Optimizing Recursive Joins in Graph Database Management Systems

Anurag Chakraborty,
David R. Cheriton School of Computer Science

13/12/23
Outline

- Background on Graph DBMS & Recursive Joins
- Why are Recursive Joins challenging?
- Query Processing for Recursive Joins
- Future Work
Outline

- Background on Graph DBMS & Recursive Joins
- Why are Recursive Joins challenging?
- Query Processing for Recursive Joins
- Future Work
Background on Graph Databases

Data Model

➢ Labeled Property Graph (LPG)

Query Language

Neo4j
Cypher Query Language

MATCH
(a:Person)-[r1:Stays_At]->(b:Place)<-[r2:Stays_At]-(c:Person)
RETURN b.Population;

- Express subgraph pattern for Pattern Matching
- Express recursive queries for Graph Path Traversal
Background on Recursive Joins

Core competency of GDBMS compared to RDBMS

(1) Easier to express in the query language of GDBMS:

Query: Return all people ‘Alice’ knows directly / indirectly and the path length between them

Cypher:

MATCH p = (p1:Person)-[:knows* SHORTEST 1..30]->(p2:Person)
WHERE p1.name = ‘Alice’ RETURN p2, length(p)

Harder to express in recursive SQL.
Background on Recursive Joins

Core competency of GDBMS compared to RDBMS

(2) Also often faster to execute in GDBMS

GDBMS have **specialized recursive join operators**

Query:

```
MATCH p = (a:Person)-[r:knows* 1..30]->(b:Person)
WHERE a.name = "Alice"
RETURN a.ID, b.ID, length(p)
```
Outline

- Background on Graph DBMS & Recursive Joins
- Why are Recursive Joins challenging?
- Query Processing for Recursive Joins
- Future Work
SoTA Approach in Analytical DBMS: Morsel-driven Parallelism

- allot fixed size “morsels” to threads (1024 - 2048 - 100,000 tuples)

- threads execute on their morsels for 1 pipeline until the pipeline breaker

*Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age, Leis et al.*
SoTA Approach in Analytical DBMS: Morsel-driven Parallelism

- allot fixed size “morsels” to threads (1024 - 2048 - 100,000 tuples)

- threads execute on their morsels for 1 pipeline until the pipeline breaker

- morsel dispatcher allots other morsels to threads, after completion of previous pipeline
Problem with Morsel-Driven Parallelism for Recursive Joins

1. Recursive Join operators find variable length / shortest path from a single source node. Usually involve some form of BFS style traversal.
1. Recursive Join operators find variable length / shortest path from a single source node.

- Most real world graphs display small world network property
- 5 or 6 steps may “traverse” the entire database
- This makes recursive joins, even from 1 source very expensive

Example:
MATCH p = (p1:Person)-[:knows* SHORTEST 1..30]->(p2:Person) 
WHERE p1.name = ‘Alice’ RETURN p2, length(p)
Problem with Morsel-Driven Parallelism for Recursive Joins

2. Dispatcher may allot a morsel with disproportionate no. of sources to a single thread
Problem with Morsel-Driven Parallelism for Recursive Joins

2. Dispatcher may allot a morsel with disproportionate no. of sources to a single thread

Example:
MATCH
p = (p1:Person)-[:knows* SHORTEST 1..30]->(p2:Person)
WHERE p1.ID < 50 RETURN length(p)
Problem with Morsel-Driven Parallelism for Recursive Joins

2. Dispatcher may allot a morsel with disproportionate no. of sources to a single thread

Example:
MATCH
p = (p1:Person)-[:knows* SHORTEST 1..30]->(p2:Person)
WHERE p1.ID < 50 RETURN length(p)

Q.) How can we parallelize pipelines with recursive join operators robustly?
(1) Make Recursive Join operator into a source operator to start the query pipeline. Threads should start BFS with a single source from this operator.
Solution (Query Plan)

Cypher query:

MATCH p = (a:Person)-[r:knows* \SHORTEST 1..30]->(b:Person)
WHERE a.ID < 1000 AND b.ID < 1000
RETURN a.ID, b.ID, length(p)

Recursive Join (RecJoin) operator must be the start of a pipeline
(2) Morselize as before among threads with effective morsel size as 1 BFS source node (Inter-RecJoin parallelism)

(3) When threads are idle, morselize a single RecJoin’s BFS Level (frontier) into granular morsels among these threads (Intra-RecJoin parallelism)
(2) Morselize as before among threads with effective morsel size as 1 BFS source node (Inter-RecJoin parallelism)

(3) When threads are idle, morselize a single RecJoin’s BFS Level (frontier) into granular morsels among these threads (Intra-RecJoin parallelism)

- Define two types of morsels: (i) BFSMorsel (single source recursive join)
  (ii) BFSLevelMorsel (subset of BFSMorsel’s join)

- Introduce a Recursive Join scheduler that distributes this work
Outline

- Background on Graph DBMS & Recursive Joins
- Why are Recursive Joins challenging?
- Query Processing for Recursive Joins
- Future Work
Recursive Join Physical Operator

- **BFSScheduler**: Operators own scheduler that distributes work to the threads.

