Robust Recursive Query Parallelism in Graph Database Management Systems

Anurag Chakraborty
University of Waterloo
Waterloo, Ontario, Canada
a8chakra@uwaterloo.ca

Semih Salihoğlu
University of Waterloo
Waterloo, Ontario, Canada
semih.salihoglu@uwaterloo.ca

ABSTRACT

Efficient multi-core parallel processing of recursive join queries is critical for achieving good performance in graph database management systems (GDBMSs). Prior work adopts two broad approaches. First is the state of the art morsel-driven parallelism, whose vanilla application in GDBMSs parallelizes computations at the source node level. Second is to parallelize each iteration of the computation at the *frontier* level. We show that these approaches can be seen as part of a design space of morsel dispatching policies based on picking different granularities of morsels. We then empirically study the question of which policies parallelize better in practice under a variety of datasets and query workloads that contain one to many source nodes. We show that these two policies can be combined in a hybrid policy that issues morsels both at the source node and frontier levels. We then show that the multi-source breadth-first search optimization from prior work can also be modeled as a morsel dispatching policy that packs multiple source nodes into *multi-source morsels*. We implement these policies inside a single system, the Kuzu GDBMS, and evaluate them both within Kuzu and across other systems. We show that the hybrid policy captures the behavior of both source morsel-only and frontier morsel-only policies in cases when these approaches parallelize well, and outperform them on queries when they are limited, and propose it as a robust approach to parallelizing recursive queries. We further show that assigning multi-sources is beneficial, as it reduces the amount of scans, but only when there is enough sources in the query.

PVLDB Reference Format:
Anurag Chakraborty and Semih Salihoğlu. Robust Recursive Query Parallelism in Graph Database Management Systems. PVLDB, 18(11): 4465 - 4477, 2025.
doi:10.14778/3749646.3749706
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/anuchak/kuzu.

1 INTRODUCTION

Modern graph database management systems (GDBMS) follow the property graph data model, which supports modeling application records in the form of nodes and edges. An important feature of GDBMSs is that their query languages have the notion of paths as a first-class citizen and special clauses to ask several common recursive queries. For example, the Cypher query language [25] has the arrow-based syntax to specify paths or the Kleene star ("*") syntax followed by the "* SHORTEST" keyword to compute shortest paths between nodes. We refer to queries that require such recursive evaluation as *recursive queries* and the specialized query language clauses in GDBMSs as *recursive clauses*.

We study the problem of how to parallelize recursive queries in GDBMSs. To explain the core problem that motivates this paper, consider the following Cypher query that finds the shortest paths that consist of Knows edges from each Person node with name Alice to other Person nodes:

```
1 MATCH p = (a:Person)-[r:Knows* SHORTEST]->(b:Person)
2 WHERE a.name = Alice RETURN p
```

An example standard plan, drawn left to right, corresponding to this query is shown in Figure 1. The plan contains a specialized shortest path operator, which scans source nodes and runs the Bellman-Ford shortest path algorithm from the sources. Often, the core computation inside the recursive algorithm is a breadth-first search like computation which can be expressed as an *iterative frontier extensions* (IFE) subroutine [4]. Briefly, in the IFE subroutine, neighbors of a *frontier* (i.e., a set) of "active" nodes' are explored to form a new frontier of active nodes, until a convergence criterion is met, such as when the next frontier after an iteration is empty.

A common approach to parallelizing queries in DBMSs is *morsel-driven parallelism* [20]. It breaks a query plan into one or more subplans ("tasks"/"pipelines"), each of which starts with a table scan operator, which scans tuples from a base or intermediate table (henceforth "leaf table"). The system executes tasks in some order and parallelizes each task T by assigning small fragments of inputs, called *morsels*, from the leaf table to worker threads, which work in parallel on T until the leaf table is consumed. This approach is adopted across many RDBMSs and GDBMSs, such as Hyper [17], Umbra [26], DuckDB [30], Neo4j [15] and Kuzu [11]. In vanilla morsel-driven parallelism, if there are not enough morsels available for all threads, e.g., when there is a single Person node with name Alice, the system can end up assigning morsels to a few threads while keeping other threads idle. However, a recursive query from even a single source node can be expensive and amenable to parallelization, as real-world graph databases tend to be heavily connected. This paper studies the question of: *How should a GDBMS that adopts morsel-driven parallelism parallelize recursive computations?*

IFE is a common subroutine used not only to perform recursive path computations but also other graph algorithms that are executed in parallel or distributed graph analytics systems, such as Ligra [31] and Pregel [23]. These systems adopt an alternative approach to parallelize the IFE subroutine. Specifically, these systems

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749706

parallelize each frontier of the IFE subroutine by assigning subsets of active nodes to different threads in each iteration. Therefore, these systems parallelize the work of each iteration of a recursive computation from a source node.

In this paper, we first describe the design space of parallelization approaches that a system can adopt based on the granularity of morsels the system can pick. We call these approaches *morsel dispatching policies* and show that this space captures and generalizes the commonly adopted approaches from above. Vanilla morsel-driven approach of GDBMSs issues *source morsels* to threads. We refer to this policy as 1T1S, for **1**-**T**hread-to-**1**-**S**ource scheduling. In contrast, parallel graph analytics systems issue morsels from the frontiers of a single source node. We refer to these as *frontier morsels* and to this policy as nT1S, for **n**-**T**hreads-to-**1**-**S**ource scheduling. We then identify a hybrid policy that issues both source morsels and frontier morsels to threads. We refer to this hybrid approach as nTkS, for **n**-**T**hreads-to-**k**-**S**ource nodes policy.

Next, we empirically analyze the pros and cons of different policies under a variety of datasets and query workloads that contain from one to hundreds of sources. We show: (i) the 1T1S policy parallelizes well when there are many sources but degrades on queries with few sources, (ii) the nT1S policy achieves limited parallelism with few sources (this limited parallelism persists with many sources). The hybrid nTkS approach captures the desired parallelism behavior of 1T1S and nT1S on queries where they parallelize well, and outperforms them on queries where they demonstrate limited parallelism. As such, we recommend it as a robust morsel dispatching policy for GDBMSs adopting morsel-driven parallelism.

Finally, we revisit the multi-source breadth-first search optimization from prior work [36], which performs concurrent breadth first searches from a batch of source nodes (implemented in DuckPGQ [35]). We show that this optimization can also be modeled as a morsel dispatching policy, in which multiple source nodes are packed into *multi-source morsels*. We describe a hybrid policy called nTkMS, for **n**-**T**hreads-to-**k**-**M**ulti-**S**ource nodes policy. Instead of dispatching source morsels that contain single source nodes, nTkMS dispatches work both as *multi-source morsels*, containing up to 64 source nodes, and frontier morsels. We show empirically that the nTkMS policy outperforms the nTkS policy, as it can reduce the amount of scans performed from the database, however only when there are enough source nodes in the query to saturate the 64-size groups.

We have implemented all of these policies in Kuzu [11], which is a columnar GDBMS that adopts morsel-driven parallelism. [1] We compare our own implementations in Kuzu with Neo4j and Ligra systems, as well as our implementation of the nTkS policy in the DuckPGQ system. This allows us to demonstrate the behavior of these policies both in a controlled manner in a single system as well as on different system implementations that are at different performance levels. Aside from the morsel dispatching suggestions we make in the paper, the details of our implementation in Kuzu can be of independent interest to readers and serve as a blueprint for how these policies can be implemented in other GDBMSs.

[1]Kuzu started as a research prototype in our research group and is now actively being developed in a spinoff company co-founded by the second author of this paper.

Figure 1: Example query plan with a recursive operator.

2 BACKGROUND

We first cover morsel-driven parallelism in more detail with an example. Then, we cover the *iterative frontier extensions (IFE)* [3, 31] algorithmic subroutine. Lastly, we describe the basic query plan structure that we assume that a GDBMS generates to evaluate recursive clauses in this paper. We note that throughout the paper, the terms "source" and "destination" do *not* indicate any direction (forward or backward) in the recursive computations. "Source" indicates the nodes from which a recursive computation finds paths to a set of "destination" nodes in some direction.

2.1 Morsel-Driven Parallelism

Consider the following SQL query that consists of a join between an Employee and Department records.

```
SELECT e.name, b.name WHERE e.age > 55
FROM Employee e, Department d WHERE e.dID = d.ID;
```

In morsel-driven parallelism [20], a DBMS breaks the query plans into subplans (a.k.a. tasks/pipelines) that are executed in some order. Consider a simple hash join-based plan for this query that builds a hash table on the Department table, which is probed by each Employee tuple. A standard approach is to break this plan into two tasks: (i) Task_1 is the subplan that builds the hash table (Figure 2); and (ii) Task_2 is the subplan that probes the hash table from Task_1. Each task is a linear chain of operators that starts with a leaf operator that scans a *leaf* table, which are distributed to multiple threads for parallel execution of the task.

Each thread W_i creates a copy of the task and scans morsels of tuples, e.g., 100K from the leaf table. These tuples are processed by the rest of the operators in the task until W_i needs to grab another morsel of tuples. This parallel computation continues until all of the leaf table's tuples are exhausted. The logic of assignment of morsels to threads is implemented by a piece of code termed *morsel dispatcher* [20].

