Robust Recursive Query Parallelization in Graph Database Management Systems

by

Anurag Chakraborty

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Anurag Chakraborty 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Recursive joins such as shortest path and variable length path queries are a core feature set of modern graph database management systems (GDBMS). Since these queries tend to be computationally expensive and may suffer from high execution time, they require efficient parallel processing using multiple cores to achieve good performance. Existing work on parallel query processing includes the morsel driven parallelism approach that distributes a unit of work (denoted as "morsel") to threads for parallel execution. We revisit this technique in the context of parallelization of recursive joins in GDBMS and discuss how the traditional approach of morsel driven query execution is inadequate to tackle recursive join queries. We show how this approach can be modified to better accommodate scalable parallelization of recursive joins. We further describe how this modified parallel query execution approach has been integrated into Kùzu, an embedded disk based columnar GDBMS. Compared to vanilla morsel driven parallelism, our modified parallel query execution approach can be orders of magnitude faster and scales well on multiple cores.

Acknowledgements

First, I want to thank my supervisor, Prof. Semih Salihoğlu, for his constant support in the past two years. This thesis would not be possible without his advice and constant feedback on my progress.
Second I would like to thank the Kùzu team, Xiyang Feng, Guodong Jin, Ziyi Chen and Chang Liu for helping me understand the internals of Kùzu and explaining to me countless times how Kùzu's query processing engine works. I am very grateful to the team for teaching me so much about database internals.
I also want to thank my thesis readers, Prof. Tamer Özsu and Prof. Ken Salem, for spending time to read my thesis.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables viii

1 Introduction 1

2 Background 6

2.1 Morsel-Driven Parallelism 6

2.2 Iterative Frontier Extensions (IFE) 9

3 System Overview 15

3.1 Query Plan Overview and `IFECall` Operator 15

3.2 IFEAlgorithm Interfaces 18

3.3 A Design Space for Scheduling Policies 22

3.3.1 1T1S Policy 22

3.3.2 nT1S Policy 23

3.3.3 nTkS Policy . . . 25
3.4 Further Optimizations and Implementation Details . . . 27
3.4.1 In-Memory Graph Index . . . 27
3.4.2 Sparse & Dense Frontier Representation . . . 27
3.4.3 Lock-Free Implementation . . . 29

4 Evaluation **32**
4.1 Experimental Setup . . . 32
4.1.1 Hardware . . . 32
4.1.2 Datasets . . . 33
4.1.3 Query workload . . . 33
4.2 Single Source Queries Workload . . . 34
4.3 Multi-source Parallelism (few sources) . . . 36
4.4 Multi-source Parallelism (many sources) . . . 38
4.5 In-Memory Graph Index Experiments . . . 41

5 Related Work **47**

6 Conclusions **49**

References **51**

List of Figures

1.1 A standard plan with a recursive operator in Graph DBMSs. 3

2.1 Hash Join Query Plan 7
2.2 Morsel-driven Parallelism - Hash Join Build Plan 8
2.3 Morsel-driven Parallelism - Hash Join Probe Plan 9

3.1 Query Plan with `IFECall` operator. 16
3.2 *1T1S* scheduling 24
3.3 *nT1S* scheduling 26
3.4 *nTkS* scheduling 26
3.5 Query Plan with `CSRIndexBuild` operator 29
3.6 In-Memory CSR Index 30

4.1 Single Source: On-Disk vs In-Memory Runtime 43
4.2 Single Source: On-Disk vs In-Memory Runtime 43
4.3 Multi-Source (few): Disk-based vs In-Memory Runtime 44
4.4 Multi-Source (few): On-Disk vs In-Memory Runtime 44
4.5 Multi-Source (many): On-Disk vs In-Memory Runtime 45
4.6 Multi-Source (many): On-Disk vs In-Memory Runtime 45

List of Tables

4.1 Datasets. 33
4.2 Single Source Parallelism: Disk-based Graph (in **milliseconds**) 35
4.3 Scalability of each frontier computation (disk-based graph). Times are in ms. 36
4.4 Multi-source (few): Scheduling Policies (On-Disk, in **milliseconds**) 37
4.5 Multi-source (many): Scheduling Policies (On-Disk, in **milliseconds**) . . . 38
4.6 CPU Cache Statistics: `nT1S` vs `nTkS` Policy 39
4.7 nTkS Policy: varying $k = 2, 4, 8, 16$ (runtime in **milliseconds**) 40
4.8 CPU Cache Statistics: `nTkS` Policy (varying k) 40
4.9 CSR Index Build Time (in **milliseconds)** 42
4.10 CPU Cycles % for Single Source (disk-based graph) 46

Chapter 1

Introduction

Modern graph database management systems (GDBMS) follow the property graph data model, which supports modeling application records in the form of a graph. Nodes represent entities and edges represent the relationships between entities. There can be different types of records, grouped under separate node or edge labels. Key-value properties can be added for both the nodes and edges in the graph. 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 [23] of the Neo4j system, dialects of which are adopted by several other GDBMSs has the arrow-based syntax to specify paths, the Kleene Star ("*") syntax to ask transitive closure of nodes, which can be extended to ask variable-length join queries ("*min..max"), a limited form of regular path queries ("$\text{label}_1|...|\text{label}_k$*"), and in some dialects the "* SHORTEST" or "* ALL SHORTEST" clauses to find the shortest paths between nodes. The inherent support for paths and such specialized clauses make it very easy to ask queries of a recursive nature, that would be harder to ask in SQL's "WITH RECURSIVE" clause. In this thesis, we refer to queries that require such recursive evaluation by the system as *recursive queries* and the specialized query language clauses in GDBMSs as *recursive clauses*.

GDBMSs, just like other classes of DBMSs that implement high-level query languages, adhere to the common architectural principles of relational data management where high-level query clauses are compiled down to query plans that consist of relational operators. The common wisdom approach to parallelizing queries in traditional DBMS architectures is *morsel-driven parallelism* [17]. Morsel-driven parallelism breaks a query plan into one or more pipelines, and schedules fine-grained execution of each pipeline by assigning small fragments of input (called "morsels") to threads. Each pipeline starts with a source scan operator and ends with a sink operator that materializes intermediate results of the pipeline.

The database maintains a pool of *worker threads* that work on copies of the same pipeline concurrently. Worker threads steal morsels of input from the source scan operators and independently push them through the pipeline. If needed, threads synchronize at the final sink operator, for example to merge their results. Once a pipeline has completed execution (when there are no more morsels to assign to worker threads), the database query scheduler proceeds to submit the next pipeline of the query for execution. In this fashion, parallel execution of queries is achieved using all cores present on a machine. This framework has been implemented across both RDBMSs, such as Hyper [13], Umbra [24], and DuckDB [28] as well as GDBMSs, such as Kùzu [8] and Neo4j [12]. The central research question of this thesis is this: *How should a GDBMS parallelize recursive computations that adopt morsel-driven parallelism?*

To explain the core parallelization problem that motivates this thesis, consider the following transitive closure-like query in Cypher. The query finds all the paths that consist of Knows edges from each Person node with name **Alice** to every other Person node, and returns their distinct names (up to the maximum recursive depth in the system):

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

An example standard plan corresponding to this query is shown in Figure 1.1. Variants of this plan are generated by many systems, such as Neo4j, Kùzu, or DuckPGQ. At a high level, a scan operator finds the set of tuples that match all Person records with the name **Alice**. Let us call these tuples `RecInput`. Then `RecInput` is given to a specialized operator that implements the transitive closure computation for each tuple in `RecInput`. Similar plans are generated for queries that involve other recursive clauses, such as shortest paths queries. `RecInput` is the source table that stores the sets of source (and in some queries also destination) pairs between which a recursive computation is performed. In vanilla morsel-driven parallelism, depending on how the query is broken into pipelines, either the source Scan Node (Person) operator or `RecInput` table would be split into morsels to parallelize work. If there are not enough morsels available for all threads, morsel-driven parallelization can end up assigning morsels to a few threads while keeping other threads idle. An example of this scenario arises when a query computes the transitive closure from a single source.

However, a recursive query from even a single source can be quite expensive and amenable to parallelization. This is due to the fact that real-world graph databases tend to be heavily connected [38]. As such, a few depths of recursive explorations from a single node may end up traversing the whole database and a vanilla morsel-driven scheduling can force the entire query to be executed in a single-threaded manner. In addition, queries

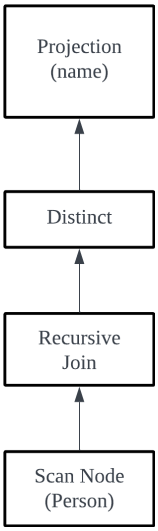


Figure 1.1: A standard plan with a recursive operator in Graph DBMSs.

can involve different numbers of sources and destination pairs. *The primary goal of this thesis is to propose a query processing design that can robustly parallelize recursive queries in GDBMSs, from those that include single to many sources (or source-destination pairs).*

Our approach is based on adopting the parallelization approach adopted in parallel graph analytics systems, such as Pregel [21] and Ligra [32], where recursive graph algorithms, such as the ones we study here as well as PageRank or graph connectivity algorithms, are implemented as a *iterative frontier extension* (IFE) subroutine, which is a breadth-first search-like computation. In this subroutine, an initial frontier of "active nodes", whose neighbors are iteratively explored using a user-defined function, such as `vertex.compute()` or `edgeCompute()`, which describes the work to do per vertex or explored edge, and form new frontiers. These iterations continue until a convergence criterion is met. These systems parallelize work *at the frontier level.* That is each frontier of a recursive computation is broken into morsels and parallelized across workers. At the same time, the programming abstractions of parallel graph analytics systems are suitable for computing recursive queries from a single source instead of multiple sources concurrently, which is handled by running a new recursive computation from each source one at a time. Running IFEs from a single source at any point in time can also limit parallelism. This is because, during an IFE, there are many sparse frontiers where there may not be enough morsels, keeping many threads idle. That is, in some recursive computations from a single source, a significant fraction of the total work to be done may not be parallelizable.

In this work, we propose a hybrid parallelization approach of vanilla morsel-driven and frontier parallelism. In our design, the recursive operators in the system belong to pipelines that are executed in a single-threaded manner. The operators implement the control flow of the recursive graph algorithm in an imperative manner using a graph and frontier-based abstraction. This contrasts with the rest of the operators in the system that pull tuples from their children operators, e.g., using a 'getNextTuples' function. The operators have access to the worker thread pool of the DBMS to generate new concurrent *IFE pipelines* in the middle of execution. IFE pipelines encapsulate computations to be performed from a single source. However, multiple worker threads can work on the same IFE pipeline and parallelize work by grabbing morsels from the frontiers of the IFE computation, akin to how graph analytics systems execute in parallel.

We demonstrate that our implementation is robust under recursive queries that contain between one to many sources to perform a recursive computation from. We compare our design to approaches that require single-source computations to be performed by one thread, as well as those that require all threads to work on a single source computation, and demonstrate the robustness of our approach. We present experiments that study the number of concurrent pipelines to issue to maximize the benefits of parallelism.

One aspect of our recursive operator implementations that may be of independent interest to users is that they work against a graph and frontier-based abstraction, similar to systems like Pregel and Ligra, instead of the rest of the operators in the system that adopt a tuple-based abstraction. This simplifies the design of the recursive operators as they can work seamlessly against an in-memory graph index that can be built on the fly, as well as one that actually scans the edge records from disk. We present experiments studying our design's scalability when creating an on-the-fly in-memory index vs when edges are scanned from disk through the buffer manager. We also answer the question of when building an in-memory graph index leads to faster end-to-end query performance.

The outline for the rest of the thesis is as follows:

- Chapter 2 provides some background on IFE-based recursive algorithms and morsel-driven scheduling implemented commonly in analytical databases for parallel query execution.
- Chapter 3 describes our query processor design, covering our query plans, operators, pipelines, and a design space of parallelization approaches based on modified morsel-driven scheduling policies. We also cover the optimizations to our overall design, such as how we build an in-memory graph index.

- Chapter 4 presents experiments on how well we parallelize recursive queries on single-source, few-source, and multiple-source queries. We also present results for the in-memory graph version of recursive queries, the cases in which it performs better than the on-disk graph query.
- Chapters 5 and 6 cover related work and conclusion.

Chapter 2

Background

In this chapter, we first provide a more detailed background and an example of morsel-driven parallelism. We then provide background on the *iterative frontier extensions (IFE)* [32, 3] algorithmic sub-routine. IFE forms the core of many commonly used algorithms in practice to evaluate recursive queries. We also base our work on parallelizing IFE-based algorithms.

