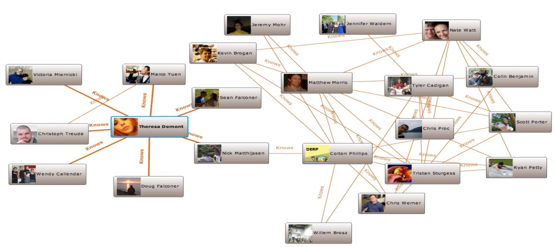
Modern Techniques for Querying Graph-Structured Relations: Foundations, Systems Implementations and Open Challenges

Amine Mhedhbi, Semih Salihoğlu

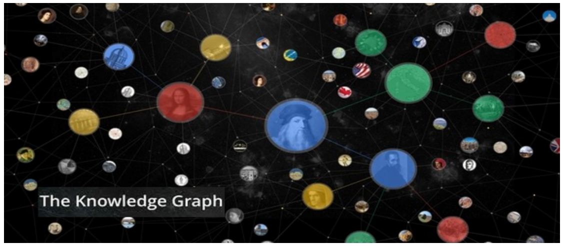
UNIVERSITY OF WATERLOO | DSG Data Systems Group

“Graph” Datasets and Workloads (1)

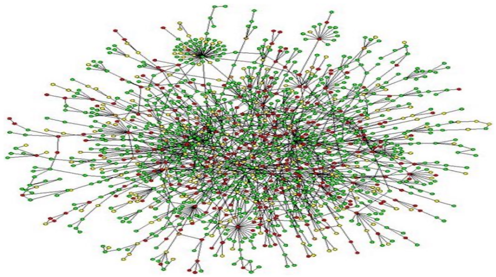
- “Relational” vs “graph” distinction is blurry:
  - most datasets can be modeled as relations or graph
- Classic “graph” datasets: social, encylopedic knowledge, or biological



Social Networks

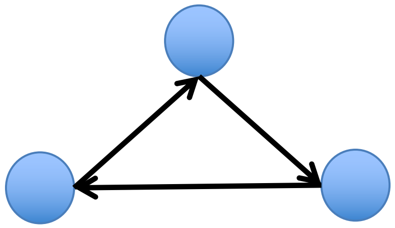


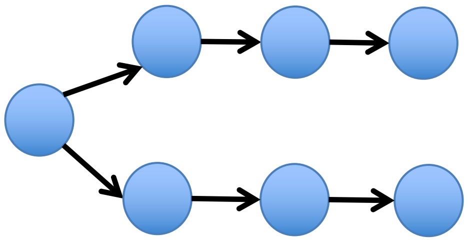
Knowledge Graphs

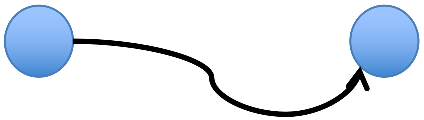


Biological Networks

- Classic “graph workloads”: finding cliques, long paths, reachability







“Graph” Datasets and Workloads (2)

- Colloquial term for datasets and workloads w/ several properties:
  1. Datasets contain *many-to-many (n-m)* relationships
     - Ex: Knows, Contacts, Calls, Transfers, etc.
  2. Queries contain many joins over n-m relationships
  3. Join queries can be cyclic or recursive
     - Ex: Cliques of contacts, indirect money transfers, etc.

The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing

Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo
{s3sahu,amine.mhedhbi,semih.salihoglu,jimmylin,tamer.ozsu}@uwaterloo.ca

ABSTRACT

Graph processing is becoming increasingly prevalent across many application domains. In spite of this prevalence, there is little research about how graphs are actually used in practice. We conducted an online survey aimed at understanding: (i) the types of graphs users have; (ii) the graph computations users run; (iii) the types of graph software users use; and (iv) the major challenges users face when processing their graphs. We describe the participants' responses to our questions highlighting common patterns and challenges. We further reviewed user feedback in the mailing lists, bug reports, and feature requests in the source repositories of a large suite of software products for processing graphs. Through our review, we were able to answer some new questions that were raised by participants' responses and identify specific challenges that users face when using different classes of graph software. The participants' responses and data we obtained revealed surprising facts about graph processing in practice. In particular, real-world graphs represent a very diverse range of entities and are often very large, and scalability and visualization are undeniably the most pressing challenges faced by participants. We hope these findings can guide future research.

PVLDB Reference Format:
Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *PVLDB*, 11(4): 420 - 431, 2017.
DOI: https://doi.org/10.1145/3164135.3164139

1. INTRODUCTION

Graph data representing connected entities and their relationships appear in many application domains, most naturally in social networks, the web, the semantic web, road maps, communication networks, biology, and finance, just to name a few examples. There has been a noticeable increase in the prevalence of work on graph processing both in research and in practice, evidenced by the surge in the number of different commercial and research software for managing and processing graphs. Examples include graph database systems [3,8,14,35,48,53], RDF engines [38,64,67], linear algebra software [6,46], visualization software [13,16], query languages [28, 52,55], and distributed graph processing systems [17,21,27]. In the academic literature, a large number of publications that study numerous topics related to graph processing regularly appear across a wide spectrum of research venues.

Despite their prevalence, there is little research on how graph data is actually used in practice and the major challenges facing users of graph data, both in industry and research. In April 2017, we conducted an online survey across 89 users of 22 different software products, with the goal of answering 4 high-level questions:

(i) What types of graph data do users have?
(ii) What computations do users run on their graphs?
(iii) Which software do users use to perform their computations?
(iv) What are the major challenges users face when processing their graph data?

Our major findings are as follows:

- *Variety:* Graphs in practice represent a very wide variety of entities, many of which are not naturally thought of as vertices and edges. Most surprisingly, traditional enterprise data comprised of products, orders, and transactions, which are typically seen as the perfect fit for relational systems, appear to be a very common form of data represented in participants' graphs.
- *Ubiquity of Very Large Graphs:* Many graphs in practice are very large, often containing over a billion edges. These large graphs represent a very wide range of entities and belong to organizations at all scales from very small enterprises to very large ones. This refutes the sometimes heard assumption that large graphs are a problem for only a few large organizations such as Google, Facebook, and Twitter.
- *Challenge of Scalability:* Scalability is unequivocally the most pressing challenge faced by participants. The ability to process very large graphs efficiently seems to be the biggest limitation of existing software.
- *Visualization:* Visualization is a very popular and central task in participants' graph processing pipelines. After scalability, participants indicated visualization as their second most pressing challenge, tied with challenges in graph query languages.
- *Prevalence of RDBMSes:* Relational databases still play an important role in managing and processing graphs.

Our survey also highlights other interesting facts, such as the prevalence of machine learning on graph data, e.g., for clustering vertices, predicting links, and finding influential vertices.

We further reviewed user feedback in the mailing lists, bug reports, and feature requests in the source code repositories of 22 software products between January and September of 2017 with two goals: (i) to answer several new questions that the participants' responses raised; and (ii) to identify more specific challenges in different classes of graph technologies than the ones we could iden-

Proceedings of the VLDB Endowment, Vol. 11, No. 4
Copyright 2017 VLDB Endowment 2150-8097/17/12... $ 10.00.
DOI: https://doi.org/10.1145/3164135.3164139

420

VLDBJ 2020

- Q1: Graph Data?
- Q2: Graph Computations?
- Q3: Graph Software?
- Q4: Main Challenges?
- Q5: Applications?

