



Graph Database Management System

Semih Salihoğlu



UNIVERSITY OF
WATERLOO

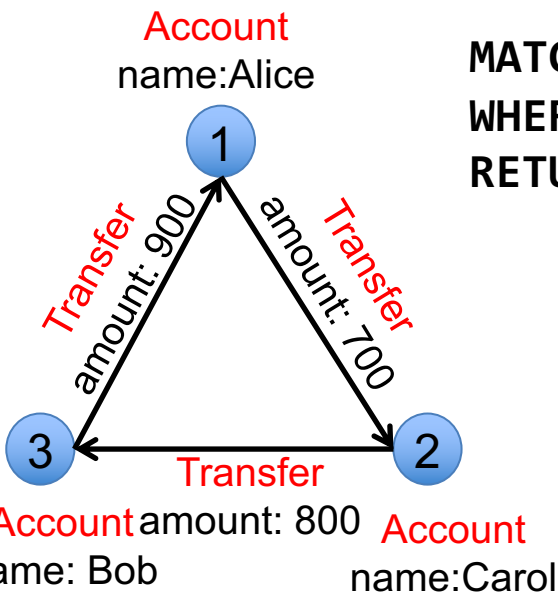


(Property) Graph DBMS Overview

- 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



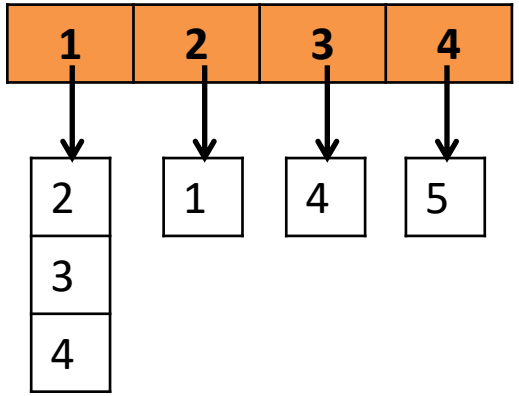
Query Language

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

Example Application Domain: Graph Datascience (GDS) Pipelines

Graph Neural Network

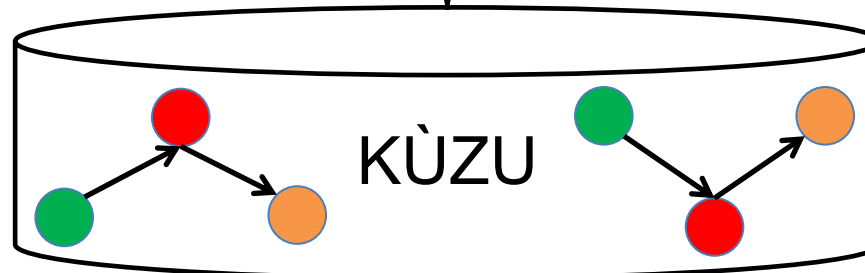
Pytorch Geometric,
DGL, Graph-AI
libraries

Python DS Libraries

Pandas, NumPy,
SciPy, etc.

Graph Analytics & Viz

NetworkX,
Graph-tool, PyVis,
Cytoscape



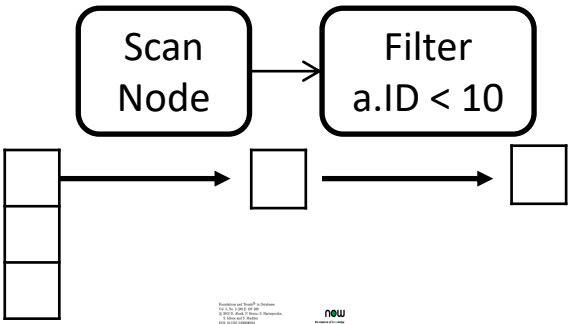
Adjacency List,
Edge Lists

Parquet/Arrow/CSV
...

Relational DBMSs

Kùzu Current Performance Features

Vectorization



NOU
The Design and Implementation of Modern Column-Oriented Database Systems

David Koop, Peter Bosc, Rainer Rastbach
Tobias Ullrich, Christian Bode, Andrej Klenk
Bodo Beckmann, Christoph Koch, Michael Hübner
Ralf Wüschel, Daniel Flato, Ulf Helmert
www.nou-db.com

Factorization

<u>b</u> , name		<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 }

Columnar Storage and List-based Processing for Graph Database Management Systems
Frank Opat, Andrej Klenk, Frank Hees
www.kit.edu

