Graph Database Management System

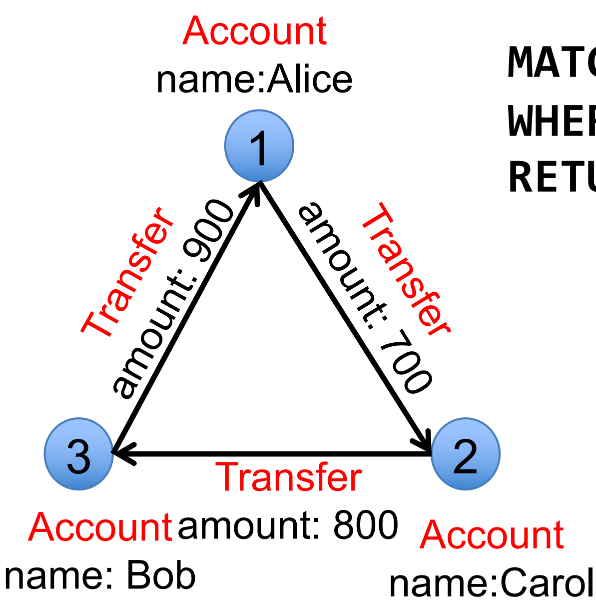
Semih Salihoğlu

(Property) Graph DBMS Overview

- Read-optimized DBMSs targeting app. data modeled as graphs.
  - ex. popular apps: recommendations, fraud detection, highly heterogenous knowledge graph/master data management

Data Model

Labeled Graph



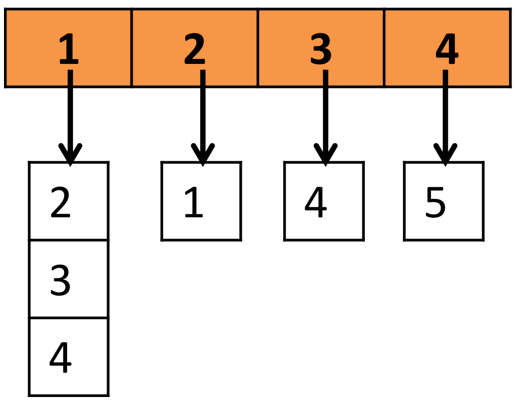
Query Language

Graph-specific SQL

```
MATCH (a)-[:Transfer]->(b)
WHERE a.name = Bob
RETURN b.name
```

System

Graph-specific storage structures, indices, operators



e.g., record ID-based join indices

Differences Between Native GDBMSs vs RDBMSs

1. Pre-defined/Pointer-based joins
   - ➢ common joins are on integer record IDs using a join index
2. Optimized support for m-n joins (later in this talk)
3. Semi-structured data model & URI-heavy datasets
4. Better support for 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

Kùzu Goals: Perfect this feature set: many-to-many and recursive joins, heterogenous/semi-structured data, strings/URIs

Kùzu Current Usability Features

1. Usability features:
   - Property Graph Data Model & Cypher query language
   - DuckDB/SQLite-like embeddable in apps
     - pip install kuzu

