Hyracks Unchained: Efficient Recursion for Navigational Queries in Apache AsterixDB

Glenn Galvizo*
Couchbase
glenn.galvizo@couchbase.com

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

ABSTRACT

Navigation is a central construct in most modern graph query languages, though the execution of navigational queries are often suboptimal at scale. Navigational queries, in contrast to other graph-based problems, do not benefit from the bulk-synchronization process that most graph processing systems hinge on. Graphix, an extension of the Big Data management system Apache AsterixDB, enables its users to author navigational queries. In this paper we explain how Hyracks, the underlying execution engine of AsterixDB (and subsequently Graphix) was extended to realize navigational queries on a shared-nothing cluster of nodes *without* per-step bulk synchronization. We include an evaluation of our solution using a workload of operational and analytical queries to show that independence, a property of path finding, can be exploited in Hyracks to efficiently execute Graphix queries.

VLDB Workshop Reference Format:
Glenn Galvizo and Michael J. Carey. Hyracks Unchained: Efficient Recursion for Navigational Queries in Apache AsterixDB. VLDB 2025 Workshop: Large-Scale Graph Data Analytics.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/graphix-asterixdb/.

1 INTRODUCTION

Recursive computation on Big Data is required in many modern applications. From the iterative processes found in machine learning to the evaluation of recursive Datalog programs, the efficient execution of recursive queries and programs remains an active area of research. A particular class of interesting queries are *navigational queries* over graph data, which aim to enumerate paths (sequences of related vertices and edges) between two vertices. These queries are most commonly expressed in graph query languages like Cypher [13] or GQL [12, 20]. Evaluation of these queries are commonly delegated to graph databases that tightly couple their storage (i.e., how vertices and edges are physically represented) and their query languages. As a consequence, users aiming to pose navigational queries over existing non-graph data must ultimately develop and maintain some form of ETL (extract-transform-load) pipeline to duplicate their existing data to a graph database. This paradigm leads to a higher cost to "own your data" (i.e., storage cost, compute cost, and development cost). We contrast the composition of multiple, narrow-purpose systems with one system that offers multiple (non-materialized) views of the same data. Graph processing systems such as Pregel [21], Giraph [5], and Pregelix [9] allow users to specify graph algorithms over their existing data to execute at massive scale using bulk synchronous parallelism (BSP). While BSP enables the efficient computation of algorithms like PageRank, navigational queries benefit less due to the fact that path finding is a) *sequential* (i.e., a path grows one vertex at a time), and b) *independent* (i.e., paths can be enumerated without synchronization).

In this paper, we will focus on how navigational queries can be executed by a distributed database *in-situ* and in a manner that exploits path finding's independence property. Specifically, we will detail how the Graphix extension of Apache AsterixDB leverages the execution engine Hyracks to realize semi-synchronous recursion for navigational queries. The remainder of this paper is organized as follows: Section 2 reviews AsterixDB, its Graphix extension, and the Hyracks execution engine. Section 3 introduces how we extended Hyracks to execute *unbounded* navigational queries. Section 4 details an evaluation of our Hyracks extension. Section 5 reviews related work around recursion for navigation.

2 BACKGROUND

In this section, we give an overview of i) AsterixDB, ii) the Graphix extension for AsterixDB, and iii) Hyracks, the runtime engine underlying AsterixDB. While discussing i), ii), and iii), we will also describe a) a social network example, b) an example AsterixDB (plus Graphix) architecture to host the social network example, and c) a navigational query example.

2.1 Apache AsterixDB

Apache AsterixDB is a Big Data management system (BDMS) designed to be a highly scalable platform for document storage, search, and analytics [2]. AsterixDB possesses a flexible, semi-structured data, model that accommodates a range of use cases —from "schema-first" to "schema-never". To query AsterixDB, SQL^{++}, a generalized form of SQL that enables the querying of semi-structured data is used [10]. To scale horizontally AsterixDB follows a shared-nothing architecture, where each node independently accesses storage and memory. All nodes are managed by a central cluster controller that a) serves as an entry point for user requests and b) coordinates work amongst the individual AsterixDB nodes. After a query arrives at the cluster controller, the query is translated into a logical plan and subsequently rewritten in a rule-based and cost-based manner to produce an optimized physical plan [7]. This optimized physical plan is then translated into a job that can run across all nodes in the cluster [8]. Datasets in AsterixDB are hash-partitioned

*Also with University of California, Irvine.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

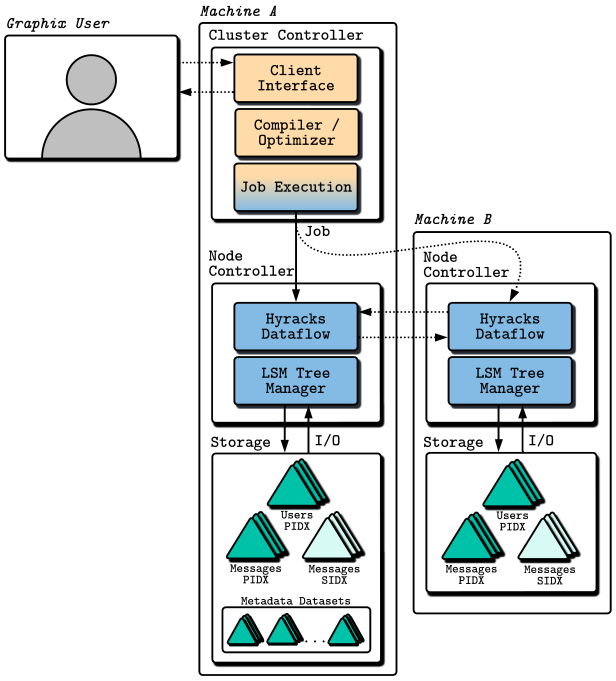

Figure 1: Architecture for the two-node AsterixDB / Graphix example cluster that supports the social network database.

across the cluster on their primary key into primary B^+ tree indexes, where the data records reside, with secondary indexes being local to the primary data on each node. Both internal datasets and secondary indexes are LSM (Log-Structured Merge) based, enabling fast ingestion performance [3].