2.1 Morsel-Driven Parallelism

Morsel-driven parallelism [17] is the state-of-the art approach to parallelizing queries in DBMSs. Morsel-driven parallelism aims to make scheduling a fine-grained runtime task by dividing input data into smaller units of work termed "morsels" and distributing these units to threads for query execution. Consider the following SQL query:

```
SELECT a.name FROM Person a, Location b
WHERE b.name = 'Waterloo' AND  a.staysIn = b.locationID;
```

This is a simple join query that returns the names of all people who stay in the location 'Waterloo'. The equivalent query expressed in Cypher is as follows:

```
MATCH (a:Person), (b:Location)
WHERE b.name = 'Waterloo' AND a.staysIn = b.locationID
RETURN a.name;
```

Modern analytical database systems execute these queries by first breaking them down into smaller portions or tasks called "pipelines". Figure 2.1 shows a simple query plan for

this query that performs the join using a hash join algorithm. In the figure two pipelines are identified: pipeline $P1$ that performs hash join build for the Location table and pipeline $P2$ that performs hash join probe using tuples from the Person table using the hash table computed $P1$.

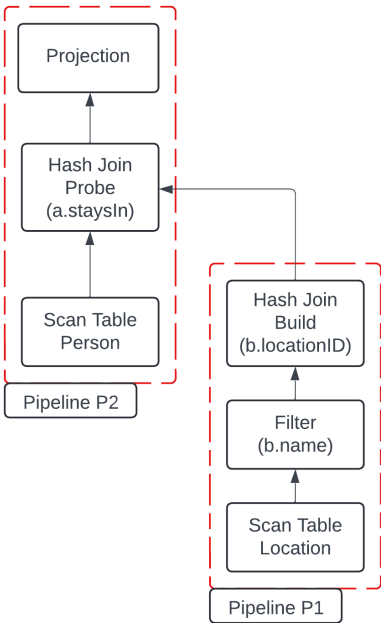


Figure 2.1: Hash Join Query Plan

Since pipeline $P2$ (Figure 2.2) is dependent on pipeline $P1$ (Figure 2.3), the DBMS schedules $P1$ first. Once $P1$ completes, pipeline $P2$ is scheduled. Each pipeline starts with a 'source' operator and ends with a 'sink' operator. The source operator is responsible for scanning the input to the pipeline, and the sink operator is usually responsible for materializing any intermediate result or the final results for the query. The input for each pipeline is a base or an intermediate table to be scanned by the source operator. We refer to this table as the *source table* of the pipeline. It is this input that is distributed to multiple threads for parallel execution of queries.

The logic of controlling and assigning morsels to processing units is decided by a piece of code termed *morsel dispatcher* [17]. The morsel dispatcher uses the worker thread pool to schedule pipeline execution. During execution, each worker thread W_i assigned to a pipeline creates a copy of the pipeline, and using the morsel dispatcher scans a morsel of tuples, e.g., 100K many tuples, from the source table. These tuples get processed by the

rest of the operators in the pipeline until W_i needs to grab another morsel of tuples.

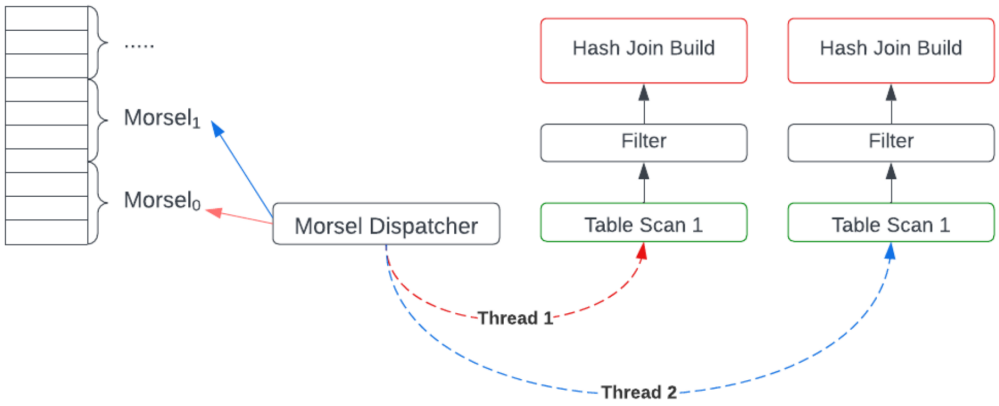


Figure 2.2: Morsel-driven Parallelism - Hash Join Build Plan

Let us go back to our example. Figure 2.2 visually shows a possible execution for pipeline $P1$ under two worker threads. The morsel dispatcher assigns $\texttt{Morsel}_0$ and $\texttt{Morsel}_1$ to $\texttt{Thread}_1$ and $\texttt{Thread}_2$ respectively. The processing of morsels in the rest of the operators happens without any synchronization except at the sink, i.e., the last operator of the pipeline, where some synchronization may be needed. For example in some hash join implementations, each worker thread may build a local hash table out of the morsels they have scanned and then synchronize to prepare a global hash table. So in Figure 2.2, the threads can execute the Filter operator independently but may synchronize to merge their local hash tables to the global hash table. When a thread finishes execution on a morsel and reaches the pipeline end, it can go back to the morsel dispatcher to receive another morsel until all morsels are exhausted.

Once this pipeline is finished, pipeline $P2$ in the waiting queue is picked up by the scheduler for execution. Figure 2.3 shows the subsequent hash join probe plan that will execute after the build phase is completed. The hash table materialized from the previous pipeline is passed to this pipeline as an input. Then, similar to the execution of $P1$, this task also involves distributing morsels of tuples among waiting threads for the probing operation.

For this parallelism paradigm to work well, morsels need to be picked small enough so any load imbalance is limited by the time a thread takes to process a morsel. In addition, morsels need to be large enough to avoid any contention at the morsel dispatching code

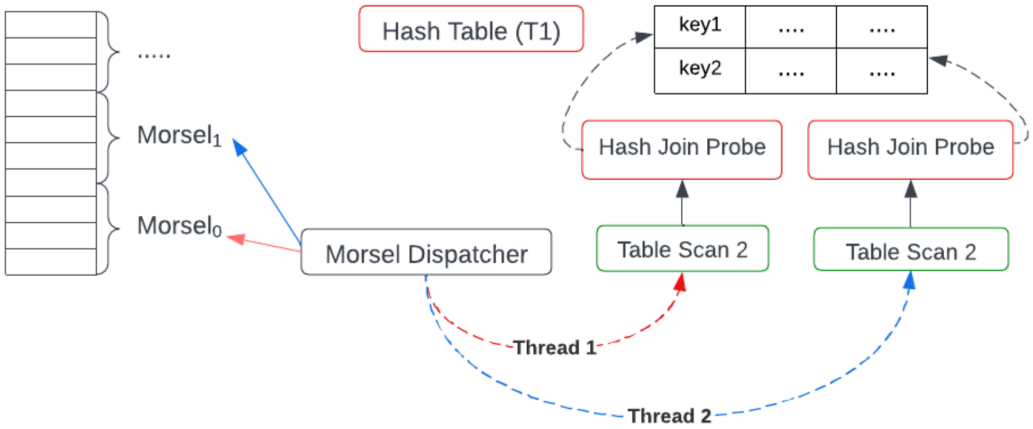


Figure 2.3: Morsel-driven Parallelism - Hash Join Probe Plan

which needs synchronization. The original paper [17] suggests setting a morsel size of 100K tuples, though we pick a smaller size for reasons we explain in Chapter 3.

2.2 Iterative Frontier Extensions (IFE)

We next review the iterative frontier extensions sub-routine that we adopt in this work to evaluate recursive queries. Algorithms based on IFE are adopted by many DBMSs Neo4j, Memgraph, or DuckPGQ, and graph analytics systems, such as Ligra, to evaluate recursive queries of graph algorithms. We also adopt IFE-based recursive algorithms in this work.

IFE as a high-level algorithmic sub-routine is an iterative BFS-like computation that starts from an initial *frontier*, which is a set of nodes whose neighborhoods should be explored to perform a computation. In many cases, such as when finding shortest paths or doing a BFS traversal from a single node, the initial frontier can contain a single node. Then, in iterations, the neighbors of each vertex in the current frontier are explored to construct a new frontier, until a convergence criterion is met.

Algorithm 2.1 shows the pseudocode of the high-level IFE computation. The pseudocode assumes that there is an `ife` object that has several functions and fields. In particular, there are `currentFrontier` and `nextFrontier` fields and an `edgeCompute()` function that performs a computation per each (outgoing) edge (u, v) of an "active" vertex u in the current frontier. In this pseudocode, if `edgeCompute()` returns true, then the

```
curFrontier = ife.initializeFirstFrontier()
while (!ife.complete()) {
    for (u : ife.curFrontier) {
        for (e : u.edges()) {
            if e.edgeCompute(e) {
                nextFrontier.add(v)
            }
        }
    }
    ife.swapFrontiers()
    ife.initializeNextFrontier()
}
```

Listing 2.1: Pseudo-code: Iterative Frontier Extension

neighbor v is put into the next frontier. The `complete()` function checks the convergence criteria, which for many algorithms is the next iteration's frontier becoming empty.

Different IFE recursive computations can have different `edgeCompute()` functions and termination conditions. Depending on the computation, IFE computations can further keep additional state, such as a value per vertex to indicate at which iteration each vertex has been visited or an edge per vertex to keep track of its BFS parent. We give two examples.

Example 1 Consider the Cypher query below that finds the unweighted shortest path from a particular source to all other Person nodes.

```
MATCH p = (a:Person)-[r:Knows* SHORTEST]->(b:Person)
WHERE a.ID = 1 RETURN b, length(p)
```

It returns the shortest path between a source node and a destination node and returns the destination and path length. We are omitting the code for returning actual paths for simplicity. Algorithm 2.2 shows the pseudo-code for the IFE computation of this query encapsulated in an SSSDSP class, for **s**ingle **s**ource **s**ingle **d**estination **IFE**. We omit common fields such as `currentFrontier` and `nextFrontier`. The pseudocode keeps two auxiliary data structures to keep track of the overall visited states and found distances to each vertex. We omit the details of these data structures. The termination condition, encapsulated in complete() function is that either the destination node is reached or the next frontier is empty. The `edgeCompute()` function is simply as follows. If the neighbor v of an active vertex u is not yet "visited", then the function sets the distance of v to the

```
class SSSDSP {
    visitedState = [0,0,...0] // 0 signifies unvisited
    pathLength = [-1, -1, ... -1] // all lengths initialized to -1

    bool edgeCompute (e=(u, v)) {
        state = visitedState[v]
        if (state == 0) {
            visitedState[v] = 1
            pathLength[v] = curIteration + 1
            return true
        }
        return false
    }

    bool complete() {
        return visitedState[dst] != 0 || curFrontier.empty()
    }

}
```

Listing 2.2: Pseudo-code: Unweighted Shortest Path

current iteration of the IFE, which we assume is available in the `curIter` field and returns true, so that v is put into the next frontier. Otherwise, it does not do anything and returns false. If the query needed to return the actual shortest paths, `edgeCompute()` would be more detailed and have additional logic to keep track of the paths.

Another example of a recursive query based on IFE computation is the transitive closure or Kleene star ('*') query, expressed in Cypher as follows:

```
MATCH p = (a:Person)-[r:Knows* 1..10]->(b:Person)
WHERE a.ID = 1 RETURN b, length(p)
```

The Kleene star query computes all paths (not just the shortest) from a source to a set of destination nodes. Algorithm 2.3 shows the pseudo-code for a kleene-star query from a source node with an upper bound on path length, which would be set to 10 in the above query. The actual code that tracks not just the path length but also the path is more complicated and has been omitted from the pseudocode shown in 2.3. We have to track the 'multiplicity' of a node that is visited at a particular level, along with all the levels at which the node is visited (until the upper bound). The multiplicity of a node is the number of times a node is visited at a particular path length. The termination condition (`complete()`) is the `curIteration` reaching the upper bound or if the `currentFrontier`

is empty.

The core problem we address in this paper is how to parallelize this IFE subroutine efficiently when it is implemented in an actual DBMS query processor. In a vanilla implementation, a DBMS adopting morsel-driven parallelism will compile the IFE subroutine into a pipeline with base scans from a table that provides tuples that identify the sources and/or destination nodes between which a recursive computation should run. Since morsels are provided from this source table, this can limit work sharing when working on a single IFE computation. For instance, distributing morsels from a single source's recursive computation and managing multiple recursive computations simultaneously would be challenging without modifying the original morsel-driven execution framework. This is the problem we address in this thesis.

We end this section with a note about our actual implementation that may be of independent interest to users. The IFE-style iterative subroutine is commonly implemented in graph analytics systems, such as Pregel or Ligra, or graph analytics libraries of GDBMSs such as Neo4j's graph data science (GDS) library, to implement a suite of batch graph analytics algorithms, including shortest paths algorithms as well as PageRank, graph connectivity, and others. These systems or libraries use an abstraction that looks similar to the pseudocode we provided above. That is, these systems or libraries implement recursive graph analytics computations using an imperative code using `Frontier` and `Graph` abstractions. However at the core engines of GDBMSs, as in every DBMS we are aware of, systems use a tuple-based (and often pull-based) dataflow abstraction, such as one where actual query processing operators use a `getNextTuples()` call to pull tuples from child operators. Let us refer to the former as a graph-based query processor abstraction and the latter as a tuple-based query processor abstraction. The dataflow style abstraction is arguably a low-level and makes it harder to implement the overall control logic than the imperative version. Having implemented IFE-style algorithms in both styles for this thesis work, we also found the graph-based iterative abstraction easier to implement some advanced optimizations that are out of the scope of this thesis. An example is the well-known direction-optimized BFS optimization [5], which can be used in the unweighted shortest path example from Example 1. In this optimization, in some iterations extensions extensions are done "backwards" from unvisited nodes to nodes that are in the latest frontier. In other iterations, extensions are done "forwards" from the frontier to unvisited nodes. Writing this logic with a graph-based abstraction is relatively straightforward compared to writing this logic in a tuple-based abstraction.

