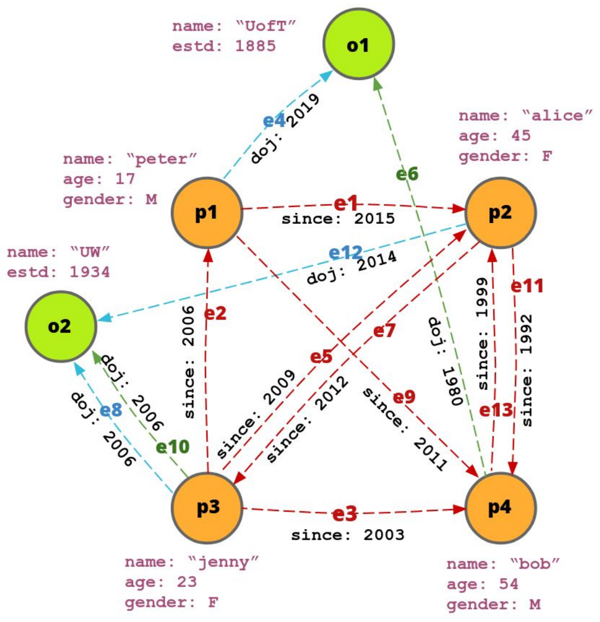
Columnar Storage and List-based Processing for Graph Database Management System

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu

Graph Database Management Systems

Read-optimized analytical systems that can perform fast many-to-many joins.



Property Graph Data Model

p1 →	e1,p2	e9,p4	e4,o1		
p2 →	e7,p3	e11,p4			
p3 →	e2,p1	e3,p4	e5,p2	e8,o2	e10,o2
p4 →	e13,p2				

Forward Adjacency Lists

p2	name	"alice"	age	45	gender	F
p4	name	"bob"	age	54	gender	M
...						
o2	name	"UW"	estd	1934		

Row-oriented Property Store

```
MATCH
(a:PERSON) - [e:FOLLOWS] ➔ (b:PERSON)
WHERE
a.age > 30 & e.since > 2000
RETURN ...
```

Cypher Query

Fast ***many-many joins*** are attributed to the adjacency lists data structure that indexes neighbours of all vertices in both directions, by default.

Analogous to ***Indexed Nested Joins in RDBMS***.

Relevance of Columnar Techniques for GDBMSs

Columnar RDBMSs are optimized for read heavy workloads that requires reading large volumes of data and processing it.

Storage

	PERSON .name	PERSON .age	PERSON .city
p4	\`bob\`	54	TO
p2	\`jenny\`	23	TO
p3	\`alice\`	45	VC
p1	\`peter\`	17	VC

Compression

0011b VC=0, TO=1

Dictionary-encoded PERSON.city

Block-based processing

```
SELECT count(*)
FROM PERSON p
WHERE p.city = `TO`
```

Scan(p.city) → Filter(p.city) → Aggregate

p1	p1	VC	p1	VC	0		
p2	p2	TO	p2	TO	~~1~~		2
p3	p3	VC	p3	VC	0		
p4	p4	TO	p4	TO	1		

BETTER CPU UTILIZATION

GDBMSs can benefit from these techniques.

However ! Not directly applicable because of major differences in the nature of the graph data and access patterns of GDBMSs

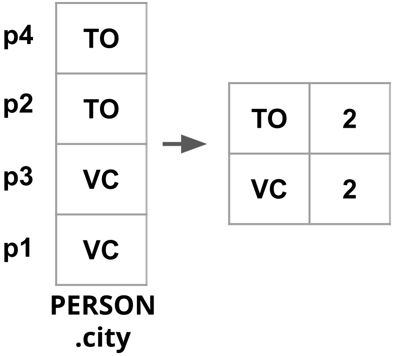
Difference in Access Patterns & Workloads (1)

Columnar RDBMSs

Queries are predominantly operations over large chunks ***sequential*** column data.

Eg. Group by and Aggregates, Filter etc.

```
SELECT p.city, count(*)
FROM Person p
GROUP BY p.city
```

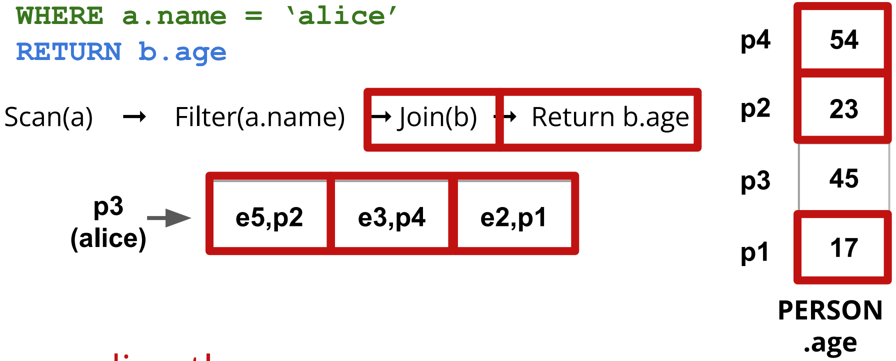


GDBMSs

Queries requires ***random*** access to large chunks of data.

Eg. n-hop queries

```
MATCH (a:Person)-[e:Knows]->(b.Person)
WHERE a.name = 'alice'
RETURN b.age
```



Cannot use vanilla columnar compression techniques directly.

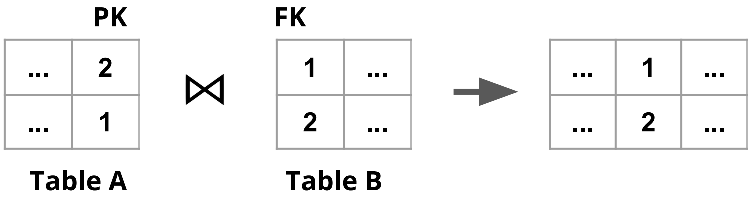
Because these techniques require decompressing an entire block sequentially

Difference in Access Patterns & Workloads (2)

Columnar RDBMSs

Primary Key - Foreign Key joins are prevalent.

