Data Systems Group

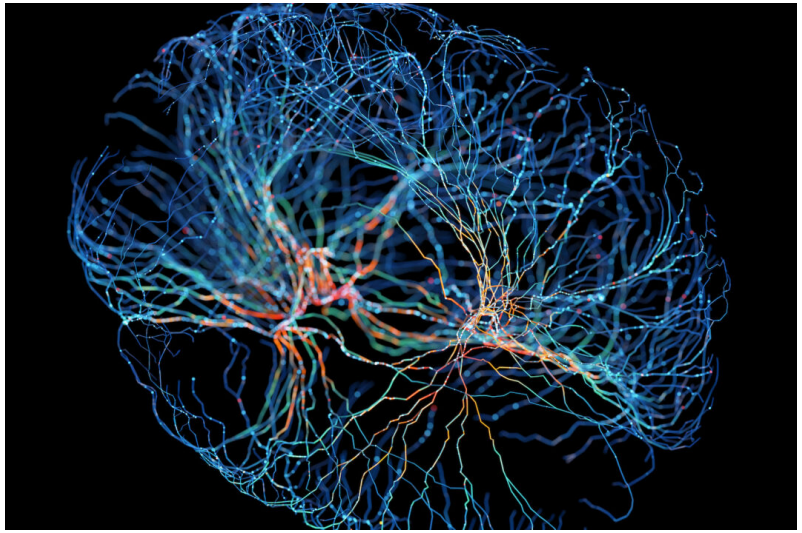
Efficient Graph Analytics on Real World Graphs

Joana M. F. da Trindade

jmf@csail.mit.edu

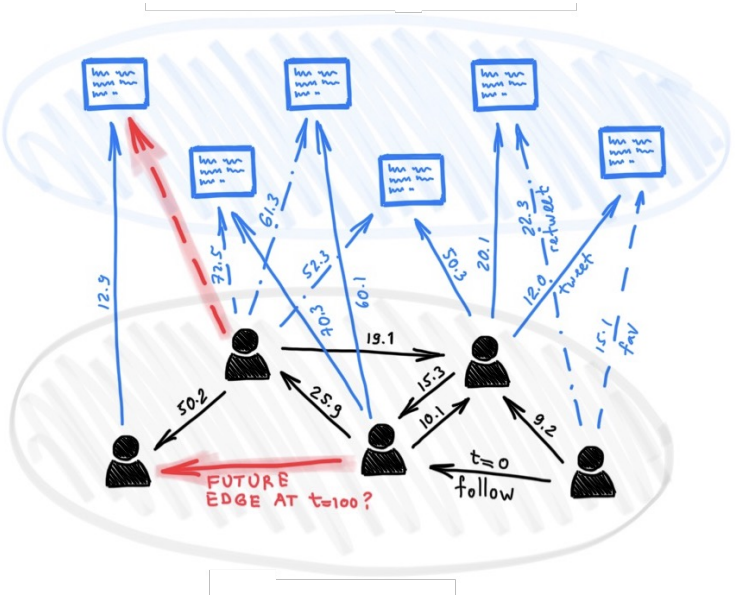
“Graphs are everywhere” they say

www.tripinfo.com/maps/ny-new-york-city



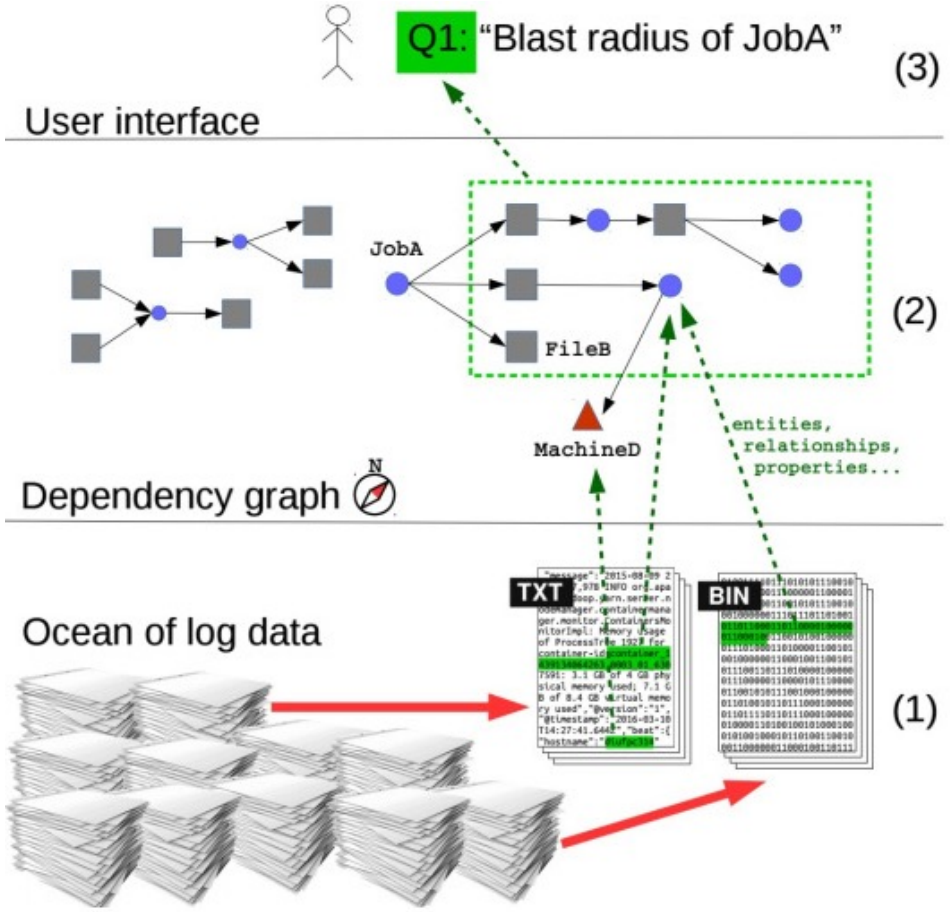
“Brain Network Mapping Study Challenges Basis for Psychiatric Distinctions” (genengnews.com)

e.g.,
- hyperlink 2012: 128B edges
- **MSFT data infra: 16B edges / week**



“Deep learning on dynamic graphs” (blog.twitter.com)

Cloud computing companies extract large graphs from production systems logs every day



Queries include:
- k-hop paths (dependency analysis)
- subgraph traversals
- temporal paths

Graphs are large and skewed:
- billions of edges every week
- both degree and temporal skew

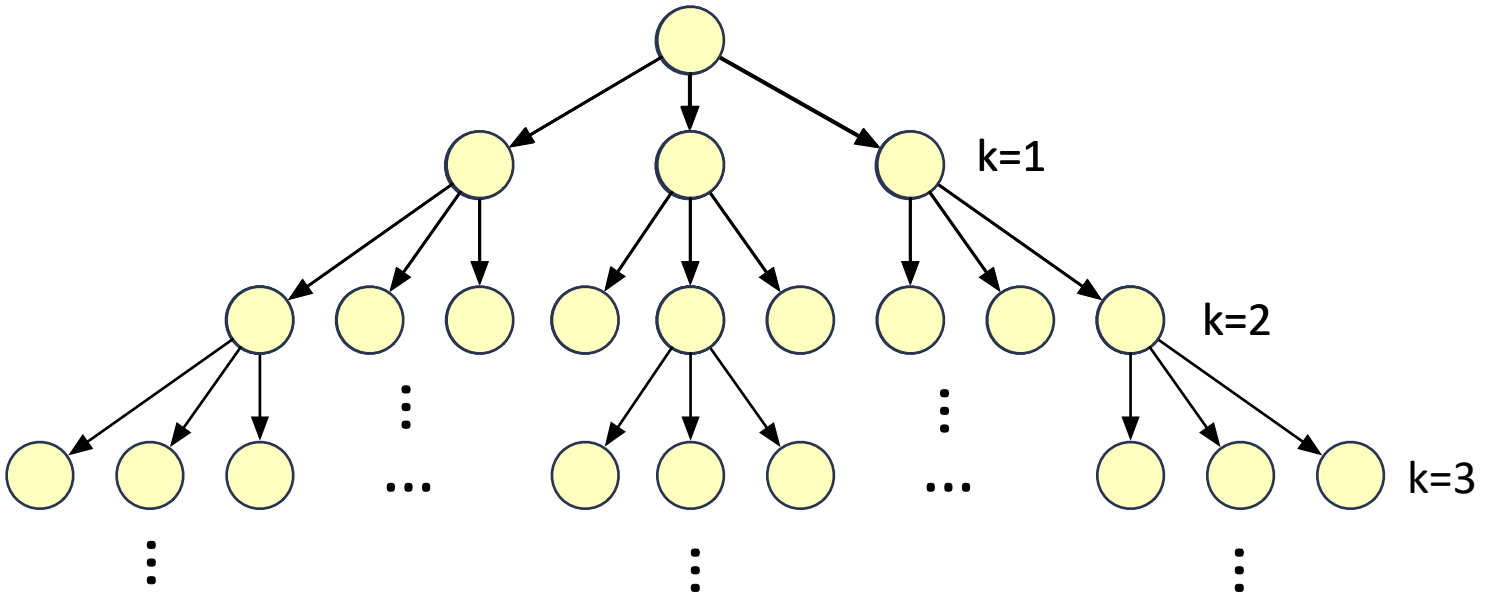
[1] R. Mavlyutov, C. Curino, B. Asipov, P. Cudre-Maroux, *Dependency-Driven Analytics: A Compass for Uncharted Data Oceans.* CIDR 2017

Problem: on large graphs, *k*-hop path queries can be expensive even for small values of *k*

Worst-case **complexity**

is proportional to:

$$\left(\max_{v \in V} deg(v) \right)^k$$



Weekly data lineage graph of Microsoft production cluster had 3.2 Billion vertices, 16.4 Billion edges, and on-disk footprint of 10+ Terabytes.

Problem: on large graphs, *k*-hop path queries can be expensive even for small values of *k*

Distributed relational engine at Microsoft took *hours* to answer it

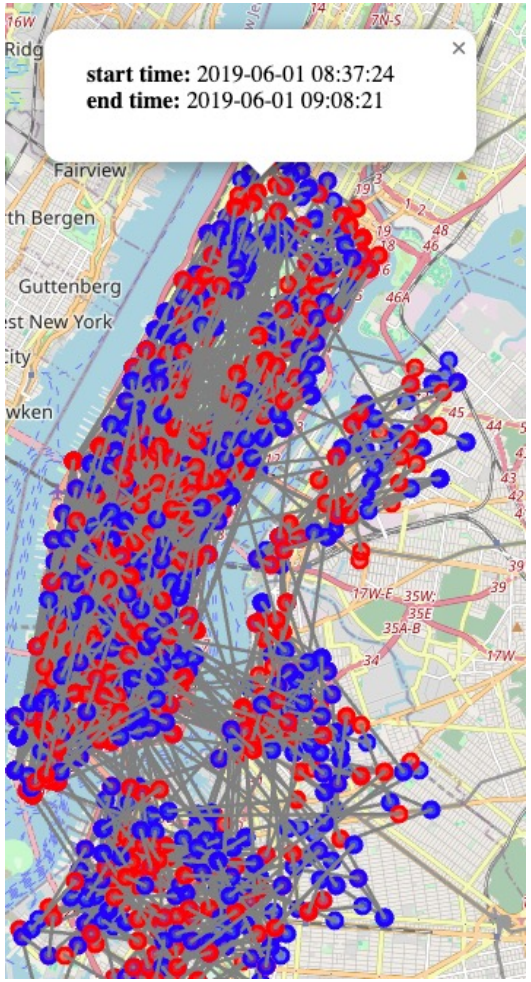
- Real-world graphs are skewed; **relational engines produce inefficient plans (recursive self-joins)**

Existing graph databases limited in performance

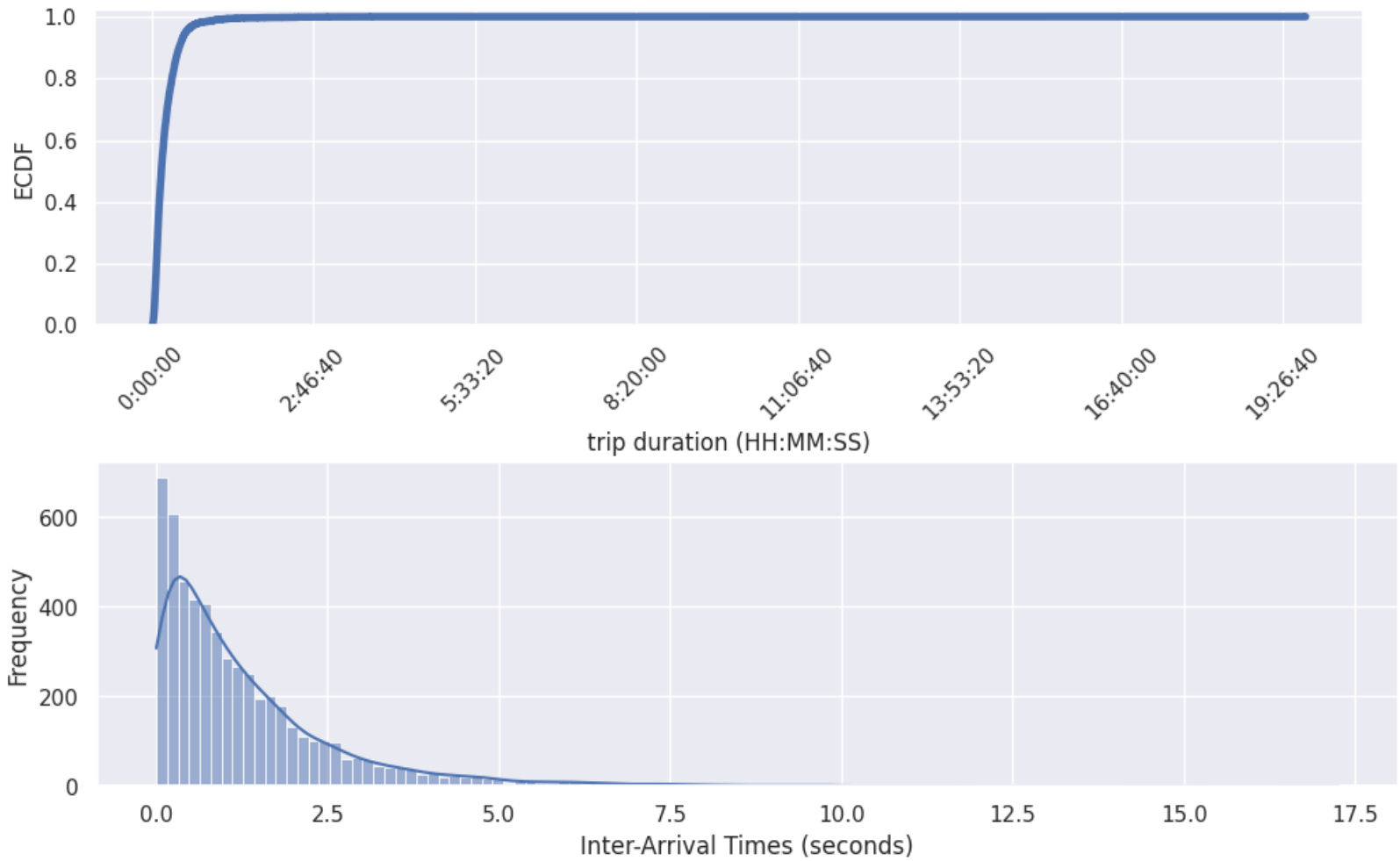
- Distributed engines (e.g., Pregel and Flink) incur lots of messaging overhead

To be used for operational decisions, these queries require short response times (seconds to mins)

Problem: one size does *not* fit all



Trip start time: [8am, 10am]



CitiBike NYC trips data (plots + viz at https://github.com/jmftrindade/citibike_analysis)

Problem: one size does *not* fit all

Real-world graph data is skewed in several dimensions

- **Degree skew**: caused by **preferential attachment** (“rich get richer”)
- **Temporal skew**: human generated event **durations and IAT** follow power-law

Graph query optimizers treat every vertex the same

- Assume i.i.d. and/or fail to capture temporal correlations

Temporal graph data offers crucial insights, yet remains unexplored

- How can we leverage temporal dynamics for query optimization?

Key Contributions of this Thesis

How can we speed up expensive graph queries (e.g., *k*-hop)?

- Introduce class of sparser graph views
- Infer materialized graph views that are beneficial for queries
- Rewrite queries in terms of those views

How can we leverage temporal dynamics for query optimization?

- Memory-efficient data representations for temporal graphs
- Selective Indexing: treat large vertices differently than others
- Leverage the above to implement efficient algorithms over temporal graphs

Thesis Outline

Kaskade: **Graph Views** for Efficient Graph Analytics

J M F da Trindade, K Karanasos, C Curino, S Madden, J Shun

ICDE 2020

Kairos: Efficient **Temporal Graph Analytics** on a **Single Machine**

J M F da Trindade, J Shun, S Madden, N Tatbul

Under submission to VLDB 2024

Kaskade: Key Technical Contributions

Graph views

- Sparser views such as *connectors* and *summarizers* (e.g., filters and aggregates)

Constraint-based view enumeration

- Mine constraints from graph's schema and data's structural properties
- Rewrite queries in terms of beneficial views

Cost model for graph views

- View size estimation leverages structural properties
- Only most beneficial views are materialized

Kaskade: Key Technical Contributions

Graph views

- Sparser views such as *connectors* and *summarizers* (e.g., filters and aggregates)

Constraint-based view enumeration

- Mine constraints from graph's schema and data's structural properties
- Rewrite queries in terms of beneficial views

~~Cost model for graph views~~

- ~~View size estimation leverages structural properties~~
- ~~Only most beneficial views are materialized~~

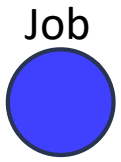
see thesis for details

What is the Graph Data Model in Databases?

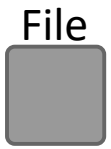
Schema defines **types** of vertices and edges.

Each vertex or edge type has a **name**, and a set of **attributes**.

Vertex Types:

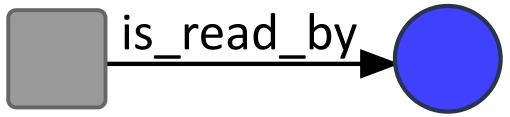


attrs.: {“id”, “start time”, “status”, …}

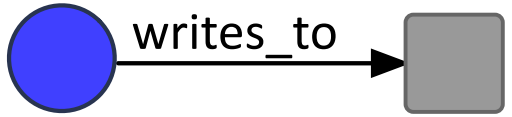


attrs.: {“location”, “size”, “last updated”, …}

Edge Types:

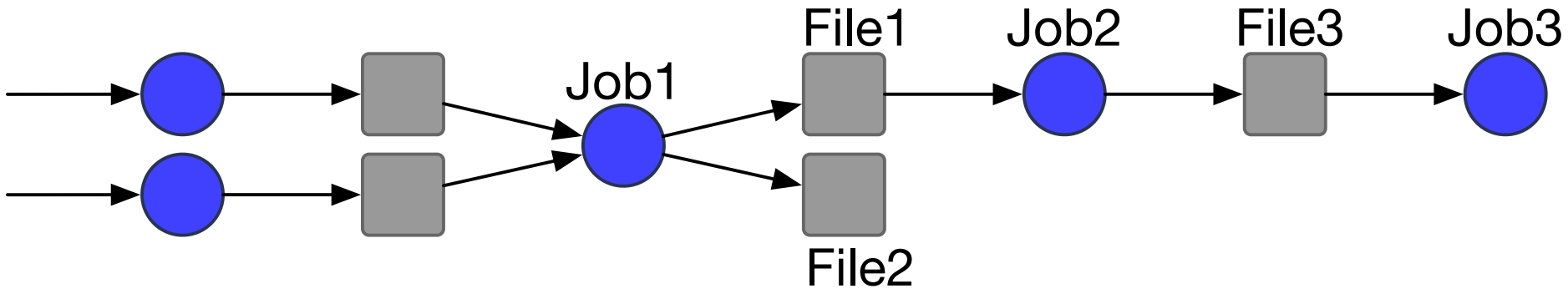


attrs.: {“start time”, “end time”, “bytes read”, …}



attrs.: {“start time”, “end time”, “bytes written”, …}

What is a Graph Query?



Q1: "Which files has Job1 written to?"

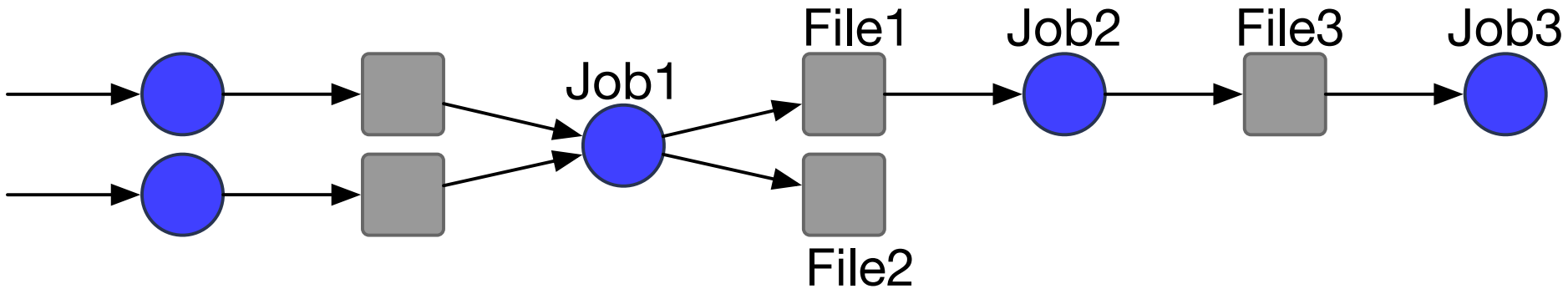
Q2: "Which jobs have read File1?"

Q3: "Which jobs may have indirectly consumed data from File1?"

Q4: "Which jobs depend on Job1?"

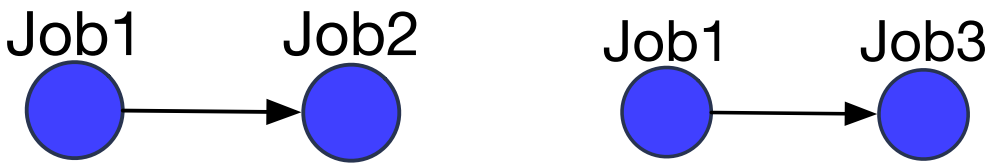
k-hop path queries

What is a Graph View?

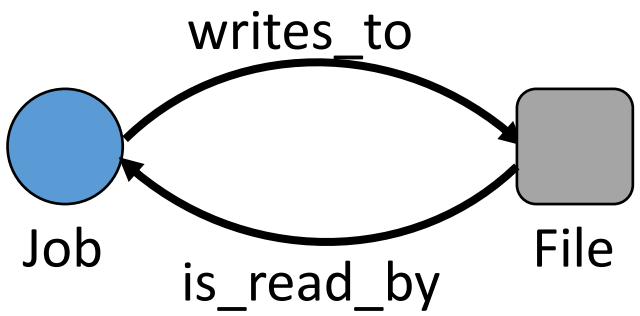


V1: "Which files has Job1 written to?"

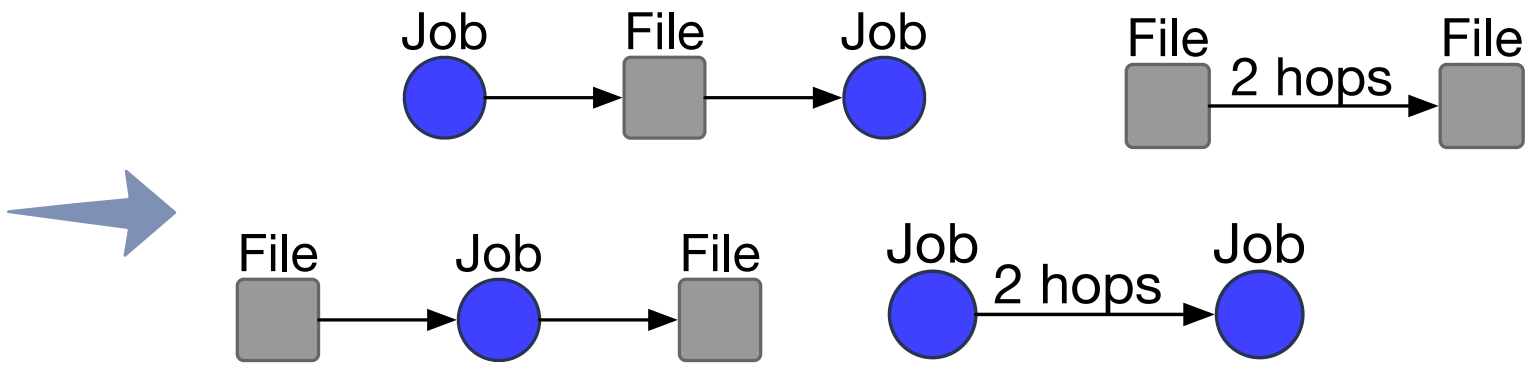
V2: "Which jobs depend on Job1?"



Key Insight: exploit regular structure in graphs to improve query performance

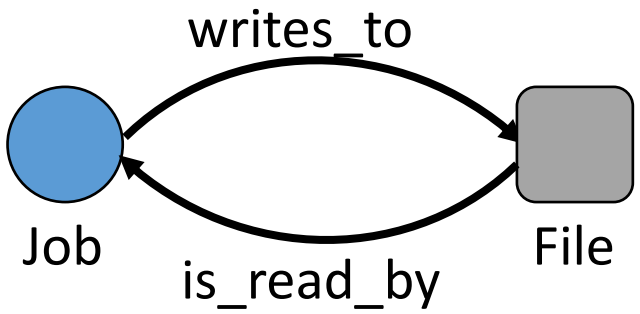


Schema for data lineage graph.

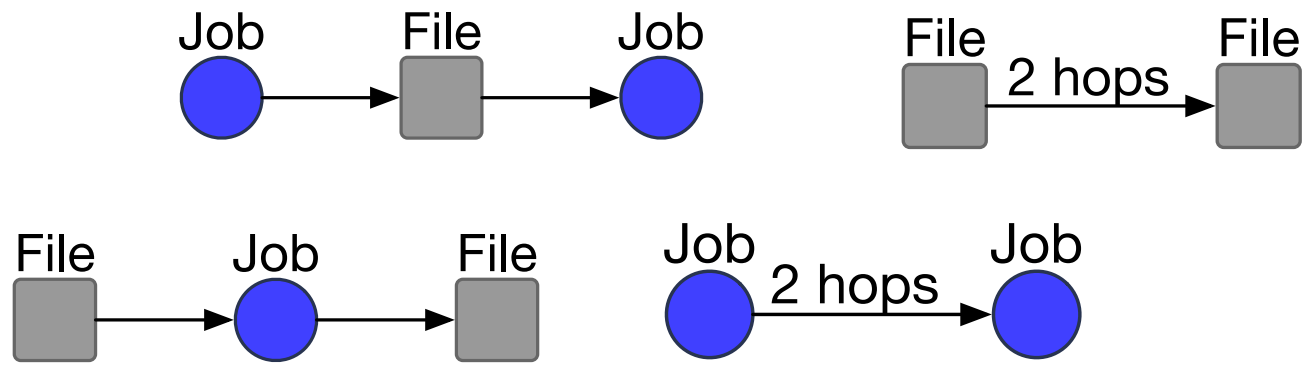


Schema dictates **regularity:** can use it to infer smaller **graph views**.

Key Insight: exploit regular structure in graphs to improve query performance



Schema for data lineage graph.



Schema dictates **regularity:** can use it to infer smaller **graph views**.

Statistic	Example
Cardinality	- No. vertices per Type - No. edges per type
Degree Distribution	- Max out-deg per vertex type - Median out-deg per vertex type

Example graph data statistics

Graph data **statistics** tell us when these views have a lot **fewer vertices and edges**, leading to much faster queries.

Problem Statement

Inputs

- Graph queries
- Graph schema
- Graph data statistics

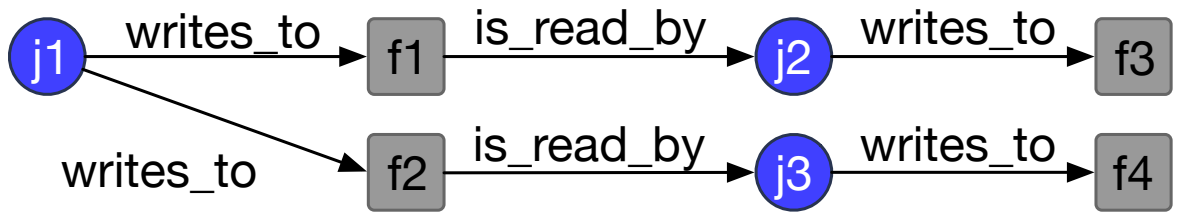
Goals

Find **graph views** that can be used to answer queries more efficiently.

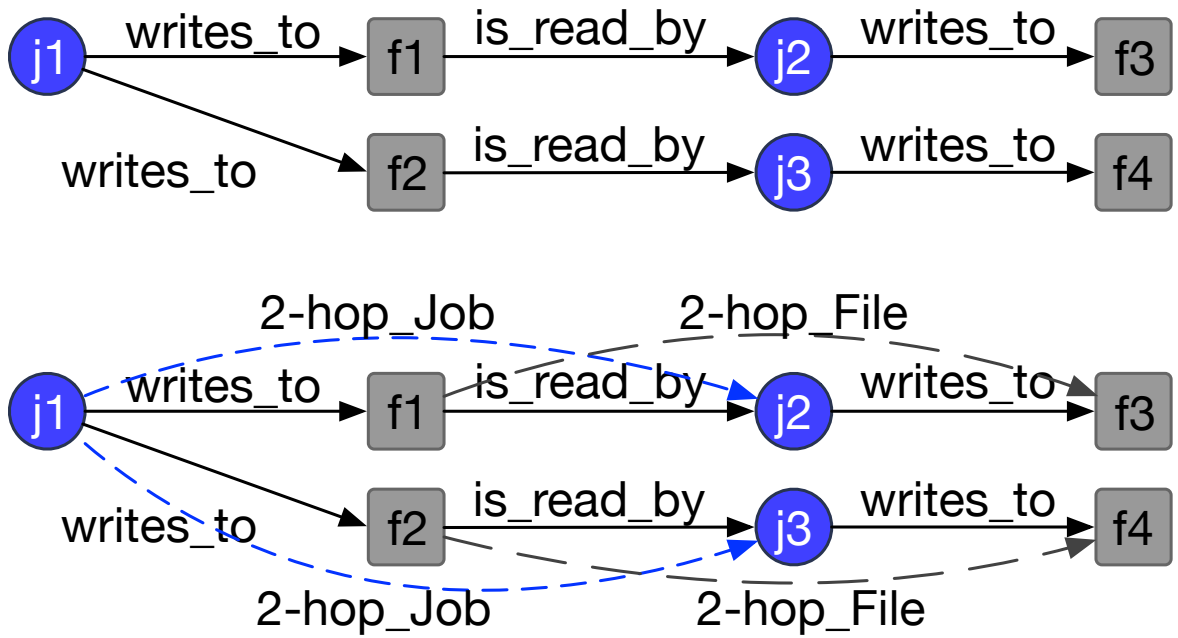
Translate original queries to equivalent version (*query rewriting*).

