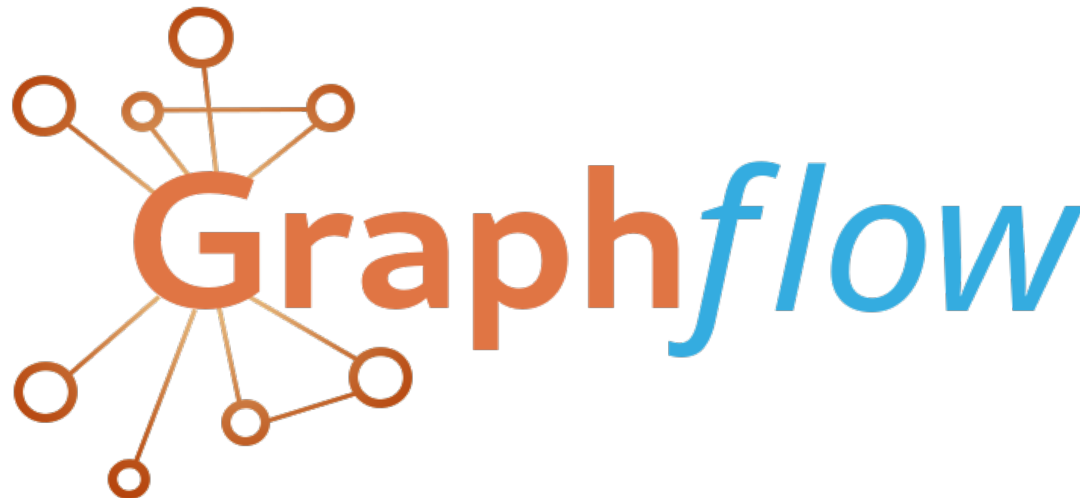
Integrating Column-Oriented Storage and Processing Techniques into GraphflowDB

Semih Salihoğlu

Joint w/ Pranjal Gupta, Amine Mhedhbi



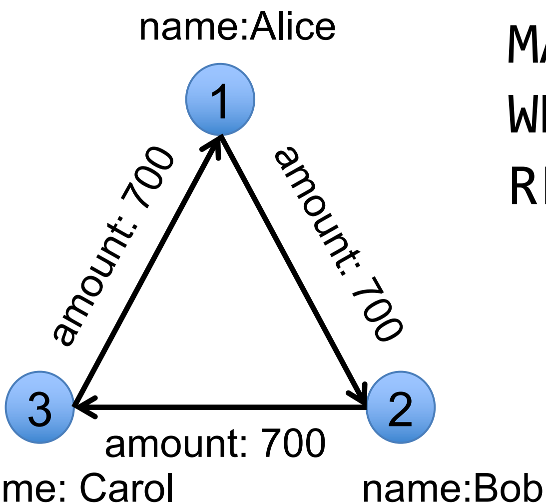
UNIVERSITY OF WATERLOO | DSg Data Systems Group

Graph Database Management Systems Overview

- Read-optimized systems designed for analytical workloads with large many-to-many (n-n) joins

Data Model

Labeled Graph



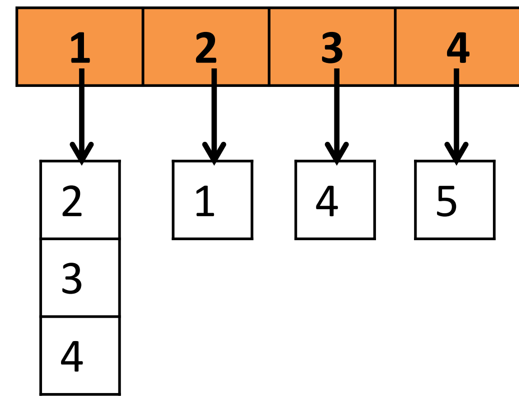
Query Language

Graph-specific SQL

```
MATCH a->b->c,c->a
WHERE a.age > 30
RETURN (a,b).amount
```

System

Storage: Graph-specific



Query Processor: Fast traversals, i.e., nested index loop joins

Differences Between "Native GDBMSs" vs RDBMSs

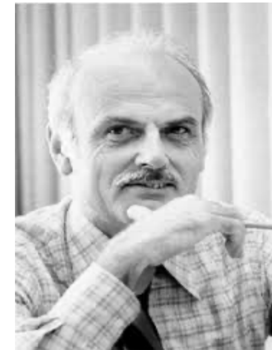
1. Pre-defined but fast joins (access paths) of n-n relationships

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.

Differences Between “Native GDBMSs” vs RDBMSs

1. Pre-defined but fast joins (access paths) of n-n relationships
2. Semi-structured data model
   - No fixed schema
3. More support for variable-length recursive join queries

“Give me all direct or indirect possible sources of money flow into Alice’s account from Canada.”

