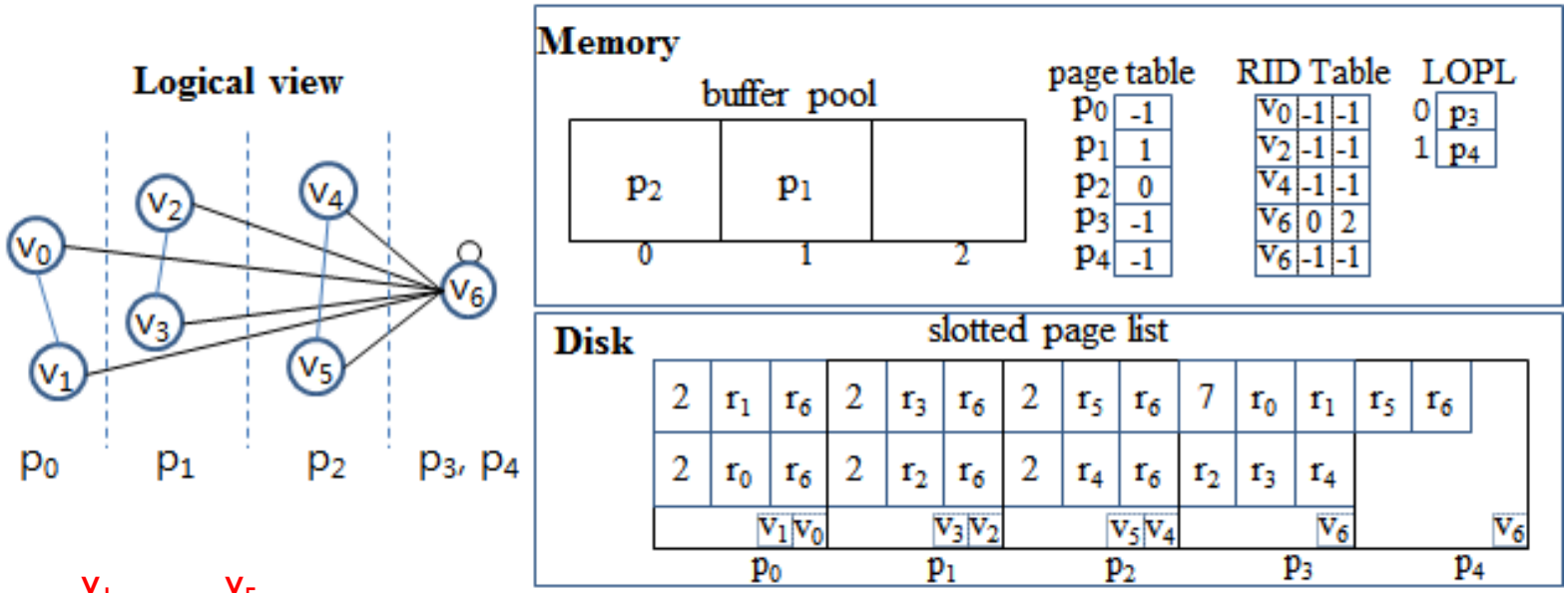
- New *computing model* for *efficiently* processing the generalized *matrix-vector multiplication* in the *column view*

- Utilizing execution thread pool and callback thread pool

Pin-and-Slide Model (cont'd)

- Given a set V of vertices of interest,
  - Identify the corresponding pages for V
  - ***Pin*** the pages in the buffer pool
  - Issue parallel asynchronous I/O requests for pages which are not in the buffer
  - Without waiting for the I/O completion, execution threads concurrently process vertices in V that are in the pages pinned
  - ***Slide*** the processing window one page at a time as soon as either an execution thread or a callback thread finishes the processing of a page