Kaskade: Graph Views for Efficient Graph Analytics

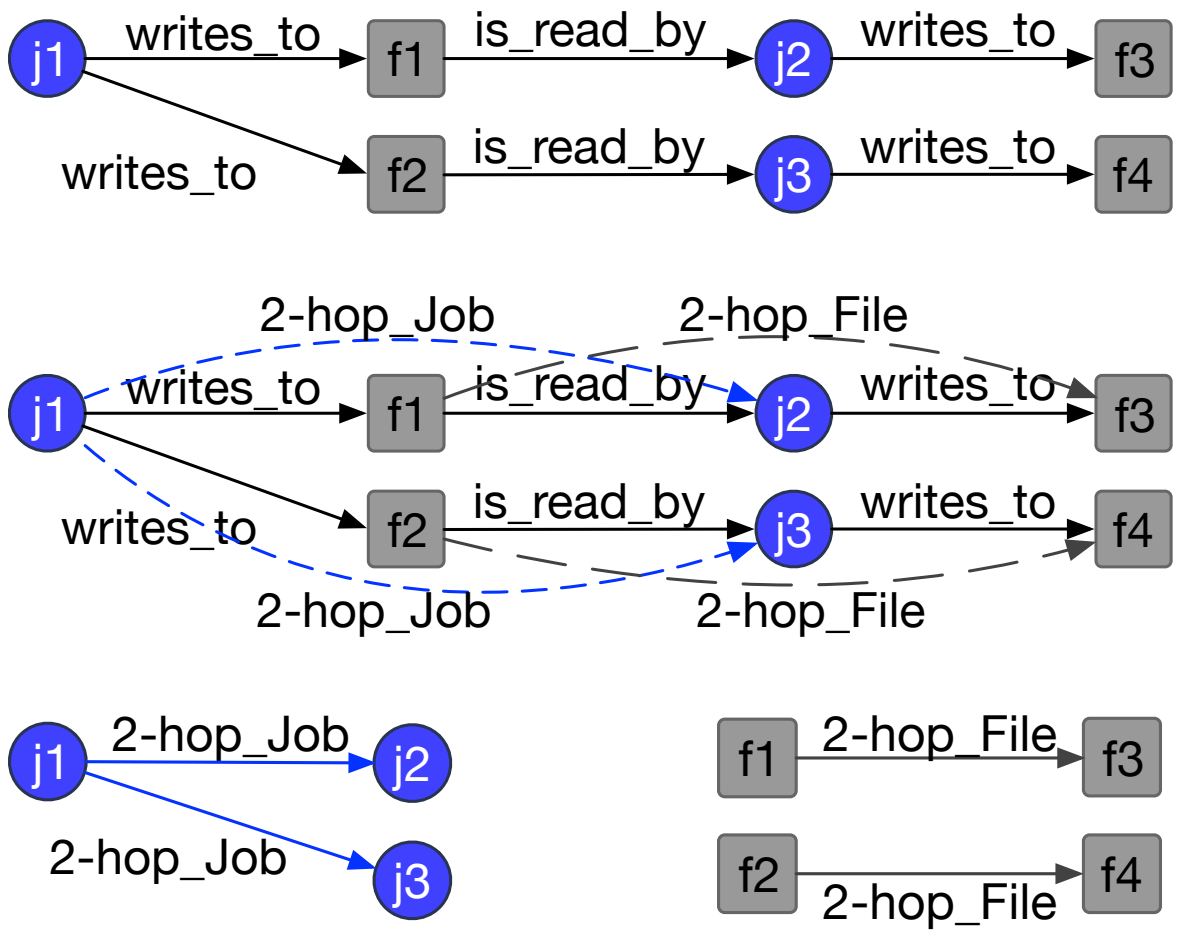
Graph view example: 2-hop connectors on data lineage graph



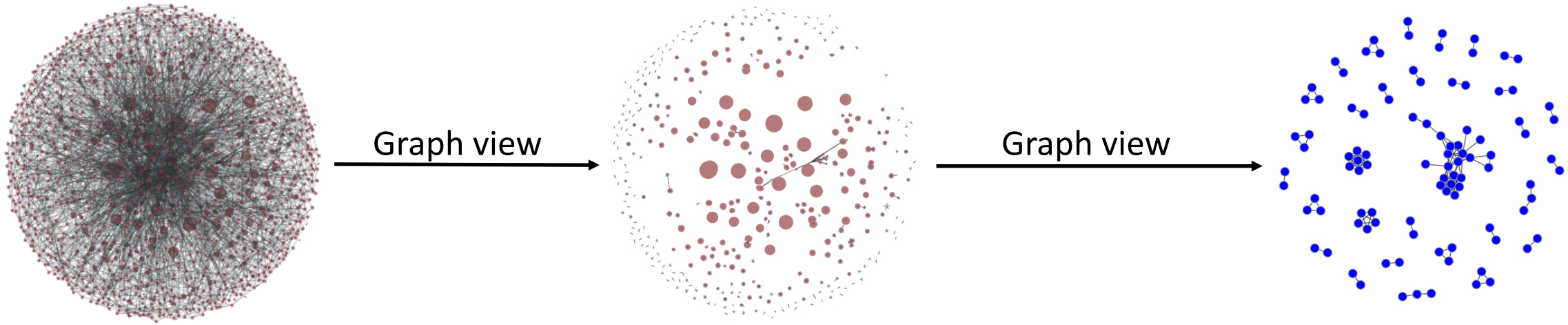
Graph view example: 2-hop connectors on data lineage graph



Graph view example: 2-hop connectors on data lineage graph



Example: input data size reduction when using graph views to answer “blast radius” query



raw graph

job-file graph

job-job graph

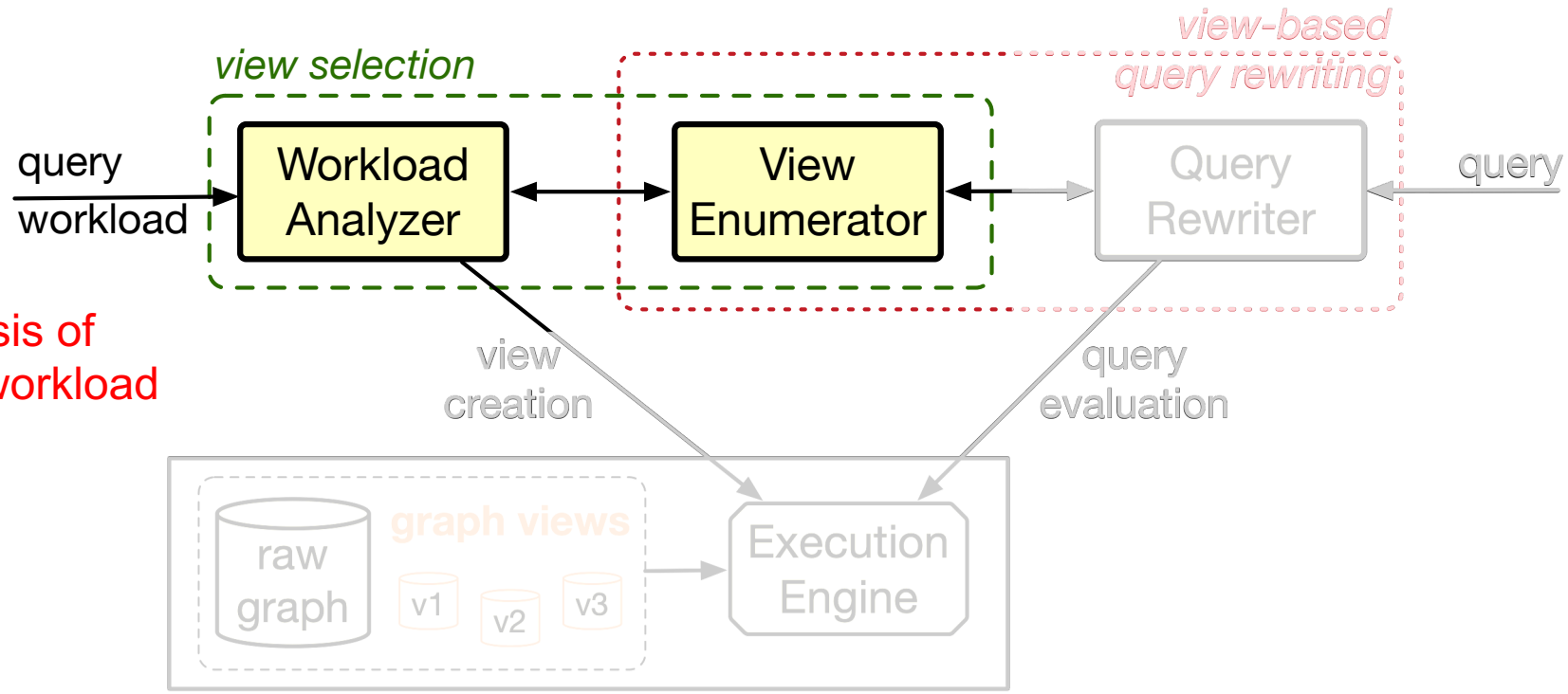
Size:
O(B) vertices
O(B) edges
O(TB) on disk

Size:
O(M) vertices
O(M) edges
O(GB) on disk

Size:
O(K) vertices
O(K) edges
O(MB) on disk

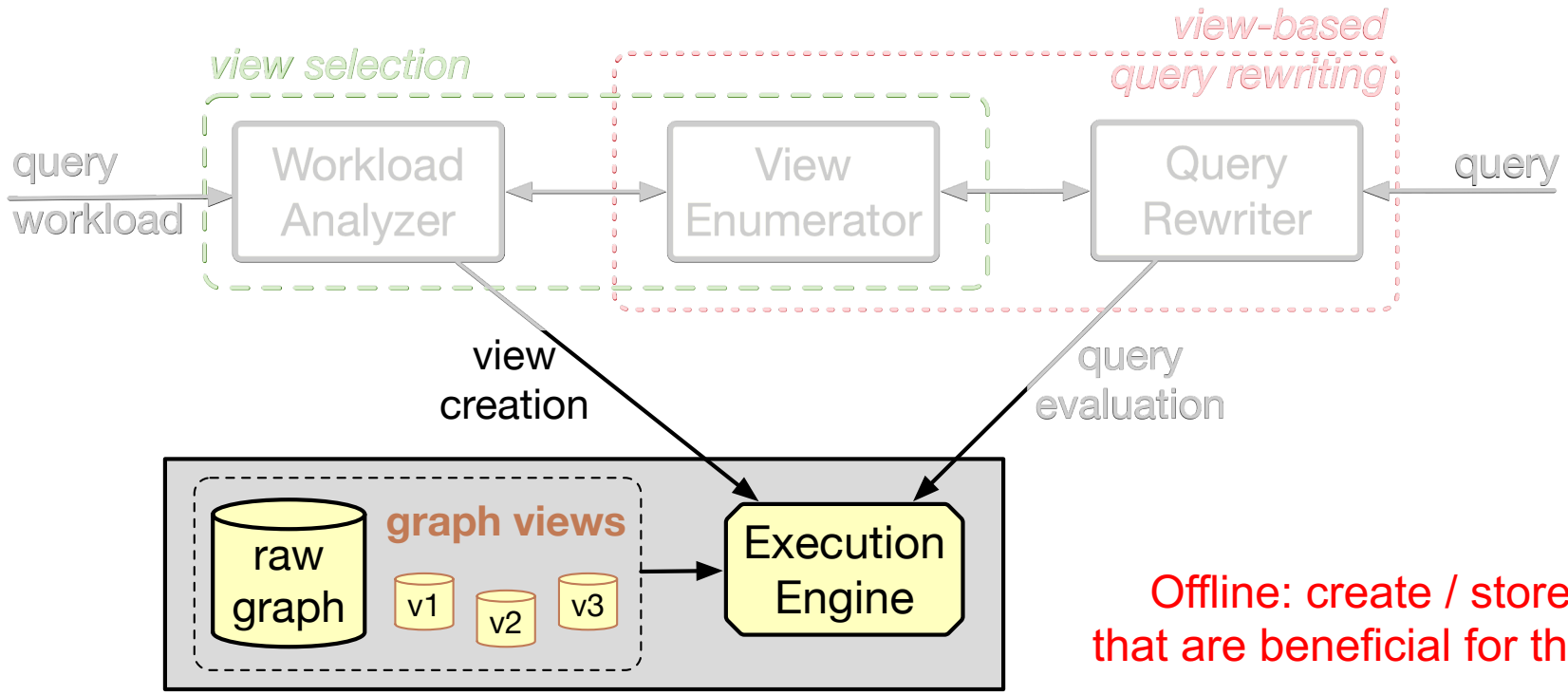
Kaskade: Graph Views for Efficient Graph Analytics

Kaskade: a system for graph query optimization using graph views



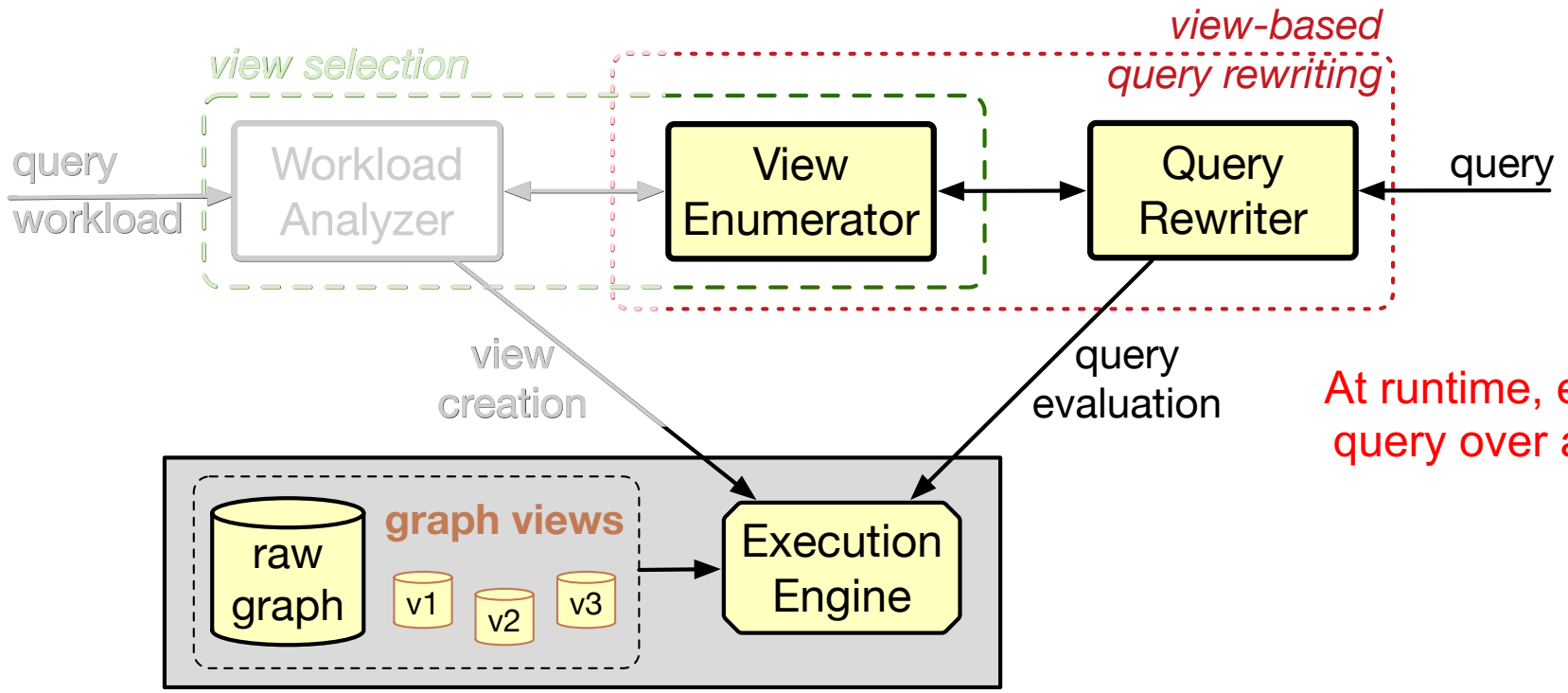
Offline: analysis of historical query workload

Kaskade: a system for graph query optimization using graph views



Offline: create / store views
that are beneficial for the queries

Kaskade: a system for graph query optimization using graph views

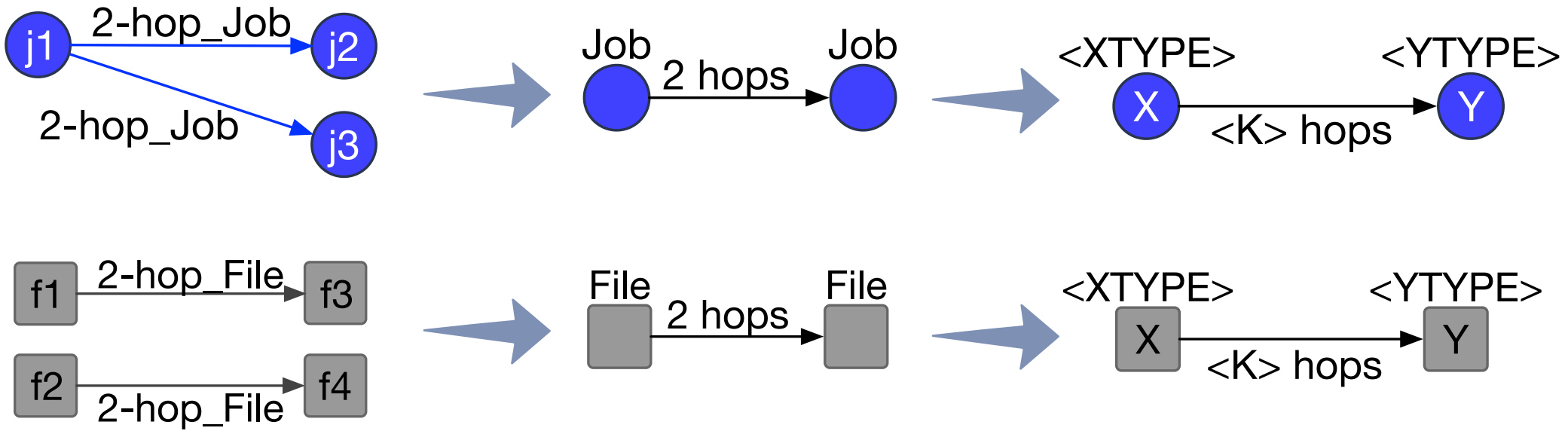


At runtime, execute rewritten query over appropriate view

Key contribution: exploit graph regularity to find efficient views for faster query execution

Graph view templates define the possible views we consider

Graph view templates define the possible views we consider



Concrete 2-hop connector views *are instantiations of* *k-hop connector view template.*

View template: kHopConnector(X, Y, XTYPE, YTYPE, K) with X=Y

Key contribution: exploit graph regularity to find efficient views for faster query execution

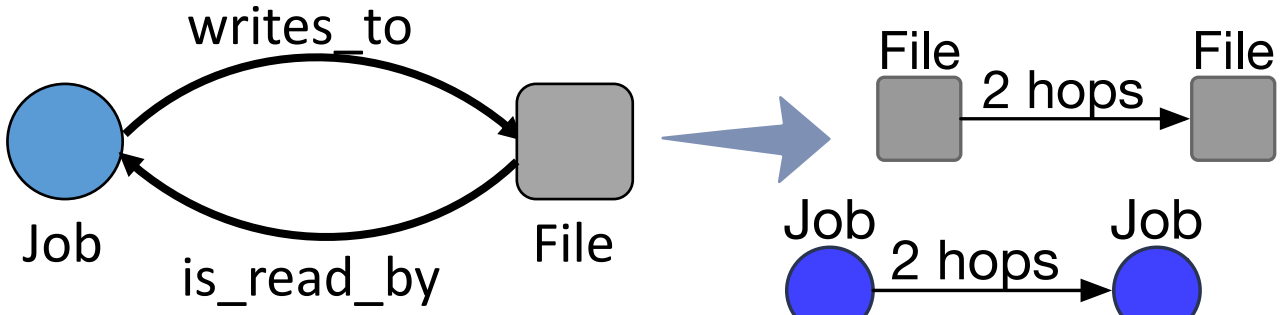
Graph view templates define the possible views we consider

Constraints are

- regular patterns imposed by the graph's schema (a graph)
- regular patterns imposed by the query (a graph)

Constraints are regular patterns imposed by the graph's schema and query

Schema is a graph which imposes constraints on what views can be enumerated.



Schema for data lineage graph.

Query is a graph which imposes constraints on what views can be enumerated.

```
MATCH (j1:Job)->(f1:File)
      (f1:File)-[r*0..k]->(f2:File)
      (f2:File)->(j2:Job)
RETURN j1, j2
```

Pattern matches all jobs up to k hops away

Key contribution: exploit graph regularity to find efficient views for faster query execution

Graph view templates define the possible views we consider

Constraints are

- regular patterns imposed by the graph's schema (a graph)
- regular patterns imposed by the query (a graph)

Constraint-based view enumeration

- exploits **regularity** on graph's schema and data's structural properties.
- **finds views** that can correctly answer queries.
- rewrites queries in terms of these views.

Problem formulation: finding graph views as a Boolean satisfiability problem

Graph view enumerators (*templates*) are encoded as logical clauses that are **satisfiable iff view is feasible** given input constraints.

e.g., a k-hop Job-Job connector view can be used iff:

- A k-hop path exists between Job vertices in the graph's schema
- A k-hop path exists between Job vertices in the query
- Only Job vertices are projected out in the query's schema

Problem formulation: finding graph views as a Boolean satisfiability problem

-A k-hop path exists between Job vertices in the query

-A k-hop path exists between Job vertices in the graph's schema

```
% k-hop connector between nodes X and Y.
kHopConnector(X, Y, XTYPE, YTYPE, K) :-
  % query constraints
  queryVertexType(X, XTYPE),
  queryVertexType(Y, YTYPE),
  queryKHopPath(X, Y, K),
  % schema constraints
  schemaKHopPath(XTYPE, TYPE, K).
```

Prolog definition for k-hop connector view template

Problem formulation: finding graph views as a Boolean satisfiability problem

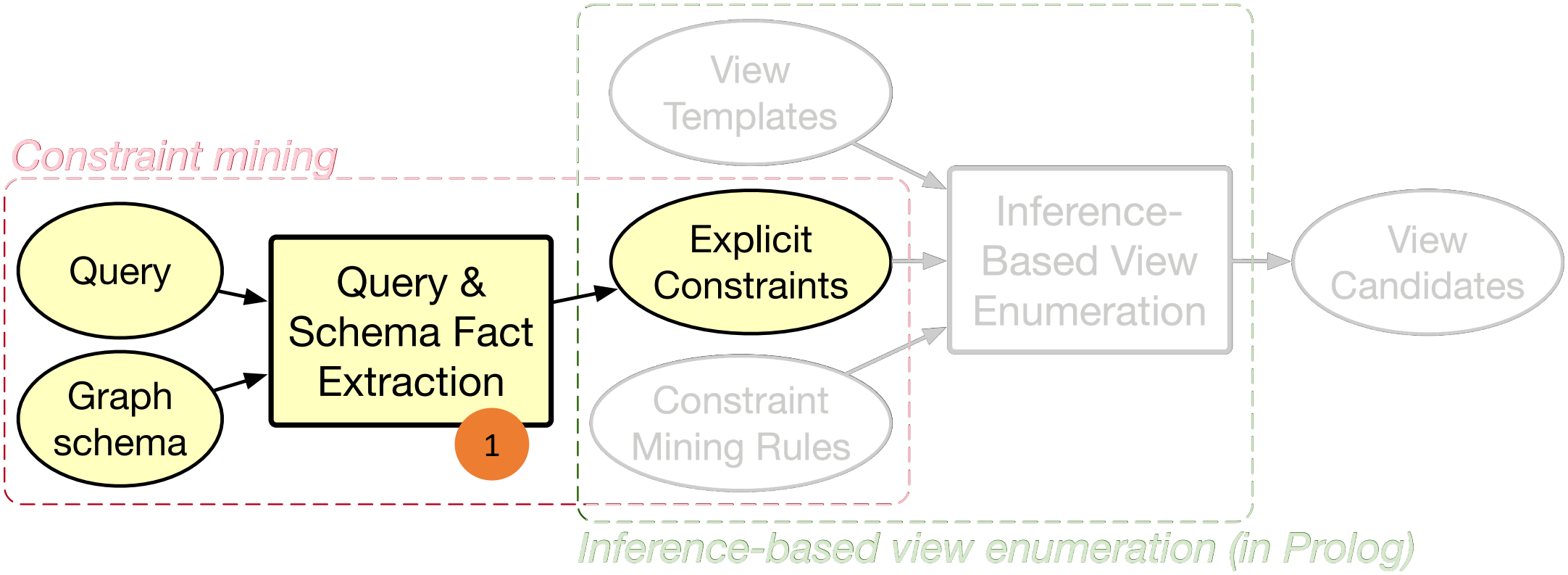
-A k-hop path exists between Job vertices in the query

-A k-hop path exists between Job vertices in the graph's schema

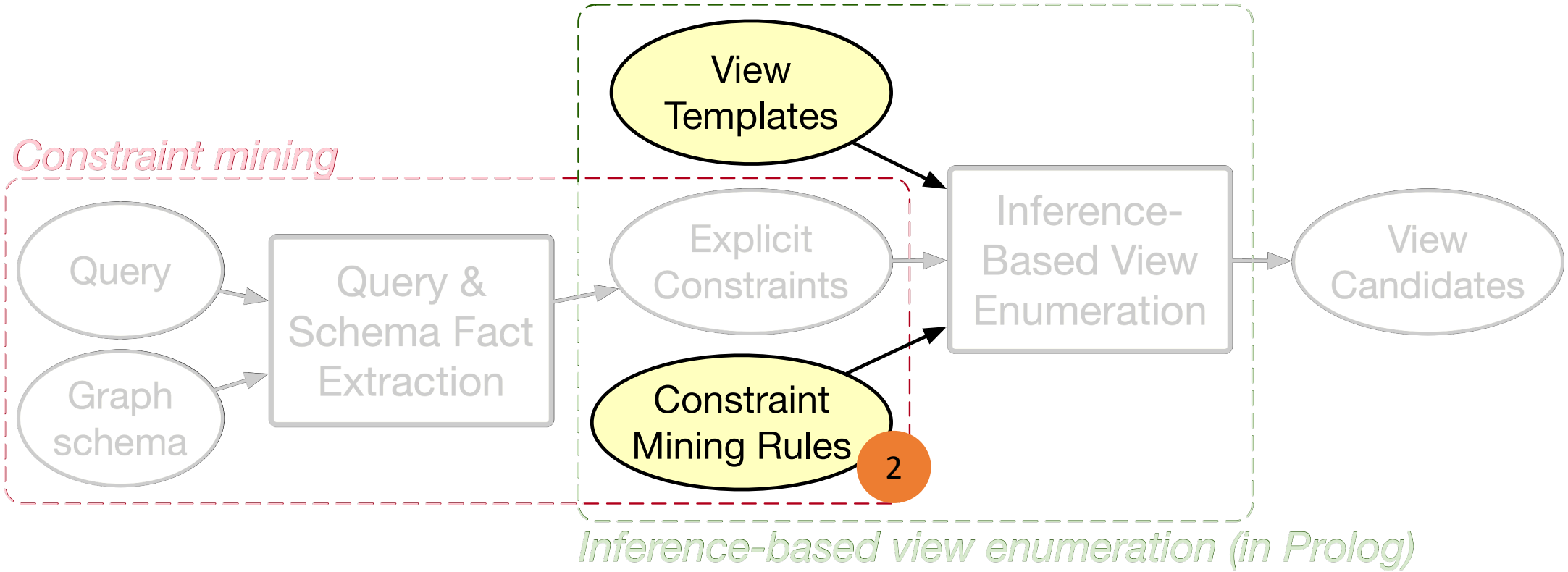
```
% k-hop connector between nodes X and Y.
kHopConnector(X, Y, XTYPE, YTYPE, K) :-
  % query constraints
  queryVertexType(X, XTYPE),
  queryVertexType(Y, YTYPE),
  queryKHopPath(X, Y, K),
  % schema constraints
  schemaKHopPath(XTYPE, TYPE, K).
```

Prolog definition for k-hop connector view template

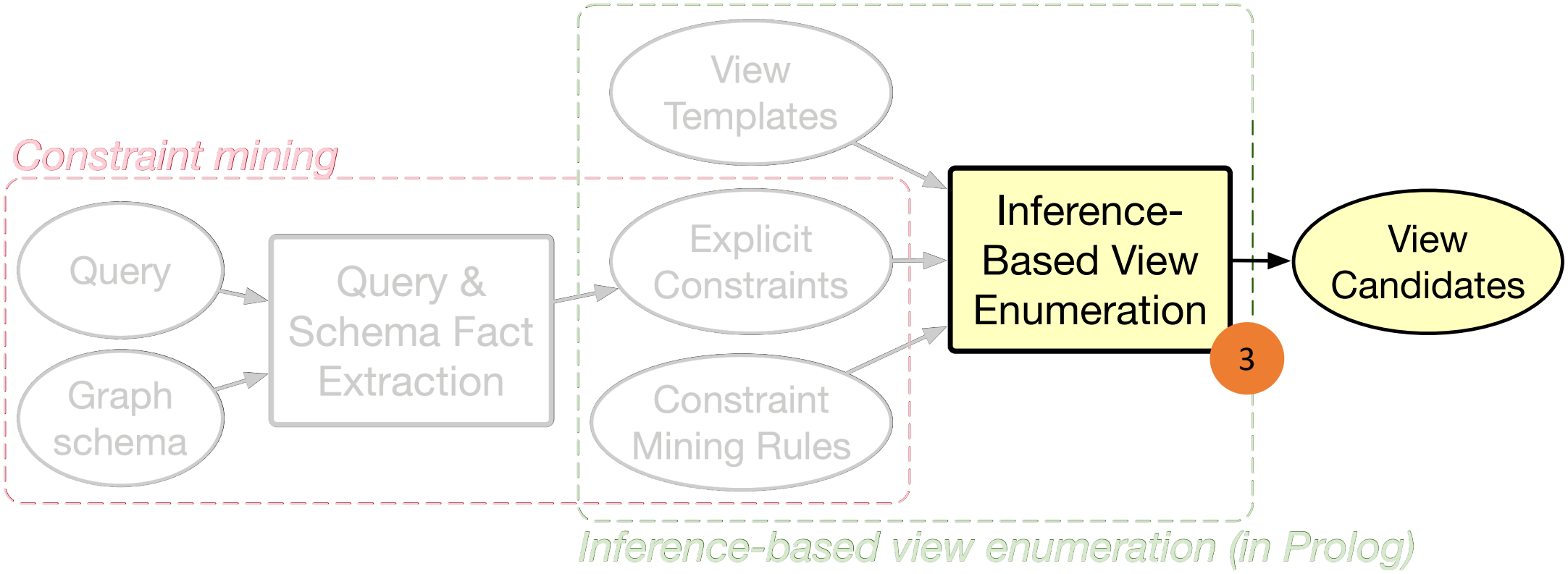
Recap: Constraint-based view enumeration



Recap: Constraint-based view enumeration



Recap: Constraint-based view enumeration



Constraint-based view enumeration: Why do we need a cost model?

