aDFS: An Almost Depth-First-Search Distributed Graph-Querying System

Abstract

Graph processing is an invaluable tool for data analytics. In particular, pattern-matching queries enable flexible graph exploration and analysis, similar to what SQL provides for relational databases. Focusing on following connections in the data, graph queries are a challenging workload because even seemingly trivial queries can easily produce billions of intermediate results and irregular data access patterns.

In this paper, we introduce aDFS: The first distributed graph-querying system that can process practically any query fully in memory, while maintaining fixed runtime memory consumption. To achieve this behavior, aDFS relies on (i) almost depth-first (aDFS) graph exploration with some breadth-first characteristics for performance, and (ii) non-blocking dispatching of intermediate results to remote edges. We evaluate aDFS both against state-of-the-art graph-querying (Neo4J and GraphFrames for Apache Spark) and graph-mining (G-Miner, Fractal, and Peregrine) engines and show that aDFS significantly outperforms the rest on a plethora of workloads, spanning from traditional graph queries to graph mining.

1. Introduction

Graph pattern-matching provides an interface for interactive exploration of graphs, similar to SQL for relational databases. It focuses on data connections, i.e., edges, allowing users to submit queries with any pattern, filter, and projection. For instance, the following PGQL [4] query:

```
SELECT a1.name, a2.name, a1.country = a2.country,
       ABS(a1.salary - a2.salary) AS salary_diff
MATCH (a1:author)-[:likes]->(a2:author),
      (a2)-[:likes]->(a1)
WHERE ABS(a1.age - a2.age) <= 10
ORDER BY salary_diff DESC
```

enumerates the `authors` of a similar age that `like` each other. Answering such a query requires to find all homomorphic matches of the query pattern in the target graph, while enforcing the given filters (e.g., `a1 IS author`) and projecting the requested output (e.g., whether `a1.country = a2.country`). Graph queries are a key tool for graph analysis, as indicated by the large number of existing graph-querying systems [17, 8, 41, 19, 46, 79, 43, 63] and languages [4, 42, 7, 6, 2].

Graph processing in general is a very active area of research. There is a plethora of graph systems for classic graph algorithms [25, 38, 54, 22, 33, 50, 23, 53, 75, 78] and for graph mining [67, 21, 14, 61, 15, 70, 18, 28, 39]. Contrary to interactive queries, graph algorithms (such as e.g., PageRank [48]) are typically used in batch computations. Graph mining includes simple examples of graph pattern matching, enabling the search of isomorphic patterns in the graph. However, it does not provide the expressive SQL-like interface with rich dynamic projection and filtering support that graph queries call for (see Section 5 for further details).

The dynamic user-defined patterns, filters, and projections, the focus on edges, and the homomorphic matching (i.e., finding *all matching permutations* of the query pattern) make graph query execution a challenging workload that needs to handle very large intermediate and final result sets, with a combinatorial explosion effect. For example, on the well-researched Twitter graph [32], the single-edge query `(a)->(b)` matches the whole graph, amounting to 1.4 billion results, and the two-edge query `(a)->(b)->(c)` amounts to 9.3 trillion matches—which means matching the `(a)->(b)->(c)->(a)` cycle needs to consider 9.3 trillion intermediate results. Compared to relational queries, queries on graphs can exhibit extremely irregular access patterns [55, 36] and lack of spatial locality, calling for low-latency data access. For these reasons, high-performance graph-querying engines ideally need to (i) control the memory requirements to avoid explosion, while (ii) keeping the computation in main memory and scaling out to a distributed system in order to handle graphs and queries that exceed the capacity of a single node.

Query execution on graphs is typically based on one of the two classic graph-traversal strategies: Either breadth-first search (BFS) or depth-first search (DFS). Both BFS and DFS have major advantages and drawbacks for distributed graph queries: BFS traversals are easier to parallelize but, as with distributed joins, suffer from explosion in the size of intermediate results, cannot be easily pipelined, and stress the network bandwidth to shuffle data across levels of pattern matching. DFS traversals reduce the size of intermediate results, but are challenging to parallelize and result in random data access patterns, wasting locality when iterating over neighbors.

In this paper, we introduce aDFS (almost-DFS): The first distributed graph-querying system that brings the best of both DFS/BFS worlds. aDFS processes graphs partitioned across multiple machines *fully in memory*, and combines BFS and DFS traversals to *bound the maximum amount of memory* required for query execution, and to achieve a *high degree of parallelism*. DFS, together with a distributed flow-control mechanism, guarantee that the amount of runtime memory remains within limits, while the BFS exploration allows for better locality and parallelization during execution.

Worker threads in aDFS mainly prioritize DFS execution for completing—and thus freeing—intermediate results. Execution switches to BFS when matching a remote edge (i.e., an edge pointing to a remote machine) or when the runtime detects that the query contains limited parallelism (i.e., a small

set of intermediate results). To elaborate, for local edges, worker threads perform DFS, unless aDFS detects that there is a limited amount of available work on the local machine, in which case they switch to per-thread BFS exploration until there is enough parallelism. For remote edges, threads buffer the matched intermediate results and continue with matching the next edge in a BFS manner (i.e., the next edge is possibly at the same depth as the current one). Once a buffer is full, the worker thread sends the message to the target machine, unless sending the message is blocked by the flow-control mechanism, which enforces the target memory limits. Section 3 expands on the design and implementation of the aDFS engine.

Section 4 thoroughly evaluates aDFS and shows that it is capable of executing trillion-scale queries, with a 10GB per-machine runtime memory cap. In our largest query, aDFS computes a 9.3 trillion count pattern on the Twitter graph with a rate of 7.3 billion matches per second. We compare aDFS to two graph engines (i.e., Apache Spark GraphFrames [17] and Neo4j [43]) and two relational database engines (i.e., MonetDB [3] and PostgreSQL [5]) using the LDBC graph and query suite [60]. aDFS completes the set of queries 43 and 53 times faster than GraphFrames and Neo4j[1] respectively, and 8 and 26 times faster than MonetDB and PostgreSQL (as Section 4.2 shows, LDBC is "relational-friendly"). We also compare aDFS to these four systems with schema-less graphs and show that either aDFS is 16 to 9,200 times faster than the rest, or the other systems simply fail to complete the queries. Finally, we compare aDFS with three state-of-the-art graph-mining systems, namely G-Miner [14], Fractal [18] and Peregrine [28], and show that aDFS is up to 12, 625, and 18 times faster, respectively, on mining-oriented workloads. We discuss related work further in Section 5.

The main contributions of this paper are the following:

- The aDFS distributed graph-querying system. To the best of our knowledge, aDFS is the first query engine that fully distributes computation over partitioned graphs and bounds runtime memory;
- The novel combination of DFS (for eager completion of intermediate results), BFS (for performance), and flow control (for controlling the size of intermediate state) to achieve performance and scalability while capping memory usage; and
- The evaluation of aDFS, which shows that aDFS significantly outperforms the state of the art and is capable of executing queries with trillions of matches.

In this paper, we enable efficient fixed pattern-matching queries, which constitute the backbone of PGQL 1.1 [4]. Designing more PGQL patterns, such as regular path queries, shortest paths, and sub-queries, is outside the scope of this paper and left for future work.

2. Background and Motivation

Representing data as a graph is becoming increasingly popular. The main advantage of graphs is that they focus on modeling fine-grained relationships between entities. In contrast, the relational model concentrates more on data and relies on the heavyweight primary-key foreign-key (PK-FK) and join mechanisms to link entities. However, when using graphs, different models, data representations, and ways of exploring them have a major effect on performance in the context of graph processing and, in particular, querying.