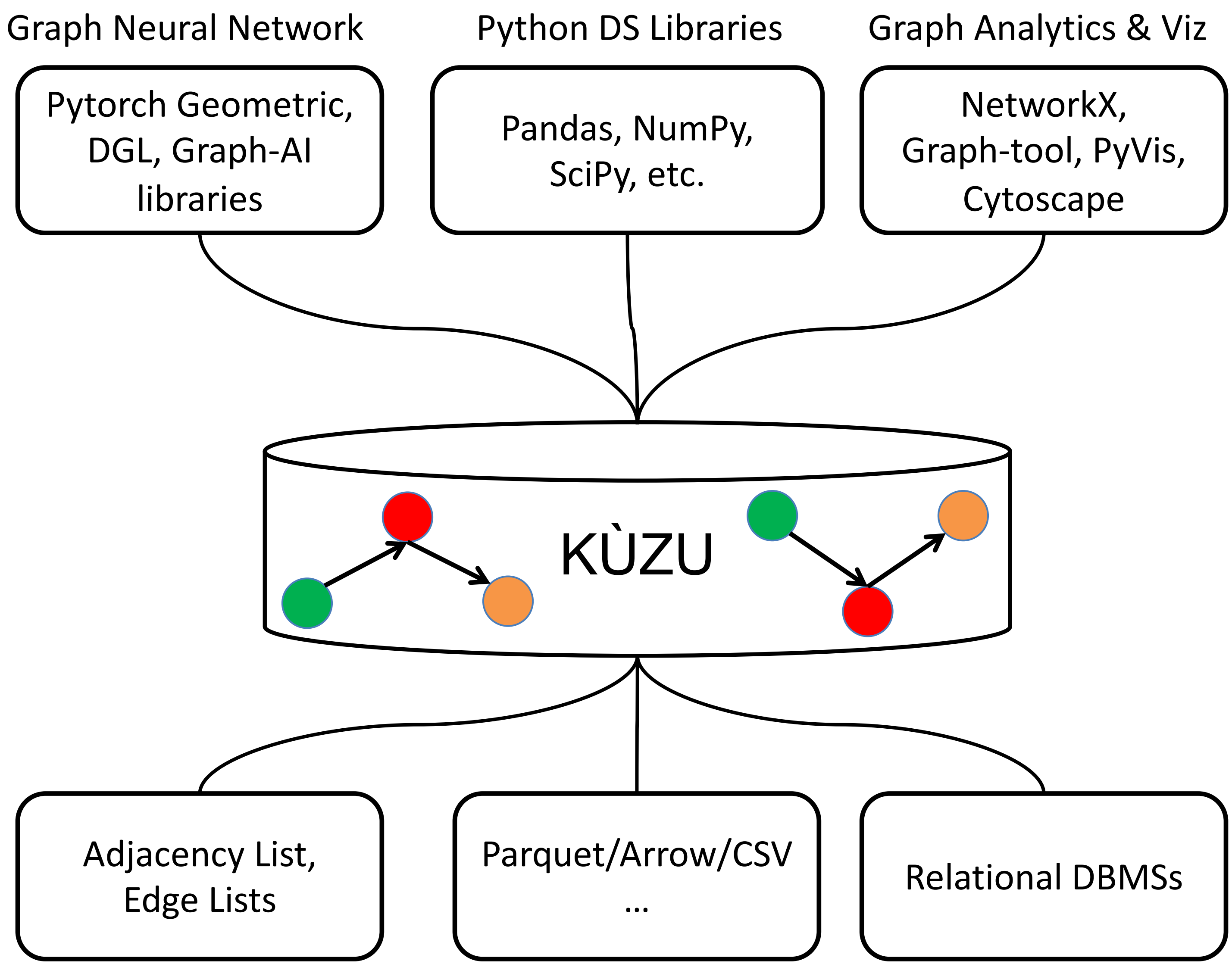
```
import kuzu

db = kuzu.database('./testdb’)
conn = kuzu.connection(db)
results = conn.execute('MATCH (u:User) RETURN u.name;’)
while results.hasNext():
      print(results.getNext())
```

   - Serializable with ACID transactions, i.e., atomic & durable
     - based on write-ahead-logging

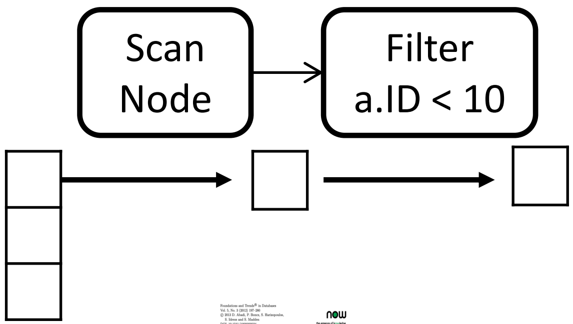
Data Systems Group

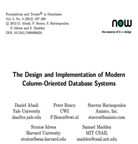
Example Application Domain: Graph Datascience (GDS) Pipelines