Since GDBMSs at their cores have tuple-based abstraction for most of their query processing operators, such as regular scans, filters, aggregations, unions, etc., implementing a separate graph-based abstraction for recursive queries may look unnecessary. Yet, we think

```
/*
 * To track the no. of times
 * a node is visited at a length.
 * The next pointer points to the
 * previous length at which the
 * node was encountered.
*/
struct multiplicity {
    int count
    int length
    multiplicity *next
}

class KleeneStar {
    // 0 signifies unvisited
    visitedState = [0,0,...0]

    // all multiplicities initialized to null
    nodeMultiplicities = [null, null, ... null]

    bool edgeCompute (e=(u, v)) {
        visitedState[v] = 1
        if (nodeMultiplicities[v].length != curIteration + 1) {
            oldMultiplicity = nodeMultiplicies[v]
            nodeMultiplicities[v] = new multiplicity
            nodeMultiplicities[v].next = oldMultiplicity
            nodeMultiplicities[v].length = curIteration + 1
        }
        nodeMultiplicititities[v].count++
        return true
    }

    bool complete() {
        return curIteration != upperBound || curFrontier.empty()
    }

}
```

Listing 2.3: Pseudo-code: Kleene Star

that building a higher-level graph abstraction that underneath uses a tuple-based abstraction significantly simplifies implementing advanced recursive query processing algorithms and modern GDBMS implementors will benefit from this engineering investment. For example, as we will show, implementing a graph-based abstraction makes it very easy to have two versions of the same recursive query processing algorithm that scan the adjacency lists from disk or from an in-memory index, using different implementations of a `Graph` class. Further, this same infrastructure can also be used to execute other graph algorithms if the GDBMS aims to have a GDS library. Adopting this abstraction in an actual query processor is indeed one of the design goals of our implementation in this thesis.

Chapter 3

System Overview

This chapter gives an overview of our query process architecture that runs recursive queries. We describe the overall query plans, the `IFECall` operator, which uses a `IFEAlgorithm` implementation that implements a recursive query algorithm using a graph abstraction. We give several examples of `IFEAlgorithm` implementations. Finally, we describe how `IFECall` and `IFEAlgorithm` interface with the rest of the query plan and the system's worker thread pool to parallelize queries. In particular, we will describe a design space of parallelization approaches to parallelize queries and suggest one of them as a robust choice for systems.

3.1 Query Plan Overview and `IFECall` Operator

Overall each recursive clause compiles to sub-plans, each of which is a separate pipeline, as follows:

1. One subplan computes a set of source and destination tuples between which a recursive algorithm will run.
2. A second subplan, consumes these tuples and consists only of the `IFECall` operator, which executes the recursive algorithm that implements the recursive clause and materializes its outputs into an intermediate table.
3. Finally, a third subplan consumes the outputs of the recursive algorithm and evaluates the rest of the query, which may consist of other `IFECall` operators.

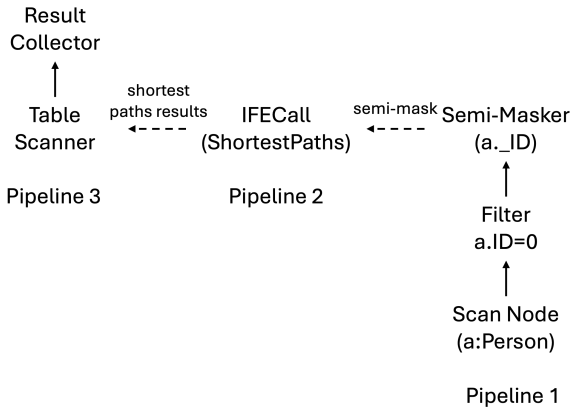


Figure 3.1: Query Plan with `IFECall` operator.

Consider as an example the following shortest paths query from Chapter 2:

```
MATCH p = (a:Person)-[r:Knows* SHORTEST]->(b:Person)
WHERE a.ID = 1 RETURN b, length(p)
```

Listing 3.1: Example recursive query

Figure 3.1 shows the query plan generated for this query. The right chain of operators finds all source and destination nodes between which the shortest paths computation should be computed. These are stored in a *semi-mask*, which is a compact data structure that stores the offsets, i.e., the records IDs, of these source and destination nodes. In the figure, since there are no filters on the destination nodes, only the offsets of the source nodes are computed. This semi-mask is shared with the `IFECall` operator, which computes the shortest paths and stores the outputs of the recursive clause in a table, which is shared with the third sub-plan, which in this query only returns the results. We simplify the third sub-plan for the purpose of presentation. In reality, the `IFECall` operator will only return the ids of destination `b` nodes and the lengths of the shortest paths from the source node `a`. It will not output the other properties of `b` nodes, which are scanned in the last sub-plan.

As shown in Figure 3.1, the overall generated plan is in dataflow abstraction. However, the actual recursive algorithm execution, which happens inside `IFECall` operator, which executes an `IFEAlgorithm` that implements a recursive algorithm using a graph abstraction. In the next section, we overview this interface.

```
class IFEAlgorithm {
public:
    virtual bool edgeCompute(nodeID u, nodeID v) = 0;
    virtual bool isComplete() = 0;
    void exec();
protected:
    Graph graph;
    SemiMask srcNodes, dstNodes;
    Frontier currentFrontier, nextFrontier;
    Table results;
    int curIter;
    TaskScheduler taskScheduler;
};

class Graph {
public:
    common::offset_t getNumNodes();
    common::offset_t getNumEdges();
    std::vector<nodeID_t> getFwdNbrs(nodeID u);
    std::vector<nodeID_t> getBwdNbrs(nodeID u);
};

class Frontier {
   bool isActive(nodeID u);
   bool setActive(nodeID u);
};
```

Listing 3.2: IFEAlgorithm and its supporting interfaces.

3.2 IFEAlgorithm Interfaces

`IFEAlgorithm` and its major components are broadly based on the graph-based interfaces provided in vertex-centric parallel graph analytics systems such as Ligra [32]. The major components of `IFEAlgorithm` and its supporting interfaces are shown in Listing 3.2. At a high level, different recursive query algorithms extend and implement the `edgeCompute()` and `isComplete()` functions. The `exec()` function implements the overall IFE subroutine making calls to the specific `edgeCompute()` and `isComplete()` functions. The overall parallelization approaches we implement in this thesis are implemented inside the `exec()` function. We will review and give examples of these functions momentarily. First, we review the other interfaces, `Graph`, `SemiMask`, `Frontier`, and `TaskScheduler`, that are used by `IFEAlgorithm`.

Before we review these interfaces, we note that all interfaces support multiple node and relationship types. However, for simplicity, we do not make this explicit in our presentation and assume that each structure supports a single node and relationship label. For example, in our actual implementation, the `getFwdNbrs()` function in the `Graph` interface takes in a relationship label as an argument as well, but we omit this in our description. Readers can assume we focus on queries that run on a graph database with a single node and relationship labels.

Graph: The `graph` field gives algorithms access to neighbors of nodes in the subgraph that the `graph` represents. Depending on the queries, `graph` can be the entire database or a subset of the database, an in-memory index storing a graph, or an on-disk implementation that actually scans tuples from the disk and buffer manager of the system. Algorithms are abstracted away from this detail and will execute correctly as long as the `graph` provides thread-safe access to the neighborhoods of necessary nodes that the algorithm needs for its computation.

Semi-masks: Semi-masks are dense bitset-like data structures that identify a set of nodes using a single byte per node. They are used in `sourceNodes` and `destinationNodes` fields to identify a set of source and destination nodes between which a recursive computation should be performed[1].

Frontier: Similar to the semi-masks, this is a bitset-like data structure that identifies a set of nodes. It exposes the `isActive()` function to read if a node is in the frontier and the `setActive()` function to put a node in the frontier. IFE-based implementations of `IFEAlgorithms` have access to this interface. There are several implementations of a

[1]We are simplifying here as in our actual implementation they also take a direction field to know in which direction, forward or backward, the recursive query is specified.

Frontier. Specifically, there are both sparse and dense implementations. As we discuss in Section 3.4, we switch between these frontier implementations depending on the number of vertices in a frontier.

Table: `IFEAlgorithms` materialize their outputs to the `result` field, which is an intermediate table implementation[2]

curIter: is an integer field indicating the current iteration of an IFE computation.

TaskScheduler: This is the system's task scheduler which provides a means to schedule tasks for worker threads to parallelize. In particular `TaskScheduler` exposes functions such as `submitTasks` which the caller, in this case a `IFEAlgorithm` implementation, can use to submit a task for worker threads to work on and for each task identify the maximum number of threads to work on the task. The caller can monitor the state of submitted tasks and issue more tasks to implement a scheduling algorithm that can implement a parallelization approach. Momentarily, we give an example of how we implement one parallelization approach that we call *nT1S* using the task scheduler interface. In Section 3.3, we describe two other parallelization approaches including the *nTkS* approach, which we recommend as the robust approach that should be used in GDBMSs.

`IFECall` task/pipeline, which executes a specific `IFEAlgorithm` is single-threaded. The query compiler which breaks down the user's query, say the one in Listing 3.1, puts the `IFECall` task of the query into the same task scheduler that `IFEAlgorithm` implementations have access to through their `taskScheduler` field. The `IFEAlgorithm` task is marked as single-threaded. This is because the main goal of `IFEAlgorithm` is not to do the computation-heavy part of a recursive algorithm, but instead to implement the general control flow of the algorithm in an imperative manner. As we demonstrate momentarily in several examples, the computation-heavy parts of recursive algorithms are put into the task scheduler as separate `Task` objects. In other words, the `IFEAlgorithm`, which is running inside a `Task` object, spawns new `Tasks` as it is executing. This is similar to the role given to single-threaded `master.compute()` interfaces in some vertex-centric graph analytics APIs [30, 6], which is separate from the parallel `vertex.compute()` or `edgeCompute()`-like functions that perform the actual computation.

We will discuss how to parallelize IFE computations momentarily. To provide examples of `IFEAlgorithm` implementations that evaluate different recursive clauses, Listing 3.3 shows a serial implementation of `exec()` function of `IFEAlgorithm` (ignore the green and red boxes for now). The code takes each source s from which an IFE computation needs

[2]We are slightly simplifying the presentation here. In Kùzu, this class is actually called FactorizedTable, as it can keep results in compressed format using factorization [8, 26], although we do not factorize results of recursive algorithms.

```
1  void exec() {
2     for (auto s : srcNodes):
3        nextFrontier.setActive(s);
4        while (!isComplete()):
5           swapCurNextFrontiers();
6           for (auto u : graph->nodes()):
7              if (curFrontier->isActive(u)):
8              for (auto v : graph->scanFwd(u)):
9                 if (edgeCompute(u, v)):
10                   nextFrontier->setActive(v);
11          curFrontier.reset();
12       outputResults();
13 }
```

Listing 3.3: Serial IFE execution.

to be executed one at a time. Let us call the execution of an IFE algorithm from a particular source node s as an *IFE computation*. Then an inner while loop starting on line 4 runs the IFE computation following the pseudocode of the IFE subroutine we showed in Algorithm 2.1.

Listing 3.4 shows the actual code that is written to implement an `IFEAlgorithm` that implements a Bellman-Ford like shortest paths algorithm. This `IFEAlgorithm` implements the "SHORTEST" recursive clause in the Cypher dialect of Kúzu. We assume that there are two additional data structures, `VisitedState` and `Paths`. `VisitedState` keep track of which vertices have been visited, while `Paths` keeps the parent edges to keep track of the shortest paths that have been found. We omit details of the actual implementations of these structures. In our actual implementation, these are implemented in a thread-safe manner using C++ atomics to synchronize concurrent modifications to the values of a particular vertex. The convergence criterion, implemented in the `isComplete()` function, is when either the next frontier is empty or when all destination nodes, which are stored in the `dstNodes` semi-mask, are visited. For reference, Listing 3.5 shows the `IFEAlgorithm` that implements the Kleene star clause in Cypher. This code is very similar to the shortest paths example, except we do not keep track of visited states, since this clause finds all paths up to a maximum iteration number of IFE (`maxIter`). These example codes are close to our actual implementation except for the parts about writing outputs, which we omit to simplify our presentation.

```
class SSSP : IFEAlgorithm {
VisitedStates visited;
Paths paths;
void edgeCompute(nodeID u, nodeID v) {
if (visited[v] == UNVISITED)
   paths[v].add(u);
   visited[v] = VISITED;
   return true;
}
void isComplete() {
   return nextFrontier.isEmpty() || allDestinationsVisited();
}}
```

Listing 3.4: Shortest paths implementation using IFEAlgorithm interface.

```
class KleeneStar : IFEAlgorithm {
Paths paths;
void edgeCompute(nodeID u, nodeID v) {
   paths[v].add(u);
   return true;
}
void isComplete() {
   return curIter == maxIter;
}}
```

Listing 3.5: Kleene star implementation using IFEAlgorithm interface.