To illustrate many of the concepts discussed in this paper, we will assume a social network example database defined as collections of AsterixDB documents. Two major entities are captured in our example: (i) `Users` and (ii) `Messages`. Two relationships are captured in our social network: (I) a `User` may *post* one or more `Message(s)` and (II) a `Message` may *reply to* exactly one `Message`.

To host our social network example, we will assume the two-node AsterixDB architecture in Figure 1 going forward. In Figure 1, a user first submits a query to the cluster controller process on machine *A*. The cluster controller process builds an optimized job that is distributed to both node controller processes on machine *A* and machine *B*. When the job is finished executing, the cluster controller responds back to the user with the results of the query. To scale outward in this architecture, an AsterixDB user would spawn more node controller processes on additional machines.

2.2 Graphix Extension

Graphix is an Apache AsterixDB extension that enables users to perform ad-hoc, partitioned-parallel, and synergistic graph plus document analytics over existing AsterixDB data in-situ [14, 15]. Graphix allows users to define property graph views that map their original AsterixDB documents to (virtual) labeled collections of vertices and edges. Using the `Users` and `Messages` datasets from Subsection 2.1, our running example consists of a Graphix property graph view `SocialNetworkGraph` with vertex types `(:User)` and

```
1 FROM GRAPH SocialNetworkGraph
2   (u:User)-[:WROTE]->(m1:Message),
3   (m1)-[r:REPLY_OF+]->(m2:Message)
4 WHERE u.id = $uid
5 SELECT u, m1, m2, r;
```

Listing 1: The running query example: find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) paths of message responses `r` from `m1` to `m2`.

`(:Message)` (defined as the `Users` and `Messages` datasets respectively) and the following edge types:

(1) `(m1:Message)-[:REPLY_OF]->(m2:Message)`, defined as the `JOIN` between two messages `m1`, `m2` where `m1.reply_of = m2.id`; and
(2) `(u:User)-[:WROTE]->(m:Message)`, defined as the `JOIN` between a user `u` and a message `m` where `u.id = m.user_id`.

The complete definition for `SocialNetworkGraph` is given in [15]. Once a Graphix graph is defined, users are free to immediately issue queries about their graph using gSQL^{++}, an extended version of SQL^{++} with Cypher-inspired path syntax. gSQL^{++} extends the `FROM` clause of SQL^{++} to additionally bind vertex, edge, and path patterns of a navigational graph pattern to SQL^{++} variables. A gSQL^{++} query issued to the cluster controller of a Graphix cluster is rewritten into a SQL^{++}-esque query (more accurately, an abstract syntax tree) that is then processed and executed using AsterixDB's optimizer and runtime.

As discussed in the introduction, our work here addresses the specific class of recursion that is required to specify *navigational query patterns* in Graphix. A navigational query pattern includes one or more *path patterns*, which qualify *paths* using regular expressions of edge labels. For example, the gSQL^{++} query in Listing 1 specifies the regular expression `REPLY_OF+`. The variable `r` is bound to all paths containing at least one `REPLY_OF` edge. Paths in Graphix are AsterixDB documents that contain two list-valued fields: i) `Vertices`, a list of vertices represented as virtual AsterixDB documents, and ii) `Edges`, a list of edges also represented as virtual AsterixDB documents. Section 3 will focus on how Line 3's pattern in Listing 1 is evaluated on the architecture of Figure 1.

2.3 Hyracks Runtime

Hyracks is the runtime engine used by AsterixDB and Graphix, enabling partitioned parallel data-flow computations on shared-nothing clusters of machines. The top-level unit of work in Hyracks is a job, described as a directed graph of *operators* and *connectors*. Hyracks operators consume and produce data, while connectors redistribute data between operators. A Hyracks operator is composed of one or more activities, each of which specifies logic for handling a frame of data. As an example, the hash `JOIN` operator is composed of two activities: one to build the hash table, and another to probe the hash table (i.e., execute the `JOIN`). We refer to the graph of all activities across all operators in a Hyracks job as an *activity graph*. Each activity later becomes instantiated as several identical tasks that are distributed to different partitions to realize the associated Hyracks job. A Hyracks operator may also have blocking requirements on its activities (e.g., a hash `JOIN` operator must finish the

activity to build its hash table before starting the activity to probe its hash table), which the Hyracks scheduler then uses to define groups of activities (known as stages) to execute in series.

At runtime, data in Hyracks is *pushed* from producers to consumers in the units of fixed-size, contiguous byte-arrays known as *frames*. Hyracks activity developers implement a set of methods that operate on frames. The primary carriers of data are *records* that are contained within frames. Activities and connectors are typically implemented in a way that maximizes the number of records in a frame before sending it to downstream consumers, although this is not a Hyracks requirement. Hyracks was designed to be data model agnostic, thus the contents of a frame are not inherent to Hyracks. As long as all Hyracks activities and connectors in a job agree on the frame format, Hyracks will move data through a job appropriately.

3 HYRACKS (UNCHAINED)

Prior to gSQL^{++}, Hyracks jobs were acyclic. In this section, we will describe three problems that can emerge in the event of *cycles* in Hyracks activity graphs: 1) the problem of *liveness*, where no progress is being made, 2) the problem of *safety*, where activity instances can deadlock due to an over-allocation of resources, and 3) the problem of *mortality*, where activity instances may never terminate. We will then describe two additions to Hyracks to realize semi-synchronous navigational Graphix queries: the `FIXED POINT` operator and a minimally invasive "decorator" to supplement the responsibility of an existing Hyracks activity.

We begin our discussion with a review of the existing "internals" of Hyracks activities. A Hyracks activity must, at a minimum, implement the `IFrameWriter` interface.* The `IFrameWriter` interface consists of following core methods, all of which are *only* called by the activity's upstream (data providing) activity.

