GRainDB:

A Hybrid Graph-Relational DBMS

Semih Salihoğlu

Joint w/ Guodong Jin

Many Appeals of a Relational-core Hybrid System

- Hybrid System: An extended RDBMS w/ *graph modeling*, *querying*, and *visualization* capabilities.

1. No Perfect Data Model

<u>Tables</u>

- Legacy data
- Non-binary relations of entities
- Good for normalization (e.g., zipcodes, days, dates)

<u>Graphs</u>

- Arguably closer to developers' mental model of real-world entities and relationships

2. No Perfect Query Language

<u>SQL</u>

- Very popular and established
- Suitable for standard data analytics, preparation, etl…

<u>Graph Query Languages</u>

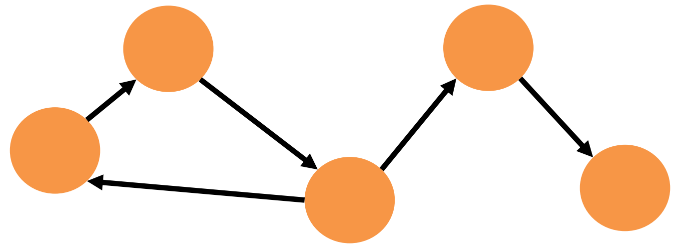
- Easier for recursive queries

```
MATCH a-[:Transfer*]->b
WHERE a.owner=Alice
```

3. Cheaper and quicker than building a completely separate GDBMS

GRainDB Vision

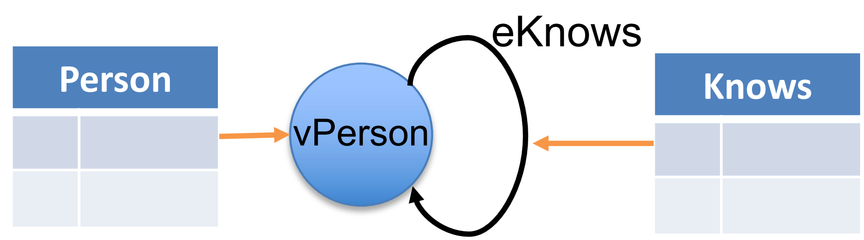
Neo4j Bloom-style Graph Visualization



G-SQL-style Seamless Table/Graph Querying

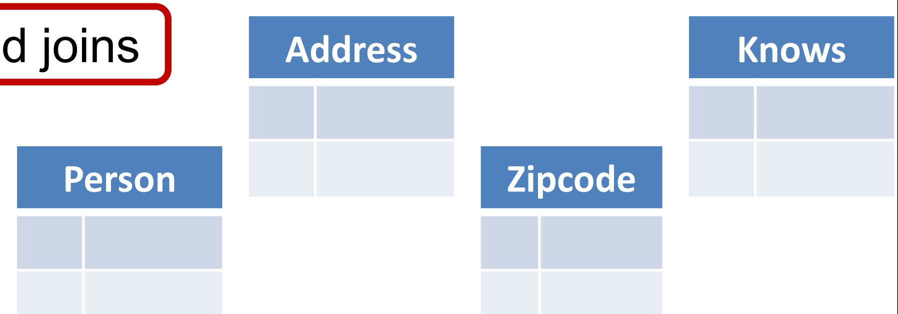
```
SELECT DISTINCT Address.zipcode
FROM (a:vPers)-[:eKnows*1..3]->(b:vPers),
     Address
WHERE a.name=Alice AND b.addID=Address.ID
```