Factorized Databases
Dan Chen, Martin Schmitz
www.kit.edu

Morsel-driven Parallelism

➤ Scans are “morselized” across threads

Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis, Peter Bosc, Albert Kemper, Thomas Neumann
*Technische Universität München *DBP
*leis@informatik.uni-tuebingen.de *bosco@informatik.uni-tuebingen.de

ABSTRACT
With modern computer architectures, the performance gains from parallelizing database queries are becoming increasingly difficult to achieve. In particular, the performance of parallel queries is often limited by the overhead of data distribution and the overhead of communication between processors. This paper introduces a novel framework for query evaluation that is designed to exploit the benefits of modern computer architectures. The framework is based on the idea of morsel-driven parallelism, where the data is divided into small morsels that can be processed independently by different processors. This approach allows for a more efficient use of the available hardware resources and leads to significant performance improvements. The framework is implemented in the Kùzu database system and has been evaluated on a wide range of queries and datasets. The results show that the framework is able to achieve up to 10x speedup over a baseline approach.

Categories and Subject Descriptors
D.1.2 Database Management Systems

Keywords
Database management, Parallel processing, Query evaluation, NUMA-aware, Many-core

Novel Join Algorithms

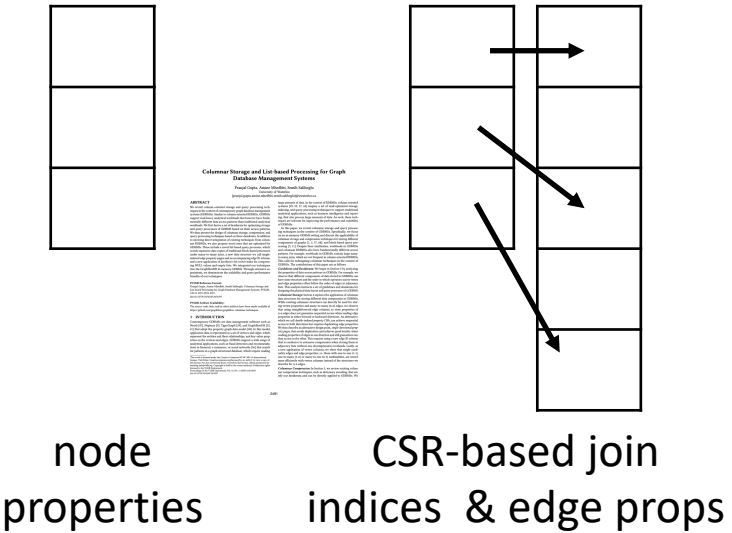
Optimizing Subgraph Queries by Combining Binary and Range-Case Graph Joins
Krzysztof Grzesiak, Sören Krieger, Frank Hees
www.kit.edu

ABSTRACT
Subgraph queries are a common type of query in graph databases. They consist of finding all subgraphs of a given graph that match a given query graph. This is a computationally expensive task, and many different algorithms have been proposed to speed it up. In this paper, we propose a novel join algorithm for subgraph queries that combines binary and range-case graph joins. This algorithm is able to handle a wider range of queries than previous algorithms and is significantly faster. We evaluate our algorithm on a wide range of queries and datasets and show that it achieves up to 10x speedup over a baseline approach.

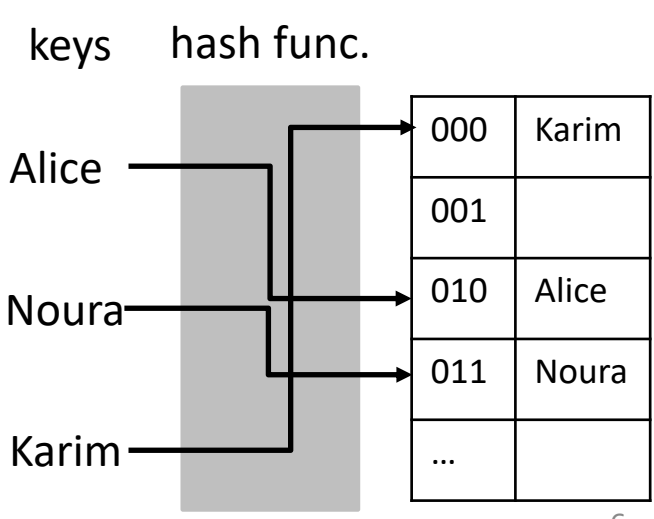
KùZU Graph Database Management System
Frank Opat, Andrej Klenk, Frank Hees
www.kit.edu

ABSTRACT
KùZU is a graph database management system that is designed to handle large-scale graph data. It is based on a novel join algorithm that combines binary and range-case graph joins. This algorithm is able to handle a wider range of queries than previous algorithms and is significantly faster. We evaluate KùZU on a wide range of queries and datasets and show that it achieves up to 10x speedup over a baseline approach.