Kùzu Current Performance Features

Vectorization



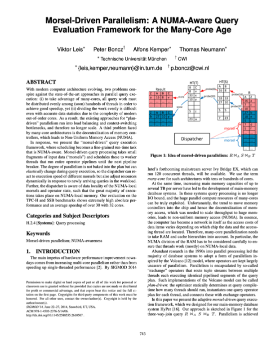


Factorization

<u>b, name</u>		<u>a</u>		<u>c</u>
{L_1, Liz}	X	{U_1,…,U_{100}}	X	{C_1, …, C_{100}}
U				
{L_2, Liz}	X	{U_{100},…,U_{199}}	X	{C_{100},…,C_{199}}

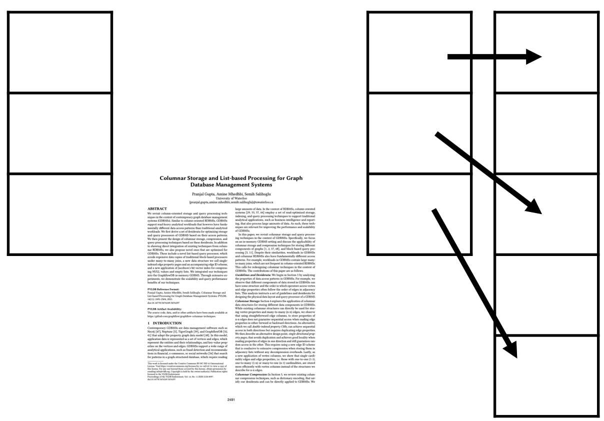
Morsel-driven Parallelism

➢ Scans are "morselized" across threads



Novel Join Algorithms

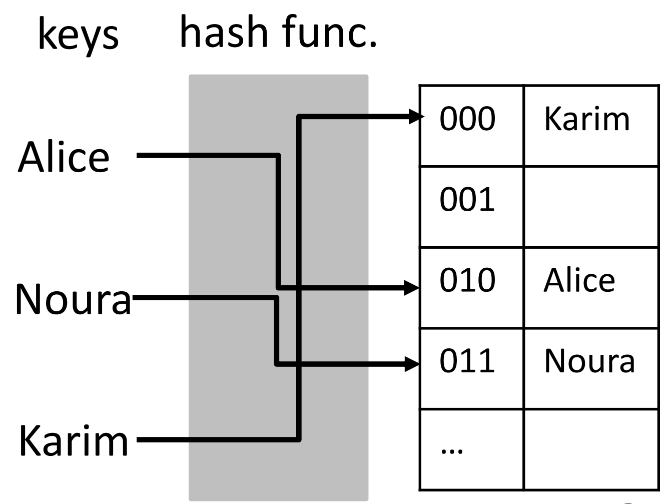
Disk-based Columnar Storage



node properties

CSR-based join indices & edge props

Disk-based Hash Index



Joins in Kùzu

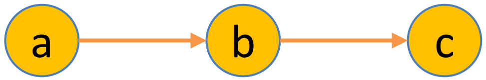
Design Goals for Fast Joins in Kùzu

1. Factorize/compress intermediate results under m-n joins
2. Always perform sequential scans of nodes, edges & properties
   - Behave similar to Hash Joins that are common in RDBMS
3. Avoid full scans of properties when possible

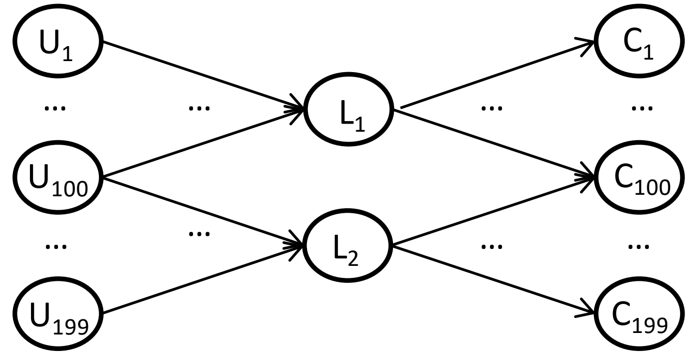
Data Systems Group

DG 1: Factorization [Olteanu et. al. SIGMOD Rec., `16]

```
MATCH
```




```
WHERE b.name='Liz' AND
RETURN a.ID, c.ID
```