Data Systems Group

Volumes of Work on Graph Query Processing

Dgraph
neo4j
TigerGraph
MEMGRAPH
KATANA GRAPH
Amazon Neptune

Native Commercial GDBMSs

GRainDB
DuckDB
UMBRA
LogicBlox
EMPTYHEADED
GR-Fusion
RelationalAI
Ag AvantGraph
GQ-Fast
Graphflow

Academic Work or Relational Systems

Goal of Tutorial: Present common techniques that have emerged and is likely to lead to wide adoption in near future.

Tutorial Motivation and Goals

- Cover a suite of modern *join techniques* for graph workloads
- For each: (i) foundation; (ii) integration approaches; (iii) open problems

Ubiquitous in native GDBMSs

Techniques
1. Predefined/Pointer-based Joins w/ Join Indices
2. Worst-case optimal join algorithms
3. Factorization

- Emerged in PODS/ICDT work
- Addresses "intermediate data growth" of m-n joins
- Finds best application in graph workloads and GDBMSs

Our opinion: Any system in the GDBMS market will need to integrate these techniques to remain competitive (among others)

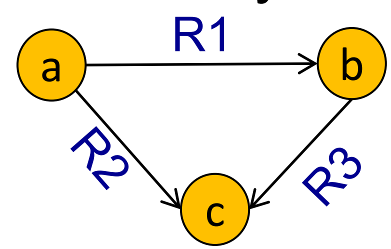
Systems and Integration Approaches Overview

DBMS	Join Type	Core Join Alg	WCOJ Algo	Data Representation Scheme
Umbra [17]	Value-based	HJ	Hash-based	Flat
GrainDB [10]	Value- and Pointer-based	HJ	Hash-based	Flat
EmptyHeaded [1]	Value-based	INLJ	Sorted indexes	Flat
GQ-Fast [13]	Value- and Pointer-based	INLJ	X	Flat
GR-Fusion [8]	Value- and Pointer-based	INLJ	X	Flat
GraphflowDB [11]	Pointer-based	HJ & INLJ	Sorted indexes	F-representations (restricted)
AvantGraph	Pointer-based	N/A	Sorted indexes	N/A
FDB [5]	Value-based	INLJ	Sorted indexes	F-representations
Neo4j	Pointer-based	HJ & INLJ	X	Flat
RDF-3X [18]	Value-based	MJ	X	Flat

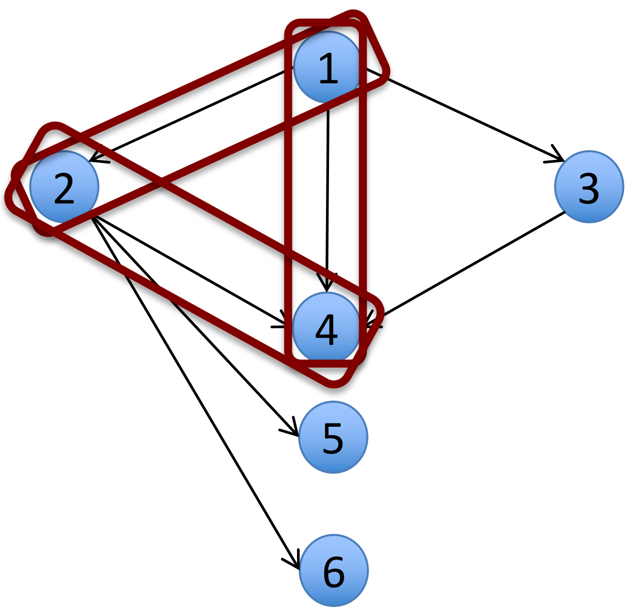
A Note on Query Notation

```
MATCH (a)->(b)->(c), (a)->(c)
```

Query



Input Graph



R1	
a	**b**
1	2
1	3
1	4
2	4
2	5
2	6
3	4

$\bowtie$

R2	
a	**c**
1	2
1	3
1	4
2	4
2	5
2	6
3	4

$\bowtie$

R3	
b	**c**
1	2
1	3
1	4
2	4
2	5
2	6
3	4

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1. Predefined Joins

2. Worst-case Optimal Joins (WCOJs)

3. Factorized Query Processing

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1. Predefined Joins

2. Worst-case Optimal Joins (WCOJs)

3. Factorized Query Processing

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1. Predefined Joins
    1.1. Foundations: Predefined vs Value-based Joins
    1.2. System Integration Approaches: GQ-Fast, GR-Fusion, GrainDB

2. Worst-case Optimal Joins (WCOJs)

3. Factorized Query Processing

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1. Predefined Joins
    1.1. Foundations: Predefined vs Value-based Joins
    1.2. System Integration Approaches: GQ-Fast, GR-Fusion, GrainDB

2. Worst-case Optimal Joins (WCOJs)

3. Factorized Query Processing

Predefined/Pointer-based vs Value-based Joins

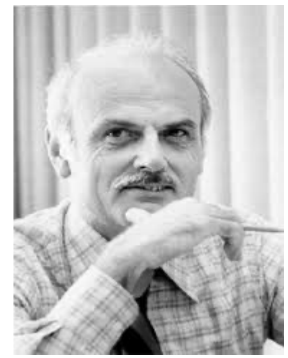
- A short history of the term ”pre-defined joins/access paths”

Network Model (1960s)

IDS: First DBMS in history

Charles Bachman

Relational Model (1970s)



Ted Codd

Much of the derivability power of the relational algebra is obtained from the SELECT, PROJECT, and JOIN operators alone, provided the JOIN is not subject to any implementation restrictions having to do with predefinition of supporting physical access paths. A system has an *unrestricted join capability* if it allows joins to be taken wherein *any* pair of attributes may be matched, providing only that they are defined on the same domain

… but also the reason GDBMSs can be very fast at those joins.

A 1962 Drawing of IDS's Data Model

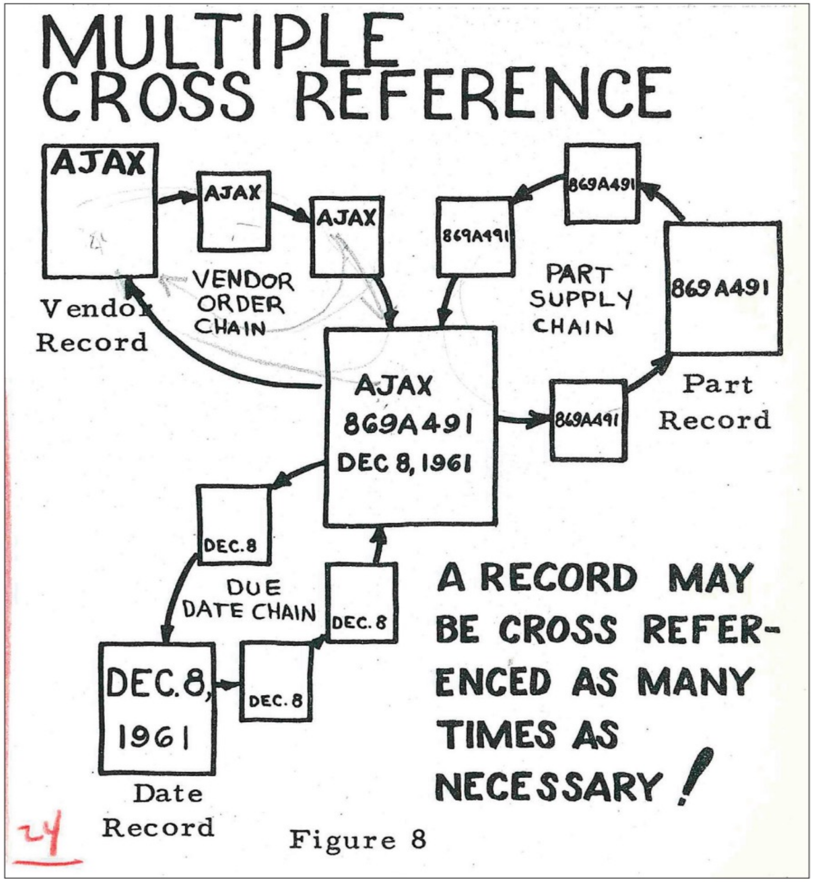


Figure 2. This drawing, from the 1962 presentation "IDS: The Information Processing Machine We Need," shows the use of chains to connect records. The programmer looped through GET NEXT commands to navigate between related records until an end-of-set condition is detected.