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

Can be done in recursive SQL but harder

Graphflow

- In-memory Graph DBMS: Property graph model & openCypher
- Two primary features:
  1. Very fast joins of n-n relationships:

GraphflowDB	RDBMS Literature
Multiway intersection-based joins	WCO joins
List-based Query Processor	Factorized DBs
A+ Indexes: Flexible Adjacency List Indexes	Partitioned and Compressed Materialized Views

  2. Scalability: compressed, in-memory columnar storage

Column Stores: Read-optimized for Analytical Workloads

Storage

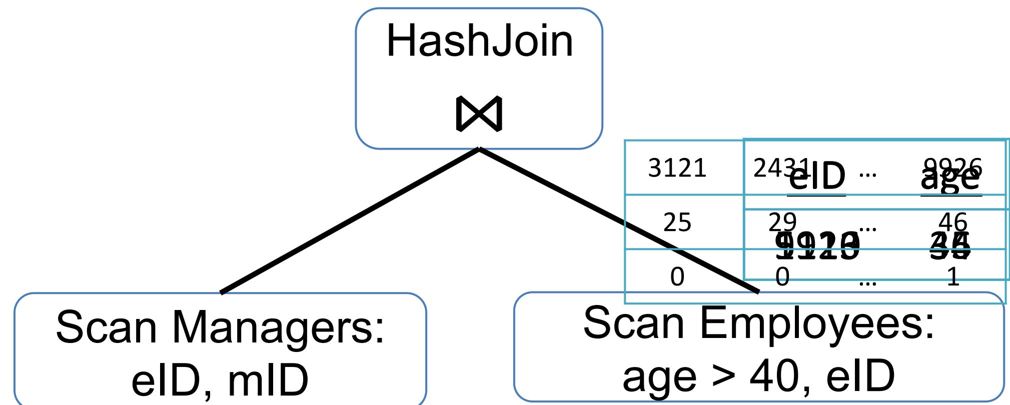
eID	age	salary
3121	25	2000
2431	29	2000
1113	34	2000
5110	35	2000
9926	46	3000

- Offset-based access
- Compression: e.g. RLE

salary val, start, len
2000, 0, 4
3000, 4, 1

Query Processor

```
SELECT *
FROM Employee E, Manager M
WHERE E.eID = M.eID AND age > 40
```



- Block at-a-time
  - Good CPU utilization

eID	3121	2431	1113	5110	9926
age	25	29	34	35	46
fmask	0	0	1	1	1

But not optimized for n-n joins and access patterns of graph workloads

Outline

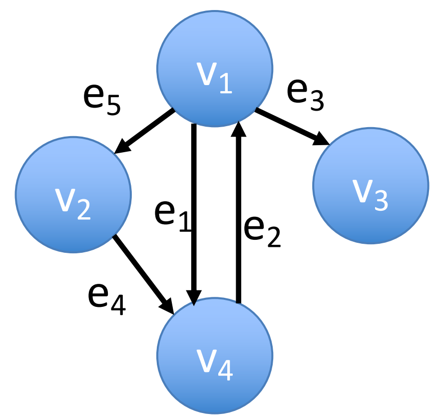
- Overview of Access Patterns in GDBMSs
- Example Columnar Data Structures & Compression Scheme: Null Compression
- List-based Query Processing

Outline

- Overview of Access Patterns in GDBMSs
- Example Columnar Data Structures & Compression Scheme: Null Compression
- List-based Query Processing

Overview of Access Patterns in GDBMSs

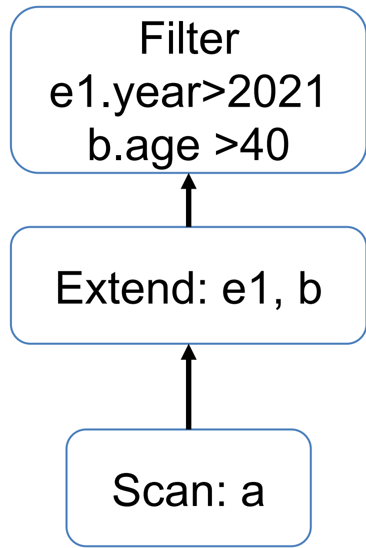
```
MATCH (a:Person)-[e:LIKES]->(b:Person)
WHERE b.age > 40 & e.year > 2021
RETURN a, b
```



1 →	(e_5,v_2)	(e_1,v_4)	(e_3,v_3)
2 →	(e_4,v_4)		
3 →	null		
4 →	(e_2,v_1)		