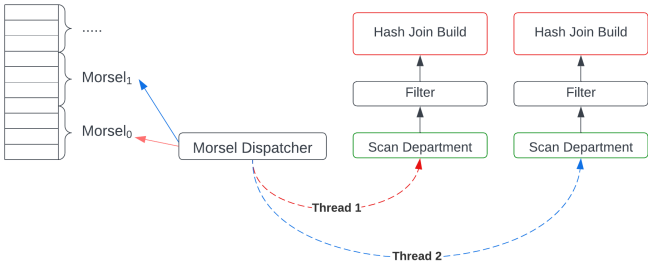


Figure 2: Morsel-driven execution of hash join build task.

Figure 2 shows a possible execution of Task_1 under two threads. The morsel dispatcher assigns Morsel_0 to Thread_1 and Morsel_1 to Thread_2. After grabbing their morsels, threads execute the rest of

```
1  for (src : srcNodes):
2     nextFrontier.setActive(src);
3     while (!curFrontier.isEmpty()):
4        swapCurNextFrontiers()
5        for (node : graph->nodes()):
6           if (curFrontier->isActive(node)):
7              for (nbr : node.scanFwd()):
8                 if (edgeCompute(node, nbr)):
9                    nextFrontier->setActive(nbr)
10       curFrontier.reset()
11    outputResults()
```

Listing 1: Serial IFE subroutine. Outer-most for loop starts an IFE subroutine from a set of source nodes.

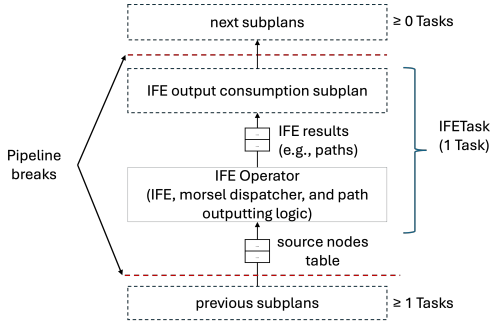


Figure 3: Recursive Clause Query Plan

the operators independently except at the last operators of subplans, which form a *pipeline break*. This is where synchronization may be needed, e.g., to build a global hash table out of local thread-level hash tables. Once Task_1 is finished, the system starts executing Task_2.

2.2 Iterative Frontier Extensions (IFE)

IFE is a BFS-like high-level algorithmic subroutine based on message passing between nodes and their neighbors. IFE is at the core of many recursive path finding operators implemented in GDBMSs, such as Neo4j [16], Kuzu[11], Memgraph [22], or DuckPGQ [35]. As such, in this paper, we focus on parallelizing recursive operators that execute IFE subroutines. x In an IFE subroutine, the computation starts from an initial *frontier*, which is a set of nodes from which the recursive computation is triggered. Then, in iterations, the neighbors of each vertex in the current frontier are explored to construct the next frontier of active nodes, until a convergence criterion is met. Depending on the particular recursive clause, a node may be visited multiple times or just once while performing the recursive computation. Listing 1 shows the pseudocode of the IFE sbroutine. Let us ignore the outer-most for loop for now. The core subroutine is between lines 3 and 10. The pseudocode is written using the `edgeCompute()` interface of systems like Pregel or Ligra. This is the interface we use in our implementation as well. For each 'active' vertex u, IFE executes `edgeCompute()` on each $e = (u, v)$ edge of u and returns true if v should be put in the next frontier. Different algorithms implement different `edgeCompute()` functions using different auxiliary data structures to store algorithm-specific per-vertex values. Below, we give an example for computing unweighted shortest path ("shortest paths" for short) lengths from a single source. Other recursive path finding algorithms, such as finding variable-length paths or path lengths, are expressed in a similar manner.

Example 1. Listing 2 shows a pseudocode `edgeCompute()` for computing the shortest path lengths from a single source s to the rest of the nodes in a graph. The algorithm keeps the lengths of the shortest paths from s to each vertex in a `len` array. This array is initialized to -∞ except s, which is set to 0. Then at each iteration, we update any node v that is visited for the first time from a currently active node u as follows. We first set `len[v]` to `len[u] + 1` and then we put v into the next frontier, by returning true.

2.3 Overall Query Plan Structure

Throughout this paper, we assume that a GDBMS compiles each recursive clause into an operator that performs IFE subroutine executions. We assume that the subplan/task in which the IFE operator runs looks as in Figure 3. Let us refer to the IFE operator's task as the IFETask. We assume that a set of prior subplans execute prior to the IFETask ("previous subplans" in the figure). The last of these subplans computes the source nodes from which a recursive path computation should be performed and passes these as a *source nodes table* to the IFETask. The IFETask starts with the IFE operator, which implements the logic of morsel dispatching, e.g., scanning source nodes from the *source nodes table*, performing the frontier extensions, as well as pipelining the output paths or path lengths of the IFE subroutines to the rest of the operators in the IFETask ("IFE output consumption subplan in the figure"). If IFETask is not the last task in the query plan, other suplans may execute after the IFETask ("next subplans" in the figure) .

```
class ShortestPathLengths {
int len[numNodes]; // initialized to UINT64_MAX
void init(src) {
  len[src] = 0
  curFrontier.add(src)
}
bool edgeCompute (e=(u, v)) {
  if (len[v] == UINT64_MAX):
    len[v] = len[u] + 1
    return true
  else:
    return false
}
```

Listing 2: `edgeCompute()` for shortest path lengths.

3 DESIGN SPACE OF SCHEDULING POLICIES

We next describe a design space of natural parallelization approaches for executing recursive path finding algorithms based on IFE, which covers and extends the two popular approaches in prior literature. Recall Listing 1, which showed the pseudocode of the serial implementation of an IFE-based algorithm. The outer-most for loop loops through each source node s in the source nodes table on line 1. Then for each s, the algorithm runs a separate IFE subroutine from s (inside a while loop). The design space we describe is based on which parts of this serial IFE-based algorithm is split into morsels for dispatching to worker threads. In Listing 1, there are 2 for loops

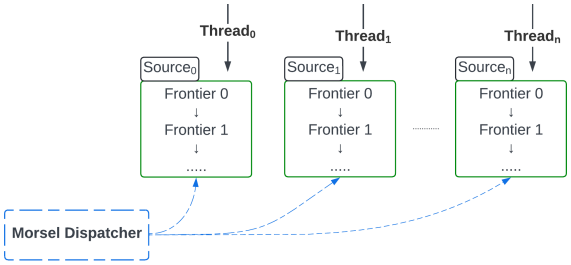


Figure 4: 1T1S scheduling policy.

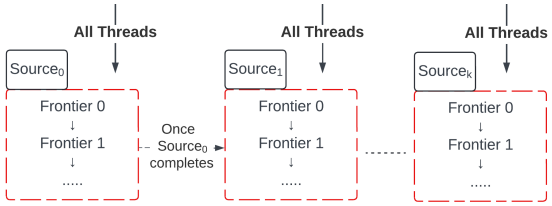


Figure 5: nT1S scheduling policy.

that contain scans that can be parallelized. These are on lines 1 and 5. The policies we describe are based on which or both of these loops they parallelize.

3.1 1T1S Policy

The first natural approach dispatches each IFE subroutine execution from a source as a unit of work (Line 2-11 in Listing 1). Specifically, the first for loop on line 1 scans each source node s and starts a new IFE subroutine from s. We call these IFE subroutines as *source morsels*. In vanilla morsel-driven parallelism, e.g., adopted in Neo4j [15], DuckPGQ [35] and existing approach in Kuzu, the scan in this first for loop is done by scans from a source table, and the recursive computation is parallelized at this scan. This is the approach we call 1T1S for **1**-**T**hread-to-**1**-**S**ource node policy. Figure 4 shows the high-level execution of this policy when there are n sources to run IFE subroutines from.

The advantage of 1T1S is that when there are many sources from which an IFE subroutine should be executed, 1T1S can easily keep threads busy. Further, under the 1T1S policy, the IFE subroutine implementations can use fast data structures that do not contain any synchronization primitives, such as hardware or software locks. This is because 1T1S guarantees that only one thread works on any IFE computation. At the same time, this approach will not be able to utilize multiple threads efficiently on queries that contain fewer sources than there are threads, e.g., only one source. Worse, if the system uses large morsel sizes, e.g., 100K as in the original morsel-driven parallelism paper [20] or 131K as in the DuckPGQ system, this approach can fail to parallelize even when there are many sources in the query.

3.2 nT1S Policy

The second natural approach dispatches work only from the inner for loop on line 5 in Listing 1. This policy takes each source s one by one and splits the scan of the current frontier of each iteration of the IFE subroutine from s into morsels. We refer to these as *frontier morsels*. Therefore, each frontier morsel is a set of active nodes and a single thread is responsible for executing the `edgeCompute()` function on each neighbor of each node in the frontier morsel. This is the approach implemented in parallel graph analytics systems, such as Ligra [31] and Pregel [23]. We call this approach nT1S, for **n**-**T**hreads-to-**1**-**S**ource node policy. Figure 5 shows the high-level execution of this policy.

The advantage of nT1S is that unlike 1T1S, nT1S can share work when there are fewer sources in the query than there are threads.

