Title

Graphix: View the (JSON) World Through Graph-Tinted Glasses

Author
Galvizo, Glenn Justo

Publication Date
2023

UNIVERSITY OF CALIFORNIA,
IRVINE

Graphix: View the (JSON) World Through Graph-Tinted Glasses

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Glenn Justo Galvizo

Dissertation Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Assistant Professor Faisal Nawab
Dmitry Lychagin, Apache AsterixDB Project

2023

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF CODE LISTINGS **vii**

LIST OF TABLES **ix**

ACKNOWLEDGMENTS **x**

VITA **xi**

ABSTRACT OF THE THESIS **xii**

1 Introduction **1**

2 Related Work **4**
2.1 Graph Processing Systems 4
2.2 Native Graph Databases 5
2.3 Database Graph Extensions 6

3 Background **9**
3.1 Apache AsterixDB 9
3.2 Social Network Example 10
3.3 SQL for JSON: SQL^{++} 12

4 Graph Model **16**
4.1 Property Graph Model 17
4.2 `CREATE GRAPH` Statement 19
4.2.1 Social Network Example 23
4.2.2 Multiple Dataset Example 26
4.2.3 Derived Property Example 29

5 Query Model **32**
5.1 SQL^{++} Query Extension 32
5.1.1 SQL-1999 Recursive Queries 33
5.1.2 Cypher Query Language 35
5.1.3 SQL-2023 Property Graph Queries 37

5.1.4 gSQL^{++} `FROM` Clause Extension . . . 39
5.2 Pattern Matching Queries . . . 41
5.2.1 Graph Pattern Matching . . . 41
5.2.2 gSQL^{++} for Pattern Matching . . . 44
5.3 Navigational Queries . . . 50
5.3.1 Path Finding (Navigation) . . . 51
5.3.2 gSQL^{++} for Navigation . . . 56
5.4 Complex gSQL^{++} Examples . . . 60
5.4.1 Optional Subgraph Matching . . . 60
5.4.2 Negative Subgraph Matching . . . 62
5.4.3 Subgraph Reachability . . . 64
5.4.4 Shortest Path Finding . . . 66
5.4.5 Cheapest Path Finding . . . 68

6 Implementation **72**
6.1 Graphix Architecture . . . 72
6.1.1 `CREATE GRAPH` Lifecycle . . . 75
6.1.2 gSQL^{++} Query Lifecycle . . . 77
6.2 Hyracks Runtime Engine . . . 79
6.2.1 Hyracks by Example . . . 80
6.2.2 Recursion Foundations . . . 93
6.2.3 Property #1: Liveness . . . 97
6.2.4 Property #2: Safety . . . 99
6.2.5 Property #3: Mortality . . . 101
6.2.6 Fixed Point Operator (1-Machine) . . . 103
6.2.7 Fixed Point Operator (n-Machines) . . . 106
6.2.8 Additional Hyracks Operators . . . 115
6.2.9 "Paths Not Traveled" (Alternatives) . . . 122
6.3 Abstract Syntax Tree Rewriter . . . 126
6.3.1 gSQL^{++} AST Rewriting . . . 127
6.3.2 gSQL^{++} Lowering to SQL^{++} . . . 130
6.4 Algebricks Query Optimizer . . . 137

7 Evaluation **143**
7.1 Experimental Setup . . . 143
7.2 Operational `IS-X` Queries . . . 146
7.3 Operational `IC-X` Queries . . . 149
7.4 Analytical `BI-X` Queries . . . 156

8 Conclusion **164**
8.1 Conclusion . . . 165
8.2 Future Work . . . 167

Bibliography **169**

Appendix A Benchmark Detail **176**
A.1 Graphix DDLs 176
A.2 Graphix Queries (in gSQL^{++}) 187

LIST OF FIGURES

Page

2.1 Comparison of different architectures for processing graphs. 7

3.1 Illustration of documents in the example social network database. 10
3.2 SQL^{++} railroad diagram illustrating the `Selection` production. 13

4.1 Illustration of "disconnected" edges that *may* exist in a graph. 17
4.2 Starting productions (as railroad diagrams) for defining a graph in Graphix. 20
4.3 Graphix railroad diagram illustrating the `VertexConstructor` production. . . 21
4.4 Graphix railroad diagram illustrating the `EdgeConstructor` production. . . . 22

5.1 gSQL^{++} railroad diagram describing the `FROM` clause extension. 39
5.2 Illustration of an graph instance and a graph query pattern. 42
5.3 gSQL^{++} railroad diagram detailing the productions for pattern matching. . . 45
5.4 gSQL^{++} railroad diagram detailing the productions for vertex patterns. . . . 46
5.5 gSQL^{++} railroad diagram detailing the productions for edge patterns. 47
5.6 Illustration of an edge labeled graph instance (used in Table 5.2). 54
5.7 gSQL^{++} railroad diagram detailing the productions for path patterns. 56
5.8 Illustration of a graph instance (used in Subsection 5.4.4). 67
5.9 Depiction of shortest path finding with the `GROUP BY` and `GROUP AS` clauses. . 68

6.1 Diagram illustrating the Graphix and AsterixDB software stack. 73
6.2 Architecture diagram detailing the processes in a two-node Graphix cluster. . 74
6.3 Illustration of different units of work in a Hyracks job. 80
6.4 Graph of Hyracks activities that execute a 1-hop query. 82
6.5 Graph of Hyracks tasks that execute a 1-hop query. 85
6.6 Graph of Hyracks activities that execute a 3-hop query. 87
6.7 Graph of Hyracks activities that execute a 1-to-3-hop query. 89
6.8 Graph of Hyracks activities that execute a 1-to-3-hop query without sorting. 90
6.9 Visualization of independent path growth in a Hyracks activity graph. 92
6.10 High level graph of Hyracks activities for executing unbounded recursion. . . 95
6.11 Illustration of a task forwarding its output buffer to another task. 96
6.12 Illustration of a mechanism to prevent liveness violations. 98
6.13 Illustration of a mechanism to prevent safety violations. 100
6.14 Illustration of a mechanism to prevent mortality violations. 102
6.15 Graph of Hyracks activities that execute an unbounded path query. 104