(1) `open()`, which initializes resources and subsequently calls the `open()` method for its own downstream activities;
(2) `nextFrame(f)`, which accepts an input frame `f` to perform some computation on using the contents of the frame; and
(3) `close()`, which a) deallocates any acquired resources, b) forwards any partial frame data that the activity currently has buffered to its downstream activities, and c) calls the `close()` method for its own downstream activities.

The typical lifecycle of a Hyracks activity instance involves (i) getting its own `open()` method called and calling `open()` for its own (downstream) consumers, (ii) accepting full frames of tuples from an upstream producer via its own `nextFrame(f)` method and calling the `nextFrame(f)` method for its own consumers when its output buffer becomes full, and finally (iii) getting its own `close()` method called and calling `close()` for its own downstream consumers.

3.1 Single Partition Recursion

Now let us consider a two-activity group composed of activities a_1 and a_2 within a *single* partition. To support recursion, activity a_1's output will be connected to activity a_2's input, and activity a_2's output will be connected to activity a_1's input. Figure 2 depicts

*Activities that act as a *source* in an activity cluster (i.e., activities that do not have any input to themselves) do not implement this interface, but source activities cannot appear inside of a cyclic activity graph (otherwise they would not be sources). For clarity, we will consider each activity as both a producer and consumer in this section.

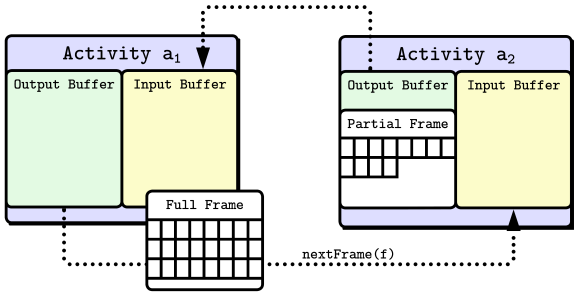

Figure 2: Depiction of activity a_1 forwarding its output buffer to activity a_2 via a_2's `nextFrame(f)` method.

an instance of a_1 directly pushing a full frame to a_2 by calling a_2's `nextFrame(f)` method. To understand how most Hyracks activities implement the `IFrameWriter` interface, the activities associated with a_1 and a_2 will only call each other's `nextFrame(f)` method when: 1) their own output buffer frame is full or 2) their upstream producer has indicated that it has no tuples left to offer (i.e., by having its own `close()` method called). This `IFrameWriter` implementation maximizes the amount of information held in a frame before the activity forwards the output buffer frame downstream, ultimately leading to better utilization of frame-transferring resources like network bandwidth. In our example in Figure 2, we note that activity a_1 is forwarding a full frame from its output buffer to the input of a_2. Activity a_2 still has a partial frame, so it does not forward its output to a_1 yet.

3.1.1 Maintaining Liveness. We will first consider the *liveness* property, which describes a group of activities that are always "making progress". In the context of navigation, liveness describes a group of activities that will eventually generate all (satisfiable) paths. Below, we describe a scenario where this liveness property is violated:

Scenario .....
Assume that both activities a_1 and a_2 from Figure 2 currently possess partially full output buffers.

Violation .....
The two activities possess the potential to perform more work, but they will not due to their "forward when full" implementation. a_1 has a partially full output buffer that it *could* forward to a_2 but does not. Similarly, a_2 has a partially full output buffer that it *could* forward to a_1 but does not.

A starting point to remedy this liveness violation involves adding the following requirements for each activity within a loop: i) the ability to forward partially full output buffers, and ii) some method of invoking this ability. We note that the first requirement has already been implemented for all activities, as a `flush()` method (originally purposed for AsterixDB feeds), so our solution only needs to consider the second requirement (i.e., by invoking the `flush()` method).

At a high level, to guarantee liveness we must add a form of inter-activity communication beyond the method calls provided by the existing `IFrameWriter` interface. Our solution is an in-band approach that leverages the `IFrameWriter` interface that each activity already implements: 1) We define two classes of frames: (i) a data frame, full of tuples, and (ii) a new *message* frame, used to pass

information to other activities downstream. 2) We non-invasively "decorate" (in the object-oriented design pattern sense) each activity in a Hyracks job that may violate liveness (i.e., those inside of a loop) to recognize and act on these new message frames. Moving back to the liveness violation scenario, both a_1 and a_2 would first be decorated to recognize message frames. Either a_1 or a_2 would then generate and forward a message frame containing a `RELEASE` directive to the decorated activity instance a_1/a_2 using a_1/a_2's `nextFrame(f)` method (we will detail *when* and *who* generates these message frames after addressing the two other properties). The callee activity would then call its own `flush()` method to forward its partial frame to the input of the caller activity. Message frames can be viewed as a form of "punctuation" [29], which are used in the context of stream processing as signals for stream processors to release state.

3.1.2 Maintaining Safety. The second property of interest is the *safety* property, which (for our purposes) describes a group of activities that will never deadlock. Below, we describe a scenario where this safety property is violated:

Scenario .....
Assume that both activities a_1 and a_2 from Figure 2 now possess full output buffers *and* full input buffers.

Violation .....
a_1 and a_2 have full input and output buffers, thus no progress can be made. The resources in contention here are frames (specifically, frames used to perform network I/O). In this safety violation scenario, activity a_1 has filled all frames within its budget and is prepared to send these frames to activity a_2. a_2, however, cannot receive these frames from a_1 since a_2 has filled all frames within its budget and is prepared to send these frames to activity a_1. Neither a_1 nor a_2 is aware of the fact that their actions are causing a deadlock.

To remedy this deadlock violation, we designate (at compile time) one activity within a cyclic activity group to avoid exhausting "shared resources" by simply moving any acquired frames (via its `nextFrame(f)` method) to a separate secondary buffer. This separate buffer possesses its own memory budget with an ability to spill to disk when full. After some point, all activities within the cyclic activity group will have forwarded everything stored in their secondary buffer to this designated activity. The designated activity will then forward everything stored in its secondary buffer, repeating this buffer-and-then-forward process until all frames are exhausted. Suppose that activity a_1 is designated to store each frame sent by a_2 to its own secondary buffer. After a_2 has given all of its frames to a_1, a_1 then forwards all of the frames in its secondary buffer to activity a_2. At a glance, this buffer-and-then-forward process may seem like Graphix is performing a form of global synchronization at each step of the computation. We reminder the reader here that all of our explanations and examples thus far have been local to a single partition. As we will later see, Graphix performs this buffer-and-then-forward process locally per partition *without* a need for inter-partition synchronization.