Flat Representation

b	**b.name**	**a**	**c**
L_1	Liz	U_1	C_1
	…	…	…
L_1	Liz	U_1	C_{100}
L_2	Liz	U_{100}	C_{100}
…	…	…	…
L_2	Liz	U_{199}	C_{100}

2x(100x100) tuples

F-Representation

b, b.name		**a**		**c**
$\{L_1$, Liz$\}$	X	$\{U_{1,}\ldots, U_{100}\}$	X	$\{C_1, \ldots, C_{100}\}$
$\{L_2$, Liz$\}$	X	$\{U_{100,}\ldots, U_{199}\}$	X	$\{C_{100}, \ldots, C_{199}\}$

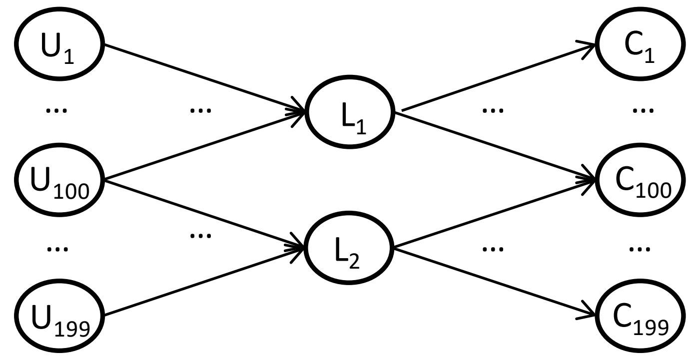
2x(100+100) tuples

➢ Key message: Outputs of a query Q can be factorized by analyzing *the conditional independence of the variables* in Q *statically during compilation*

Kùzu Intermediate Relations: *Factorized Vectors*

- Standard vectorized query processors: single group of vectors
- Kùzu: uses multiple "factorized" groups of vectors

```
MATCH (a)→(b)→(c)
WHERE b.name='Liz' AND
RETURN a.ID, c.ID
```



Vector Group 1

<u>b</u>	<u>b.name</u>
L_1	Liz
L_2	Liz

curIdx = 1

flat

Vector Group 2

<u>a</u>
U_1
U_2
…
U_{100}

curIdx = -1

unflat

Vector Group 3

<u>c</u>
C_1
C_2
…
C_{100}

curIdx = -1

unflat

Amine Mhedhbi
on academic job market!


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

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu

University of Waterloo

VLDB '21

- Each represents either 1 (flat) or a set of values (unflat)

DG 2 & 3: Sequential Scans But Avoid Full Scans

- Standard Hash Joins & Sequential file scans:
  - Sequential scans ✓
- Hash Joins w/ *sideways information passing*
  - Avoid full file scans ✓
- Challenge: How to combine with factorization and obtain large number of factorization structures?
- Solution: 2 operators: SJoin and ASPJoin