Example



$I = (0, \overset{v_1}{1}, 1, 1, 0, \overset{v_5}{1}, 1)$

1. identify pages (p_0, p_1, p_2, p_3, p_4) for I
2. Pin p_1 and p_2
3. Issue asynchronous I/O request for p_0
4. Execution threads process v_2, v_3, and v_5 concurrently
5. On completion of I/O request for p_0, callback threads process v_1
6. After processing any page, unpin the page and slide execution window
   i.e., process p_3 and finally process p_4

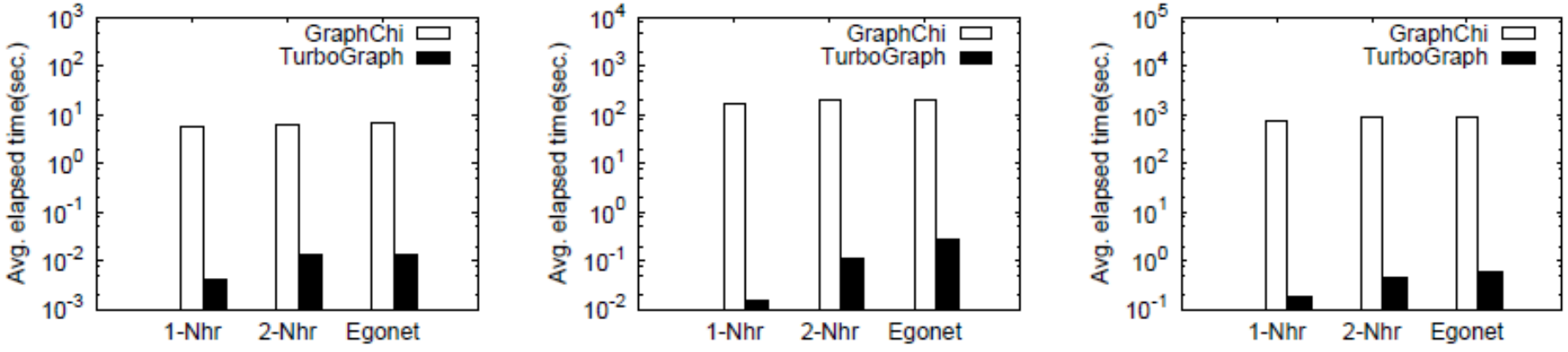
Processing Graph Queries

- We support graph queries based on matrix-vector multiplication
  - Targeted queries processing only part of a graph
  - Global queries processing the whole graph
- Targeted queries
  - BFS, K-step neighbors, Induced subgraph, K-step egonet, K-core, cross-edges etc.
- Global queries
  - PageRank, connected component

Experimental setup

- Datasets
  - LiveJournal (4.8M vertices), Twitter (42M vertices), YahooWeb (1.4B vertices)
- Intel i7 6-core PC with 12 GB RAM
- 512GB SSD (Samsung 840 series)
- Bypass OS cache to guarantee real I/Os
- Main competitor
  - GraphChi

Targeted Queries

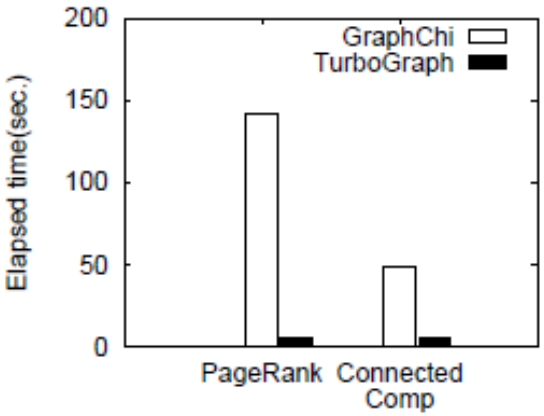


(a) LiveJournal. (b) Twitter. (c) YahooWeb.