3.1.3 Maintaining Mortality. The last property we will consider is the *mortality* property, which guarantees that every activity will eventually terminate (i.e., call `close()`) when there is no work left to do. For activity groups with cycles, we can easily show a violation of this mortality property:

Scenario .....
Assume that both activities a_1 and a_2 from Figure 2 now possess completely empty input and output buffers. There exists no work left to do.

Violation .....
The two activities a_1 and a_2 *could* terminate but do not. In order for activity a_1 to finish, its upstream producer a_2 must call a_1's `close()` method. Conversely, activity a_2 will only call the `close()` of a_1 when activity a_1 calls a_2's `close()` method. Clearly, neither a_1 nor a_2 can call `close()`. This termination problem is inherent to *all* Hyracks jobs with cycles, as an activity is only aware of its upstream producers (indirectly via the `IFrameWriter` interface).

We will first detail our solution for a single partition and later show that we can remedy this mortality violation globally so as to adhere to our previously defined "non-globally-blocking" objective. To start, we note that an activity a can only reason about the termination status of its own immediate upstream producer (i.e., by having a's own `close()` method invoked). Our solution requires i) designating (at compile time) one activity to call `close()` when there exist no tuples left to process, ii) getting all activities within the loop to report on their status, and iii) sending the statuses of each activity to the designated "`close()`-er" activity. Graphix realizes these requirements by extending the use of message frames in the previous liveness section. Our solution starts with either activity a_2 or a_1 giving the `RELEASE` directive along with an *uncolored marker* inside a message frame to activity a_1/a_2 via a_1/a_2's `nextFrame(f)` method. If the callee activity has any tuples left to process, it will color in the marker and forward the message frame downstream (back to the caller activity). Otherwise, the callee activity will push an uncolored marker frame. When our caller activity receives the inevitable uncolored marker frame, the caller will then invoke the `close()` method of a_1/a_2. The callee activity will subsequently invoke the `close()` method of the original caller, terminating the computation. This use of message frames was inspired by the use of punctuation in the FFP (flying fixed point) operator of [11] for the problem of cyclic stream processing.

3.2 Fixed Point Operator

Returning to our running example query (specifically, the path pattern `(m1)-[r:REPLY_OF+]->(m2)`), Figure 3 defines a partial graph of activity instances distributed across a two-node cluster that finds all paths `r` from the source messages $\texttt{m}_1$ to the destination messages $\texttt{m}_2$ while satisfying our liveness, safety, and mortality properties. For brevity, we omit the subgraph of activities (the "..." at the bottom) involved in the computation of the variables `u` (the source user), $\texttt{m}_1$ (the source message) and $\texttt{r}_0$ (an initial zero-length path). As described in the previous three sections, the solution to each problem caused by cycles in the activity graph involves (at a minimum) elevating the responsibility of some designated activity. Starting at the bottom, we define a new group of activities controlled by a `FIXED POINT` operator. Algebraically, this operator acts like a `UNION ALL` operator with one output and two inputs: the *anchor* input (from below) and the *recursive* input. In Figure 3, `FIXED POINT` binds the variable $\texttt{m}_{\text{prev}}$

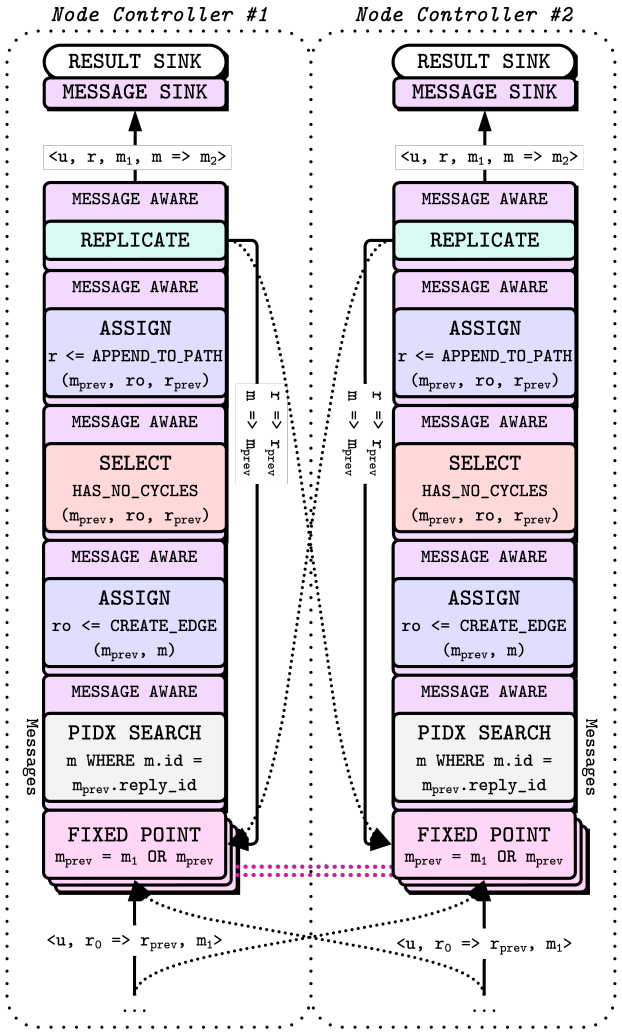

Figure 3: Subgraph of Hyracks activity instances (distributed across a two-node cluster) used to find the positive closure of `REPLY_OF` edges connecting `Message` vertices.

to m_1 (from the anchor input) and m_{prev} (from the recursive input). Instances of the `FIXED POINT` operator are additionally responsible for: (i) generating message frames with the `RELEASE` directive and an uncolored marker to forward to their immediate downstream activity (here, the `PIDX SEARCH`); (ii) buffering incoming frames from the `REPLICATE` activity to maintain safety; and (iii) determining whether or not it is appropriate to call `close()`.