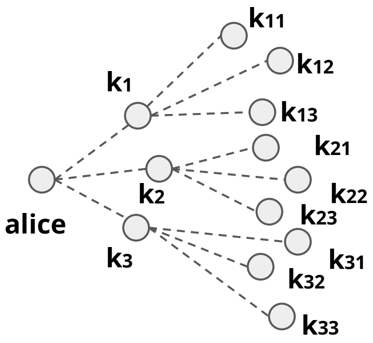
Intermediate join results do not grow.



GDBMSs

Many-to-many joins are prevalent.

Intermediate results grow with ***value replication***.



alice	k_1	k_{11}
alice	k_1	k_{12}
alice	k_1	k_{13}
...	...	...
alice	k_3	k_{32}
alice	k_3	k_{33}

Intermediate tuples

Vanilla Block-based Processing do not avoid value replication and hence, can be inefficient.

Data Systems Group

Research Contributions

	Columnar RDBMSs		GDBMSs
STORAGE	Vanilla append-only Columns	DIRECT ADAPTATION	Vanilla columns for vertex properties.
		NOVEL	Single-index edge property pages.
COMPRESSION		DIRECT ADAPTATION	Dictionary Encoding
	NULL Compression using bit-strings *[Abadi, CIDR 2007]*	NOVEL	NULL Compressing using Jacobson's bit vector index
		NOVEL	Compression using new edge ID scheme
QUERY PROCESSING	Block-based processing	NOVEL	List-based processing

Single-indexed Edge Property Pages

Observation

Edge and Vertex properties are read in the same order as edges in the adjacency lists.

```
MATCH (a:PERSON) - [e:FOLLOWS] ➔ (b:PERSON)
WHERE a = p3
RETURN e.since, b.age
```

[Scan a] ➔ [Filter a = p3] ➔ [Join a with b] ➔ RETURN e.since, b.age

p3 →	e5,p2	e3,p4	e2,p1

e1	2015
e11	1992
e3	2003
e7	2012
e5	2009
e9	2011
e2	2006
e13	1999

KNOWS
since

p4	54
p2	23
p3	45
p1	17

PERSON
age

REQUIRED PROPERTY

Edge properties should be stored & read sequentially in the order edges are stored in the adjacency lists.

However ! We cannot do the same for vertex properties.

Columnar Storage

Simple append-only Columns:

Can be used to store vertex properties.

Direct access to the property value using offset in column.

Suboptimal with edge properties, as does not satisfy our required of sequential reads. **NON-SEQ READS**

p4	54
p2	23
p3	45
p1	17

PERSON
age

Doubly-indexed Edge Property Lists:

Inspired by Adjacency Lists.

SEQ READS

2x DATA DUPLICATION

KNOWS
since

FWD

p1 →	2015	2011	
p2 →	2012	1992	
p3 →	2006	2003	2009
p4 →	1999		

BWD

p1 →	2006		
p2 →	2015	2009	1999
p3 →	2012		
p4 →	2003	2011	1992

Single-indexed Edge Property Lists.

Keep property lists in only one direction.

Provide for random access of the property when edges are read from adjacency list of opposite direction.

SEQ READS IN ONE DIRECTION

For an edge, requires an offset in the appropriate property list to access its property:

p1 →	2015	2011	
p2 →	2012	1992	
p3 →	2006	2003	2009
p4 →	1999		

FWD

New Edge ID scheme: **EDGE LABEL** | **SRC/DEST VERTEX INFO** | **LIST-LEVEL OFFSET**

- SRC/DEST VERTEX INFO: src vertex if the edge properties are indexed in property list by fwd adjacency list
- LIST-LEVEL OFFSET: Offset in the property list

Optimized for updates: **Single-indexed Edge Property Pages**

Evaluation

- Datasets: LDBC100, FLICKR, WIKI
- Queries: n-Hop with either Filter or Aggregation on the last Join
- Configurations: **COL$_E$** - Simple append-only columns for storing edge properties
  PAGE$_E$ - Single-directional edge property pages for edge properties

		LDBC100		WIKI		FLICKR	
		1H	**2H**	**1H**	**2H**	**1H**	**2H**
P_F	COL$_E$	0.55	65.22	2.97	42.92	1.88	888.30
	PAGE$_P$	0.16 **3.4x**	34.22 **1.9x**	0.96 **3.1x**	16.48 **2.6x**	0.42 **4.5x**	189.39 **4.7x**
P_B	COL$_E$	1.23	131.01	6.33	99.28	2.40	1009.84
	PAGE$_P$	1.29 **0.9x**	134.43 **1.0x**	6.10 **1.0x**	91.75 **1.1x**	2.25 **1.1x**	1183.14 **0.9x**

Data Systems Group

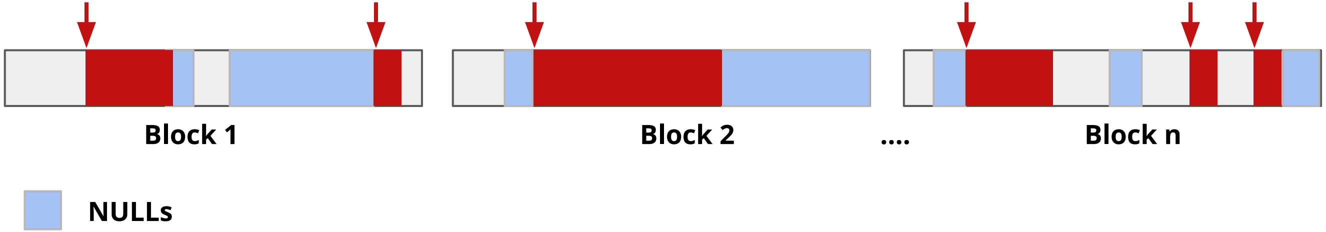
NULL Compression
using Jacobsons' bit vector index

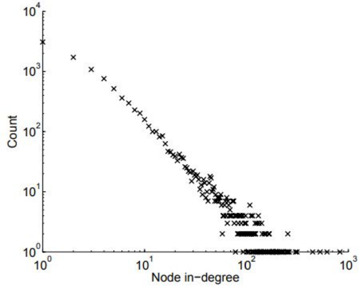
Observation

Properties on edges and vertices can be very sparse.

The degree of most vertices are super small. (by power law)

Reading vertex property (in FILTER) and adjacency list (in JOIN) is like reading small chunk of data followed by random access.