- Turing Award Lecture: Programmer As a Navigator
- Bachman's Talk at Computer History Museum

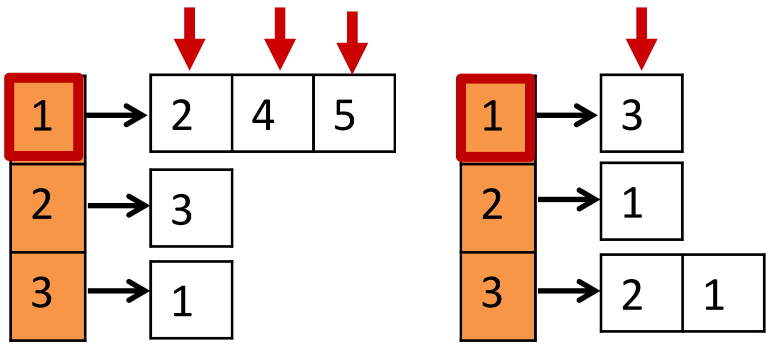
Data Systems Group

Common GDBMS Approach to Joins

1. Adjacency lists Join Index
2. Index Nested Loop Join-like Algorithms
3. Dense ID-based access (vs a hash function or B+ tree based)

```
MATCH a-[:Trn]->b-[:Trn]->c
WHERE b.owner = “Alice”
RETURN a.owner, c.owner
```

Scan Node Property a.owner=Alice → Extend (a)-[:Trnsfr]->(b) → Extend (a)<-[:Trnsfr]-(c)



a	b	c
3	1	[illegible]

Accounts	
ID	**owner**
1	Alice
2	Bob
3	Carol
…	…

Transfers		
src	**dst**	**amount**
1	2	700
2	3	800
3	1	900
1	4	500
1	5	400
…	…	…

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Open challenges.

1. Predefined Joins
    1.1. Foundations: Predefined vs Value-based Joins
    1.2. System Integration Approaches: GQ-Fast, GR-Fusion, GrainDB

2. Worst-case Optimal Joins (WCOJs)

3. Factorized Query Processing

Integration Approaches

- GDBMS: Already ubiquitous
- RDBMSs: Several proposals for join indices + dense ID-based joins
  - All provide DDL statements to define “graph views”
  - All use system-level row identifiers (RIDs) as pointers

System	Approach
GQ-Fast [Lin et al. VLDB ’16, ICDE ‘17]	Decoupled Processor, INLJ
GR-Fusion [Hassan et al., EDBT ’18, SIGMOD ‘18]	Decoupled Processor, INLJ
GRainDB [Jin et al. VLDB ‘22]	Single Processor, Hash Joins

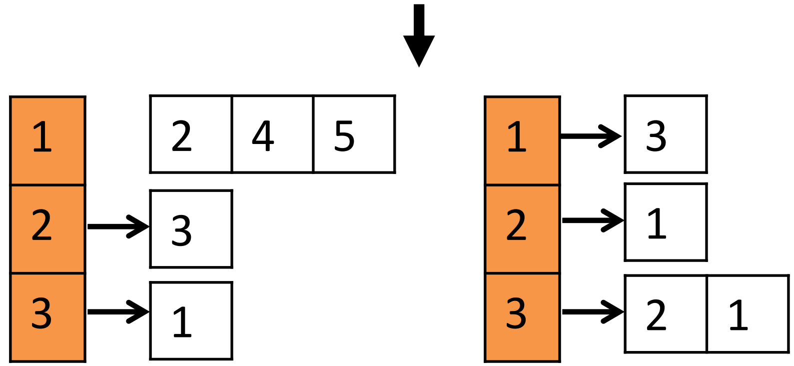
GR-Fusion (1): Graph Views & Join Index Creation

Accounts		
RID	owner	balance
1	Alice	1K
2	Bob	5K
3	Carol	7K
…	…	…

Transfers		
src	**dst**	amount
Alice	Bob	700
Bob	Carol	800
Carol	Alice	900
…	…	…

Customer	
owner	**job**
Alice	Doctor
Bob	Student
Carol	Lawyer
…	…

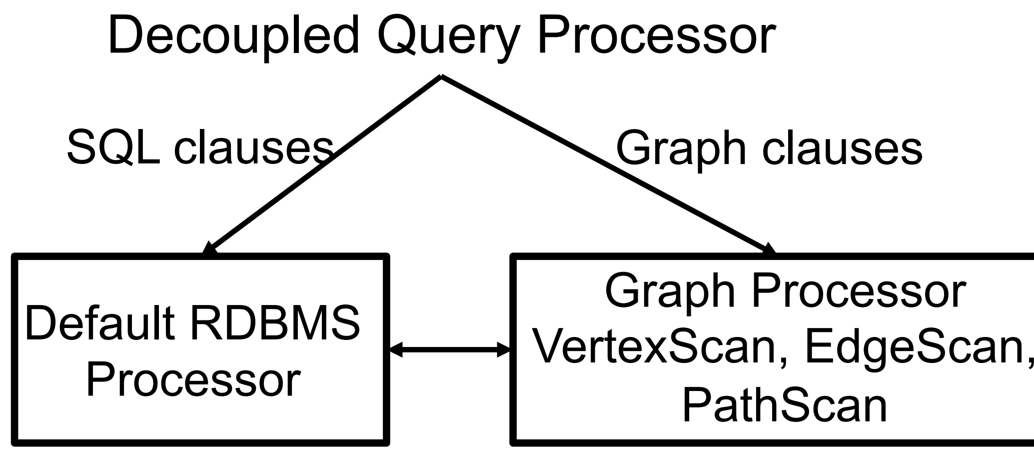
```
CREATE GRAPH VIEW FinancialGraph
VERTEXES(ID=owner, balance=balance) FROM Accounts
EDGES (FROM=src, TO=dst, amount=amount) FROM Transfers
```