Datasets in AsterixDB are hash partitioned using their primary key, thus records into the `FIXED POINT` operator must first be forwarded to the appropriate partition using the same hash function used to partition the `Messages` dataset and the `Messages` search key (the `reply_id` field of our previous message m_{prev}). m_{prev} is then used to probe the `Messages` dataset for reachable messages `m` using the `PIDX SEARCH` operator. After performing this primary index search, the record `ro` is built to capture this traversal of a `REPLY_OF`-labeled edge. To avoid cycles that may emerge from the underlying data, the subsequent `SELECT` operator is used to check if `m` or `ro` already exist in r_{prev}. The topmost `ASSIGN` then assembles the path `r` using `m`, `ro`, and r_{prev}. Finally, tuples are forwarded downstream to the `MESSAGE SINK` operator *and* back to the recursive input of the `FIXED POINT` for instances of the previous path to grow in length.

Each activity in the loop ($a \in A_{loop}$, where A_{loop} represents a cyclic activity group) is decorated with a "`MESSAGE AWARE`" wrapper. A decorated activity proxies the `open()`, `close()`, and `flush()` methods of a, but alters the functionality of `nextFrame(f)`. Upon receiving a message frame it: 1) calls a's `flush()` method; 2) colors the marker of the message frame if the previous `flush()` call sent any tuples downstream; and 3) forwards the potentially modified message frame to its downstream consumer. To localize the use of message frames (to avoid having to decorate *all* downstream activities in a plan with `MESSAGE AWARE`), a `MESSAGE SINK` operator is used to forward only the data frames to the `RESULT SINK` operator.

Note that the "modify-and-forward" action that each decorated activity performs here for message frames allows the `FIXED POINT` operator to reason about the status of all activities in the loop after generating the message frames. If the `FIXED POINT` operator receives a message frame containing a marker that has been colored, then it knows that at least one activity within the loop has generated more tuples (thus, it would be erroneous to call `close()`). In this case of a frame with a colored marker, `FIXED POINT` generates a new message frame containing the `RELEASE` directive and an uncolored marker to push downstream. If `FIXED POINT` receives a message frame containing a marker that has not been colored, then it can conclude that the loop has no tuples left to process *locally* (see the next section for reasoning about the distributed case). When `FIXED POINT` has no tuples left to process, `FIXED POINT` calls the `close()` method of its downstream consumer to close all activities within the loop.

3.3 Distributed Termination

The liveness and safety properties of the previous section do not require coordination from activity instances of other partitions. The mortality property, however, requires the consideration of *all* activity instances across *all* partitions to avoid premature / incorrect calls to `close()`. Similar to how we designated the `FIXED POINT` operator to manage the termination of all activities in a cyclic activity group, we designate one `FIXED POINT` instance out of all `FIXED POINT` instances to coordinate the termination for *every* cyclic activity group. By default in Graphix, we designate the `FIXED POINT` in the first machine (node controller #1 in Figure 3) as the "coordinator" to manage this activity cluster state. To facilitate communication between each `FIXED POINT` across machines, a custom communication channel is used between the coordinating `FIXED POINT` instance and the other `FIXED POINT` instances (depicted in Figure 3 by the double pink dotted lines)[†]. To minimize the network chatter between machines and to reduce the message-to-data-frame ratio during runtime, message frames remain local; they *do not* travel across the network at partitioned or broadcast connectors.

[†]In practice, this communication channel between different `FIXED POINT` instances is realized using an M:N hash-partitioned connector with a self-loop connecting the `FIXED POINT` back to itself. Consequently, we did not need to modify the task distribution infrastructure built for AsterixDB.

Algorithm 1 Algorithm used by the `FIXED POINT` coordinator to terminate all `FIXED POINT` instances.

```
1:  wait until REQ is received from all participants

2:  broadcast VOTE_ON_A to all participants          ▷ Parallel voting!
3:  while not all participants have voted with ACK_A do
4:      if any participant responds with NACK_A then
5:          broadcast CONTINUE to all participants
6:          goto line 1

7:  for all participant ∈ participants do            ▷ Sequential voting!
8:      send VOTE_ON_B to the participant
9:      if the participant responds with NACK_B then
10:         broadcast CONTINUE to all participants
11:         goto line 1

12: broadcast TERMINATE to all participants
```

We now move to the actions the coordinator `FIXED POINT` must take to inform each "participant" `FIXED POINT` that it can safely terminate. Algorithm 1 depicts the process the coordinator performs. The coordinator starts by waiting for each participant to send a "request-to-terminate" (dubbed the `REQ` event). This `REQ` event is given to the coordinator by a participant when the participant observes that it has no tuples (we detail the participant algorithm next). When the coordinator receives a `REQ` event from each participant, the coordinator can conclude that each participant has observed a lack of tuples in its own partition *for some instant*. Calling `close()` now would be erroneous, however, because partitions pass tuples to other partitions *asynchronously*. We can easily visualize an example where a participant sends `REQ` to the coordinator, only to receive more tuples immediately after transmitting its status. To handle this asynchronous nature, the status-checking that each participant performs must inevitability be serialized in order to guarantee correctness [22, 28].

To minimize the impact of serialized participation, the status checking has been divided into two phases: the A phase and the B phase. At the start of the A phase (starting on Line 1 of Algorithm 1), the coordinator has received the `REQ` event from all participant. The coordinator then *broadcasts* the `VOTE_ON_A` event to each participant. A participant during the A phase responds with either `ACK_A` or `NACK_A`. If *any* participant responds with `NACK_A`, the coordinator i) eagerly informs all participants that their request-to-terminate is not granted and ii) returns to Line 1 to wait for all participants to send a new `REQ` event. The purpose of the A stage is to increase the liveness / throughput of the looping computation, as the message frame that each participant uses to check the status of its own partition also contains the `RELEASE` directive to flush the buffers of its corresponding activities. If all participants respond with `ACK_A`, then the coordinator moves to the B phase. The B phase (starting on Line 6) consists of serialized status checking, where each participant is issued a `VOTE_ON_B` event. If a participant responds with `NACK_B`, our coordinator informs all participants that their `REQ` is not granted and returns to Line 1. If all participants respond with `ACK_B`, our coordinator broadcasts the `TERMINATE` event to all participants to conclude the looping computation.