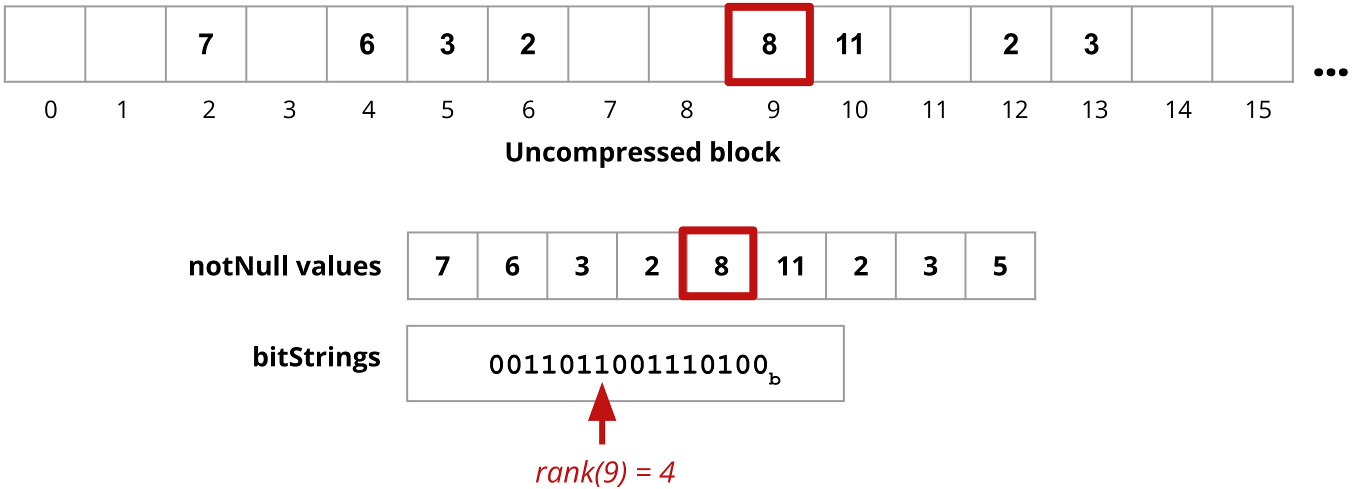
REQUIRED PROPERTY

If compression is used, decompressing arbitrary data elements in a compressed block should happen in constant time.

NULL Compression

Existing solution to compress columns using bit-strings. ***[Abadi, CIDR 2007]***

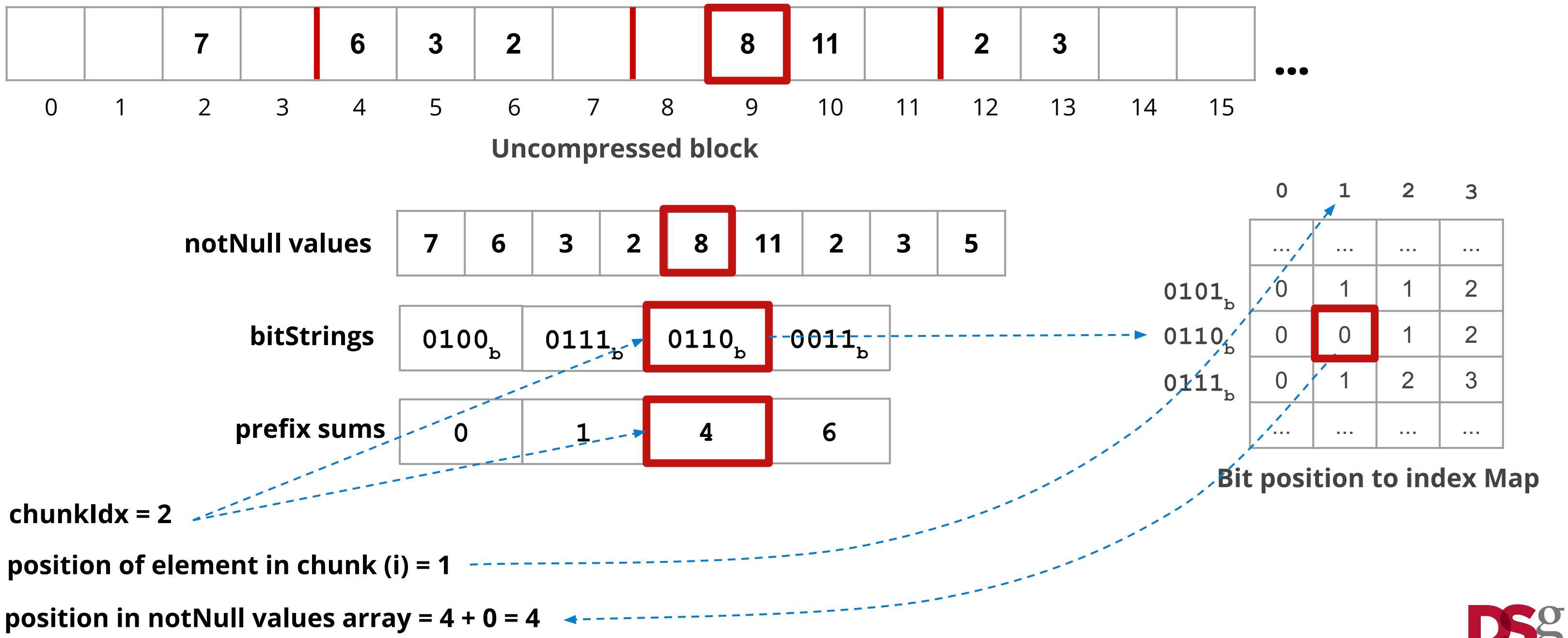
Suitable only for sequential access of column. Do not support reading from arbitrary offsets of columns.