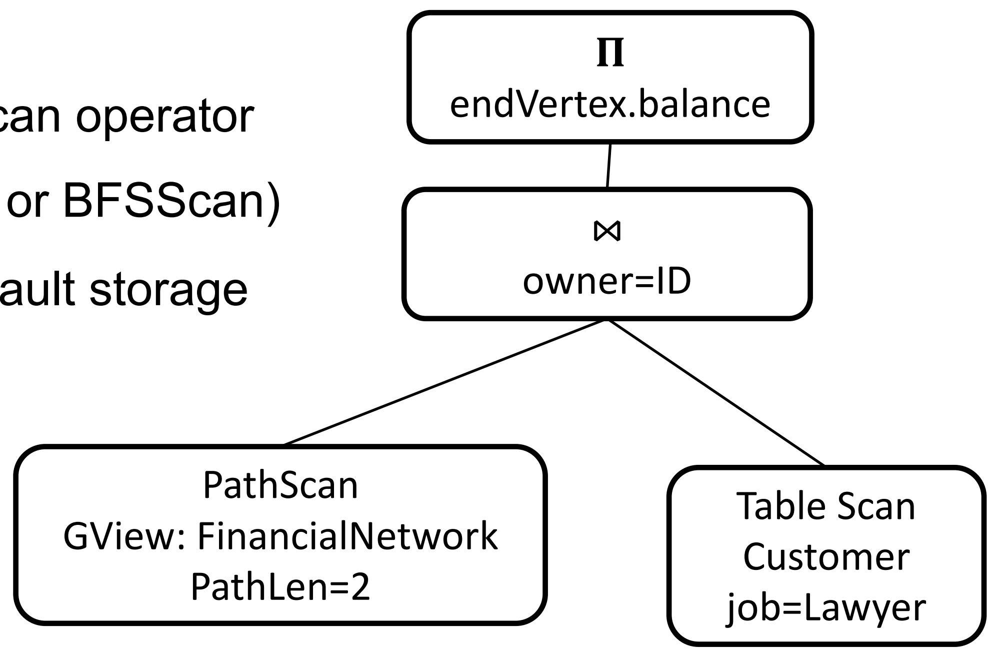
- Only the “topology”=join index is materialized
- Properties are in the system’s default storage

GR-Fusion (2): Decoupled Query Processor

```
SELECT PS.EndVertex.balance
FROM Customers C, FinGraph.Paths PS
WHERE C.job = 'Lawyer’
      AND PS.StartVertex.ID=C.owner
      AND PS.Length = 2
```



- PathScan:
  - Appears simply as another table scan operator
  - But implicitly does INLJ (DFSScan or BFSScan)
- Last projection: does lookups in the default storage



Data Systems Group

Pros and Cons of Decoupled Approaches

Pros	Cons
Easier to integrate	Only “graph” queries benefit
Can do very advanced processing: e.g., GR-Fusion has ShortestPathScan	Use of INLJ ops have performance disadvantages (next slides)

GRainDB Motivation: INLJ vs Hash Joins (1)

```
MATCH (a:P)-[e:Knows]->(b:P) WHERE a.ID < X RETURN count(*)
```

Standard GDBMS Plan: INLJ ops

Scan Node a.ID < x → Extend (a)-[:Knows]->(b)

✓ benefits from predicate

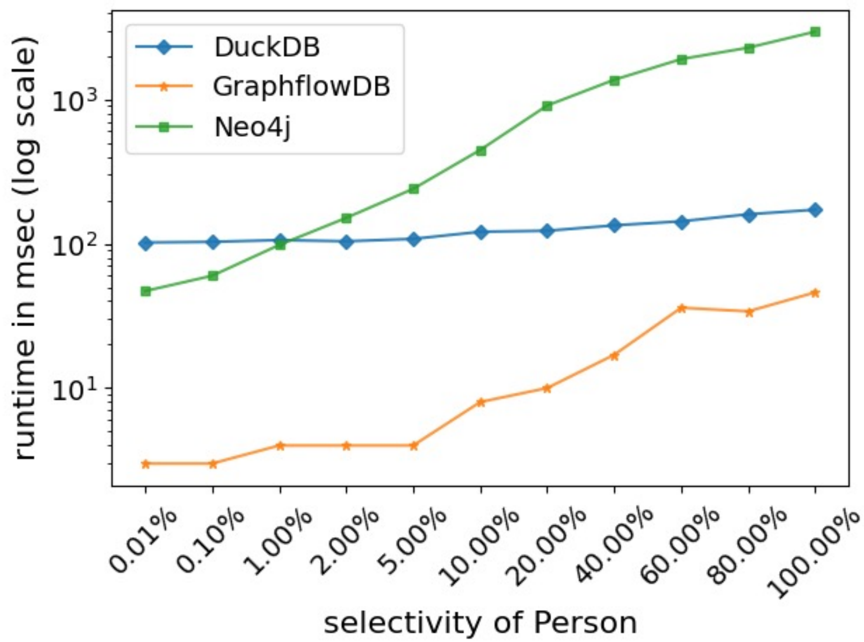
Standard RDBMS Plan: Hash Join

Hash Join e.srcID = a.ID

Scan Knows (e)

Scan Person (a) a.ID < X

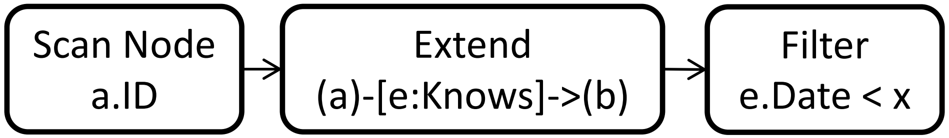
X no benefits from predicate



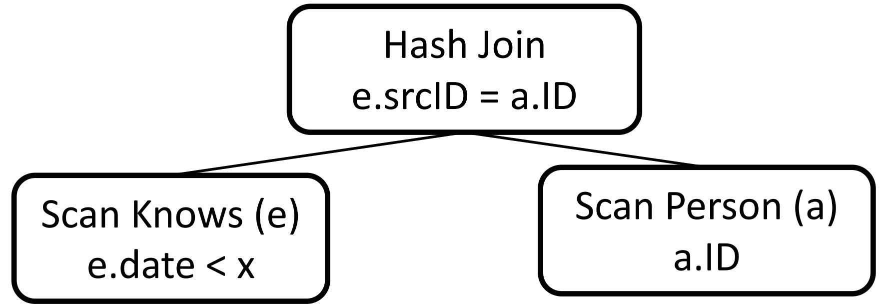
GRainDB Motivation: INLJ vs Hash Joins (2)

```
MATCH (a:P)-[e:Knows]->(b:P) WHERE e.date < X RETURN count(*)
```

Standard GDBMS Plan: INLJ ops

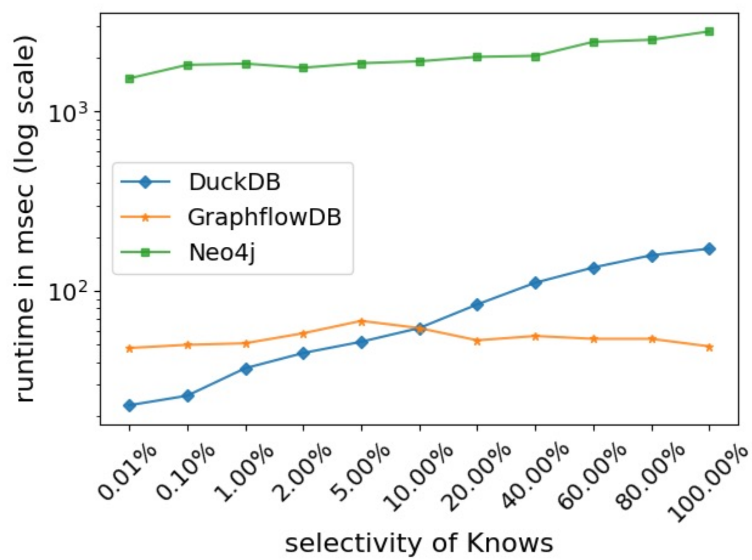


Standard RDBMS Plan: Hash Join



X no benefits from predicate

✓ benefits from predicate (even replace probe/build)