3.3 A Design Space for Scheduling Policies

We next describe a design space of natural parallelization approaches for recursive query algorithms. Henceforth, we refer to these parallelization approaches as scheduling policies. The design space is based on which parts of the serial `IFEAlgorithm` execution, shown in Listing 3.3, that the policy parallelizes. In the serial execution, there are 2 for loops that contain scans and which can be parallelized. These are on lines 2 and 6. We describe a design space that consists of three natural approaches, one that parallelizes only the scans in the outer for loop (*1T1S*), one that parallelizes the scans in the inner for loop (*nT1S*), and one that parallelizes scans at both for loops (*nTkS*) and is the scheduling policy that we implement as a robust approach to parallelizing recursive queries.

The *nT1S* policy only runs 1 source at a time as a parallel task (n threads, 1 source), *1T1S* policy assigns every thread 1 source as a single thread task (1 thread, 1 source) and finally the hybrid *nTkS* policy executes k independent sources as parallel tasks (n threads, k sources). We have implemented each policy in Kúzu and will present experiments comparing their performances in Chapter 4.

3.3.1 1T1S Policy

The first for loop on line 2 scans each source node s from the list of source nodes to the entire IFE computation, and starts a new IFE computation, e.g., the shortest paths, from s. Let us call the execution of an IFE algorithm from a particular source node s as an *IFE computation*. In vanilla morsel-driven parallelism approach of DBMSs, 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 that DuckPGQ [35] or Neo4j [12] adopts when executing the recursive clauses (though more than 1 IFE computation can be assigned to a single thread in a single morsel). This is the approach we call *1T1S*. That is, this approach parallelizes entire IFE computations (Line 3-12 in Listing 3.3) as a unit of work.

The advantage of 1T1S is that when there are many sources from which an IFE computation should be performed, 1T1S can easily keep threads busy. 1T1S can also be implemented using faster data structures, e.g., without using any atomics or synchronization primitives, compared to policies in which multiple threads can work on the same IFE computation. At the same time, this approach will not be able to utilize multiple threads efficiently on queries that require computing one or a few IFE computations.

Listing 3.6 shows the pseudocode of the IFEAlgorithm's `exec()` function in our implementation of the 1T1S policy. Each IFE computation is wrapped around an `IFETask`,

```
void exec() {
   vector<IFETask> tasks;
   for (auto s : srcNodes):
      auto ifeTask = IFETask(this, s); // set to ife and s fields
      ifeTask.setMaxThreads(1);
      tasks.push(ifeTask);
   taskScheduler.schedule(tasks);
}

class IFETask : Task {
private:
   IFEAlgorithm ife;
   nodeID s;
void run() {
   ife.nextFrontier.setActive(s);
   while (!ife.isComplete()):
       ife.swapCurNextFrontiers();
       for (auto u : ife.graph->nodes()):
            if (ife.curFrontier->isActive(u)):
                for (auto v : ife.graph->scanFwd(u)):
                    if (ife.edgeCompute(u, v)):
                        ife.nextFrontier->setActive(v);
       ife.curFrontier.reset();
   ife.outputResults();
}}
```

Listing 3.6: 1T1S scheduling policy.

which is scheduled to run in a single-threaded manner. For simplicity, the pseudocode schedules all tasks at once. Our actual implementation is more complex and we schedule as many tasks as there are worker threads and monitor when tasks finish and launch. Finally, Figure 3.2 further shows pictorially the units of work that worker threads execute.

3.3.2 nT1S Policy

Listing 3.3 contains a second scan on line 6, where each frontier of each IFE computation is scanned to run the `edgeCompute()` function. Instead of the scan in the first for loop, this scan can also be split into morsels and parallelized. We call the approach that parallelizes only these scans as *nT1S*. This is the approach implemented in parallel graph analytics systems, such as Pregel [21] and Ligra [32], that provide a vertex-centric API to implement batch graph analytics libraries.

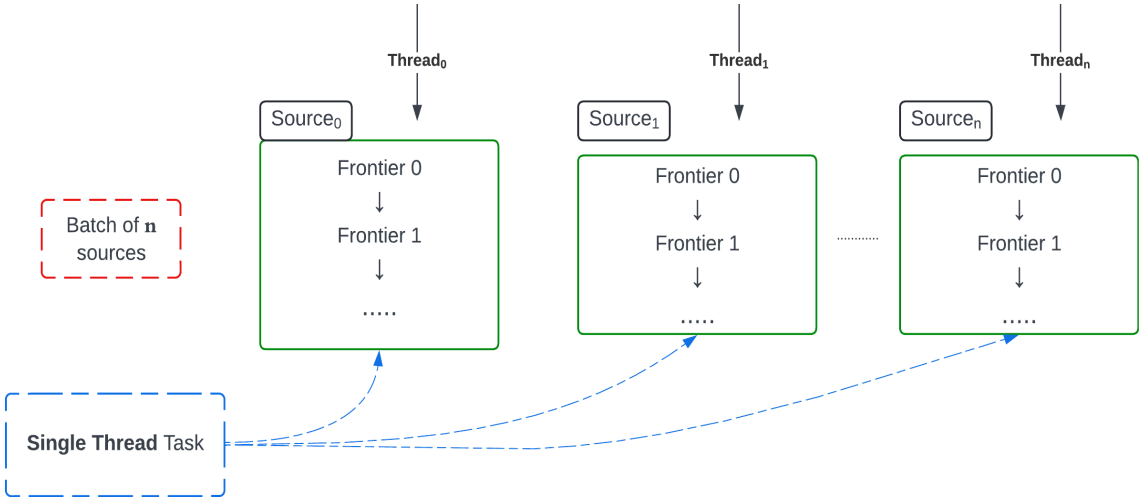


Figure 3.2: *1T1S* scheduling

The advantage of nT1S is that when there are one or a few IFE computations to perform in a query, nT1S can share work. However, this approach is limited by how much a single IFE computation can parallelize. Due to Amdahl's Law [2] the scalability of any program will be limited to to its parallelizable parts. However, in any IFE computation, there will be factors limiting the parallelizable work. Most importantly there will be frontiers that are sparse and do not have enough work, such as the first frontier which only has a single node in it (the source). In addition, there are parts where workers need to synchronize, such as when grabbing frontier morsels or when updating common data structures, or system components, such as the buffer managers. Therefore by its nature, parallelizing at the frontiers is more prone to such limitations than parallelizing entire IFE computations, where most, if not all of these limiting factors can be avoided.

Listing 3.7 shows the pseudocode of the IFEAlgorithm's `exec()` function in our implementation of the nT1S policy. nT1S takes each source s from which an IFE computation needs to be executed one at a time, which are not parallelized (unlike the 1T1S policy). Then, each frontier extension step of each IFE computation is parallelized by submitting a `FrontierTask` to the task scheduler, whose computation is also shown in Listing 3.7. Importantly, in nT1S policy, all available worker threads work on each `FrontierTask`[3]. Each worker thread grabs a morsel of nodes to run the `edgeCompute())` function inside the while loop on line 13. This is done in a `getMorsel()` function (which was omitted

[3]We assume in the pseudocode that if `setMaxThreads()` is not called, all worker threads can work on a task

```
1  void exec() {
2     for (auto s : srcNodes):
3        nextFrontier.setActive(s);
4        while (!isComplete()):
5           swapCurNextFrontiers();
6           taskScheduler.schedule(FrontierTask(this)); // this set to ife
7           curFrontier.reset();
8  }
9
10 class FrontierTask : Task {
11 IFEAlgorithm ife;
12 void run() {
13    for (auto frontierMorsel: ife.curFrontier.getMorsel()) {
14       for (auto u : frontierMorsel):
15          if (ife.curFrontier->isActive(u)):
16             for (auto v : ife.graph->scanFwd(u)):
17                if (ife.edgeCompute(u, v)):
18                   ife.nextFrontier->setActive(v);
19 }}
```

Listing 3.7: nT1S scheduling policy.

in the `Frontier` interface description above for simplicity). A frontierMorsel is a range of node IDs. For each node u in the frontier morsel, if u is active in the current frontier, the worker thread reads u's neighbors and runs the `edgeCompute()` function. Figure 3.3 pictorially shows the units of work that worker threads execute.

3.3.3 nTkS Policy

A third approach we propose is the hybrid scheduling policy that combines parallelization of both scans. We call this approach *nTkS* and consider it the robust approach to parallelizing recursive queries. The *nTkS* scheduling policy aims to run 'k' concurrent IFE computations in parallel (Line 2) while simultaneously running each IFE computation in parallel (Line 4). This gives the benefit of *1T1S* scheduling where if we have a lot of sources available to run independently we can execute them (a constraint 'k' is placed on maximum such sources) and also the benefit of *nT1S* scheduling where a single IFE computation can be executed by multiple threads. It only adds the overhead of having to write the `IFEAlgorithm` code in a thread-safe manner (using C++ atomics) and having to manage 'k' IFE computations at the same time.

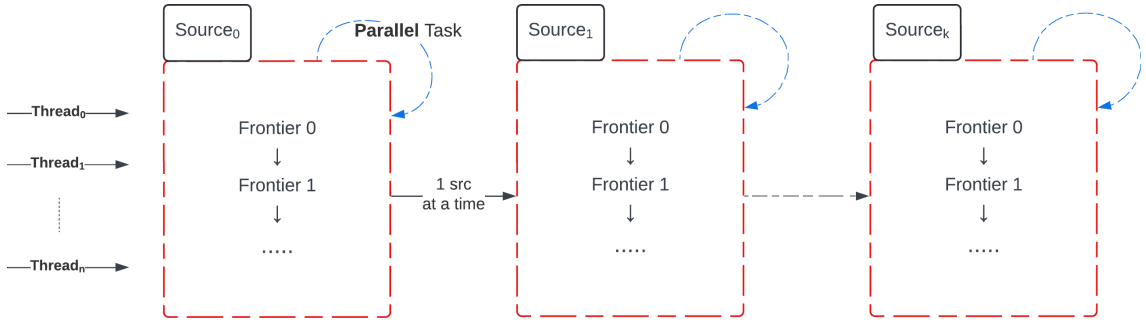


Figure 3.3: *nT1S* scheduling

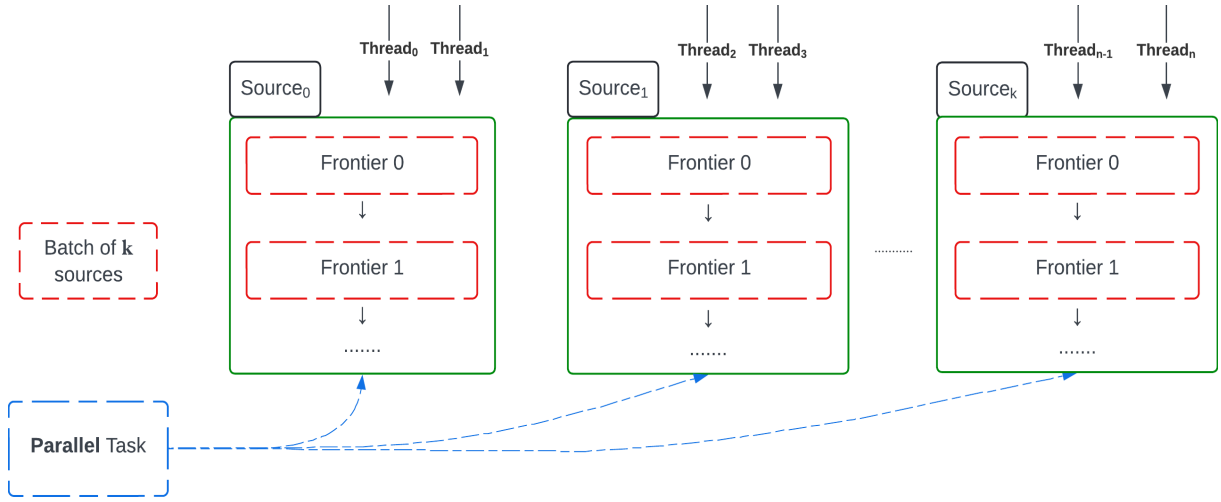


Figure 3.4: *nTkS* scheduling

Listing 3.8 shows the pseudo-code of the IFEAlgorithm's exec() function in our implementation of the *nTkS* policy. At any point there are 'k' IFE sources being run at the same time, each has a task we refer to as a `srcTask` associated with it. We omit some of the exact details of how *nTkS* functions (such as how each 'k' `srcTask` is maintained and launched for new tasks) and give a high level overview through Listing 3.8. When a source's IFE computation is complete (Line 7) we increment the total sources completed and use the `srcTask` to start a new IFE computation with the next source. If it is not complete (Line 13, the next frontier has to be prepared and the `srcTask` needs to be resubmitted to the `TaskScheduler`. The frontier extension for each IFE computation is parallelized similar to how the *nT1S* policy parallelizes it.