	Threads (→)						Speedup
IFE Level (↓)	**1**	**2**	**4**	**8**	**16**	**32**	
L0 (src node)	1	1	2	1	1	1	**1.0x**
L1 (17 nodes)	3	2	2	1	1	2	**1.5x**
L2 (2053 nodes)	6	4	4	2	2	3	**2.0x**
L3 (64326 nodes)	65	38	20	14	11	7	**9.3x**
L4 (276175 nodes)	190	109	60	38	20	16	**11.9x**
L5 (56731 nodes)	31	21	14	6	4	6	**5.2x**
L6 (6044 nodes)	5	4	2	2	2	3	**1.7x**
L7 (1465 nodes)	3	2	2	2	1	2	**1.5x**
L8 (458 nodes)	2	2	1	2	1	2	**1.0x**
L9 (93 nodes)	1	1	1	1	1	1	**1.0x**
L10 (27 nodes)	1	1	1	1	1	1	**1.0x**
L11 (7 nodes)	1	1	1	1	1	1	**1.0x**
Total Runtime	331	198	138	95	72	68	**4.8x**

Table 1: Scalability of each frontier level (in ms).

However, this approach is limited by how much a single IFE subroutine from one source can parallelize. Due to Amdahl's Law for parallelism [2], the scalability of any program will be limited to its parallelizable parts. In recursive path finding algorithms, the frontiers tend to start sparse, grow large, and shrink again and become sparse. The sparse frontiers limit how much parallelism can be achieved cumulatively.

Table 1 is an example experiment demonstrating this behavior in our implementation of the nT1S policy in Kuzu. The experiment runs a shortest path query from a single source to all destinations on the LDBC100 graph, returning the lengths of paths. LDBC100 graph contains 448K nodes and 19.9M edges. The table presents how many nodes exist in each frontier and how much speedup there is on each frontier as we scale the number of worker threads on the system from 1 to 32. As shown, when a frontier is dense, as in level 4, we can obtain good scalability of 11.9x, yet on other frontiers, which cumulatively add up to 36% of the computation, there is at most 4.1x speedup and often much less. This leads to an overall speedup of 4.8x. This phenomenon limits the scalability of using nT1S policy alone.

3.3 nTkS Policy

We next identify a third hybrid policy that combines parallelization of both loops. We call this approach nTkS policy, for **n**-**T**hreads-to-**k**-**S**ource nodes. nTkS maintains the execution of multiple k concurrent IFE subroutines, i.e., source morsels. However, it is not the source morsels that are given as morsels to threads. Instead, each frontier of each source morsel is split into morsels but different threads can be dispatched frontier morsels from different source morsels. Figure 6 shows the high-level execution of this policy.

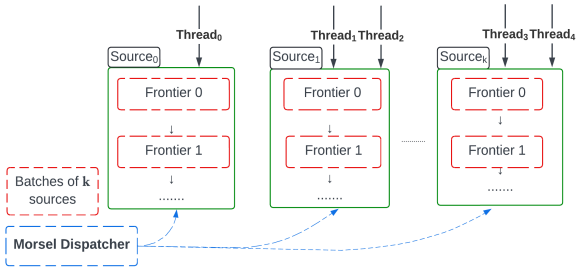


Figure 6: nTkS scheduling policy.

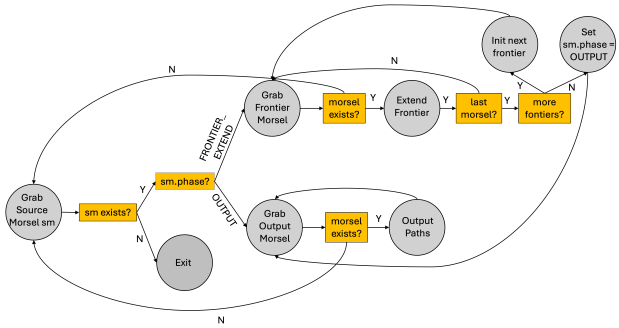


Figure 7: Control flow of IFE operator and morsel dispatcher.

The advantage of the nTkS policy is that when there are few source nodes in the query, nTkS aims to mimic the behavior of nT1S. When there are many source nodes, nTkS can outperform nT1S as when frontiers of a source morsel gets sparse, it can keep idling threads active on other source morsels with denser frontiers. When there are many source nodes, nTkS behaves similar to 1T1S. However, nTkS can outperform 1T1S because whenever there are fewer source morsels than number of threads, while 1T1S starts keeping threads idle, nTkS starts behaving more like nT1S where multiple threads work on the same source morsel. This situation can arise in two ways: (i) either the original query does not have many source nodes; or (ii) the query contains many source nodes initially but as the computation progresses some source morsels finish, and there are fewer source morsels left to work on than the number of threads.

3.4 Multi-Source Morsels

We next review the multi-source BFS (MS-BFS) optimization from reference [36], which is also used by the DuckPGQ system [35]. MS-BFS organizes up to 64 IFE subroutines, IFE_1, ..., IFE_{64}, as a unit and runs them concurrently. For example, if a shortest path query contains 64 or more source nodes, MS-BFS runs the shortest path computation from 64 sources at a time. That is, first, the first frontiers of all 64 IFE subroutines are extended concurrently, then the second frontiers, then the third frontiers, so on and so forth. For each vertex u in the graph, we store 64 bits to represent u's active state in each of the 64 concurrent IFEs, i.e., the i'th bit represents whether u is active in IFE_i. These bits are referred to as "lanes" in reference [36]. If u is active in iteration j for t IFE subroutines, say IFE_{ℓ_1},...,IFE_{ℓ_t}, then instead of scanning the neighbors of u t times, we can scan them once and run t `edgeCompute()` on each of those neighbors. MS-BFS also reduces the amount of writes performed to set nodes active in frontiers. Specifically, one can set a neighbor v of u active in up to t IFE subroutines with a small number of bitwise operations.

We can model MS-BFS also as a morsel dispatching policy that packs multiple source nodes on line 1 of Listing 3 into *multi-source morsels*. Using this optimization, we can have variants of the previous policies. For example, if the query contains 128 source nodes, the nTkMS policy, for **n**-**T**hreads-to-**k**-**M**ulti-**S**ource nodes, can launch two multi-source morsels and have multiple threads grab frontier morsels from each multi-source morsel. We will empirically evaluate the behavior of nTkMS in Section 5.

4 IMPLEMENTATION DETAILS

We next describe the details of our implementation of these policies in Kuzu. Our implementation can be found here [6]. Briefly, Kuzu is a columnar, disk-based GDBMS. It has a vectorized [1, 11], i.e., batch-at-a-time, query processor that adopts morsel-driven parallelism. The query processor is pull-based, where parent operators pull data from their children tuples via `getNextTuples()` function calls. Kuzu stores the adjacency lists of nodes in disk-based compressed sparse-row (CSR) structures, and access to adjacency lists happens through the system's buffer manager.

```
1  struct SourceMorsel {
2    Frontier currentFrontier, nextFrontier;
3    int curIter; // current iteration of the IFE subroutine
4    IFEPhase phase; // one of enum {FRONTIER_EXTENSION, OUTPUT}
5    AuxState auxState; // other algorithm-specific data, e.g., Parents
6  }
7  class IFE : PhysicalOperator {
8  MorselDispatcher md;
9  EdgeCompute ec; // recursive clause-specific edgeCompute function
10 Graph graph; // provides iterator interface to scan nbrs
11 DestinationNodeMask targetDsts; // possible destination nodes
12 bool getNextTuples() {
13  while (true) {
14    SrcMorsel sm = md.grabSrcMorselIfNecessary(&sm);
15    if (sm == null): return false; // exit
16    if (sm.phase == OUTPUT):
17      OutputMorsel om = md.grabOutputMorsel(&sm);
18      if (om == null): continue;
19      outputPaths(om, sm, dsts); // pipeline outputs to parent op
20      return true;
21    else if (sm.phase == EXTEND_FRONTIER):
22      FrontierMorsel fm = md.grabFrontierMorsel(&sm);
23      if (fm == null): continue;
24      extendFrontier(fm, sm, ec, graph);
25      // if frontier finished, update phase or start new frontier
26      sm.checkIfFrontierFinished();
27   }}}
28 void extendFrontier(FrontierMorsel fm, SourceMorsel sm,
29                     EdgeCompute ec, Graph graph) {
30   for (auto u : fm):
31     if (sm.cFrontier->isActive(u)):
32       for (auto v : graph->scanFwd(u)):
33         if (ec.compute(u, v, sm)):
34           sm.nFrontier->setActive(v);
```

Listing 3: Pseudocode of the IFE operator.

4.1 IFE Operator

Listing 3 shows the pseudocode implementation of the generic IFE operator, which we implemented as a standard physical operator in Kuzu. The operator is constructed with several fields:

- **MorselDispatcher** implements the different policies we outlined in Section 3 (discussed more momentarily).
- **Graph** is an interface that provides functions, such as scanFwd, that provide an iterator interface to scan neighbors of vertices in the database. Internally, it translates these calls to calls that read database records through the system's buffer manager.
- **EdgeCompute** implements the edgeCompute() function that implements a specific instance of an IFE-based recursive algorithm. We implemented different edgeCompute() functions for the different recursive clauses in Cypher and whether the query requires computing outputs actual paths or only the path lengths, which require operating on separate auxiliary data structures.[2] For reference, Listing 4 shows an example edgeCompute() computing the shortest paths clause that returns the actual paths (instead of path lengths).
- **DestinationNodeMask** targetDsts is a "node mask" that keeps track of the specific nodes in the graph, for which an output path or path length needs to be returned. The mask is an array of boolean values of size the total nodes in the graph.

```
35 class ShortestPaths : EdgeCompute {
36 void compute(nodeID u, nodeID v, SourceMorsel sm) {
37    if (!sm.auxState.visited[v])
38       sm.auxState.parents[v].addParentEdge(u, sm.curIter);
39       sm.auxState.visited[v] = true; // atomic memory_relaxed op
40 }}
```

Listing 4: edgeCompute function that computes shortest paths.