Using Jacobson's bit vector index

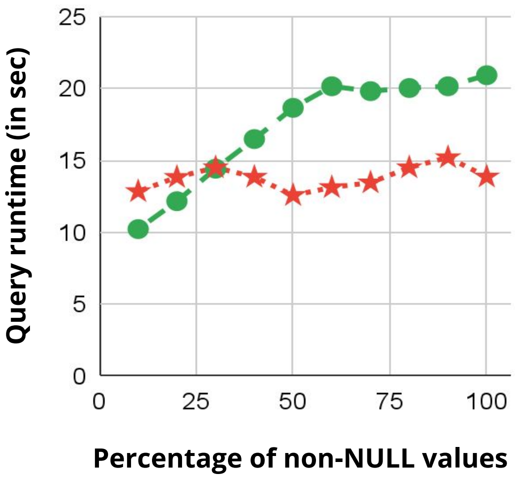
Solution: Calculate the rank of an offset in constant time.

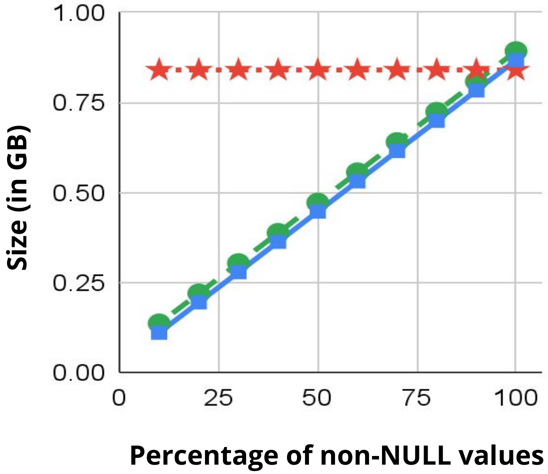
Use Jacobson's rank index. ***[Jacobson, FOCS 1989]***



Evaluation

- Datasets: 1 vertex property column with 220M entries)
  with different %s of NULL values
- Queries: 1-Hop with Property access
- Configurations: ★ **Uncompressed** ● **J-NULL** ■ **Vanilla-NULL**





List-based Processing

Volcano-styled Processing

```
MATCH (a:PERSON)-[e:FOLLOWS]→(b:PERSON)
WHERE a = p3
RETURN e.since, b.age
```

[Scan a] → [Filter a = p3] → [Join a with b] → RETURN e.since, b.age

p3	→	e5,p2	e3,p4	e2,p1

a	e	b	b.age	e.since
p3	[illegible]	[illegible]	[illegible]	[illegible]

e1	2015
e11	1992
e3	2003
e7	2012
e5	2009
e9	2011
e2	2006
e13	1999

KNOWS
since

p4	54
p2	23
p3	45
p1	17

PERSON
age

Bad Cache Locality !

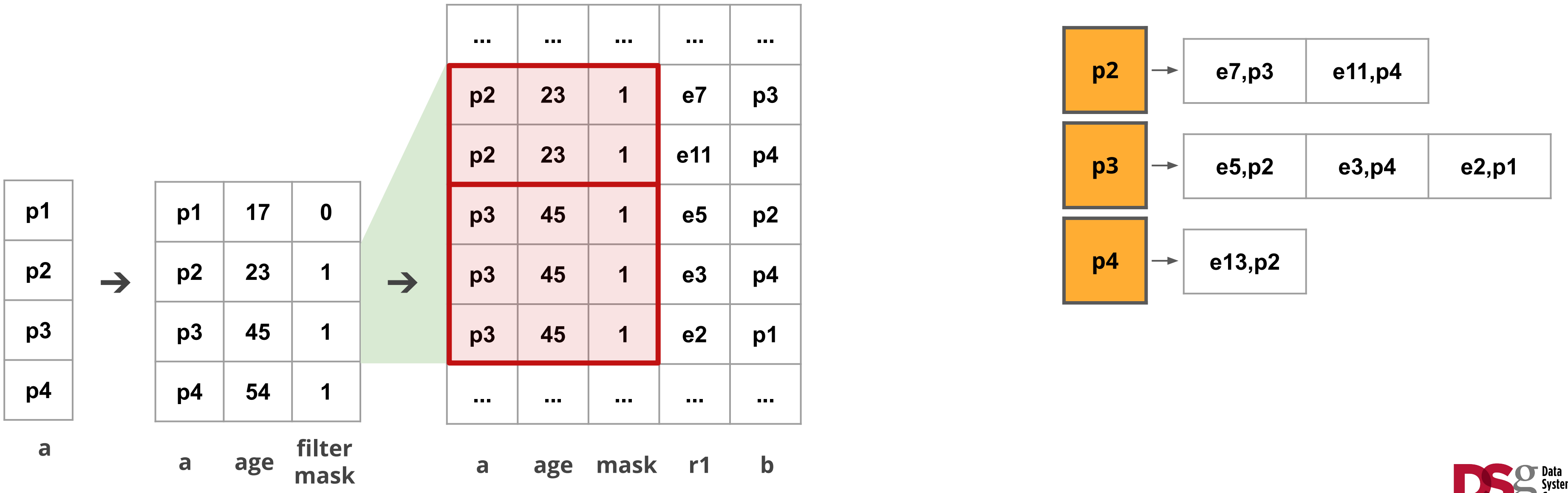
Do not harness the fact that adjacency lists and edge properties are stored sequentially in memory.

Data Systems Group

Shortcomings of Block-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN …



Shortcomings of Block-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] → [Filter a.age > 20] → [Join a with b] → [Join b with c] → RETURN ...

a
p1
p2
p3
p4

→

a	age	filter mask
p1	17	0
p2	23	1
p3	45	1
p4	54	1

→

a	age	mask	r1	b
...	...	...	...	...
p2	23	1	e7	p3
p2	23	1	e11	p4
p3	45	1	e5	p2
p3	45	1	e3	p4
p3	45	1	e2	p1
...	...	...	...	...

→

a	age	mask	r1	b	r2	c
...	...	...	...	...	...	...
p3	45	1	e5	p2	e7	p3
p3	45	1	e5	p2	e11	p4
p3	45	1	e3	p4	e13	p2
p3	45	1	e2	p1	e1	p2
p3	45	1	e2	p1	e9	p4
...	...	...	...	...	...	...