2.1. The Property Graph (PG) Model

Property graphs represent the graph topology as vertices and edges, and store *properties* and *labels* separately. Properties can be associated to any vertex or edge and take the form of typed key-value pairs. Labels are key-only and represent types or categories, e.g., *person* or *animal*. Separating the topology from properties avoids the proliferation of edges and allows for quick traversals of the graph over its real structure. As we detail in Section 5, although we focus in this paper on PG graphs, our solutions could be potentially applied to alternative models, such as RDF [11].

2.2. Graph Pattern-Matching Queries

Several languages for graph querying exist, such as PGQL [4], SPARQL [6], Gremlin [7], and Cypher [42]. In its simplest form, graph querying makes it possible to find patterns in graphs, with filters and projections. aDFS uses PGQL, which is modeled after SQL: Projection and aggregation operations are the same as SQL, including `GROUP BY` and `ORDER BY`. PGQL adds support for graph patterns and vertex and edge labels. For example, the query presented in Section 1 adds the `MATCH` clause to an otherwise valid SQL query. It matches patterns that are homomorphic[2] to the `(a1)->(a2)->(a1)` cycle, while enforcing filters (e.g., `a1` has label `author`) and it projects or aggregates the requested data—including even arbitrary expressions—out of the matched vertices and edges.

2.3. Graphs vs. Relational Joins and RDF

In PG graph engines, edges are stored explicitly and can be traversed directly. In contrast, in relational databases, relationships are represented with PK-FK. Note that the de-facto implementation of RDF triplets results in similar PK-FK behavior. Following any relationship means joining two tables—or doing a self-join if the keys belong to the same table—and producing the intermediate result. Therefore, while matching multiple-hop paths is a relatively cheap operation in graph engines since it only needs to "follow some pointers," doing the same thing in SQL requires a chain of multiple expensive join operations that materialize intermediate result sets each time. Consequently, graph engines can be much more efficient

[1] Using Neo4j Community Edition (benchmarks not audited by Neo4j).

[2] Graph queries require *homomorphic matching*, as data is projected out of all matches, even if they are permutations of each other. In our example query, if $author_a$ and $author_b$ like each other and the filters are satisfied, the result will include both rows with $author_a$ as variable `a1` and as variable `a2`. In contrast, *isomorphic matching* would return a single match. Homomorphic matching returns at least as many results as isomorphic matching, hence early-pruning techniques used for isomorphic matching are not applicable to graph queries. In a graph query, isomorphic matching can be simulated with filters or with query specifiers, such as `GROUP BY`.

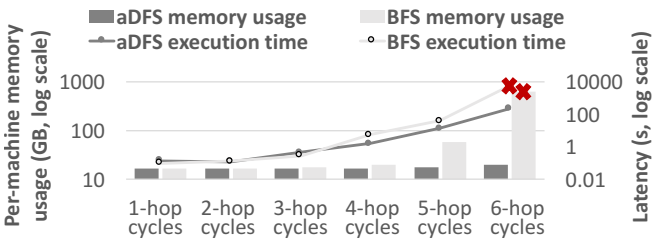

Figure 1: Matching cycles using aDFS vs. BFS.

than relational databases when it comes to matching graph patterns (see Section 4 for a comparison).

2.4. DFS/BFS for Graph Exploration

DFS can expand one intermediate result at a time, starting from the first variable in the pattern and continuing to the next until the whole pattern is matched. However, this behavior results in totally random accesses and is impractical for distributed graph traversals: The only way to continue with strict DFS is to directly send the intermediate result to the remote machine and wait until it is picked up and completed.

Thus, graph exploration is traditionally done using BFS: For each query edge (hop), the entire result set is computed, and only then does the exploration of the next hop start. This approach has two main advantages: (i) it is easy to implement, as work is naturally divided into simple steps (hops), and (ii) it is relatively easy to parallelize, as the entire input is known before processing a hop (of course, skewed vertex degrees still pose a problem). However, BFS has one major shortcoming: Because the intermediate result set is produced between stages, an intermediate result-set explosion can quickly occur.

Figure 1 illustrates this issue showing the average total memory usage of machines and execution time when matching cycles of various lengths using aDFS and BFS (implemented in our runtime) on a small graph [1] (875K vertices and 5.1M edges). While both approaches are able to match cycles of length one to four with similar performance, the memory consumption of BFS explodes for five-hop cycles at approximately 60GB on each of the eight machines used for the experiment, and BFS crashes with six-hop cycles after 96 minutes when one machine runs out of memory (~768GB). Meanwhile, the memory consumption of aDFS is almost constant.

3. aDFS: A Pattern Matching and Querying System for Distributed Graphs

The main design goals of aDFS are (i) enabling fast, fully in-memory distributed queries of any size, while (ii) allowing for limited, controllable memory consumption during execution. The rationale for these two goals is as follows. First, high-performance graph queries demand in-memory execution and the ever-increasing size of data calls for distribution. Second, server systems, especially in cloud deployments, are shared by multiple concurrent users, hence no single query can be permitted to saturate the system memory. aDFS achieves these two goals through the following design principles.

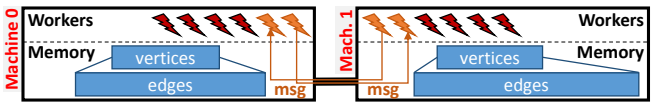

Figure 2: High-level architecture of aDFS.

1. **§3.3: DFS-first and asynchronous communication.** The eager match completion of DFS gives aDFS fine-grained control on the size of intermediate results during query execution, but strict DFS would be inefficient when matching a remote edge, i.e., an edge that leads to a remote machine. For that reason, worker threads do not block when encountering a remote edge, but place the intermediate result in a message buffer and continue with other local work instead. The buffer batches the intermediate results and, once full, it is asynchronously sent to the remote machine for further processing. Threads only need to block if flow control dictates so.
2. **§3.4: Flow control.** Cross-machine communication is controlled through a flow-control mechanism that caps the number of in-flight intermediate result buffers. The finite nature of these message buffers allows configuring the amount of runtime memory that aDFS requires, while the flow-control mechanism guarantees query termination and deadlock freedom.
3. **§3.5: Dynamic DFS/BFS balance.** Besides the BFS style of buffering for remote edges, aDFS includes a dynamic approach for deciding whether to go DFS or expand with BFS for local matches in order to improve parallelism, locality, and work sharing across threads.

Before diving into these design principles, we first present the architecture of aDFS from a high-level point of view (see Section 3.1) and describe how aDFS generates execution plans for graph queries (see Section 3.2).

3.1. High-Level aDFS Architecture

Figure 2 shows the high-level architecture of aDFS. Graphs are kept in memory and are partitioned across machines based on simple random vertex partitioning. For efficient traversals, graphs are stored in the classic CSR (Compressed Sparse Row) graph format. Due to graph partitioning, messaging is necessary for moving intermediate results to the machine which holds the target vertex. aDFS maintains two threads on dedicated cores on each machine for messaging; a sender and a receiver. Consequently, worker threads in aDFS place their messages in software queues, from where they are picked up by the sender. That way, aDFS also supports zero-copy messaging over InfiniBand: Workers can use a set of pre-allocated buffers, directly registered with the network card.

3.2. Distributed Query Execution Planning

Users submit declarative PGQL queries [4] to aDFS. As Figure 3 illustrates, each query goes through three transformation steps (marked i through iii) before being executed in step iv.