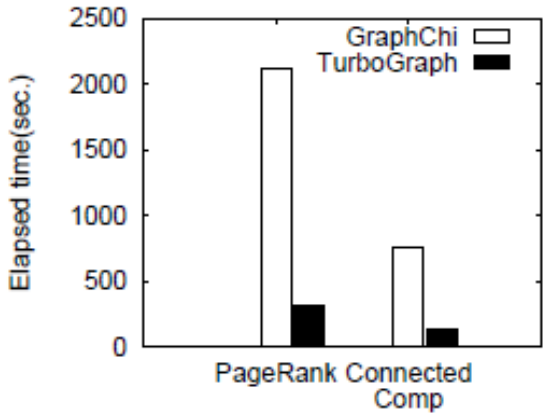
Figure 5: Average elapsed time (Targeted queries).

TurboGraph outperforms GraphChi by up to *four orders of magnitude*.

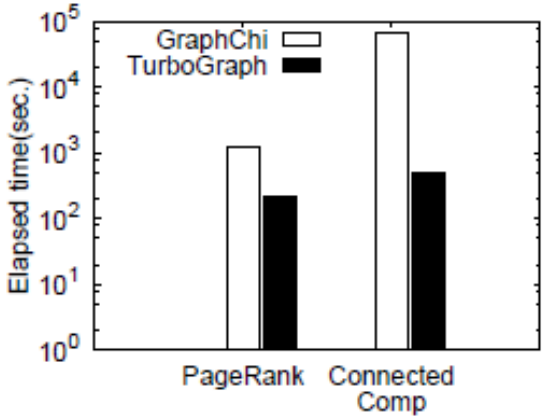
Global Queries



(a) LiveJournal.



(b) Twitter.



(c) YahooWeb.

TurboGraph outperforms GraphChi by up to

27.69 times for PageRank.

144.11 times for Connected Component[+].

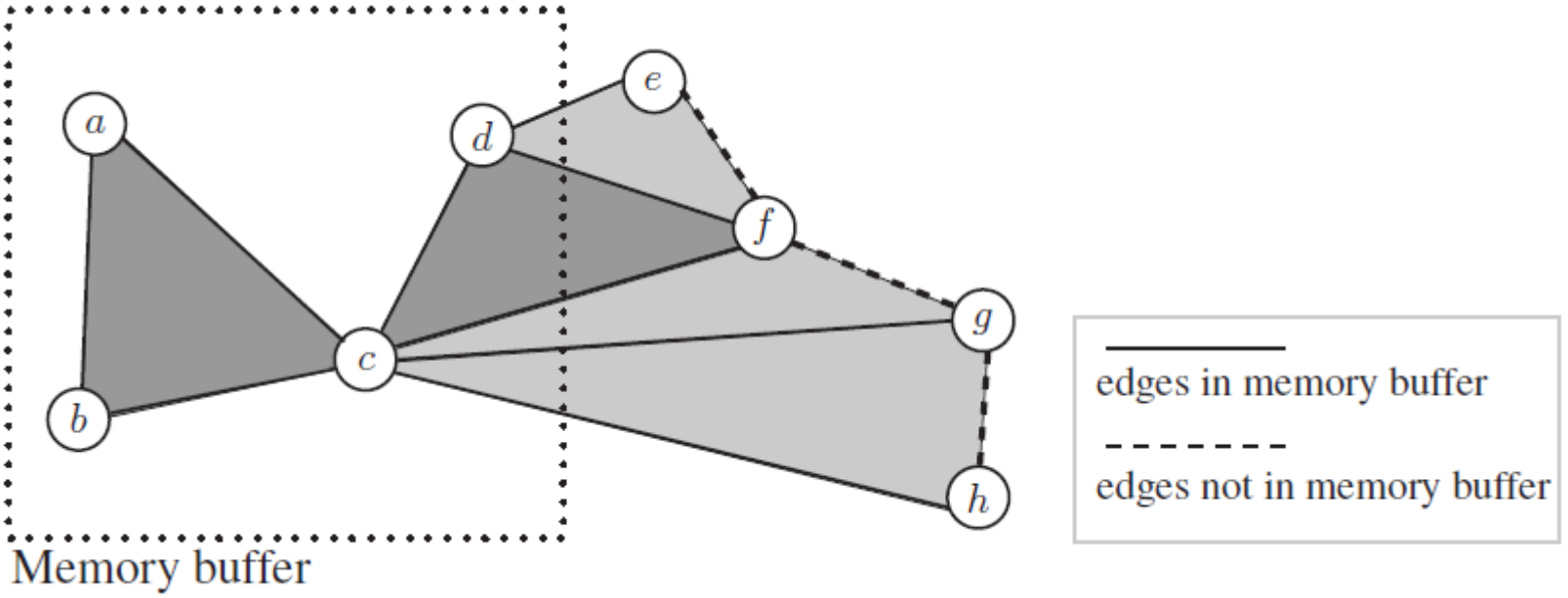
[+]*upcoming paper for details and much faster performance*

OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs [SIGMOD14]

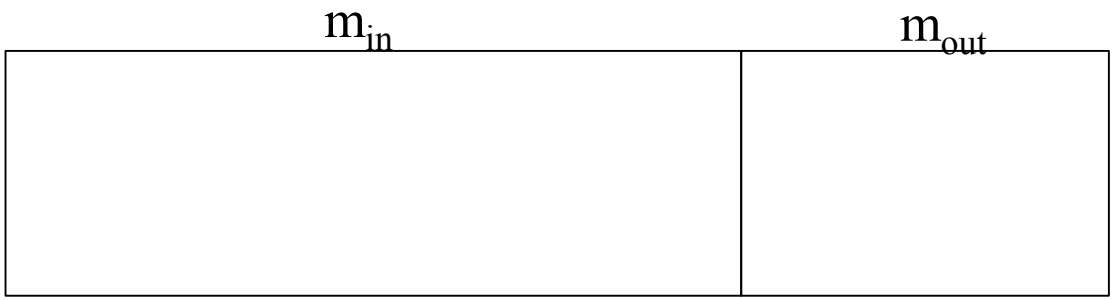
Highlights of OPT

- an overlapped and parallel disk-based triangulation framework for billion-scale graphs
- At the macro level, overlaps the internal triangulation and the external triangulation
- At the micro level, overlaps the CPU and I/O operations
- Achieves almost ideal performance