Making RDBMSs Efficient on Graph Workloads Through Predefined Joins

Guodong Jin, Renmin University of China

Semih Salihoglu, University of Waterloo, Canada

VLDB `22

GRainDB: A Relational-core Graph-Relational DBMS

Guodong Jin, Renmin University of China

Nafisa Anzum, University of Waterloo

Semih Salihoglu, University of Waterloo

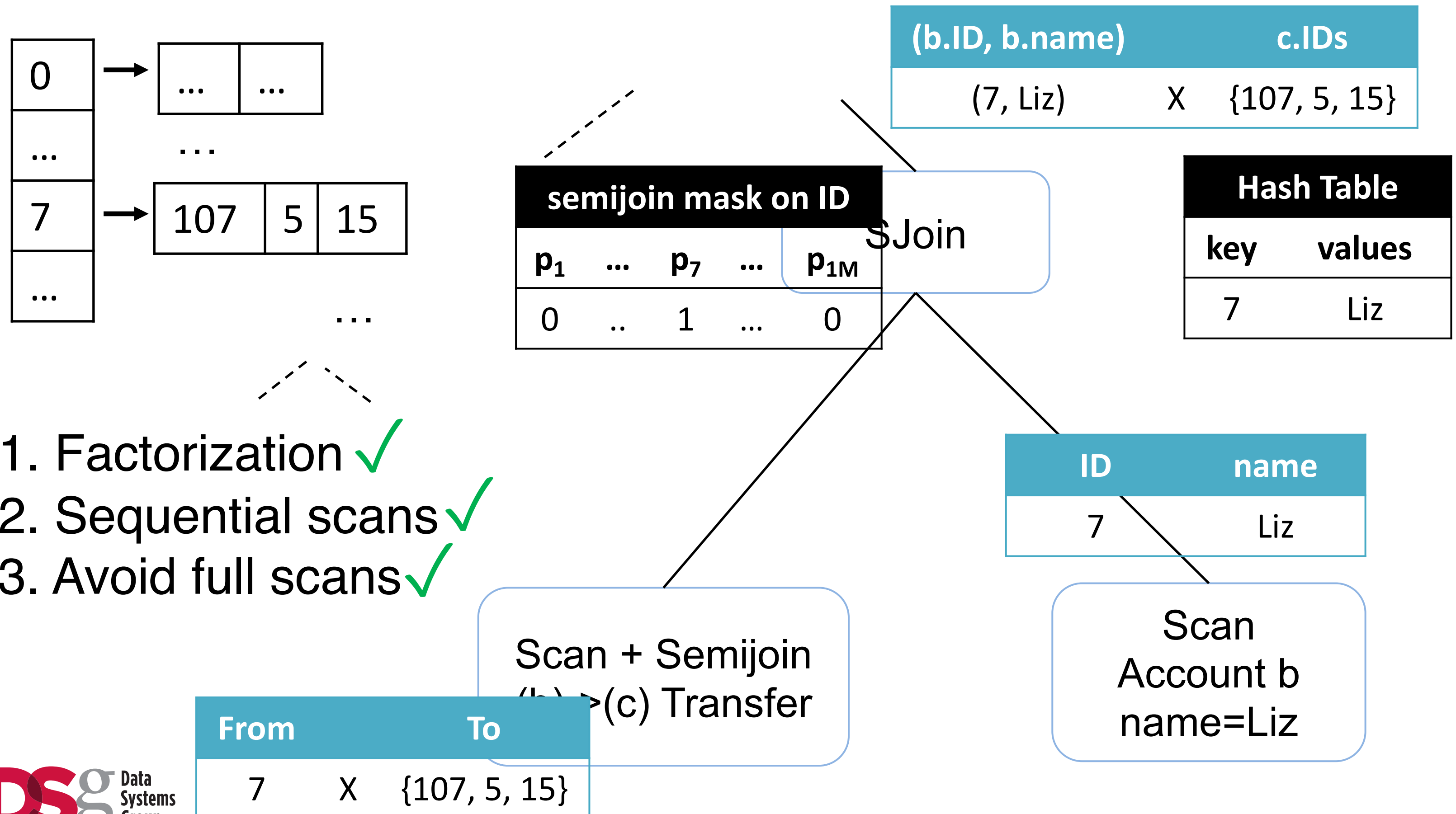
CIDR `22

Based on our work on DuckDB's hash joins (but flat processing)

SJoin Example

- Pass IDs of “joining” nodes sideways from build side of HJs to the probe sides to avoid full table scans

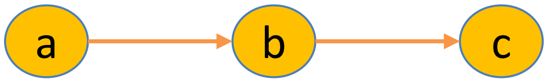
```
MATCH (b:Account)-[:Transfer]->(c:Account)
WHERE b.name='Liz' RETURN c.ID
```



1. Factorization ✓
2. Sequential scans ✓
3. Avoid full scans ✓

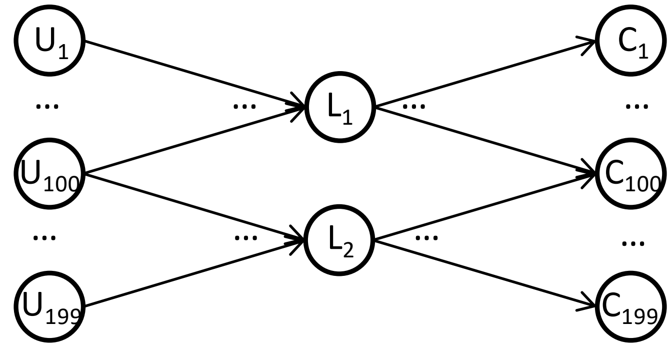
Data Systems Group

Example: Back to 2-hop query

MATCH 

WHERE b.name='Liz'