Step i: Logical query planner. The first step translates the PGQL query into a logical query plan, which consists of the

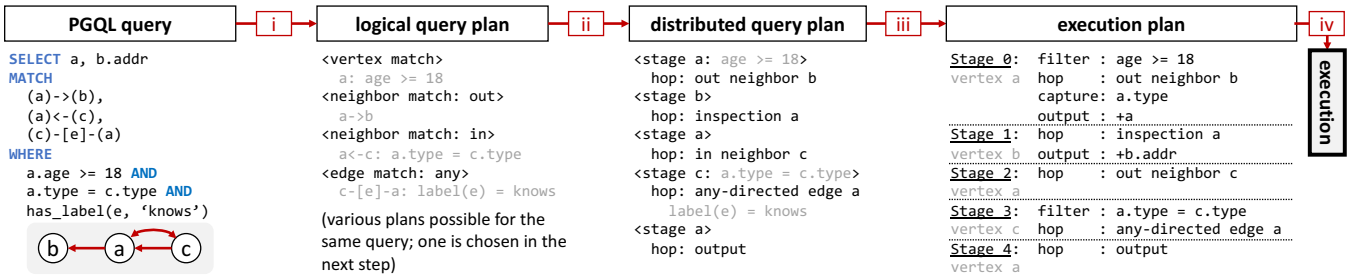

Figure 3: From a PGQL query to aDFS execution. Three transformation steps before execution.

logical operators of Table 1. Similar to relational query planning, a given query can be executed by multiple logical query plans. In the example of Figure 3, an alternative plan could rewrite the query as `(a)-[e]-(c)->(a)->(b)`. This first step directly translates the query to an admissible plan, which is then optimized in the following steps.

Step ii: Distributed query planner and optimizer. This step specializes the logical query plan by taking into account the specific characteristics of aDFS's runtime. The query planner rewrites the logical plan in terms of *stages* and transitions from one stage to another (called *hops*). A stage is responsible for matching or accessing exactly one vertex and contains all the information necessary for matching the corresponding vertex and for transitioning to the next vertex with a hop. In the example of Figure 3, the topmost stage "a" matches the first vertex `a` of the query, while the next one matches `b`. An out-neighbor hop takes the execution from `a` to `b`.

aDFS supports four types of hops that specialize for distributed execution: *neighbor match*, *edge match*, *output*, and *inspection*. Neighbor and edge hops have the same behavior as the corresponding logical operators in Table 1. An *output* hop produces a final match using the current intermediate result and is always used in the last stage of a match.

Inspection hops are specific to distributed processing: They bring the current intermediate result back to an already matched vertex in order to continue query evaluation. In the example of Figure 3, after matching `a` and `b` of `(a)->(b)`, the query requires again the neighbor list of the already matched vertex `a` in order to continue with matching `(a)<-(c)`. Since the matched vertex `b` might be in a different machine than `a`, the query planner introduces an inspection step to "link" this disconnected pattern and bring back the context to the machine of `a`. If `a` resides in the current machine, an inspection hop is essentially a no-op.

In this step, aDFS rewrites the logical query plan with a cost-based optimizer implemented with dynamic programming and based on the following heuristics: (i) heavily filtered vertices are preferred for the earlier stages of the plan, (ii) inspection hops are not free and increase the plan's cost, and (iii) the cost of an edge hop is approximately *log* of the cost of a neighbor hop, as it can be implemented with a binary search in the neighbor list of the source vertex.

Op.	Example	Short description
vertex match	`(x)`	Match vertices of the graph (without following any edge)
neighbor match	`(x)->(y)`	Having matched the left vertex `x`, match its neighbors `y`. Can be in-, out-, or any-directional
edge match	`(x)->...` `(y)->(x)`	Vertex `x` is known (already visited). Test if `x` exists in the neighbor list of the left vertex `y`. Can be in-, out-, or any-directional

Table 1: Graph operators used in the logical query plan.

Steps iii–iv: Execution plan and execution. Finally, the aDFS engine generates a concrete execution plan. Apart from stages and hops, the execution plan contains filters (on vertices and edges), as well as information on what data should be included in the intermediate results in order to execute filters of later stages and produce the final output. For example, in the query of Figure 3, Stage 0 must collect `a.type`, since it is required by the filter of Stage 3. Similarly, Stage 0 must put vertex `a` in the intermediate result as it is part of the projection of the query. Overall, each stage builds up the intermediate result such that another thread, local or remote, can pick it up and continue the computation. The resulting execution plan is then submitted to the aDFS runtime, on which we focus next.

3.3. aDFS's Depth-First Runtime

The runtime of aDFS is based on the *stage* and *hop* constructs, described above. aDFS initiates the query by applying Stage 0 (matching of the first vertex variable of the execution plan) to each vertex of the graph. This bootstrapping process happens across machines, i.e., each machine starts from the locally-stored vertices, and in parallel within each machine, i.e., each worker thread handles a distinct set of vertices and performs the bootstrapping process on these vertices one after the other. Hops that follow remote edges send the intermediate match (batched) to the destination remote machine where they are picked up and continued by a local thread.

Bootstrapping a match. Figure 4 includes a high-level activity diagram of the aDFS runtime. Completing the execution of this diagram from Stage 0 to the last query stage implements the complete matching starting from a single vertex of the graph. We explain these steps using the example of Figure 5. Text in the *blue italic face* represents the activities in Figure 4. The aDFS runtime assigns vertex `Joe` (the gray rounded rectangle of Figure 5) to a worker thread `t`, which tries to generate new matches. The thread first tries to match `Joe` with Stage 0's `p1` using "*apply stage*." If the filter `p1.name = "Joe"` returns false, the thread would try to *backtrack* to a previous

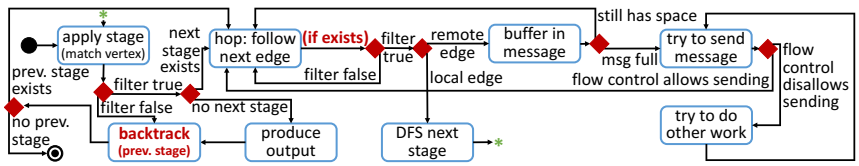

Figure 4: Matching operations starting from a given vertex. The backtrack activity represents returning to the previous DFS stage. Similarly, if the conditional in the red bold font returns false, the execution backtracks to the previous stage (if any).

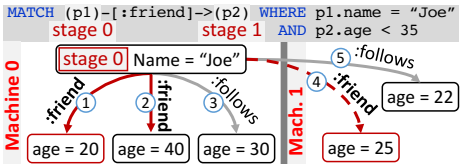

Figure 5: Example graph query execution. Rounded rectangles represent vertices, red vertices and edges are matched.

stage and, because there is none, it would simply complete this invocation. If there are more top-level vertices to explore, `t` would start again with a different vertex.

In the example of Figure 5, we assume that the execution plan matches vertex `p1` as Stage 0. `p1` matches `Joe` and `t` continues with the *hop: follow next edge* operation, starting from edge ①. Since the `:friend` label filter is satisfied and the edge is local, `t` proceeds via *DFS next stage* to Stage 1 where `p2` is matched with the vertex with `age = 20`. Now, since `p2.age < 35` is satisfied and there is no next stage, `t` produces a query output row and *backtracks* to Stage 0 to continue with the next edge. At this point, the `:friend` edge ② is followed, but the filter is not satisfied; that vertex has `age = 40`. Backtracking to Stage 0 brings us to edge ③ with label `:follows`, which is not matched.