6.16 Diagram detailing the decoration of an activity instance. 105
6.17 Graph of three Hyracks task clusters executing some cyclic computation. . . 107
6.18 Diagram detailing the internal processes of the `FIXED POINT` operator. 109
6.19 Algorithm (as a FSM) describing the actions of a `FIXED POINT` coordinator. . 111
6.20 Algorithm (as a FSM) describing the actions of a `FIXED POINT` participant. . 114
6.21 Activity graph that illustrates how the `PBJ` (`JOIN`) operator executes. 117
6.22 Activity graph that illustrates how the `TOP K` operator executes. 120
6.23 Potential alternative solution #1 to realize recursion in Hyracks. 123
6.24 Potential alternative solution #2 to realize recursion in Hyracks. 125
6.25 Demonstration of the shared vertex pattern AST rewrite. 129
6.26 Transformation of a gSQL^{++} query into an equivalent SQL^{++} query. 132
6.27 Transformation of a gSQL^{++} query into a *nearly* equivalent SQL^{++} query. . 134
6.28 Description of the anchor and recursive members of a navigational query. . . 136
6.29 Graph of Algebricks operators to realize a navigational query. 139

7.1 LDBC social network database (SNB) schema diagram. 144
7.2 Plots comparing Graphix vs. Neo4j for queries `IS-X` at `SF=1`. 147
7.3 Plots comparing Graphix vs. Neo4j for queries `IS-X` at `SF=100`. 147
7.4 Plots comparing Graphix vs. Neo4j for queries `IC-X` at `SF=1`. 150
7.5 Plots comparing Graphix vs. Neo4j for queries `IC-X` at `SF=100`. 151
7.6 Plots comparing Graphix vs. Neo4j for queries `BI-X` at `SF=1`. 160
7.7 Plots comparing Graphix vs. Neo4j for queries `BI-X` at `SF=100`. 161

LIST OF CODE LISTINGS

3.1 Set of “schema-first” dataset DDLs (CREATE TYPE and CREATE DATASET). . . . 11
3.2 Set of “schema-never” dataset DDLs (CREATE TYPE and CREATE DATASET). . . 12
3.3 SQL^{++} query that correlates two datasets in the SELECT clause. 14
3.4 Example set of JSON results for the query in Listing 3.3. 14
3.5 SQL^{++} GROUP AS query to return groups formed by a GROUP BY clause. 15
3.6 Example set of JSON results for the query in Listing 3.5. 15

4.1 Graphix CREATE GRAPH DDL to define a property graph view. 24
4.2 Set of dataset DDLs used to define additional datasets for Subsection 4.2.2. . 27
4.3 VERTEX definition of a CREATE GRAPH DDL that maps two datasets. 27
4.4 EDGE definition of a CREATE GRAPH DDL that maps two datasets. 28
4.5 SQL^{++} query to compute a weight property between two users. 30
4.6 EDGE definition of a CREATE GRAPH DDL that includes a computed property. . 30

5.1 Recursive SQL query that computes the transitive closure for three users. . . 34
5.2 Cypher query that computes the transitive closure for three users. 36
5.3 SQL/PGQ graph creation DDL for the graph used in Listing 5.4. 38
5.4 SQL/PGQ query that computes the transitive closure for three users. 38
5.5 gSQL^{++} query that computes the transitive closure for three users. 40
5.6 gSQL^{++} query that specifies a graph pattern in the FROM clause. 49
5.7 Example result found in the result set of the query in Listing 5.6. 49
5.8 gSQL^{++} query that specifies a RPQ (path pattern) in the FROM clause. 58
5.9 Example result found in the result set of the query in Listing 5.8. 58
5.10 gSQL^{++} query that specifies optional pattern matching with LEFT MATCH. . . 61
5.11 gSQL^{++} query that specifies optional pattern matching with LEFT JOIN. . . . 61
5.12 Example set of JSON results for the queries in Listing 5.10 and Listing 5.11. 61
5.13 gSQL^{++} query that specifies negative pattern matching. 63
5.14 Alternative query to Listing 5.13 that explicitly JOINs vertex patterns. 63
5.15 Example set of JSON results for the queries in Listing 5.13 and Listing 5.14. 63
5.16 gSQL^{++} query that determines the reachability between vertex groups. . . . 65
5.17 Alternative to Listing 5.16 that uses GROUP BY instead of SELECT DISTINCT. . 65
5.18 Example set of JSON results for the queries in Listing 5.16 and Listing 5.17. 65
5.19 gSQL^{++} query that determines the shortest path between vertex groups. . . 67
5.20 gSQL^{++} query that determines the cheapest path between vertex groups. . . 69

6.1 gSQL^{++} query that specifies an unbounded path of REPLY_OF edges. 79
6.2 gSQL^{++} query that specifies a path of exactly 1 REPLY_OF edge. 82

6.3 gSQL^{++} query that specifies a path of exactly 3 `REPLY_OF` edges. 87
6.4 gSQL^{++} query that specifies a path of 1 to 3 `REPLY_OF` edges. 89
6.5 gSQL^{++} query that specifies the shortest path of `REPLY_OF` edges. 121
6.6 "SQL^{++}"-like translation of a navigational pattern matching query. 137

LIST OF TABLES

Page

5.1 Table describing different morphism classes for some example graph instance. 43
5.2 Table enumerating all "non-repeat-edge" paths for some example graph instance. 55

6.1 Table summarizing the notation used for all graphs of Hyracks activities. . . 83

7.1 Table comparing Graphix vs. Neo4j for queries `IS-X` and `IC-X` at `SF=1`. 153
7.2 Table comparing Graphix vs. Neo4j for queries `IS-X` and `IC-X` at `SF=100`. . . 154
7.3 Table comparing Graphix vs. Neo4j for queries `BI-X` at `SF=1`. 162
7.4 Table comparing Graphix vs. Neo4j for queries `BI-X` at `SF=100`. 163

ACKNOWLEDGMENTS

I could not have undertaken this journey without my advisor, Professor Michael J. Carey. His consistent positive attitude and patience made the weekly meetings we had an hour I could always look forward to. In addition to technical knowledge he brought to every conversation, Professor Carey always made these conversations "fun" and engaging. He has been an amazing advisor that has not only deepened my knowledge and appreciation for databases, but has helped shape me into the individual I am today.

Sincere thanks to Professor Chen Li and Professor Faisal Nawab for providing me with invaluable feedback during my many Graphix presentations and really helping me further my communication skills. I am also grateful to Dmitry Lychagin for joining my dissertation committee. His help in reviewing the code I pushed to the AsterixDB codebase and his guidance during my internship at Couchbase advanced my software engineering skills.