Graph Modeling



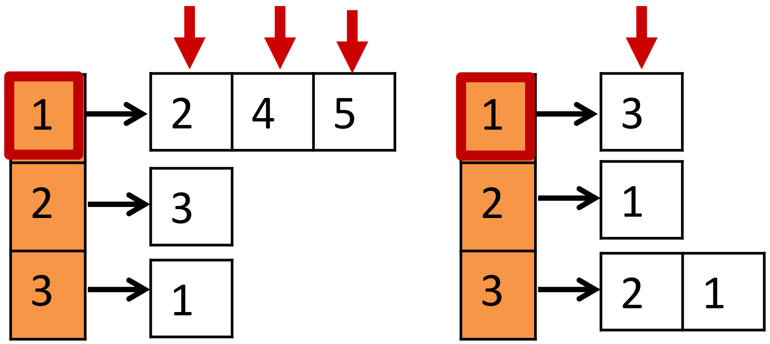
DuckDB

+ Pre-defined pointer-based joins
+ Factorization
+ Worst-case optimal joins
+ Recursive joins



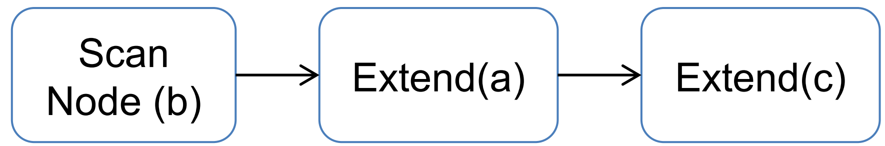
Predefined Pointer-based Joins in GDBMSs

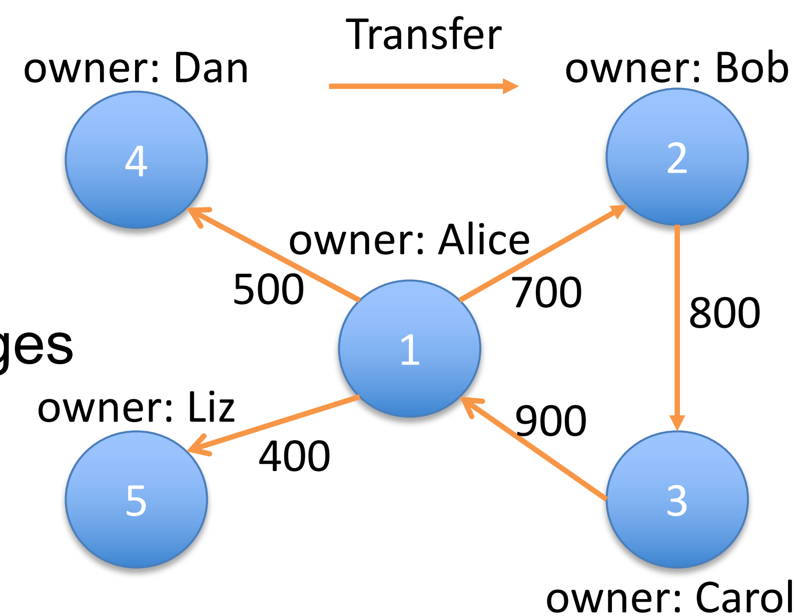
- Primary Difference in Join Processing in GDBMSs vs RDBMSs:
  - Pointer vs Value-based joins



a	b	c
3	1	4

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





- Adjacency Lists = An Index Over Edges
- ID-based Nested Index Loop Joins

Data Systems Group

Predefined Pointer-based Joins in GRainDB

```
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	**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

- Columnar RDBMS use Row IDs (RIDs) as system-level pointers

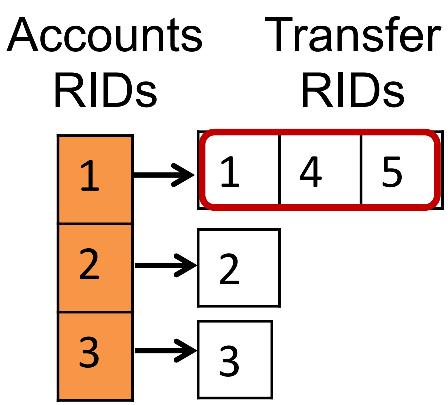
Data Systems Group

Step 1: RID Materialization and RID Index