Thread `t` is now done with local edges and starts processing the remote ones (aDFS does not necessarily match all local edges first). The first one, edge ④, has label `:friend`, thus `t` places the current intermediate result in a messaging buffer targeting Machine 1 (*buffer in message*). Once the buffer fills up, `t` *tries to send the message* to the destination. As Section 3.4 describes in more detail, flow control might temporarily block `t` from sending the message; in that case, `t` *tries to do some other work* (e.g., handling an incoming message). Once the thread returns from performing these other tasks, it retries sending the blocked message. Finally, `t` attempts to match the last remote edge ⑤, which does not match because of its label. With all the edges of vertex `Joe` explored and no previous stage to backtrack to, `t` completes this invocation.

Handling incoming messages (intermediate results). Workers eagerly try to receive and process remote messages, always prioritizing the latest stage with available work. Threads try to process messages: (i) before starting new work, i.e., before *apply stage* at Stage 0 (new top-level vertex), (ii) when flow control (temporarily) disallows message sending—in this case, the impacted thread picks up a new message to process while waiting for flow control to release the blocked message, and (iii) once the matching operations (see Figure 4) have completed on all local vertices—at that point, workers continuously wait for incoming messages to complete any pending work from remote machines.

3.4. Flow Control

aDFS allows specifying the total memory size `M` of the messaging buffers to hold the intermediate results in any machine, making it possible to cap runtime memory utilization. Besides these buffers, aDFS only needs a small per-thread, per-stage, additional memory allocation to hold the current ongoing local match and metadata for thread blocking.

In order to enforce this memory cap, aDFS employs a simple flow-control protocol. aDFS partitions the intermediate-result buffers across the query stages, such that no stage can consume all buffers (required for deadlock freedom). When a buffer with intermediate results is full, the corresponding worker requests permission to send the buffer to the target machine. The flow-control protocol keeps track of the amount of data `D` that has been sent to that machine but not yet processed. If `D` is above a threshold (computed based on the memory cap `M`; a machine does not accept more than `M/#Machines` worth of intermediate results from any other), flow control blocks the message transmission (controlled per stage, not for the whole query) and the thread continues with some other work before retrying to send the message. Once a message has been processed, the handling thread informs the source machine that this chunk of intermediate results has been completed and makes this memory available for another message.

Flow-control performance. Figure 6 compares the query execution latency without flow control (i.e., all messages of intermediate results are sent as they are produced) to different per-machine flow-control limits in aDFS (the graphs are described in Table 2). In this experiment, we use a buffer size of 256KB and eight machines; see Section 4 for details on the experimental setup. The per-machine limit N is the total number of outgoing buffers that this query execution is allowed to have, therefore it also dictates the maximum amount of memory M that a machine can utilize during the execution of that query. Since all intermediate results could be targeting a single machine at some point during execution, $M = N \times$ (size of one buffer) $\times$ (number of machines).

The queries we execute are simple `SELECT COUNT(*)` and include basic patterns such as `(a)->(b)->(a)` (Q1 and Q2) and `(a)->(b)->(c)` (Q3 to Q6), with different filters. The results show that aDFS is not very sensitive to the flow-control limit, unless the limit is very low, e.g., 512 messaging buffers. In that case, the flow control only allows a single outstanding message per worker, per stage, per machine.

Figure 7 gives more insights in the execution of Q3 with Livejournal: `SELECT COUNT(*) MATCH (a)->(b)->(c)`. The figure shows the maximum number of incoming and outgoing messages for the busiest stage on any of the eight machines, as well as the number of cases where the flow-control

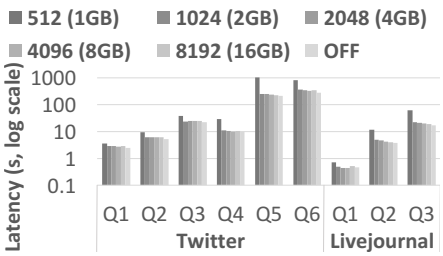

Figure 6: Performance of simple queries (8 machines) with different flow-control limits. In parentheses: Total per-machine max. memory consumption.

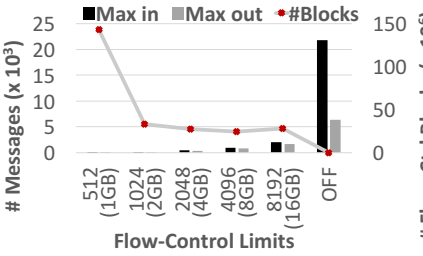

Figure 7: Messaging and blocking statistics on Q3 Livejournal with different flow-control limits. In parentheses: Total per-machine max. memory consumption.

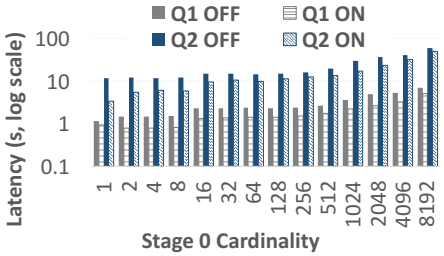

Figure 8: aDFS with dynamic local-edge BFS "ON" or "OFF" for two queries while varying the number of intermediate results of the first query stage.

limits were reached. For very low limits ($N = 512$ messages) the amount of blocking is very high, thus penalizing performance (more than 3× higher latency). Still, the overhead for switching stages due to flow control is generally low: Setting N to 8,192 results in only ~10% performance loss compared to no flow-control (`OFF`), while reaching 10 times fewer maximum incoming messages (2,087 vs. 21,793) and four-times fewer outgoing messages (1,636 vs. 6,430).

3.5. Dynamic BFS for Local Edges in aDFS

For remote edges, aDFS essentially does (per-thread) BFS: A thread matching a remote edge simply buffers the intermediate result and continues exploring and matching the same stage, which might produce new intermediate results.

While local processing could happen in pure DFS, doing so can result in artificially limited parallelism for queries that produce small sets of intermediate results. A characteristic example is queries with a very narrow starting Stage 0, such as `MATCH (a)->... WHERE ID(a) = X`; this narrow-start behavior appears in several real-life queries (e.g., the LDBC queries of Section 4). In such a query, the whole Stage 0 might produce a single intermediate result, giving limited opportunities for parallelism. For these workloads, DFS can significantly delay the expansion of intermediate results that are produced in the system (both locally and through messages).

In aDFS, we solve this DFS limitation by dynamically switching depth-first exploration to per-thread breadth-first for local edges. aDFS maintains per-stage statistics indicating how many buffers of intermediate results are available for threads to undertake. A low number of intermediate results means that the stage has not expanded enough, hence some threads could end up not having sufficient work to perform. When threads in aDFS are processing a local edge, they use this information to decide whether to perform DFS or BFS, i.e., buffer the intermediate result in a local buffer and continue at the same stage.

In practice, we keep these local buffers small, i.e., up to a few kilobytes, in order to promote quick local work creation. We further use a *DFS threshold* to decide when to work depth-first: When the sum of the number of local buffers (produced by the breadth-first expansion) plus the number of message buffers from remote machines is greater than 2× or 4× the number of threads, threads switch to DFS. Having a low threshold plus small local buffers allows aDFS to keep the maximum additional memory consumption limited: If the DFS threshold is set to n, the maximum number of threads is t, the size of local buffers is b, and the query contains s stages, the maximum additional memory in a machine is $(n+t)*(s-1)*b$. In the configuration used for our experiments ($t = 28$, $n = 4t = 128$, $s \leq 11$, and $b = 8{,}192$), local buffers consume less than 12MB additional memory.