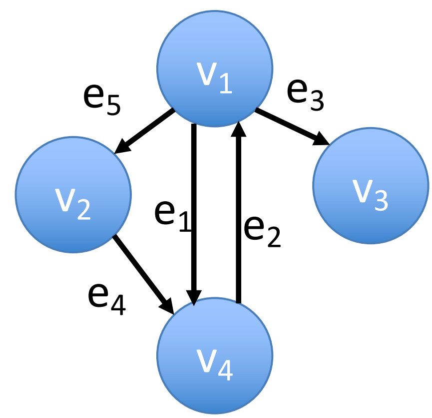
year
2001
2020
2017
1995
2003

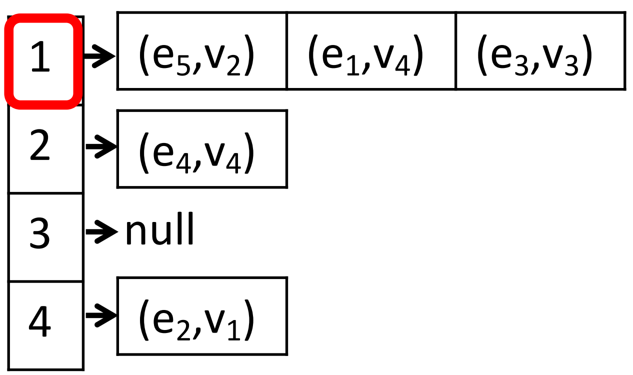
age
46
22
23
65



Overview of Access Patterns in GDBMSs

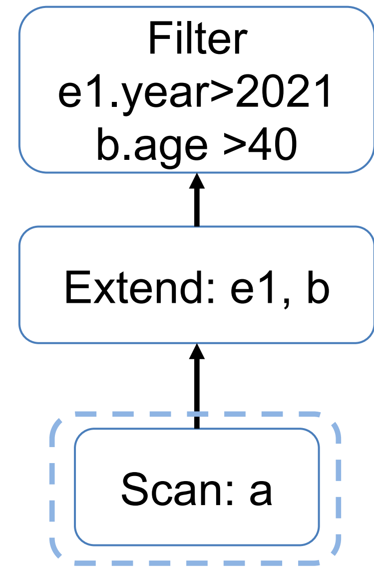
```
MATCH (a:Person)-[e:LIKES]->(b:Person)
WHERE b.age > 40 & e.year > 2021
RETURN a, b
```





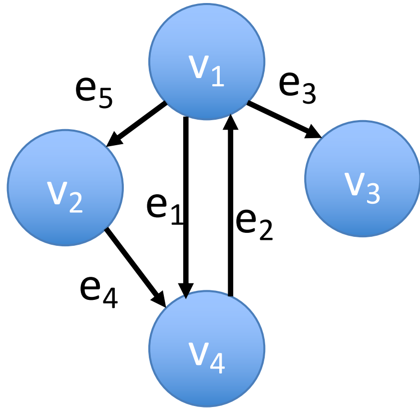
year
2001
2020
2017
1995
2003

age
46
22
23
65



Overview of Access Patterns in GDBMSs

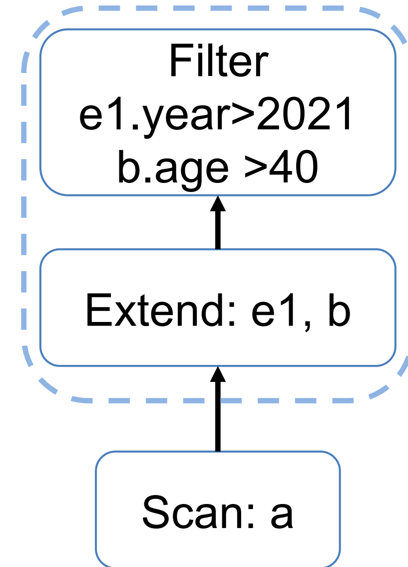
```
MATCH (a:Person)-[e:LIKES]->(b:Person)
WHERE b.age > 40 & e.year > 2021
RETURN a, b
```



1 →	(e_5,v_2)	(e_1,v_4)	(e_3,v_3)
2 →	(e_4,v_4)		
3 →	null		
4 →	(e_2,v_1)		

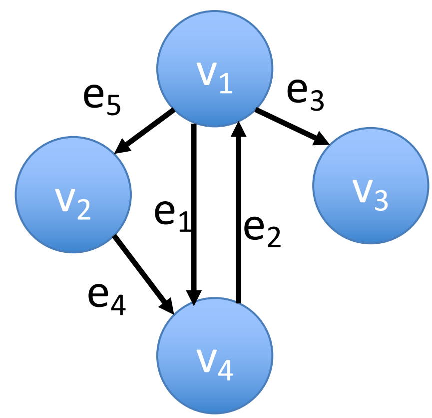
year
2001
2020
2017
1995
2003

age
46
22
23
65



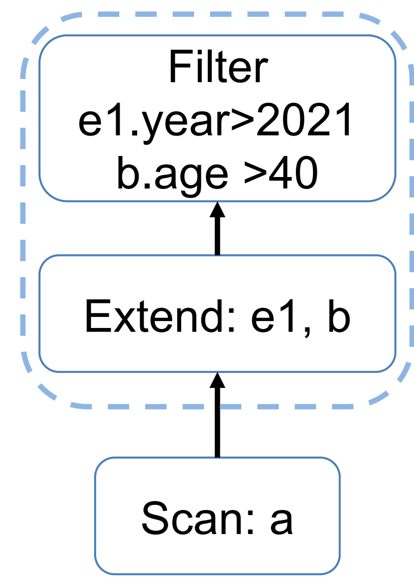
Overview of Access Patterns in GDBMSs

```
MATCH (a:Person)-[e:LIKES]->(b:Person)
WHERE b.age > 40 & e.year > 2021
RETURN a, b
```