Figure 7 summarizes the high-level control flow of the IFE operator assuming a morsel dispatcher that gives source morsels instead of multi-source morsels. The thread W running the operator, inside a while loop, first grabs a source morsel sm from the morsel dispatcher md by calling grabSrcMorselIfNecessary() (line 14). It is inside this function that different morsel dispatching policies are implemented.

The SourceMorsel structure, shown on top of Listing 3, represents the state of an IFE subroutine execution from a single source s. It contains the data structures that are needed by the IFE subroutine, which include the current and next frontiers, current iteration of the IFE subroutine, and the auxiliary data structures to store the intermediate paths that are computed (auxState field). It further contains a phase field, which can take one of two values:

- FRONTIER_EXTENSION indicates that the IFE subroutine has not yet finished, i.e., the frontiers have not yet converged.
- OUTPUT indicates that the IFE subroutine has finished, i.e., all paths from s have been computed and now these paths need to be pipelined to the parent operator.

After W calls grabSrcMorselIfNecessary(), if there are no more source morsels (line 15), the computation has finished and the operator exits. Otherwise, if sm's phase is FRONTIER_EXTENSION, W grabs a frontier morsel from sm and runs edgeCompute() on each active node in this frontier morsel. Obtaining frontier morsels is a simple operation that is independent of morsel dispatching policy, and returns back a range of integer node IDs. After W finishes its frontier morsel it calls sm.checkIfFrontierFinished() (line 26), which checks if all active nodes in the current frontier are processed and if W is the last thread to finish its frontier morsel. If so, then W either moves the computation to the next frontier or if there are no active nodes in the next frontier, then sets sm's phase to OUTPUT.

If sm's phase is OUTPUT, then the thread grabs an *output morsel*. The output morsel is a range of node IDs that represent destination nodes. For each valid destination node d, if the computation found paths from s to d, then W outputs each path or path length from s to d.

[2]We found the Graph and edgeCompute() interfaces very helpful in implementing different IFE-based recursive algorithms and recommend it to system developers. These higher-level interfaces are better fits for implementing recursive path finding algorithms than the standard tuple or vector-based query processor interfaces of DBMSs to scan and operate on database records.

4.2 Data Structure Implementations

Frontier: We use a dense frontier implementation to store active nodes in the current and next frontier. Dense frontiers are arrays that store one boolean value per vertex in the graph. When frontiers are swapped at the end of each iteration, we have a variant of the *sparse frontier optimization* from Ligra [31]. Specifically, if the number of nodes in the next frontier is less than 1/8th of all nodes, we construct an additional sparse version of the frontier in a single threaded manner and write the ID of every active node to an array.
Parents: When computing paths, we keep paths compactly by keeping track of the "parents", i.e., last edges that were used to visit each node. Figure 8 gives an overview of the data structure we use. Our data structure consists of: (i) a dense pre-allocated array that keeps an 8-byte pointer for each vertex (initialized to null pointers) and that is shared across all threads; and (ii) a set of memory buffers that are owned by and written to by separate threads. When thread Ti needs to write a parent v for node u, it writes to its memory buffer a tuple with the information about v, specifically v's ID and the edge ID of the (v, u) edge, and an 8-byte pointer to the next parent edge on the path. Therefore, for each edge of each path we compute, we store an additional 24 bytes. Then Ti updates u's pointer at the dense array to point to this tuple using compare-and-swap operation (CAS). This computation happens inside addParentEdge() function on the Parents data structure, as shown in the example edgeCompute() function in Listing 4. Figure 8 shows an example when two threads $T1$ and $T2$ attempt to add two parent edges to u, respectively from w_1 and w_2.
Shortest Paths-specific structures: Shortest paths computations has the property that each node u can be active only once in one frontier. To ensure nodes are not put into the frontiers multiple times, we maintain a global *visited array*, which is a dense structure that keeps a boolean value per vertex. Further, if a shortest paths computation computes only path lengths instead of using the Parent structure, we use dense structures to store only the lengths of paths, which we store as 1 byte in our implementation, which is enough to store the path lengths in the datasets in our experiments.
Data structures for multi-source shortest paths implementations: Here, we followed the overall implementation in reference [36] and use 64 bits (for 64 lanes) to represent the active state of each node in frontiers. We used two frontiers for current and next and a third visited data structure that also stores 8 bytes (64 bits) per node that indicate for which of the up to 64 IFE subroutines, IFE_1, ..., IFE_ℓ, a node is visited or in the frontier of. Therefore,

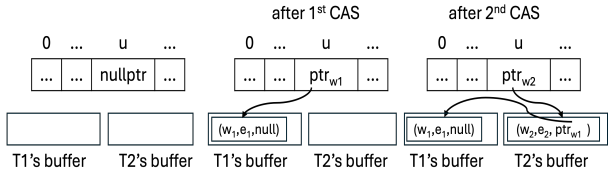


Figure 8: Parents data structure to keep multiple paths.

these auxiliary data structures require 3*8=24 bytes per node per multi-source morsel. Operations on frontiers and visited arrays can be done with efficient bitwise operations to update a single node's value concurrently for multiple IFE sub-routines. We also require auxiliary data structures, such as path lengths and parents data structures for each source in a multi-source morsel. In total, for each multi-source morsel if there are 64 sources in it, the upfront memory requirements are as follows. If we are returning only path lengths, we allocate 24 + 1*64 = 88 bytes per node in the graph. We do not require further memory during the computation. If we are returning paths, we allocate upfront 24 + 8*64= 536 bytes per node and allocate more data during the computation for the thread-level memory buffers to store additional edges in the paths computed.

Finally, we note that suppose a node u is active in iteration i for at least one IFE subroutine. We can tell this by inspecting that its frontier value X is not equal to 0. Further, to update the correct auxiliary data structures, we need to compute which bits of X are exactly 1. For this, we use the builtin C++ function [12] that gives the index of the first 1 bit in X and we use it iteratively to read the index of every 1 bit.

4.3 Scheduling Policy Implementations

Different scheduling policies are implemented as different morsel dispatcher logics for assigning source morsels to threads. This is represented by the `grabSrcMorselIfNecessary(SourceMorsel sm)` function of the `MorselDispatcher` class in Listing 3. Our implementations of `1T1S` is straightforward. We discuss only `nTkS` since `nT1S` is a special case of `nTkS`. `nTkS` is configured with a $k \geq 1$ value and launches up to k source morsels as follows. Whenever a thread W asks for source morsel to work on, as long as there are fewer than k source morsels that are already launched, and there are more sources in the query, W is dispatched a new source morsel `sm`. Then, W keeps working on `sm` as long as it can grab frontier morsels from it (or if `sm` is in OUTPUT phase, then as long as W can output paths from `sm`). That is, our implementation of `nTkS` is "sticky". W can be dispatched another source morsel if it cannot find a frontier morsel from `sm`, which happens at the end of each iteration i of the IFE subroutine.

We also implemented the `nTkMS` policy. In this policy, threads get a `MultiSourceMorsel` struct that represents up to 64 concurrent IFE subroutines and the necessary auxiliary data structures. This policy is also configured with a $k \geq 1$ value and follow the same logic for dispatching new multi source morsels as the `nTkS` policy.

5 EVALUATION

We next evaluate our different morsel dispatching policies under a variety of queries that contain different numbers of sources. We further study what is a good value of k for both `nTkS` and `nTkMS` policies and the performance of the `nTkMS` policy. All of our datasets and queries can be found in our code repo [6].

5.1 Experimental Setup

Baselines and Kuzu configurations: We compare 4 different policies that we implemented in Kuzu with three additional baselines:

- Kuzu-`1T1S` , Kuzu-`nT1S` , Kuzu-`nTkS` , and Kuzu-`nTkMS`, represent the policies we implemented in Kuzu. Throughout our evaluations, we set k to 32 in Kuzu-`nTkS` configuration, except in Section 5.5. For experiments using Kuzu-`nTkMS`, we specify the k values explicitly.
- Neo4j: We use Neo4j's v5.16, which implements a morsel dispatching policy that is akin to the `1T1S` policy. The difference is instead of source morsels, Neo4j assigns "(source, destination)-morsels" to threads. Specifically, each thread is assigned a morsel of 1024 (source destination) pairs and independently computes shortest paths between each pair using an IFE-based computation, which extends frontiers one step from the source node side and one step from the destination node side. However, Neo4j does not dispatch work at the frontier level. As such, this approach behaves similarly to the `1T1S` policy. As we will explain in detail below, our queries limit the number of source nodes but not the destination nodes. Since Neo4j assigns morsels based on (source, destination) pairs, Neo4j becomes very inefficient on these queries. Therefore, we test Neo4j on modified queries that limit the number of destination nodes to 1024.
- Ligra: We use Ligra, which implements frontier-level parallelism, as an external baseline for the `nT1S` policy.
- `Duck-nTkS`: We tried using the original version of DuckPGQ as a baseline for comparison, however we encountered several problems. Similar to Neo4j, DuckPGQ also issues (source, destination) morsels. Even when we limited the destination nodes in our queries, we found DuckPGQ's MS-BFS implementation to be very slow and also not parallelize well. We reached out to the authors of DuckPGQ [35] and verified that this was expected behavior. Instead, we modified the DuckPGQ code and implemented our `nTkS` policy considering only the source nodes and without the MS-BFS optimization. We refer to this modified DuckPGQ as `Duck-nTkS`. Our implementation can be found here [10]. Unlike Kuzu, which has a disk-based CSR index that gets created once when the database is constructed, DuckPGQ creates an on-the-fly in-memory CSR for each query. We observed that the creation of these in-memory CSRs take a significant time, so instead of end-to-end times, we only report the time taken by `Duck-nTkS` to perform the IFE computation.