Controlled Experiment. Figure 8 illustrates the benefits of this local-match BFS mode with the following two queries:

```
1:(a)->()->() WHERE ID(a)<$i
2:(a)->()-[e]->()->() WHERE e.cost<0.5 AND ID(a)<$i
```

using the Twitter graph extended with a uniform random edge property with values in $[0.0, 100.0)$. In both queries, the `ID(a)<$i` filter configures the cardinality of the first query stage, translating to a potentially narrow starting point. In Query 2, the edge filter also guarantees that the third stage includes a small number of intermediate results. The dynamicity of aDFS brings significant performance benefits, especially for queries with very narrow starting points. For example, for Q1 with $i = 1$, Machine 0 hosts the match for Stage 0; without the breadth-first mode ("OFF"), a single thread handles all the 55K local edges which lead to Stage 1. In contrast, enabling dynamic local BFS ("ON") generates more work early on and allows splitting the work among local threads, each of which operates on approximately 2,000 vertices for Stage 1.

LDBC Q20. We also briefly analyze the BFS-mode benefits on LDBC Q20 (see Section 4 for more details):

```
MATCH (tC:tagClass)<-[:subClassOf]-(:tagClass)
  <-[:hasType]-(:tag)<-[:hasTag]-(:post|comment)
WHERE tC.name IN ('Politics', 'Art', 'Country')
```

In this query, the first two stages match `tagClass`es and Stage 0 results in only three intermediate results due to the filter. The local BFS optimization brings 32% latency benefits (8 vs. 5.5 seconds), by better paralellizing the work across threads. Without the optimization, the most busy thread, i.e., the one that "gets stuck" in performing local DFS work the most, spends 4 seconds in these local explorations: It matches about 1,000 vertices in Stage 1, which result in 5.2 million local matches in Stage 2 and 5 million in Stage 3. In comparison, with the optimization, the most busy thread spends only 1.6 seconds in DFS work: It handles 4 million local

Graph	#V	#E	Schema	Description
Livejournal [10]	484K	68.9M	no	Users and friendships
URandom	100M	1B	no	Uniform random edges
Twitter [32]	42.6M	1.47B	no	Tweets and followers
LDBC(100) [60]	283M	1.78B	yes	LDBC social
Webgraph-UK [12]	77.7M	2.97B	no	2006 .uk domains

Table 2: The set of graphs we use for evaluation.

edges in Stage 2, which it successfully distributes to other threads with approximately 500 local BFS buffers. Overall, enabling dynamic local BFS provides significant speedup on realistic workloads, while incurring at most 5% slowdown in pathological cases.

4. Evaluation

The goal of our evaluation is to understand how well aDFS performs as compared against other engines (graph, relational or mining) that could be used in similar use cases, and how aDFS scales as we increase the number of machines.

4.1. Experimental Settings

Hardware details. We use a cluster of eight nodes, each with two sockets of Intel Xeon E-2690 v4 at 2.60GHz, with 14 cores (hyperthreads disabled/DVFS enabled), for 28 cores in total. Each processor contains 756GB of DDR4-2400 memory and LSI MegaRAID SAS-3 3108 storage. Each node includes a Mellanox Connect-X InfiniBand card, all connected to an EDR InfiniBand network (100Gbit/s).

Graphs and queries. Unless specified otherwise, we use the five graphs of Table 2 for our experiments. The scope of this paper covers user-provided fixed-pattern queries, thus aDFS implements only a subset of PGQL 1.1 and does not support constructs such as subqueries. Accordingly, we use the 12 LDBC Business Intelligence (BI) standard queries [60] supported by PGQL 1.1 (later PGQL versions support the remaining LDBC queries). Out of these 12 queries, four represent simple path patterns (i.e., Q4, Q17, Q23, Q24) and are directly supported in aDFS. The remaining either include regular path queries (e.g., `SELECT ... MATCH p1 -/:knows*/-> p2`), or include sub-queries in projection or filters (e.g., `SELECT ... FROM (SELECT ...) ...`). We devise a simplified variant of these queries in order to support the benchmark specification as close as possible. For example, the original query 6 is:

```
SELECT id(person),
SUM((SELECT COUNT(*) MATCH (m)<-[:replyOf]-(:cmnt))) AS rN
SUM((SELECT COUNT(*) MATCH (:prsn)-[:likes]->(m))) AS lN,
COUNT(*) AS msgN
MATCH (tag:tag) <-[:hasTag]- (m:post|comment)
        -[:hasCreator]-> (person:prsn)
WHERE tag.name = ?
GROUP BY person, tag
ORDER BY msgN + (2 * rN) + (10 * lN) DESC, id(person)
```

We simplify the query by removing the two `COUNT` subqueries in projections and from `ORDER BY`. We plan to extend the PGQL support in aDFS in future work.

Note that the queries include patterns of varying complexity, e.g., the one in query 6 above is rather simple, while query 17 matches the following complex pattern:

```
(x:person)-[:livesIn]->(c1:city)-[:partOf]->(cy:country),
(y:person)-[:livesIn]->(c2:city)-[:partOf]->(cy),
(z:person)-[:livesIn]->(c3:city)-[:partOf]->(cy),
(x)-[:knows]-(y)-[:knows]-(z)
```

Methodology. We perform 15 runs of each query and report the median latency. For each experiment set, we execute the queries in a per-graph round-robin fashion in order to reduce caching effects. We use eight machines for aDFS, GraphFrames, G-Miner, and Fractal and make sure that all leverage InfiniBand. The four other engines are single machine.

Engines and their configurations. We configure aDFS to use up to 4,192 messaging buffers of 256KB per machine for messaging. This setting translates to approximately 1GB of intermediate results that can be produced per machine and limits the worst-case maximum memory consumption of a single machine to approximately 8GB (1GB outgoing, plus 7GB incoming). We further use the configuration of Section 3.5 for the local-edge dynamic BFS, resulting in up to a few MBs of extra memory per machine. Altogether, the aDFS runtime consumes approximately 10GB per machine. Of course, the graph (that resides in memory) and the final query results consume extra memory than these 10GB.

We first compare aDFS to four systems—two graph and two relational engines—which we describe below. In Section 4.4, we further compare aDFS to three graph-mining systems.

GraphFrames [17] is a distributed graph querying engine built on top of Apache Spark [72, 9]; we use version 0.7 on top of Spark 2.4.1 with 600GB executor memory per machine. ***Neo4j*** [43] is a single-machine graph database, which use stores its data on disk but uses an in-memory cache for performance (caching effects are obvious in the first run of each query). We use Neo4j Community Edition 3.5.3 and allow it to manage the full memory of the machine. ***MonetDB*** [13, 3] is an in-memory column-store relational database. Its distributed support is rather rudimentary, resulting in worse than single-machine performance for our join-heavy workloads. Therefore, we use MonetDB 11.31.13 on a single machine, configured to use the whole 756GB of memory. ***PostgreSQL*** [5] is a relational database. We use version 11.2, tuned for a single connection with memory cache size of 564GB and 198GB of shared buffers. For both MonetDB and PostgreSQL, we use the optimized schema/indices designed for the original LDBC evaluation paper [60]. We choose these four systems as they cover a broad spectrum of data-processing engines: Distributed analytics (i.e., dataframes), single-machine graph databases, and in-memory or traditional relational databases.