Algorithm 2 Algorithm used by every `FIXED POINT` participant to work with its coordinator to call `close()`.

```
1:  function CheckAndFlushPartition( )
2:      f ← uncolored marker frame with RELEASE directive
3:      push f downstream
4:      while f is not returned do
5:          forward all data frames downstream
6:      return f

7:  while anchor input is not closed do
8:      forward all frames downstream
9:  repeat
10:     f ← CheckAndFlushPartition( )
11: until f is uncolored
12: send REQ to coordinator

13: for phase ∈ [A, B] do
14:     wait for VOTE_ON_A / VOTE_ON_B from coordinator
15:     if CheckAndFlushPartition( ) is uncolored then
16:         send ACK_A / ACK_B to coordinator
17:     else
18:         send NACK_A / NACK_B to coordinator
19:         goto line 10          ▷ Eagerly continue local processing!
20:     wait for coordinator to respond
21:     if coordinator responds with CONTINUE then
22:         goto line 10

23: call close() downstream
```

To finish our discussion on the `FIXED POINT` operator, we describe the process that every `FIXED POINT` participant performs in Algorithm 2. Starting on Line 7, a participant does not perform any election related actions until all of its anchor input tuples have been exhausted. Before participating in the election, a participant simply forwards all frames from both inputs downstream. Once the upstream activity bound to the `FIXED POINT` operator's anchor input calls `close()`, the routine `CheckAndFlushPartition()` is invoked. This routine pushes a message frame f with the special `RELEASE` directive to mandate that recipient activities (i.e., those decorated with "`MESSAGE AWARE`") must invoke their own `flush()` method. The aforementioned "modify-and-then-forward" actions of every decorated activity in the loop will eventually move f back to the `FIXED POINT` operator (via the `FIXED POINT` operator's recursive input). If f is colored, then a `FIXED POINT` instance cannot safely conclude that there is no work on its local partition. While `FIXED POINT` is waiting to receive f (on Line 4), data frames are forwarded downstream. On Line 9, `CheckAndFlushPartition()` is repeated until an uncolored marker is returned to the `FIXED POINT`. We note that a partition will never have more than one message frame in circulation along with its data frames, thus minimizing the message-to-data-frame ratio.

Once a `FIXED POINT` participant receives an uncolored marker, the participant sends a `REQ` event to the election coordinator. On Line 13, our participant enters the A phase. The `FIXED POINT` participant waits for a `VOTE_ON_A` message from the coordinator, at which it performs the `CheckAndFlushPartition()` routine to check

	$n = 1$	$n = 2$	$n = 4$	$n = 8$
Workload Time (s)	15591.53	5300.60	2886.64	72.89
% from `BI-17`	99.9%	99.7%	99.3%	63.4%
% from `IS-*`	0.1%	0.3%	0.7%	36.6%
Overhead ($M/M+D$)	0.004	0.008	0.081	0.17

Table 1: Table enumerating the LDBC SNB workload execution times and per-node marker overhead (as a percentage of all frames during the progress determination phase) for Hyracks clusters of varying size (n).

the status of its local partition. If a colored marker is received as a result of this status checking, the participant responds with `NACK_A` and returns to Line 7. Otherwise (if an uncolored marker is received), the participant responds to the coordinator with `ACK_A` and waits for the coordinator's response. If the coordinator then responds with `CONTINUE`, then some other `FIXED POINT` participant has received a colored marker on its partition (thus, all participants subsequently return to Line 7). Otherwise, the B phase is entered for all participants. With respect to the `FIXED POINT` participant, there exists no functional difference between the A phase and the B phase. Once a `FIXED POINT` participant advances through both A and B phases, the participant can safely call `close()` downstream.

4 EVALUATION

In this section, we detail two experiments to answer a) "How performant is our solution?", b) "What overhead does the use of marker frames incur?", and c) "How asynchronous is our solution?" For a more comprehensive evaluation comparing the end-to-end performance of the Graphix extension with a modern graph database, Neo4j, see [14, 15].

4.1 Experimental Setup

Our experiments use a subset of the LDBC social network benchmark (abbrv. LDBC SNB) [4], which describes a set of operational and analytical queries about a social network. With respect to the structure of the social network graph, LDBC's data generator produces networks that adhere to the Homophily principle (i.e. persons with similar interests and behavior know each other) and with vertex degrees similar to Facebook. The specific graph used for our experiments possesses 3.7M vertices and 10.2M edges. For our workload, we chose three LDBC SNB queries whose query plans in Graphix utilize the `FIXED POINT` operator (i.e., those that require unbounded recursion): `IS-2`, `IS-6`, and `BI-17`, where `IS` refers to the "short-interactive" queries of the benchmark and `BI` refers to the "business intelligence" queries of the benchmark. A total of 110 individual executions comprise the full workload: a) 50 executions of query `IS-2`, b) 50 executions of query `IS-6`, and c) 10 executions of query `BI-17`. All artifacts used for the experiments in this paper can be found at: https://github.com/graphix-asterixdb/benchmark.