Disk-based Columnar Storage



Disk-based Hash Index

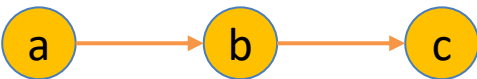


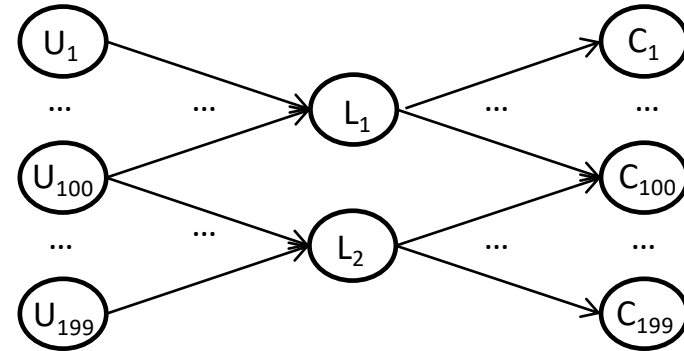
Joins in Kùzu

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

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

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



Flat Representation

<u>b</u>	<u>b.name</u>	<u>a</u>	<u>c</u>
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

<u>b, b.name</u>		<u>a</u>		<u>c</u>
{L ₁ , Liz}	X	{U ₁ , ..., U ₁₀₀ }	X	{C ₁ , ..., C ₁₀₀ }
{L ₂ , Liz}	X	{U ₁₀₀ , ..., U ₁₉₉ }	X	{C ₁₀₀ , ..., C ₁₉₉ }

2x(100+100) tuples

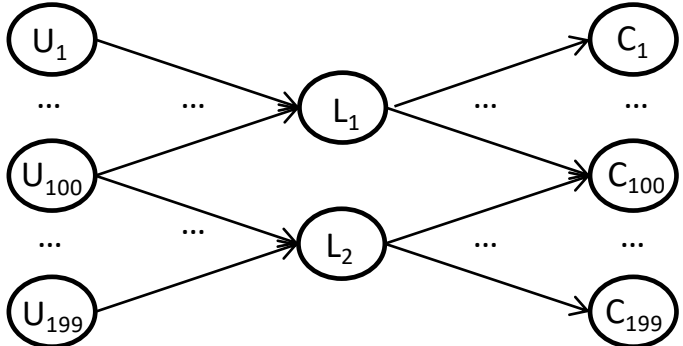
- Key message: Outputs of a query Q can be factorized by analyzing *the conditional independence of the variables in Q statically during compilation*

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

curlidx = 1
flat

Vector Group 2

<u>a</u>
U ₁
U ₂
...
U ₁₀₀

curlidx = -1
unflat

Vector Group 3

<u>c</u>
C ₁
C ₂
...
C ₁₀₀

curlidx = -1
unflat



Amine Mhedhbi
on academic job market!

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

Tareq Gajjar, Amine Mhedhbi, Sushil Kulkarni
University of Waterloo
tareq.gajjar@uwaterloo.ca, amine.mhedhbi@uwaterloo.ca

ABSTRACT
We analyze columnar-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDMS). Unlike for column-oriented relational databases, graph-oriented relational databases have not been well studied. We first discuss the design space for graph-oriented relational databases and then propose a novel storage and query processing architecture for GDMS based on list-based processing. We then present the design of columnar storage, compression, and query processing techniques for the proposed architecture. We evaluate the proposed architecture by comparing it with existing architectures. Our results show that the proposed architecture outperforms existing architectures in terms of storage efficiency and query processing performance. The proposed architecture is the first to support list-based processing for graph-oriented relational databases. We hope this work will inspire further research in this area.

1 INTRODUCTION
Columnar storage is a data representation technique that stores data in columns instead of rows. This technique is widely used in relational databases for its ability to compress data and improve query performance. In graph-oriented relational databases, columnar storage is used to store graph data. However, existing graph-oriented relational databases do not support list-based processing for graph-oriented relational databases. This work proposes a novel storage and query processing architecture for graph-oriented relational databases that supports list-based processing. The proposed architecture is the first to support list-based processing for graph-oriented relational databases. We hope this work will inspire further research in this area.