Special thanks goes to Vinayak Borkar for his help in formulating ideas for realizing recursion in Hyracks. Thanks should also go to Professor Yannis Papakonstantinou for his review of the first Graphix paper and his continued support after presenting my work at UC San Diego. Many thanks to Amarnath Gupta and Professor Subhasis Dasgupta from UC San Diego for their input on the first Graphix paper as well. I would also like to acknowledge UCI student Sushrut Borkar for his help "alpha-testing" early versions of Graphix. Thanks to Rodney (JT Douglas) for his work on the Graphix logo.

I am also thankful for my friends and colleagues from the Information Systems Group and the HPI Research Center at UC Irvine. The weekly reading groups, the various events, and the interactions we had has helped sustain me throughout this endeavor. Thank you Eliot Wong-Toi, Harry Bendekgey, Gavin Kerrigan, Markelle Kelly, Federica Zoe Ricci, Pratyoy Das, Vishal Chakraborty, Sadeem Saleh Alsudais, Yiming Lin, Nada Lahjouji, Ashwin Gerard Colaco, and every other ISG + HPI @ UC Irvine student.

I'd like to acknowledge Ian Maxon and Wail Alkowaileet for helping me with the Apache processes and also being wonderful people to chat and drink with. Lastly, I'd like to mention my colleagues (from both the AsterixDB project and Couchbase). Thank you Shiva Jahangiri, Chen Luo, Gift Sinthong, Thomas Hütter, Till Westmann, Ali Alsuliman, Vijay Sarathy, and Murali Krishna.

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).

VITA

Glenn Justo Galvizo

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, California*

Masters of Science in Computer Science **2021**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science **2019**
University of Hawaii at Manoa *Honolulu, Hawaii*

PUBLICATIONS

Multi-valued Indexing in Apache AsterixDB **Jan 2023**
Information Systems (Special issue on DOLAP 2022: Design, Optimization, Languages and Analytical Processing of Big Data)

On Multi-Valued Indexing in AsterixDB **Mar 2022**
International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) co-located with the 25th International Conference on Extending Database Technology

SOFTWARE

Graphix Extension `https://graphix.ics.uci.edu/`
An extension for Apache AsterixDB that enables navigational pattern matching queries on a property graph view of data in AsterixDB, in-situ.

Apache AsterixDB `https://asterixdb.apache.org/`
A scalable, open-source Big Data management system (BDMS) that provides storage, indexing, and management for large semi-structured data.

ABSTRACT OF THE DISSERTATION

Graphix: View the (JSON) World Through Graph-Tinted Glasses

By

Glenn Justo Galvizo

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Michael J. Carey, Chair

The increasing prevalence of large graph data has produced a variety of research and applications tailored toward graph data management. Users aiming to perform graph analytics will typically start by importing existing data into a separate graph-purposed storage engine. The cost of maintaining a separate system (e.g., the data copy, the associated queries, etc. . .) just for graph analytics may be prohibitive for users with Big Data. Furthermore, using separate systems for mixed-model analytics (e.g., JSON and graph) requires specialized solutions. In this thesis, we introduce Graphix and show how it enables property graph views of existing document data in AsterixDB, a Big Data management system boasting a partitioned-parallel query execution engine.

This thesis starts with a description of how Graphix property graphs naturally extend the AsterixDB document model to define vertices and edges. We detail how users can specify Graphix graphs in a manner that handles a wide variety of document-to-graph mappings while maintaining the schema-flexibility offered by AsterixDB. Next, we explain how users can query their Graphix graphs. The Graphix query language (gSQL^{++}) minimally extends AsterixDB's query language (SQL^{++}) to express synergistic graph and traditional (multi-model) analytics. After describing the user model aspects of Graphix, we detail how AsterixDB was extended to accommodate a recursive graph query construct: path finding.

We focus on how the AsterixDB runtime layer was extended to realize semi-synchronous, partitioned-parallel recursion. We later discuss how to extend the query optimization layer as well as the query parsing and AST rewriting layer to reuse as much of AsterixDB as possible. This thesis concludes with an evaluation of Graphix against a native graph database, Neo4j. We show that Graphix is able to scale horizontally to perform on-par with (and in some cases, even outperform) Neo4j for many kinds of operational and analytical queries — ultimately illustrating that users might not need a separate graph database just to issue graph queries.

Chapter 1

Introduction

Research in the field of graph data management has seen an explosion over the past decade. Teams developing applications with a graph-only workload in mind from the start have a large selection of graph databases to chose from. These types of users however, are not the norm — the typical user of a graph database also has non-graph-workloads that they must design around [60]. This design effort is further complicated when dealing with Big Data and out-of-core workloads. A common architecture that these types of users employ involves the stitching of multiple, more narrow-purpose systems together. For example, consider a two-DBMS (database management system) architecture composed of a document DBMS $\mathcal{D}$ and a graph DBMS $\mathcal{G}$ to analyze the relationships found in $\mathcal{D}$. This architecture has several consequences:

1. Some form of ETL (extract-transform-load) pipeline must be developed to duplicate the data from database $\mathcal{D}$ to $\mathcal{G}$ and then maintained to ensure consistency.
2. Additional storage, compute resources, development, and maintenance need to be allocated to accommodate both $\mathcal{D}$ and $\mathcal{G}$ (increasing the cost to own the data).
3. Multi-model workloads (i.e., analytics over $\mathcal{D}$ *and* $\mathcal{G}$) require specialized solutions.

Furthermore, graph databases like Neo4j are limited by their inability to scale outward, leaving users of such databases with few options when their queries run slower than desired (or not at all) due to excessive data volume. In this thesis, we challenge the two-DBMS architecture previously described. We describe the following desiderata for a new architecture that enables both graph and non-graph workloads:

In-Situ (Zero Copy) Query Processing ....

To avoid the complexities that come with creating and maintaining multiple copies of data, both users and systems *should not* duplicate data for the sole purpose of managing different user models.

Synergistic Graph and Traditional Analytics ....

Users familiar with one data model *should not* have additional barriers to work with other models of the same data. The "accidental complexity" involved in integrating multiple user models *should* be minimized.

Partitioned-Parallel, Scalable Execution ....

Users *should* be able to work with data volumes larger than memory. Users *should not* have to sacrifice performance to realize all of the aforementioned points.

We find that most existing solutions satisfy 1/3 of the points above. Graphix is our solution to satisfy these desiderata: it takes a view-based approach to answering graph queries on JSON data in-situ and at scale (i.e., to *view* the JSON world through "graph-tinted" glasses). The contributions of this work include: 1) a graph view user model and DDL that naturally extends an underlying document model, 2) a query extension for expressing graph and traditional (multi-model) analytics in synergy, 3) a description of how to translate navigational pattern matching queries into partitioned-parallel executions, and 4) a performance comparison with a native graph database.

