Graph Database Management System

Semih Salihoğlu
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

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

**System**
- Graph-specific storage structures, indices, operators

![Diagram showing labeled graph and graph-specific SQL example.](attachment:/image.png)

- 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

*Kuzu 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

   ```python
   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

KÛZU

- Adjacency List, Edge Lists
- Parquet/Arrow/CSV ...
- Relational DBMSs
Kùzu Current Performance Features

**Vectorization**
- Scan Node
- Filter a.ID < 10

**Factorization**
- \( b, \text{name} \)
- \( a \)
- \( c \)
- \( \{L_1, \text{Liz}\} \times \{U_1, \ldots, U_{100}\} \times \{C_1, \ldots, C_{100}\} \)
- \( \{L_2, \text{Liz}\} \times \{U_{100}, \ldots, U_{199}\} \times \{C_{100}, \ldots, C_{199}\} \)

**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**
- Keys
- Hash func.
- Alice
- Noura
- Karim
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
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 \( a \rightarrow b \rightarrow c \)
WHERE \( b \).name = ‘Liz’ AND
RETURN \( a \).ID, \( c \).ID

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

VLDB `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 ✓
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\},...,\{a_{k1}, a_{k1}.name\}\} \times \{\{c_1, c_1.name\},...,\{c_{k2}, c_{k2}.name\}\}

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.
ASPJoin Example

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

Scan + Semijoin Account c.name

Semijoin mask on ID

Scan Factorized Table

Accumulate Factorized Table

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

Scan Account b name = Liz

Hash Table

<table>
<thead>
<tr>
<th>key/c.ID</th>
<th>value/name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Alice</td>
</tr>
<tr>
<td>15</td>
<td>Ken</td>
</tr>
<tr>
<td>107</td>
<td>Noura</td>
</tr>
</tbody>
</table>

b.ID c.IDs

7 X {107, 5, 15}

b.id {c.IDs, c.names}

7 X {(107, Noura), (5 Alice), (15, Ken)}
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

<table>
<thead>
<tr>
<th>Selectivity</th>
<th>Kuzu</th>
<th>Kuzu-INLJ</th>
<th>Umbra</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01%</td>
<td>0.33s</td>
<td>0.01s</td>
<td>1.90s</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.41s</td>
<td>0.11s</td>
<td>4.05s</td>
</tr>
<tr>
<td>1%</td>
<td>0.96s</td>
<td>1.04s</td>
<td>12.30s</td>
</tr>
<tr>
<td>10%</td>
<td>3.89s</td>
<td>10.39s</td>
<td>230.35s</td>
</tr>
<tr>
<td>100%</td>
<td>31.98s</td>
<td>92.35</td>
<td>TO</td>
</tr>
</tbody>
</table>
More in the Paper

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

[GitHub logo]

[Website link]