Combinatorial problem: **constraints reduce search space** of possible views, but can still enumerate **more views than can fit in memory**.

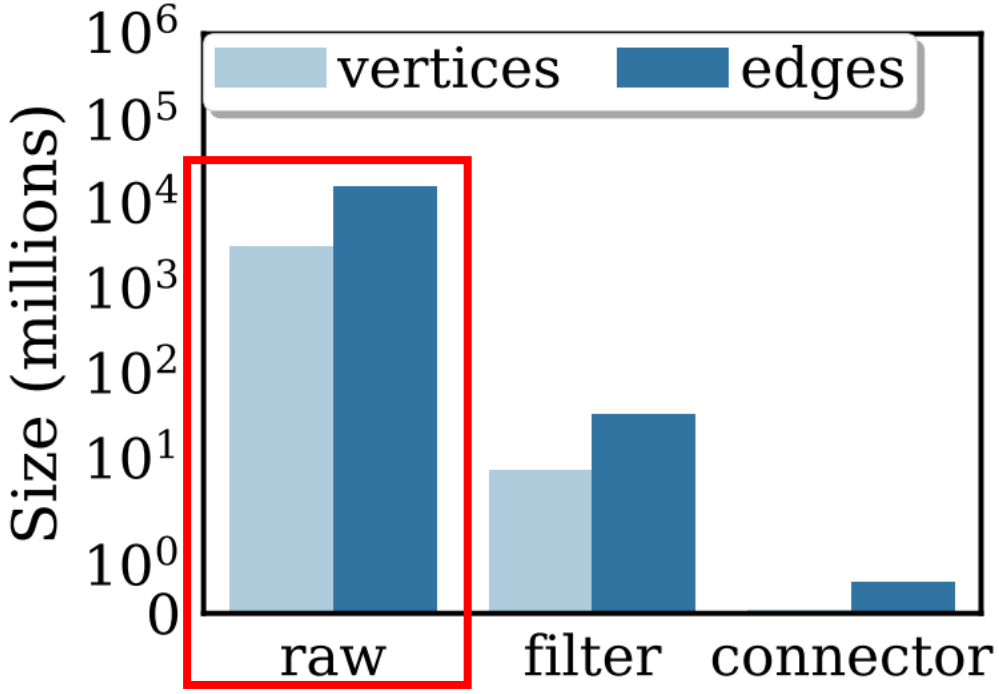
Cost model helps us **estimate how beneficial** it is to materialize a view.

Views are costed in terms of estimated size (I/O dominates).

Optimizer picks best view subject to resource budget (memory).

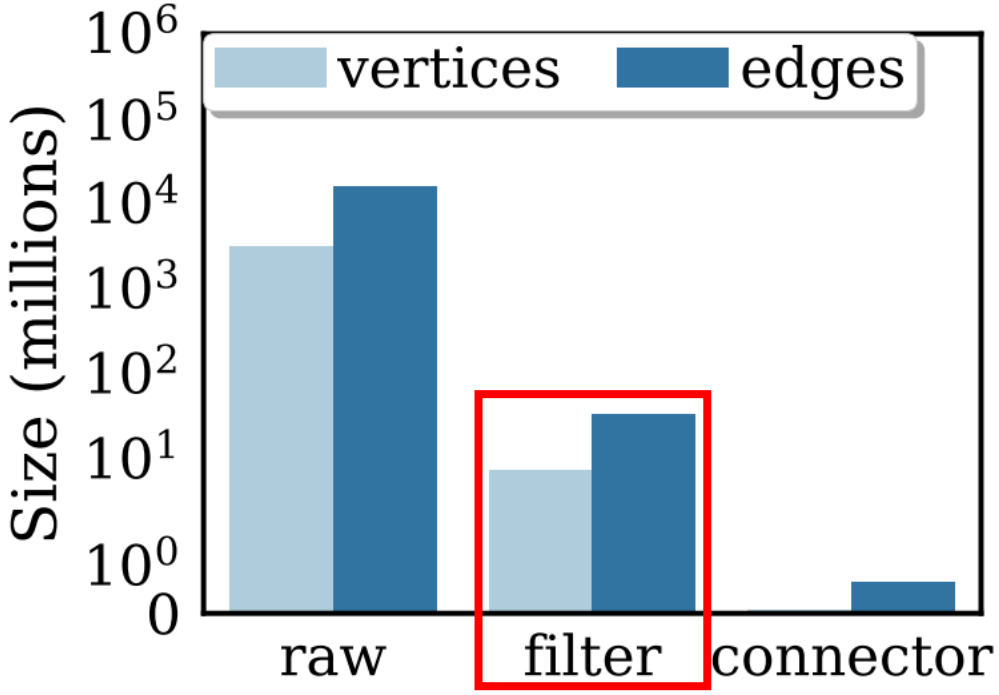
Experimental Results

Example gains in size reduction



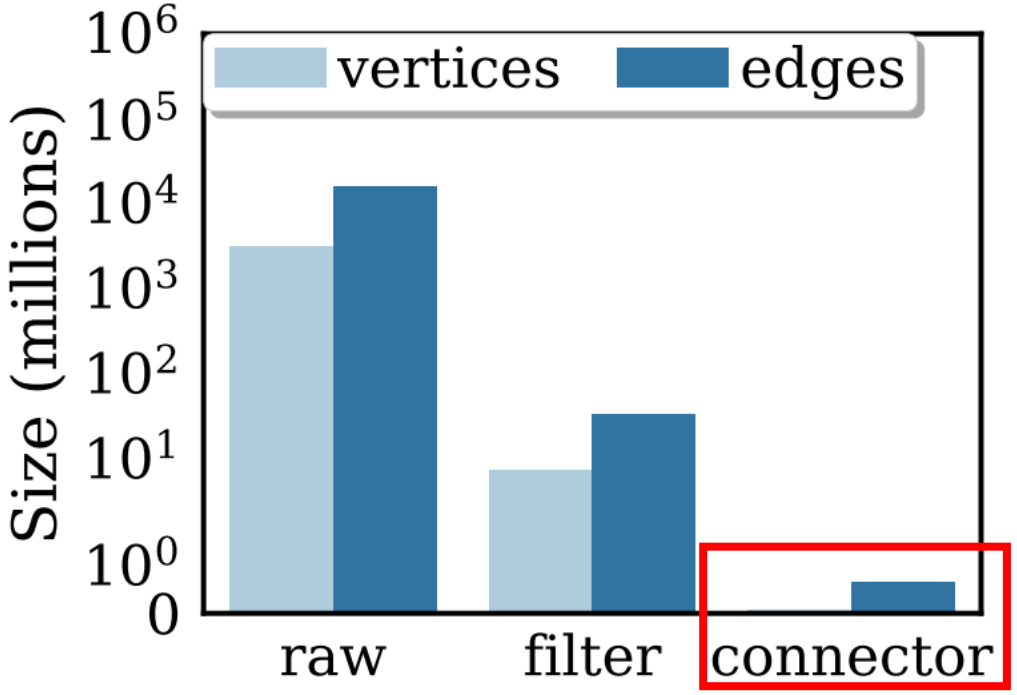
Original data lineage raw graph has billions of vertices and edges.

Example gains in size reduction



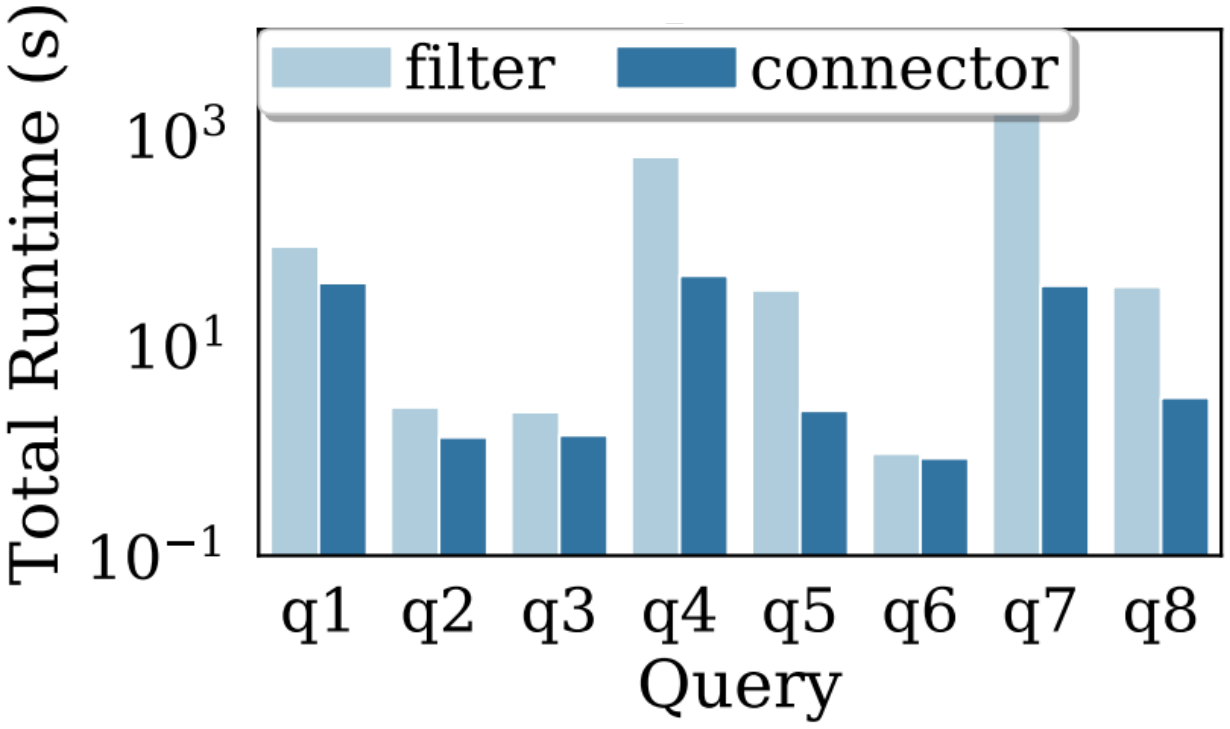
Graph view that filters out unnecessary vertex/edge types has millions of edges.

Example gains in size reduction



A 2-hop job-job connector graph view further reduces size.

Example gains in query runtime



Runtime gains compound: orders of magnitude faster on smallest view.

Kaskade: Key Technical Contributions

Graph view selection for query rewriting

- Introduced class of graph views that are beneficial for different queries
- Constraint-based view enumeration is guided by queries and structural properties of graph

Significant performance gains

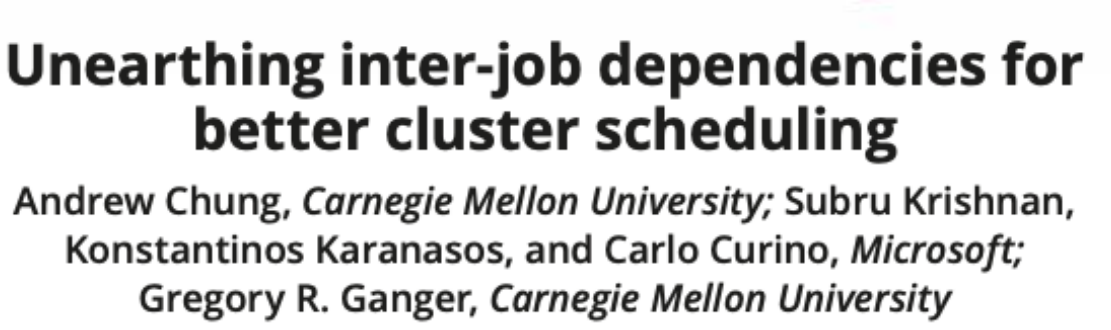
- Up to 50x faster response times, orders of magnitude reduction in size
- Rewriting techniques are engine-agnostic: only rely on fundamental graph transformations

Kaskade: Misc. Contributions

Wrote graph views extraction pipeline, made available in prod

Pipeline served as basis for other papers from Microsoft folks, e.g.,

SIGMOD 2019



OSDI 2020

Thesis Outline

Kaskade: **Graph Views** for Efficient Graph Analytics

J M F da Trindade, K Karanasos, C Curino, S Madden, J Shun

ICDE 2020

Kairos: Efficient **Temporal Graph Analytics** on a **Single Machine**

J M F da Trindade, J Shun, S Madden, N Tatbul

Under submission to VLDB 2024

Kairos: Key Technical Contributions

Efficient **parallel implementation of algorithms over temporal graphs**, e.g.

- minimal paths (earliest arr., latest dep., fastest, shortest duration)
- centrality (betweenness using fastest or shortest duration)
- k-core, ranking, and connectivity

Efficient data layouts

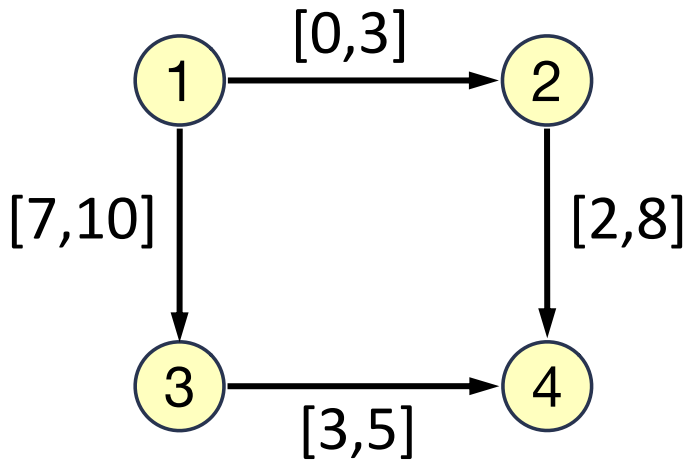
- **T-CSR: compressed sparse row** for temporal graphs
- **TGER**: highly optimized **parallel index** for temporal edges

Selective Indexing

- chooses best access method to retrieve vertex's nbors at runtime

What is a temporal graph?

$$G = (V, E, \tau)$$



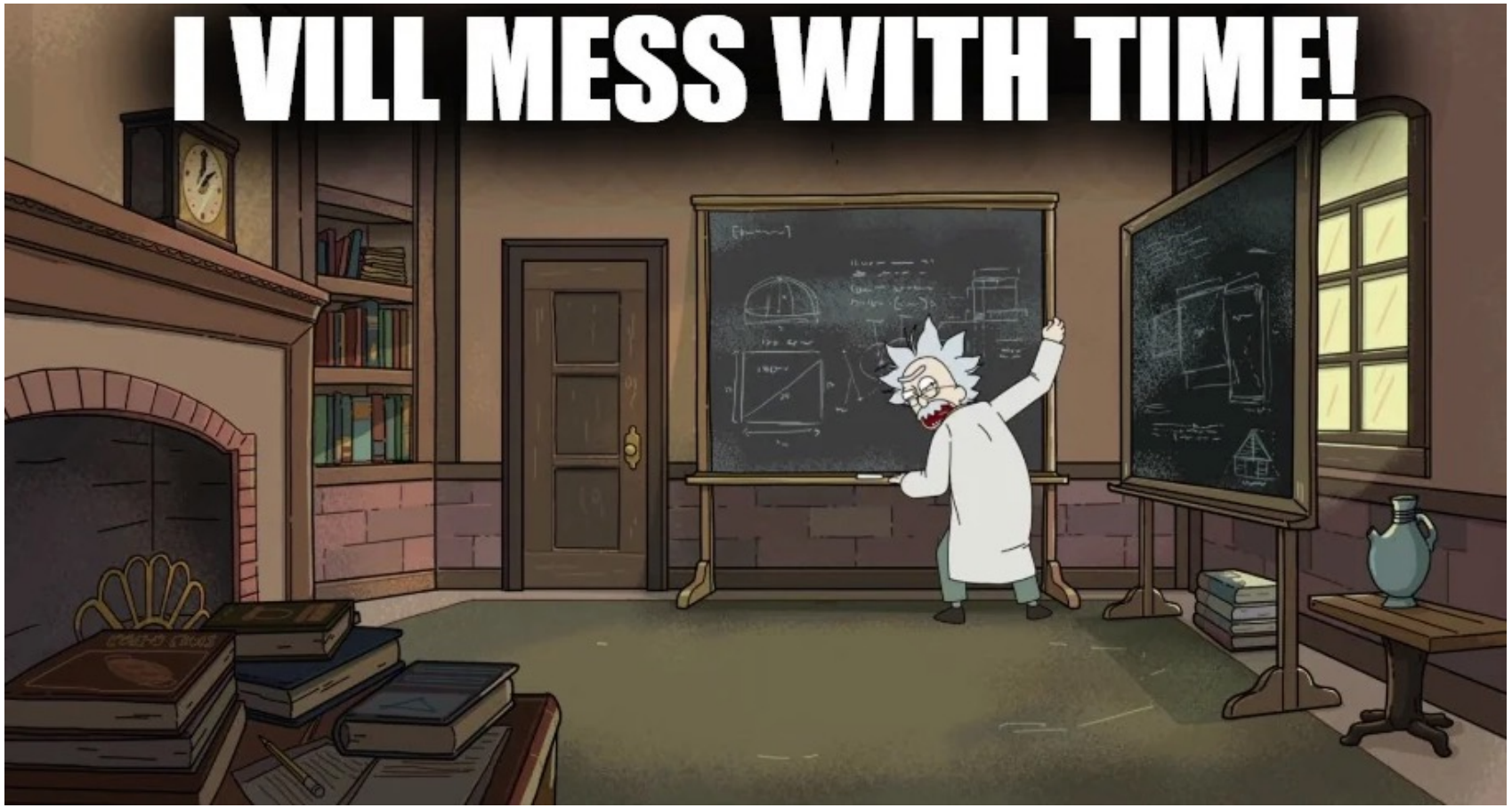
$V = \{1, 2, 3, 4\}$

$E = \{(1,2), (1,3), (2, 4), (3, 4)\}$

$\tau: E \rightarrow \mathbb{N} \times \mathbb{N}$

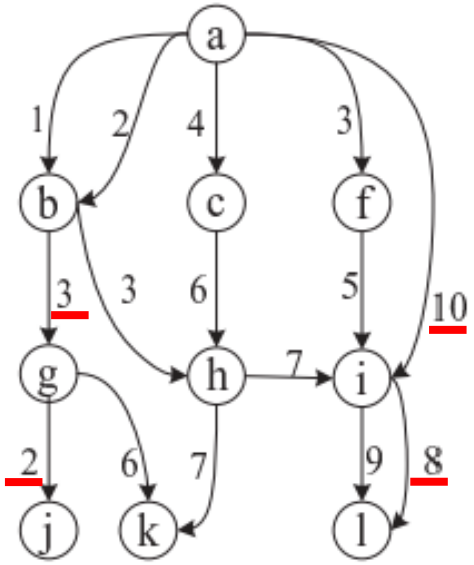
AKA as “temporal network”: edges only valid during [start time, end time] interval

But really, why temporal graphs?

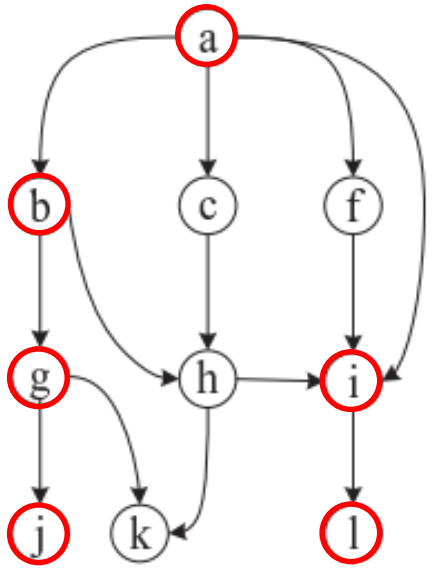


Rick and Morty – A Rickle In Time (Season 2, Episode 1)

Why do we care about temporal graphs? Certain questions answered incorrectly without time info



(a) Temporal Graph



(b) Static Graph

Paths may be invalid or suboptimal when temporal info is ignored

- <a, b, g, j> is not valid
- <a, i, l> is not shortest

H. Wu et al, *Path Problems in Temporal Graphs.* VLDB 2014

Why do we care about temporal graphs? Certain questions answered incorrectly without time info

Temporal Networks

Petter Holme[1,2,3] and Jari Saramäki[4]

[1]*IceLab, Department of Physics, Umeå University, 901 87 Umeå, Sweden*
[2]*Department of Energy Science, Sungkyunkwan University, Suwon 440–746, Korea*
[3]*Department of Sociology, Stockholm University, 106 91 Stockholm, Sweden*
[4]*Department of Biomedical Engineering and Computational Science, School of Science, Aalto University, 00076 Aalto, Espoo, Finland*

communications physics

Use of temporal contact graphs to understand the evolution of COVID-19 through contact tracing data

Mincheng Wu[1,2,8], Chao Li[1,2,8], Zhangchong Shen[2], Shibo He[1,2✉], Lingling Tang[3], Jie Zheng[4], Yi Fang[5], Kehan Li[2], Yanggang Cheng[2], Zhiguo Shi[6], Guoping Sheng[3], Yu Liu[5], Jinxing Zhu[5], Xinjiang Ye[5], Jinlai Chen[5], Wenrong Chen[5], Lanjuan Li[7✉], Youxian Sun[1,2] & Jiming Chen[1,2✉]

Temporal Graph Benchmark for Machine Learning on Temporal Graphs

Shenyang Huang[1,2,*] Farimah Poursafaei[1,2,*] Jacob Danovitch[1,2] Matthias Fey[3]
Weihua Hu[3] Emanuele Rossi[4] Jure Leskovec[7] Michael Bronstein[8]
Guillaume Rabusseau[1,5,6] Reihaneh Rabbany[1,2,6]
[1]Mila - Quebec AI Institute, [2]School of Computer Science, McGill University
[3]Kumo.AI, [4]Imperial College London, [5]DIRO, Université de Montréal
[6]CIFAR AI Chair, [7]Stanford University, [8]University of Oxford

You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis

Qi Wang[1], Wajih Ul Hassan[1], Ding Li[2], Kangkook Jee[3], Xiao Yu[2]
Kexuan Zou[1], Junghwan Rhee[2], Zhengzhang Chen[2], Wei Cheng[2], Carl A. Gunter[1], Haifeng Chen[2]
[1]University of Illinois Urbana-Champaign [2]NEC Laboratories America, Inc. [3]University of Texas at Dallas
{qiwang11, whassan3, kzou3, cgunter}@illinois.edu kangkook.jee@utdallas.edu
{dingli, xiao, rhee, zchen, weicheng, haifeng}@nec-labs.com

Kairos: Key Technical Contributions

Efficient **parallel implementation of algorithms over temporal graphs**, e.g.

- minimal paths (earliest arr., latest dep., fastest, shortest duration)
- centrality (betweenness using fastest or shortest duration)
- k-core, ranking, and connectivity

Categories of temporal graph analytics tasks

Temporal Minimal Paths

- Provenance: tracking information flow in machine generated graphs
- Indoor routing: temporal shortest paths for robot navigation
- Epidemiology: identifying infection transmission paths for contact tracing

Temporal Connectivity

- Social networks: analyzing community evolution

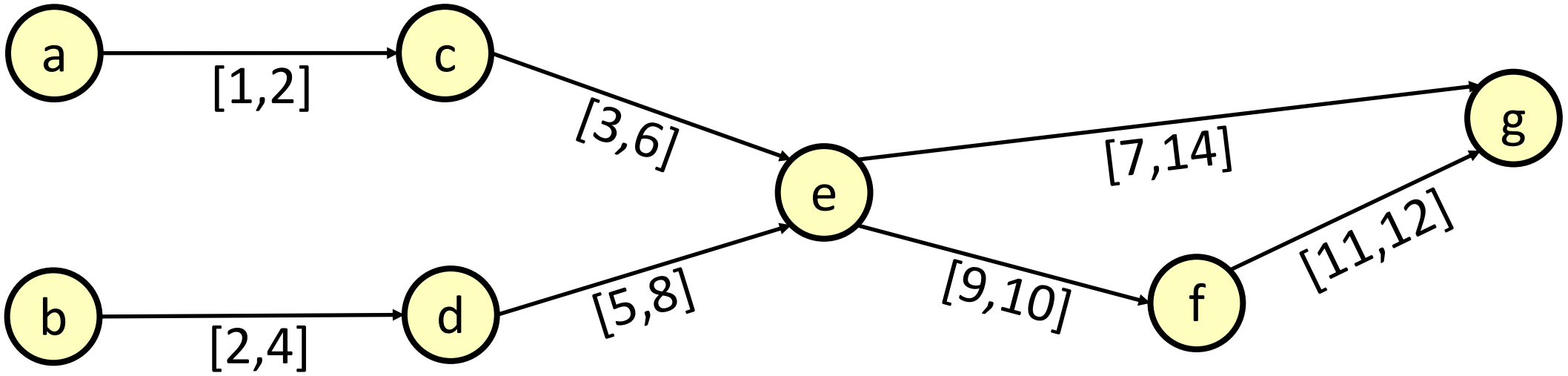
Temporal Centrality

- Epidemiology: identifying critical nodes in the spread of infections

Time-constrained reachability

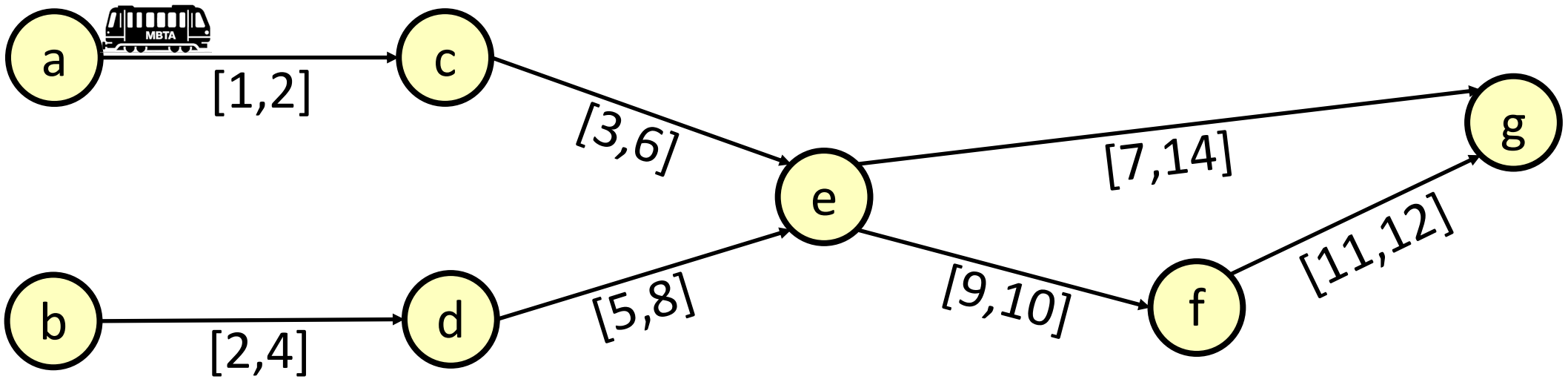
- Social networks: analyzing influence propagation

Example T. Minimal Path: Earliest Arrival



Input: **graph *G***, source vertex *u* in *V*, time interval $[t_s, t_e]$
Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$

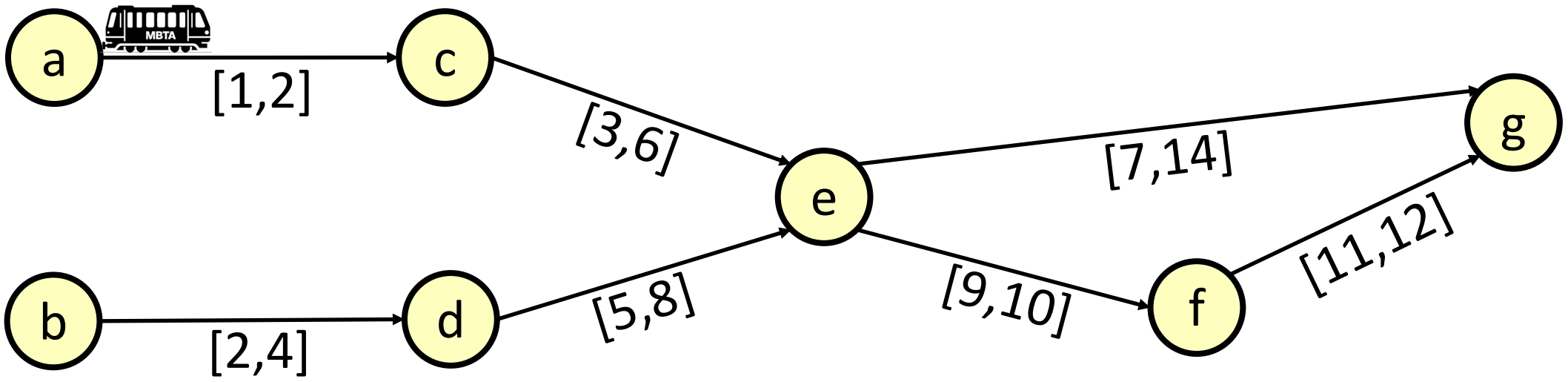
Example T. Minimal Path: Earliest Arrival



Input: graph *G*, **source vertex *u* in *V***, time interval $[t_s, t_e]$
Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$