The rest of this thesis is organized as follows: Chapter 2 describes related work around querying graph data. Chapter 3 reviews (i) Apache AsterixDB, the Big Data management system used for this research, (ii) AsterixDB's query language SQL^{++}, and (iii) a running social network example database. Chapter 4 introduces the graph model of Graphix, demonstrating how users can map existing data to a graph view. Chapter 5 details our query model and query language extension: gSQL^{++}. Chapter 6 explains the implementation and architecture of Graphix. Chapter 7 details an evaluation of Graphix against the native graph database Neo4j. Chapter 8 concludes this thesis and lists some potential future work.

Chapter 2

Related Work

The database community has had no shortage of work trying to tackle the management of large graphs. While many graph problems can (and have) been solved using non-graph-purposed systems, in this chapter we consider systems whose user model deals with graph-specific abstractions. Related work can be grouped into three areas: (i) graph processing systems, (ii) (native) graph databases, and (iii) database graph extensions for non-graph-purposed databases.

2.1 Graph Processing Systems

Big Graph processing systems such as Pregel [42] and Giraph [10] were designed to provide a vertex-oriented message-passing-based abstraction for distributed graph algorithms to run on shared-nothing clusters in a bulk-synchronous-parallel (BSP) fashion. Another system designed for graph processing is GraphX [26], which uses a simpler API (Resident Distributed Graphs, or RDGs) and adopts Spark as its runtime. In an effort to provide similar vertex-centric abstractions without the need for bulk synchronization, systems like GraphLab [41] and GiraphUC [30] were designed to process large graphs in an asynchronous manner. A graph processing system that used the same runtime engine as Graphix is Pregelix [13], designed to gracefully scale distributed graph algorithms for out-of-core workloads. While Big Graph processing systems have been shown to be highly performant and scalable [75], their "think like a vertex" paradigm still requires users to develop a program to interact with their APIs. We contrast graph processing systems with more traditional database systems, where a declarative query language like SQL is used to build ad-hoc queries with less developer effort. Our work is largely orthogonal to graph processing systems, as we target the specific problem of *navigational pattern matching* and not all *graph algorithms*. Keeping with the informal motto of AsterixDB, "one size fits a bunch", Graphix aims to target "a bunch" of use cases really well as opposed to targeting all use cases with a user-model impedance mismatch.

2.2 Native Graph Databases

Native graph databases like Neo4j [49] and TigerGraph [69] were designed to challenge traditional relational database systems by building a new database from the ground-up (storage, execution, and user model) with graph primitives in mind. Amazon Neptune [4], while not a *native* graph database (since it is built on top of AWS's existing data platforms), presents users with only a graph data model. The two leading graph user models are the property graph model and the resource description framework (RDF) graph model. In the property graph model, users reason about their data as a directed multi-graph of labeled vertices and edges, where each vertex and edge can possess a set of key-value pairs (known as properties). In the RDF model, users reason about their data as a directed graph of labeled edges captured in the form of subject-predicate-object triples. Property graphs have seen significantly more adoption and are supported by all three of the aforementioned systems [67].

With respect to the query model of graph databases, there are two leading query languages for the property graph model — Cypher [25] and Gremlin [57] — and one standardized language for the RDF model — namely SPARQL [55].

Use of graph databases requires users of existing non-graph-databases to build ETL pipelines to copy their data over to the chosen graph database. In addition to the increased cost to own the data, native graph databases like Neo4J are unable to scale horizontally. TigerGraph and Amazon Neptune are offerings that have the ability to scale horizontally, but they still suffer from the problem of requiring duplicate copies of data. In contrast, Graphix operates on existing data in-situ without a need to stitch separate systems together.

2.3 Database Graph Extensions

Work on extending existing, non-graph-purposed systems with graph extensions can be split into two areas: (i) re-purposing an existing system to handle a graph data model, and (ii) translating queries for a graph data model into the query model understood by an existing system. While the former (Item i) has seen a lot of interest [64, 38], these systems possess the same flaw as graph databases from the previous section: they require maintaining duplicate copies of existing data. We will focus on the latter work (Item ii) which most closely relates to Graphix. We give a high-level comparison between graph processing systems, graph (+ non-graph) database systems, and database graph extensions in Figure 2.1.

Unipop Graph [72] and Cytosm [25] are middleware systems that translate graph queries into queries for another system. Cytosm translates Cypher queries into queries on a relational store, but it does not support unbounded recursion. Unipop Graph translates Gremlin and SPARQL queries into one or more queries on a NoSQL or relational store, but it performs its joins outside of the underlying database. Neither project has had any updates in over 5

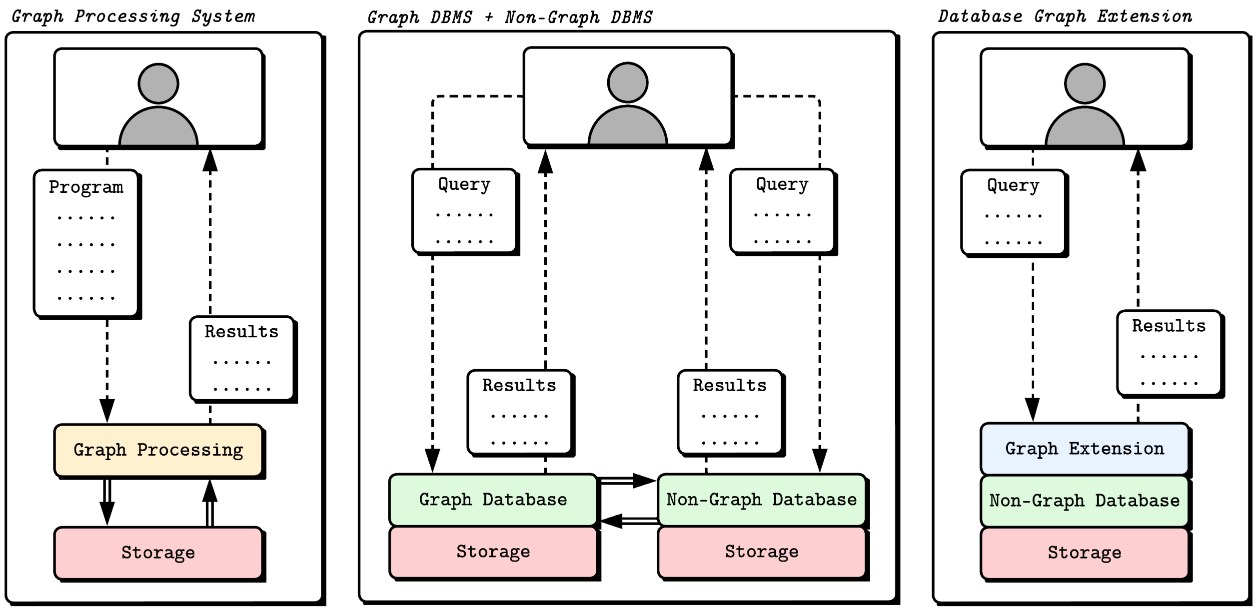


Figure 2.1: Comparison between graph processing systems, graph + non-graph database systems, and database graph extensions.

years. Graphix, on the other hand, is a derivative of AsterixDB, allowing it to a) perform joins closer to the data, and b) extend the optimizer and runtime to leverage information about the *original* graph query.