RETURN a.name, b.ID, c.name

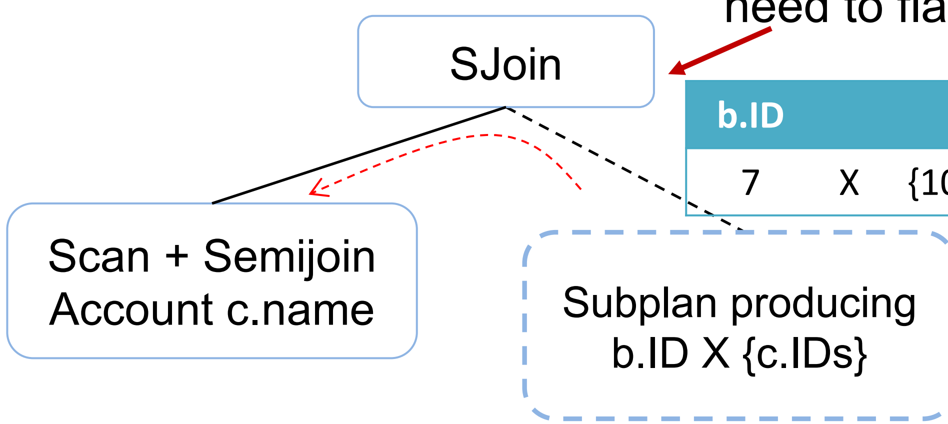


Desired factorization : $\{(a_1, a_1.name),\ldots,(a_{k1}, a_{k1}.name)\}$ X b_i X $\{(c_1,c_1.name),\ldots,(c_{k2}, c_{k2}.name)\}$

unflat flat unflat

Problem: If we want to join a.name and c.name values with a.ID and c.ID, we often need to "flatten" them when performing hash joins.