***u* = a**

Example T. Minimal Path: Earliest Arrival

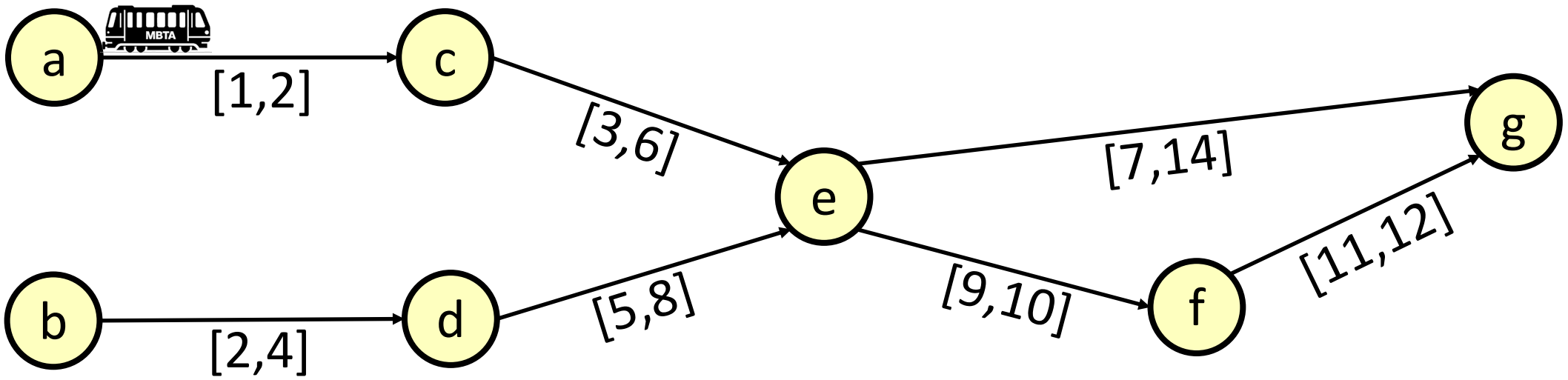


Input: graph *G*, source vertex *u* in *V*, **time interval [t_s, t_e]**
Output: earliest arrival from *u* to every vertex *v* in V within [t_s, t_e]

u = a

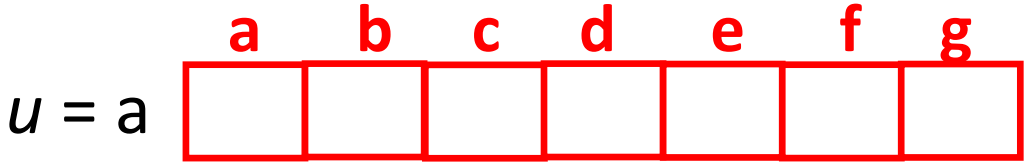
[t_s, t_e] = [1,15]

Example T. Minimal Path: Earliest Arrival



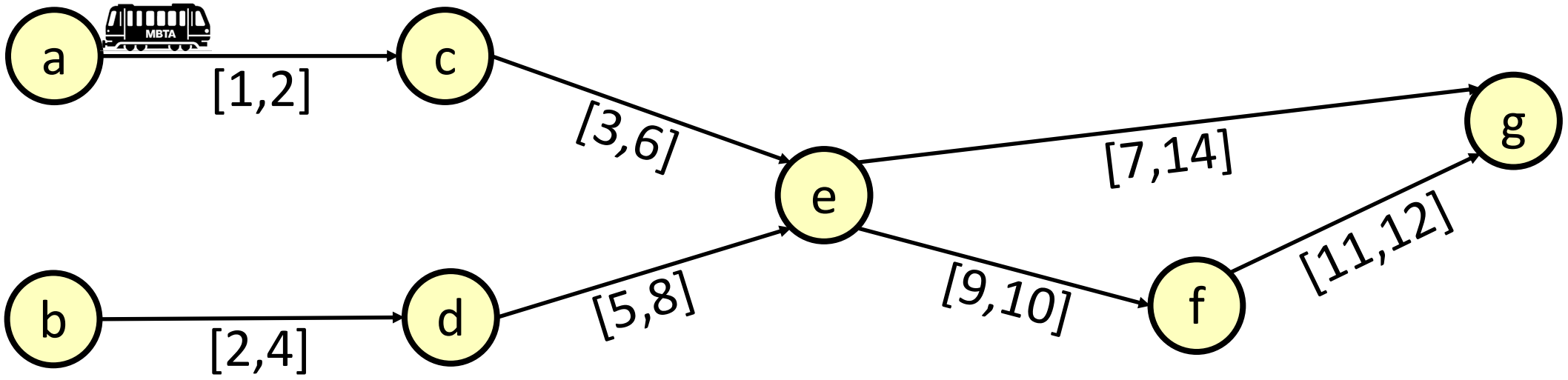
Input: graph *G*, source vertex *u* in *V*, time interval $[t_s, t_e]$

Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$

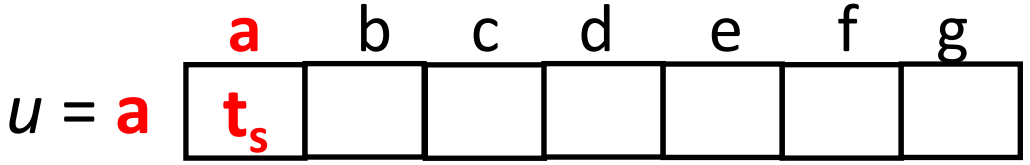


$[t_s, t_e] = [1,15]$

Example T. Minimal Path: Earliest Arrival

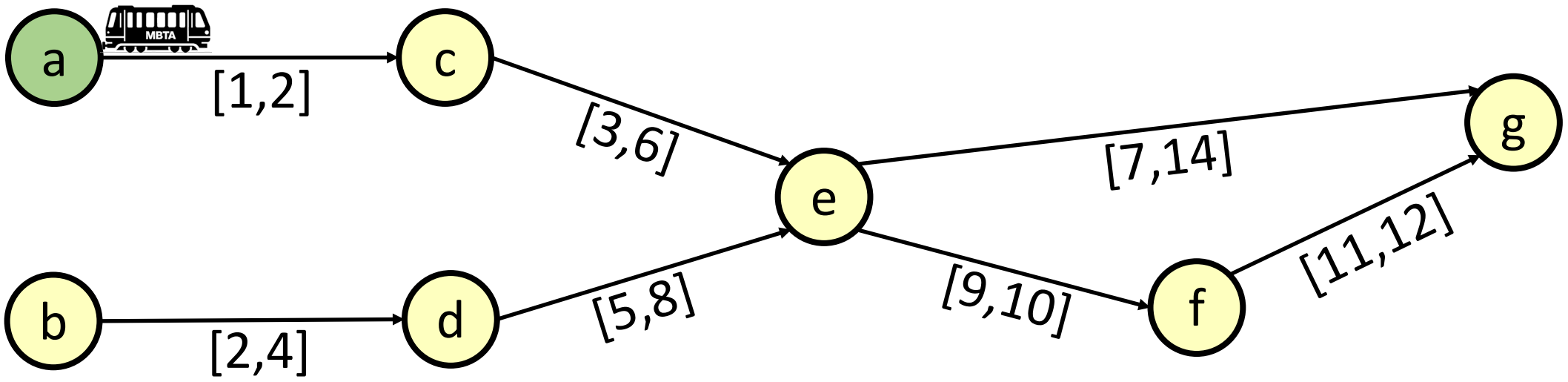


Input: graph *G*, source vertex *u* in *V*, time interval $[t_s, t_e]$
Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$



$[t_s, t_e] = [1,15]$

Example T. Minimal Path: Earliest Arrival

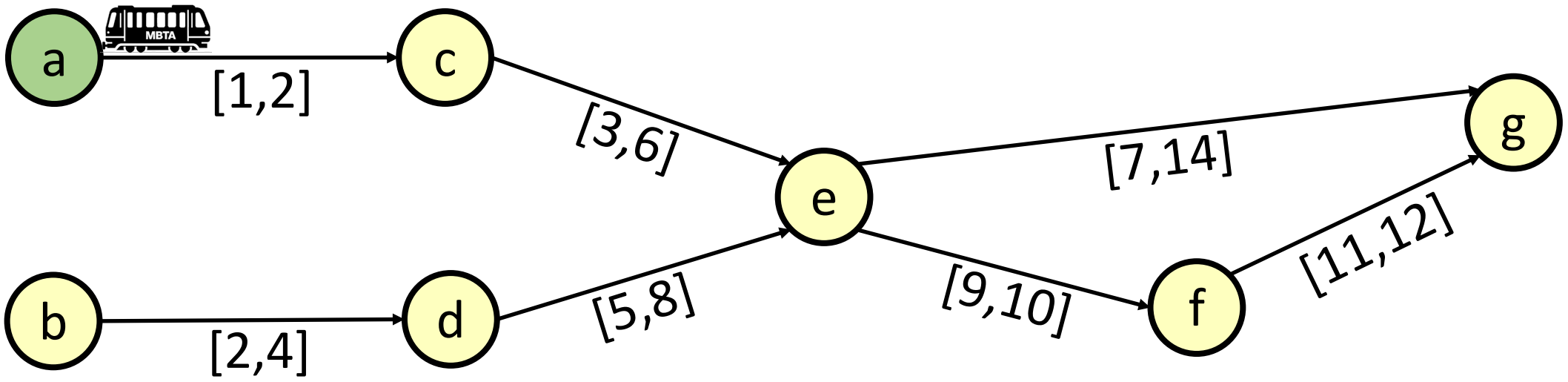


Input: graph *G*, source vertex *u* in *V*, time interval $[t_s, t_e]$
Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$

	a	b	c	d	e	f	g
u = **a**	1						

$[\mathbf{t_s}, t_e] = [\mathbf{1}, 15]$

Example T. Minimal Path: Earliest Arrival

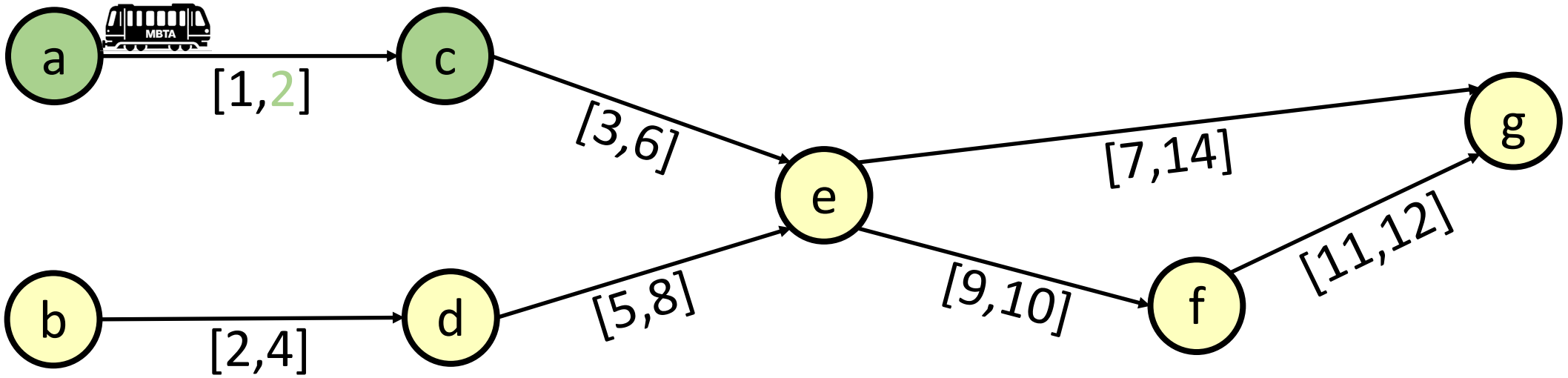


Input: graph *G*, source vertex *u* in *V*, time interval $[t_s, t_e]$
Output: earliest arrival from *u* to every vertex *v* in V within $[t_s, t_e]$

	a	b	c	d	e	f	g
u = a	1	∞	∞	∞	∞	∞	∞

$[t_s, t_e] = [1,15]$

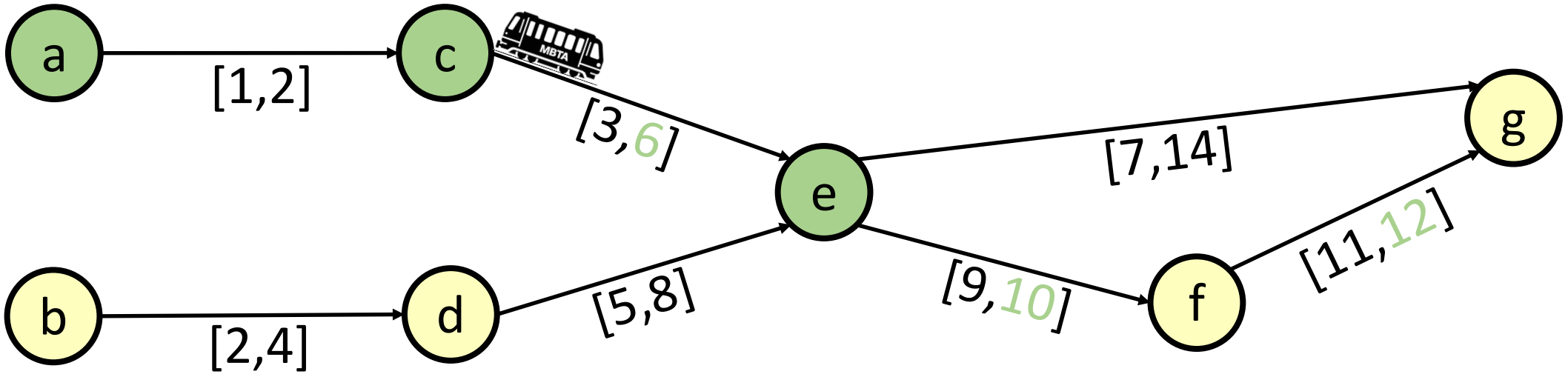
Example T. Minimal Path: Earliest Arrival



	a	b	c	d	e	f	g
u = a	1	∞	2				

$[t_s, t_e] = [1,15]$

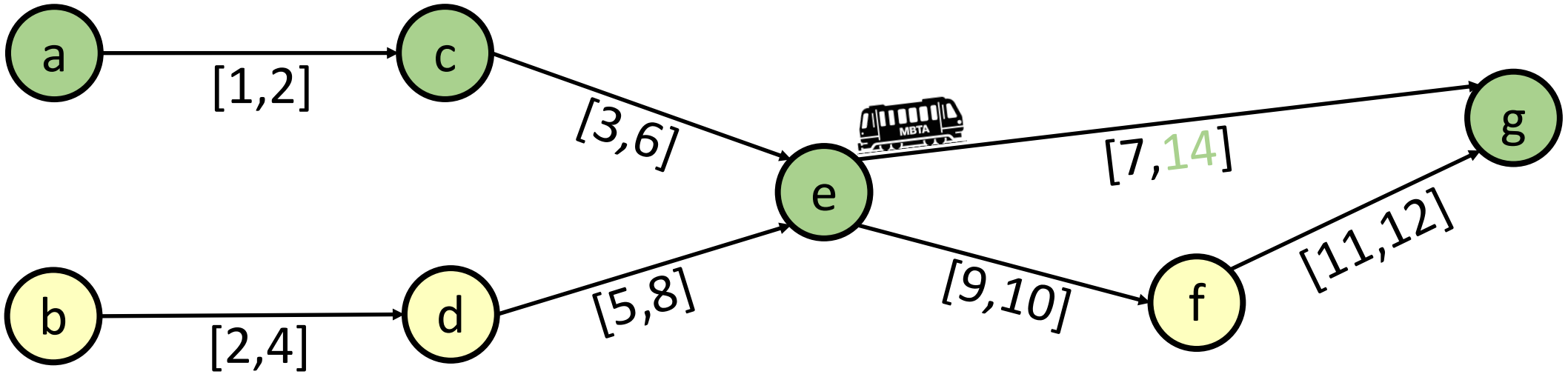
Example T. Minimal Path: Earliest Arrival



	a	b	c	d	e	f	g
u = a	1	∞	2	∞	6		

$[t_s, t_e] = [1,15]$

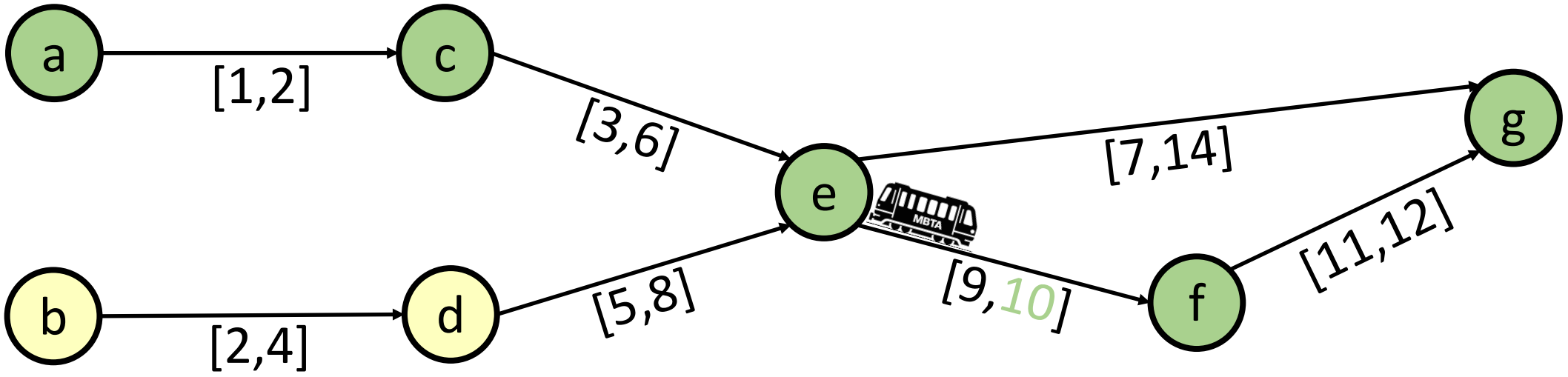
Example T. Minimal Path: Earliest Arrival



	a	b	c	d	e	f	g
u = a	1	∞	2	∞	6	∞	14

$[t_s, t_e] = [1,15]$

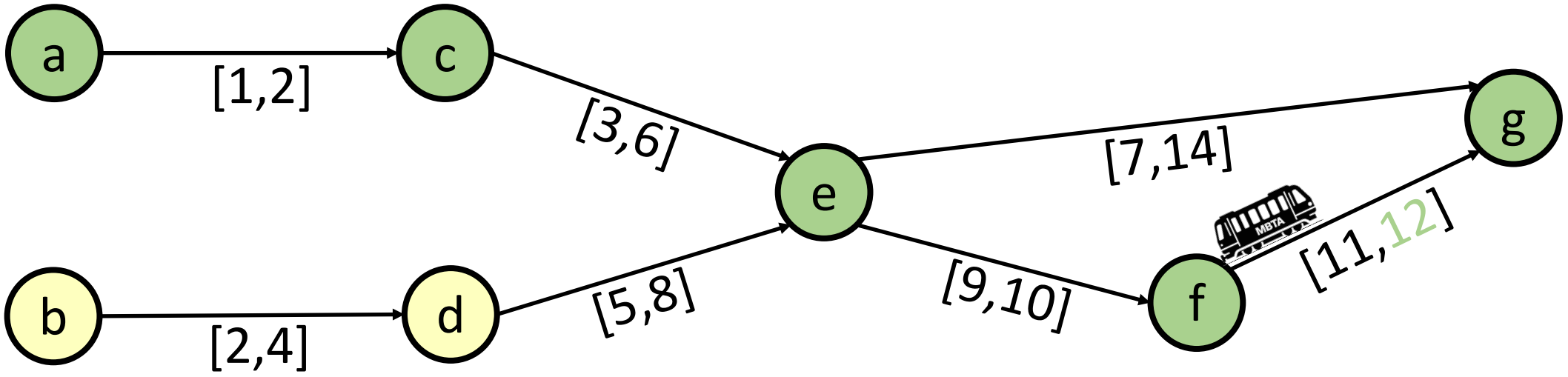
Example T. Minimal Path: Earliest Arrival



	a	b	c	d	e	f	g
u = a	1	∞	2	∞	6	10	14

$[t_s, t_e] = [1,15]$

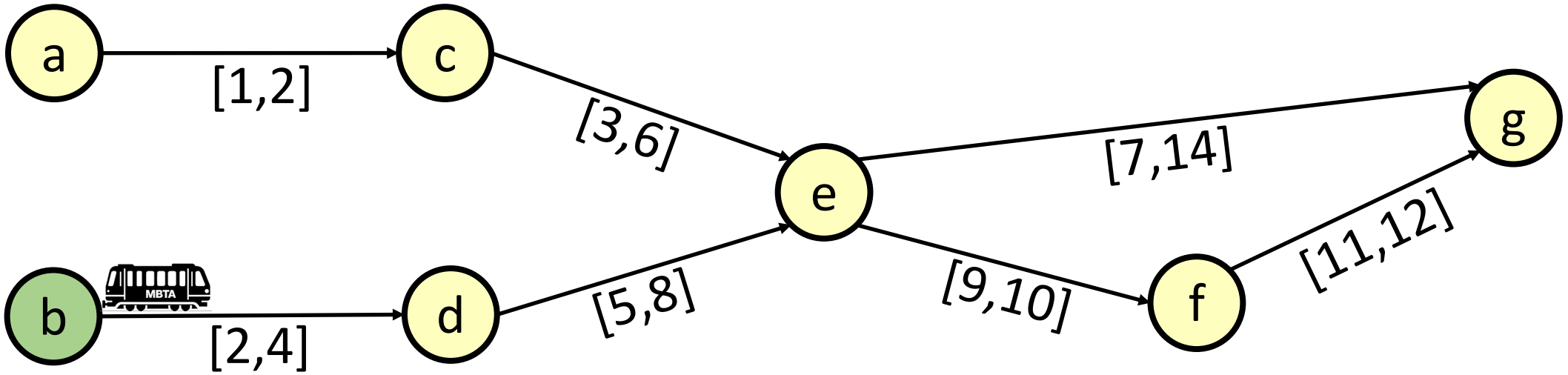
Example T. Minimal Path: Earliest Arrival



	a	b	c	d	e	f	g
u = a	1	∞	2	∞	6	10	12

$[t_s, t_e] = [1,15]$

Example T. Minimal Path: Earliest Arrival



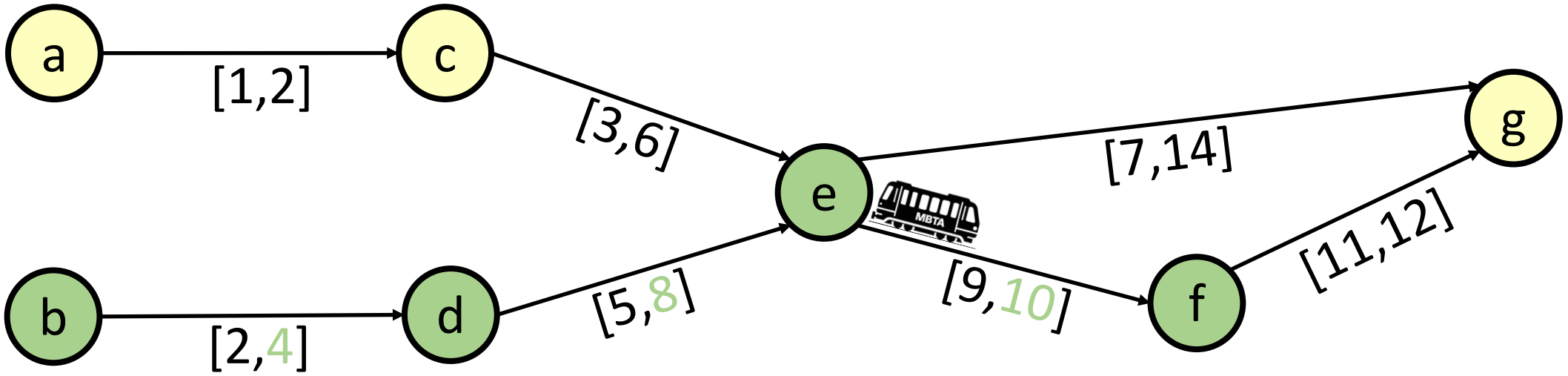
u = a	a	b	c	d	e	f	g
	1	∞	2	∞	6	10	12

$[t_s, t_e] = [1,15]$

u = **b**	a	**b**	c	d	e	f	g
	∞	**2**	∞	∞	∞	∞	∞

$[\mathbf{t_s}, t_e] = [\mathbf{2},11]$

Example T. Minimal Path: Earliest Arrival



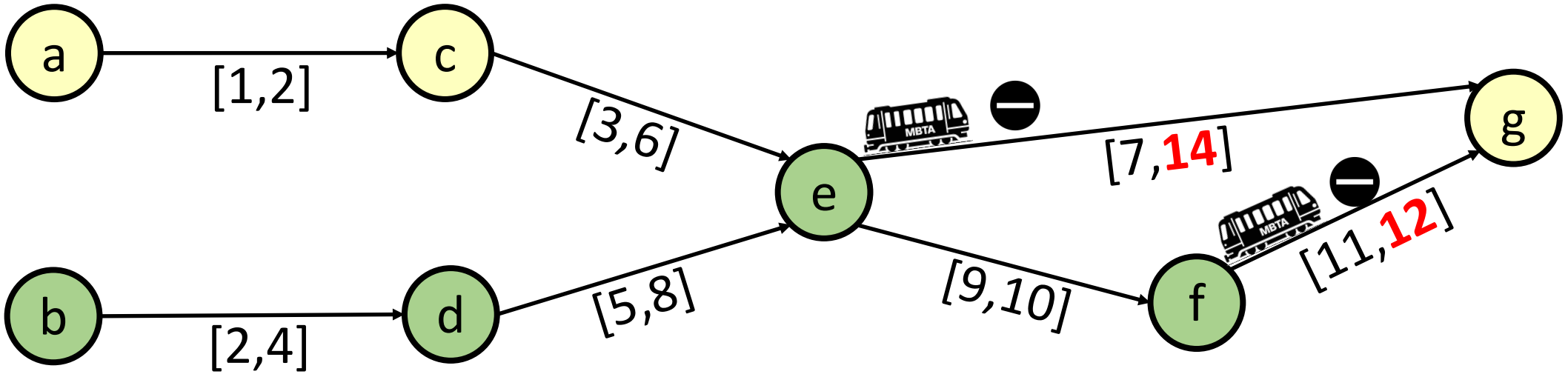
u = a	a	b	c	d	e	f	g
	1	∞	2	∞	6	10	12

$[t_s, t_e] = [1,15]$

u = b	a	b	c	d	e	f	g
	∞	2	∞	4	8	10	∞

$[t_s, \mathbf{t_e}] = [2,\mathbf{11}]$

Example T. Minimal Path: Earliest Arrival



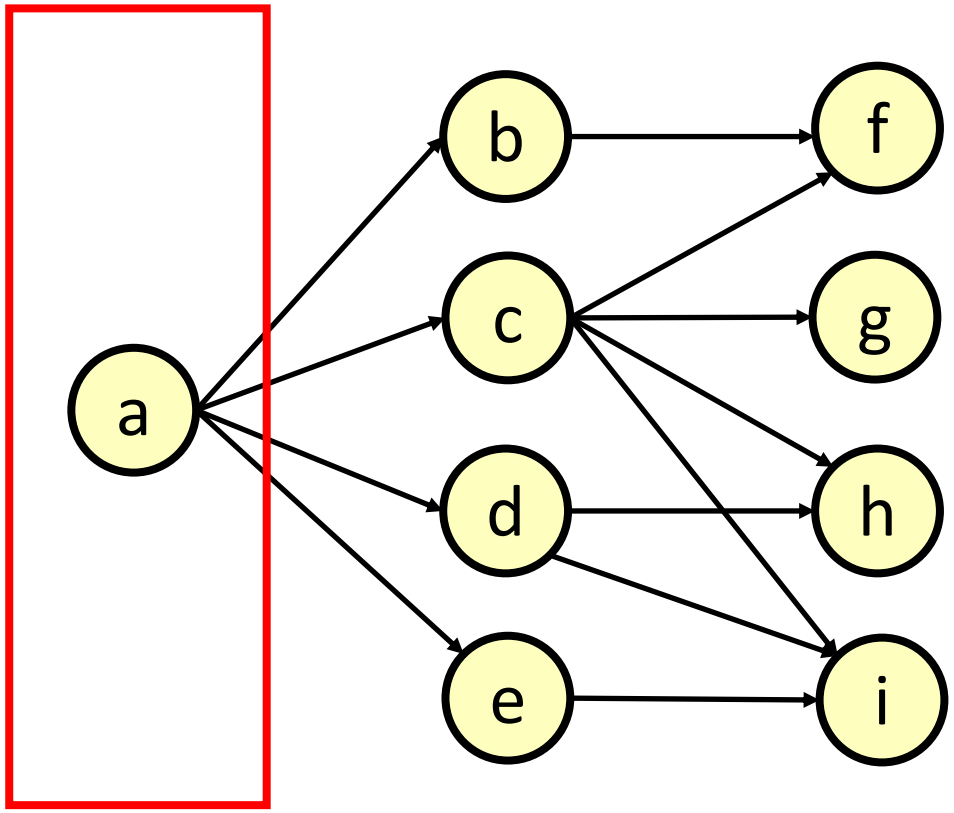
u = a	a	b	c	d	e	f	g
	1	∞	2	∞	6	10	12

$[t_s, t_e] = [1,15]$

u = b	a	b	c	d	e	f	g
	∞	2	∞	4	8	10	∞

$[t_s, \mathbf{t_e}] = [2,\mathbf{11}]$

Efficient parallel implementations



...

Extends **Ligra** to temporal setting:
- can process **each frontier in parallel**

...

`temporalEdgeMap``(vertexSet, …)`:
- applies fn to all vertices in each parallel frontier

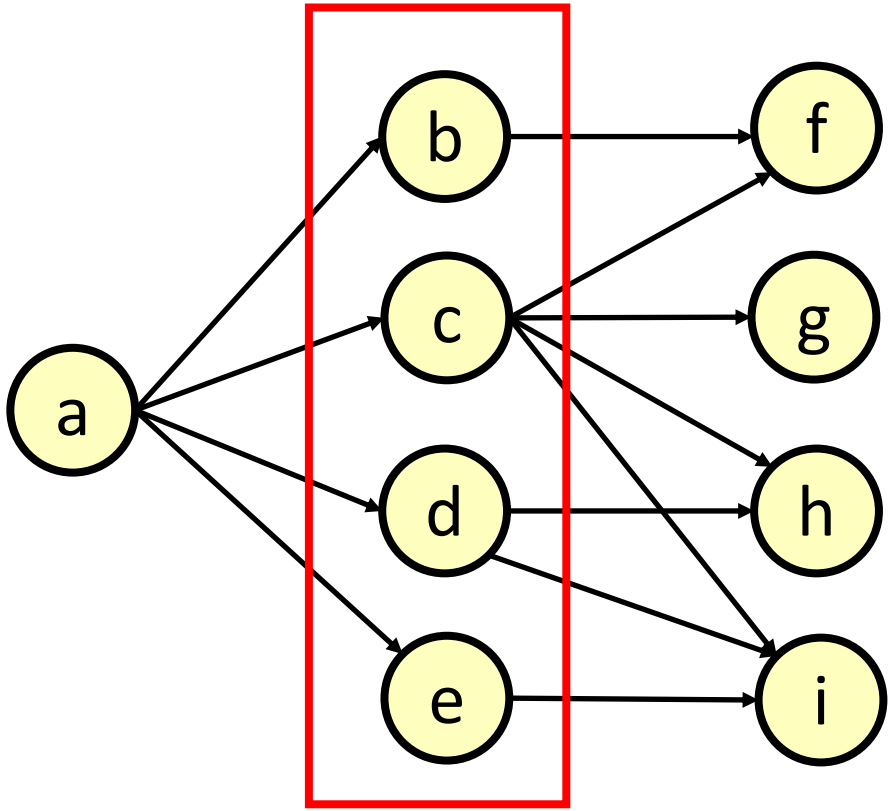
...