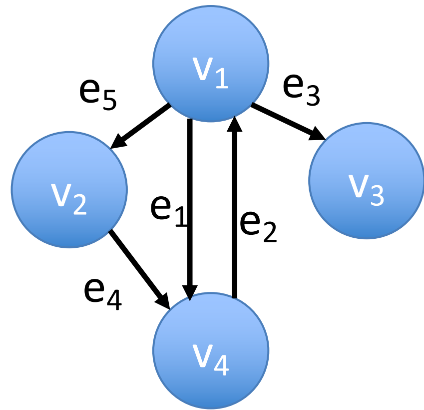
1 →	(e_5,v_2)	(e_1,v_4)	(e_3,v_3)
2 →	(e_4,v_4)		
3 →	null		
4 →	(e_2,v_1)		

year	**age**
2001	46
2020	22
2017	23
1995	65
2003	



Overview of Access Patterns in GDBMSs

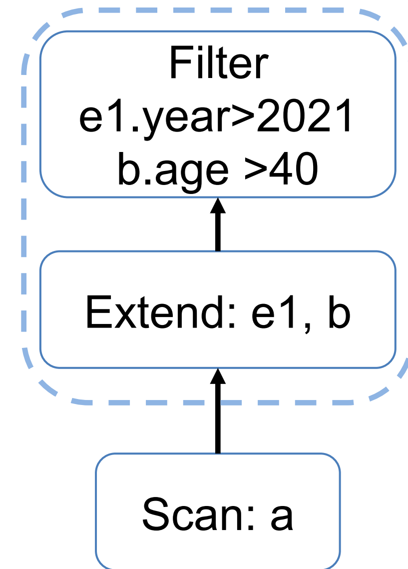
```
MATCH (a:Person)-[e:LIKES]->(b:Person)
WHERE b.age > 40 & e.year > 2021
RETURN a, b
```



1 →	(e_5,v_2)	(e_1,v_4)	(e_3,v_3)
2 →	(e_4,v_4)		
3 →	null		
4 →	(e_2,v_1)		

year
2001
2020
2017
1995
2003

age
46
22
23
65



Edge/vertex properties are read in the order they appear in adj. lists.

Desiderata 1: Store edge (but not vertex) properties in the same order

- b/c each edge accessed from 2, but each v from deg(v) locations

Desiderata 2: Require constant decompression time for vertex props.

- b/c access won't be sequential (unless system incurs deg(v) replication)

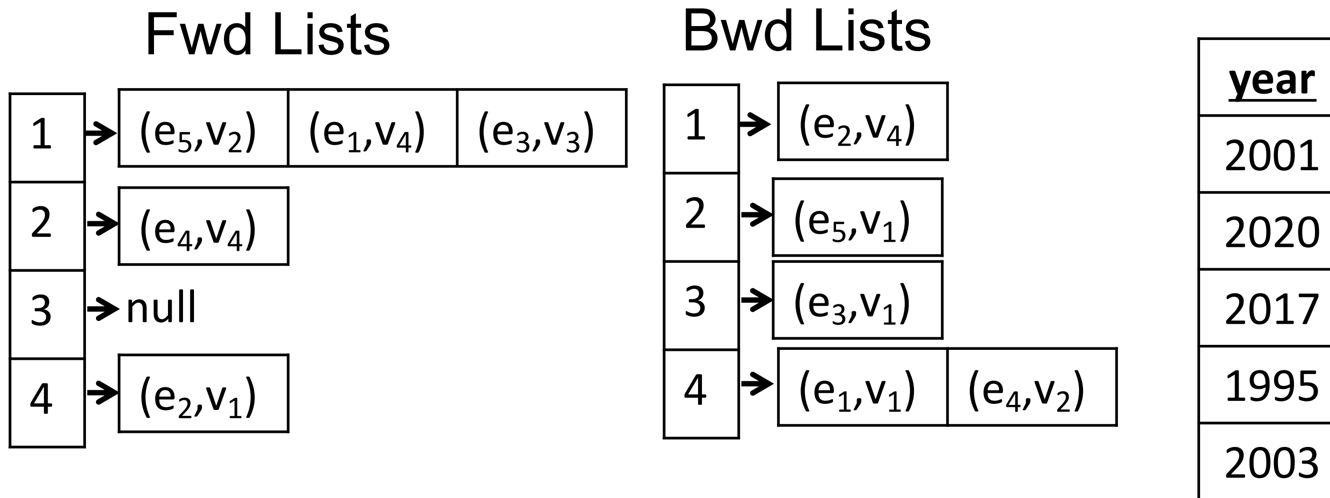
Outline

- Overview of Access Patterns in GDBMSs
- Example Columnar Data Structures & Compression Scheme: Null Compression
- List-based Query Processing