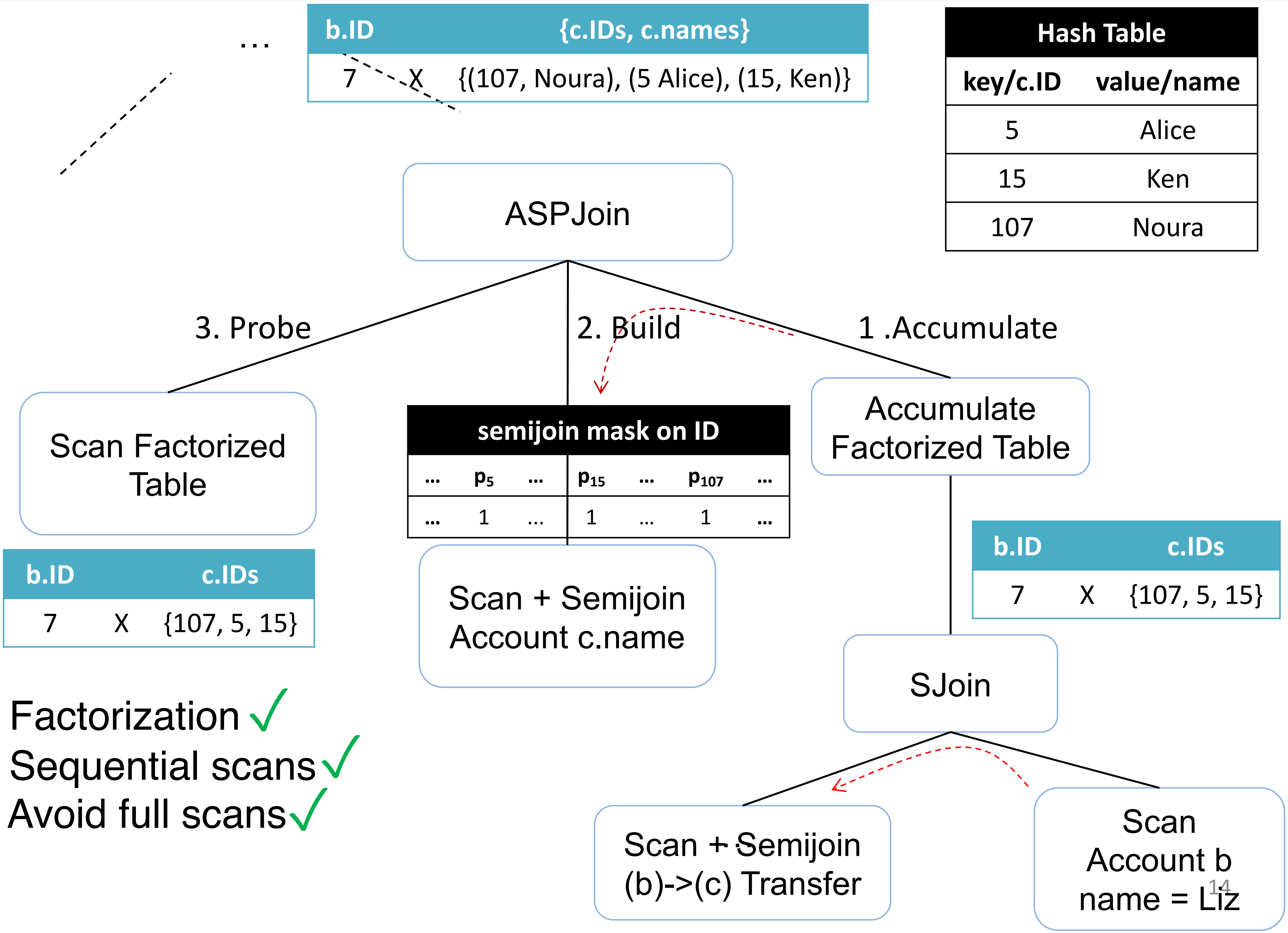
need to flatten c.ID to join c.name.



b.ID		c.IDs
7	X	{107, 5, 15}

Data Systems Group

ASPJoin Example



1. Factorization ✓
2. Sequential scans ✓
3. Avoid full scans ✓

Example Microbenchmark Experiment

```
MATCH (a:Comment)<-[:Likes]-(b:Person)-[:Likes]->(c:Comment)
WHERE b.ID < X
RETURN min(a.ID), min(b.ID), min(c.ID)
```

- LDBC 100: 220M Comments & 0.5M Person nodes, 242M Likes edges
- 8 threads, 64GB RAM

Selectivity	Kùzu	Kùzu-INLJ	Umbra
0.01%	0.33s	**0.01s**	1.90s
0.1%	0.41s	**0.11s**	4.05s
1%	**0.96s**	1.04s	12.30s
10%	**3.89s**	10.39s	230.35s
100%	**31.98s**	92.35	TO

Data Systems Group

More in the Paper

- Extension to worst-case optimal joins for “cyclic” joins
- How we generate query plans
- Overview of other system components

KÙZU* Graph Database Management System

Xiyang Feng** Guodong Jin** Ziyi Chen Chang Liu Semih Salihoğlu
{x74feng,guodong.jin,z473chen,c.liu,semih.salihoglu}@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

Datasets and workloads of popular applications that use graph database management systems (GDBMSs) require a set of storage and query processing features that RDBMSs do not traditionally optimize for. These include optimizations for: (i) many-to-many (m-n) joins; (ii) cyclic joins; (iii) recursive joins; (iv) semi-structured data storage; and (v) support for universal resource identifiers. We present Kùzu, a new GDBMS we are developing at University of Waterloo that aims to integrate state-of-art storage, indexing, and query processing techniques to highly optimize for this feature set.

This paper serves the dual role of describing our vision for Kùzu and the system’s factorized query processor, which is based on two design goals: (i) achieving good factorization structures under m-n joins; and (ii) ensuring sequential scans that avoid entire scans of columns and join indices when possible. As we show these two goals can sometimes conflict and we describe our core binary and worst-case optimal (multiway) join operators that simultaneously achieve both goals. Kùzu is actively being developed to be a fully functional open-source DBMS with the goal of wide user adoption.