`update(temporalEdge):`
- applies fn to all nbors in each parallel frontier

...

see thesis for details

Efficient parallel implementations



…

Extends **Ligra** to temporal setting:
- can process **each frontier in parallel**

…

`temporalEdgeMap``(vertexSet, …)`:
- applies fn to all vertices in each parallel frontier

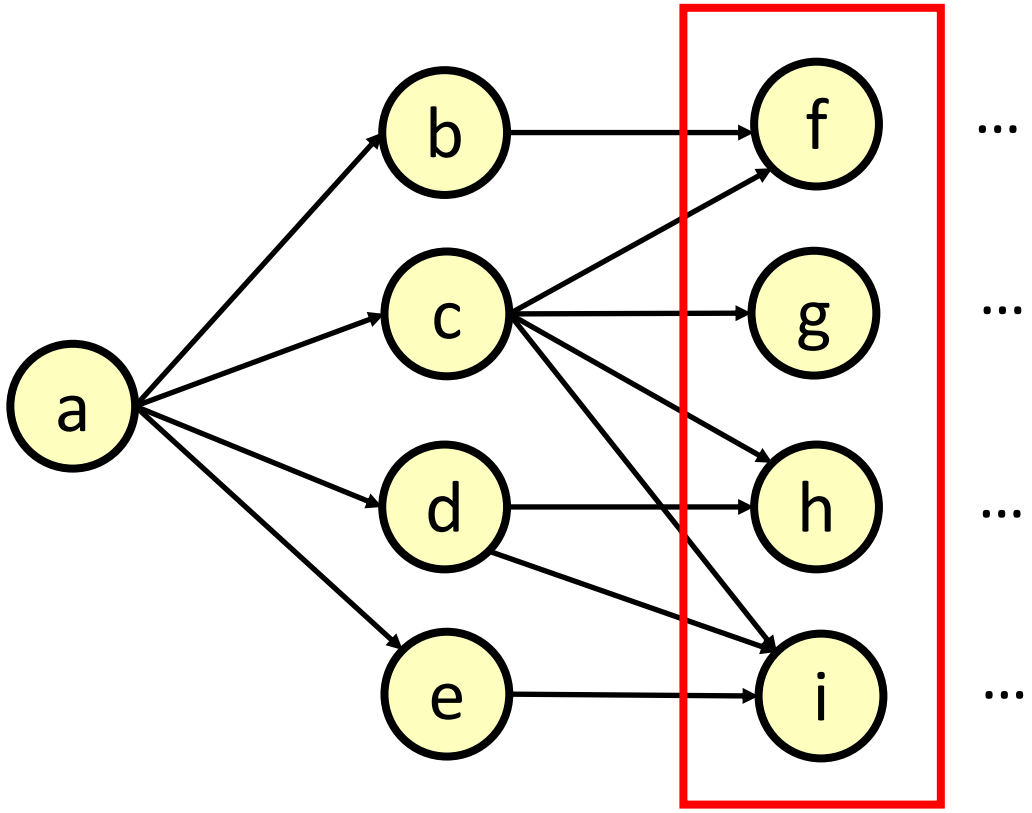
…

`update(temporalEdge):`
- applies fn to all nbors in each parallel frontier

…

see thesis for details

Efficient parallel implementations



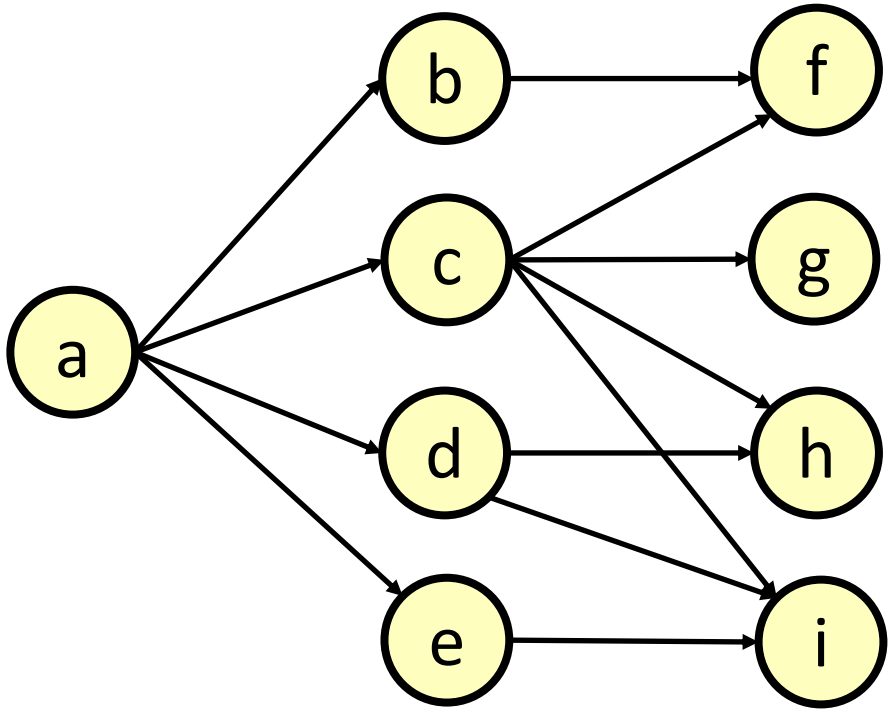
Extends **Ligra** to temporal setting:
- can process **each frontier in parallel**

`temporalEdgeMap``(vertexSet, …)`:
- applies fn to all vertices in each parallel frontier

`update(temporalEdge):`
- applies fn to all nbors in each parallel frontier

see thesis for details

Efficient parallel implementations



...

Extends **Ligra** to temporal setting:
- can process **each frontier in parallel**

...

`temporalEdgeMap``(vertexSet, …)`:
- applies fn to all vertices in each parallel frontier

...

`update(temporalEdge):`
- applies fn to all nbors in each parallel frontier

...

see thesis for details

Julian Shun *et al*, *"Ligra: A Lightweight Graph Processing for Shared Memory."*

Julian Shun *et al*, *"Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+"*

Kairos API extends Ligra to temporal setting

Interface	Description
VertexSet	Represents a subset of vertices $V' \subseteq V$
TemporalEdgeSet	Represents a subset of temporaledges $E' \subseteq E$.
OrderingPredicate(A: temporaledge, B: temporaledge, T : OrderingPredicate-Type): bool	Evaluates the order between two temporal edges based on one of the three ordering predicate types: Succeeds, StrictlySucceeds, or Overlaps
TemporalGraph(V: VertexSet, E: TemporalEdgeSet, P: OrderingPredicate)	Constructs a temporal graph using given input vertices, temporaledges, and ordering predicate.
VertexMap(U : VertexSet, G : TemporalGraph, F : vertex → bool): VertexSet	Applies $F(u)$ for each $u \in U$; returns a VertexSet$\{u \in U \mid F(u)$=true$\}$.
TemporalEdgeMap(U:TemporalEdgeSet, G:TemporalGraph, F:temporaledge → bool): TemporalEdgeSet	Applies $F(u)$ for each $u \in U$; returns a TemporalEdgeSet$\{u \in U \mid F(u)$=true$\}$.

Kairos: Key Technical Contributions

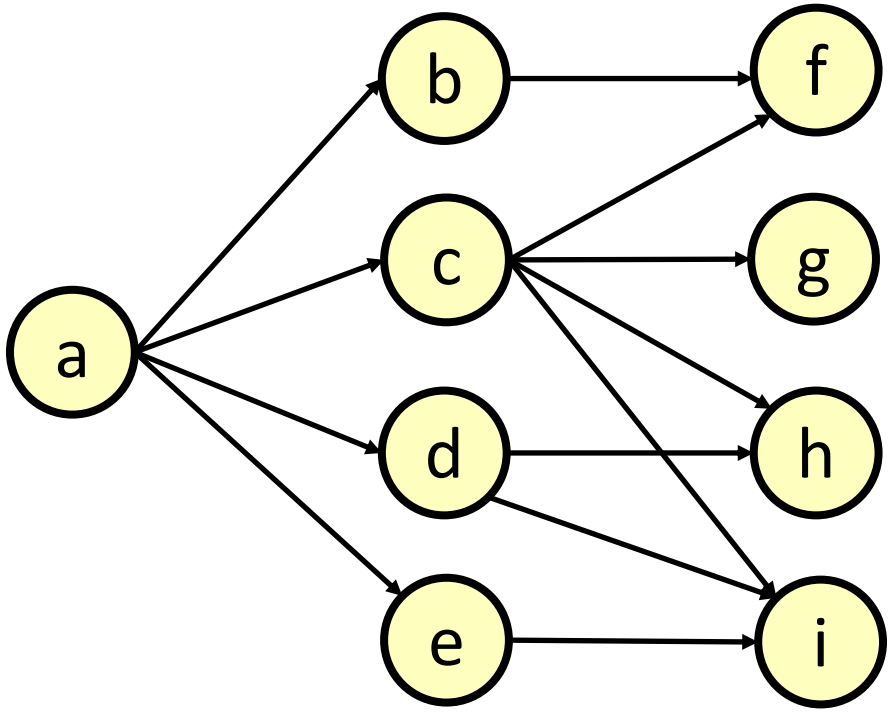
Efficient **parallel implementation of algorithms over temporal graphs**, e.g.

- minimal paths (earliest arr., latest dep., fastest, shortest duration)
- centrality (betweenness using fastest or shortest duration)
- k-core, ranking, and connectivity

Efficient data layouts

- **T-CSR: compressed sparse row** for temporal graphs
- **TGER**: highly optimized **parallel index** for temporal edges

Problem: efficient graph data layout

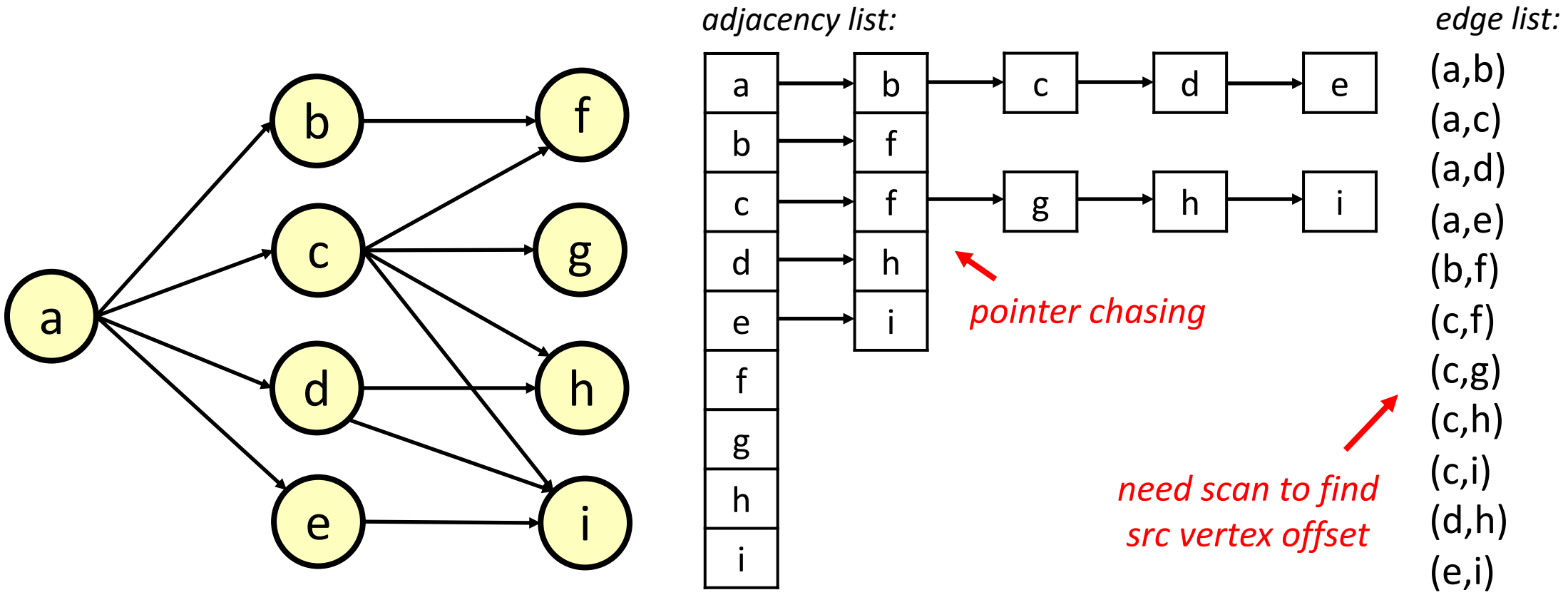


	a	b	c	d	e	f	g	h	i
a	0	**1**	**1**	**1**	**1**	0	0	0	0
b	0	0	0	0	0	**1**	0	0	0
c	0	0	0	0	0	**1**	**1**	**1**	**1**
d	0	0	0	0	0	0	0	**1**	**1**
e	0	0	0	0	0	0	0	0	**1**
f	0	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0

>85% zeros

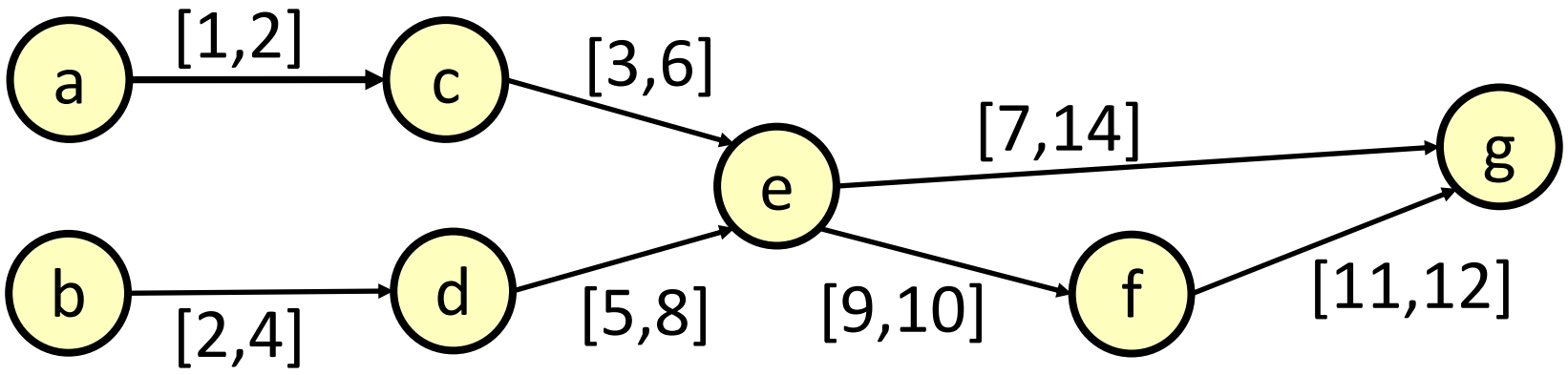
Adjacency matrix is not space efficient for sparse graphs

Problem: efficient graph data layout



Adjacency list and *edge list* use less space, but are not cache efficient

Solution: T. Compressed Sparse Row (T-CSR)



position where nbors start in the other arrays ↘

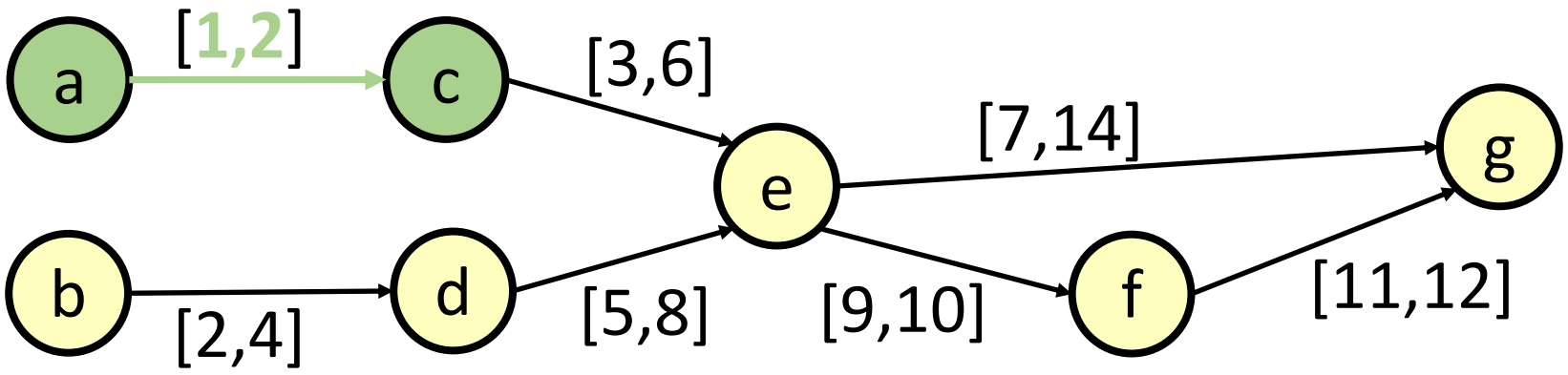
	a	b	c	d	e	f	g
offsets:	0	1	2	3	4	6	6

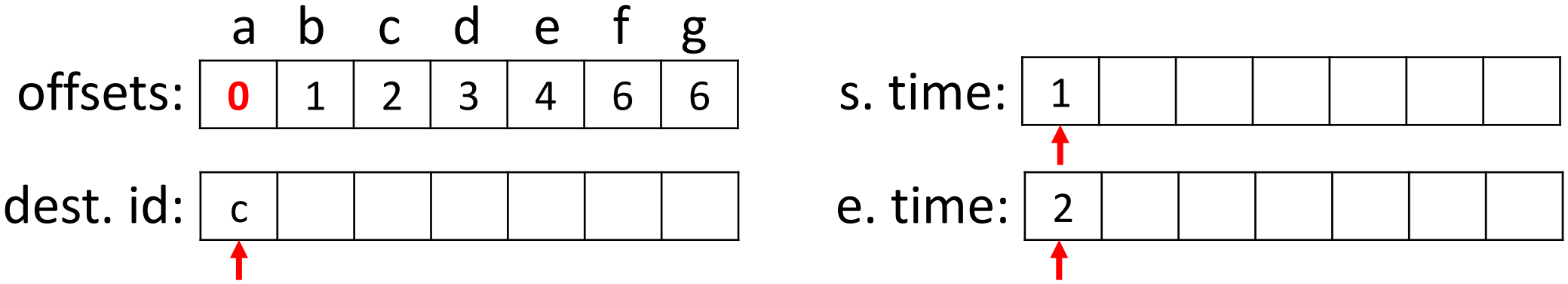
dest. id:							

s. time:							

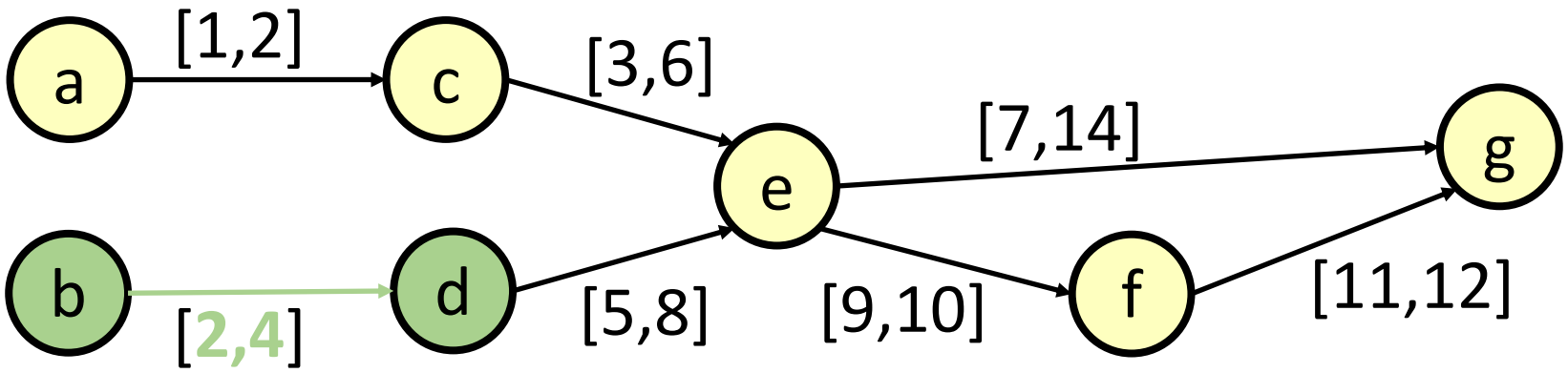
e. time:							

Solution: T. Compressed Sparse Row (T-CSR)





Solution: T. Compressed Sparse Row (T-CSR)



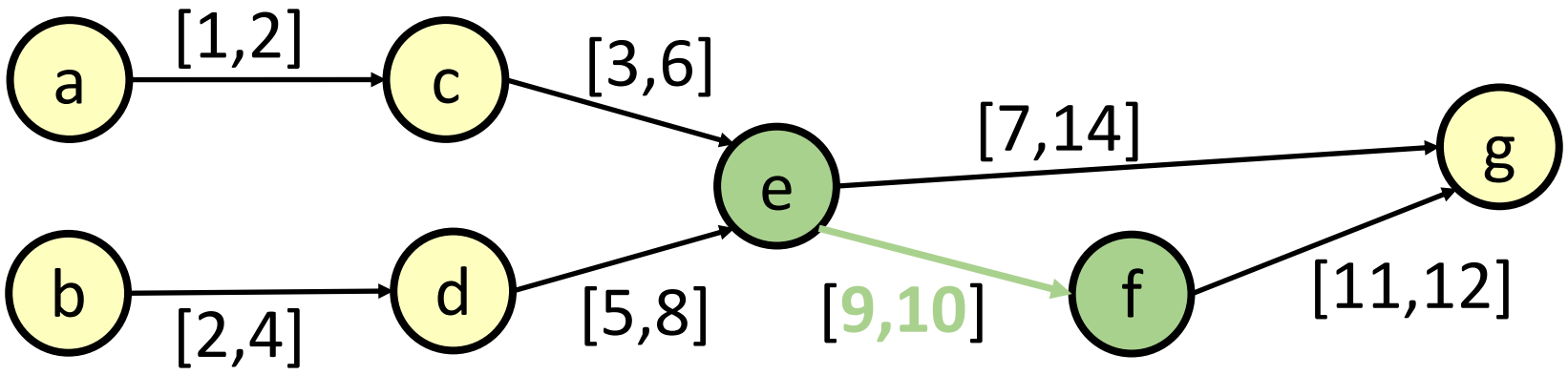
	a	b	c	d	e	f	g
offsets:	0	1	2	3	4	6	6

dest. id:	c	d					

s. time:	1	2					

e. time:	2	4					

Solution: T. Compressed Sparse Row (T-CSR)



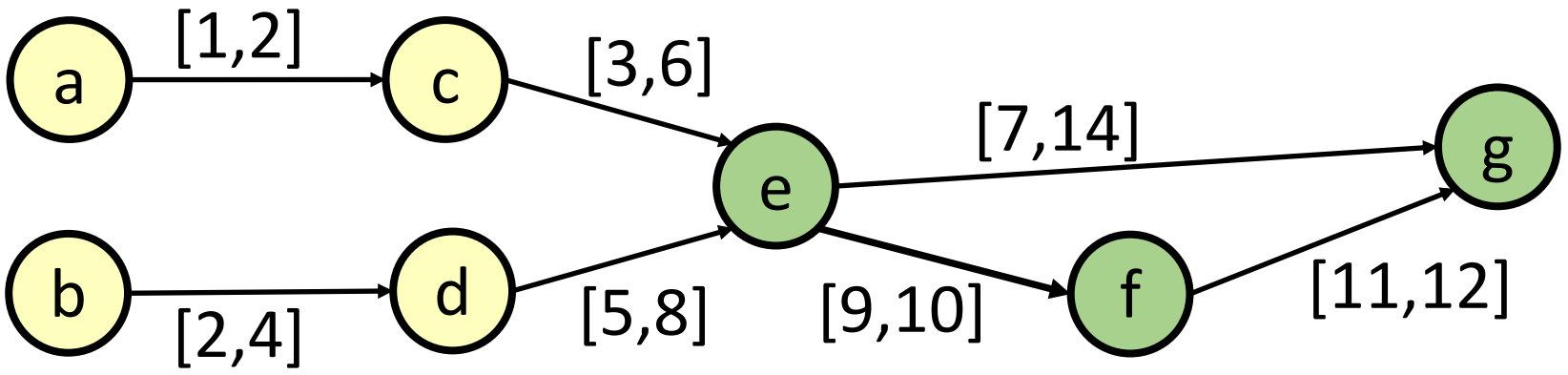
	a	b	c	d	e	f	g
offsets:	0	1	2	3	**4**	6	6

dest. id:	c	d	e	e	f		

s. time:	1	2	3	5	9		

e. time:	2	4	6	8	10		

Solution: T. Compressed Sparse Row (T-CSR)



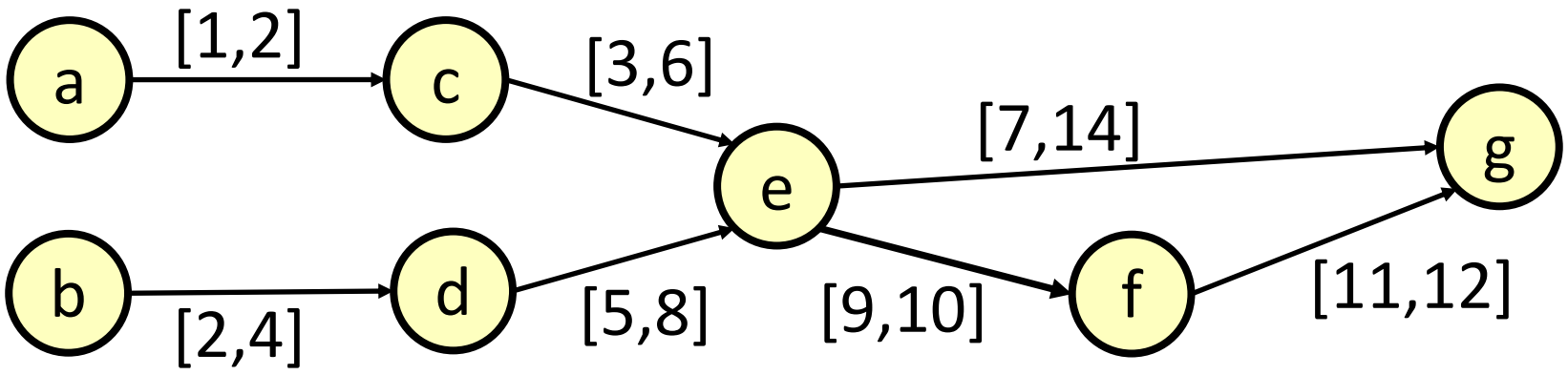
	a	b	c	d	e	f	g
offsets:	0	1	2	3	**4**	6	6

dest. id:	c	d	e	e	f	g	

s. time:	1	2	3	5	9	7	

e. time:	2	4	6	8	10	14	

Solution: T. Compressed Sparse Row (T-CSR)



	a	b	c	d	e	f	g
offsets:	0	1	2	3	4	6	6
dest. id:	c	d	e	e	f	g	g

s. time:	1	2	3	5	9	7	11
e. time:	2	4	6	8	10	14	12

Problem: T-CSR still requires linear scan

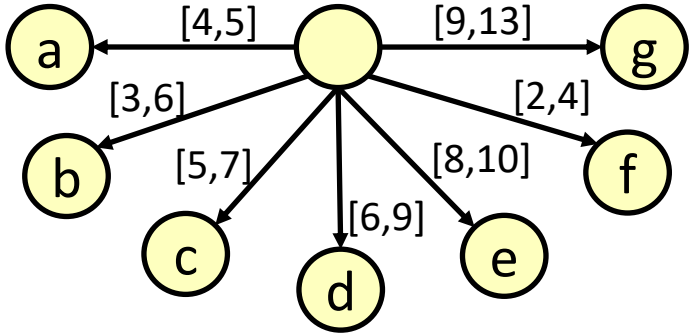
Parallel scan starting at vertex offset is cache efficient, but still O(*degree*)

In real-world graphs, single vertex degree can be O(Millions)

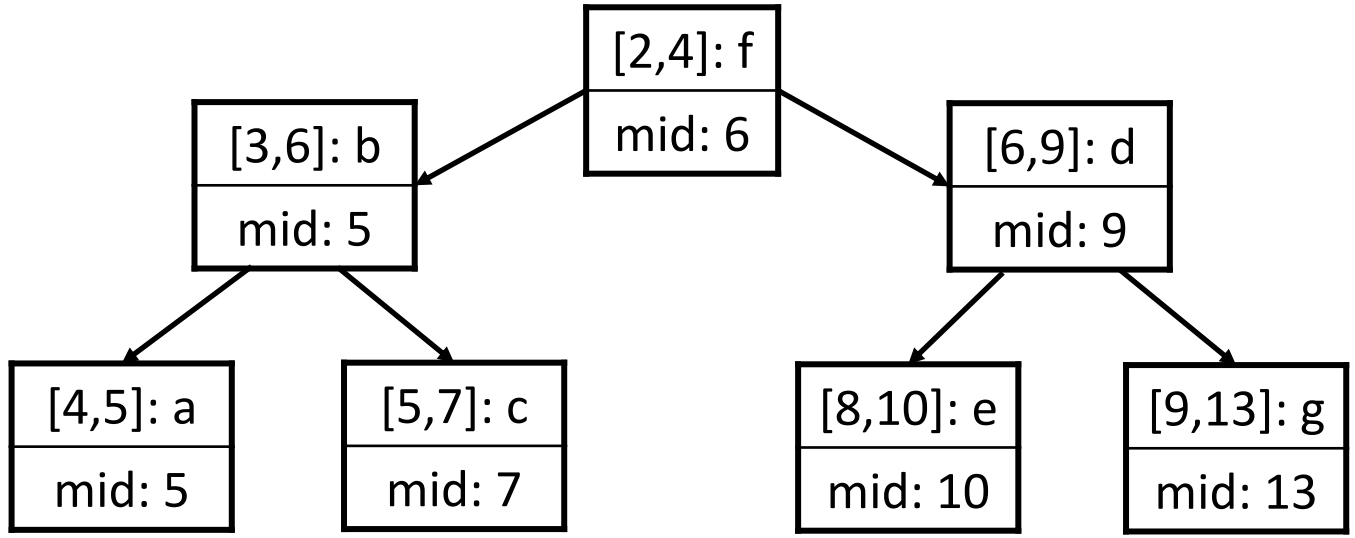
Name	$\|V\|$	$\|E\|$	$\max_{v \in V} deg^+(v)$	$\max_{v \in V} deg^-(v)$	$avg_{v \in V} deg(v)$
bitcoin [37]	4.80×10^7	1.13×10^8	2.66×10^6	2.53×10^6	4
netflow [38]	3.72×10^8	1.20×10^9	5.78×10^5	1.82×10^8	12
reddit-reply [37]	1.17×10^7	6.46×10^8	3.92×10^5	9.94×10^5	110
stackoverflow [39]	6.02×10^6	6.34×10^7	1.01×10^5	93143	20
transportation [40]	41794	7.93×10^7	50881	50625	3798
twitter-cache [41]	94226	8.55×10^8	5.21×10^6	2.31×10^5	36328
synthetic	10^7	10^9	5.65×10^7	3.08×10^7	99