DSg Data Systems Group

Example Columnar Data Structures: Edge Properties (1)

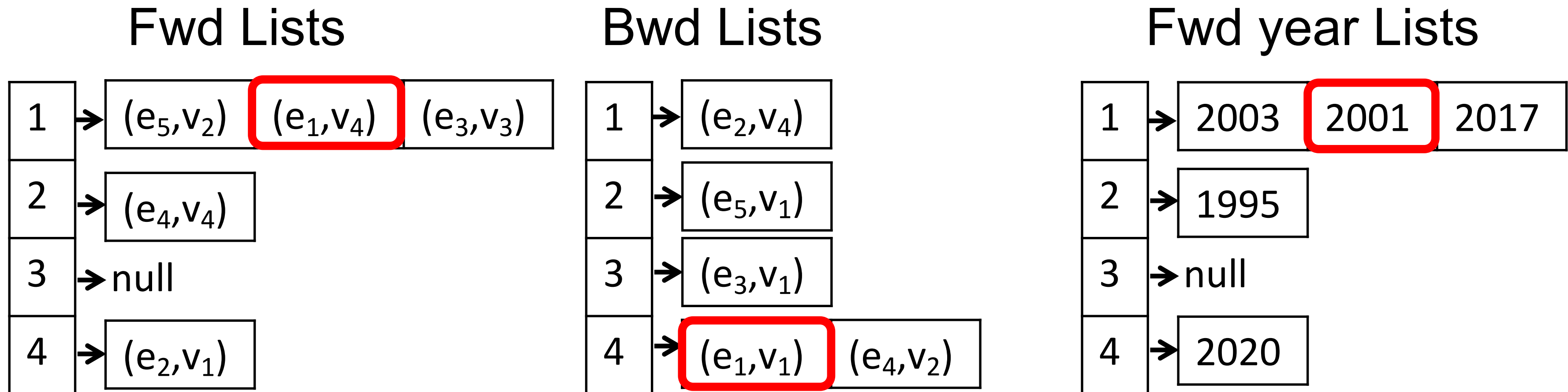
- Recall Desiderata 1: Store edge properties in order of adj lists.
- Option 1: Vanilla edge columns. Random access on both directions.



- Option 2: Double-indexed edge property lists

Example Columnar Data Structures: Edge Properties (2)

- Option 3: Single-indexed edge property lists



- Sequential reads in fwd direction.
- Can we get random backward access?
  - Solution: edge ID scheme w/ list-level offsets
    - E.g: e_1: (edge label, srcID, list-offset), so e_1: (KNOWS, v_1, 2)
  - *Unlike columnar RDBMS, GDBMS physically store IDs/offsets*
  - Problem: deletions leave gaps; recycling list-level offsets is hard.

Final implementation is a variant of this data structure called Edge Property Pages that is more update friendly

Example Compression Schemes: Null Compression (1)

- Desiderata 2: Require constant decompression time for vertex props.
- Existing schemes designed for sequential access [Abadi, CIDR 07]
- For random access: compute in O(1) the *rank* of positions in bitmap
- Solution: Enhance w/ Jacobson's rank index [Jacobson, FOCS '89]

original:

v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
--	--	7	--	6	3	2	--	--	8	11	--	2	3	--	--

bitmap: 0 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0

prefix sums: 0 1 4 6

nonNulls: 7 6 3 2 8 11 2 3

?