4.2 Measuring Overhead

We begin our discussion with the sum of execution times of all three queries (i.e., all 110 executions) for Hyracks clusters of sizes

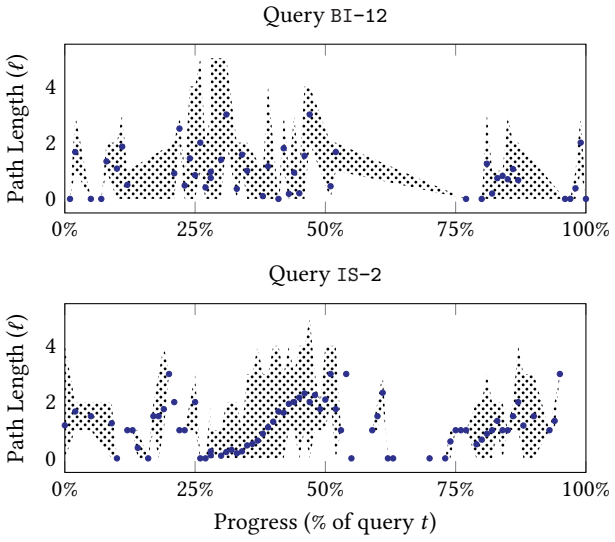

Figure 4: Plots illustrating the average (in blue), minimum, and maximum path lengths over the per-query progress for our LDBC SNB workload running on a cluster of size $n = 8$.

$n = 1$ to $n = 8$ in Table 1. We observe a roughly 3x speedup from $n = 1$ to $n = 2$, a 2x speedup from $n = 2$ to $n = 4$, and a 39x speedup from $n = 4$ to $n = 8$. The large scale up factor from $n = 4$ to $n = 8$ is because all queries are able to execute entirely in memory when $n = 8$. As would be expected, most of the execution time for each cluster configuration is dominated by the analytical query `BI-17`. The `BI-17` query plan (compiled by Graphix, before optimization by Algebricks) involves two `FIXED POINT` operators and nine `JOIN` operators. Furthermore, `BI-17` accesses a larger portion of the entire graph when compared to `IS-2` and `IS-6`. At $n = 8$, the impact of our experiment logging (used to log each frame and each individual path specifically for this paper) starts to dominate the "actual" execution time (i.e., the time without our experiment logging). For a more thorough discussion on end-to-end performance without this logging impact, we refer the interested reader to [14, 15].

Table 1 also shows the average percentage of marker frames M to data frames D processed per-node *during the progress determination phase* for Hyracks clusters of $n = 1$ to $n = 8$. As a reminder, marker frames are not sent between nodes (so the overhead here is local to each node). The ratio of marker frames increases with cluster size for the *same* volume of graph data. At $n = 1$, only 0.4% of frames circulated are marker frames. For the same workload at $n = 8$, roughly 17% of the frames locally circulated are markers. This is because less data exists per node, but markers still need to be circulated during the progress determination phase to correctly terminate. We note that each marker frame possesses $F - 11$ bytes of unused data, where F refers to the frame size (by default $F = 32$ KB). This overhead, however, does not grow with the graph data size, as only one marker frame is ever in circulation per node. As shown by the end-to-end execution times, the performance gains enabled by partitioned-parallelism through Hyracks outweigh this overhead.

4.3 Measuring Asynchronicity

For our second set of experiments, we are interested in determining whether or not our solution is indeed able to capitalize on the independent nature of path finding. To quantify asynchronicity, we will focus on the length ℓ of a path. Every time a path instance completes a cycle around a cyclic activity graph, the path instance's length grows by one. For example, paths bound to `r` in Figure 3 grow by one in length every time the path is processed by the topmost `ASSIGN`.

In Figure 4, we display the range of *concurrently* existing path lengths ℓ over the progress (as a percentage) of queries `BI-17` and `IS-2` executed on a Hyracks cluster of size $n = 8$. Data points for ℓ are recorded whenever paths are created or incremented in length (via the `APPEND_TO_PATH` function, illustrated in Figure 3). Visually, a larger area (i.e., more black dots) indicates more asynchronicity. In contrast, a bulk-synchronous approach would possess no area since the length of all of its paths would grow together. The execution of both `BI-17` and `IS-2` exhibit asynchronicity, with `BI-17` possessing an average ℓ range of 1.81 and `IS-2` possessing an average ℓ range width of 1.54. In the case of query `IS-2`, we observe that paths of length 4 are computed almost immediately (within the first 1% of the query). All paths that can be computed locally (per-node) can be computed without waiting for other nodes, minimizing the "endgame" impact of synchronous-based approaches such as [1].

5 RELATED WORK

Efficient evaluation of recursive queries has long been an active field of research. As a reminder, our work specifically targets ad-hoc navigational queries over virtual property graphs distributed across a shared-nothing cluster of workers.

We start with the established 1999 SQL standard which expresses linear recursion through recursive CTEs (common table expression) [19]. A recursive CTE is composed of two subqueries: i) the (non-recursive) anchor member and ii) the recursive member. All navigational queries that do not involve backtracking can be represented using recursive CTEs. In Equation 1, we represent the evaluation of a recursive CTE R as our anchor member a and the repeated union of our recursive member $r(R_{t-1})$ (where r is a function of the previous iteration of R) until $r(R_{t-1})$ yields no new results:

$$\begin{aligned} t_0 &: R \Leftarrow a \\ t_1 \rightarrow t_N &: R \Leftarrow R \cup r(R) \end{aligned} \tag{1}$$

Naive evaluation of recursive CTEs involves the sequential bottom-up execution of all steps in Equation 1. Naive evaluation ultimately incurs duplicate work due to the use of the entire R as input to the recursive member r for each iteration. Semi-naive evaluation [25] avoids this duplicate work by first maintaining a set that possesses newly generated tuples from the previous iteration: $\Delta_t = R_t - R_{t-1}$. Δ_t is then used in place of R as input to the recursive member r. Systems like Postgres realize recursive CTEs using semi-naive evaluation (where Δ_t is a table that is updated per iteration) [24].