4.2. aDFS vs. Other Engines: LDBC

Experiment. We perform an end-to-end comparison of aDFS to the four aforementioned engines. We use the LDBC graph and BI queries which constitute an unfavorable workload for aDFS and GraphFrames. In particular, the LDBC graph has a relational schema, carefully partitioned in tables, such as `person`, and `post`. For relational databases (as well as Neo4j), this schema enables the exploration of small parts

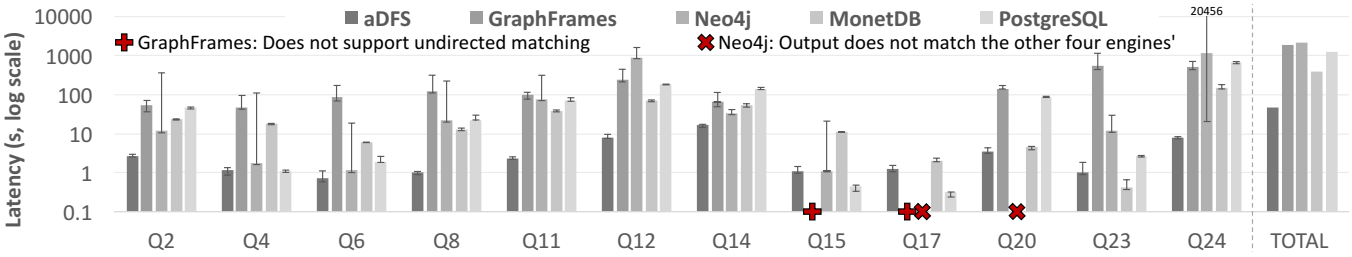

Figure 9: aDFS vs. other graph and relational engines on LDBC. Q*N* is the *N*th LDBC BI query. Error bars show min/max latencies.

of the graph for most queries. For example, the pattern `(:post)-[:hasCreator]->(:person)` (taken from an actual query) needs to only access the tables `post` and `person`, which are a relatively small part for the graph. In contrast, aDFS and GraphFrames operate on the original graph model, where the whole dataset is a single graph. The end result is that these two systems perform more broad exploration even on queries that are very narrow in terms of schema accesses.

Optimizing for relational schemas is outside of the scope of this work. Still, we choose this workload for our end-to-end comparison as it gives a glimpse to queries that can be expressed well both in graph and relational engines. In the next sections, we compare the engines with schema-less graphs.

Results. Figure 9 depicts the query latencies of the five systems. For most queries, aDFS is one to two orders of magnitude faster than GraphFrames. aDFS delivers 102× average speedup and takes 43× less total time that GraphFrames to complete the 10 out of 12 supported queries. GraphFrames translates graph queries to dataframe joins, offered by Apache Spark, which are significantly slower than aDFS's graph traversals. Additionally, GraphFrames is memory hungry, consuming hundreds of gigabytes of memory in comparison to the small footprint of aDFS. Furthermore, aDFS completes the 10 supported queries 53 times faster than Neo4j, with 35× average speedup, although Neo4j leverages the graph schema and utilizes the large amount of memory as graph cache (i.e., the whole graph resides in memory after the first run).

Comparing aDFS to the two relational engines, MonetDB and PostgreSQL, we notice two different behaviors depending on the query size. On the one hand, for large queries, such as Q12 and Q24, which expand to large parts of the graph with long paths, aDFS is significantly faster. On the other hand, for small, very relational queries, such as Q15, Q17, and Q23, the relational engines can be faster than aDFS. This is expected given that just the distributed bootstrapping and coordination overheads in aDFS account for several tens of milliseconds. These different queries highlight the trade-off between the relational table-focused joins compared to the graph exploration approach of aDFS. Overall, aDFS completes the whole set of queries 8.4 and 26 times faster than MonetDB and PostgreSQL, respectively. The average speedups are 10× and 25× against MonetDB and PostgreSQL, respectively. Conversely, MonetDB is 2.4× faster than aDFS on Q23, while PostgreSQL is on average 2.6× faster for queries 4, 15, and 17. Our BFS adaptation of aDFS, described in Section 2.4, is 30% slower than aDFS (not shown), while consuming more memory.

In conclusion, aDFS achieves better overall performance than the four other engines while consuming lower/fixed runtime memory. Both these characteristics are essential for a graph processing server which targets large workloads and possibly multiple concurrent users.

4.3. aDFS vs. Other Engines: Large Schema-Less Queries

Experiment. The classic property graph model is schema-less, which enables users to easily query the whole dataset (unlike the relational model which requires several joins and unions of results). We thus compare aDFS to the other four engines with the schema-less graphs of Table 2. For the relational engines, the graph consists of two tables: One for vertices and another for edges. Regarding queries, we use two simple patterns, a cycle `(a)->(b)->(a)` as Q1 and a two-hop path `(a)->(b)->(c)` as Q2, combined with aggregations in the `SELECT` clause (variant "a" performs a `COUNT(*)` and variant "b" `AVG` aggregations on a random vertex property). The conclusions remain the same for other patterns and projections (not shown in the graphs). Note that it is impossible to evaluate with more elaborate patterns, as the competing engines can barely handle the simple patterns that we use.

Results. Figure 10 depicts the results. In most cases, aDFS is about 2 orders of magnitude faster than the other systems. For the large queries and graphs, we also see that the other engines are either not able to complete the queries within eight hours, or crash. In particular, GraphFrames crashes after having consumed its 600GB of executor memory.

The speedups of aDFS over the other engines (for the completed queries where there is no timeout) are: 16 to 62× for GraphFrames, 1,105 to 9,200× for Neo4j, 20 to 169× for MonetDB, and 60 to 190× for PostgreSQL. Neither the join-based engines (GraphFrames, MonetDB, and PostgreSQL) nor Neo4j are able to handle well these immense graph explorations, although they have access to hundreds of gigabytes of memory. In particular, Neo4j spills to disk, hence the extreme performance difference compared to aDFS. Clearly, for graphs and queries at this scale, a fast graph-optimized solution such as aDFS, which handles them easily, is required. aDFS easily handles these queries. With the largest query (Q2a on Twitter) aDFS performs a 9.3T `COUNT` in 1,286 seconds, resulting in 7.3B matches per second, while consuming less than 10GB per-machine memory for intermediate results.

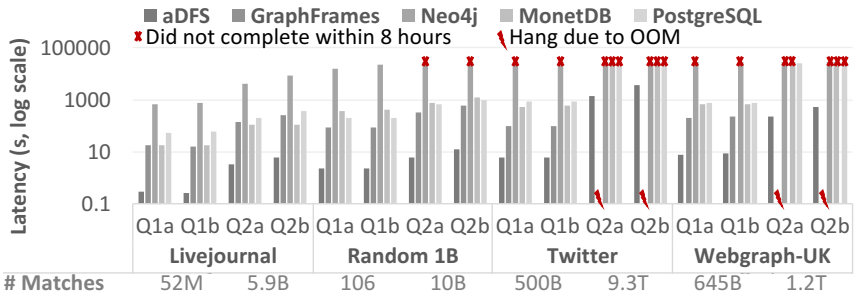

Figure 10: aDFS vs. other graph and relational engines on simple-pattern queries.