Data Systems Group

Shortcomings of Block-based Processing

```
MATCH  (a:PERSON) - [r1:FOLLOWS] ➔ (b:PERSON)
       (b:PERSON) - [r2:FOLLOWS] ➔ (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN …

Shortcoming #1

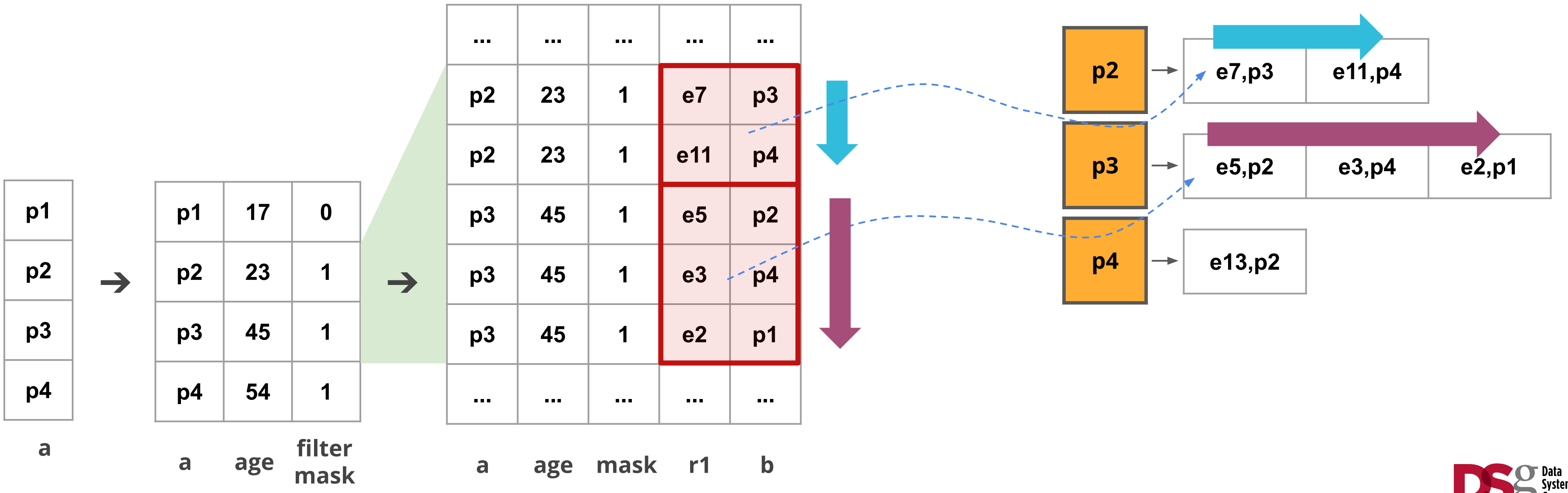
High amount of data replication in intermediate data chunks.

Because intermediate data chunk represents a set of **flat values** using 1 group of vectors.

Shortcomings of Block-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN ...



Shortcomings of Block-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] ➔ (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] ➔ (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN …

Shortcoming #2

Copies exact replicas of adjacency lists to intermediate data chunk.

List-based Processing

Factorized representation of intermediate data [Olteanu, SIGMOD Rec. 2016].

a	age	r1	b
...	...	...	...
p2	23	e7	p3
p2	23	e11	p4
p3	45	e5	p2
p3	45	e3	p4
p3	45	e2	p1
...	...	...	...

→

{p2, 23} X { (e7,p3), (e11,p4) }

U

{p3, 45} X { (e5,p2), (e3,p4), (e2,p1) }

a, age   r1   b

List Groups

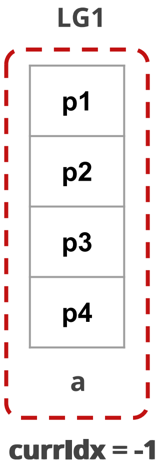
- Instead of 1 group of vectors, **multiple groups** that either be a single tuple or list of tuples.
- Instead of fixed-size vectors, **variable-size vectors** that depend on adjacency list sizes.
- Avoid materializing adjacency lists in list groups

List-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] ➔ (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] ➔ (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN

....



List-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] → [Filter a.age > 20] → [Join a with b] → [Join b with c] → RETURN

....

LG1

a	age	mask
p1	17	0
p2	23	1
p3	45	1
p4	54	1

currIdx = -1

List-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] → [Filter a.age > 20] → [Join a with b] → [Join b with c] → RETURN

....

LG1

a	age	mask
p1	17	0
p2	23	1
p3	45	1
p4	54	1

currIdx = 1

LG2

r1	b
e7	p3
e11	p4

currIdx = -1

p2	e7,p3	e11,p4

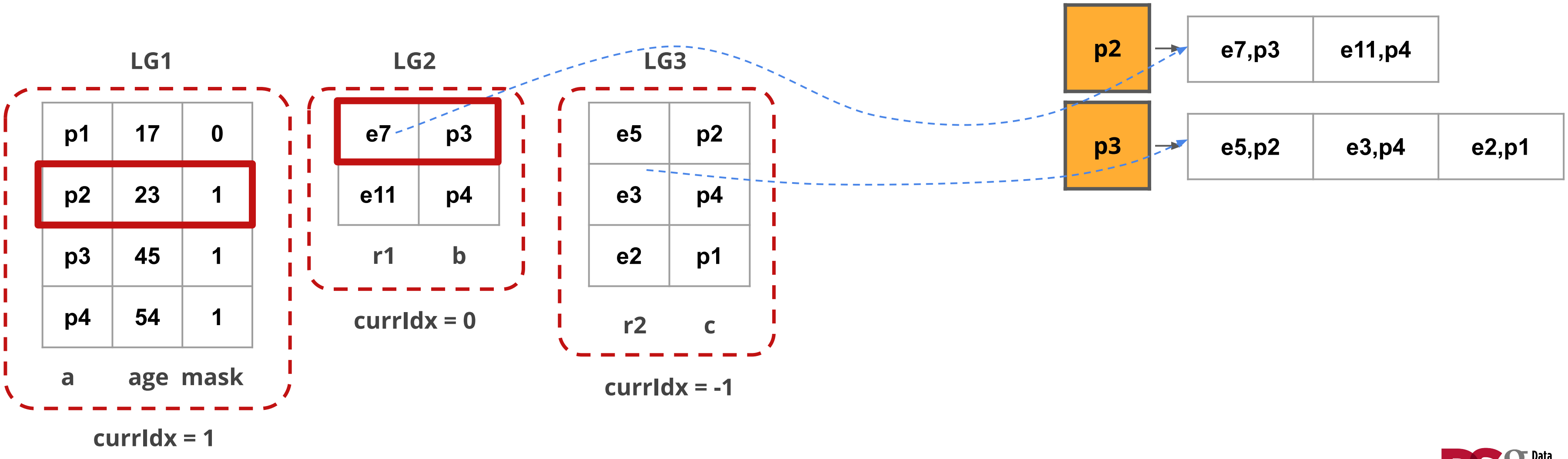
Performance speedups upto 19x on LDBC100

List-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] → [Filter a.age > 20] → [Join a with b] → [Join b with c] → RETURN

....

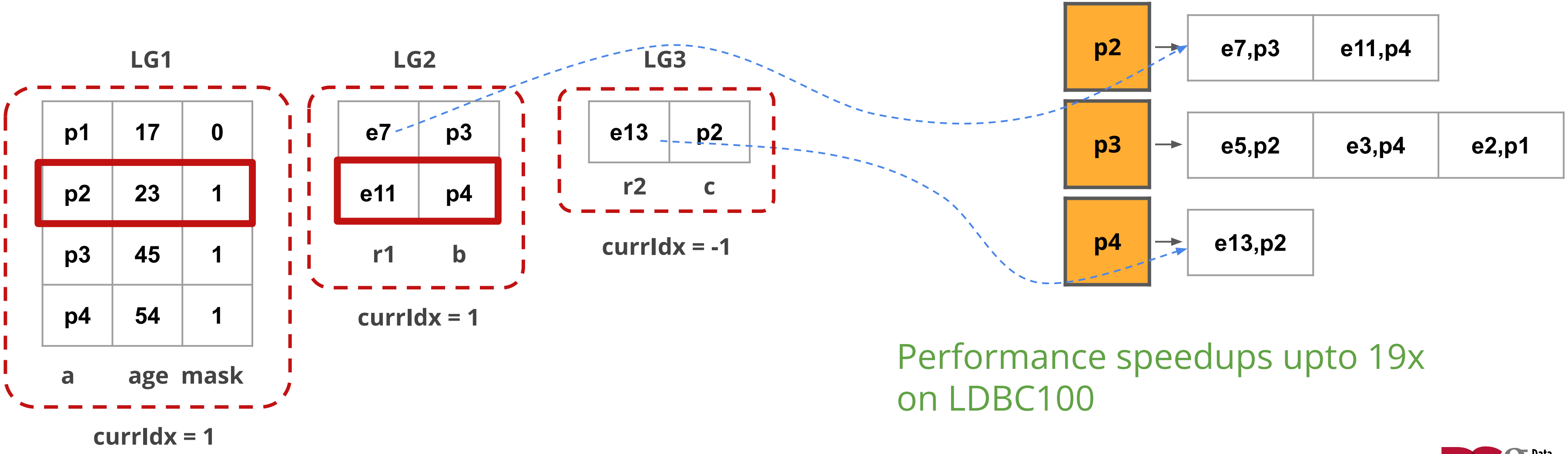


List-based Processing