Prominent non-open-source offerings include Oracle Spatial and Graph [54], DataStax Enterprise Graph [20], and IBM Db2 Graph [68]. Oracle Spatial and Graph gives users the option to load their existing data into memory as a graph and issue their queries in-core, or to translate a limited subset of graph queries into equivalent SQL queries on existing data (allowing for out-of-core execution). DataStax Enterprise Graph allows users to query their underlying Cassandra (column family) store with Gremlin. While Cassandra has an excellent ability to scale outward, DataStax Enterprise Graph inherits its significant limitations for analytics (i.e., queries require careful physical tuning via index creation before being able to execute). IBM Db2 Graph, in contrast to the two aforementioned systems, was designed with a similar goal as Graphix: to allow users to execute both graph and relational analytics on existing data, in-situ. What Graphix does differently than IBM Db2 Graph is two-fold:

(1) Graphix users operate on a flexible underlying data model (i.e., a *document model* vs. a traditional *relational model*), simplifying the user model when reasoning over graphs and the source data. (2) Graphix presents a unified query model, allowing users to integrate navigational pattern matching with the underlying query language. The query model behind IBM Db2 Graph clearly separates its graph analytics component (written in Gremlin) and its relational analytics component (written in SQL), resulting in a less-than-synergistic user model.

Chapter 3

Background

3.1 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^{++} (detailed later this chapter in Section 3.3), a generalized form of SQL that enables the querying of semi-structured data, is used. To scale horizontally it follows a shared-nothing architecture, where each node independently accesses storage and memory. All nodes are managed by a central cluster controller that serves as an entry point for user requests and 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 [11]. This optimized physical plan is then translated into a job that can run across all nodes in the cluster [12]. Datasets in AsterixDB are hash-partitioned across the cluster on their primary key into primary B^{+} tree indexes, where the data records

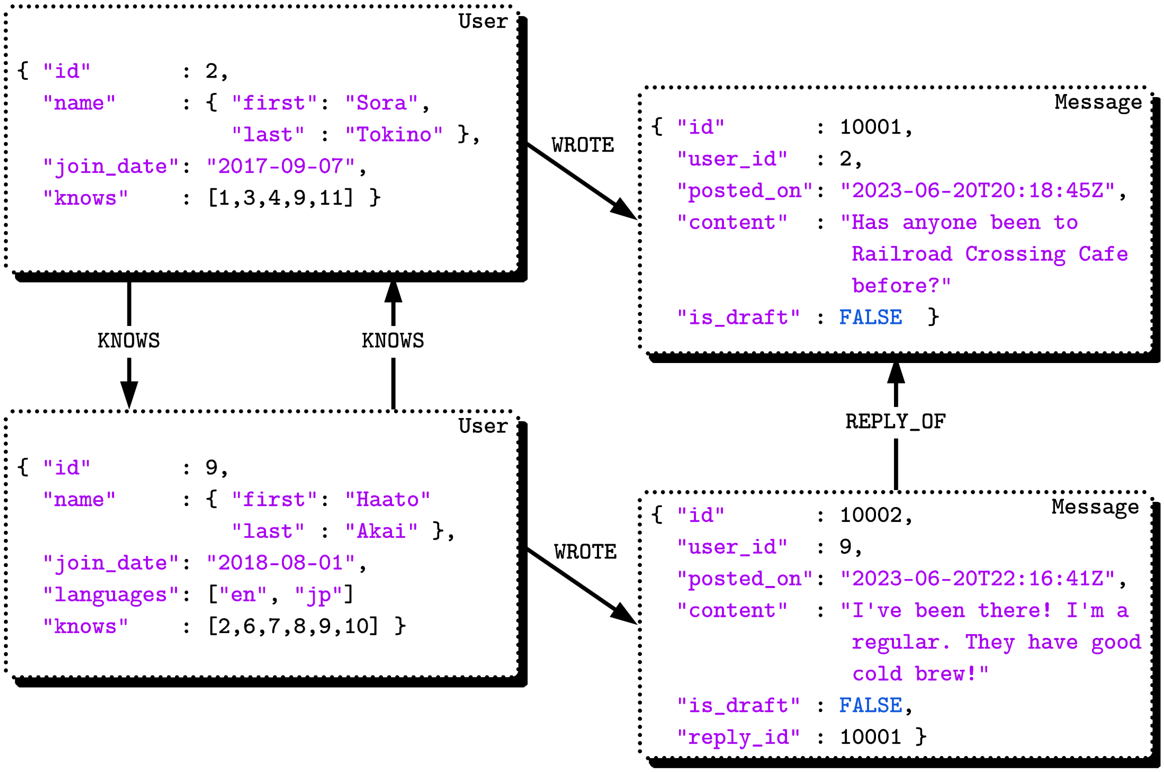


Figure 3.1: Example documents of two `Users`, two `Messages`, and their relationships.

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].

3.2 Social Network Example

To aid in illustrating several Graphix concepts, we introduce a running example: a social network. We start by designing our social network database as a collection of documents. Two major entities are captured in this example: (i) `Users` and (ii) `Messages`. Three relationships are captured in our social network: (I) a `User` may *post* one or more `Message(s)`, (II) a `Message` may *reply to* exactly one `Message`, and (III) a `User` may *know* one or more

```
CREATE TYPE UsersType AS {
    id        : bigint,
    name      : { first : string,
                  last  : string },
    join_date : string,
    languages : [string]?,
    knows     : [bigint]
};
CREATE DATASET Users (UsersType) PRIMARY KEY id;

CREATE TYPE MessagesType AS {
    id        : bigint,
    user_id   : bigint,
    posted_on : string,
    content   : string,
    is_draft  : boolean,
    reply_id  : bigint?
};
CREATE DATASET Messages (MessagesType) PRIMARY KEY id;
```

Listing 3.1: Set of "schema-first" dataset definitions for the `Users` and `Messages` datasets.

other `User(s)`. Examples of these entities and relationships are given in Figure 3.1. We highlight three parts of our social network schema that differ from a similar schema in the traditional relational model:

1. Data can be nested, as shown by the `name` field of the two `User` documents.
2. Many-to-many relationships can be folded into a single entity, as shown by the `knows` arrays of the two `User` documents.
3. A field present in one document of some collection may not be present in another document of that same collection, as shown by the `languages` field of the two `User` documents and the `reply_id` field of the two `Message` documents.