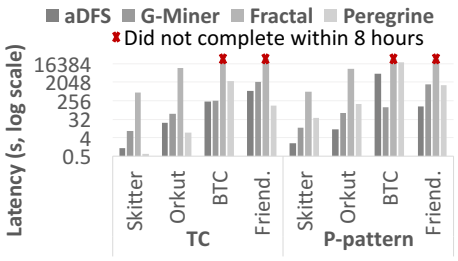

Figure 11: aDFS vs. graph-mining engines on triangle counting and pattern matching.

4.4. aDFS vs. Graph Mining

Experiment. We compare aDFS to three graph-mining engines, namely ***G-Miner*** [14] and ***Fractal*** [18], which are distributed, as well as ***Peregrine*** [28], which is single machine. (Note that we requested the artifact of Automine [39] for evaluation, but the authors were not able to provide us with it.) We use workloads from the G-Miner paper [14]: TC, i.e., Triangle Counting, and counting instances of a more complex pattern referred to as the P-pattern; with the four graphs that are used to evaluate these operations in the paper. For aDFS, we express both triangles and the P-pattern as graph queries.

Results. Figure 11 includes the performance of the four systems. Triangle counting (TC) is a classic isomorphism problem and highlights the difference between isomorphic and homomorphic matching: For the three graph-mining engines, the search for "unique" triangles is baked in the pattern-matching algorithm, whereas in aDFS, we implement isomorphic matching with dynamic filtering (i.e., `(a)->(b)->(c)->(a) WHERE ID(a)<ID(b) AND ID(b)<ID(c)`). This results in expensive filtering and heavier cross-machine communication compared to the other engines. Still, aDFS is faster than G-Miner and Fractal for all graphs by up to 7× and 531×, respectively. Peregrine outperforms all other engines including aDFS on three out of the four graphs, as it is able to intersect adjacency lists to quickly find common neighbors, an optimization that performs particularly well for triangles and which can be implemented in a straightforward manner on a single machine, where the whole graph is accessible.

The P-pattern does not require explicit isomorphic checks, as its vertices are differentiated by labels. We express it as:

```
(c:c)->(b1:b)->(:a)->(c)->(b2:b)->(:d) WHERE b1<>b2
```

in PGQL. When matching the P-Pattern, aDFS significantly outperforms all other engines for all but one datapoint (G-Miner on BTC); it is on average 12 and 65 times faster than Peregrine and Fractal, respectively, and 8 times faster than G-Miner on three graphs. G-Miner achieves the best performance on BTC mainly because it replicates the target vertex label with each edge, which increases locality and reduces communication traffic. Such an optimization is not practical in a real-world system in which vertices can have many labels and properties of various types: Replicating vertex labels and properties for each edge can have unacceptable memory overhead.

Overall, although aDFS is designed for different workloads, i.e., expressive graph queries, it is still very competitive, and most of the times faster than state-of-the-art graph-mining engines on mining-oriented workloads.

4.5. aDFS Scalability

Experiment. We use the LDBC workload to illustrate the scalability of aDFS as we vary the number of machines.

Results. Figure 12 includes the speedups, normalized to the latency of a single machine. Overall, aDFS exhibits very good scalability: The average speedup from one to two machines is 1.6×, from one to four machines is 2.5×, and from one to eight machines is 5.4×. These numbers include various distributed coordination, query compilation, and more fixed overheads, meaning that the actual runtime scalability is even better. Looking at the pure pattern execution time, without coordination overheads, group by, and order by, the speedup improves to 1.7×, 2.6×, and 6× from one to two, four, and eight machines respectively (not shown). aDFS is designed to scale: More machines translate to more compute resources, more buffers for intermediate results, and often more BFS exploration and higher network utilization, as the percentage of remote edges increases with the number of machines.

5. Related Work

Database Management Systems (DBMSs). DBMSs offer graph support via a multi-model premise, but focus on SQL-like querying rather than pattern-matching querying [40, 47, 46, 37]. Kalinsky et al. [29] acknowledge that using DBMS joins for graph pattern matching is suboptimal, and propose hardware support to alleviate the issue. In contrast to DBMSs, aDFS is an efficient in-memory distributed graph engine that considers graph storage and queries as first-class citizens and focuses on analytical rather than transactional workloads.

Graph Algorithms. There is a plethora of related work for executing graph algorithms (such as PageRank [49]). Single-machine solutions focus on various topics such as proposing DSLs [25] or programming models [44] for graph algorithms, performance optimizations [57, 59], leveraging hardware features such as NUMA [74] and GPUs [45, 77], or supporting out-of-core computing [76]. Distributed solutions focus on topics such as asynchrous processing and performance [22, 35], efficient partitioning [71, 75, 78], leveraging hardware features such as RDMA [68], support for secondary storage [53], distributing sequential algorithms [20],

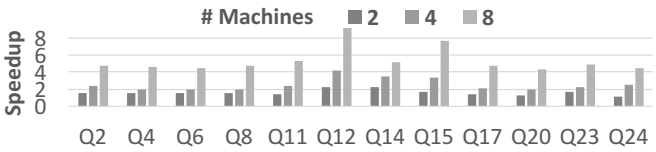

Figure 12: Scalability of aDFS (relative to using one machine).

approximate computing [64], alternative programming paradigms [69], or fault tolerance [65, 16]. aDFS focuses on graph queries rather than algorithms, but it shares features with some of these distributed solutions, such as the use of asynchronous processing or (random) graph partitioning.

Graph Querying. A number of single-node graph-querying engines were proposed by academia: Sun et al. [58] and Lin et al. [34] build relational and transactional engines, Graphflow [30] is an active graph database that supports evaluating one-time and continuous subgraph queries, and TurboFlux [31] optimizes fast continuous subgraph matching over a fast graph update stream. Roth et al. prototype distributing simple DFS exploration [52].

There are numerous industrial graph-querying solutions, because queries are key to allow users to effortlessly express any pattern, filter, and projection. Neo4j [43] is a single-machine engine that supports Cypher [42] queries. Amazon Neptune [8] is built for the Amazon cloud. Facebook Dragon [19] builds indices on updates for accessing data. Microsoft Graph Engine [41] is an in-memory data processing engine based on Trinity [55], and TigerGraph [63] distributes GSQL [2] queries based on the source vertex data for a given query hop. Furthermore, there are also open-source distributed solutions. JanusGraph [62], which supports Gremlin [7] queries, uses distributed graph storage but does not distribute computation. GraphFrames [17], implements graph pattern matching with Spark using joins of dataframes.

To the best of our knowledge, aDFS is the first truly distributed query engine on fully-partitioned graphs that bounds memory while maintaining fast query performance.

Graph Mining. While graph querying aims to match patterns, enriched with filters and dynamic projections, graph mining aims to find subgraph patterns characterized by complex aggregate measures [67, 21]. Examples include triangle counting, maximal clique finding, community detection, and graph matching [51, 14, 39]. Technically, graph query engines typically employ a vertex/edge-centric processing approach: A state is maintained per vertex and communicated to its neighbors [61, 39]. Graph mining engines typically follow a subgraph-centric (often undirected and schema-less) processing approach: They attach information to a large amount of intermediate results composed of subgraphs [39] rather than specific vertices. Additionally, while graph-querying engines search homomorphic patterns by default, graph-mining engines search for isomorphic patterns [18, 28, 15].