Figure 3.4 shows the representation of how *nTkS* executes. The outer green box rep-

resents the IFE source as a unit of work and the inner red box represents the frontier extensions within a source as a unit of work.

3.4 Further Optimizations and Implementation Details

3.4.1 In-Memory Graph Index

The **Graph** interface described in Section 3.2 is used by the `IFEAlgorithm` to access neighbors of nodes when performing recursive computation. The data can either reside on disk or, as a **C**ompressed **S**parse **R**ow (CSR) index saved in memory. This CSR index is referred to as `InMemGraph`, and allows the `IFEAlgorithm` to bypass reading data through Kùzu's Buffer Manager.

Figure 3.5 shows the query plan generated when the `IFEAlgorithm` needs to be performed on the `InMemGraph`. The plan is similar to the on-disk case (Figure 3.1) except an additional `CSRIndexBuild` pipeline is appended below the `IFECall` pipeline. The CSR Index is constructed in parallel and passed to `IFECall`. When threads perform recursive computation they use APIs exposed by `InMemGraph` to scan the CSR and read neighbors of nodes. An additional benefit of the in-memory index is that it does not require copying data from Buffer Manager pages to intermediate structures to read neighbors of nodes.

Figure 3.6 shows the in-memory structure of the CSR Index. To efficiently build the index in parallel, we morsellize the `CSRIndexBuild` pipeline by giving out morsels (currently the size is fixed at 2048). Each thread that receives a morsel of 2048 nodes builds the CSR Index for its morsel (referred to as `MorselCSR` in Figure 3.6) and merges it to the global CSR Index. To lookup neighbors of nodes during `IFECall`, we translate the `MorselCSR` to which a node belongs and then sequentially scan the neighbors from the index. Two separate arrays are maintained to lookup the neighbors of a node, the `csrStartPos` array that keeps track of the starting position of each node's neighbors and `nbrNodeOffsets` array that stores the actual neighbors. For a node u, the range of neighbors can be read by scanning `nbrNodeOffsets` from the position `csrStartPos[`u`]` to `csrStartPos[`u`+1]`.

3.4.2 Sparse & Dense Frontier Representation

When performing the iterative BFS extension, the BFS Frontier whose neighbors are being explored currently (`curFrontier`) can be expressed in two ways: (a) sparse frontier and, (b)

```
void exec() {
   numCompletedSrcs = 0; // track completed sources
   srcNodeTasks(k); // keep ‘k’ tasks for ‘k’ active sources
   // keep submitting sources until complete
   while (true && numCompletedSrcs < totalSrcs) {
       for (auto srcTask : srcNodeTasks) {
           if (srcTask.isComplete()) {
               numCompletedSrcs++;
               srcTask.reset();
               srcTask.setActive(nextSrcNode);
               taskScheduler.schedule(srcTask);
           } else {
               srcTask.ife.curFrontier.reset();
               srcTask.ife.swapCurNextFrontiers();
               taskScheduler.schedule(srcTask);
           }
       }
   }
   // exit once all IFE sources are complete
}

class FrontierTask : Task {
IFEAlgorithm ife;
void run() {
   for (auto frontierMorsel: ife.curFrontier.getMorsel()) {
      for (auto u : frontierMorsel):
         if (ife.curFrontier->isActive(u)):
            for (auto v : ife.graph->scanFwd(u)):
               if (ife.edgeCompute(u, v)):
                  ife.nextFrontier->setActive(v);
   }
}
```

Listing 3.8: nTkS Policy Implementation

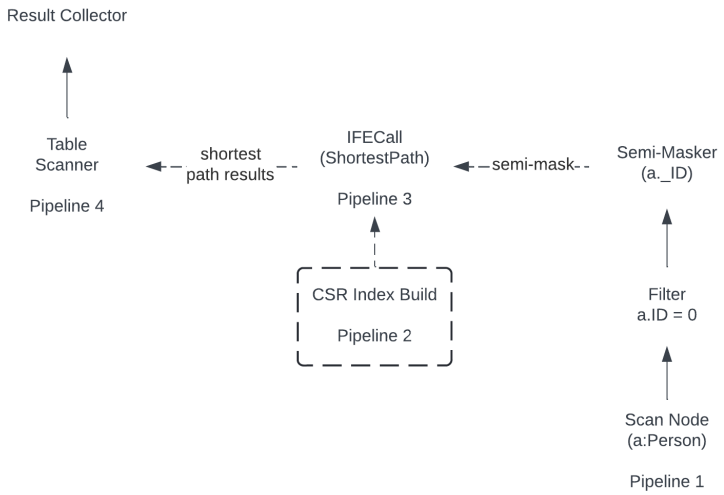


Figure 3.5: Query Plan with `CSRIndexBuild` operator

dense frontier. The main aim of representing the frontier in two ways is space efficiency. The threshold for switching between a sparse and dense representation is chosen to be $(1/8)th$ of $(total\ nodes)$ of the node table on which the `IFECall` is being performed.

The intuition behind this heuristic is as follows: we can either allot a 1 *byte* value (0 or 1) to indicate a node exists in `curFrontier` or, we have to use 8 *bytes* to indicate if a node exists in the frontier (Kùzu assigns unique 8 *byte* identifiers to nodes). This condition can be expressed as follows:

$$\boxed{\begin{aligned}&(1\ byte * total\ nodes) > (8\ bytes * actual\ frontier\ size)\\&(total\ nodes\ /\ 8) > actual\ frontier\ size\end{aligned}}$$

When the actual frontier size is greater than $(1/8)th$ of total nodes in the graph, we use a *sparse* representation and assign a 1-byte value to all nodes in the graph, a value of 1 for all nodes part of the frontier. When the frontier size is less than $(1/8)th$ of total nodes, we switch to a *dense* representation and keep track of every unique node ID (8 *bytes*) part of `curFrontier`.

3.4.3 Lock-Free Implementation

To enable multiple threads to perform BFS extension for the same source's IFE algorithm,

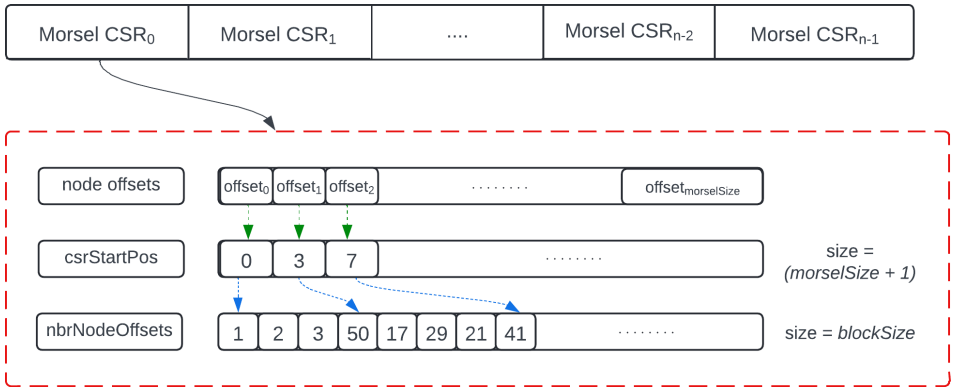


Figure 3.6: In-Memory CSR Index

we use C++ atomics for thread-safe updates to shared data structures. An atomic operation is a special CPU instruction that guarantees atomic accesses and updates of memory locations. Listing 3.4 which was discussed previously provided a snippet for shortest path `IFEAlgorithm` implemented on a single thread. Listing 3.9 shows unweighted shortest path `IFEAlgorithm` with its `edgeCompute` function implemented using atomics. All updates to the `VisitedStates` and `PathLength` data structure are done using these atomics. There are 3 atomic operations being performed:

- `ATOMIC_CAS`: takes 3 arguments, a pointer to a memory location, the value expected to be present in this memory location and the new value to be written, returns true if the operation was successful in updating the memory location with the new value, else false
- `ATOMIC_LOAD`: takes 1 argument, pointer to a memory location, reads atomically the value of the memory location guaranteeing the thread reads all updates made to that location
- `ATOMIC_STORE`: takes 2 arguments, pointer to a memory location and the new value to be placed in the memory location, guarantees that all atomic reads that execute after this operation read the updated value

In the `edgeCompute` function described in Listing 3.9, we first do an atomic read of the state of neighbor node v (Line 6) to check if it has been visited already. If not visited, we try to update the state of this node to visited (Line 8). Based on whether this CAS operation is successful, the thread that succeeds increments the total destinations visited and stores the path length value for the node (Line 10).

```
class SSSP : IFEAlgorithm {
    VisitedStates visited;
    PathLength pathLength;

    bool edgeCompute (e=(u, v)) {
        state = ATOMIC_LOAD(&visitedState[v]);
        if (state == UNVISITED) {
            if (ATOMIC_CAS(&visitedState[v], state, VISITED) {
                ife.numVisitedDst++;
                ATOMIC_STORE(&pathLength[v], curIteration + 1);
            }
            return true;
        }
        return false;
    }

}
```

Listing 3.9: Lock-free Implementation: Unweighted Shortest Path

Chapter 4

Evaluation

We integrated the three morsel-driven parallelism approaches into Kúzu, an embedded columnar disk- based GDBMS. This section presents a suite of experiments with the primary goal of demonstrating the robustness of the nTkS approach. To this end, we evaluate how well nTkS parallelizes on 3 sets of microbenchmark queries that require computing a recursive query from a single source, a few sources, and many sources. We compare the scalability of nTkS with 1T1S, which captures the behavior of vanilla morsel-driven approach, and nT1S, which captures the behavior of vertex-centric parallel graph analytics systems. In addition, we present experiments that aim to answer when using an in-memory CSR is beneficial over just scanning data from disk and through buffer manager. Specifically, we provide experiments that answer how much performance benefits we obtain from using an in-memory CSR index on our 3 sets of microbenchmark queries.

4.1 Experimental Setup

4.1.1 Hardware

We use a single machine with Intel Xeon CPU E5-2670 @2.60GHz processors and 256GB of memory. The machine has 16 physical cores and 32 logical cores across 2 NUMA nodes. It has a standard 3-level cache spread across L1, L2 and L3 cache; L1 (data) cache size: 32KB, L1 (instruction) cache size: 32 KB, L2 cache size: 256 KB, L3 cache size: 20 MB. Kúzu uses a memory-backed region managed by a custom buffer manager to read on-disk graph data, the memory limit for this region is set to 64GB for the experiments.

4.1.2 Datasets

Table 4.1 lists the datasets used for our benchmark queries. These datasets include synthetic graphs generated with LDBC benchmark tools [34], social network graphs, and a streaming platform graph created from data released by Spotify [14]. The LDBC synthetic dataset, generated with a scale factor of 100, is referred to as LDBC100. The social network dataset used is the LiveJournal graph from SNAP [33]. We choose these datasets as each is large enough containing between 20 million edges to 1.9 billion edges, so even single source recursive queries are expensive enough that they have room for parallelization. Further, these datasets allow us to test our parallelization approaches and in-memory CRS index on graphs with different average degree characteristics, varying from 14 (LiveJournal), 44 (LDBC100), and 535 (Spotify). When we load these datasets into Kúzu, we use "Node" and "Rel" as labels of the nodes and edges.

Name	$\|V\|$	$\|E\|$
LDBC100	448,626	19,941,198
Livejournal	4,847,571	68,993,773
Spotify	3,604,454	1,927,482,013

Table 4.1: Datasets.

4.1.3 Query workload

For running evaluation we use microbenchmarks generated for single and multi-source shortest path queries. The queries find the shortest path starting from the source nodes to the rest of the nodes in the graph, i.e., the destination nodes' semi-masks contain all the nodes in the graph. The queries return the destination nodes that were reached and their path lengths. Our goal is to evaluate the parallelism behavior of the actual recursive queries. To focus on this goal, we do not return actual paths to reduce the size of the outputs, which can grow a lot, especially on queries that output shortest paths to all destinations from many source nodes. The overall structure of the queries looks as follows[1]

```
MATCH p = (a:Node)-[r:Rel* SHORTEST]->(b:Node)
WHERE a.id <= x
RETURN b, length(p)
```

[1]There is an alternative equivalent way to run these microbenchmark queries using Cypher CALL statement that contains a `shortest_paths` function, which is what we used in our experiments.

The condition in WHERE is different on different microbenchmarks. We use 3 microbenchmarks on each dataset. First are queries that contain single sources. Second, are queries with few sources, where we set the predicate in the WHERE clause to contain 10 nodes. Third are queries with many sources, where we set the predicate to contain 50 sources. The sources for single-source queries are picked randomly. The sources for workloads with few and many sources use a random but contiguous range predicate on the `id` property of nodes in the WHERE clause. Unless otherwise specified, all queries are run with 5 different sets of sources, for each set we run the queries 3 times and take the average execution time. Then we average the runtime across the 5 different sets. For nTkS, the concurrent IFE computations, i.e., the value of k is set to the minimum between the total threads available for the query and the total sources for the query. For example, in the multi-source (few) query (10 sources), if we have 4 threads available, then `nTkS` will launch at most 4 independent sources. This heuristic for k works well but is not optimal. In a separate experiment, we demonstrate the behavior of nTkS using different k values experimentally in Section 4.4. For 1T1S, we set the number of concurrent IFE computations to the number of threads.