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

Guangdong Jin
jin@gdsc.cs.ubc.ca
Simon Fraser University, Canada

Smith Sabharwal
smithsab@uwaterloo.ca
University of Waterloo, Canada

ABSTRACT

Join is one of the most commonly used data structures to build data in computer database applications and tables, which are often represented as graph. In this paper, we propose a novel approach to improve the performance of graph and relational data joins in RDBMSs, leading to a significant performance gain in graph workloads. We propose a novel approach to improve the performance of graph and relational data joins in RDBMSs, leading to a significant performance gain in graph workloads. We propose a novel approach to improve the performance of graph and relational data joins in RDBMSs, leading to a significant performance gain in graph workloads.

GRainDB: A Relational-core Graph-Relational DBMS

Guangdong Jin
jin@gdsc.cs.ubc.ca
Simon Fraser University, Canada

Nafiseh Azam
nafiseh@uwaterloo.ca
University of Waterloo, Canada

Smith Sabharwal
smithsab@uwaterloo.ca
University of Waterloo, Canada

ABSTRACT

Join is one of the most commonly used data structures to build data in computer database applications and tables, which are often represented as graph. In this paper, we propose a novel approach to improve the performance of graph and relational data joins in RDBMSs, leading to a significant performance gain in graph workloads. We propose a novel approach to improve the performance of graph and relational data joins in RDBMSs, leading to a significant performance gain in graph workloads.



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

VLDB '22

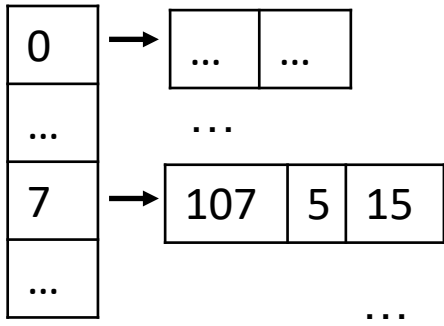
CIDR '22

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
```



semijoin mask on ID				
p ₁	...	p ₇	...	p _{1M}
0	..	1	...	0

(b.ID, b.name)		c.IDs
(7, Liz)	X	{107, 5, 15}

Hash Table	
key	values
7	Liz

ID	name
7	Liz

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

Scan + Semijoin
(b) -> (c) Transfer

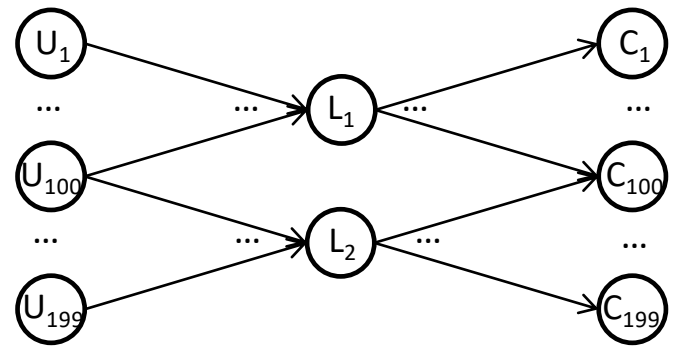
Scan
Account b
name=Liz

From	To
7	X {107, 5, 15}

Example: Back to 2-hop query

```

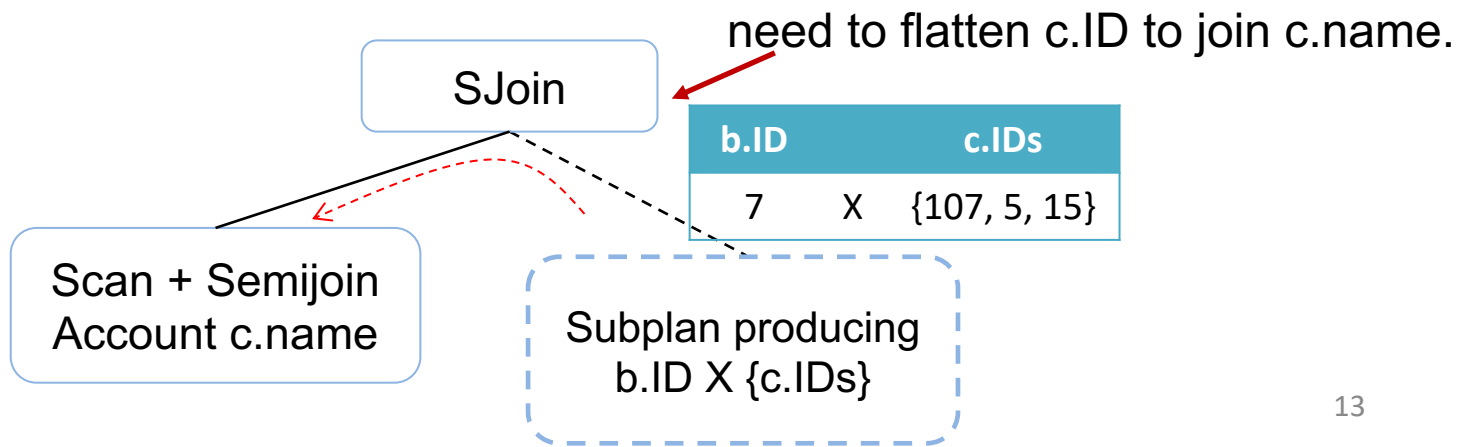
MATCH (a) -> (b) -> (c)
WHERE b.name='Liz'
RETURN a.name, b.ID, c.name
    