Navigational queries can also be expressed in Datalog, thus we point to systems like LogicBlox [6], BigDatalog [27], and Soufflé [26] for evaluating navigational queries on top of logic databases. BigDatalog, Souffle, and early versions of LogicBlox all use semi-naive evaluation as their core execution method. With respect to general execution engines like Hyracks, systems like Naiad [23] and Myria [17] have been purposed to execute Datalog programs. In terms of architecture, BigDatalog, Naiad, and Myria are able operate on a shared-nothing cluster of workers (enabling scale-out like Hyracks). We draw analogs to these distributed Datalog systems and to the graph processing systems mentioned in Section 1 (Pregel [21] and Giraph [5]) that enable massively parallel execution of graph algorithms in a bulk-synchronous-parallel (BSP) fashion [30]. We note that systems that accept recursive CTEs, Datalog programs, and Pregel-like programs ultimately target a broader class of queries that we are interested in, hindering the sequential and independent properties of the navigational query subclass. GiraphUC [18] and Mitos [16] are other systems that leverage similar independence properties, though for more general recursion. GiraphUC in particular is a graph processing system that focuses on relaxing the synchronization required for problems like navigational queries, but it lacks important features found in more complete database systems (e.g., a declarative query language, comprehensive query optimization, secondary indexes, etc...).

6 CONCLUSION

In this paper we have detailed how the Hyracks engine of AsterixDB was extended to handle navigational queries in a partitioned-parallel and pipelineable manner. Existing solutions for partitioned-parallel recursion have not targeted navigation, leading to less-than-optimal executions that do not leverage path-finding's independence property.

To extend Hyracks, which was originally purposed for accepting a directed *acyclic* graph of operators, we identified three problems that emerge in directed *cyclic* operator graphs: a) liveness (the lack of progress due to partial output buffers), b) safety (the over-allocation of resources), and c) mortality (the inability to safely terminate). A `FIXED POINT` operator and a marker-aware task decorator were added to Hyracks to solve the aforementioned problems, enabling the execution of navigational queries for Graphix. As shown by our evaluation, our solution is able to capitalize on the independent nature of path finding. Note that although this work was done within the context of Hyracks and Graphix, many of the concepts explored here could be applied to other systems with data-parallel execution engines. We invite readers to try Graphix at https://graphix.ics.uci.edu.

ACKNOWLEDGMENTS

We would like to acknowledge Vinayak Borkar for his input on how ad-hoc recursion in Hyracks should be handled. We would also like to thank (former) UCI student Sushrut Borkar for his help writing the queries used in our evaluation. This research was supported in part by NSF awards IIS-1838248, IIS-1954962, and CNS-1925610, by the HPI Research Center in Machine Learning and Data Science at UC Irvine, and by the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] Foto N. Afrati, Vinayak Borkar, Michael Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. 2011. Map-Reduce Extensions and Recursive Queries. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala,

Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1951365.1951367

[2] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment* 7 (2014), 1905–1916.

[3] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (June 2014), 841–852. https://doi.org/10.14778/2732951.2732958

[4] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *ArXiv* abs/2001.02299 (2020). https://api.semanticscholar.org/CorpusID:264427448

[5] Apache Giraph. [n.d.]. Apache Giraph, an Iterative Graph Processing System Built for High Scalability. Available at https://giraph.apache.org.

[6] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. https://doi.org/10.1145/2723372.2742796

[7] Vinayak Borkar, Yingyi Bu, E. Preston Carman, Nicola Onose, Till Westmann, Pouria Pirzadeh, Michael J. Carey, and Vassilis J. Tsotras. 2015. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 422–433. https://doi.org/10.1145/2806777.2806941

[8] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 1151–1162. https://doi.org/10.1109/ICDE.2011.5767921

[9] Yingyi Bu, Vinayak R. Borkar, Jianfeng Jia, Michael J. Carey, and Tyson Condie. 2014. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *Proceedings of the VLDB Endowment* 8 (2014), 161–172.

[10] Michael J. Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 5501–5510. https://doi.org/10.1109/ICDE60146.2024.00438

[11] Badrish Chandramouli, Jonathan Goldstein, and David Maier. 2009. On-the-Fly Progress Detection in Iterative Stream Queries. *Proceedings of the VLDB Endowment* 2, 1 (aug 2009), 241–252. https://doi.org/10.14778/1687627.1687655

[12] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2246–2258. https://doi.org/10.1145/3514221.3526057

[13] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[14] Glenn Galvizo. 2023. *Graphix: View the (JSON) World Through Graph-Tinted Lenses.* PhD Thesis. University of California, Irvine, Irvine, CA.

[15] Glenn Galvizo and Michael J. Carey. 2024. Graphix: "One User's JSON is Another User's Graph" . In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3070–3083. https://doi.org/10.1109/ICDE60146.2024.00238

[16] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1428–1439. https://doi.org/10.1109/ICDE51399.2021.00127

[17] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. 2014. Demonstration of the Myria Big Data Management Service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 881–884. https://doi.org/10.1145/2588555.2594530

[18] Minyang Han and Khuzaima S. Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proceedings of the VLDB Endowment* 8 (2015), 950–961.

[19] ISO Central Secretary. 1999. *Information Technology — Database Languages — SQL — Part 2: Foundation (SQL / Foundation)*. Standard ISO/IEC 9075-2:1999. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/62711.html

[20] ISO/IEC. [n.d.]. Graph Query Language GQL Standard. Available at https://www.gqlstandards.org.

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a System for Large-Scale Graph Processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010).

[22] Jayadev Misra. 1983. Detecting Termination of Distributed Computations Using Markers. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. 290–294.

[23] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[24] PostgreSQL. 02-05-2025. 7.8.2 Recursive Queries in WITH Queries (Common Table Expressions). https://www.postgresql.org/docs/current/queries-with.html#QUERIES-WITH-RECURSIVE.

[25] Y. Sagiv. 1987. Optimizing Datalog Programs. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (San Diego, California, USA) *(Principles of Database Systems '87)*. Association for Computing Machinery, New York, NY, USA, 349–362. https://doi.org/10.1145/28659.28696

[26] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[27] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[28] Rodney W. Topor. 1984. Termination Detection for Distributed Computations. *Inform. Process. Lett.* 18, 1 (1984), 33–36. https://doi.org/10.1016/0020-0190(84)90071-1

[29] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering* 15 (2003), 555–568. https://api.semanticscholar.org/CorpusID:3155983

[30] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends in Databases* 7, 1–2 (Jan 2017), 1–195. https://doi.org/10.1561/1900000056