4.2 Single Source Queries Workload

This section evaluates parallelism achieved on a single source recursive join shortest path query by nTkS. We use our microbenchmark queries ensuring that the predicate in the WHERE clause matches a single source node `a`. We run all queries using an on-disk `Graph` implementation, i.e., all neighbors are scanned from the disk and through the buffer manager. We run the queries starting from 1 thread to 16 threads, doubling the number of threads in each experiment. We repeat this suite of experiments under each of the scheduling policies. We expect that nTkS's behavior mimics nT1S's behavior and these policies both benefit from parallelism even when there is a single source node to work on. In contrast, we do not expect 1T1S's performance to improve with more parallelism.

Table 4.2 shows the execution time for LDBC100, Livejournal, and Spotify datasets. The table reports the execution number measures the `IFECall` time. The end-to-end query times are slightly slower than these but parallelize in a similar fashion. However, since our work focuses on parallelizing the recursive query computations, we report and interpret the behavior of the time taken by `IFECall`. As we expect, across all 3 datasets both nT1S and nTkS policies scale with additional threads, while 1T1S does not. Further nT1S and nTkS perform very similarly across datasets, confirming that nTkS reduces to the behavior of nT1S on single source queries. On a single thread, `1T1S` policy runs the fastest compared to

	Threads (→)	1	2	4	8	16
		IFE Call	IFE Call	IFE Call	IFE Call	IFE Call
	nT1S	410	215 (1.9x)	127 (3.2x)	82 (5.0x)	65 (6.3x)
LDBC100	1T1S	404	395 (1.0x)	391 (1.0x)	394 (1.0x)	391 (1.0x)
	nTkS	440	218 (2.0x)	134 (3.3x)	81 (5.4x)	66 (6.7x)
	nT1S	2491	1210 (2.0x)	740 (3.4x)	469 (5.3x)	371 (6.7x)
Livejournal	1T1S	2241	2248 (1.0x)	2239 (1.0x)	2249 (1.0x)	2239 (1.0x)
	nTkS	2637	1275 (2.0x)	788 (3.4x)	485 (5.4x)	377 (7.0x)
	nT1S	13399	6525 (2.1x)	3436 (3.9x)	1964 (6.8x)	1161 (11.5x)
Spotify	1T1S	12472	12612 (1.0x)	12620 (1.0x)	12428 (1.0x)	12491 (1.0x)
	nTkS	14193	7124 (1.9x)	3868 (3.7x)	2013 (7.0x)	1225 (11.6x)

Table 4.2: Single Source Parallelism: Disk-based Graph (in **milliseconds**)

`nT1S` and `nTkS` since it does not require atomic synchronization while updating the shared memory data structures and only submits a single thread task for BFS extension. Starting with two threads however, `nT1S` and `nTkS` begin to outperform `1T1S`.

From Amdahl's Law [2] we know that the benefits of parallelization are limited only to those parts of the overall program that are parallelizable. If there are serial parts within the overall program execution, those parts will not gain any performance improvement with more threads. Due to this, the actual speedup gained with 16 physical processors for the overall execution time cannot reach 16x and will be lower.
There are several inherently serial parts of our parallel recursive computation implementation, such as fetching a "morsel" which requires synchronization, the process of initialization, and submission of the job that needs to be parallelized to the `TaskScheduler`, and the initialization of the next BFS frontier between each parallel BFS extension phase. Some BFS frontiers are sparse and do not gain a significant speedup with more threads beyond a threshold. Finally, scans are done through the buffer manager, which contains some locking mechanism at the database page level while reading the contents of pages leading to additional synchronization overhead. All of these factors play a role in limiting achieving perfect scalability.

	Threads (→)					
BFS Level (↓)	**1**	**2**	**4**	**8**	**16**	
Level 0 (src node)	1	1	1	1	1	**1.0x**
Level 1 (17 nodes)	4	4	2	3	3	**1.3x**
Level 2 (2053 nodes)	11	7	4	6	4	**2.8x**
Level 3 (64326 nodes)	61	48	24	21	13	**4.7x**
Level 4 (276175 nodes)	266	92	48	36	17	**15.6x**
Level 5 (56731 nodes)	36	27	16	8	6	**6.0x**
Level 6 (6044 nodes)	9	6	3	3	3	**3.0x**
Level 7 (1465 nodes)	8	3	3	2	2	**4.0x**
Level 8 (458 nodes)	4	5	2	2	2	**2.0x**
Level 9 (93 nodes)	4	2	2	2	2	**2.0x**
Level 10 (27 nodes)	2	3	2	2	2	**1.0x**
Level 11 (7 nodes)	2	3	1	2	2	**1.0x**
Write Distance	16	12	5	2	1	**16.0x**
Total Runtime	445	217	117	94	62	**7.2x**

Table 4.3: Scalability of each frontier computation (disk-based graph). Times are in ms.

To demonstrate that spar frontier levels do not parallelize well, we performed a further analysis for an LDBC dataset query. We picked a particular source node and measured how much runtime was spent exploring neighbors of each frontier level. Table 4.3 shows how much speedup is achieved for each frontier level. We observe that the densest frontiers parallelize well (BFS Level 4 - $276,175$ nodes), 15.6x with 16 threads and the overall runtime speed up is 7.2x. Sparse frontiers, such as levels $1, 2$, and $6-11$, do not similarly benefit from more threads. This behavior alone is a major factor limiting scalability if a system only breaks frontiers into morsels to parallelize as the nT1S policy does. One can achieve more parallelism by using a combination of morsels at the source and frontier level as done by the nTkS policy.

4.3 Multi-source Parallelism (few sources)

We next evaluate how well nTkS parallelizes queries that contain a few number of sources. We set the number of sources in our microbenchmark queries to 10. We compare the performance of all 3 policies nT1S, 1T1S, and nTkS under varying parallelism levels. We expect all policies to parallelize in this case. Specifically we expect nT1S to parallelize

	Threads (→)	1	2	4	8	16
		IFE Call	IFE Call	IFE Call	IFE Call	IFE Call
	nT1S	2507	1456 (1.7x)	844 (2.9x)	580 (4.3x)	559 (4.5x)
LDBC100	1T1S	2481	1649 (1.5x)	871 (2.9x)	566 (3.9x)	329 (7.5x)
	nTkS	2642	1443 (1.8x)	933 (2.8x)	376 (7.0x)	255 (10.3x)
	nT1S	19250	11393 (1.7x)	6426 (3.0x)	3868 (5.0x)	3181 (6.0x)
Livejournal	1T1S	18626	9865 (1.9x)	6055 (3.0x)	4502 (4.1x)	2733 (6.8x)
	nTkS	19122	11103 (1.7x)	5589 (3.4x)	3182 (6.0x)	1865 (10.3x)
	nT1S	126642	64419 (1.9x)	32517 (3.9x)	17434 (7.3x)	10470 (12.1x)
Spotify	1T1S	125345	64512 (1.9x)	38844 (3.2x)	27552 (4.6x)	17186 (7.3x)
	nTkS	127165	64331 (1.9x)	33281 (3.8x)	17856 (7.1x)	10364 (12.3x)

Table 4.4: Multi-source (few): Scheduling Policies (On-Disk, in **milliseconds**)

similarly to the single source case since we can expect a similar parallelism behavior on each IFE computation. We also expect 1T1S to perform well, since it can now utilize 10 threads to work on 10 different IFE computations. However, its scalability will be limited because it cannot utilize all threads at least when there are 16 worker threads. Further 1T1S is more prone to the *curse of the slowest IFE computation* phenomenon where the threads that finish their IFE computations do not have a mechanism to help the slowest IFE computation. We expect nTkS to scale as well or better than nT1S because it has more flexibility than nT1S to parallelize. For example when there are 16 threads and a sparse frontier, 15 threads do not have to remain idle. Some of those threads can work on other IFE computations. We also expect nT1S to perform better than 1T1S as it has a mechanism where idle threads can help other slow IFE computations.

Table 4.4 shows the results of the experiment. The results verify our expectation that all policies parallelize, nT1S parallelizes similar to our single-source query experiments, and that nTkS parallelizes as well or better than nT1S and nTkS. For example, on LDBC100 nTkS achieves 10.3x improvement while 1T1S achieves 7.5x and nT1S achieves 4.5x scalability when using 16 threads. An interesting observation we make here and that we will analyze in more detail is that on Spotify, our largest graph with the highest average degree, nT1S performs better than 1T1S and almost identically to nTkS. Our next set of experiments, which use 50 sources, will demonstrate a similar behavior. We will further analyze and explain this behavior after we present those experiments.

	Threads (→)	1	2	4	8	16
		IFE Call	IFE Call	IFE Call	IFE Call	IFE Call
	nT1S	12594	7664 (1.6x)	4289 (2.9x)	2769 (4.6x)	2664 (4.7x)
LDBC100	1T1S	12916	6582 (1.9x)	3431 (3.8x)	1849 (7.0x)	1213 (10.6x)
	nTkS	12634	7270 (1.7x)	3304 (3.8x)	1771 (7.1x)	1078 (11.7x)
	nT1S	94249	57969 (1.6x)	32016 (3.0x)	19501 (4.8x)	14503 (6.5x)
Livejournal	1T1S	91638	47684 (1.9x)	26265 (3.5x)	15265 (6.0x)	9363 (9.8x)
	nTkS	95457	55189 (1.7x)	27123 (3.5x)	14615 (6.5x)	8315 (11.5x)
	nT1S	623362	322610 (1.9x)	162814 (3.8x)	89855 (7.0x)	50969 (12.2x)
Spotify	1T1S	619310	318411 (1.9x)	167421 (3.7x)	100857 (6.1x)	65802 (9.4x)
	nTkS	620703	321456 (1.9x)	181326 (3.4x)	91058 (6.8x)	56445 (11.0x)

Table 4.5: Multi-source (many): Scheduling Policies (On-Disk, in **milliseconds**)

4.4 Multi-source Parallelism (many sources)

This section evaluates parallelism achieved when there are many more sources than the number of worker threads in the system. We modified our microbenchmark queries to now contain 50 sources. Now nTkS and 1T1S can launch up to 16 concurrent IFE computations when running with 16 threads. We expect nT1S to perform similarly as in previous experiments. We also generally expect a similar behavior from 1T1S and nTkS yet we expect them to improve their relative performance compared to nT1S, i.e., outperform nT1S with a larger margin, at least on LDBC100 and LiveJournal because they can maintain their advantage of keeping every thread busy for a larger period of time. Further, we expect 1T1S and nTkS to perform close to each other, since their behaviors will be very similar for a long time, where each thread will work on a separate IFE computation. Only when executing the last batch of 16 IFE computations, will their behavior start differing and nTkS will have an advantage over 1T1S to keep more threads busy.

Table 4.5 shows the results. As shown in the table, nT1S achieves similar levels of parallelism as before. When using 16 threads, nT1S achieves 4.7x improvement on LDBC100 (vs 4.5x in the previous set of experiments), 6.5x improvement on LiveJournal (vs 6.0x), and 12.2x improvement on Spotify (vs 12.1x). Further 1T1S and nT1S on LDBC100 and LiveJournal outperform nT1S by a larger margin. For example, 1T1S outperforms nT1S by a margin of 2.2x LDBC100 (2264ms vs 1213ms) vs 1.7x in the "few source" experiments. The behavior is similar for nTkS. In fact, as we expected, 1T1S and nTkS perform closer across the datasets compared to the few-source experiments. For example, on LiveJournal, nTkS outperforms 1T1S by a margin of only 1.1x (9363ms vs 8315ms) compared to the few-source experiments where the margin was 1.5x (2733ms vs 1865ms). Overall nTkS is

	nT1S (16 threads)	**nTkS** (16 threads, k = **16**)
IFE Call Time (in milliseconds)	50969	56445
LLC-Loads Throughput	86.007 Million / sec	71.59 Million / sec
LLC-Loads Misses	2,671,313,394	6,760,259,373 (2.53x)

Table 4.6: CPU Cache Statistics: `nT1S` vs `nTkS` Policy

again competitive or outperforms both nT1S and 1T1S.