To describe the documents from Figure 3.1 in AsterixDB, we'll need to define the types of the `Users` and `Message` datasets. In Listing 3.1, we define the `Users` and `Messages` datasets using the `UsersType` and `MessagesType` respectively. The `UsersType` defines the mandatory fields `id`, `name`, `join_date`, and `knows` as well as their types. `UsersType` also has one optional field:

```
CREATE TYPE GenericType AS {
    _id: uuid
};
CREATE DATASET Users (GenericType) PRIMARY KEY _id AUTOGENERATED;
CREATE DATASET Messages (GenericType) PRIMARY KEY _id AUTOGENERATED;
```

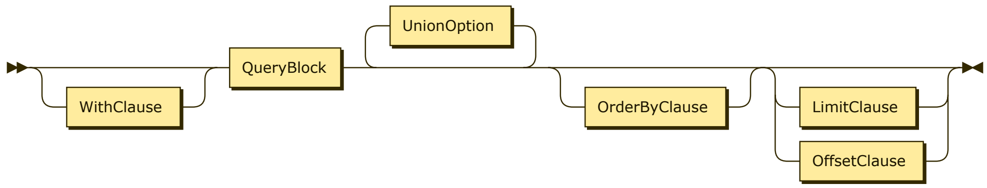
Listing 3.2: Set of "schema-never" dataset definitions for the `Users` and `Messages` datasets.

`languages`. In the case of AsterixDB, "optional" means that the `languages` field could be associated with a `NULL` value *or* the `languages` field could be absent (i.e., `MISSING`) from the document entirely. The `MessagesType` defines the mandatory fields `id`, `user_id`, `posted_on`, `content`, and `is_draft`. `MessagesType` also has one optional field: `reply_id`.

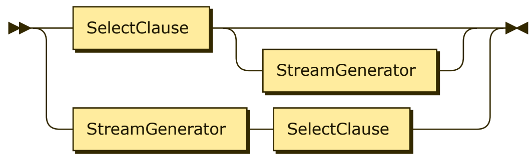
In Listing 3.2, we show how we could instead define the `Users` and `Messages` dataset with another set of DDLs. Here, both the `Users` and `Messages` datasets share the same `GenericType` in their definition. `GenericType` defines a `_id` field which serves as an auto-generated primary key for both `Users` and `Messages`. In contrast to the *schema-first* definition given by Listing 3.1, the DDLs in Listing 3.2 represent a *schema-never* approach to defining the social network datasets. AsterixDB accepts both definitions (as well as a range of possibilities in between), which means that Graphix must also accommodate the same range of schema flexibility.

3.3 SQL for JSON: SQL^{++}

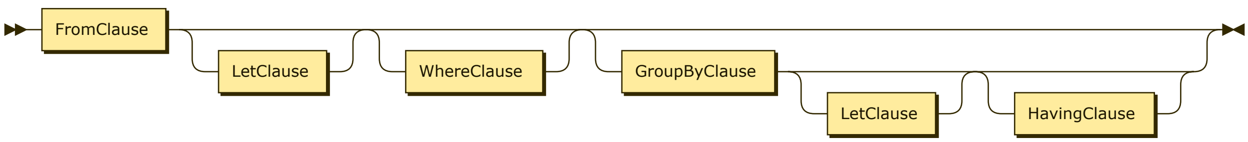
SQL^{++} is a query language purposed for JSON, semi-structured data, while being backwards-compatible with SQL [52, 14]. This backwards compatibility enables easy adoption by existing SQL users. A `SELECT` query in SQL^{++} is expressed using the `Selection` production in Figure 3.2a. Following Figure 3.2a from left to right, SQL^{++} users are able to: (a) express common table expressions (CTEs) using the `WITH` clause, (b) union results of queries using the `UNION ALL` operator, (c) sort records using the `ORDER BY` clause, and (d) limit the number



(a) SQL^{++} grammar for the `Selection` production.



(b) SQL^{++} grammar for the `QueryBlock` production.



(c) SQL^{++} grammar for the `StreamGenerator` production.

Figure 3.2: Grammar used to define a `Selection` in SQL^{++}.

of results using the **LIMIT** clause. The `QueryBlock` (Figure 3.2b) production is where SQL^{++} slightly deviates from SQL: In SQL^{++}, we can either place the **SELECT** clause at the start of the query (conforming to standard SQL) *or* at the end of the query to more closely reflect how queries are processed. We choose the latter style for the SQL^{++} queries (and as we'll see later, the gSQL^{++} queries) in this paper. Lastly, we have the `StreamGenerator` production (Figure 3.2c) which captures the **FROM**, **LET** (a SQL^{++} clause that binds an expression to a variable), **WHERE**, **GROUP BY**, and **HAVING** clauses. Conceptually, the `StreamGenerator` production generates a tuple streams of bound variables.

In SQL^{++}, **FROM** clause variables are allowed to be bound to *any* JSON element. In contrast, SQL only binds **FROM** clause variables to regularized and structured tuples. Subqueries in SQL^{++} are first-class citizens, allowing for greater composability than SQL subqueries (which are restricted to returning scalar or **NULL** values). To demonstrate these features, suppose we want to issue a query on our social network to find users with non-**NULL** last names

```
FROM
    Users u
WHERE
    u.name.last IS NOT NULL
SELECT
    u.id AS uid,
    ( FROM
          Messages m
      WHERE
          m.user_id = u.id
      SELECT VALUE
          m.id ) AS mids;
```

Listing 3.3: SQL^{++} query that correlates two datasets in the **SELECT** clause.

```
{ "uid": 2,  "mids": [10001,10003,10010,10011] }
{ "uid": 9,  "mids": [10002,10089] }
{ "uid": 16, "mids": [] }
```

Listing 3.4: Result set for the query in Listing 3.3.

and all messages they have written. A legal way to express this query in SQL^{++} is given in Listing 3.3. We illustrate two more features of SQL^{++} that are not present in SQL:

1. In SQL^{++}, subqueries can be used to build arrays of documents. In Listing 3.3, we use a subquery to create records containing arrays of message IDs.
2. In SQL^{++}, the **SELECT** clause is used to bind variables of tuples from the `StreamGenerator` production to *documents*. The **SELECT VALUE** variant is used to return arrays of the expression `m.id` (i.e., arrays of integers), instead of arrays of documents containing `m.id`.