Hardware: We use a single machine with Intel Xeon E5-2670 CPU @2.60GHz processors and 512GB of memory. The machine has 16 physical cores (32 virtual cores) across 2 NUMA nodes with 32 KB of L1 data cache, 32KB L1 instruction cache, 256 KB L2 cache and 20 MB L3 cache sizes. We use the systems in their default settings,

e.g., Kuzu sets its buffer manager size by default to 80% of the host memory (410 GB).

Datasets: Table 2 lists the datasets used in our experiments, which include the following: (i) LDBC100 is a synthetic graph generated with LDBC social network benchmark [34] at scale 100; (ii) LiverJournal is the LiveJournal social network graph from the SNAP graph dataset repository [32]; (iii) Spotify is a graph released by the music streaming platform Spotify representing songs and pairs of songs listened together in listening sessions [18]; and (iv) Graph500-28 is one of the a synthetic graphs released by the LDBC benchmark at scale 28. We choose these datasets as they vary in their sizes from 20 million to 4.2 billion edges, but each one is large enough that even recursive queries from a few sources have room for benefiting from parallelization.

Name	$\|V\|$	$\|E\|$	Avg Degree
LDBC100	448,626	19,941,198	44
LiveJournal	4,847,571	68,993,773	14
Spotify	3,604,454	1,927,482,013	535
Graph500-28	121,242,388	4,236,163,958	35

Table 2: Datasets.

Query workloads: We used query workloads that find shortest paths starting from a set of source nodes to the rest of the nodes in the graph, and returning the lengths of the paths or actual paths. In Cypher, these queries have the below structure:

```
MATCH p = (a:Node)-[r:Rel* SHORTEST]->(b:Node)
WHERE a.id IN [s1, ... sn]
RETURN len(p)
```

si, ..., *sn* values above identify the IDs of sources. We picked each source node in our workloads randomly and ensured that we can perform at least 3 levels of IFE computation. We used 3 values for *n* that limit the number of source nodes as follows:

- **1-source workloads** in which the queries contain 1 source.
- **8-source workloads** in which the queries contain 8 sources.
- **64-source workloads** in which the queries contain 64 sources.

In Section 5.6, when we evaluate Kuzu-nTkMS policy, we use workloads with 128 and 256 sources as well.

Ligra does not have a query language and its existing shortest path algorithm finds shortest path lengths from a single source. We modified the code slightly to run a separate IFE subroutine from multiple source one at a time. Ligra's existing shortest paths algorithm only computes path lengths and not the actual paths, so we did not use queries that return paths for Ligra.

As discussed above, for Neo4j, we also modified the queries in 1/8/64-source workloads by putting a separate predicate to limit the number of destinations to 1024. Limiting the queries to contain 1024 destinations ensures that these queries have the same number of (source, destination) morsels in Neo4j as there are source morsels in Kuzu configurations. With this workload change, we can no longer compare the raw performance of Neo4j with our other baselines. However our primary goal is to compare the parallelism behaviors of these systems under representative workloads instead of comparing their raw performances.

Our experiments measure the systems we study with 1, 2, 4, 8, 16, and 32 threads. Tables 3 and 4 report numbers only for the experiments that contain 1, 8, and 32 threads, since these are the representative parallelism levels to explain the behaviors we want to highlight. The longer version of our paper contains additional figures that show the performances of these systems with 2, 4, and 16 threads. All reported numbers are based on execution on a warmed up database. We first run each query once to warm up each system's buffer manager cache, discard the first execution number, then report the average of an additional 3 more executions. The average runtime deviation was 8% across our experiments. The median deviations were 12.5%, 11%, 5%, and 3% on any experiment, respectively, on LDBC, LJ, Spotify and Graph500-28.

5.2 1-Source Workloads

In our first set of experiments we measure the runtime of our baselines on our 1-source workloads as we increase the parallelism from 1 thread to 32 threads. We expect that systems that implement 1T1S to not parallelize and those that implement frontier parallelism (i.e., nT1S policy) to parallelize with more threads. We also expect systems that implement the nTkS policy to parallelize. In particular, we expect Kuzu-nTkS to mimic the behavior of Kuzu-nT1S.

Tables 3a and 4a show the runtimes under our workloads that return path lengths and paths, respectively. The tables also report the CPU utilizations of the systems. Recall that the machine we use supports 32 virtual cores/threads. As an example, a 50% utilization indicates that 16 of the threads were busy doing useful work.

Observe that as we expect, both Kuzu-1T1S and Neo4j, which implements a 1T1S policy, cannot benefit from additional threads. Instead, Kuzu-nT1S and Ligra benefit from additional threads. On the queries that return path lengths, Ligra achieves between 9.0x to 16.4x improvement, while Kuzu-nT1S achieves between 5.0x to 12.8x improvement. Importantly, Kuzu-nTkS mimics the behavior of Kuzu-nT1S, producing almost identical runtime numbers. Similarly Duck-nTkS demonstrates comparable scalability factors to Kuzu-nT1S on all datasets except for LDBC100. This is because the underlying DuckDB system allocates as many threads to any pipelines as there are number of *row groups* [9] in the underlying scanned table of a pipeline. A row group in DuckDB contains 122880 rows. The pipeline for the IFE computation on LDBC scans from the node table in DuckDB, which contains 448626 rows. So the IFE pipeline is assigned only 4 threads on LDBC and the CPU utilization of Duck-nTkS is around 4/32=12.5%.

We note that Kuzu-nT1S and Ligra are two independent systems at different performance levels. Ligra is generally faster because it is a purely in-memory system and accesses adjacency lists directly, while Kuzu-nT1S accesses them through the buffer manager. Note that the Duck-nTkS numbers we report are also faster than Kuzu-nTkS for two reasons. First, DuckPGQ also uses an in-memory CSR and does not use a buffer manager. Second, recall that we only report the IFE computation pipeline of Duck-nTkS, as Duck-nTkS's end-to-end runtime is dominated by its on-the-fly in-memory CSR construction pipeline.

	Threads (→)	1	8	CPU (%)	32	CPU (%)
LDBC	nTkS	331	95 (3.5×)	18	73 (4.5×)	67
	nT1S	310	91 (3.4×)	18	62 (5.0×)	70
	1T1S	312	304 (1.0×)	3	310 (1.0×)	3
	Ligra	130	17 (7.6×)	21	14 (9.3×)	82
	Neo4j	121	122 (1.0×)	3	121 (1.0×)	3
	D-nTkS	69	29 (2.4×)	12	27 (2.6×)	12
LJ	nTkS	1975	440 (4.5×)	22	296 (6.7×)	85
	nT1S	1890	354 (5.3×)	23	274 (6.9×)	85
	1T1S	1814	1789 (1.0×)	3	1811 (1.0×)	3
	Ligra	1411	190 (7.4×)	24	140 (10.1×)	92
	Neo4j	324	343 (1.0×)	3	323 (1.0×)	3
	D-nTkS	518	123 (4.2×)	23	71 (7.3×)	89
Sp	nTkS	15012	2007 (7.5×)	24	1184 (12.7×)	95
	nT1S	14389	2106 (6.8×)	24	1121 (12.8×)	95
	1T1S	14372	14289(1.0×)	3	14197 (1.0×)	3
	Ligra	20428	3394 (6.1×)	24	1243 (16.4×)	99
	Neo4j	3481	3562 (1.0×)	2	3561 (1.0×)	2
	D-nTkS	7976	1203 (6.6×)	24	630 (12.7×)	95
G-28	nTkS	131963	18662 (7.1×)	22	14854 (8.9×)	91
	nT1S	128090	17980 (7.1×)	23	14032 (9.1×)	92
	1T1S	126929	125116(1.0×)	3	120198(1.0×)	3
	Ligra	113276	17116 (6.6×)	22	12582 (9.0×)	92
	Neo4j	136929	135116(1.0×)	3	119092(1.0×)	3
	D-nTkS	61591	10557 (5.8×)	22	6288 (9.8×)	91

(a) 1-Source Workload

1	8	CPU (%)	32	CPU (%)
2177	320 (6.8×)	24	190 (11.5×)	97
2089	485 (4.3×)	17	401 (5.2×)	66
2077	358 (5.8×)	22	346 (5.8×)	23
635	118 (5.4×)	21	86 (7.4×)	79
808	157 (5.1×)	22	159 (5.1×)	23
602	174 (3.5×)	13	173 (3.5×)	13
15800	2311 (6.8×)	24	1258 (12.6×)	99
15708	3141 (5.0×)	18	2244 (7.0×)	70
15749	2581 (6.1×)	23	2316 (6.8×)	24
8803	1543 (5.7×)	20	871 (10.0×)	78
14993	2541 (5.9×)	23	2458 (6.1×)	24
4062	745 (5.5×)	23	404 (10.1×)	98
121401	16845 (7.2×)	24	8723 (14.0×)	98
121307	16617 (7.3×)	24	8985 (13.5×)	93
121703	20627 (5.9×)	23	19503 (6.2×)	24
131963	23992 (5.5×)	22	10828(12.2×)	95
63138	12428 (5.1×)	22	12380 (5.2×)	23
67712	10213 (6.6×)	23	5514 (12.3×)	98
716313	100548 (7.1×)	23	64943 (11.0×)	96
713313	127377 (5.6×)	17	90292 (7.9×)	69
710989	106117 (6.7×)	22	103557(7.0×)	23
647141	156599 (4.1×)	17	69137 (9.4×)	74
1819017	271495 (6.7×)	23	267502(6.9×)	23
348438	58744 (6.0×)	23	33725 (10.3×)	95