```
MATCH (a:PERSON) - [r1:FOLLOWS] → (b:PERSON)
      (b:PERSON) - [r2:FOLLOWS] → (c:PERSON)
WHERE a.age > 20
RETURN ...
```

[Scan a] ➔ [Filter a.age > 20] ➔ [Join a with b] ➔ [Join b with c] ➔ RETURN

....



Performance speedups upto 19x on LDBC100

Evaluation

- Queries: n-Hop with either Filter or Aggregation on the last Join
- Configurations: **GF-CV** - Column-oriented, Volcano-styled processing
  GF-CL - Column-oriented, List-based processing
- Datasets: LDBC100, FLICKR, WIKI

			1-hop	2-hop	3-hop
LDBC100	FILTER	GF-CV	24.6	1470.5	40252.4
		GF-CL	7.7	116.2	2647.3
			3.2x	12.7x	15.2x
	COUNT(*)	GF-CV	13.4	241.9	6947.3
		GF-CL	4.2	18.9	357.9
			3.2x	12.8x	19.4x

Data Systems Group

References

Column Stores for Wide and Sparse Data.
Daniel J. Abadi. CIDR 2007

Column-Stores vs. Row-Stores: How Different Are They Really?
Daniel J. Abadi, Samuel Madden, and Nabil Hachem. SIGMOD 2008

Factorized Databases.
Dan Olteanu and Maximilian Schleich. SIGMOD Record 2016.

C-store: A Column-Oriented DBMS.
Mike Stonebraker, et. al.. Making Databases Work 2019

MonetDB: Two Decades of Research in Column-oriented Database Architectures.
Stratos Idreos et. al.. IEEE Data Engineering Bulletin 2012

Space-efficient Static Trees and Graphs.
Guy Jacobson. FOCS 1989

Vectorwise: Beyond Column Stores.
Marcin Zukowski and Peter A. Boncz. IEEE Data Engineering Bulletin 2012

Volcano - An Extensible and Parallel Query Evaluation System
Goetz Graefe. IEEE Transactions on Knowledge and Data Engineering, TKDE 1994