```
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
…	…

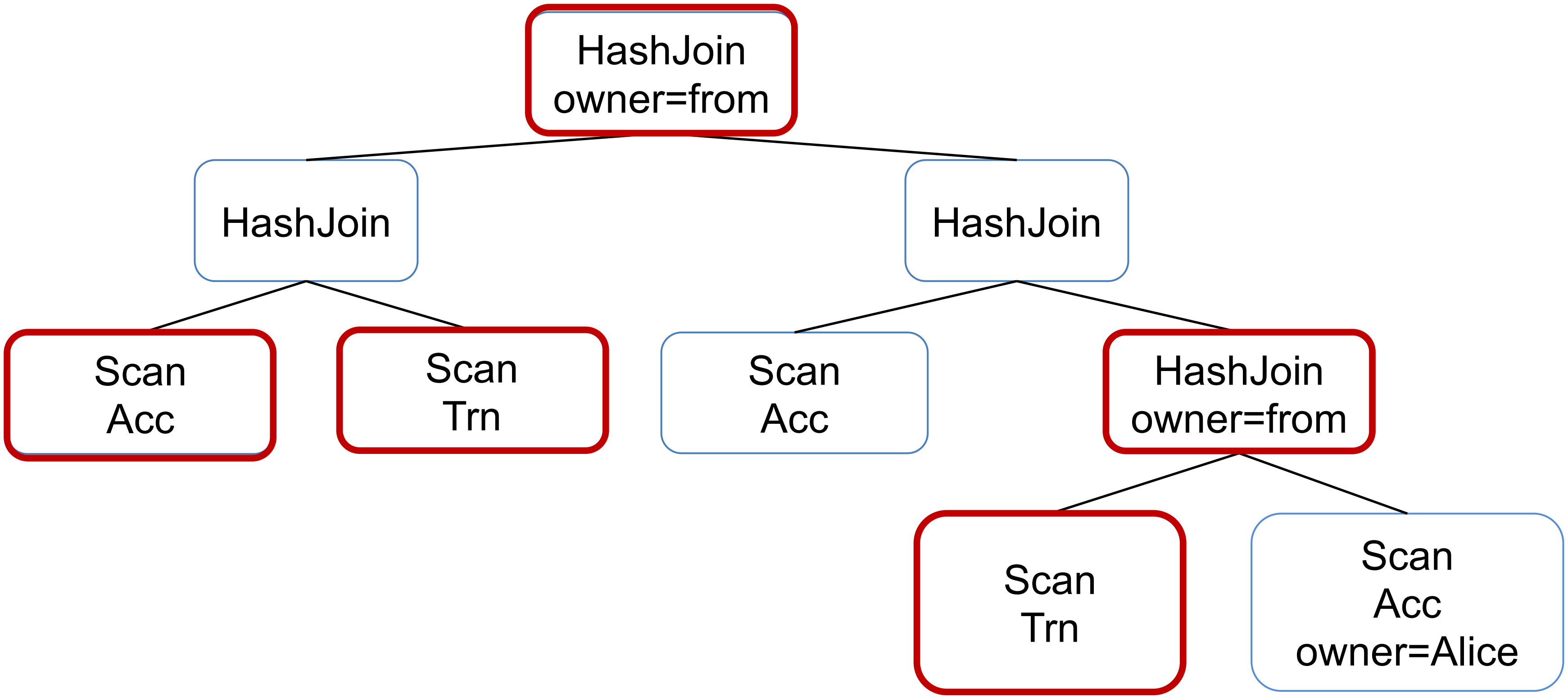
		Transfers			
RID	F	RID	From	To	amount
1	1	1	Alice	Bob	700
2	2	2	Bob	Carol	800
3	3	3	Carol	Alice	900
4	1	4	Alice	Dan	500
5	1	5	Alice	Liz	400
…	…	…	…	…	…



RID Index

Step 2: Rule-based Query Planning

```
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
```