(b) 8-Source Workload

1	8	CPU (%)	32	CPU (%)
15683	2140 (7.3×)	24	1073 (14.6×)	98
15598	3391 (4.6×)	18	3058 (5.1×)	67
15609	2230 (7.0×)	21	1334 (11.7×)	80
3463	815 (4.2×)	19	498 (7.0×)	75
7581	1025 (7.4×)	21	681 (11.1×)	81
4329	1178 (3.6×)	12	1165 (3.7×)	12
124197	18601 (6.7×)	24	8952 (13.9×)	98
123957	25824 (4.8×)	19	16310 (7.6×)	72
124087	20681 (6.0×)	21	12285 (10.1×)	82
66141	11617 (5.7×)	20	7622 (8.7×)	78
113919	18374 (6.2×)	21	10689 (10.7×)	80
34222	5692 (6.0×)	24	3074 (11.1×)	98
955169	132885 (7.2×)	24	82740 (11.5×)	91
955109	136444 (7.0×)	24	72109 (13.3×)	94
948231	155448 (6.1×)	21	93884 (10.1×)	80
1065394	184212 (5.8×)	20	91849 (11.6×)	92
2381885	590591 (4.0×)	19	246495 (9.7×)	80
818891	133735 (6.1×)	23	58913 (13.9×)	93
4959436	795891 (6.2×)	24	416996 (11.9×)	95
4929436	1095430 (4.5×)	18	631978 (7.8×)	70
4939436	809743 (6.1×)	21	519940 (9.5×)	73
4407690	1275313 (3.5×)	17	523052 (8.4×)	74
5898289	966932 (6.1×)	22	620872 (9.5×)	78
2748338	407708 (6.7×)	24	231375 (11.8×)	96

(c) 64-Source Workload

Table 3: Runtime (ms) and CPU utilizations (for 8 and 32 threads) for path length queries. D-nTkS is `Duck-nTkS`.

	Threads (→)	1	8	CPU (%)	32	CPU (%)
LDBC	nTkS	971	152 (6.4×)	18	89 (10.9×)	87
	nT1S	951	158 (6.1×)	18	87 (11.0×)	89
	1T1S	959	961 (1.0×)	3	965 (1.0×)	3
	Neo4j	172	175 (1.0×)	3	178 (1.0×)	3
	D-nTkS	202	67 (3.0×)	13	67 (3.0×)	13
LJ	nTkS	5129	954 (5.4×)	22	540 (9.5×)	87
	nT1S	5117	979 (5.2×)	23	531 (9.6×)	88
	1T1S	5095	5079 (1.0×)	3	5158 (1.0×)	3
	Neo4j	1110	1069 (1.0×)	3	1057 (1.0×)	3
	D-nTkS	1345	250 (5.4×)	23	141 (9.6×)	88
Sp	nTkS	24471	3437 (7.1×)	24	1847 (13.3×)	93
	nT1S	24171	3597 (6.7×)	24	1778 (13.6×)	94
	1T1S	24729	24829(1.0×)	3	24419(1.0×)	3
	Neo4j	5671	5579 (1.0×)	2	5694 (1.0×)	2
	D-nTkS	13001	1826 (7.3×)	24	981 (13.2×)	94
G-28	nTkS	212099	28984 (7.3×)	22	19325 (11.0×)	91
	nT1S	215005	29019 (7.4×)	23	19201 (11.2×)	92
	1T1S	210019	208298(1.0×)	3	205902(1.0×)	3
	Neo4j	30192	31777 (1.0×)	3	30395 (1.0×)	3
	D-nTkS	98992	13527 (7.4×)	22	9019 (11.2×)	91

(a) 1-Source Workload

1	8	CPU (%)	32	CPU (%)
5175	727 (7.1×)	24	391 (13.3×)	99
4961	844 (5.9×)	17	586 (8.5×)	68
4991	745 (6.7×)	23	674 (7.4×)	24
3485	601 (5.8×)	22	580 (6.1×)	23
1431	485 (3.0×)	13	477 (3.0×)	13
39981	5701 (7.1×)	24	3236 (12.4×)	99
40343	6165 (6.5×)	18	4101 (9.8×)	71
39091	6023 (6.5×)	23	5505 (7.1×)	25
50195	9842 (5.1×)	23	9126 (5.5×)	24
10278	1465 (7.0×)	24	830 (12.5×)	99
194690	27462 (7.1×)	24	13235 (14.7×)	97
193905	25392 (7.6×)	24	13063 (15.0×)	98
192998	25733 (7.5×)	23	24775 (7.8×)	24
34044	5158 (6.6×)	22	5007 (6.8×)	24
108589	15317 (7.2×)	23	7081 (14.9×)	98
1236007	167390 (7.4×)	23	99185 (12.5×)	95
1229780	168026 (7.3×)	17	119275 (10.3×)	65
1231109	163930 (7.5×)	22	167270 (7.4×)	24
247201	41898 (5.9×)	23	40524 (6.1×)	23
601234	81423 (7.8×)	23	48246 (12.6×)	95

(b) 8-Source Workload

1	8	CPU (%)	32	CPU (%)
35827	4808 (7.4×)	24	2315 (15.5×)	98
35992	6209 (5.8×)	18	4530 (8.0×)	68
35920	5434 (6.6×)	21	3292 (11.0×)	79
29474	4151 (7.1×)	21	2631 (11.2×)	80
9889	3015 (3.3×)	12	2996 (3.3×)	13
318855	44892 (7.1×)	24	22089 (14.5×)	97
321421	48396 (6.6×)	19	32281 (9.8×)	70
321879	49519 (6.5×)	21	27025 (11.9×)	80
478855	61391 (7.8×)	21	36140 (13.2×)	80
87859	12369 (7.8×)	24	6086 (14.9×)	97
1578916	217542 (7.3×)	24	126666 (12.5×)	90
1550691	203740 (7.6×)	24	110323 (14.1×)	95
1561234	207060 (7.5×)	21	148688 (10.5×)	81
2023172	293213 (6.9×)	19	175928 (11.5×)	82
1188343	163729 (7.2×)	23	98210 (12.2×)	90
8315386	1285103 (6.5×)	24	615954 (13.5×)	94
8419386	1326979 (6.3×)	18	889690 (9.5×)	69
8419193	1360128 (6.2×)	21	764686 (11.0×)	75
10759211	1605852 (6.7×)	22	847182 (12.7×)	77
4608082	712157 (6.6×)	24	301182 (15.3×)	94

(c) 64-Source Workload

Table 4: Runtime (ms) and CPU utilizations (for 8 and 32 threads) for path queries. D-nTkS is `Duck-nTkS`.

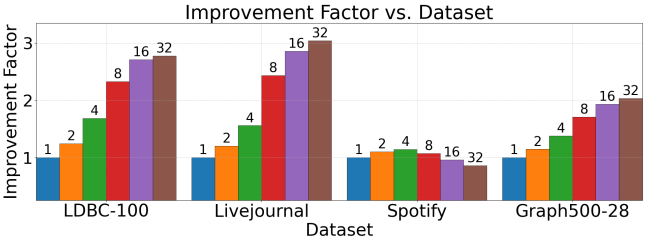
5.3 8-source Workloads

We next evaluate our baselines on workloads that contain multiple sources but smaller than the number of available threads, using our 8-source workloads. Here we expect that systems that implement `1T1S` policies to benefit from parallelism up to 8 threads, which are the number of sources in these workloads. We expect `nT1S` policies to behave similar to the single-source experiments, since these policies should repeat their previous behaviors on each source. Kuzu-`nTkS` should on the other hand outperform both policies. This is because unlike `1T1S`, it can use all threads when more than 8 threads are available and unlike `nT1S` it is not limited by the amount of parallelism achievable on a single frontier.

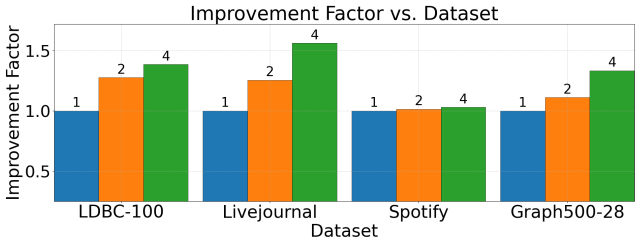
Tables 3b and 4b show our results for all of our datasets. Observe that now both Kuzu-`1T1S` and Neo4j parallelize up to 8 threads and then flatten as there are not enough source morsels to issue to threads. This is why the CPU utilization for both these cases reach at most 25%, which corresponds to 8 threads out of 32, across all datasets. Kuzu-`1T1S` achieves between 5.8x and 7.8x improvements, while Neo4j achieves between 5.1x and 6.9x improvements. Further, Kuzu-`nT1S` and Ligra achieve similar scalability levels as before: on the queries that return path lengths, Ligra achieves between 7.4x

to 12.2x improvement, while Kuzu-nT1S achieves between 5.2x to 13.5x improvement.