Assume that executing our Listing 3.3 query yields the three results in Listing 3.4. We see that the user `id` = 16 has not written any messages, therefore the `mids` array in their result record is empty.

```
FROM
    Users u
GROUP BY
    SUBSTR(u.join_date, 0, 4)
    GROUP AS g
HAVING
    COUNT(*) < 12
SELECT
    SUBSTR(u.join_date, 0, 4)      AS join_year,
    ( FROM g SELECT VALUE g.u.id ) AS uids;
```

Listing 3.5: SQL^{++} `GROUP AS` query to return groups formed by a `GROUP BY` clause.

```
{ "join_year": "2017", "uids": [2] }
{ "join_year": "2021", "uids": [64,65,66,67,68,69,70,71,72,73,74] }
```

Listing 3.6: Result set for the query in Listing 3.5.

Another noteworthy aspect of SQL^{++} is its `GROUP AS` clause, allowing users to query over groups that they create through the SQL `GROUP BY` clause. Contrast this with SQL, whose `GROUP BY` clause only allows reasoning over aggregate values of groups. Suppose we want to group all `Users` by their join year and return the *groups* of user IDs for groups that have less than 12 elements. We can use the SQL^{++} query in Listing 3.5 to realize this grouping, with an example result set given in Listing 3.6. Both results in Listing 3.6 have `uids` arrays that adhere to the `HAVING` clause, where the length of both arrays are less than 12. Given that SQL^{++} is the query language used by AsterixDB, SQL^{++} also serves as the foundation for the Graphix query language extension: gSQL^{++}.

Chapter 4

Graph Model

Having described the social network schema in Section 3.2, we will now use Graphix to define a graph view that users can issue queries on. The task of authoring these graph views will typically be left to the *database designers*, or, users who are intimate with the existing database design (though this is not to say that other types of users cannot create their own views). The purpose of this mapping step is to provide logical data independence for defined graph views, isolating the other "actors on the scene" (i.e., administrators, data analysts, application programmers, etc. . .) from the underlying AsterixDB data model.

In this chapter, we will first discuss the graph modeling constructs available for users to define a graph schema with. We will then walk through how Graphix users can build a mapping from valid SQL^{++} expressions to a set of definitions for vertices and edges in a graph.

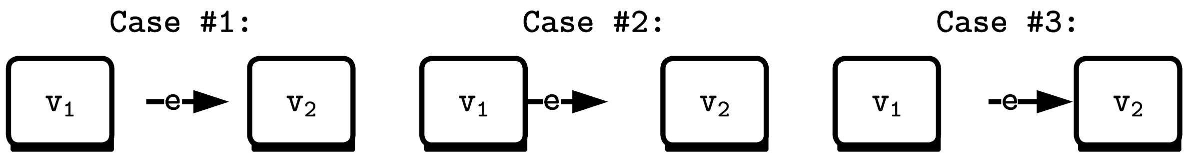


Figure 4.1: Example of "disconnected" edges that *may* exist in a graph.

4.1 Property Graph Model

The graph model that Graphix targets is the *property graph model.* At the core of the model are vertices and edges that connect vertices, but the property graph model adds a few more constructs that have made the model flexible enough for expressing schemata in a variety of domains. Specifically, a property graph a) is directed, b) is vertex and edge labeled, c) permits parallel edges (i.e., more than one edge can connect the same two vertices), and d) associates a set of key-value pairs (known as *properties*) with each vertex and edge.

We define a property in Graphix as the tuple $G = (V,\ E,\ I,\ \lambda)$ where: 1) V is a finite set of vertices, 2) E is a finite set of edges, 3) λ is a labeling function for vertices and edges (formally, $\lambda : (V \cup E) \rightarrow \mathcal{B}$ where $\mathcal{B}$ is the universe of all labels*), and 4) I is a finite set of incidence triples $I \subset (V \times E \times V)$. A single vertex $v \in V$ is defined as a set of key-value pairs. A single edge $e \in E$ is defined in the exact same way: as set of key-value pairs. For some vertex or edge $x \in (V \cup E)$, $\lambda(x)$ can be thought of as the "type" or "class" of the input vertex or edge. Finally, we move to our incidence triple set: $(v_1,\ e,\ v_2) \in I$ denotes that edge $e \in E$ connects source vertex $v_1 \in V$ and destination vertex $v_2 \in V$. $(v_1,\ e,\ v_2) \notin I$ denotes one of three cases (visually given in Figure 4.1):

1. v_1 is not connected to e and e is not connected to v_2;
2. v_1 is connected to e, but e is not connected to v_2; and
3. v_1 is not connected to e, but e is connected to v_2.

*In other graph databases like Neo4j, vertices and edges may possess a non-negative number of labels (making the range of λ a power set of $\mathcal{B}$). In Graphix, we instead require that each vertex and edge must have a single label.

We also note that this definition for I technically defines G as a property *hypergraph*, where a single edge may connect more than two vertices. A directed *hyperedge* $e \in E$ (i.e., an edge in a hypergraph) that connects one vertex $v_1 \in V$ to two other vertices $v_2 \in V$, $v_3 \in V$ is implied by the existence of the two incidence triples $(v_1,\ e,\ v_2)$ and $(v_1,\ e,\ v_3)$. We contrast our property graph definition with formalisms found in other literature [5, 8] that specify a more traditional incidence function (i.e., one that maps edges E to pairs of vertices $(V \times V)$). One of the design philosophies underlying Graphix (and AsterixDB) is to "accept the data as is". A few ill-formed edges or vertices should not impede the processing of graph data, hence we assume a hypergraph structure for our graph data in the remainder of this thesis.