BFSScheduler controls total no. of concurrent *BFSMorsel* to at most $k$. Max limit is set to $n$ (total threads).

$n$ Threads, $k$ BFSMorsel

$nTkS$ scheduler
Recursive Join Physical Operator

- **BFSScheduler**: Operators own scheduler that distributes work to the threads.

(1) BFS Scheduler launches a new BFS recursive join from a source if total active BFS Morsel < k
**Recursive Join Physical Operator**

- **BFSScheduler**: Operators own scheduler that distributes work to the threads.

1. BFS Scheduler launches a new BFS recursive join from a source if total (active BFS < k)

2. If not, scheduler iterates over all active BFSMorsel to find BFS with most work and allot subset of the join as a BFSLevelMorsel.
Returning Path (Shortest / All Shortest)

- maintain a global *visited array*

- update node states as they are encountered

- use lightweight *lock-free* synchronization

- additionally maintain *multiplicity* (all shortest path)

- use *atomic CAS* operations to update states and track \{source node, edge\} of nodes

- use *atomic fetch and add (faa)* operations to update multiplicity (for path length)
Returning Path (Variable Length)

- maintain a global *visited array*

- update node states as they are encountered

- use lightweight *lock-free* synchronization

- additionally maintain *multiplicity* at *different levels*

- use *atomic CAS* operations to update states and track 
   \{source node, edge\} of nodes at *different levels*

- use *atomic fetch and add (faa)* operations to update 
  multiplicity (for path length)
Results

**Microbenchmark: (LDBC-100)**

MATCH (a:Person)-[r:knows* \textsc{shortest} 1..30]->(b:Person) WHERE a.ID = 94 return b.ID, length(r);
Total tuples: \textbf{407,396}

<table>
<thead>
<tr>
<th>Kùzu (Baseline MDP - 32 threads)</th>
<th>Kùzu (nTkS - 32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>728.6 ms</td>
<td>61 ms \textit{(12x faster)}</td>
</tr>
</tbody>
</table>

**Microbenchmark: (LiveJournal)**

MATCH (a:lj_node)-[r:lj\_rel* \textsc{shortest} 1..30]->(b:lj_node) WHERE a.id < 1000 return b.ID, length(r);
Total tuples: \textbf{4,237,533,225}

<table>
<thead>
<tr>
<th>Kùzu (Baseline MDP - 32 threads)</th>
<th>Kùzu (nTkS - 32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>158 s</td>
<td>105 s \textit{(1.5x faster)}</td>
</tr>
</tbody>
</table>
## Results

### Microbenchmark: (graph500-23)

MATCH (a:nodes)-[r:rels* ALL SHORTEST 1..30]->(b:nodes) WHERE a.id = 307 RETURN r;
Total tuples: 105,576,064

<table>
<thead>
<tr>
<th>Kuzu (Baseline MDP - 32 threads)</th>
<th>Kuzu (nTkS - 32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>511s</td>
<td>35s (14.6x faster)</td>
</tr>
</tbody>
</table>

### Microbenchmark: (graph500-24)

MATCH (a:nodes)-[r:rels* 1..4]->(b:nodes) WHERE a.id = 0 RETURN r;
Total tuples: 126,749,073

<table>
<thead>
<tr>
<th>Kuzu (Baseline MDP - 32 threads)</th>
<th>Kuzu (nTkS - 32 threads)</th>
</tr>
</thead>
<tbody>
<tr>
<td>634s</td>
<td>37.6s (16.9x faster)</td>
</tr>
</tbody>
</table>
Outline

- Background on Graph DBMS & Recursive Joins
- Why are Recursive Joins challenging?
- Query Processing for Recursive Joins
- Future Work
Future Work

➢ Integrating other techniques [Multi source BFS (MS-BFS), Bidirectional BFS]

➢ Storing paths in a compressed manner for vectorized execution ?

➢ Weighted Shortest Path (Dijkstra, Bellman Ford, ...) ?
Thank You