Single-machine engines include RStream [66], AutoMine [39], and Peregrine [28]. Distributed engines include Arabesque [61], NScale [51], G-thinker [70], G-Miner [14], ASAP [27], and Fractal [18]. aDFS shares features with some of these engines: G-Miner [14] uses asynchrony, while G-Thinker [70] and Fractal [18] attempt to bound memory consumption. aDFS both uses asynchrony and adaptive graph traversals that bound memory consumption. Of course, graph-mining engines could also leverage our design. Graph-mining engines do pattern matching, albeit not optimized for expressive graph querying with filters and dynamic projections. Moreover, they usually rely on a simple graph model that does not support any number of labels or properties of any types for any vertex or edge.

RDF Graphs. The Resource Description Framework (RDF) uses {*subject*, *predicate*, *object*} triples to represent graphs, which can be queried with languages such as SPARQL [6]. RDF became popular with the semantic web and has been the model of choice for many graph databases starting in the early 2000's [73, 24]. A number of works have focused on distributed RDF graphs [26, 56].

Although the RDF model is equivalent to PG in terms of expressiveness, there are differences: RDF adds links for every graph data piece, including constant literals, it does not have explicit vertices/edges—yet it can be viewed as a graph—and it does not store properties separately. Triples force RDF engines to process and join a much larger number of intermediate results using e.g., a key-value style storage, and lose the graph structure, resulting in slower neighbor lookup. To address these drawbacks, some RDF engines use asynchronous processing [24], or compute graph indices, using e.g., the CSR representation, to mock the graph structure [73]. aDFS focuses on PGQL queries on the PG model, avoiding complex and expensive joins. We note that the pattern-matching part of query execution is largely orthogonal to the graph model and aDFS's techniques could be used for RDF graphs.

6. Concluding Remarks

Conclusions. We have introduced aDFS: An almost-DFS engine for pattern-matching queries on distributed graphs. aDFS is able to execute virtually any query on any in-memory graph using at most a fixed, configurable amount of memory. aDFS is also very fast and scalable. We compared aDFS to seven state-of-the-art engines with diverse characteristics—graph-focused or relational, distributed or single machine, in-memory or disk-based—and showed that aDFS is up to orders of magnitude faster than these engines.

Limitations and future work. aDFS uses simple algorithms for query optimization and graph partitioning, as this paper focused on runtime support for distributed graph querying. In the future, we intend to improve distributed query planning and optimization, together with graph partitioning and caching. We will also consider query-optimization opportunities when the underlying data has a relational-style schema (as described in Section 4.2). In particular, introducing graph-schema support would offer opportunities for pruning the exploration space.

References

[1] Google Web Graph. https://snap.stanford.edu/data/web-Google.html.

[2] GQL Standard. https://www.gqlstandards.org.

[3] MonetDB, An Open-Source Database System. https://www.monetdb.org.

[4] PGQL: Property Graph Query Language. https://pgql-lang.org/spec/1.1/.

[5] PostgreSQL, An Open-Source Relational Database. https://www.postgresql.org.

[6] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[7] Tinkerpop, Gremlin. https://github.com/tinkerpop/gremlin/wiki.

[8] Amazon. Neptune. https://aws.amazon.com/neptune.

[9] Apache. Spark. https://spark.apache.org.

[10] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *SIGKDD*, 2006.

[11] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The Semantic Web. *Scientific American*, 284(5), 2001.

[12] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Web Graph. *SIGIR Forum*, 42(2), 2008.

[13] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking The Memory Wall in MonetDB. *Commun. ACM*, 51(12), 2008.

[14] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *EuroSys*, 2018.

[15] Soumyava Das and Sharma Chakravarthy. Duplicate Reduction in Graph Mining: Approaches, Analysis, and Evaluation. *IEEE KDD*, 30(8), 2018.

[16] Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Phoenix: A Substrate for Resilient Distributed Graph Analytics. In *ASPLOS*, 2019.

[17] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *GRADES*, 2016.

[18] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *SIGMOD*, 2019.

[19] Facebook. Facebook Dragon. https://code.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/.

[20] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing Sequential Graph Computations. In *SIGMOD*, 2017.

[21] Brian Gallagher. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. In *AAAI Fall Symposium*, 2006.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in A Distributed Dataflow Framework. In *OSDI*, 2014.

[24] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD*, 2014.

[25] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, 2012.

[26] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[27] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *OSDI*, 2018.

[28] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *EuroSys*, 2020.

[29] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. The TrieJax Architecture: Accelerating Graph Operations Through Relational Joins. In *ASPLOS*, 2020.

[30] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, 2017.

[31] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *SIGMOD*, 2018.

[32] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. What Is Twitter, A Social Network Or A News Media? In *WWW*, 2010.

[33] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[34] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. Fast In-Memory SQL Analytics on Typed Graphs. *PVLDB*, 10(3), 2016.

[35] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J.M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8), 2012.

[36] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1), 2007.

[37] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast Query Processing via Graph Exploration. *PVLDB*, 9(12), 2016.

[38] 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 *SIGMOD*, 2010.

[39] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *SOSP*, 2019.

[40] Microsoft. Microsoft Azure Cosmos DB. https://azure.microsoft.com/en-gb/services/cosmos-db/.

[41] Microsoft. Microsoft Graph Engine. https://www.graphengine.io.

[42] Neo4j. Cypher. https://www.neo4j.org/learn/cypher.

[43] Neo4j. Neo4j Graph Database. https://www.neo4j.org.

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*, 2013.

[45] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *ASPLOS*, 2018.

[46] Open Query. MariaDB OQGRAPH. https://openquery.com.au/products/graph-engine.

[47] OrientDB. OrientDB, A Multi-Model Database. https://orientdb.com.

[48] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1999.

[49] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[50] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. Managing Large Graphs on Multi-Cores with Graph Awareness. In *USENIX ATC*, 2012.

[51] Abdul Quamar, Amol Deshpande, and Jimmy Lin. NScale: Neighborhood-Centric Large-Scale Graph Analytics in the Cloud. *The VLDB Journal*, 25(2), 2016.

[52] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *GRADES Workshop*, 2017.

[53] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-Out Graph Processing From Secondary Storage. In *SOSP*, 2015.

[54] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *SOSP*, 2013.

[55] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

[56] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *OSDI*, 2016.

[57] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, 2013.

[58] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD*, 2015.

[59] Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Subramanya Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High Performance Graph Analytics Made Productive. *PVLDB*, 8(11), 2015.

[60] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An Early Look at The LDBC Social Network Benchmark's Business Intelligence Workload. In *GRADES Workshop*, 2018.

[61] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *SOSP*, 2015.

[62] The Linux Foundation. JanusGraph. https://janusgraph.org.

[63] TigerGraph. TigerGraph - Distributed Query Mode. https://www.tigergraph.com/distributed-query-mode/.

[64] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *ASPLOS*, 2017.

[65] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *ASPLOS*, 2017.

[66] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. In *OSDI*, 2018.

[67] Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1), July 2003.

[68] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. GraM: Scaling Graph Computation to the Trillions. In *SoCC*, 2015.

[69] Chengshuo Xu, Keval Vora, and Rajiv Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *ASPLOS*, 2019.

[70] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR*, abs/1709.03110, 2017.

[71] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *PVLDB*, 7(14), 2014.

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

[73] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB*, 6(4), 2013.

[74] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *PPoPP*, 2015.

[75] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the Hidden Dimension in Graph Processing. In *OSDI*, 2016.

[76] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *ASPLOS*, 2018.

[77] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: An Efficient Path-Based Iterative Directed Graph Processing System on Multiple GPUs. In *ASPLOS*, 2019.

[78] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*, 2016.

[79] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: A Graph-Based SPARQL Query Engine. *The VLDB Journal*, 23(4), 2014.