1 INTRODUCTION

Modern GDBMSs adopt a graph/network data model and SQL-like high-level query languages that have several graph-specific constructs, such as arrows to describe joins and Kleene star to describe reachability between records. At their cores, GDBMSs are relational in the sense that they map their query language constructs to relational operators, such as join, project, filter, or group by, that process and output sets/relations of tuples. In practical use, GDBMSs power several analytics-oriented applications that are popular in fraud detection systems, recommendation engines, master data and knowledge management, among other domains.

The workloads of these application require several storage and processing features that existing RDBMSs are generally not optimized for. These features include: (i) many-to-many (m-n) joins; (ii) cyclic join queries, such as when finding cyclic graph patterns; (iii) recursive joins, such as those used for reachability computations; (iv) ability to store semi-structured data, i.e., whose columns/property names and types are not defined to the system apriori; and (v) storage and processing of universal resource identifiers (URIs), which are strings identifying entities in knowledge graphs encoded as RDF-style triples. This paper presents *Kùzu*, a new GDBMS that we are developing at University of Waterloo that aims to optimize these features through the integration of existing and novel state-of-art storage, indexing, and query processing techniques.

Kùzu’s design is informed by our insights from discussions with many users of GDBMSs, which we previously published as a user

*Kù-zu (‘bright’ + ‘to know’) is a Sumerian word for“wisdom” [13].
**Equal contributions.

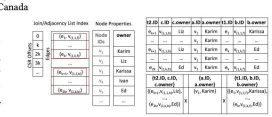


Figure 1: Left: columnar storage of Kùzu. Right: flat and factorized output of the 2-hop query in Example 1.

survey paper [28], and developing our group’s previous system GraphflowDB [12, 15, 18, 19], from which it differs in many important aspects. GraphflowDB was an in-memory non-transactional prototype system we used for our research agenda. Instead, Kùzu aims to be a fully functional, user-facing, and a *highly scalable* GDBMS, which was the most pressing challenge of users in our survey [28]. To that end, Kùzu is disk-based and scales out of memory, implements disk-based primary key and join indices, and integrates more robust and scalable join capabilities than GraphflowDB. In addition, Kùzu is transactional and aims to provide the core DBMS functionalities to be user-facing. In its current usage vision, Kùzu is an open source[1] embeddable library suitable for quickly developing pipelines in the graph data science ecosystem. This is inspired by DuckDB [1, 26] and originally by SQLite [3] [2].

This paper focuses on Kùzu’s processor, which is block-based as in modern analytical DBMSs and further aims to satisfy two goals:

(i) Intermediate relations of m-n joins should be factorized [23], i.e., represented as Cartesian products instead of flat tuples.
(ii) *Scans should always be sequential* and when possible only scan necessary blocks from base columns or join indices.

As we next demonstrate, these two design goals are often in conflict even in very simple queries:

Example 1. *Consider the query below that asks for the owners of the source and destination accounts of each 2-hop money transfer facilitated by Karim’s accounts:*

```
MATCH (c:Acc)-[t2:Trnsfr]->(a:Acc)-[t1:Transfer]->(b:Acc)
WHERE a.owner = 'Karim' RETURN c.owner, b.owner
```

Consider a columnar storage that we assume in this paper, where both adjacency lists and node properties are stored in columnar structures as in Figure 1. Consider a k-regular database, where a node v_i has k outgoing/incoming neighbors $\{v_{(i,1,f/b)}, ..., v_{(i,k,f/b)}\}$, where $v_{(i,j,f/b)}$

[1]The system is planned to be open sourced in the last quarter of 2022.
[2]Kùzu is also inspired by other open-source DBMS projects originally from research groups, such as PostgreSQL and MonetDB, which have also achieved wide adoption.

Team


Xiyang Feng


Guodong Jin


Ziyi Chen


Chang Liu


Anurag Chakraborty


Mushi Wang


Wei Pang

Honorary Team Members

Amine Mhedhbi

Pranjal Gupta

pip install kuzu!

KÙZU

website