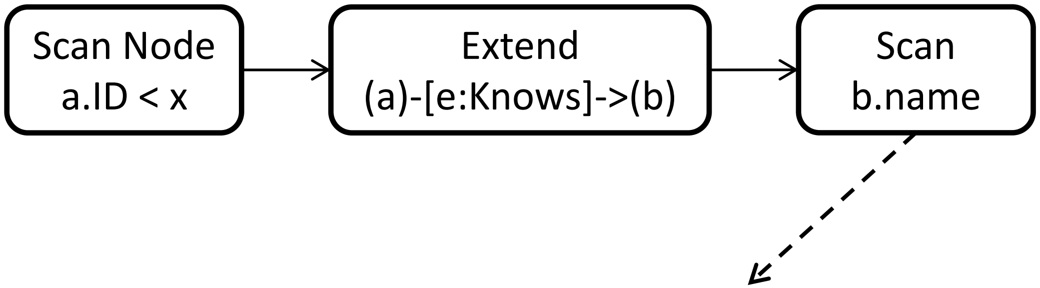


GRainDB Motivation: INLJ vs Hash Joins (3)

- Further problem with INLJs: Worse When Reading Node Properties

```
MATCH (a:P)-[e:Knows]->(b:P) WHERE a.ID < X RETURN b.name
```

Standard GDBMS Plan: INLJ ops



Effectively another INLJ operator:

joins (a.ID, e.ID, b.ID) tuple with (b.ID, b.name)

But leads to non-sequential/random reads b/c neighbors have no locality

Predefined/Pointer-based Joins in GRainDB: Goals

1. Always perform sequential reads
2. But benefit from both node/edge predicates
   - Achieved through sideways information passing
3. Do not develop a second “graph” processor:
   - Speed up existing primary-foreign key joins with a join index
   - In the spirit of old-fashioned join index of Valduriez but using modern data structures and join algorithms

Data Systems Group

Predefined Pointer-based Joins in GRainDB

Accounts	
RID	owner
1	Alice
2	Bob
3	Carol
…	…

Transfers			
RID	**From**	**To**	amount
1	Alice	Bob	700
2	Bob	Carol	800
3	Carol	Alice	900
4	Alice	Dan	500
5	Alice	Liz	400
…	…	…	…

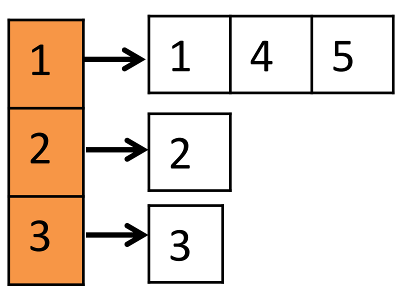
- Step 1: Predefine a Primary Key-Foreign Key Join E.g.:

  FROM: Accounts, Transfers

  WHERE Accounts.owner = Transfers.From

Accounts RIDs

Transfer RIDs

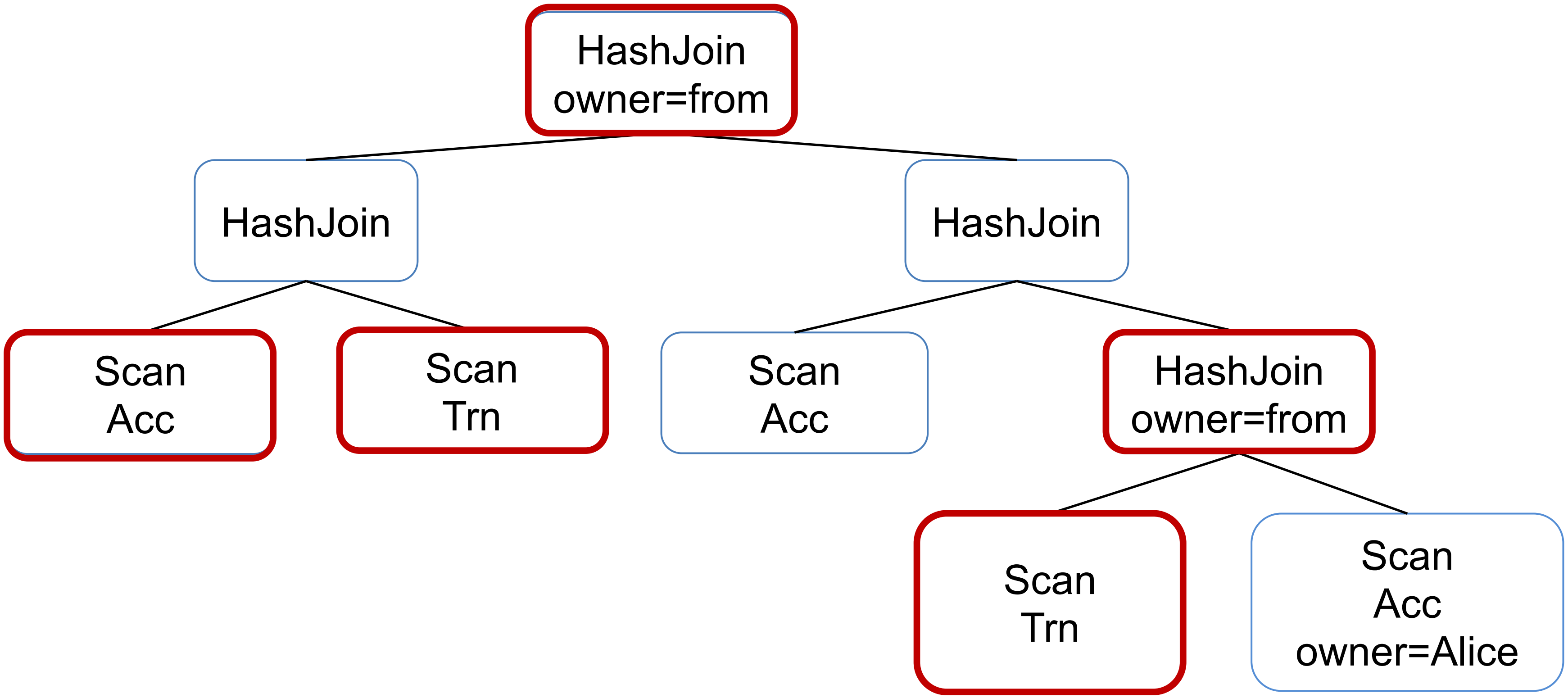


RID Index

Step 2: Rule-based Query Planning

Example “2-hop query”

```
SELECT a.owner, c.owner
FROM Acc a, b, c, Trn t1, t2
WHERE b.owner = Alice AND
a.owner=t1.From AND t1.To=b.owner AND
t1.To=t2.From AND t2.to=c.owner
```