value at pos 9 = nonNulls[4+1] = 8

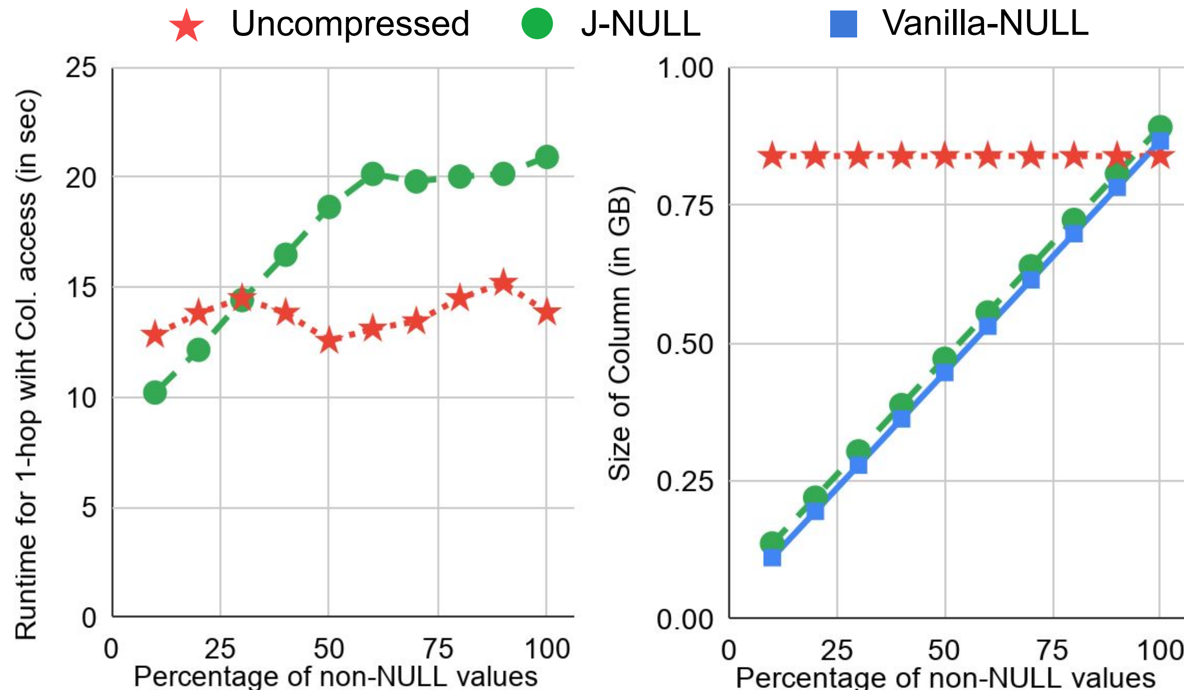
	0	1	2	3
	…	…	…	…
0101	0	1	1	2
0110	0	1	2	2
0111	0	1	2	3
	…	…	…	…

Bit pos-to-index map

Example Compression Schemes: Null Compression (2)

- LDBC100 w/ different NULL densities on creationDate column (220M)

```
MATCH (a:Person)-[e:LIKES]->(b:Comment)
RETURN b.creationDate
```



Within 1.2x-1.5x of uncompressed; faster w/ density is <30%

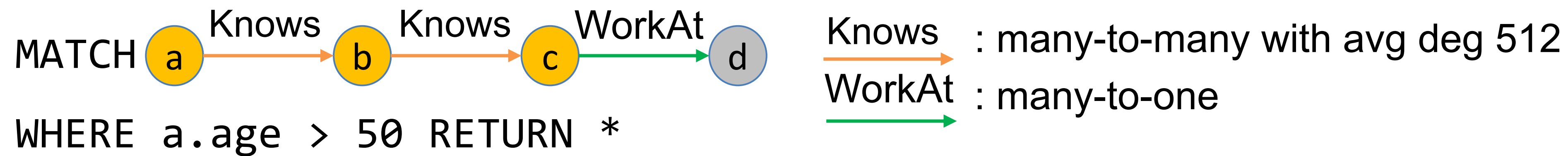
Used to compress properties as well as empty lists in CSRs.

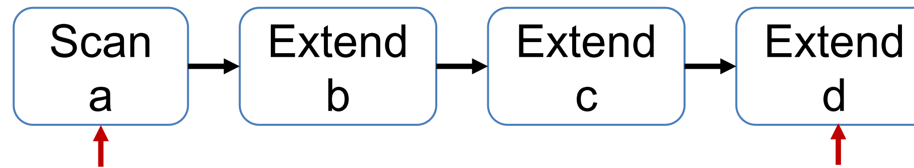
Data Systems Group

Outline

- Overview of Access Patterns in GDBMSs
- Example Columnar Data Structures & Compression Scheme: Null Compression
- List-based Query Processing

Traditional Block-based Processing on N-N Joins





a	age	fmask
a_1	51	1
a_2	19	0
…	…	…
a_{1024}	60	1

a	age	b	c	d	fmask
a_1	51	b_1	c_1	d_1	1
a_1	51	b_1	c_2	d_2	1
…	…	…	…	…	…
a_1	51	b_1	c_{512}	d_{512}	1
a_1	51	b_2	c_{513}	d_{513}	1
…	…	…	…	…	…
a_1	51	b_2	c_{1024}	d_{1024}	1

Problem 1: Value repetition (or selector indices) b/c 1 group of vectors is used

Problem 2: Fixed length blocks (e.g., 1024) which don’t align with adj. list sizes

List-based Processor (1)

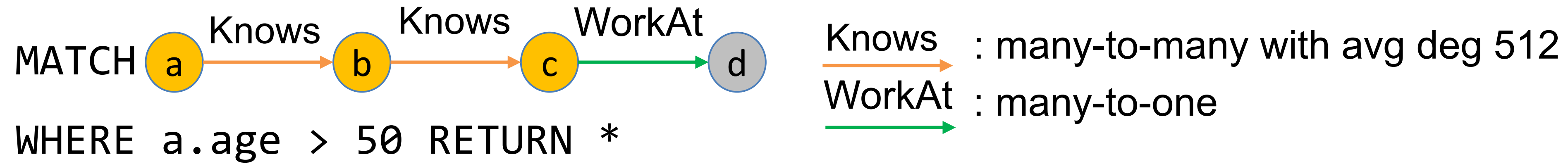
- Factorization [Olteanu, SIGMOD Rec. ‘16]