Internal triangles vs external triangles

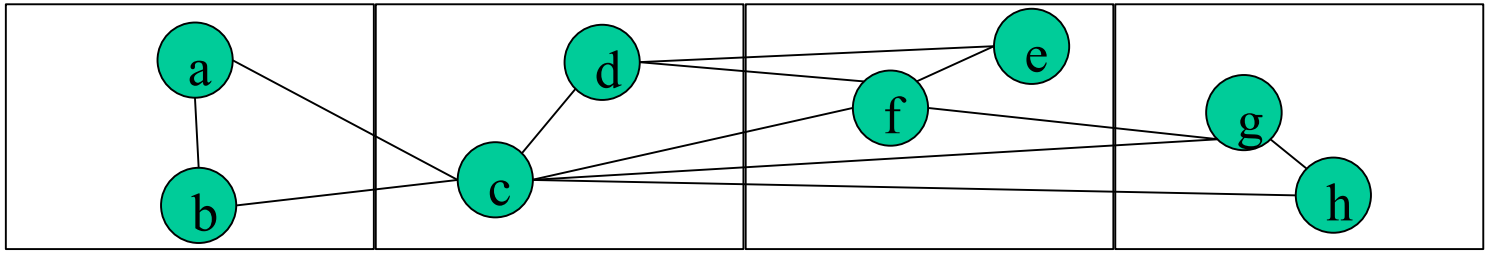


Running example



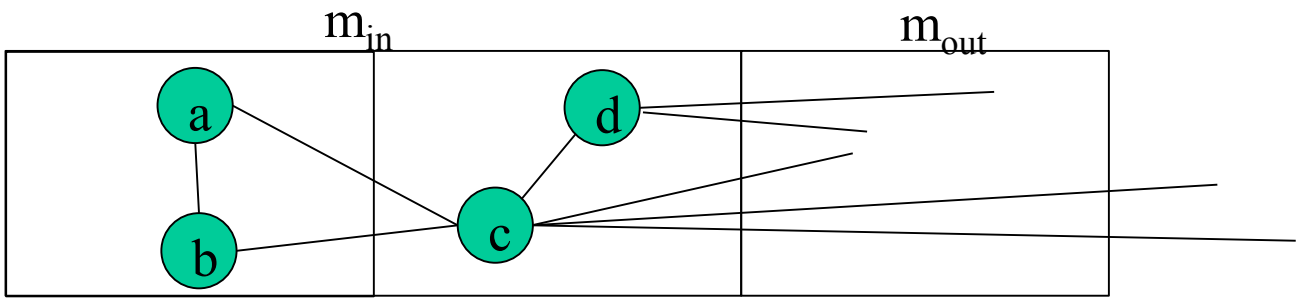
MEMORY

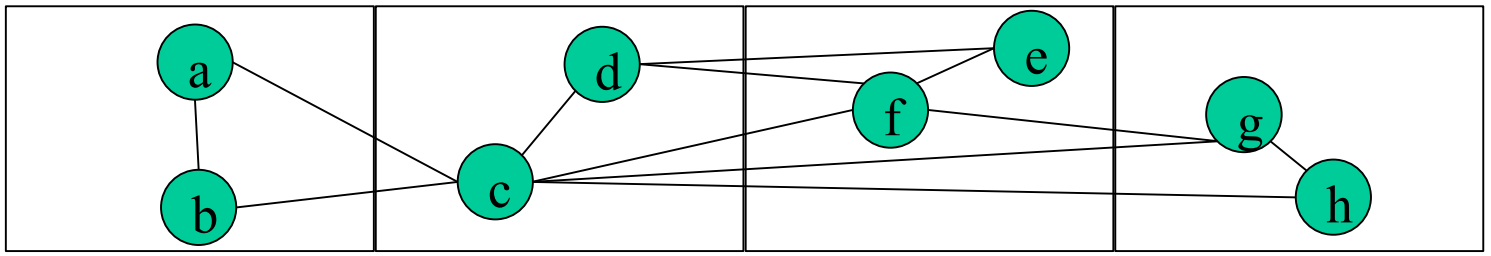
DISK



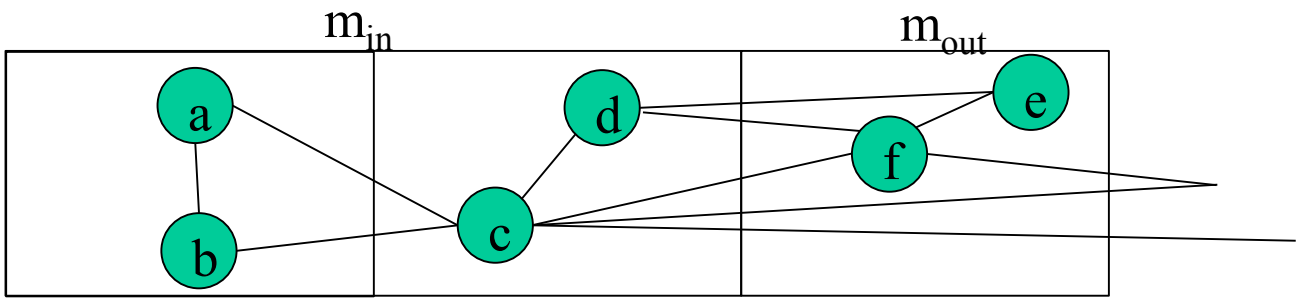
p_1 p_2 p_3 p_4