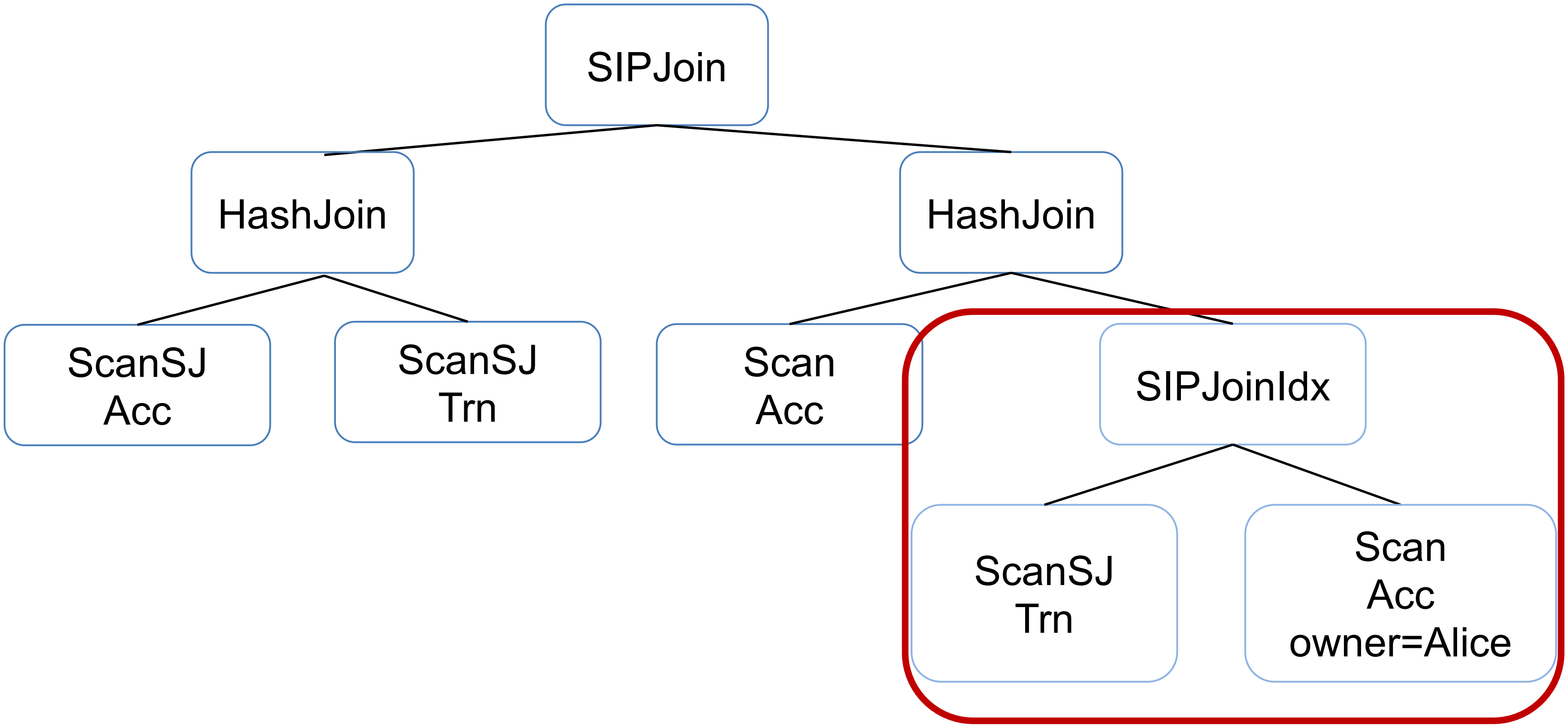


1. Replace some HashJoins -> SIPJoin or SIPJoinIdx
2. Replace some Scans -> ScanSemiJoins (ScanSJ)

Step 2: Rule-based Query Planning

Example “2-hop query”

```
SELECT a.owner, c.owner
FROM Acc a, b, c, Trn t1, t2
WHERE b.owner = Alice AND
a.owner=t1.From AND t1.To=b.owner AND
t1.To=t2.From AND t2.to=c.owner
```

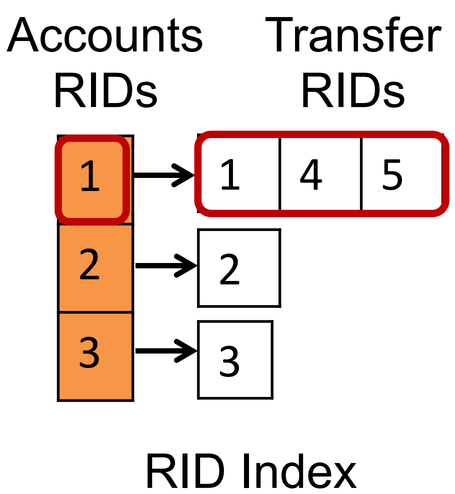


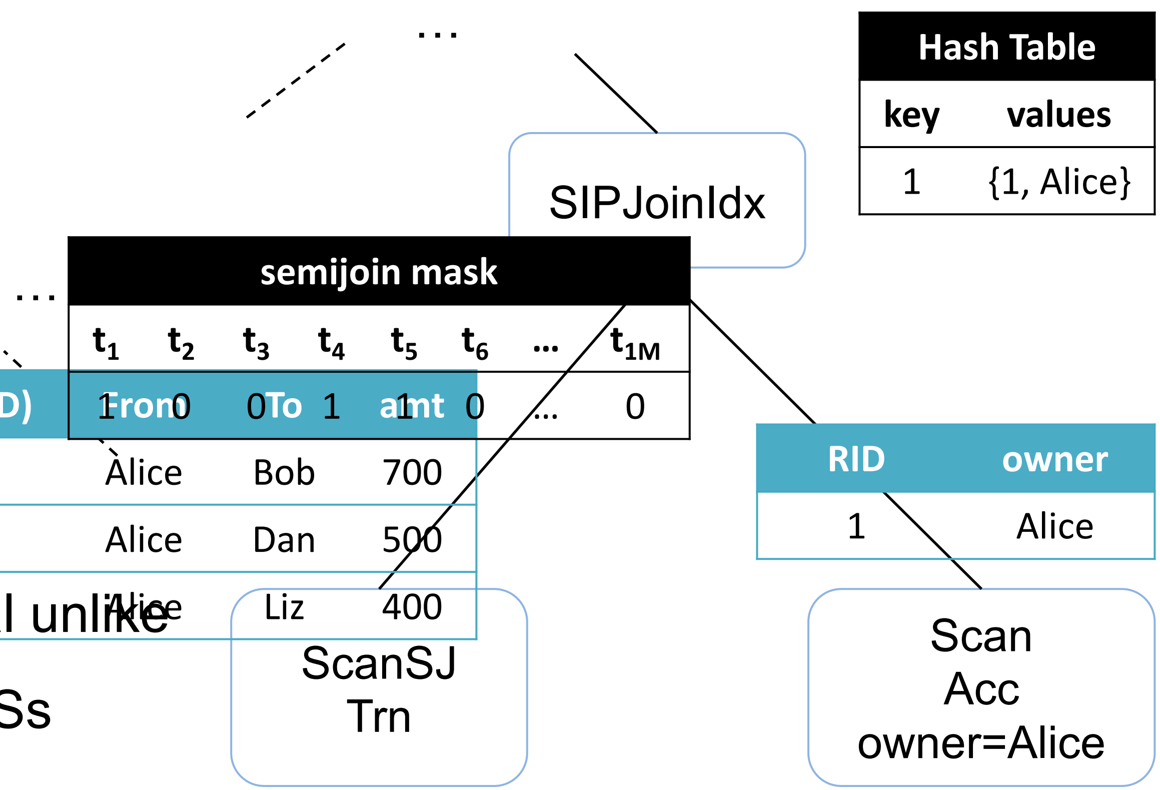
Step 3: Sideways Information Passing & Semijoins

```
SELECT a.owner, c.owner
FROM Acc a, b, c, Trn t1, t2
WHERE b.owner = Alice AND
a.owner=t1.From AND t1.To=b.owner AND
t1.To=t2.From AND t2.to=c.owner
```

Accounts	
RID	owner
1	Alice
2	Bob
3	Carol
...	...