Solution: T. Graph Edge Registry (TGER)

input vertex out-neighbors:

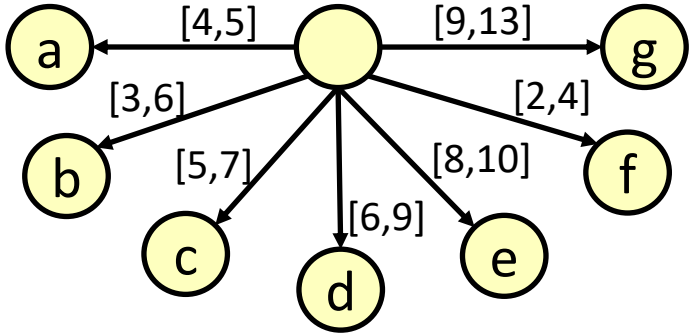


Corresponding TGER:

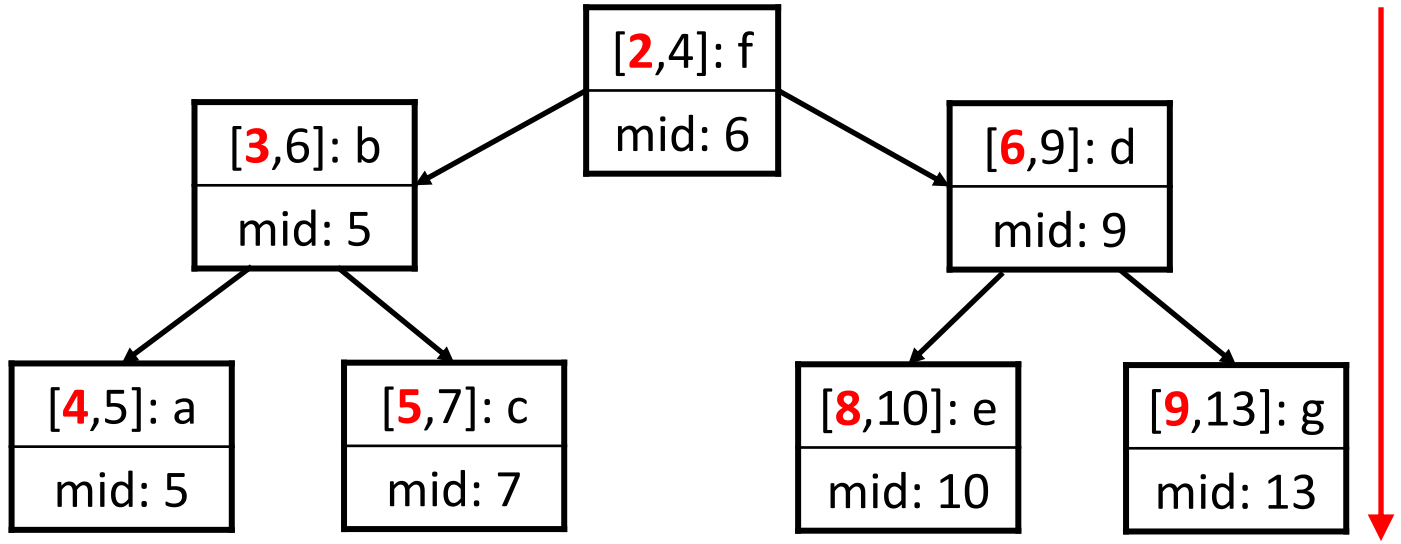


Solution: T. Graph Edge Registry (TGER)

input vertex out-neighbors:



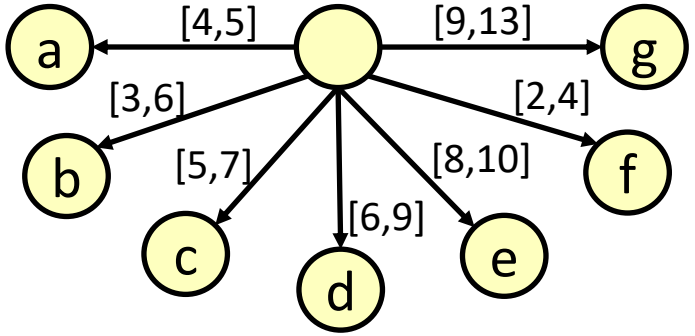
Corresponding TGER:



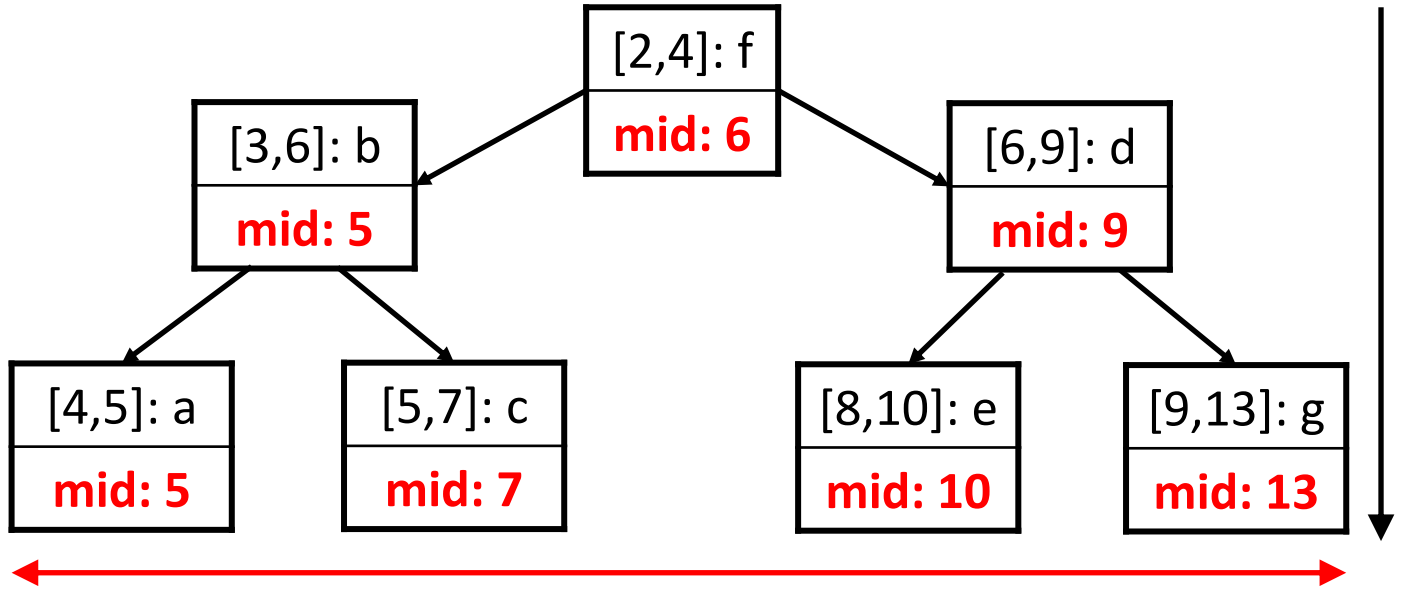
acts as heap on start times

Solution: T. Graph Edge Registry (TGER)

input vertex out-neighbors:



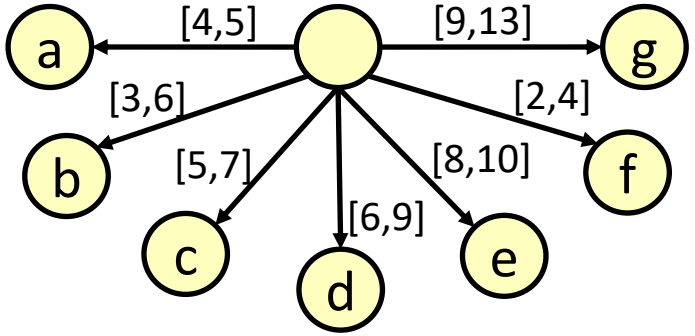
Corresponding TGER:



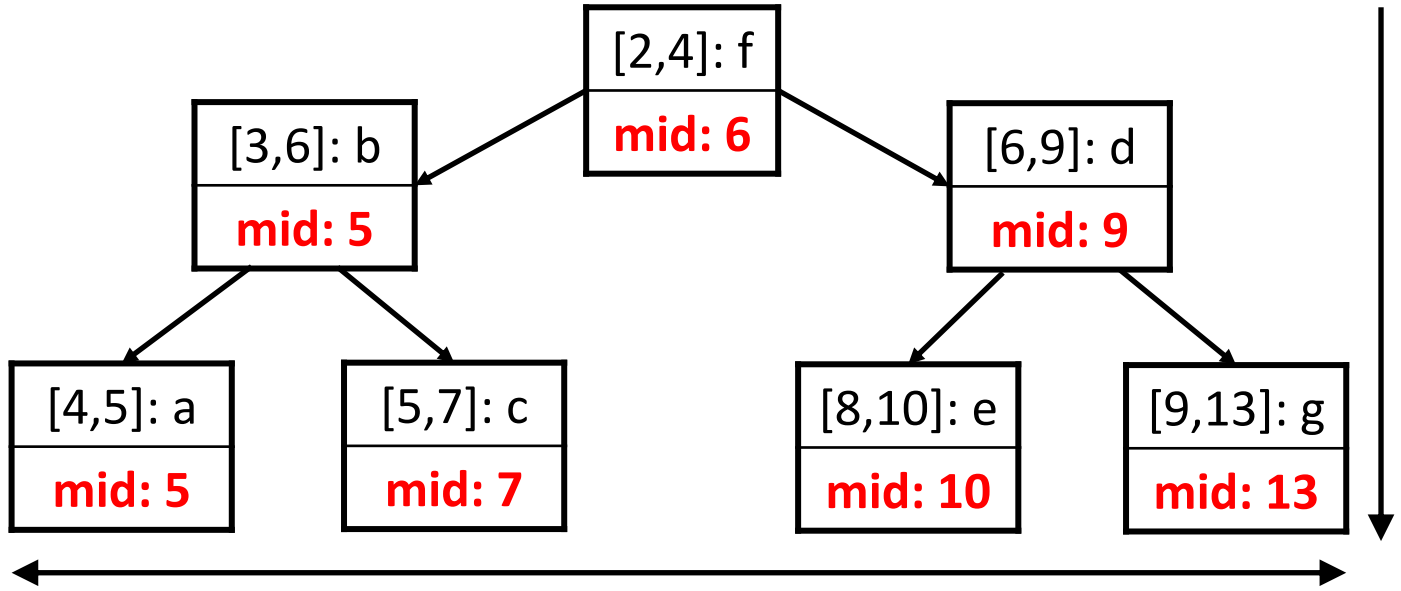
acts as heap on start times
acts as BST on end times

Solution: T. Graph Edge Registry (TGER)

input vertex out-neighbors:



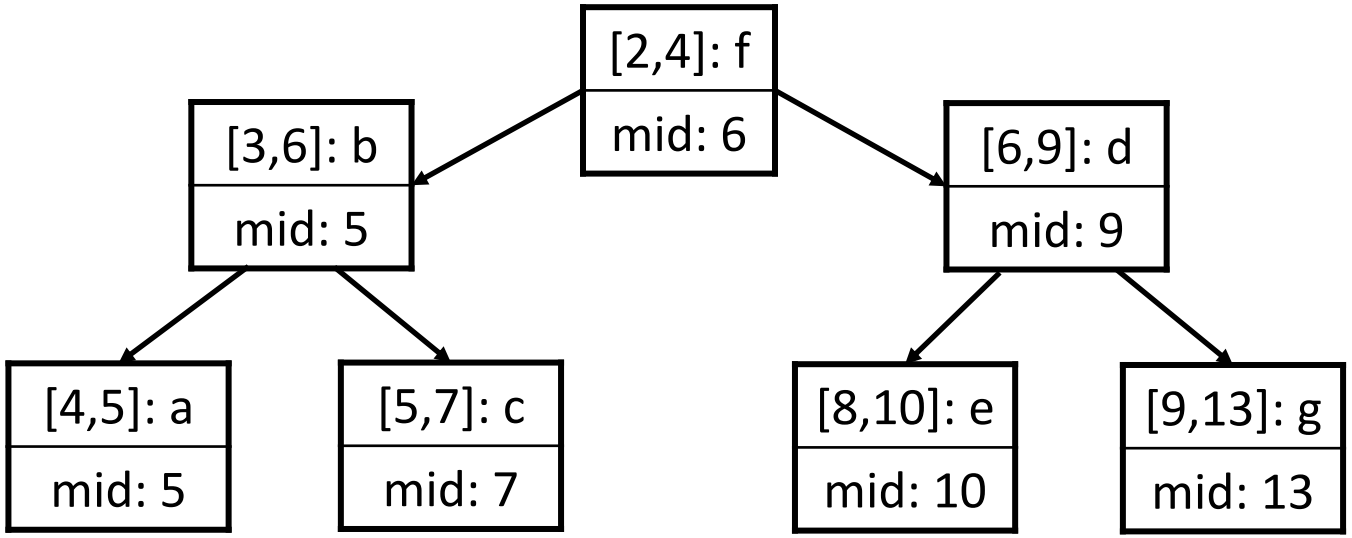
Corresponding TGER:



acts as heap on start times
acts as BST on end times

parallel construction (see thesis for details):
- parallel sort edges by *start time*, root is first element
- recurse: split subtrees by left and right of median *end time*

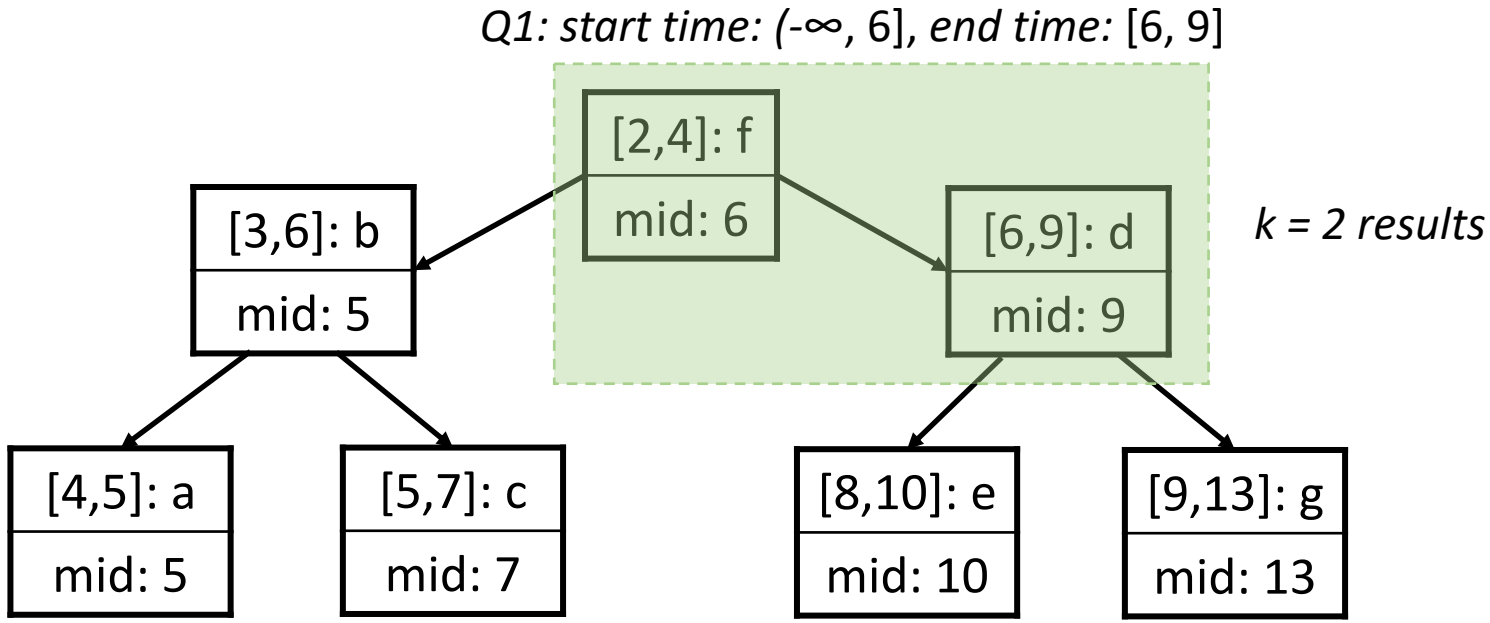
TGER supports 3-sided queries in log(n) + k



3-sided:

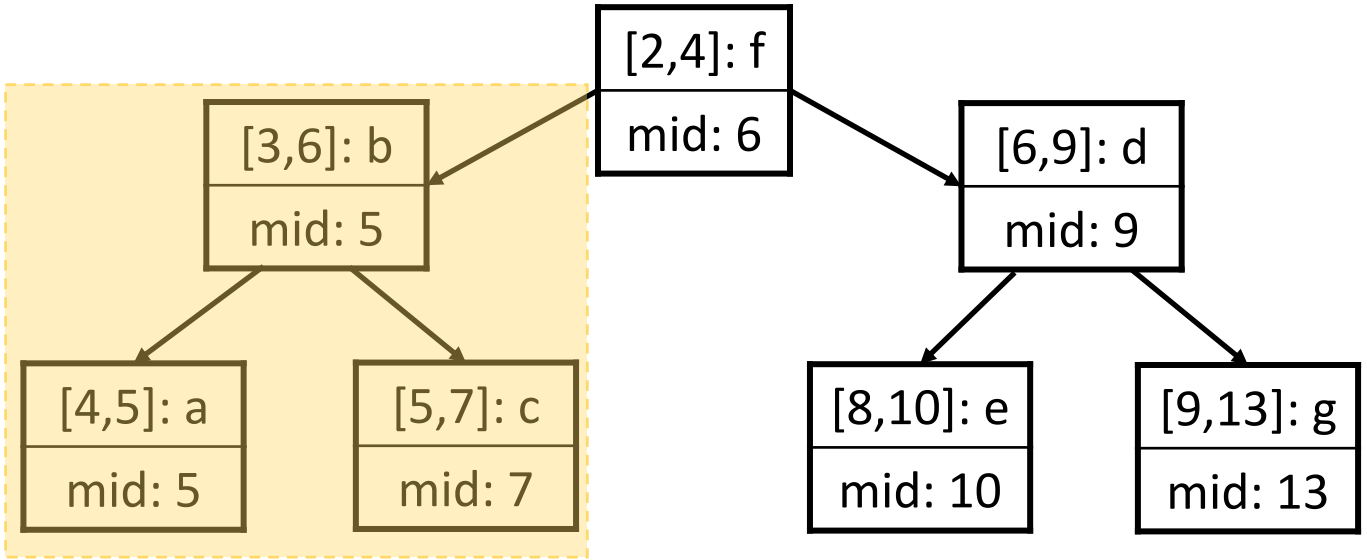
- 1-side (upper range) for heap dimension
- 2-sides (lower + upper range) for BST dimension

TGER supports 3-sided queries in log(n) + k



3-sided:
- 1-side (upper range) for heap dimension
- 2-sides (lower + upper range) for BST dimension

TGER supports 3-sided queries in log(n) + k



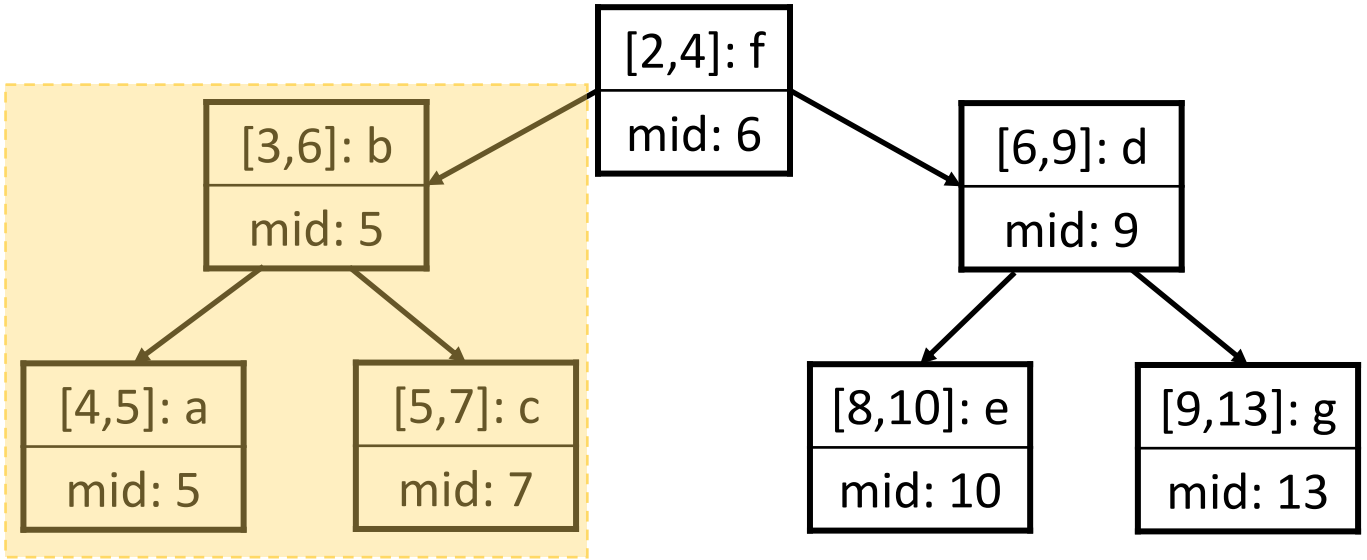
k = 3 results

Q2: start time: (-∞, 5], *end time:* [5, 8]

3-sided:

- 1-side (upper range) for heap dimension
- 2-sides (lower + upper range) for BST dimension

TGER supports 3-sided queries in log(n) + k



k = 3 results

Q2: start time: (-∞, 5], *end time:* [5, 8]

3-sided:
- 1-side (upper range) for heap dimension
- 2-sides (lower + upper range) for BST dimension

parallel query processing (see thesis for details):
- per worker: local array stores results
- final merge: prefix sum

Problem: scan vs index trade-off

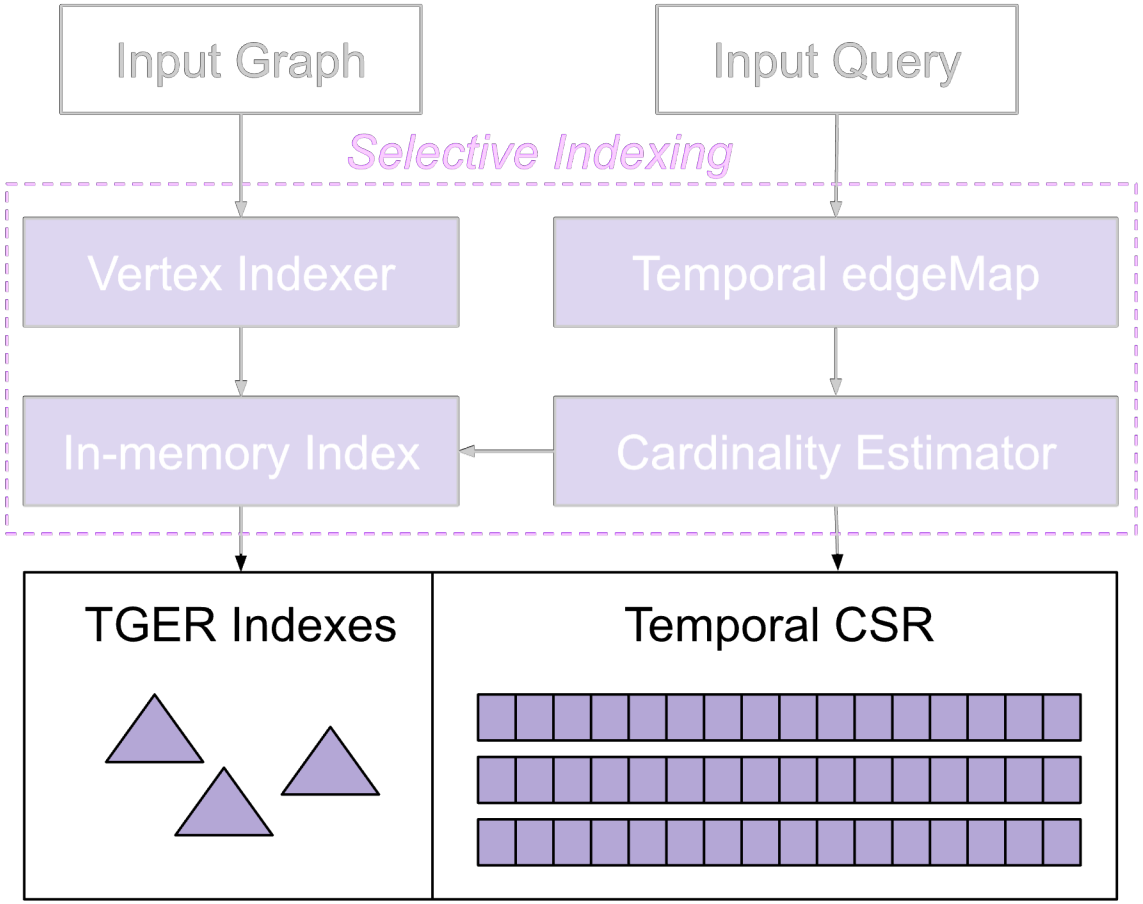
T-CSR

- Good parallelism: scan over arrays
- O(n) query complexity: not great when n is large

TGER

- O(log n + k) query complexity: beats T-CSR when k <<< n
- Tree-like data structure: parallelism is not as efficient

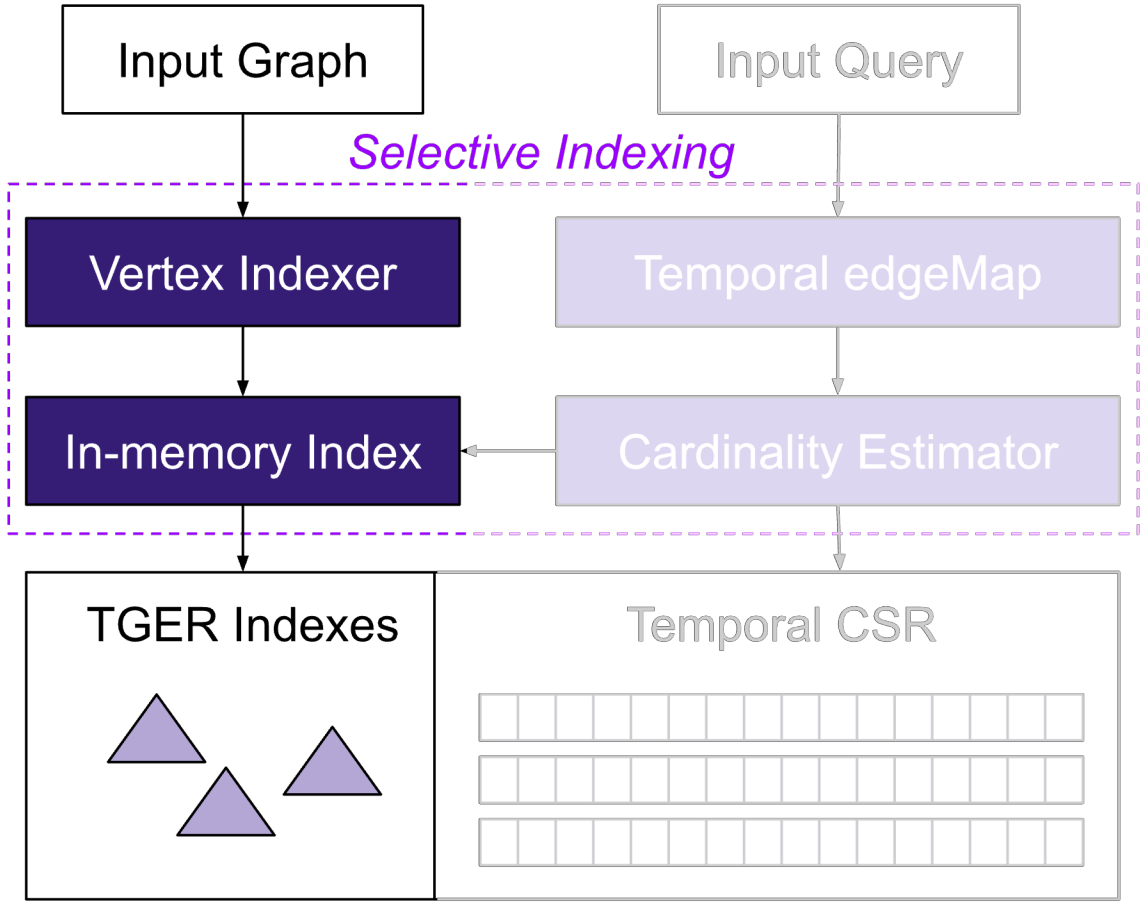
Solution: Selective Indexing



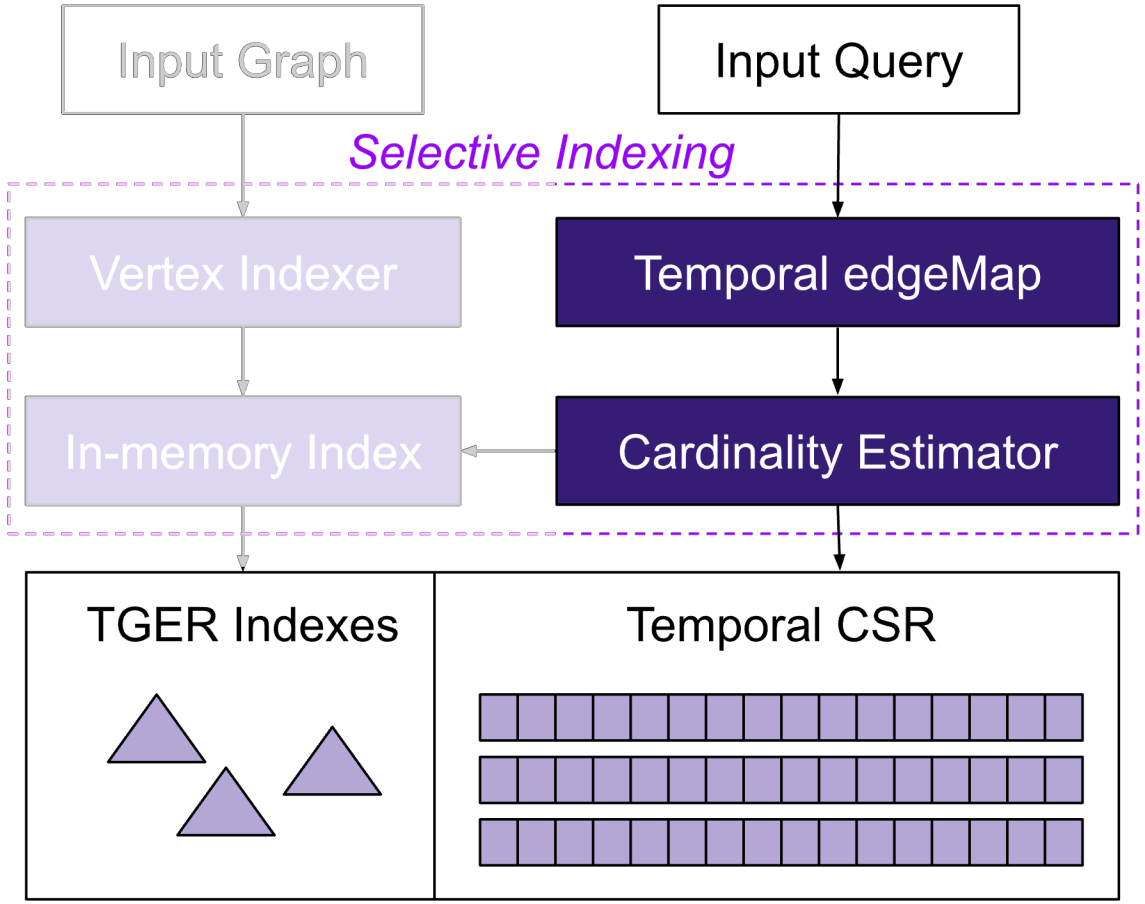
Two data representations for
in-memory temporal edges