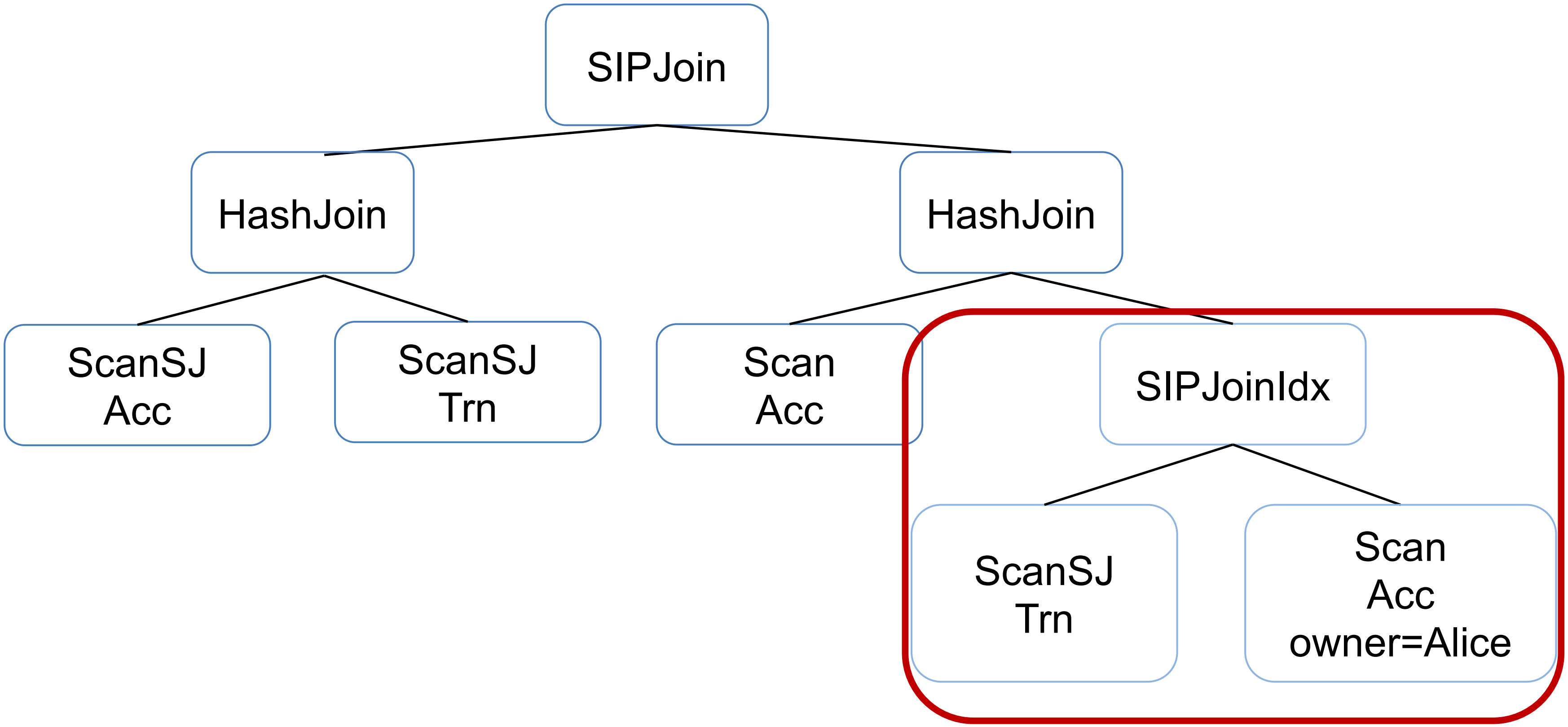


1. Replace some HashJoins -> SIPJoin or SIPJoinIdx
2. Replace some Scans -> ScanSemiJoins (ScanSJ)

Data Systems Group

Step 2: Rule-based Query Planning

```
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
```