Kuzu-nTkS achieves more robust parallelism on the same workloads, parallelizing between 11.0x-14.0x. Duck-nTkS behaves similar to Kuzu-nTkS except on LDBC, where it again is assigned only 4 threads due to row group based processing. The improvements of Kuzu-nTkS over Kuzu-nT1S is especially visible on the LDBC graph, where Kuzu-nT1S achieves a very limited parallelism of 5.2x. Instead, Kuzu-nTkS can achieve 11.5x improvement here, improving absolute runtime by 2.1x. This indicates that when the frontier of a specific IFE subroutine gets sparse and Kuzu-nT1S starts keeping threads idle, Kuzu-nTkS can keep those threads active on other concurrently running IFE subroutines. This behavior is also reflected by the higher CPU utilization numbers for Kuzu-nTkS than other policies. We observe similar patterns on the queries that return paths, where Kuzu-nT1S achieves between 8.5x-14.1x improvement factors. In contrast, Kuzu-nTkS achieves more robust improvements factors of between 12.4x to 14.7x. Spotify is an outlier here, where the runtime and CPU utilization gap between nTkS and nT1S policy is not significant. This is because Spotify has a very high average degree, which leads to denser frontiers. Therefore, even if a single source's IFE computation is active at any point, there is enough work to keep all threads busy and we do not need to launch multiple concurrent IFE computations. As a result, nT1S scales better on Spotify than other datasets.



(a) nTkS experiments.



(b) nTkMS experiments.

Figure 9: (a) Kuzu-nTkS and (b) Kuzu-nTkMS performances under varying k. The values on top of the bars are k values. Improvement factors are over using k=1.

5.4 64-source Workloads

Our next set of experiments evaluates the behavior of nTkS and our baselines when the query has more sources than total threads in the system. We now expect nTkS to mimic the behavior of systems that implement the 1T1S policy, but we can also expect it to beat its performance. The reason for this is that in the last phases of the computation for 1T1S, once number of available sources go below 32, 1T1S policies start to keep some threads idle. Instead in nTkS, those threads start helping other IFE subroutines that are continuing to execute. Finally, we expect the nT1S policies to behave similar to their previous behaviors.

Tables 3c and 4c show our results for all of our datasets. As expected, Ligra and Kuzu-nT1S continue behaving similar to the previous experiments. Observe that we no longer see Neo4j and Kuzu-1T1S policies flattening as there are enough source morsels to keep threads busy. Neo4j now achieves between 9.5x to 13.2x improvements while Kuzu-1T1S achieves between 7.8x to 11.9x improvements. Importantly, except on the Spotify dataset, Kuzu-1T1S now consistently outperforms Kuzu-nT1S and achieves better scalability numbers.

Kuzu-nTkS achieves between 11.5x and 15.5x improvements. Duck-nTkS similarly achieves between 11.1x and 15.3x improvements (again except on LDBC). Observe that Kuzu-nTkS is consistently competitive with or outperform Kuzu-1T1S. This is because once the number of active IFE subroutines reduces below the number of threads, Kuzu-1T1S starts keeping threads idle. This is why it consistently achieves lower CPU utilization numbers than Kuzu-nTkS. Instead, Kuzu-nTkS keeps those threads busy by dispatching them work from IFE subroutines that have not yet finished.

In summary, our experiments so far demonstrate that the nTkS *policy is a robust approach to obtaining good scalability across query workloads that contain few source nodes to those with more source nodes than the number of threads.* This is because although nTkS implements a mechanism to dispatch work from the frontiers of each IFE subroutine, whenever frontiers get sparse and there is contention, it moves threads to other frontiers from which more work can be given. Therefore, we recommend it to system developers as a robust morsel dispatching policy.

5.5 Effects of the Choice of k

Recall that the parameter k in the nTkS policy determines the number of concurrent sources that will be dispatched to threads. So far in our experiments, we set this value to 32, which is the maximum number of threads we use in our experiments. This value ensures that threads work on separate sources when possible and only work together on the same source when the total active sources is constrained through k or if there are no new sources left to launch. Our next set of experiments study the effects of the choice of k on the performance of nTkS. We use 32 threads and use our 64-sources workload that return path lengths and vary k from 1 to 32. Figure 9a shows the improvement factors we get as we increase k over using k=1. We see that increasing k generally improves performance by up to 3x on our datasets, except in Spotify, where we see a 1.15x performance loss.

To better understand this differing behavior, we analyzed the CPU metrics using Linux perf tool on the Spotify experiments. First, no matter what the value of k is, the CPU utilization is high across all Spotify experiments. CPU utilization is a good metric for how

well the computation is parallelizing, e.g., is there contention on locks, but not necessarily whether the CPUs are doing actual work or stalled for memory accesses. Instead, the runtime behavior nTkS policy on Spotify is related to the CPU cache hit rate. Table 5 shows the LLC-Load throughputs for each experiment from Figure 9. LLC-Load is the number of memory requests that was served from the L3 cache. Observe that except for Spotify, LLC-Load increases in each dataset as k increases, which correlates with the runtime pattern in Figure 9. In contrast for Spotify, we see a decrease after $k = 4$ (from 50.1M to 38.2M).

We hypothesized that this behavior must be driven by the average degree in these graphs. Spotify has an average degree of 535. This is much larger than other datasets, which have an average degree of at most 44. First, this explains why the LLC throughput is significantly larger in Spotify (between 38-50M loads/second) compared to other datasets (at most 24M loads/second). That is, the IFE computation performs a lot more memory accesses in general. More importantly, a high average degree implies that on average, nodes will have many common neighbors. Therefore, threads that are extending the frontiers of the same source morsel access the common locations in the same visited array structure to check if a neighbor is already visited or not. To verify this, we provide additional metrics in the longer version of this paper [7] that directly measures the average number of times each node is accessed in the visited array across experiments on different datasets. This leads to a higher L3 cache hits. However, as k increases, this cache locality decreases, since threads start working on different auxiliary data structures and access different visited arrays.

To verify that increasing k can lead to lower cache locality as graphs get denser, we performed a further controlled experiment. We generated a set of random Erdős-Renyi graphs, each with 5 million nodes, and increasing average degrees from 25 to 500. Then we repeated our experiment studying the effect of k. Figure 10 shows our results. Starting with an average degree of 100 (500M edges), increasing k can start degrading performance. Further, the denser the graph, the lower the value of k at which further increases in k start degrading the performance. Specifically at average degrees 100 (500M edges), 250 (1.25B edges), and 500 (2.5B edges), performance degrades at $k = 16$, $k = 8$, and $k = 4$, respectively. *In summary, the optimal choice of k depends on how dense the input graph is. On the one hand, increasing k increases the total parallelism we can extract from the* nTkS *policy and improves performance. At the same time, as the average degrees of input graphs increase, this gain can be offset by a loss of CPU cache locality and in balance degrade performance. Systems that use the* nTkS *policy can use the average degrees as a proxy to select an appropriate k value.*

	k →	1	2	4	8	16	32
LDBC	Time	4.1	3.3	2.3	1.5	1.3	1.2
	LLC Tp	10.9	11.4	13.9	19.4	23.6	23.9
	CPU %	66	78	85	88	95	98
LJ	Time	37.5	31.2	22.6	13.5	10.3	9.7
	LLC Tp	6.2	6.5	7.2	9.5	10.7	10.9
	CPU %	87	90	92	93	96	98
Sp	Time	82.8	71.8	68.7	73.0	82.8	95.6
	LLC Tp	40.4	48.5	50.1	48.6	43.1	38.2
	CPU %	94	96	98	97	95	91
G-28	Time	938.9	766.0	640.0	492.9	449.9	432.0
	LLC Tp	12.7	15.1	17.2	21.2	23.0	24.0
	CPU %	70	80	87	92	95	96

Table 5: Runtime (s) vs LLC Throughput (Tp, Million/s). 64-src workload

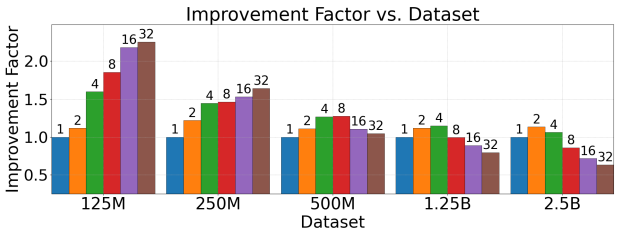


Figure 10: Varying k for random Erdős-Renyi graphs.

5.6 Multi Source Morsel Optimization

Our final set of experiments evaluate the performance benefits of nTkMS policy. Recall that MS-BFS has the benefit of reducing the amount of scans by sharing scans across multiple IFE subroutines. At the same time, it has some overheads. Specifically, when a node u is visited in an IFE, i.e., put into a frontier, its 64-bit visited status changes. We need an additional loop to find out all the frontiers that u is part of to update the other data structures. We first evaluate this optimization on query workloads that contain 1 to 256 sources using Kuzu-nTkMS configuration with $k = 4$. As a baseline, we use the standard Kuzu-nTkS configuration with $k = 32$.

Figure 11 shows the performance improvements of Kuzu-nTkMS over Kuzu-nTkS as we increase the number of sources in the queries. The solid and dashed lines represent the results when using query workloads that return lengths and paths, respectively. First, observe that Kuzu-nTkMS is often slower than Kuzu-nTkS when using fewer than 32 sources. Unless enough sources are available for a single multi-source morsel, the overhead of Kuzu-nTkMS is higher than its benefits. When we saturate the "lanes", using the optimization improves performance, 1.4x-4.4x across the different datasets and query workloads.