Solution: Selective Indexing

Offline: build index for large vertices



Solution: Selective Indexing



At runtime, cost model guides access method decision (index vs scan)

Selective Indexing: Cost Model

$$T_v = c \cdot [\log(deg(v)) + k]$$

cost of accessing nbors using TGER

$$S_v = c' \cdot deg(v)$$

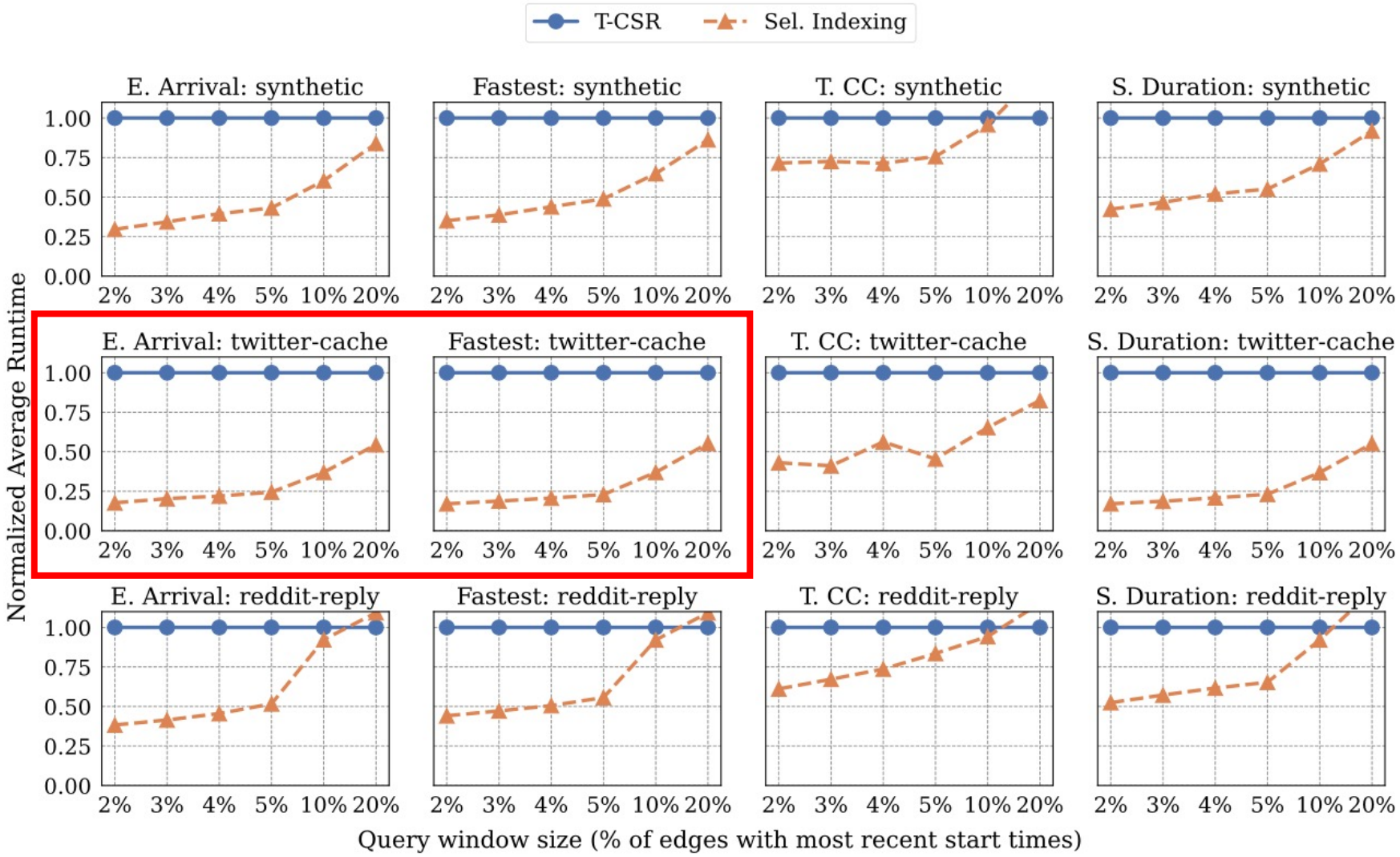
cost of accessing nbors using T-CSR

$$C_v = \begin{cases} T_v & \text{if } \beta \leq \theta_{\text{sel}} \\ S_v & \text{otherwise} \end{cases}$$

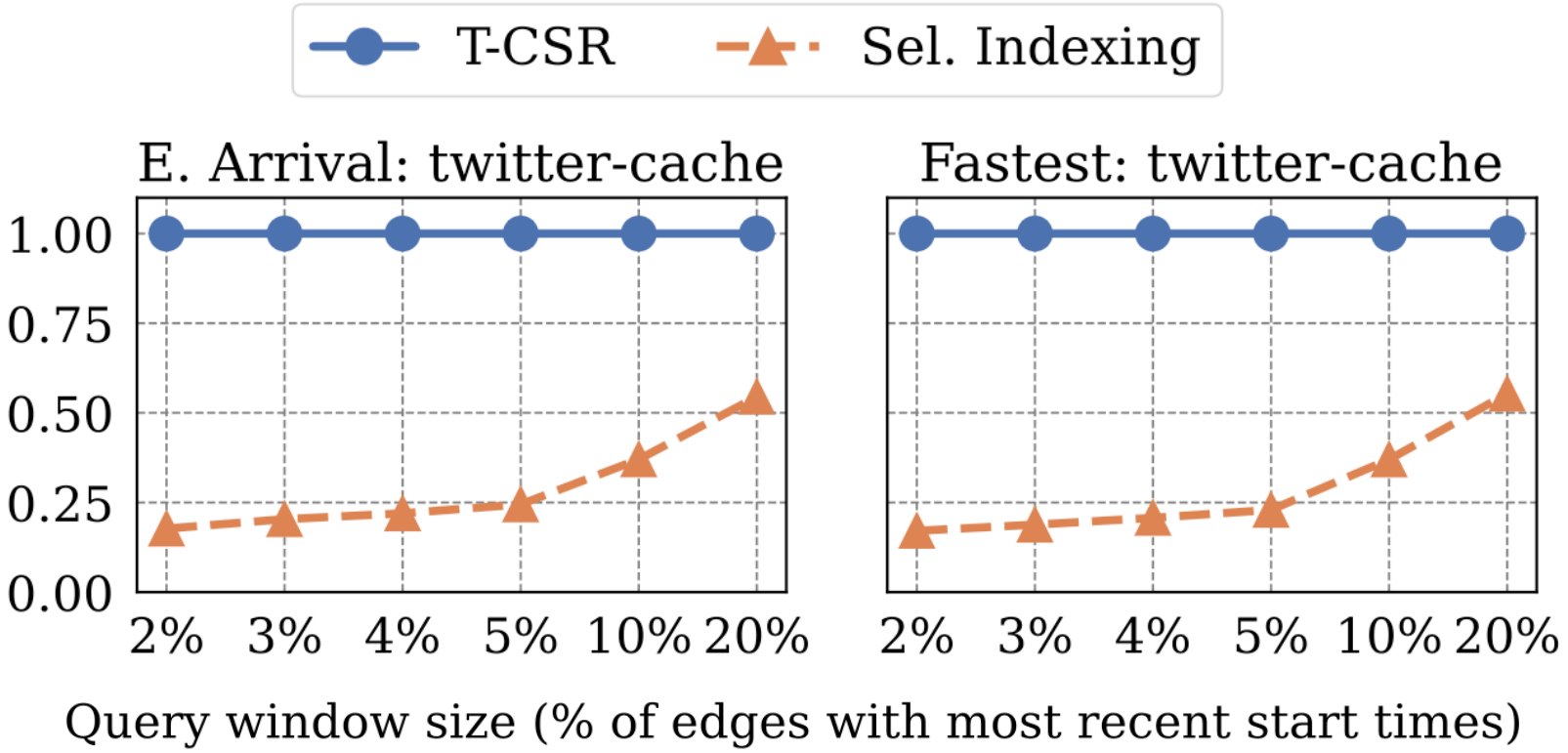
Impacting factors

- Vertex degree distribution
- Temporal edges distribution
- Query workload (e.g., selectivity)

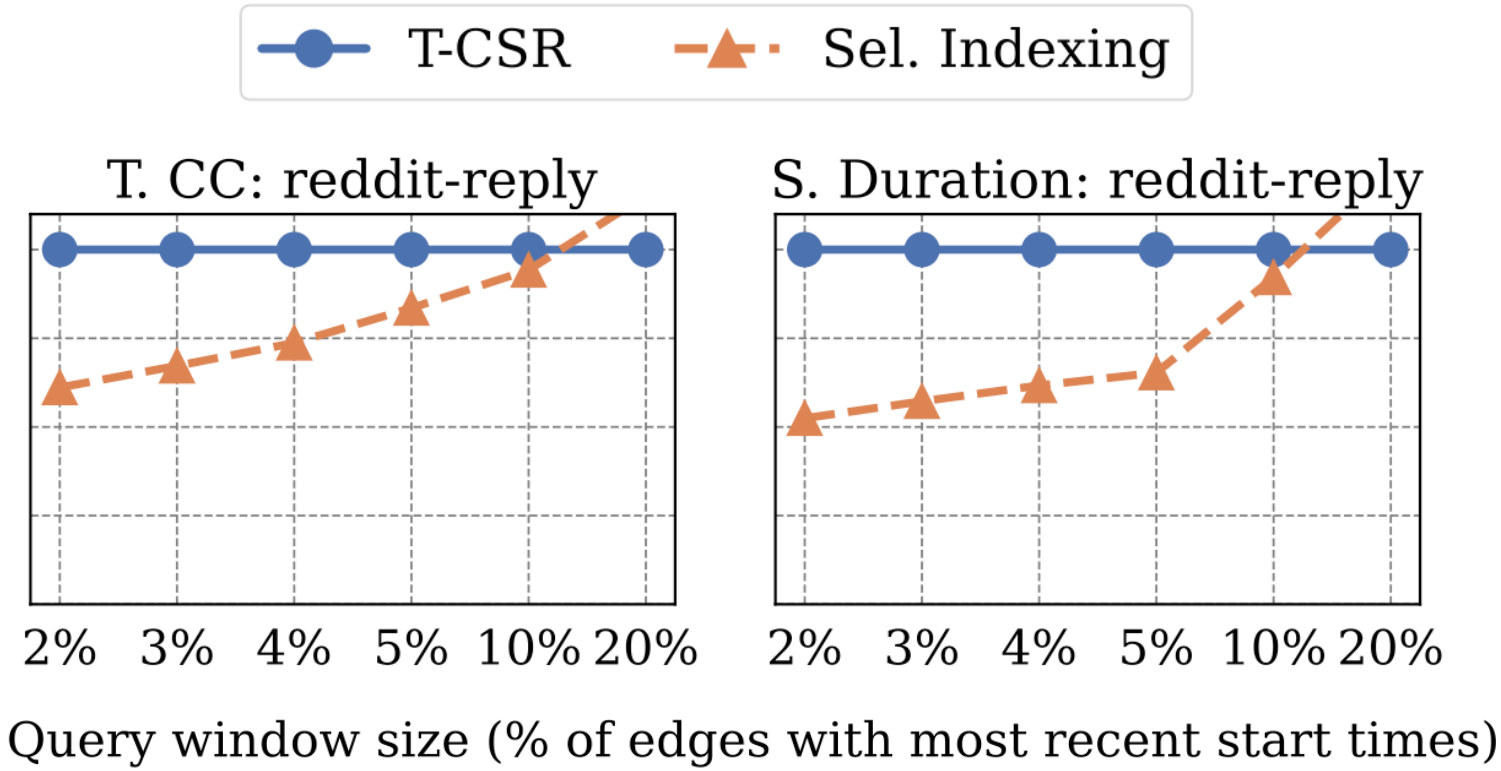
Experimental Results (see thesis for more)



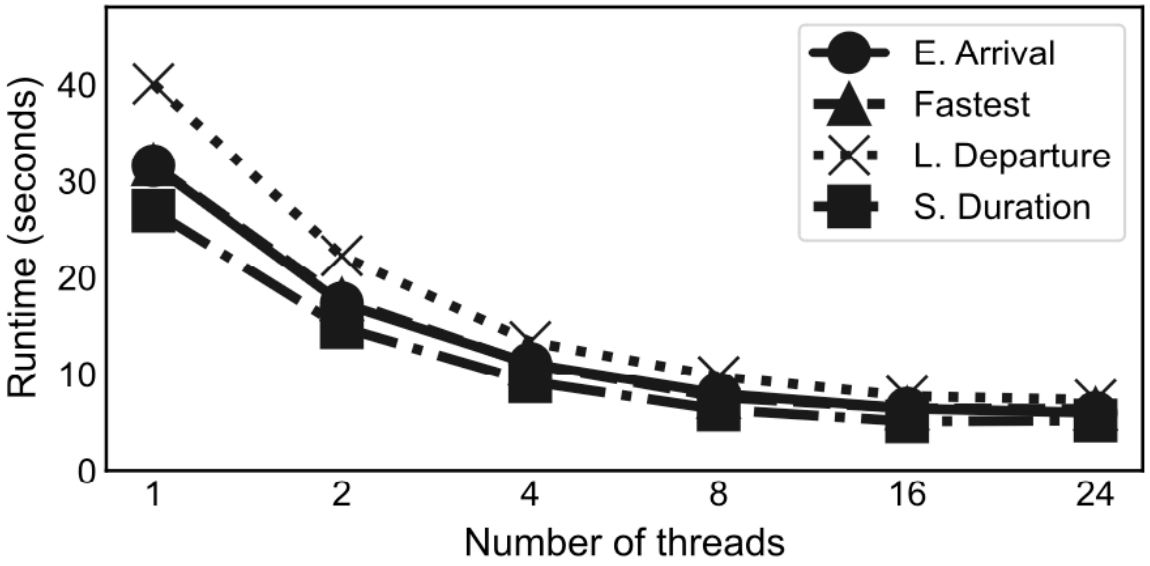
Experimental Results (see thesis for more)



Experimental Results (see thesis for more)



Experimental Results (see thesis for more)



500M edges synthetic data set (aggressively skewed)

O(secs): orders of magnitude faster than distributed engines

Kairos Contributions

Selective Indexing chooses best access method to retrieve vertex's nbors

TGER: highly optimized **parallel index** for temporal edges

Efficient parallel implementation of algorithms over temporal graphs

Good performance gains

- Orders of magnitude vs distributed alternatives for graphs that fit within a single server
- T-CSR baseline already good, but TGER is better for highly selective queries

Other contributions

- Work w/ Intel collaborators: Optane NVRAM for storing TGER indexes
- Open-sourced T-CSR baseline last year; open source remaining after paper acceptance

Future Work

Kaskade

- Expand view types
- Incremental view maintenance + automated view management

Kairos

- TGER and T-CSR dynamism: support edge removal / insertion
- Data compression + adaptive indexing instead of binary decision
- Distributed processing: requires skew-aware partitioning

Integrate both systems

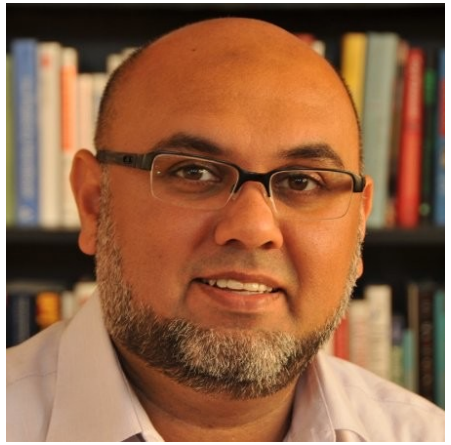
- Temporal views: time-respecting path contractions can lead to view explosion
- Selective indexing naturally extends to non-temporal settings

Acknowledgements (Sam and Julian)

Acknowledgements (Committee + Collabs)

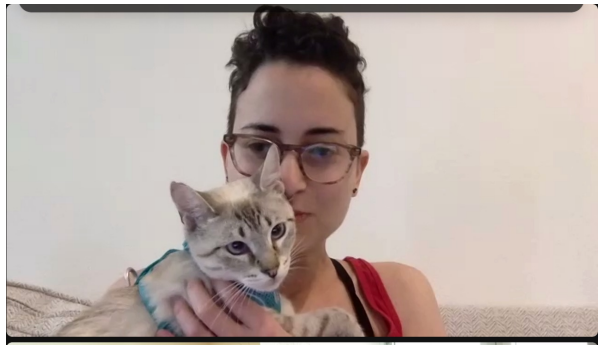




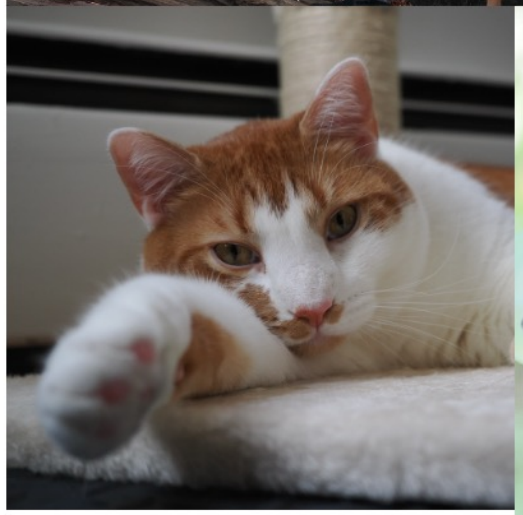
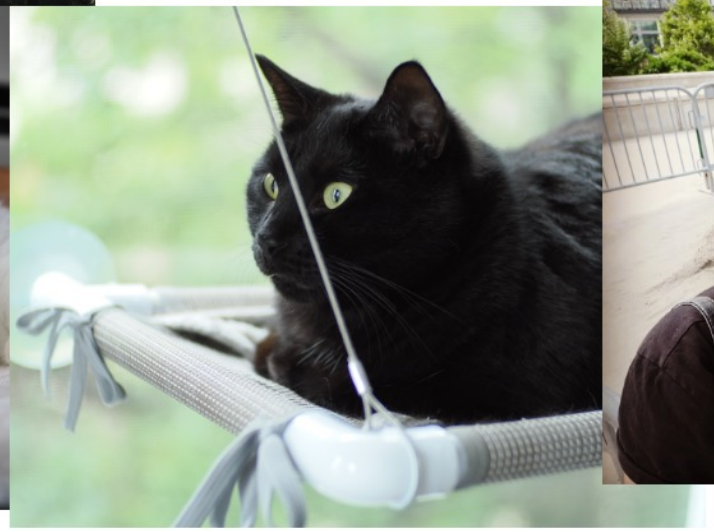

Acknowledgements (MIT DSG)

Acknowledgements (MIT + Friends)



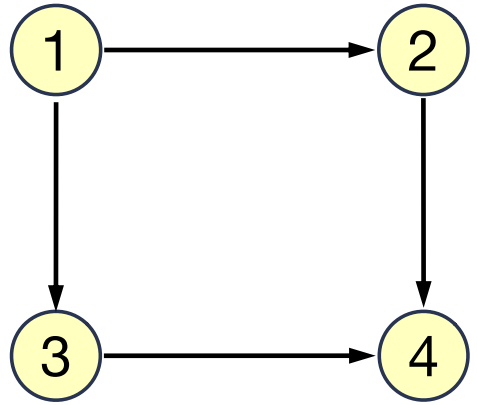
Acknowledgements (Family)




THANK YOU!

Problem formulation: finding graph views as a satisfiability problem



Logical clause example (Prolog syntax):

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), path(Z, Y).
```

Facts extracted from input queries and schema are axioms (*constraints*).

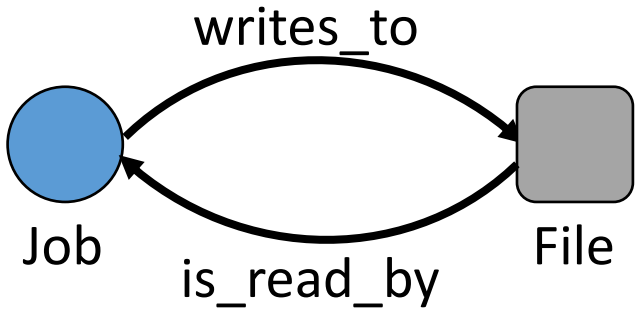
```
edge(1, 2). edge(1, 3). edge(2, 4). edge(3, 4).
```

Graph view enumerators (*templates*) are encoded as logical clauses that are **satisfiable iff view is feasible** given input constraints.

A single template can **enumerate multiple views**, but only most beneficial are chosen, e.g., smaller in size, useful to more than one query, etc.

Constraint-based view enumeration: Query and schema fact extraction

Schema:



Query:

```
MATCH
(j1:Job)-[:writes_to]->(f1:File)
(f1:File)-[r*0..8]->(f2:File)
(f2:File)-[:is_read_by]->(j2:Job)
RETURN j1, j2
```

Explicit facts extracted from graph's schema:

```
schemaVertex('Job'). schemaVertex('File').
schemaEdge('Job', 'File', 'writes_to').
schemaEdge('File', 'Job', 'is_read_by').
```

Explicit facts extracted from query's graph pattern:

```
queryVertex(j1). queryVertexType(j1, 'Job').
queryVertex(f1). queryVertexType(f1, 'File').
queryVertex(f2). queryVertexType(f2, 'File').
queryVertex(j2). queryVertexType(j1, 'Job').
queryEdge(j1, f1).
queryEdgeType(j1, f1, 'writes_to').
queryEdge(f2, j2).
queryEdgeType(f2, j2, 'is_read_by').
queryVariableLengthPath(f1, f2, 0, 8).
```

Constraint-based view enumeration: Inferring view candidates

Each **view template** yields **set of candidate views** that are feasible given input schema and query constraints.

Explicit facts extracted from graph's schema:

```
schemaVertex('Job'). schemaVertex('File').
schemaEdge('Job', 'File', 'writes_to').
schemaEdge('File', 'Job', 'is_read_by').
```

Explicit facts extracted from query's graph pattern:

```
queryVertex(j1). queryVertexType(j1, 'Job').
queryVertex(f1). queryVertexType(f1, 'File').
queryVertex(f2). queryVertexType(j1, 'File').
queryVertex(j2). queryVertexType(j1, 'Job').
queryEdge(j1, f1).
queryEdgeType(j1, f1, 'writes_to').
queryEdge(f2, j2).
queryEdgeType(f2, j2, 'is_read_by').
queryVariableLengthPath(f1, f2, 0, 8).
```

Feasible assignments for example view template "kHopConnectorSameVertexType":