Transfers				
RID	F(RID)	From	To	amount
1	1	Alice	Bob	700
2	2	Bob	Carol	800
3	3	Carol	Alice	900
4	1	Alice	Dan	500
5	1	Alice	Liz	400
...	...	...	...	...





Hash Table	
key	values
1	{1, Alice}

semijoin mask							
t_1	t_2	t_3	t_4	t_5	t_6	...	t_{1M}
1	0	0	1	1	0	...	0

RID	F(RID)	From	To	amt
1	1	Alice	Bob	700
4	1	Alice	Dan	500
5	1	Alice	Liz	400

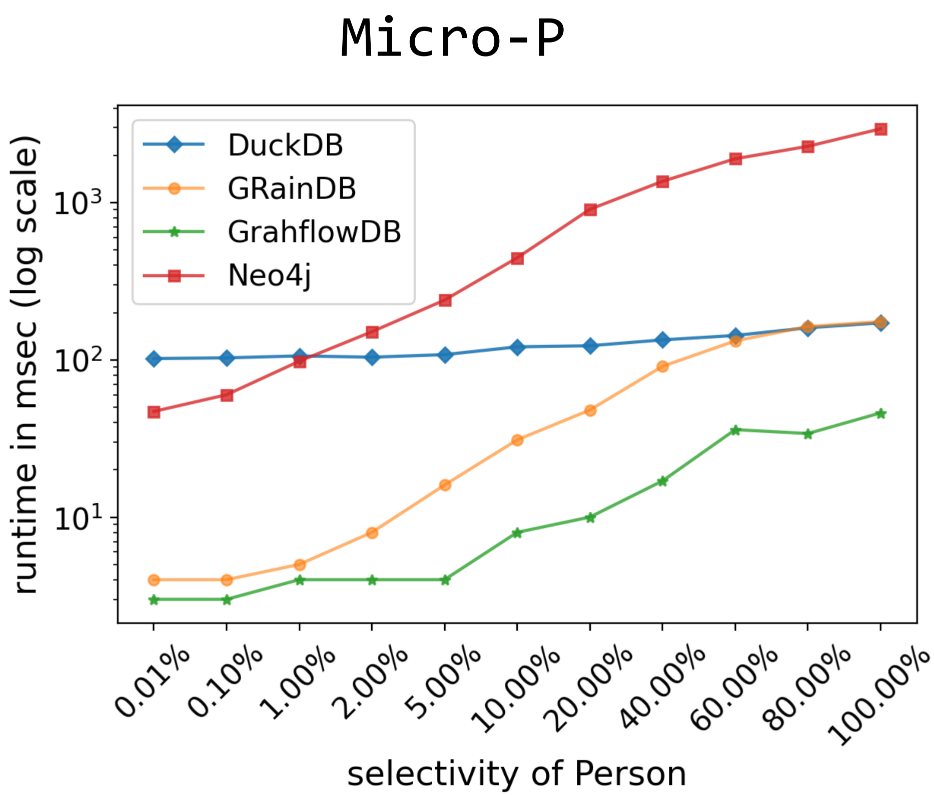
RID	owner
1	Alice

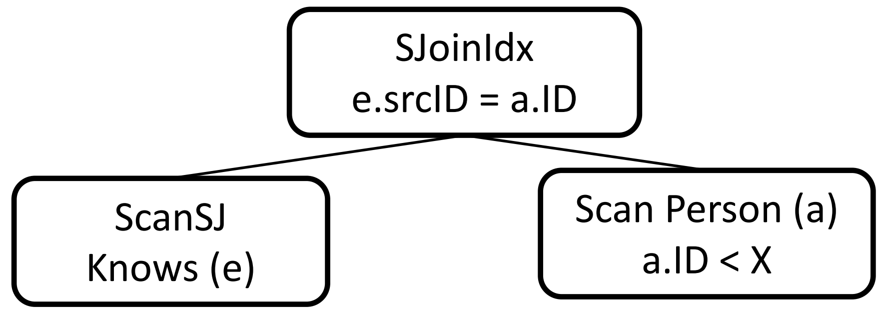
- Use RIDs as pointers
- All scans are sequential unlike nested loop joins of GDBMSs

Data Systems Group

GRainDB Microbenchmark Behavior (1)

```
MATCH (a:P)-[e:Knows]->(b:P)
WHERE a.ID < X
```

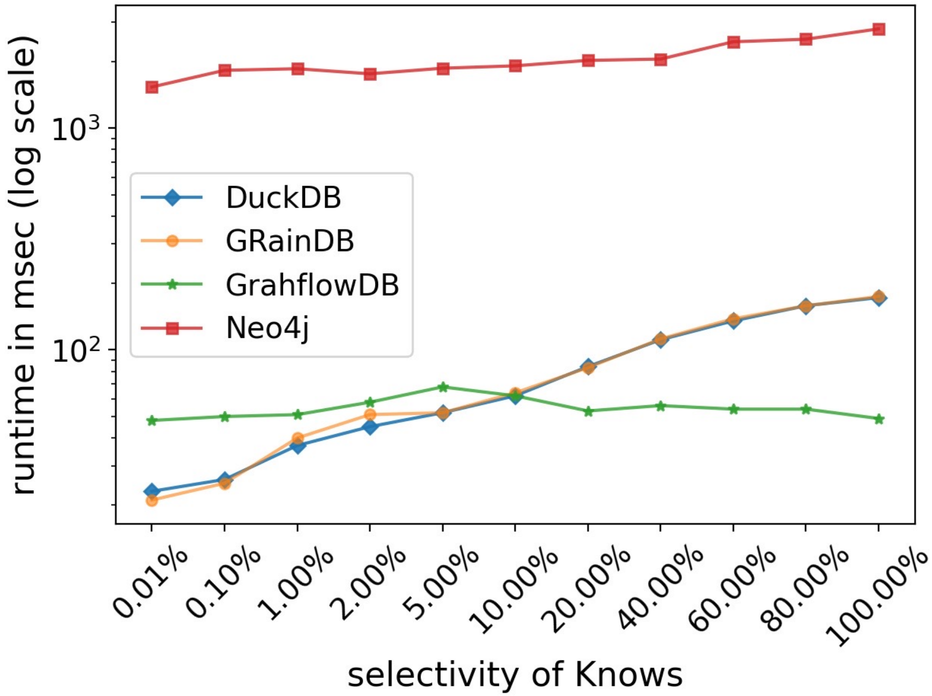


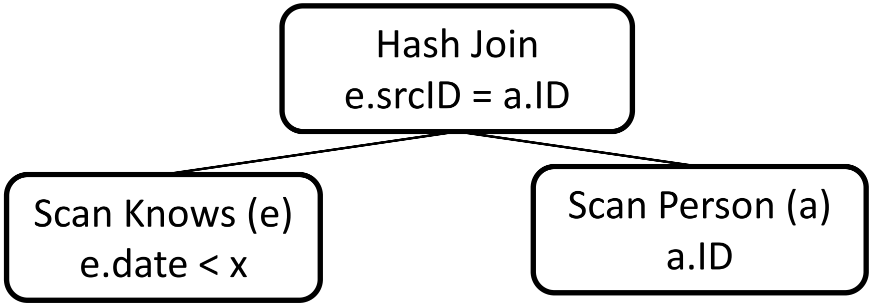


GRainDB Microbenchmark Behavior (2)

```
MATCH (a:P)-[e:Knows]->(b:P)
WHERE e.date < X
```

Micro-K





Summary

- Existing Approaches use RIDs and create join indices
- GR-Fusion & GQ-Fast use decoupled processors with INLJ ops:
  - Easier to integrate
  - Can provide more advanced query processing features
  - But INLJs can degrade in particular due to non-sequential reads
- GRainDB use a single integrated processor with HashJoins
  - Any PK-FK can benefit
  - Keeps all scans sequential
  - Unclear how to integrate more advanced processing (e.g., shortest path computations)

Open Challenges

- How about merge-joins: RDF-3X was based on MJs?
- Little work on optimizing queries
  - Each optimizer is rule-based
  - General wisdom: rule-based optimizers are rigid
- How much of join index-based operators can be implemented w/ UDFs?