The only behavior of the policies that deviates from our general expectations is that on Spotify, nT1S is competitive and even slightly faster than both 1T1S and nTkS. To better understand this behavior, we measured CPU cache statistics collected at runtime for the Spotify query with `nT1S` policy and `nTkS` policy: Table 4.6 shows CPU Cache statistics collected for the multi-source query with the 2 policies. `LLC` stands for **L**ast **L**evel **C**ache (also known as L3 cache), misses in this cache level cause memory (RAM) access to occur which is more expensive to access. All statistics were collected using the `perf stat -d` command on Linux, a performance analysis tool. We observe a direct correlation between the runtime difference of the policies (`nT1S` faster than `nTkS`) and performance statistics:

- `LLC-Load` throughput (total successful load instructions per second) for `nT1S` is higher compared to `nTkS`
- `LLC-Load` misses occurring for the query with `nTkS` policy is 2.53x more compared to `nT1S` policy

These statistics indicate that the `nTkS` policy with 16 concurrent IFE computations ($k = 16$) is becoming *cache-bound* leading to more misses in `LLC-Cache`. One reason for this is that with 16 concurrent IFEs, 16 concurrent frontiers are being scanned by 16 separate threads, where we cannot expect a locality in the scans they perform. However when all threads are working on the same frontier, threads grab frontier morsels consecutively, so they perform scans of similar regions. In Kúzu's buffer manager, which is based on the VM-cache technique [16], consecutive disk pages are also located consecutively in memory. In addition, although writes to the frontiers are not consecutive, they are to only 1 frontier in the case of nT1S, while they are to separate frontiers in nTkS (or 1T1S). These properties should lead to better cache performance for nT1S.

This raises the question of how nTkS behaves under different k values. In our next experiments, we compared the performance of nTkS on each dataset with k values set to 2, 4, 8, and 16. Table 4.7 shows our results. We see that the optimal value of k changes across the datasets, where on the two smaller datasets of LDBC100 and LiveJournal, increasing k brings better performance but the improvements are minor after k is equal to 4. In fact, the behavior of k=8 and 16 are very similar on these datasets. On the dense datasets, there is some slowdown for increasing k but the slowdown is again minor for k=4 or 8. In general, these experiments indicate that setting k=8 is a robust choice in our system.

	nTkS ($k = 2$)	**nTkS** ($k = 4$)	**nTkS** ($k = 8$)	**nTkS** ($k = 16$)
LDBC100	1908.33	1371.67 (1.4x)	1172.33 (1.6x)	1078 (1.8x)
Livejournal	12107.33	9729.67 (1.2x)	8641.66 (1.4x)	8315 (1.5x)
Spotify	47947	48292 (0.9x)	51754 (0.9x)	56445 (0.8x)

Table 4.7: nTkS Policy: varying $k = 2, 4, 8, 16$ (runtime in **milliseconds**)

Finally, we verified that the slowdown in the runtime when increasing the k value correlates with the CPU cache misses. Table 4.8 shows the statistics. With *($k = 2$)*, we observe the lowest number of `LLC-Load` misses and the best throughput. The execution time with this configuration (47947 ms) is faster than `nT1S` policy runtime (50969 ms). As the value of k increases from 2 to 16, the runtime also keeps increasing along with the `LLC-Load` misses. This shows that while having multiple concurrent running recursive computations may be a good idea for a multi-source (few) query when this number grows large (k is high), `nTkS` policy can become cache-bound leading to slower runtime.

	nTkS ($k = 2$)	**nTkS** ($k = 4$)	**nTkS** ($k = 8$)	**nTkS** ($k = 16$)
IFE Call Time (in milliseconds)	47947	48292	51754	56445
LLC-Loads Throughput	86.703 Million /sec	84.934 Million / sec	78.65 Million / sec	71.59 Million / sec
LLC-Load Misses	2,737,661,411	3,150,468,011	4,371,444,495	6,760,259,373

Table 4.8: CPU Cache Statistics: `nTkS` Policy (varying k)

Overall, these experiments indicate that setting k=4 or k=8 is a robust choice in our setting. These settings keep a good balance between maximum concurrent running recursive computations while also giving good runtime performance.

4.5 In-Memory Graph Index Experiments

So far, all of our experiments used an on-disk graph inside the IFE algorithms. Our next set of experiments aims to answer: When and by how much does using an in-memory graph index improve the performance of queries? Recall that when using an in-memory graph index, our system adds an additional index creation pipeline, similar to the hash join build pipelines of hash join algorithms. We also evaluate how well this pipeline parallelizes.

The primary advantage of using an in-memory index is that all scans of adjacency lists during the IFE algorithm computation bypass the buffer manager. The primary advantage of this is in fact not a reduction in I/O.[2] This is because assuming that the buffer manager has enough memory to store the index, which stores uncompressed versions of the adjacency lists, it should also be able to cache all of the disk pages that store the adjacency lists in compressed format. This is because Kúzu uses a disk-based CSR to store adjacency lists and our in-memory index implementation is also CSR-based.[3] Therefore we can expect the size of the disk image of the adjacency lists that are scanned by the on-disk graph to be commensurate or even smaller than the size of the in-memory graph index. Therefore we expect the buffer manager to also cache these in memory. However, accesses from the in-memory index should still be faster since it avoids any buffer manager code that ensures thread safety for concurrent access and decompresses the data from pages into the buffers of the disk-based `Graph`. Ultimately, we can expect performance improvements but it should be limited because there should not be visible I/O benefits. Further, we expect queries with multiple sources to benefit more from using the in-memory graph index as more IFE computations can benefit from the index, i.e., a larger fraction of the computation can benefit from the index and offset the cost incurred to create the index.

We repeated our 3 microbenchmark query workloads from the previous sections, this time using our in-memory index instead of the disk-based graph. We used the nTkS policy. Except for the first experiments on the index building performance, where we vary the number of worker threads, we use 16 threads and set k to 8 (due to reasons specified at the end of Section 4.4) in our experiments.

[2]In Kúzu, similar to other DBMSs that have vectorized query processors [1], all scans copy data from buffer manager to vectors. The disk-based `Graph` implementation provides an interface over these scanned vectors to the `IFEAlgorithm`.

[3]Further, because Kúzu is columnar, the adjacency lists are stored separately from any other properties on the edges (though our datasets do not have any edge properties).

Threads	LDBC100	Livejournal	Spotify
1	270	1876	15330
2	138 (1.9x)	1103 (1.7x)	7825 (1.9x)
4	64 (4.2x)	514 (3.7x)	3893 (3.9x)
8	40 (6.8x)	239 (7.9x)	2363 (6.5x)
16	29 (9.3x)	168 (11.2x)	1730 (8.9x)

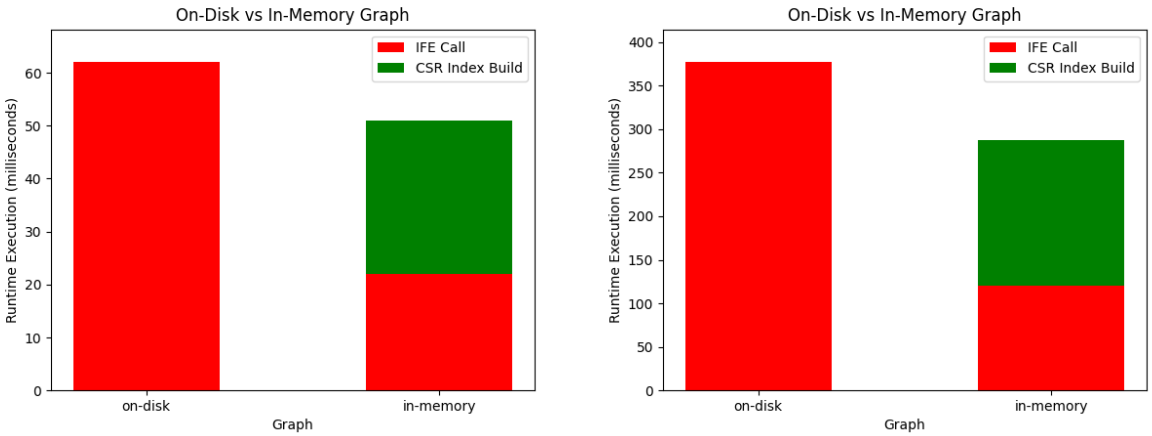
Table 4.9: CSR Index Build Time (in **milliseconds)**

We first present how well the index building pipeline parallelizes. Table 4.9 shows the runtime for index building time for the 3 datasets. We see that across all datasets The best runtime is achieved with **16 threads**. We observe good scalability across all datasets. When using 16 threads, the improvements are between 8.9 and 11.2x. This indicates that this pipeline does not have major scalability bottlenecks.

Next, we present our experiments using the single-source query workload. Figures 4.1 and 4.2 show the comparison for the single source query on LDBC100, Livejournal and Spotify datasets. For the in-memory index configuration, the figures report the index-building component separately from the time the `IFECall` pipeline takes. We see that as expected across all datasets, using an in-memory index improves the `IFECall` pipeline's runtime, between 1.7x and 3.1x. The end-to-end runtimes improve between 1.2x and 1.3x on LDBC100 and LiveJournal, but not on Spotify. Figures 4.3-4.6 present the results on the few-sources and many-sources workloads. As expected when queries include more sources, then using an in-memory graph starts outperforming using an on-disk graph between 2.3x-3.5x and consistently with increasing factors.

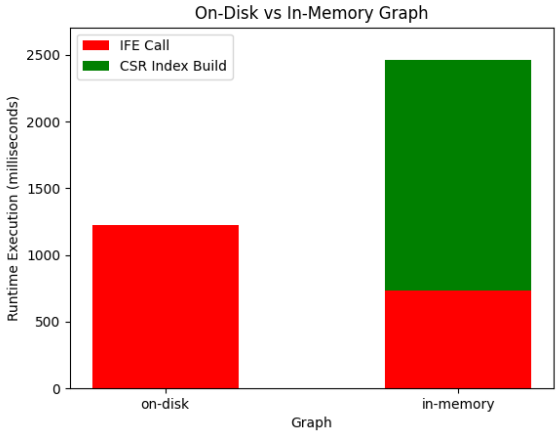
However, the number of sources is one factor that determines whether using the index will improve performance. In particular, the benefits of the in-memory graph are more visible on LDBC100 and LiveJournal compared to Spotify, where benefits are smaller. This is partly due to the fact that the index creation time takes relatively more time for Spotify compared to the cost of the `IFECall` operator. In addition, the relative benefits obtained even on the `IFECall` is also lower on Spotify. For example, on the many-source query workload, the `IFECall` pipeline improves respectively by 2.5x and 2.9x on LDBC100 and Livejournal, while it improves by 1.4x on Spotify. We profiled where CPU cycles are spent in the `IFECall` pipeline for each dataset.

Table 4.10 shows the CPU cycles % spent on Buffer Manager (**BM**) functions and performing the actual recursive computation (**IFECall**). We notice that for Livejournal and LDBC100, BM operations such as decompression when reading from a page and page



(a) LDBC100 (b) Livejournal

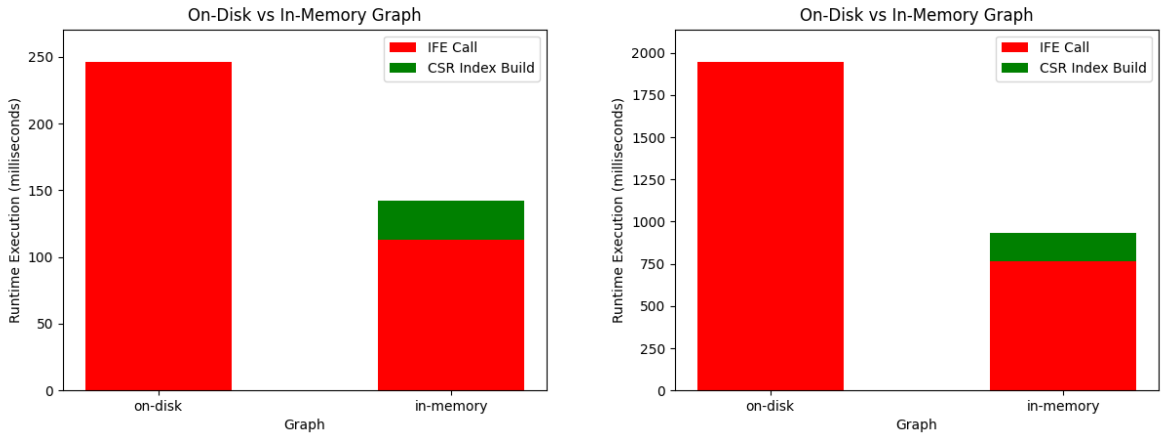
Figure 4.1: Single Source: On-Disk vs In-Memory Runtime



(a) Spotify

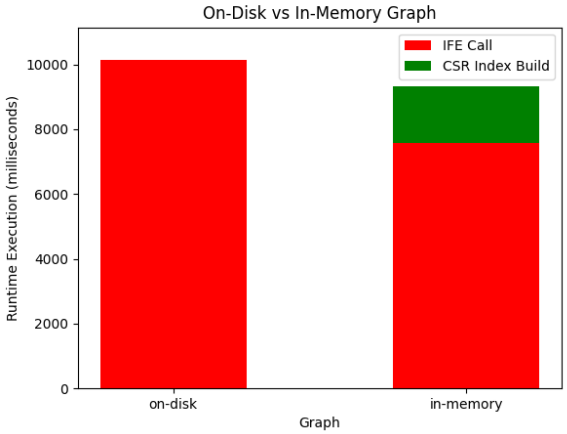
Figure 4.2: Single Source: On-Disk vs In-Memory Runtime

scan take significantly more CPU cycles compared to running the `IFECall` function. The decompression function takes the most cycles for both Livejournal and LDBC100 (indicated in red).
However, for Spotify, most CPU cycles are spent running the `IFECall` function (41% cycles) compared to decompression which takes only 18% cycles. This indicates that LDBC100