Observe that starting with 128 sources, experiments that return paths run out of memory on Graph500-28. Recall from Section 4.2 that we pre-allocate 536 bytes per node, per multi-source morsels. Therefore, the pre-allocated memory requirements for 2 multi-source morsels for Graph500-28, which has ~120 million nodes, is 128 GB. With the space needed to store the actual paths, this exceeds the memory capacity of our machine. Returning path lengths requires only 21 GB allocation for 2 multi-source morsels and the size of the memory footprint does not increase during computation.

In our next and final set of experiments, we repeat our experiment from Section 5.5 and study the effect of k on Kuzu-nTkMS. We use the 256-source workload returning path lengths and vary k. Figure 9b shows that as long as the system is not running out of resources, a larger k improves performance. For Kuzu-nTkMS

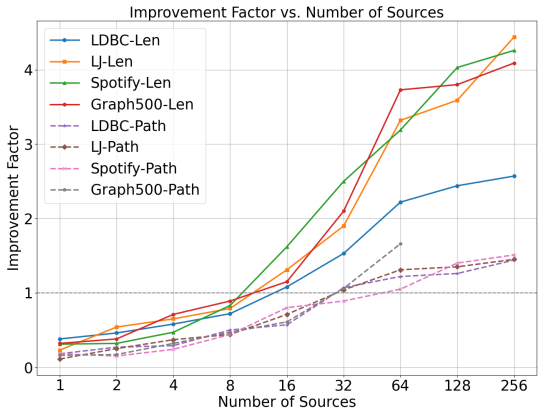


Figure 11: nTkMS (k = 4) improvement over nTkS (k = 32).

the improvement factors are less compared to Kuzu-nTkS policy because as long as there are more that 64 sources in the queries, even when $k = 1$, a single multi-source morsel keeps threads busy. An advantage of packing multiple sources together is that frontiers generally remain denser because they represent the union of 64 IFE computations running at the same time. Finally, similar to our previous experiments we see that Kuzu-nTkMS benefits least on our Spotify graph, which has the highest average degree, on which the CPU utilization is consistently already very high for $k = 1$.

Prior work [35] has provided experiments for MS-BFS only on queries that return lengths and only on workloads that saturate at least one multi-source morsel. Our conclusions for MS-BFS are more nuanced. *Specifically, we observe important runtime benefits only when the number of sources are large enough to saturate many of the lanes in a multi-source morsel. Further, on queries that return paths, the memory footprint of computing the paths for a large number of sources can be prohibitively large.* Systems implementing this optimizations should decide when to trigger this optimization based on the return type and the number of source nodes in the queries.

6 RELATED WORK

Morsel-driven parallelism is related to the seminal Volcano system's [13, 14] parallelism model. Graefe has introduced several parallelism techniques in the context of the Volcano system. Importantly, he introduces *horizontal parallelism*, which refers to two separate approaches. First is *bushy parallelism*, which runs different subplans that do not depend on each other independently on separate threads at the same time. This is a form of *task-level parallelism*. Second is *intra-operator parallelism*, which splits the input of an operator into multiple partitions. This is a form of *data-level parallelism*. Morsel-driven parallelism a form of data parallelism that extends Volcano's intra-operator parallelism, where the inputs are partitioned in a much more fine-grained manner than in the Volcano system (always at the leaves of the subplans/tasks). While many modern systems adopt data-level parallelism, such as DBMSs adopting morsel-driven parallelism, Spark [37], and Timely Dataflow [24], there are also systems that adopt hybrid task- and data-level parallelism approaches [5].

Our design space includes the vertex-centric parallelism approach of parallel or distributed graph analytics systems. Systems such as Pregel [23], Ligra [31], GraphChi [19], GraphLab [21] have been based on this model of parallelism. These systems have vertex- or edge-centric APIs, such as Ligra's `edgeCompute()` function, that assume an IFE-based computation. Programmers implement these functions to describe the computation that should be done per vertex or edge. Then, the system automatically parallelizes the executions of these functions using vertex-centric parallelism that parallelize work at each frontier. Our implementation uses Ligra's API to implement parallel versions of recursive algorithms, including recursive path finding algorithms, such as transitive closure, shortest paths [19, 23, 27, 31]. Implementing recursive algorithms in a DBMS with a vertex-centric abstraction however, deviates from the tuple-based abstraction that is used in other physical operators in a DBMS. Another parallelism approach is to use OpenMP [28] or a similar multi-threading library to automatically parallelize loops using their own thread pools. These threading libraries are adopted in several in-memory graph analytics libraries, such as graph-tool [29], which support a suite of batch graph algorithms. IFE-based graph algorithms that can be modeled as sparse matrix multiplication-based computations are also studied extensively in the high-performance computing literature. There is extensive work in this literature on communication-avoiding algorithms that aim to minimize the communication between a set of distributed/parallel compute nodes by advanced data partitioning approaches [8, 33]. For example, in 2D partitioning approaches, a matrix M, each row of which represents an adjacency lists, is partitioned into submatrices whose dimensions have size roughly the square root of M's dimensions. Our work differs from this literature in that we assume an IFE algorithm executing inside a DBMS that has a standard data layout in which the threads can scan the entire adjacency lists of each node.

7 CONCLUSIONS

In this paper we studied several approaches in prior literature to parallelize recursive path-finding queries in GDBMSs: (i) vanilla morsel-driven parallelism of GDBMSs; (ii) frontier-level parallelism of graph analytics libraries; and (iii) the multi-source morsel optimization. We first showed that these approaches can all be integrated to into DBMSs that adopt morsel-driven parallelism as different morsel dispatching policies that assign source (or multi-source) morsels and frontier-morsels to threads. We then extended them with hybrid policies, which we call nTkS and nTkMS policies that dispatch both source and frontier morsels. We then experimentally evaluated and analyzed their pros and cons on a variety of query workloads, especially recommending the nTkS policy as a robust policy to parallelize recursive queries. Our work focused on queries using walk semantics of paths, which allows multiple nodes and edges to be repeated. One important venue for future work is to study queries under the two other path semantics in modern graph query languages, which are trail and acyclic. These semantics respectively limit edges and nodes from being repeated in the paths. It is important to study the optimizations that can be applied to directly compute these semantics, especially under parallel execution.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Foundations and Trends. https://doi.org/10.1561/1900000024

[2] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*.

[3] Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. 2022. Optimizing differentially-maintained recursive queries on dynamic graphs. *PVLDB* 15, 11 (2022).

[4] Khaled Ammar, Siddhartha Sahu, Semih Salihoğlu, and M. Tamer Özsu. 2022. Optimizing Differentially-maintained Recursive Queries on Dynamic Graphs. *PVLDB* 15, 11 (2022).

[5] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid parallelization strategies for large-scale machine learning in SystemML. *PVLDB* 7, 7 (2014).

[6] Anurag Chakraborty. 2025. Paper Main Github repo. https://github.com/anuchak/kuzu.

[7] Anurag Chakraborty and Semih Salihoğlu. [n.d.]. Technical Report: Robust Recursive Query Parallelism in Graph Database Management Systems. https://github.com/anuchak/kuzu/blob/master/recursive_joins_vldb_revision.pdf.

[8] Jim Demmel. 2012. Communication Avoiding Algorithms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*.

[9] DuckDB. 2025. Parallelism (Multi-Core Processing). https://duckdb.org/docs/stable/guides/performance/how_to_tune_workloads.html#parallelism-multi-core-processing.

[10] DuckPGQ. 2025. Modified DuckPGQ Fork. https://github.com/anuchak/duckpgq-fork.

[11] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. Kùzu Graph Database Management System. In *Conference on Innovative Data Systems Research (CIDR)*.

[12] Inc. Free Software Foundation. 2025. GCC Builtins. https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/Other-Builtins.html.

[13] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*.

[14] Goetz Graefe. 1994. Volcano-An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994).

[15] Neo4j Inc. 2025. Neo4j Github repo. https://github.com/neo4j/neo4j.

[16] Neo4j Inc. 2025. Neo4j Graph Data Science Library Github repo. https://neo4j.com/docs/graph-data-science/current/introduction/.

[17] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*.

[18] Raunak Kumar, Paul Liu, Moses Charikar, and Austin R. Benson. 2020. Retrieving Top Weighted Triangles in Graphs. In *WSDM*.

[19] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *OSDI*.

[20] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.

[21] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB* 5, 8 (2012).

[22] Memgraph Ltd. 2025. Memgraph Main Github repo. https://github.com/memgraph/memgraph.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*.

[24] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, Iterative Data Processing With Timely Dataflow. *CACM* 59, 10 (2016).

[25] Inc. Neo4j. 2024. OpenCypher. https://opencypher.org/.

[26] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*.

[27] Maurizio Nolé and Carlo Sartiani. 2014. Processing Regular Path Queries on Giraph. In *CEUR Workshop*.

[28] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf

[29] Tiago P. Peixoto. 2025. graph-tool Python library Github repo. https://graph-tool.skewed.de/.

[30] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD 2019*.

[31] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. *SIGPLAN Not.* 48, 8 (2013).

[32] Stanford Network Analysis Project (SNAP). 2024. LiveJournal social network. https://snap.stanford.edu/data/soc-LiveJournal1.html.

[33] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par*.

[34] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *PVLDB* 16, 4 (2022).

[35] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. Property Graph Queries (SQL/PGQ). *Conference on Innovative Data Systems Research (CIDR)* (2023).

[36] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The More the Merrier: Efficient Multi-source Graph Traversal. *PVLDB* 8, 4 (2014).

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*.