```



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

↑ 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.



ASPJoin Example

b.ID		{c.IDs, c.names}
7	X	{{(107, Noura), (5, Alice), (15, Ken)}

Hash Table	
key/c.ID	value/name
5	Alice
15	Ken
107	Noura

ASPJoin

3. Probe

2. Build

1. Accumulate

Scan Factorized Table

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

semijoin mask on ID						
...	p ₅	...	p ₁₅	...	p ₁₀₇	...
...	1	...	1	...	1	...

Scan + Semijoin Account c.name

Accumulate Factorized Table

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

SJoin

Scan + Semijoin (b)->(c) Transfer

Scan Account b name = Liz

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

More in the Paper

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

KÜZÜ* Graph Database Management System

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

ABSTRACT
Databases and workloads of popular applications that use graph database management systems (GDBMS) require a set of storage and query processing features that RDBMS 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 Küzü, 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 Küzü 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 (multway) join operators that simultaneously achieve both goals. Küzü is actively being developed to be a fully functional open-source DBMS with the goal of wide user adoption.

1 INTRODUCTION
Modern GDBMS 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, GDBMS 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, GDBMS 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 GDBMS 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 a priori; 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 Küzü, 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. Küzü's design is informed by our insights from discussions with many users of GDBMS, which we previously published as a user

id	id ₁	id ₂	id ₃	id ₄	id ₅	id ₆	id ₇	id ₈	id ₉	id ₁₀
1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11	12	13
4	5	6	7	8	9	10	11	12	13	14
5	6	7	8	9	10	11	12	13	14	15
6	7	8	9	10	11	12	13	14	15	16
7	8	9	10	11	12	13	14	15	16	17
8	9	10	11	12	13	14	15	16	17	18
9	10	11	12	13	14	15	16	17	18	19
10	11	12	13	14	15	16	17	18	19	20

id	id ₁	id ₂	id ₃	id ₄	id ₅	id ₆	id ₇	id ₈	id ₉	id ₁₀
1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11	12	13
4	5	6	7	8	9	10	11	12	13	14
5	6	7	8	9	10	11	12	13	14	15
6	7	8	9	10	11	12	13	14	15	16
7	8	9	10	11	12	13	14	15	16	17
8	9	10	11	12	13	14	15	16	17	18
9	10	11	12	13	14	15	16	17	18	19
10	11	12	13	14	15	16	17	18	19	20

Figure 1: Left: columnar storage of Küzü. 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, Küzü 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, Küzü 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, Küzü is transactional and aims to provide the core DBMS functionalities to be user-facing. In its current usage vision, Küzü is an open source¹ 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 Küzü's processor, which is block-based as in modern analytical DBMS 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)-[1:Transfr]->>(a:ACC)-[1: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 $\{v_{i,1}, \dots, v_{i,k}, f, b\}$, where $v_{i,1}, f, b$

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

Team



Xiyang Feng



Guodong Jin



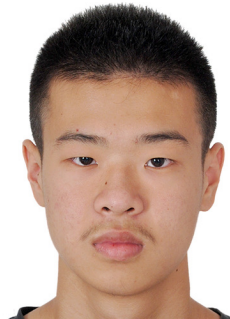
Ziyi Chen



Chang Liu



Anurag
Chakraborty



Mushi Wang



Wei Pang

Honorary Team Members

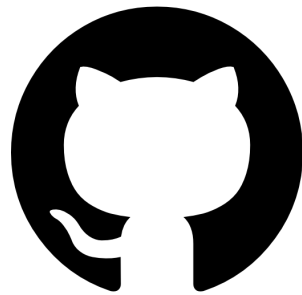


Amine Mhedhbi



Pranjal Gupta

pip install kuzu!



[website](#)