```
(X='j1', Y='j2', XTYPE='Job', K=2)
(X='j1', Y='j2', XTYPE='Job', K=4)
(X='j1', Y='j2', XTYPE='Job', K=6)
(X='j1', Y='j2', XTYPE='Job', K=8)
(X='j1', Y='j2', XTYPE='Job', K=10)
```

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

Number of potential k-length paths

$$\hat{E}(G,k) = \boxed{\binom{n}{k+1}} \cdot \left[\frac{m}{\binom{n}{2}}\right]^k$$

Example: expected number of k-length simple paths in an Erdos-Renyi random graph

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

Probability that it is an actual path

$$\hat{E}(G,k) = \binom{n}{k+1} \cdot \left[\frac{m}{\binom{n}{2}}\right]^k$$

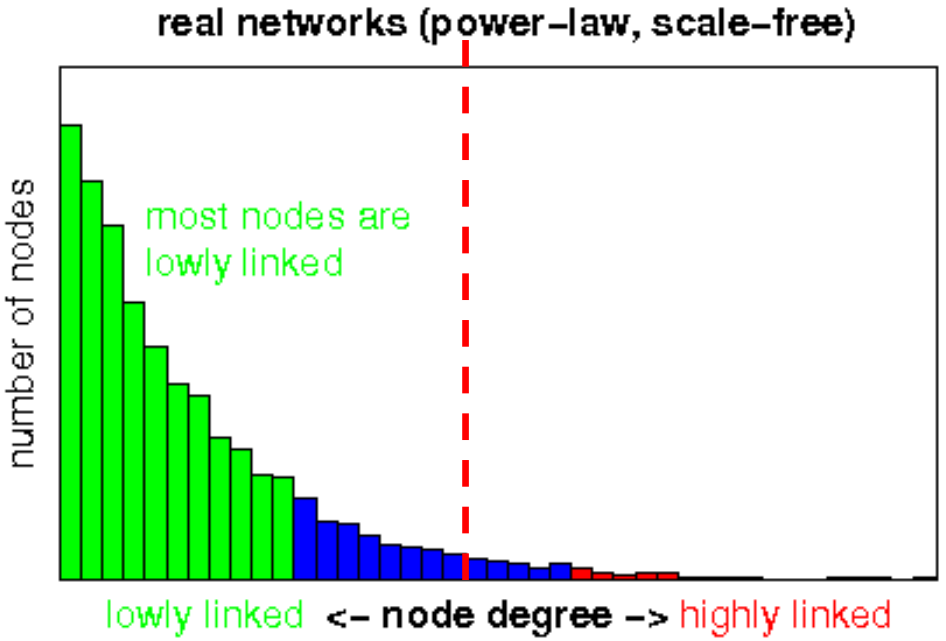
Example: expected number of k-length simple paths in an Erdos-Renyi random graph

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

Parameterized by percentile out-degree

$$\hat{E}(G, k, \alpha) = n \cdot deg_{\alpha}^{k}$$

Example: estimating number of directed k-length paths in *homogeneous* graph



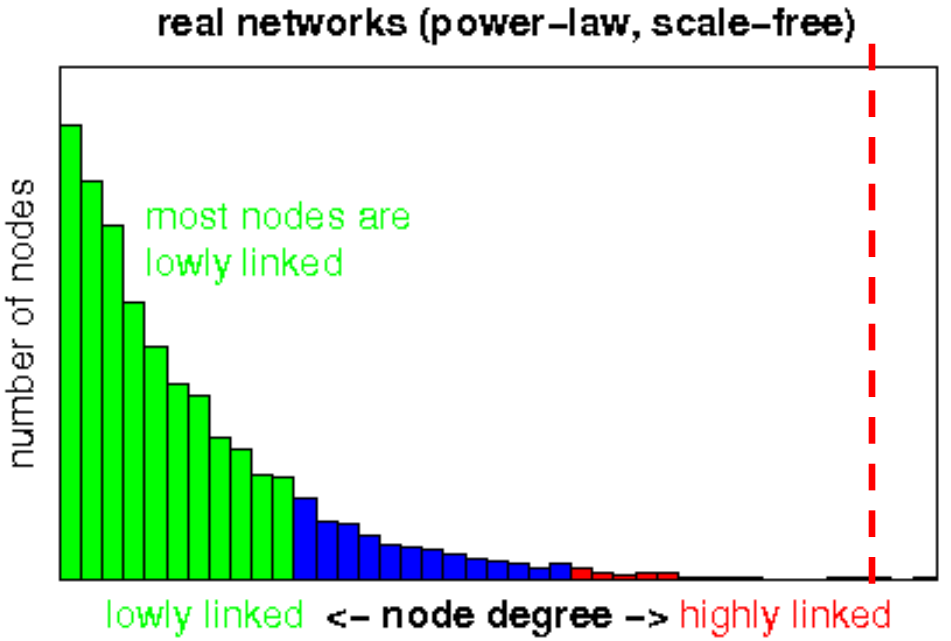
e.g., α = 50 (median)

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

Parameterized by percentile out-degree

$$\hat{E}(G, k, \alpha) = n \cdot deg_{\alpha}^{k}$$

Example: estimating number of directed k-length paths in *homogeneous* graph



e.g., α = 95 (95th %ile)

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k$$

Example: size estimator for k-hop connector views

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k$$

For each type of vertex that can be an edge source

Example: size estimator for k-hop connector views

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

How many vertices of that type do we have?

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k$$

Example: size estimator for k-hop connector views

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k$$

Structural Property: out-degree percentile

Example: size estimator for k-hop connector views

Cost Model for Graph Views: Structural Properties Guide View Size Estimation

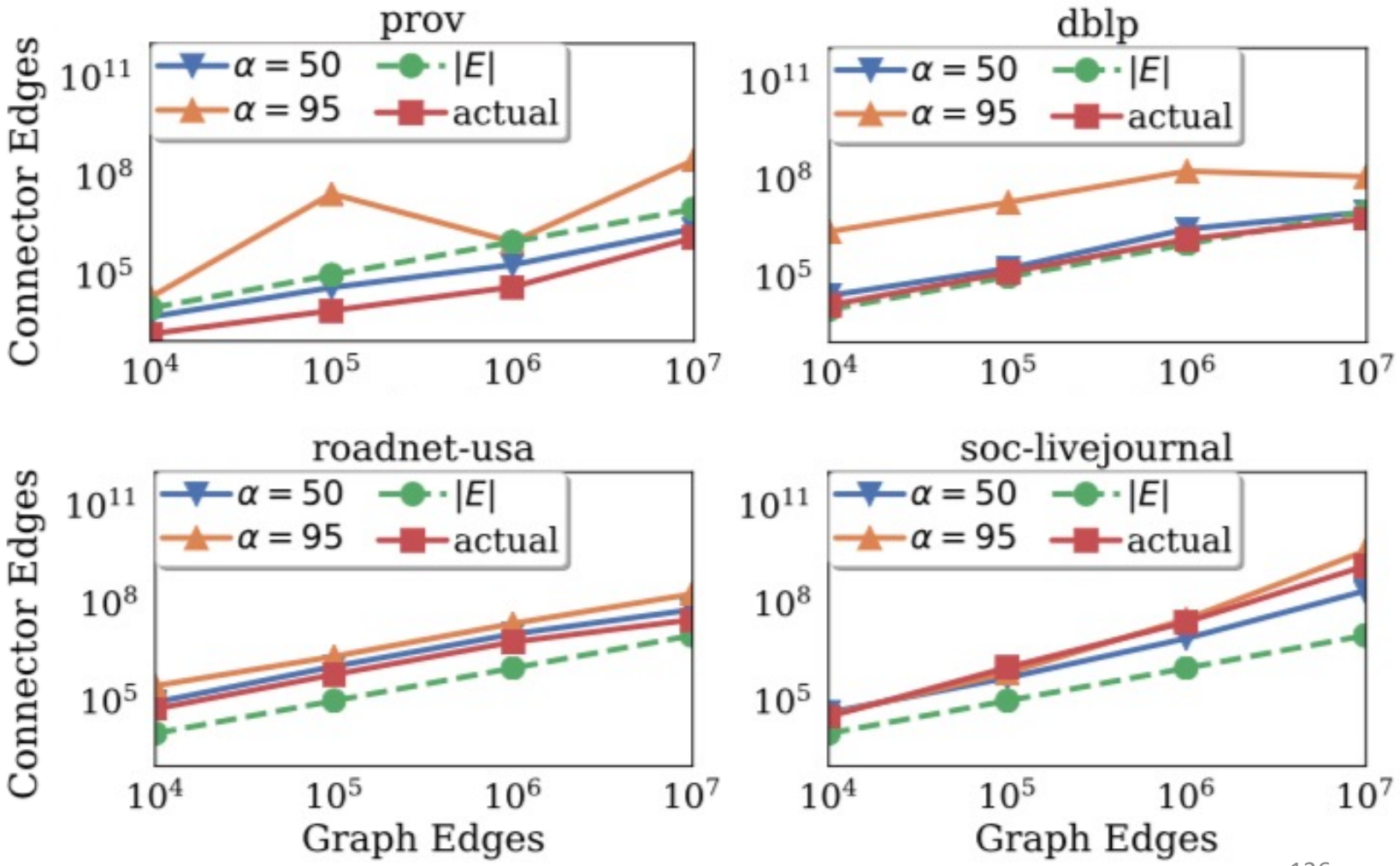
Number of likely k-length paths in G

$$\hat{E}(G, k, \alpha) = \sum_{t \in T_G} n_t \cdot (deg_\alpha(n_i))^k$$

Example: size estimator for k-hop connector views

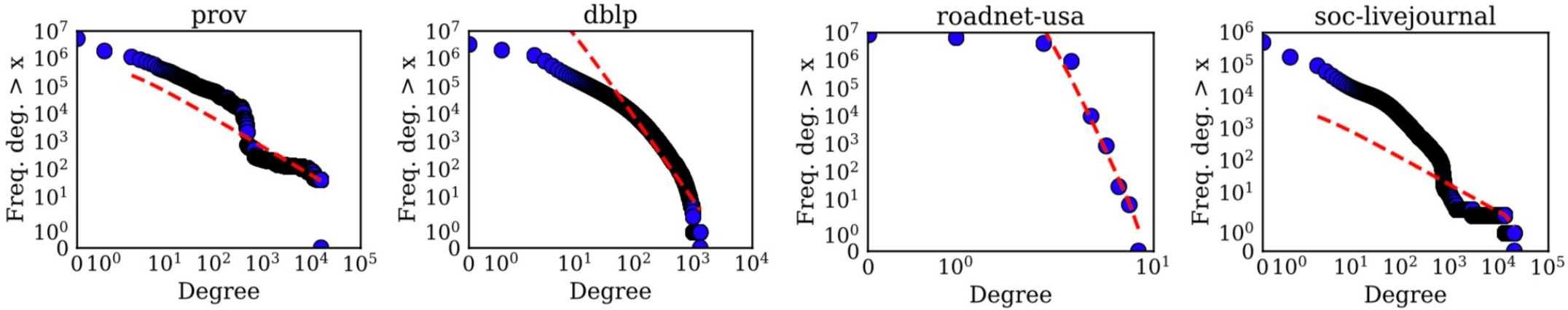
Cost Model: K-Hop Connector Size Estimator

We find that α = 95 provides a reasonable upper bound



Datasets

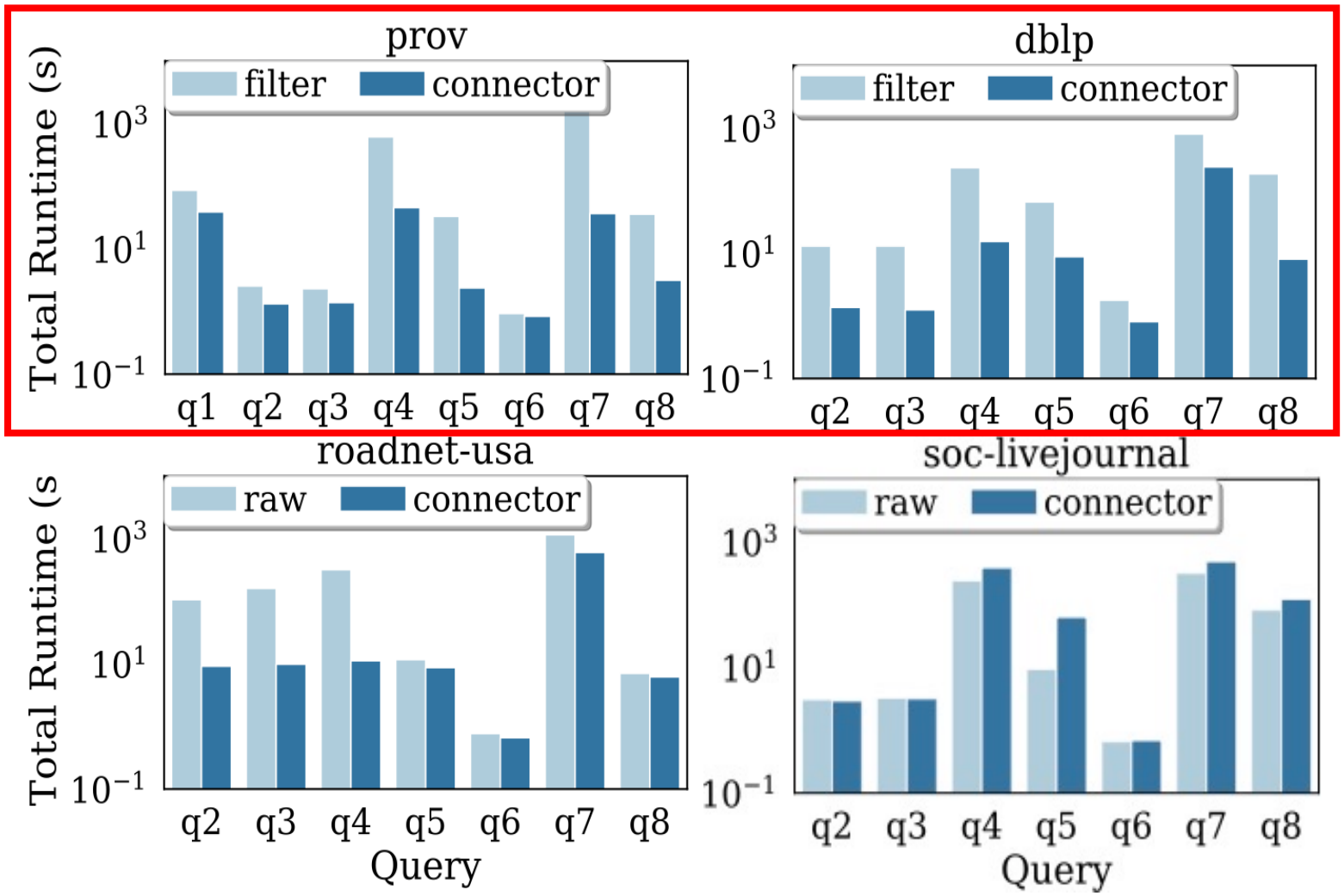
Short Name	Type	No. Vertices	No. Edges
prov (raw)	Data lineage	3.2 Billion	16.4 Billion
prov (summarized)	Data lineage	7 Million	34 Million
dblp-net	Co-authorship	5.1 Million	24.7 Million
soc-livejournal	Social network	4.8 Million	68.9 Million
roadnet-usa	Road network	23.9 Million	28.8 Million



Query Workload

Query	Operation Type	Result Type
Q1: Job Blast Radius	Retrieval	Subgraph
Q2: Ancestors	Retrieval	Set of Vertices
Q3: Descendants	Retrieval	Set of Vertices
Q4: Path lengths	Retrieval	Bag of scalars
Q5: Edge Count	Retrieval	Single scalar
Q6: Vertex Count	Retrieval	Single scalar
Q7: Community Detection	Update (property)	N/A
Q8: Largest Community	Retrieval	Subgraph

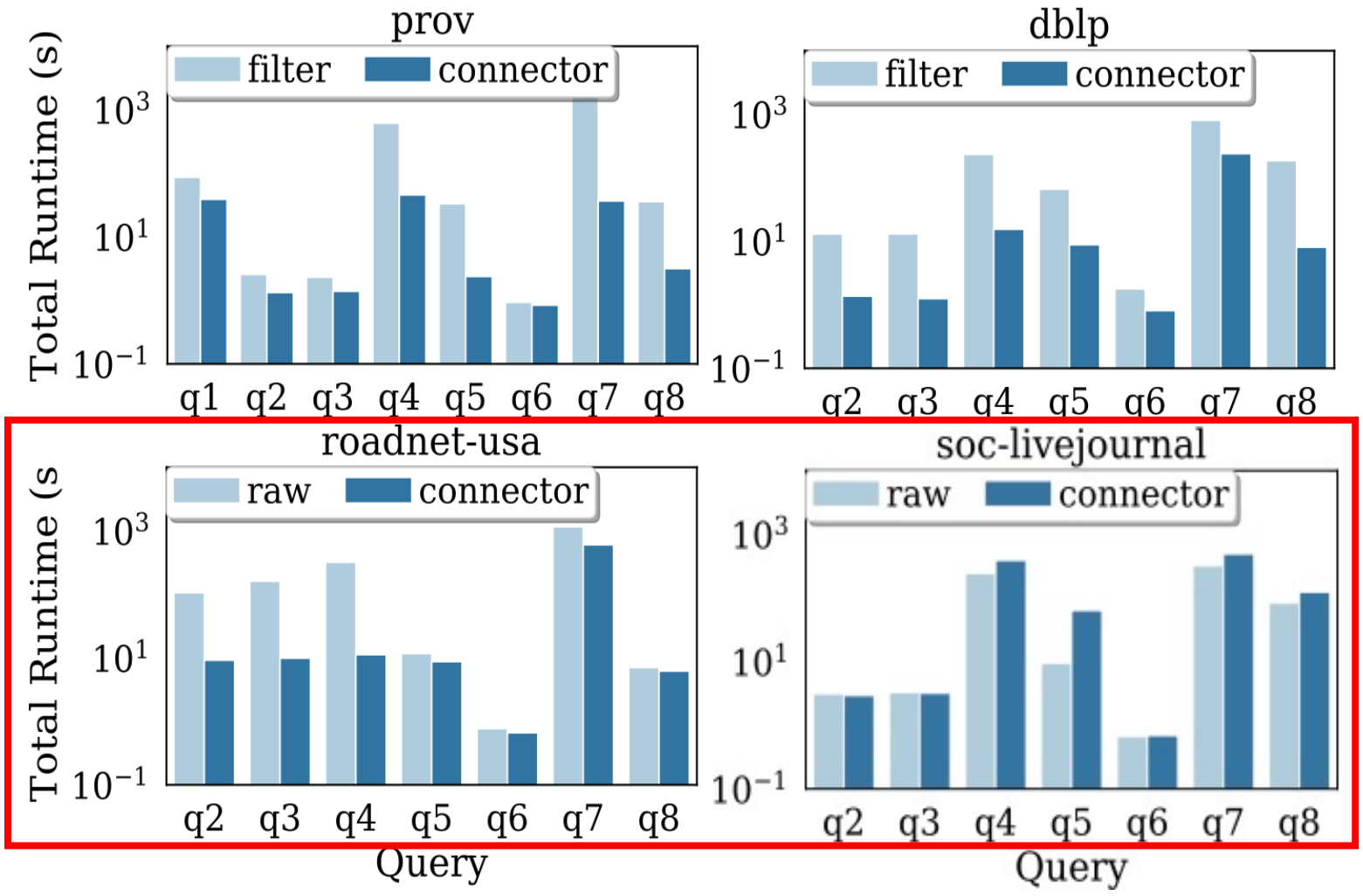
Query runtimes for rewrites over summarizer and and over 2-hop connector views



Q1: Job Blast Radius
Q2: Ancestors
Q3: Descendants
Q4: Path lengths
Q5: Edge Count
Q6: Vertex Count
Q7: Community Detection
Q8: Largest Community

Highly-structured graphs: virtually every query benefits from views

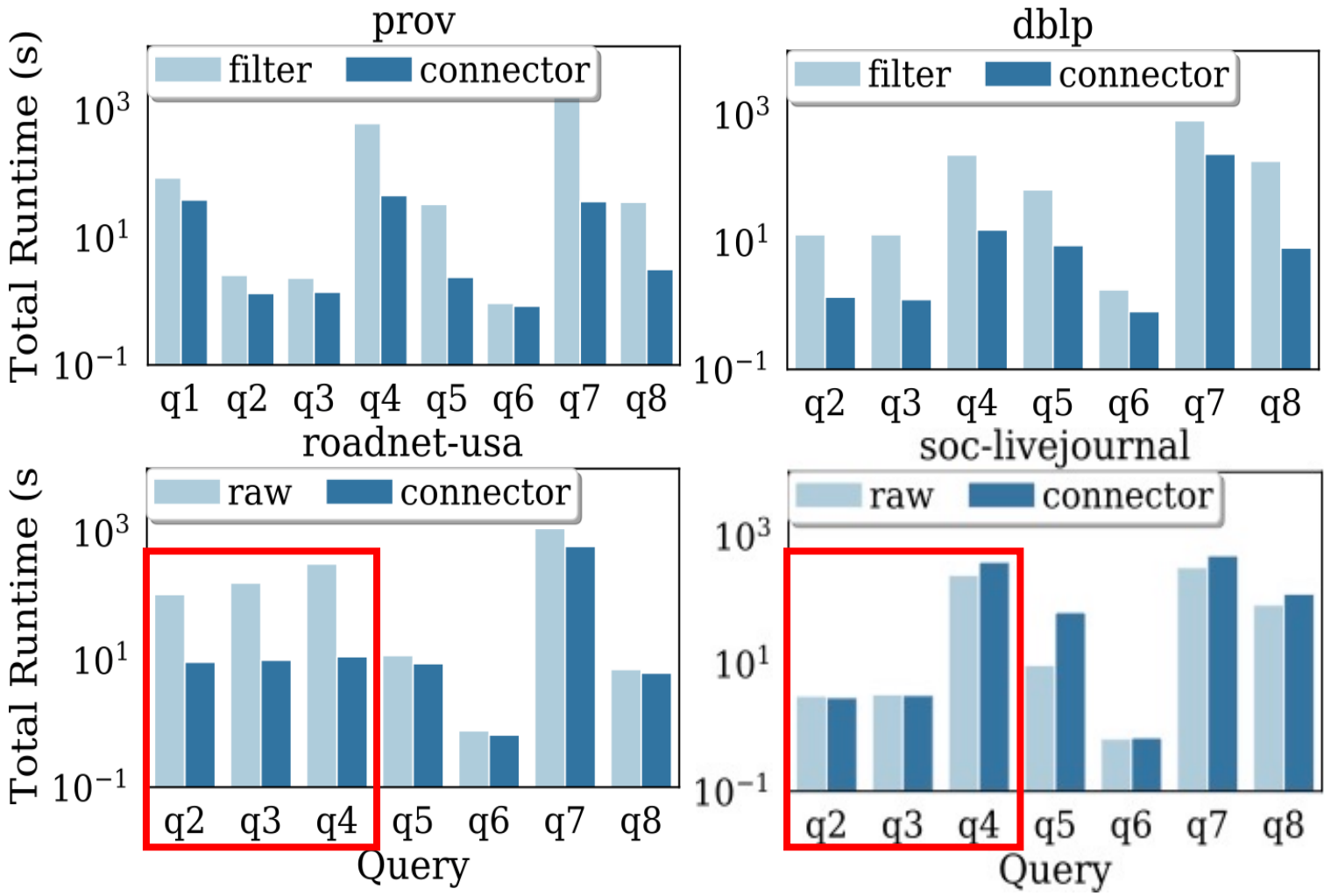
Query runtimes for rewrites over summarizer and and over 2-hop connector views



Q1: Job Blast Radius
Q2: Ancestors
Q3: Descendants
Q4: Path lengths
Q5: Edge Count
Q6: Vertex Count
Q7: Community Detection
Q8: Largest Community

Less structured graphs: vertex-to-vertex connector is larger than original graph.

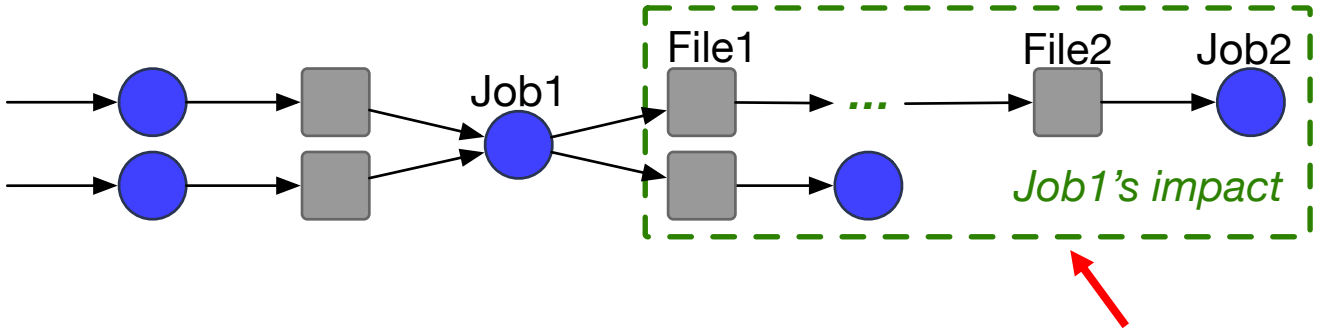
Query runtimes for rewrites over summarizer and and over 2-hop connector views



Q1: Job Blast Radius
Q2: Ancestors
Q3: Descendants
Q4: Path lengths
Q5: Edge Count
Q6: Vertex Count
Q7: Community Detection
Q8: Largest Community

Non-intuitive result due to larger fraction of long paths in road network.

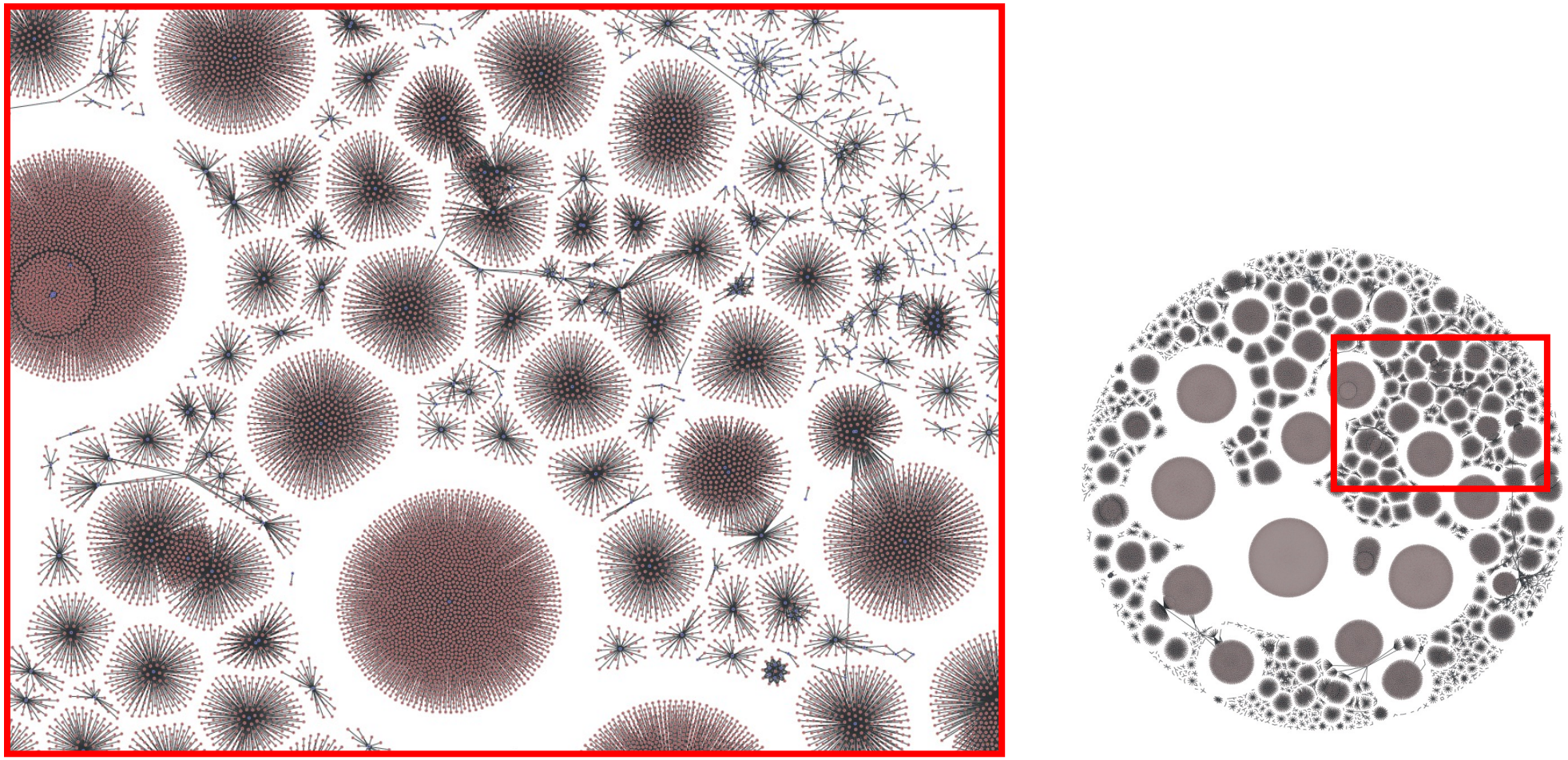
K-hop path query example: Job "blast radius" over machine generated lineage graphs



Several incident response use cases (e.g., identifying privacy violations) in datacenters consist of answering *k*-hop path queries.

R. Mavlyutov *et al*, *Dependency-Driven Analytics: A Compass for Uncharted Data Oceans.* CIDR 2017

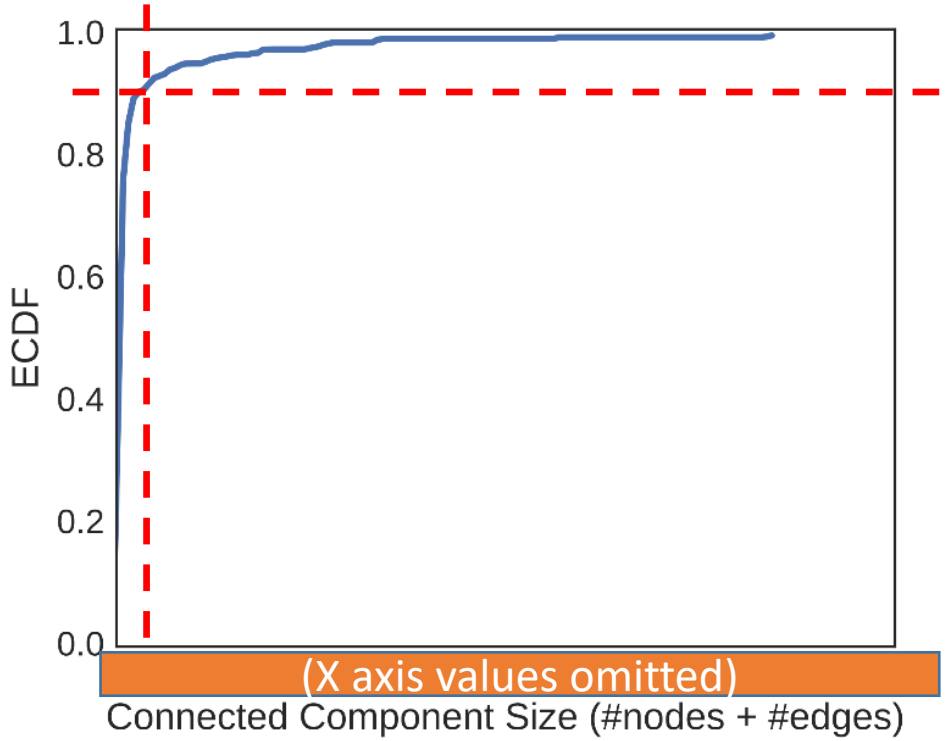
Real-world graphs are skewed (degree distr.)



Sample from Microsoft prod cluster data lineage graph: each CC is a pipeline

Real-world graphs are skewed (degree distr.)

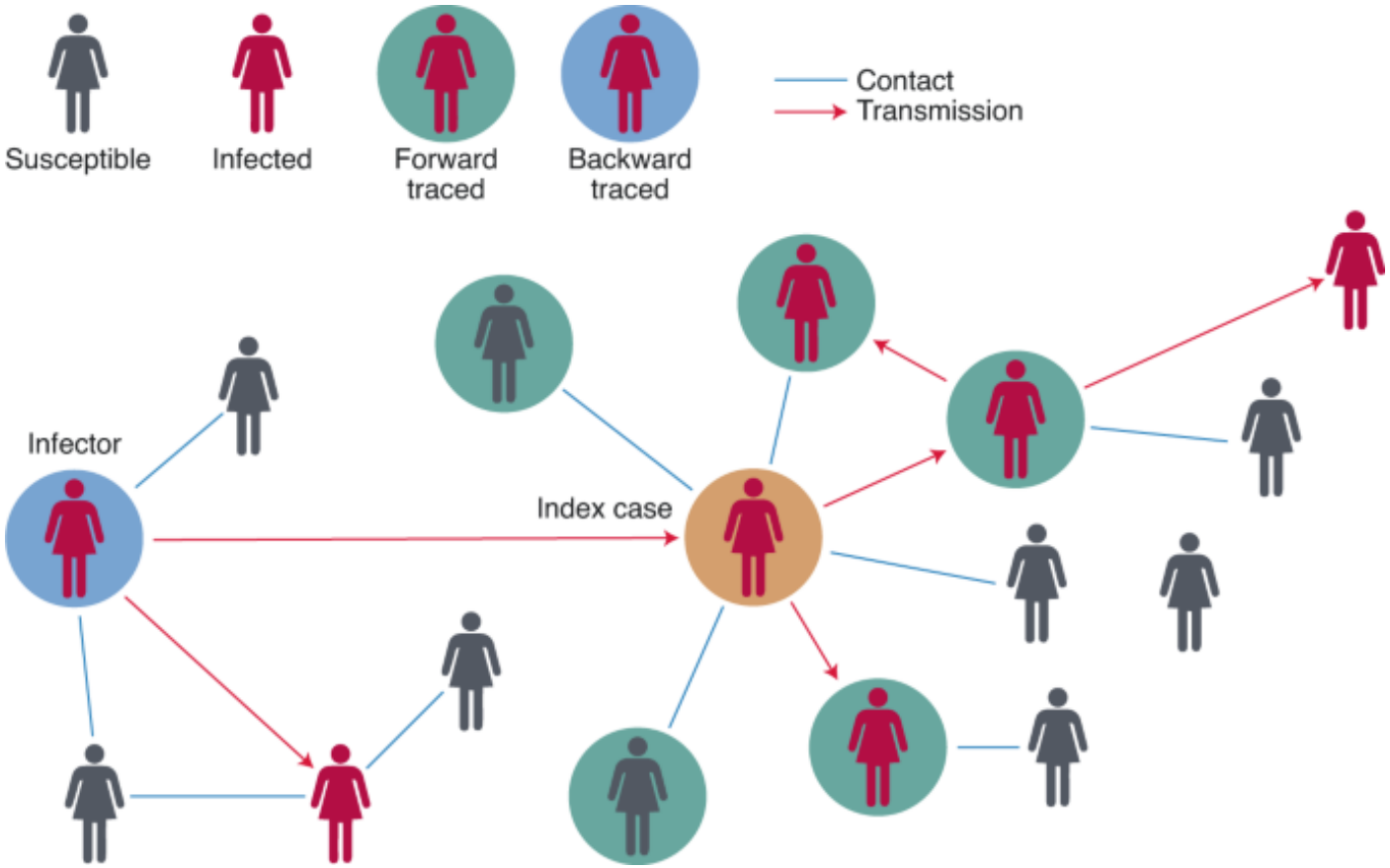
90th %ile CC size is orders of magnitude smaller than largest CC



e.g., a single job produced output consumed by almost all pipelines

Sample from Microsoft prod cluster data lineage graph: each CC is a pipeline

Why do we care about temporal graphs? Certain questions answered incorrectly without time info



Direction is incorrect if edge start time is missing.

False positives / negatives if edge duration is missing.

M. Johannes and M. Kretzschmar, *Forward thinking on backward tracing.* Nature Physics 2021

Is TGER sufficient for retrieving vertex's neighbors in all temporal walks of interest?