Load p_1 and p_2

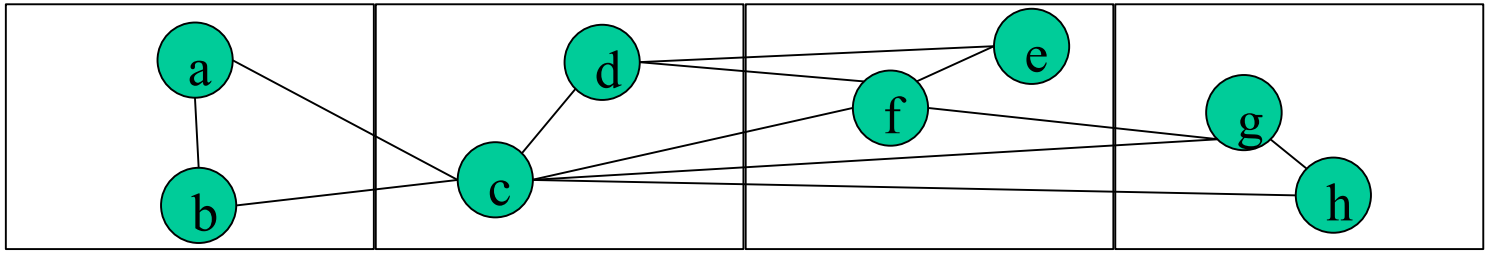




Load p_3



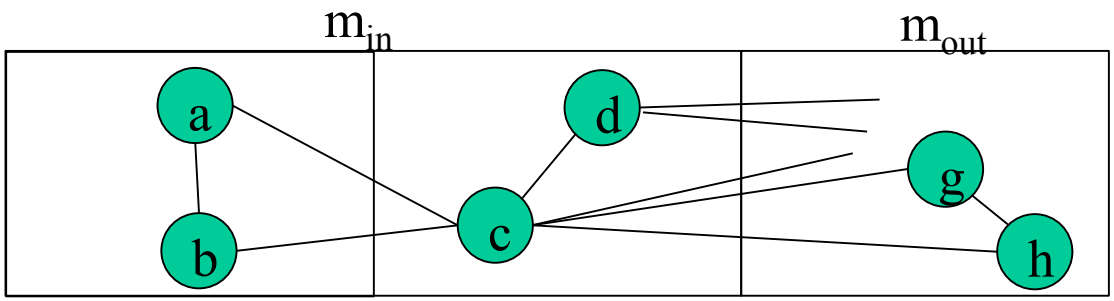
MEMORY



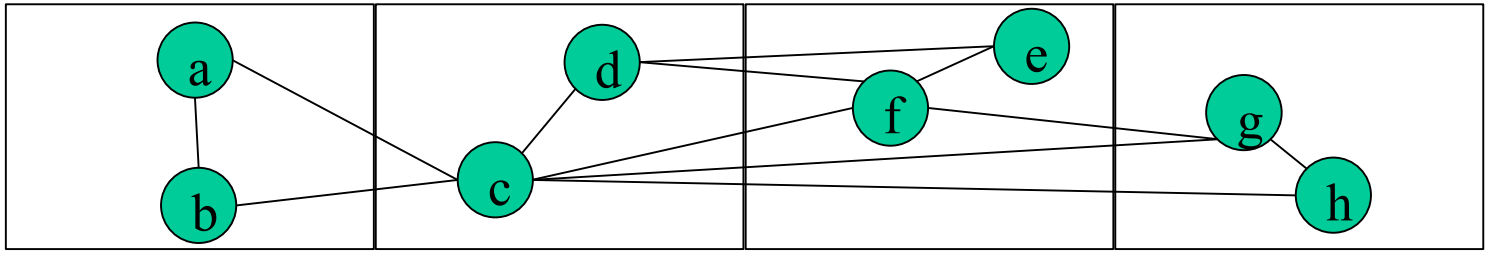
DISK

p_1 p_2 p_3 p_4

Load p_4



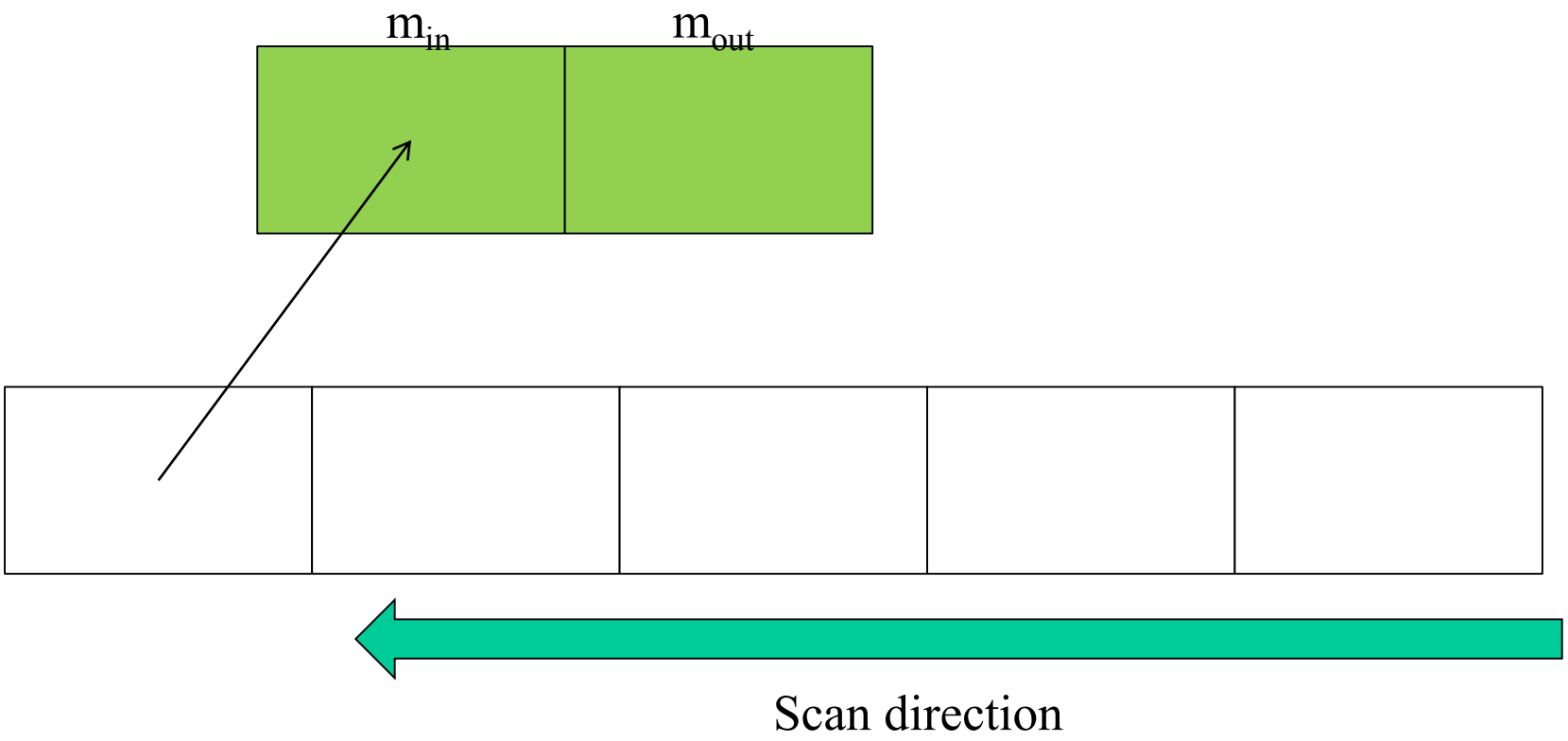
MEMORY



DISK

p_1 p_2 p_3 p_4

Effective disk access order of OPT



Comparison with GraphChi

	LJ	ORKUT	TWITTER	UK
OPT	6.39	18.51	469.40	480.918
GraphChi-Tri	85.87	196.95	1850.26	4046.77
GraphChi-Tri/OPT	13.44	10.64	3.94	8.41

For the Yahoo dataset, OPT outperforms GraphChi by 31 times!