(a) LDBC100 (b) Livejournal

Figure 4.3: Multi-Source (few): Disk-based vs In-Memory Runtime

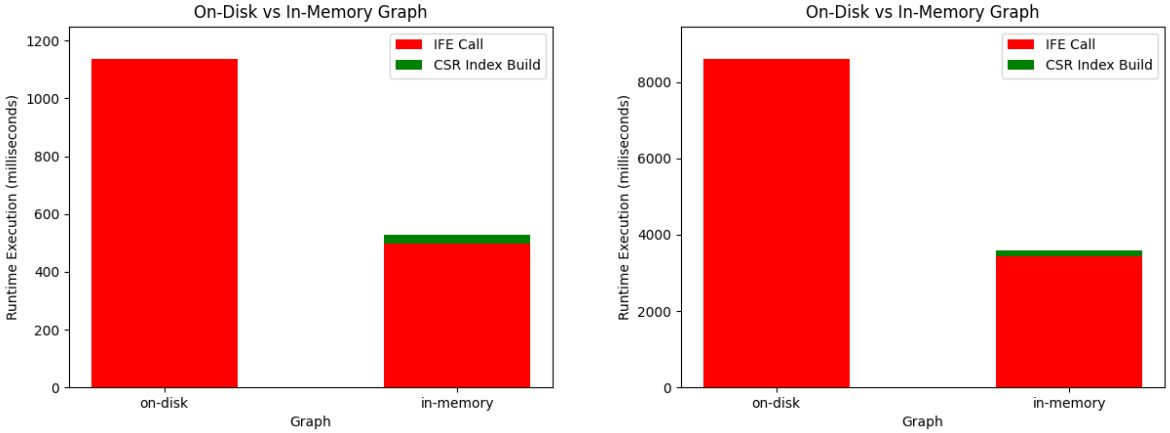


(a) Spotify

Figure 4.4: Multi-Source (few): On-Disk vs In-Memory Runtime

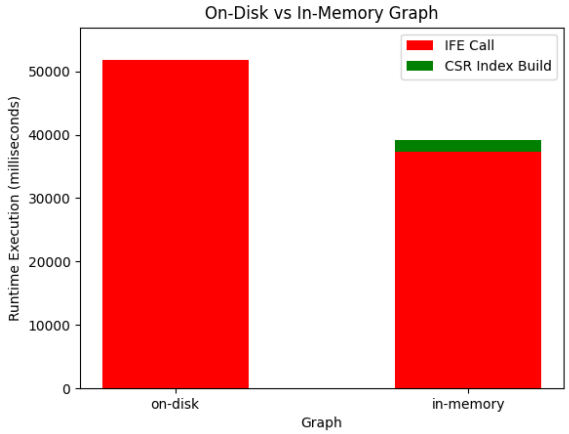
and Livejournal have a greater benefit in bypassing the Buffer Manager leading to faster overall execution time. For the Spotify dataset, since the bottleneck is not BM operations but the `IFECall`, using `InMemGraph` is not giving that much of a benefit.

This is likely due to the much higher average degree of Spotify (535 compared to 14 for LiveJournal and 44 for LDBC100). Decompression in the buffer manager is computation-



(a) LDBC100 (b) Livejournal

Figure 4.5: Multi-Source (many): On-Disk vs In-Memory Runtime



(a) Spotify

Figure 4.6: Multi-Source (many): On-Disk vs In-Memory Runtime

heavy but accesses sequential memory locations, so achieves good cache locality. However, the frontier extension work accesses random locations in memory. So the higher the average degree, we can expect more CPU cycles to be spent in frontier extension compared to decompression. Therefore the benefits of avoiding decompression, which is the primary benefit of bypassing the buffer manager, will lead to smaller performance improvements.

	BM: Decompress Page	**IFECall**: extendFrontier	**BM**: Page Scan
LDBC100	32	14	11
Livejournal	47	12	10
Spotify	18	41	22

Table 4.10: CPU Cycles % for Single Source (disk-based graph)

These experiments indicate that we can expect larger benefits from using an in-memory index on smaller and sparser graphs and on queries that require performing many IFE computations.

Chapter 5

Related Work

Morsel-driven parallelism is related to the seminal Volcano system's [10, 9] parallelism models. Volcano is the system that has invented the iterator pull-based model of query execution using `next()` or `getNextTuple()`-like functions. Many databases including Kúzu and Neo4j adopt this approach although systems may differ in how many tuples are passed between operators. Kúzu for example is a vectorized system and passes a vector (by default 2048) tuples[1]. Graefe has introduced both *vertical* and *horizontal* parallelism strategies within the context of the Volcano system. Vertical parallelism can break a single sub-plan into 2 parts where the operators in the upper operators can work independently "consuming" the data provided from the lower operators, which can independently "produce" more data. Horizontal parallelism refers to two separate approaches. First is *bushy parallelism*, which runs different sub-plans that do not depend on each other independently on separate threads at the same time. Second is *intra-operator parallelism*, which splits the input of an operator into multiple partitions. Morsel driven parallelism is an extension to intra-operator parallelism, where the inputs are partitioned in a much fine-grained manner than in the Volcano system (always at the leaves of the pipelines). There are also major implementation differences between modern systems that adopt morsel-driven parallelism and Volcano. For example, in Volcano, an exchange operator is placed in query plans to obtain parallelism. Parallel threads are spawned by exchange operators as the query plan is executing. This contrasts with the simpler designs in modern systems that maintain a pool of worker threads.

Our design is influenced by the vertex-centric parallelism approach of parallel or distributed graph analytics systems. Many such systems, such as Pregel [21], Ligra [32],

[1] A separate optimization called factorization can change the number of tuples passed [8]

GraphChi [15], GraphLab [19] have been based on this model of parallelism. These systems have vertex- or edge-centric APIs, such as a Pregel's `vertex.compute()` or Ligra's `edgeCompute()` functions, that assumes an IFE-based computation. Programmers implement these functions to describe the computation that should be done per vertex or edge. Then, the systems automatically parallelizes the executions of these functions using vertex-centric parallelism that parallelize work at each frontier. Out of these systems, our implementation is based on Ligra's API. These APIs have been used implement parallel versions of many recursive algorithms, including transitive closure, shortest paths and regular path queries [21, 32, 15, 25].

This approach is also integrated into the graph data science (GDS) libraries of GDBMSs as well. For example, Neo4j's GDS library contains some algorithms that implement IFE-based subroutines. In fact, Neo4j's recursive clauses in Cypher, such as Kleene star compile to `VarLengthExpand` operator, which is parallelized by Neo4j's vanilla morsel-driven parallelism approach. However, its shortest path algorithms only exist in its GDS library as a procedure call, which is parallelized using a vertex centric approach. Memgraph's GDS library called Mage contains algorithms that use OpenMP [27] to parallelize for loops in IFE-like algorithms. As such these systems either do vanilla morsel-driven or only vertex-centric parallelism, which can be suboptimal to our hybrid approach. Further, these implementations currently do not scale to out-of-memory large graph, and require work only on in-memory "projected" graphs, and use either OpenMP or separate thread pools than the pool used by the core DBMS. Our implementation of vertex-centric parallelism is more tightly integrated into the core DBMS, e.g., uses the thread pool of the DBMS, and our recursive algorithms seamlessly work on both disk-based and in-memory graph indices.

There are algorithms to evaluate recursive queries that are not based on an IFE subroutine. For example, Neo4j's GDS library contains procedure calls that implement shortest paths algorithms based on Dijkstra's algorithm. Similarly some algorithms are based on depth-first search-based algorithms. Some of these algorithms, such as Dijkstra's don't have obvious ways to parallelize using morsel-driven or other techniques, so are generally executed serially.

Finally, our work is also related to the multi source BFS (MS-BFS) algorithm [36], which is a technique to compute multiple BFS's concurrently and reduce the overall number of times a node is visited redundantly. It is implemented in DuckPGQ [35] system, which supports the SQL/PGQ standard [22] on top of the relational analytical DBMS DuckDB [28]. Our current implementation can be extended to also integrate this optimization, where 64 sources at a time can form an IFE computation and be scheduled as a morsel though we have not considered this approach within the scope of this thesis.

Chapter 6

Conclusions

Modern analytical DBMSs utilize morsel-driven parallelism for efficient parallel execution of queries. However, the vanilla design of morsel-driven parallelism is not suitable to handle recursive join queries since it is an expensive operation and may suffer from skew leading to slow execution time. In this thesis, we studied how to modify the original design of the morsel-driven execution framework to better accommodate the efficient scheduling of recursive join queries, based on the common IFE subroutine.

We explored a design space for parallelizing recursive join queries, which includes the *1T1S* policy, which mimics a version of the vanilla morsel-driven approach. *1T1S* schedules morsels to threads at the finest granular level possible, i.e., one source IFE computation at a time, from the base scans of the table that stores the sources from which an IFE computation should be performed. We then presented the *nT1S* policy, which mimics the parallelism done by parallel graph analytics libraries, which parallelize IFE computations at each iteration (or frontier). We then presented the hybrid scheduling approach *nTkS* that combines *1T1S* and *nT1S* approaches. Similar to *1T1S*, *nTkS* schedules multiple IFE computations at the same time yet similar to *nT1S* implements a mechanism where multiple threads can work on the same frontier of the same IFE computation. We presented experiments that demonstrate the robustness of *nTkS* on a set of micro-benchmark queries and how it is competitive with or outperforms the other two policies.

We also described the implementation details of our query processor design for recursive queries, which is based on a graph- and frontier-based abstraction instead of the rest of the query processor, which is based on a tuple-based abstraction. Our graph-based abstraction uses the tuple-based abstraction at several places, such as when scanning neighborhoods of nodes from disk during the recursive querying. We showed how this abstraction simplifies

implementing a version of our recursive query processing (IFE) algorithms based on an in-memory graph index. The core IFE algorithms work against a generic `Graph` abstraction. The disk-based version of the algorithms use a `Graph` implementation that uses the tuple-based scan operators underneath, while the in-memory version of the algorithms use a `Graph` implementation that scans an in-memory index. We also evaluated the pros and cons of using an in-memory graph index. All our implementations are integrated into Kúzu and we hope our description in this thesis forms a blueprint that other GDBMSs can also adopt.

References

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

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

[3] Khaled Ammar, Siddhartha Sahu, Semih Salihoglu, and M. Tamer Özsu. Optimizing differentially-maintained recursive queries on dynamic graphs. *Proceedings of the VLDB Endowment (PVLDB)*, 15(11):3186–3198, July 2022.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5), sep 2017.

[5] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[6] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1804–1815, aug 2015.

[7] World Wide Web Consortium. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/, jul 2024.

[8] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *Conference on Innovative Data Systems Research (CIDR)*, 2023.

[9] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

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

[11] Neo4j Inc. Graph Data Science Library. https://neo4j.com/docs/graph-data-science/current/introduction/, jul 2024.

[12] Neo4j Inc. Neo4j Graph Database. https://neo4j.com/, 2024. [Online; accessed 17-July-2024].

[13] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp and olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, 2011.

[14] Raunak Kumar, Paul Liu, Moses Charikar, and Austin R. Benson. Retrieving top weighted triangles in graphs. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, 2020.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[16] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proceedings of the ACM on Management of Data*, 1(1), may 2023.

[17] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[18] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, 2012.

[19] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment (PVLDB)*, 5(8):716–727, apr 2012.

[20] Memgraph Ltd. Memgraph Advanced Graph Extensions. https://github.com/memgraph/mage, jul 2024.

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[22] ISO/IEC JTC 1/SC 32 Data management and interchange. Sql/pgq. *ISO International Standard*, 2023.

[23] Inc. Neo4j. OpenCypher. https://opencypher.org/, aug 2024.

[24] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *Conference on Innovative Data Systems Research*, 2020.

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

[26] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, sep 2016.

[27] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[28] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[29] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theor. Comp. Sys.*, 61(1):31–83, jul 2017.

[30] Semih Salihoglu and Jennifer Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013.

[31] Markus L. Schmid. Conjunctive regular path queries with capture groups. *ACM Trans. Database Syst.*, 47(2), may 2022.

[32] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013.

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

[34] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. The ldbc social network benchmark: Business intelligence workload. *Proceedings of the VLDB Endowment (PVLDB)*, 16(4):877–890, dec 2022.

[35] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 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. The more the merrier: efficient multi-source graph traversal. *Proceedings of the VLDB Endowment (PVLDB)*, 8(4):449–460, dec 2014.

[37] Bryan B. Thompson, Michael Personick, and Martyn Cutcher. The bigdata® rdf graph database. In *Linked Data Management*, 2014.

[38] Yanghua Xiao and Bin Shao. Billion-node graph challenges. In *IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 40 No. 3*, 2017.