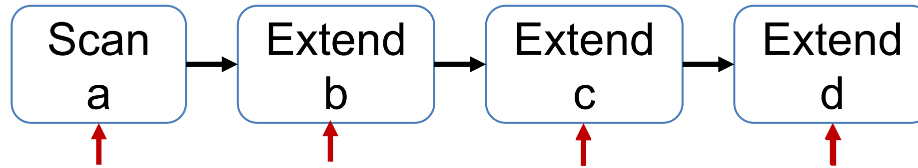
a	age	b	c	d
a_1	51	b_1	c_1	d_1
a_1	51	b_1	c_2	d_2
…	…	…	…	…
a_1	51	b_1	c_{512}	d_{512}
a_1	51	b_2	c_{513}	d_{513}
…	…	…	…	…
a_1	51	b_2	c_{1024}	d_{1024}

<table>
<tr><th>a, age</th><th></th><th>b</th><th></th><th>c, d</th></tr>
<tr><td>{a_1, 51}</td><td>X</td><td>{b_1}</td><td>X</td><td>{[c_1, d_1] [c_2, d_2] …, {c_{512}, d_{512}]}</td></tr>
<tr><td colspan="5">U</td></tr>
<tr><td>{a_1, 51}</td><td>X</td><td>{b_2}</td><td>X</td><td>{[c_{513}, d_{513}] [c_{514}, d_{514}] …, {c_{1024}, d_{1024}]}</td></tr>
</table>

- *List groups*
  - Represent intermediate tuples in multiple vector groups
  - Variable-sized vectors aligned to adjacency lists
  - Vectors storing node and edge IDs are pointers to lists in storage

List-based Processor (2)





List Group 1

a	age	fmask
a_1	51	1
a_2	19	0
…	…	…
a_{1024}	60	1

curIdx = 1

List Group 2

b	fmask
b_1	1
b_2	1
…	…
b_{512}	1

curIdx = 1

List Group 3

c	d	fmask
c_{513}	d_{513}	11
c_{514}	d_{514}	11
…	…	…
c_{1024}	d_{1024}	11

curIdx = -1

List-based Processor (3)

- Modified version of LDBC SNB Interactive Complex Read queries
- Scale Factor 10, 30M Vertices and 176.6M Edges
- 2.6GHz CPU, 512GB RAM, single threaded execution
- GF-RT: Graphflow w/ old **r**ow-based storage & **t**uple-at-a-time Volcano processor

	IC1	IC2	IC4	IC5	IC6	IC7	IC8	IC9	IC11	IC12
GF-LBP	36.7	32.4	13.1	1565.2	113.0	3.0	2.6	1519	11.1	34.2
GF-RT	88.4 **2.4x**	45.2 **1.4x**	57.3 **4.4x**	8925.0 **5.7x**	333.1 **3.0x**	6.3 **2.1x**	7.0 **2.7x**	2098 **1.4x**	19.2 **1.7x**	84.9 **2.5x**

runtimes in seconds

Larger differences on queries with joins followed w/ aggregations

Larger differences w/ baseline column stores and Neo4j

Other Techniques

- Other columnar data structures (e.g., one-many or one-one edges)
- ID compression schemes
- More details on LBP

Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta, amine.mhedhbi, semih.salihoglu}@uwaterloo.ca

ABSTRACT

We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on the access patterns in GDBMSs. We then present the design of columnar storage, compression, and query processing techniques based on these requirements. In addition to showing direct integration of existing techniques from columnar RDBMSs, we also propose novel ones that are tailored for GDBMSs. These include a novel list-based query processor, which avoids expensive data copies of traditional block-based processors under many-to-many joins and avoids materializing adjacency lists in intermediate tuples, a new data structure we call single-indexed edge property pages and an accompanying edge ID scheme, and a new application of Jacobson's bit vector index for compressing NULL and empty lists. We integrated our techniques into the GraphflowDB in-memory GDBMS. Through extensive experiments, we demonstrate the scalability and performance benefits of our columnar storage and the query performance benefits of our list-based processor.

1. INTRODUCTION

Contemporary GDBMSs are data management software such as Neo4j [7], Neptune [1], TigerGraph [12], and Graphflow [41, 48] that adopt the property graph data model [9]. In this model, application data is represented as a set of vertices, which represent the entities in the application, directed edges, which represent the relationships between entities, and key-value properties on the vertices and edges.