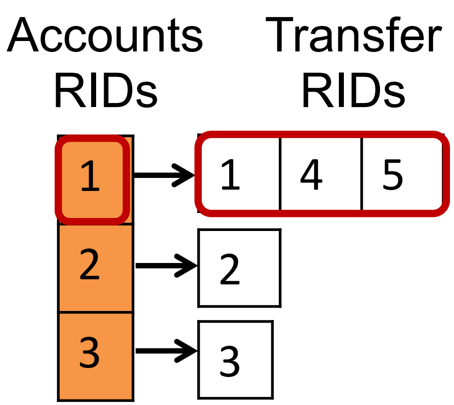


Step 3: Sideways Information Passing & Semijoins

```
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
…	…	…	…	…



RID Index

Hash Table	
key	values
1	{1, Alice}

SIPJoinIdx

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

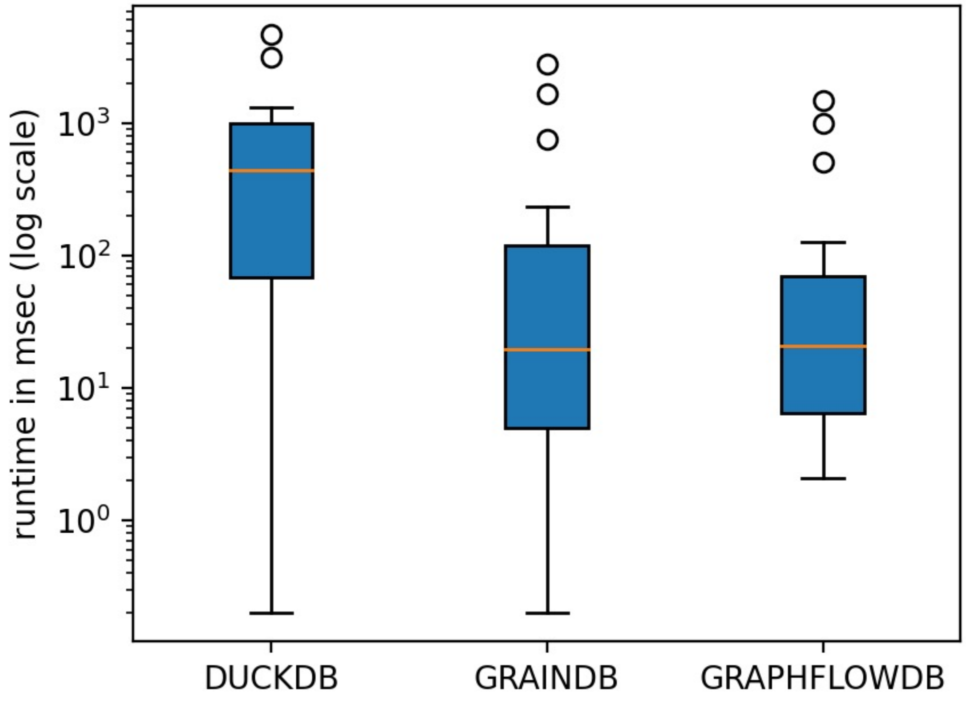
ScanSJ Trn

Scan Acc owner=Alice

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

Experiment: LDBC Social Network Graph Benchmark

- LDBC 10 Benchmark: ~10GB
- Dual 2.6GHz Intel CPU, 256GB RAM
- In-Memory Performance



The researcher, engineer, and hero!

Guodong Jin

Making RDBMSs Efficient on Graph Workloads Through Predefined Joins

Guodong Jin
jinguodong@ruc.edu.cn
Renmin University of China

Semih Salihoglu
semih.salihoglu@uwaterloo.ca
University of Waterloo, Canada

ABSTRACT

Joins in native graph database management systems (GDBMSs) are predefined to the system as edges, which are indexed in adjacency list indices and serve as pointers. This contrasts with and can be more performant than value-based joins in RDBMSs and has lead researchers to investigate ways to integrate predefined joins directly into RDBMSs. Existing approaches adopt a strict separation of graph and relational data and processors, where a graph-specific processor uses left-deep and index nested loop joins for a subset of joins. This may be suboptimal, and may lead to non-sequential scans of data in some queries. We propose a purely relational approach to integrate predefined joins in columnar RDBMSs that uses row IDs (RIDs) of tuples as pointers. Users can predefine equality joins between any two tables, which leads to materializing RIDs in extended tables and optionally in RID indices. Instead of using the RID index to perform the join directly, we use it primarily in hash joins to generate semi-join filters that can be passed to scans using sideways information passing, ensuring sequential scans. In some settings, we also use RID indices to reduce the number of joins in query plans. Our approach does not introduce any graph-specific system components, can execute predefined joins on any join plan, and can improve performance on any workload that contains equality joins that can be predefined. We integrated our approach to DuckDB and call the resulting system *GRainDB*. We demonstrate that GRainDB far improves the performance of DuckDB on relational and graph workloads with large many-to-many joins, making it competitive with a state-of-the-art GDBMS, and incurs no major overheads otherwise.

1 INTRODUCTION

Perhaps the two most commonly used data structures to model data in enterprise database applications are tables, which are the core structures of relational database management systems (RDBMSs), and graphs, which are the core structures of several classes of systems, most recently of property graph database management systems (GDBMSs for short), such as Neo4j [6], TigerGraph [7], DGraph [2], and GraphflowDB [22, 28, 34–36]. Aside from developer preference for using a graph-specific data model and query language, GDBMSs target what are colloquially referred to as *graph workloads*, which refer to workloads that contain large many-to-many joins. For example, these workloads appear in social networking applications for finding long paths between two people over many-to-many friendship relationships or in financial fraud detection applications for finding fraudulent patterns across many-to-many money transfers across bank accounts.