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Ope
challenges.

1. Predefined Joins

2. Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins

3. Factorized Query Processing

Data
Systems
Group

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Ope
challenges.

1. Predefined Joins

2. Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2. System Integration Approaches

3. Factorized Query Processing

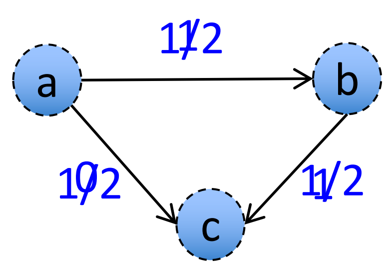
Worst-case Optimal Join Sizes

Given Q: R1 ⋈ R2 ⋈ … ⋈ Rn, what's the max |OUT|?

Theorem 1 (AGM, FOCS 2008):

Assume $|R_i|$ are equal. Let $\vec{e} = (e_1 \dots e_n)$ be a fractional edge cover:

Then: $|OUT| \leq IN^{|\vec{e}|}$ (IN is total input size)



ρ^* = 3/2

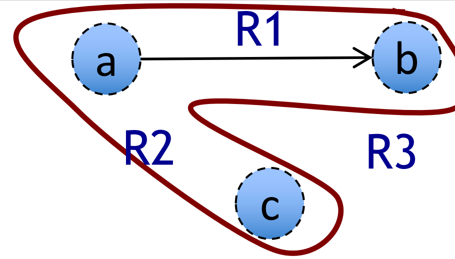
$$|OUT| \leq IN^{3/2}$$

ρ^* : weight of minimum fractional edge cover

$$|OUT| \leq IN^{\rho^*}$$

Traditionally: Binary Join (BJ) Plans

Table(s)/Q-Edge(s)-at-a-time Joins



R1	
a	b
1	2
1	3
1	4
…	

R2	
a	c
1	2
1	3
1	4
…	

R3	
b	c
1	2
1	3
1	4
…	

INT_1		
a	b	c
1	2	2
1	2	3
1	2	4
1	3	2
1	3	4
…		

Output		
a	b	c
1	2	4
1	3	4

BJ Plans are provably suboptimal!

E.g: can generate m^2 intermediate tuples on a graph with m edges (AGM bound is $m^{1.5}$)

Generic Join: A WCO Algorithm (NPRR, 2013)

a	b
1	2
1	3
1	4
2	4
2	5
2	6
3	4

$\cap$

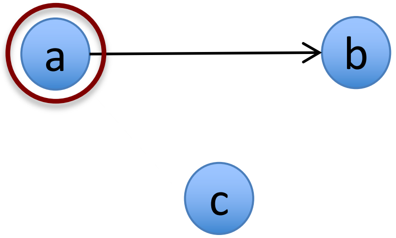
a	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

b	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

<u>Column/Q-Vertex-at-a-time</u>

Order q-vertices: say: a,b,c

INT_1
a



Generic Join: A WCO Algorithm (NPRR, 2013)

a	b
1	2
1	3
1	4
2	4
2	5
2	6
3	4

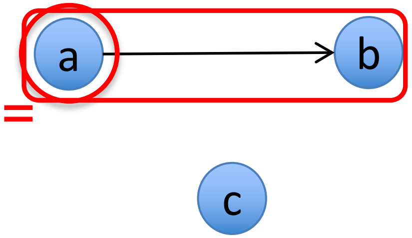
a	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

b	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

<u>Column/Q-Vertex-at-a-time</u>

Order q-vertices: say: a,b,c

Q =



INT_1
a
1
2
3

INT_2	
a	b

Generic Join: A WCO Algorithm (NPRR, 2013)

a	b
1	2
1	3
1	4
2	4
2	5
2	6
3	4

a	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

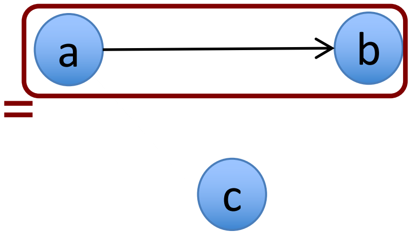
b	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

$\cap$

Column/Q-Vertex-at-a-time

Order q-vertices: say: a,b,c

Q =



INT_1
a
1
2
3

INT_2	
a	b
1	2
1	3

Generic Join: A WCO Algorithm (NPRR, 2013)

a	b
1	2
1	3
1	4
2	4
2	5
2	6
3	4

a	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

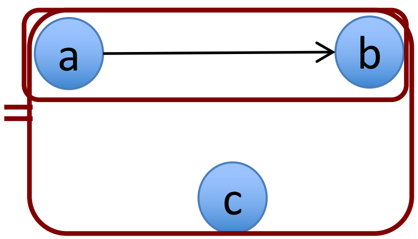
b	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

$\cap$

<u>Column/Q-Vertex-at-a-time</u>

Order q-vertices: say: a,b,c

Q =



INT_1
a
1
2
3

INT_2	
a	b
1	2
1	3
2	4
2	5
2	6
3	4

Output		
a	b	c

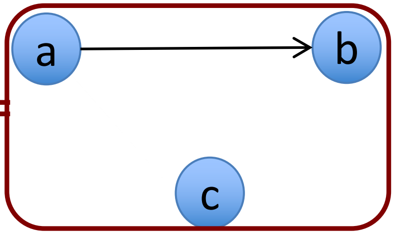
Generic Join: A WCO Algorithm (NPRR, 2013)

a	b
1	2
1	3
1	4
2	4
2	5
2	6
3	4

a	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

b	c
1	2
1	3
1	4
2	4
2	5
2	6
3	4

<u>Column/Q-Vertex-at-a-time</u>

Q = 

Order q-vertices: say: a,b,c

INT_1
a
1
2
3

INT_2	
a	b
1	2
1	3
2	4
2	5
2	6
3	4

Output		
a	b	c
1	2	4
1	3	4

Theorem: GJ is WCO for any query (under any ordering)

E.g. will generate $\leq m^{1.5}$ intermediate tuples

Summary of Two Theorems

Theorem 1 (AGM Bound):

Assume $|R_i|$ are equal. Let $\vec{e}^* = (e_{1^*} \dots e_{n^*})$ be min frac. edge cover:

Then: $|OUT| \leq IN^{\rho^* = |\vec{e}|}$ (IN is total input size)

Theorem 2 (GJ is WCO): Runtime of GJ ≤ AGM

(for *any query* & *any q-vertex ordering* (QVO))

Message: To be WCO:

do q-vertex-at-a-time matching *w/ multiway intersections*.

Data Systems Group

Outline of Query Processing Techniques to Cover

For each we cover: a) Foundations; b) System implementations; and c) Ope
challenges.

1. Predefined Joins

2. Worst-case Optimal Joins (WCOJs)
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   $\boxed{\text{2.2. System Integration Approaches}}$

3. Factorized Query Processing

Data
Systems
Group