GDBMSs have lately gained popularity to support a wide range of analytical applications, from fraud detection and risk assessment in financial services to recommendations in e-commerce and social networks [54]. These applications have workloads that search for patterns in a graph-structured database, which often requires reading large amounts of data. In the context of RDBMSs, column-oriented systems [10, 38, 55, 61] employ a set of read-optimized storage, indexing, and query processing techniques to support traditional analytics applications, such as business intelligence and reporting, that also process large amounts of data. As such, these columnar techniques are relevant for improving the performance and scalability of GDBMSs.

In this paper, we revisit columnar storage and query processing techniques in the context of GDBMSs. Specifically, we focus on an in-memory GDBMS setting and discuss the applicability of columnar storage techniques [55], compression schemes [14, 16, 63], and vector-based query processing [17, 24] for storing and accessing different components of the system. Even though analytical workloads that are run on GDBMSs and those on column-oriented RDBMSs exhibit many similarities, they have different fundamental data access patterns. This calls for redesigning columnar techniques in the context of GDBMSs. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for designing the physical data layout and query processor of a GDBMS.

Columnar Storage: Section 4 explores the application of columnar data structures for storing different components of GDBMSs. While existing columnar structures can directly be used for storing vertex properties and many-to-many (n-n) edges, we observe that using a straightforward columnar structure, which we call *edge columns*, to store properties of n-n edges is suboptimal as it does not guarantee sequential access when reading edge properties in either forward or backward directions. An alternative, which we call *double-indexed property CSRs*, can achieve sequential access in both directions but requires duplicating edge properties, which can be undesirable as graph-structured data often contain orders of magnitude more edges than vertices. We then describe an alternative design point, *single-directional property pages*, that avoids duplication and achieves good locality when reading properties of edges in one direction and still guarantees random access in the other when using a new edge ID scheme that we describe. Our new ID schemes allow for extensive compression when storing them in adjacency lists without decompression overheads. Lastly, as a new application of vertex columns, we show that single cardinality edges and edge properties, i.e. those with one-to-one (1-1), one-to-many (1-n) or many-to-one (n-1) cardinalities, are stored more efficiently with vertex columns instead of the structures we described above for properties of n-n edges.

Columnar Compression: In Section 5, we review existing

1

Find on arXiv next Monday

References on GraphflowDB

Graphflow: An Active Graph Database

Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, Semih Salihoglu

Demo
SIGMOD 2017

Optimizing One-time and Continuous Subgraph Queries using Worst-Case Optimal Joins

AMINE MHEDHBI, CHATHURA KANKANAMGE, SEMIH SALIHOGLU, University of Waterloo

Worst-case Opt. Joins
VLDB 2019
TODS 2021

A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems

Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, Semih Salihoglu

A+ Indexes
(ICDE 2021)

Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu

Columnar Techniques
(arXiv 2021)

Graphflow

Data Systems Group

How Are Graphs and GDBMSs Used In Practice?

A User Survey

Sahu et. al. VLDBJ 19 2019

The VLDB Journal (2020) 29:595–618
https://doi.org/10.1007/s00778-019-00548-x

SPECIAL ISSUE PAPER

The ubiquity of large graphs and surprising challenges of graph processing: extended survey

Siddhartha Sahu[1] · Amine Mhedhbi[1] · Semih Salihoglu[1] · Jimmy Lin[1] · M. Tamer Özsu[1]

Received: 21 January 2019 / Revised: 9 May 2019 / Accepted: 13 June 2019 / Published online: 29 June 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

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 performed an extensive study that consisted of an online survey of 89 users, a review of the mailing lists, source repositories, and white papers of a large suite of graph software products, and in-person interviews with 6 users and 2 developers of these products. Our 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. Based on our interviews and survey of the rest of our sources, we were able to answer some new questions that were raised by participants' responses to our online survey and understand the specific applications that use graph data and software. Our study 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, scalability and visualization are undeniably the most pressing challenges faced by participants, and data integration, recommendations, and fraud detection are very popular applications supported by existing graph software. We hope these findings can guide future research.

Keywords User survey · Graph processing · Graph databases · RDF systems

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 [13,20,26,49,65,73,90], RDF engines [52,96], linear algebra software [17,63], visualization software [25,29], query languages [41,72,78], and distributed graph processing systems [30,34,40]. 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 are actually used in practice and the major challenges facing users of graph data, both in industry and in 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?

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00778-019-00548-x) contains supplementary material, which is available to authorized users.

✉ Siddhartha Sahu
s3sahu@uwaterloo.ca

Amine Mhedhbi
amine.mhedhbi@uwaterloo.ca

Semih Salihoglu
semih.salihoglu@uwaterloo.ca

Jimmy Lin
jimmylin@uwaterloo.ca

M. Tamer Özsu
tamer.ozsu@uwaterloo.ca

1 University of Waterloo, Waterloo, Canada

Springer

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

Students

Amine
Mhedhbi

Pranjal
Gupta

Xiyang
Feng

Guodong
Jin

Siddhartha
Sahu

Shahid
Khaliq

Chathura
Kankanamge

Thank you & Questions?