At the same time, several economic and technical factors have lead researchers to investigate techniques to support efficient graph querying natively inside RDBMSs. For example, it is recognized that the data stored in many specialized GDBMSs are extracted from RDBMSs [12, 45, 48, 51]. In many enterprises, users replicate parts of the tabular data stored in RDBMSs to a GDBMS because their applications require the fast join capabilities of GDBMSs. In addition, many applications require other processing on their graph workloads beyond evaluating large many-to-many joins, such as running predicates on node and edge properties or grouping and aggregations, for which RDBMSs already employ efficient techniques. Therefore leveraging mature RDBMS technology to support graph workloads natively is highly appealing to both users and vendors: users can avoid the challenges of duplicating data and keeping multiple systems in sync, while vendors can avoid the efforts to develop a new system from scratch. We revisit this goal and research challenge in the context of columnar RDBMSs, which are similar to GDBMSs in that they also target read-heavy analytical workloads. Our specific goal is to extend a columnar RDBMS natively with the fast join capabilities of GDBMSs.

Several prior approaches leverage RDBMS technology to evaluate graph workloads. One approach simply exposes a separate graph querying layer to users and implements a translation component that outputs SQL versions of queries, with no or minimal modifications to the query processor of the RDBMS. This approach is not focused on performance and is commonly employed in commercial products, such as IBM DB2 Graph [48], SQLGraph [47], SAP Hana's graph database extension [44].

A second approach introduces a new graph-specific query processor that co-exists with the existing processor of the RDBMS. This has been most recently adopted by the GR-Fusion system [23, 24]. Specifically, SQL is extended to contain graph-specific constructs, using which users create graphs. The topologies of these views, i.e., the vertices and edges without properties, are stored in native adjacency list indexes, which are used during query processing for graph traversals/many-to-many joins, using new graph-specific operators, such as `EdgeScan` and `PathScan`. Parts of queries that refer to graph-specific constructs compile to these specialized graph operators, while the non-graph parts of queries compile to existing operators of the RDBMS. GQ-Fast [32] is another system that develops a separate query processor and storage sub-system specialized for graphs. GQ-Fast is not integrated into an RDBMS but the authors' envisioned integration [32] is similar to GR-Fusion's dual processor approach. Aside from being heavy-weight integration approaches that develop separate graph-specific components within

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[Making RDBMSs Efficient on Graph Workloads Through Predefined Joins]

Thank you & Questions?