Conclusions

- We have presented a series of graph processing frameworks for large-scale graphs
  - iGraph 1.0 for graph indexing
  - iGraph 2.0 + $\text{Turbo}_{\text{ISO}}$ for subgraph isomorphism
  - TurboGraph for graph analytics

Q/A

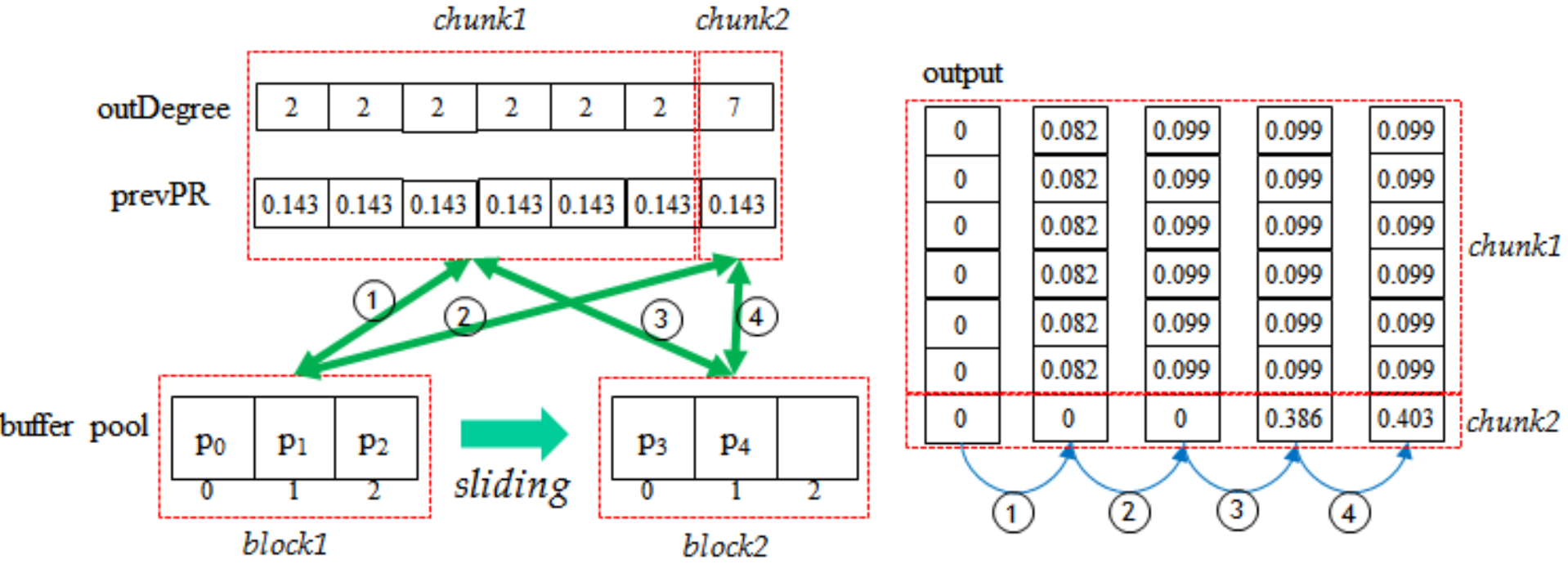
Handling general vectors

- Indicator vector can be implemented as a bitmap
- However, what if we want to use general vectors instead?
  - Consider PageRank where we need random accesses to pagerank values and out-degrees in two general vectors

Main idea of handling general vectors

- Adopt the concept of block-based nested loop join
- a general vector is partitioned into multiple **chunks** such that each chunk fits in memory
- Regard the pages pinned in the current buffer as a **block**
- **Join** a block with a chunk of each random vector in-memory until we consume all chunks
- Hide this mechanism as much as possible from users! (see paper)

Example



$$uo.Compute(v_i, Iterator(v_i.adj)) = 0.85\sum_{j \in v_i.adj} \frac{prevPR[j]}{outDegree[j]} + \frac{0.15}{|V|}$$