Visit us at: ***http://graphflow.io***

Graphflow

End.

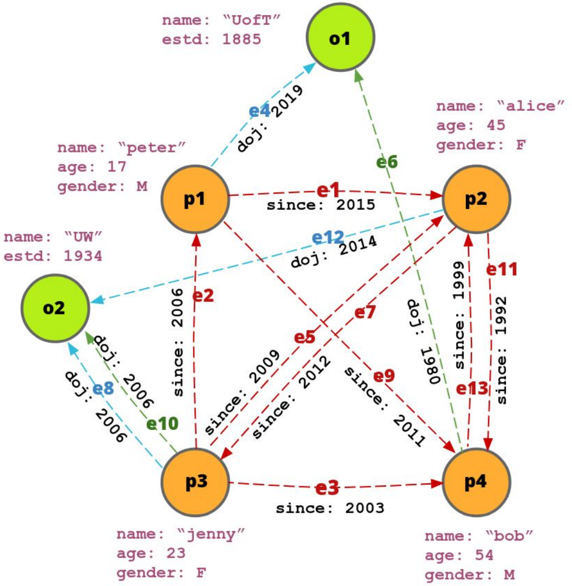
Data Systems Group

Questions?

Extras

Graph Database Management System (GDBMS)

GDBMSs are systems are *read-optimized* systems that are known to support read heavy workloads that predominantly contains large number of *many-to-many joins*.



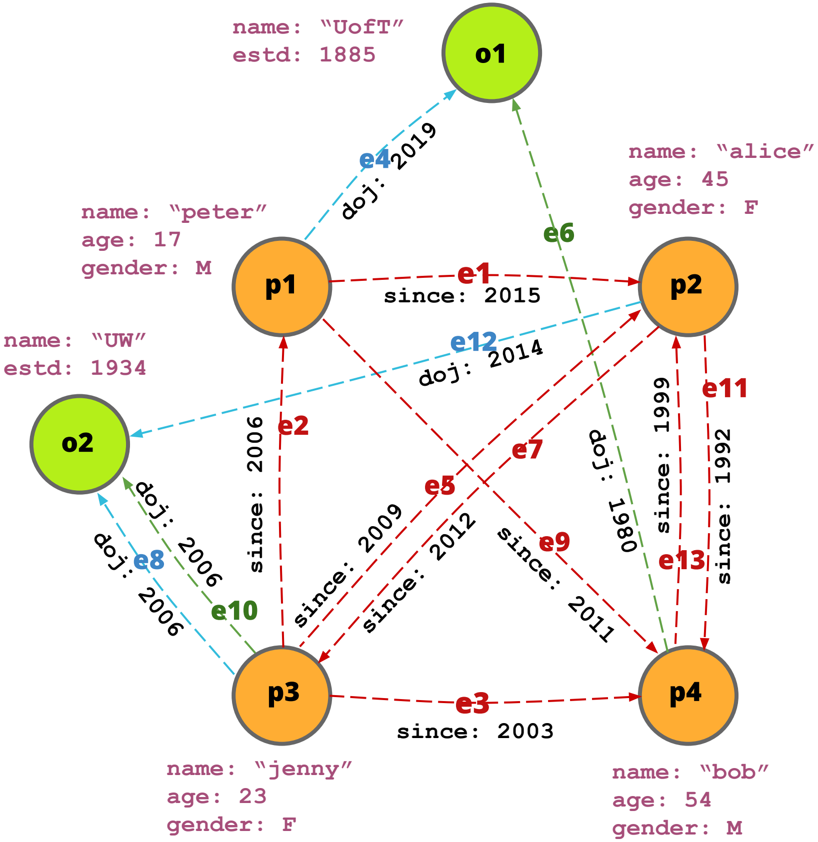
Graph Data

p1 →	e1,p2	e9,p4	e4,o1		
p2 →	e7,p3	e11,p4			
p3 →	e2,p1	e3,p4	e5,p2	e8,o2	e10,o2
p4 →	e13,p2				

Adjacency Lists

Applications like fraud detection, recommendation systems, social networks.

Graph Databases 101: Model



- **Vertices**
- Vertices have a **vertex type**

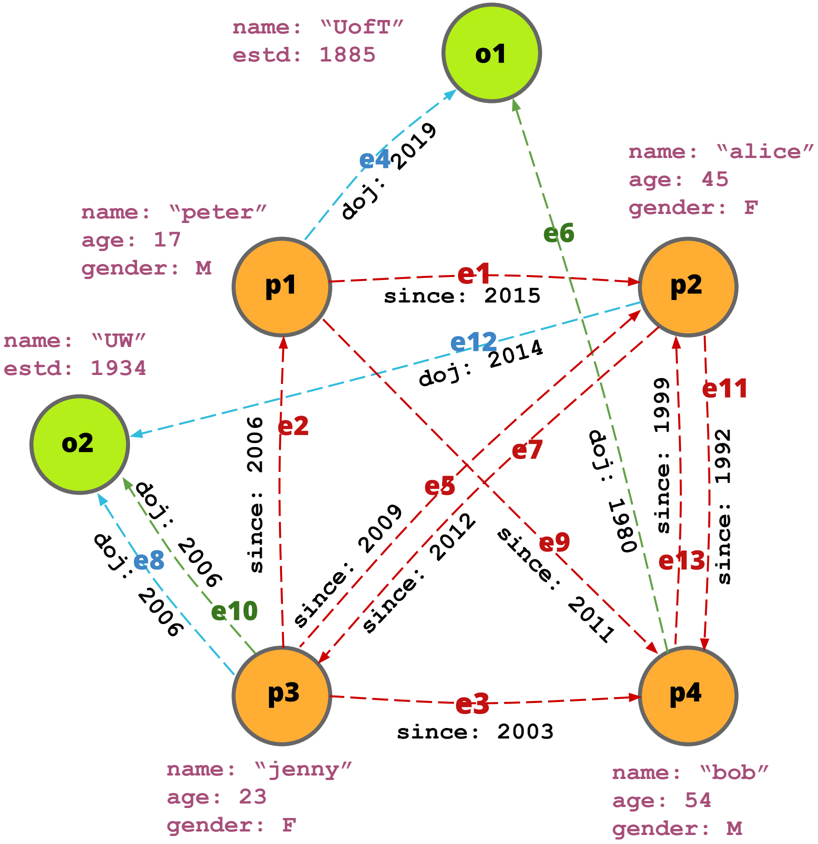
  ORG PERSON

- Directed **edges** connect vertices
- Edges have a **label**

  FOLLOWS WORKAT STUDYAT

- Schemaless, but
  Arbitrary key-value **properties** on edges and vertices

Graph Databases 101: Storage

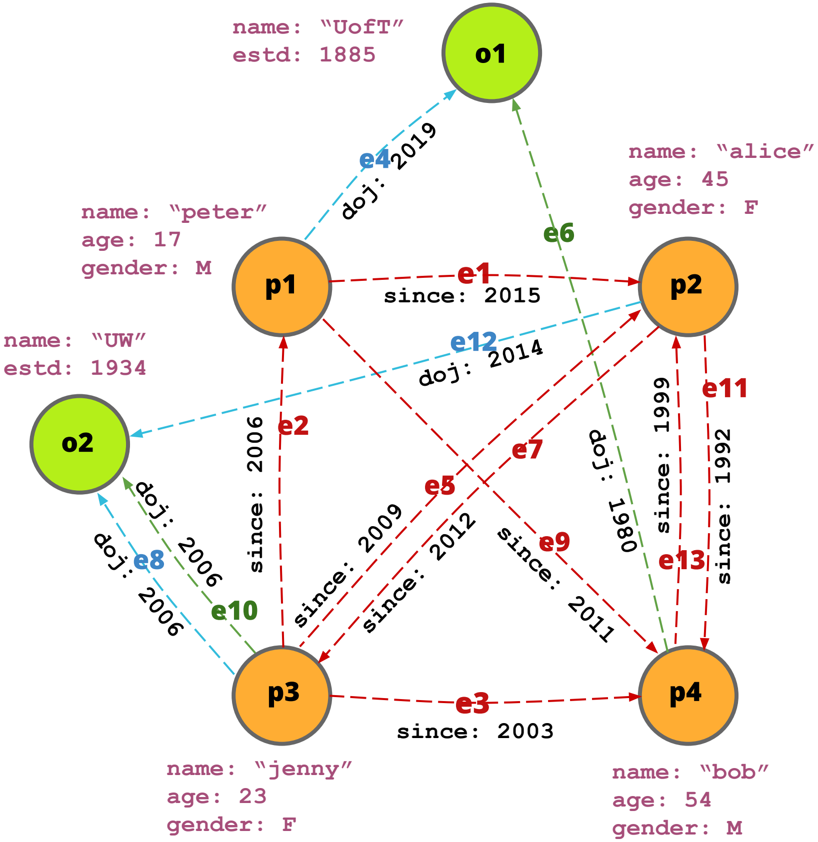


Storing the graph topology:

p1	→	e1,p2	e9,p4	e4,o1		
p2	→	e7,p3	e11,p4			
p3	→	e2,p1	e3,p4	e5,p2	e8,o2	e10,o2
p4	→	e13,p2				
o1						
o2						

Typically a CSR, a columnar structure.

Graph Databases 101: Storage



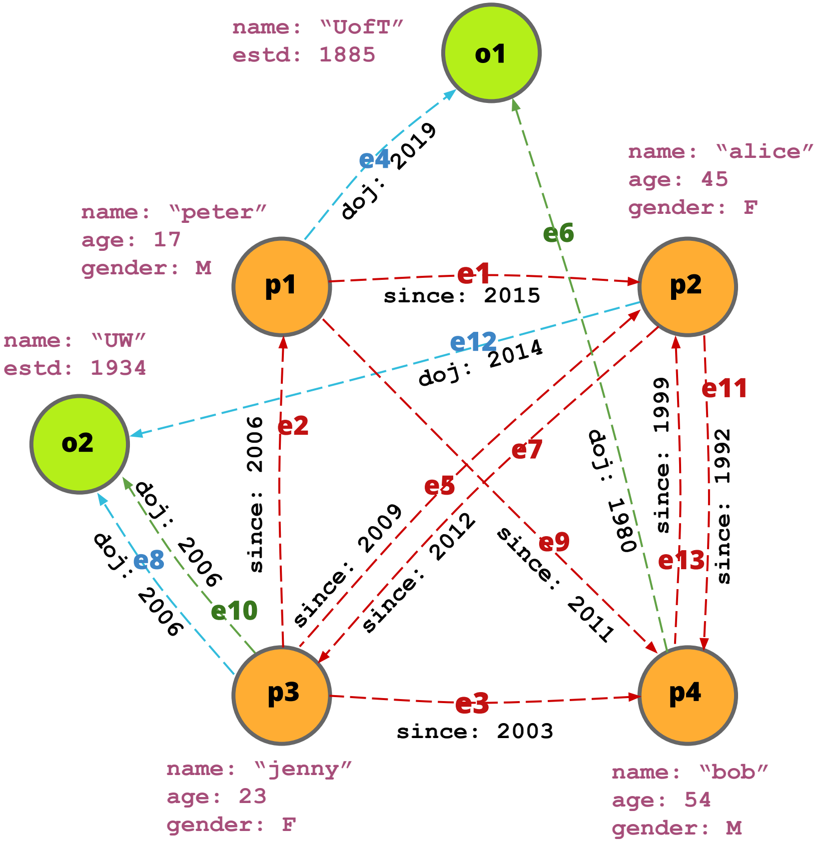
Storing the properties:

p2	name	"alice"	age	45	gender	F
p4	name	"bob"	age	54	gender	M
...						
o2	name	"UW"	estd	1934		

e1	since	2015
e11	since	1992
e2	since	2006
...		

e4	doj	2019
e10	doj	2006
...		

Graph Databases 101: Query Processing



Cypher query:

```
MATCH
(a:PERSON) - [e:FOLLOWS] ➔ (b:PERSON)
WHERE
a.age > 30 & e.since > 2000
RETURN ...
```

subgraph patterns

constraints

Operators for query execution:

- Scan
- Filter
- Join
- GroupBy and Aggregate

Query plan:

[Scan a] ➔ [Filter on a.age] ➔ [Join a with b]

➔ [Filter on e.since] ➔ RETURN ...

DSg Data Systems Group

Column-oriented RDBMS

Introduces a plethora of techniques to make analytical queries fast.

- Columnar storage

  Positional offset based access to values in the column

	PERSON.age	PERSON.gender
p4	54	M
p2	23	F
p3	45	F
p1	17	F

- Columnar compression on homogenous data

Dictionary-encoded PERSON gender	
`0111b`	M=0, F=1

- Late Materialization and Operations over Compressed Data

- Block-based Processing

  Good cache locality. **Not optimized for many-to-many joins**

Motivation

Workloads on GDBMSs and Columnar RDBMS are similar but have fundamentally different access patterns.

Can we push the storage and performance limits of GDBMs by using the techniques that are tailored for achieving high performance in columnar RDBMSs ?

Sub questions:

- **What are the specific requirements for designing the storage layer of GDBMSs ?**
- **Can the existing columnar techniques directly apply to various components of the GDBMSs ?**
- **Can we present new techniques where existing techniques do not directly apply ?**
- **How can we adapt query processor to perform better with many-to-many joins ?**