We contrast the property graph model against the older RDF (resource description framework) model, the latter of which was designed to provide standards for processing "data about data". At the heart of the RDF model lies a collection of $(s,\ p,\ o)$ triples, where s refers to a subject, p refers to a predicate (i.e., an edge label), and o refers to an object. Given X_I, a set of Information Resource Identifiers (IRIs), X_B, a set of "blank" nodes (used to help declare the existence of a predicate), and X_L, a set of literals, an $(s,\ p,\ o)$ triple is described below:

$$(s,\ p,\ o) \in (X_I \cup X_B) \times X_I \times (X_I \cup X_B \cup X_L) \tag{4.1}$$

We can loosely visualize a collection of $(s,\ p,\ o)$ triples as a graph where each vertex belongs to the set $(X_I \cup X_B \cup X_L)$ and each edge belongs to the set X_I. Note that our vertex and edge sets are not disjoint: edges can be used to described other edges. This general notion of what "resources" are in IRIs is what makes such a feature possible, allowing users to describe a wide variety of graph structures. The RDF* model further generalizes what resources are by allowing $(s,\ p,\ o)$ triples themselves to be the subject of another triple (i.e., enabling triples about triples).

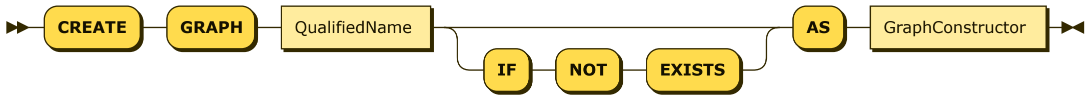
In spite of the RDF model's age, the property graph model has seen significantly more adoption by graph database vendors [67]. In an effort to store existing RDF data into property graph databases, the problem of defining efficient transformations between the property graph model and the RDF model (as well as the richer RDF* model) has been studied in [9, 1]. The decision to use property graphs over RDF triples in Graphix was influenced not only by existing industry solutions, but also with the similar modeling concepts found in both documents and property graphs. As we demonstrate in this chapter and the next (Chapter 5), we can leverage this similarity for a synergistic user model.

4.2 `CREATE GRAPH` Statement

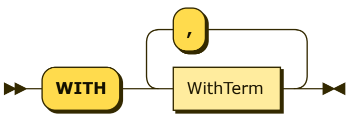
To start, let us consider a *document* in AsterixDB. For our discussion, a document can either a record from an AsterixDB dataset *or* an object-valued result of an AsterixDB query. We formally define a document as "a set of key-value pairs". A document in AsterixDB shares the exact same definition for a vertex and edge in the previous section. Thus, we draw attention to central point in this chapter and the next:

> A Graphix vertex is an AsterixDB document (either materialized or non-materialized).
>
> A Graphix edge is an AsterixDB document (either materialized or non-materialized).

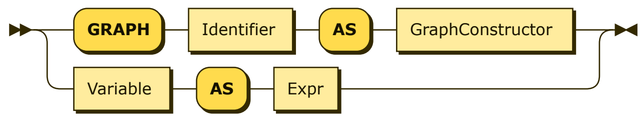
An instance of a vertex is assigned a single label and contains two sets of fields: (a) a set of fields that are denoted (but not enforced) as its *primary key*, and (b) an optional set of fields that correspond to the properties of the vertex. An instance of an edge in Graphix is assigned a single label and is always directed, which allows us to define an edge in three distinct sets of fields: (i) a set of fields that form a foreign key reference to a source vertex, known as the *edge source key*, (ii) a set of fields that form a foreign key reference to a destination vertex, known as the *edge destination key*, and (iii) an optional set of fields that correspond to the properties



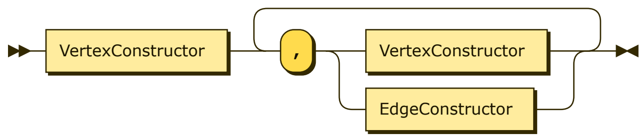
(a) Grammar for the `CREATE GRAPH` statement.



(b) Grammar for the `WITH` clause.



(c) Grammar for the `WithTerm` production.

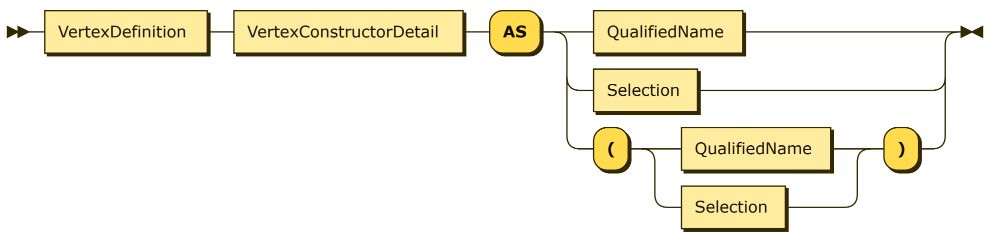


(d) Grammar for the `GraphConstructor` production.

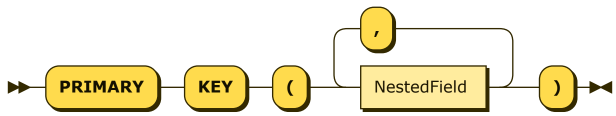
Figure 4.2: Starting productions (as railroad diagrams) for defining a graph in Graphix.

of an edge. To realize the incidence triple set I, both the edge source key and edge destination key are *later* (graphs are non-materialized in Graphix) used in queries to construct a two-way `JOIN` between three document collections: 1) a source vertex document collection $D_{\texttt{SOURCE}}$, 2) an edge document collection $D_{\texttt{EDGE}}$, and 3) a destination vertex document collection $D_{\texttt{DEST}}$ such that: $D_{\texttt{SOURCE}} \bowtie_1 D_{\texttt{EDGE}} \bowtie_2 D_{\texttt{DEST}}$. The $\bowtie_1$ represents an `INNER JOIN` using the primary key of $D_{\texttt{SOURCE}}$ and the source key of $D_{\texttt{EDGE}}$, and the $\bowtie_2$ represents an `INNER JOIN` using the primary key of $D_{\texttt{DEST}}$ and the destination key of $D_{\texttt{EDGE}}$. We describe more on how queries on Graphix graphs are realized in Chapter 6.

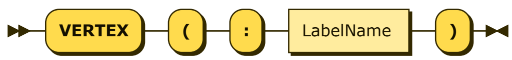
Figure 4.2 illustrates the starting productions for defining a graph in Graphix. A user can create a managed graph with the `CREATE GRAPH` statement (Figure 4.2a), allowing the graph to be stored in Graphix and used in future requests. Managed graphs will also prevent the deletion of datasets, views, and functions that are used in the graph itself. Graphix users can also define temporary graphs in the context of a single query with the `WITH` clause, which is particularly useful when initially building and debugging graph mappings. In both the `CREATE GRAPH` production and `WITH` clause, a graph $G = (V,\ E,\ I,\ \lambda)$ is specified using



(a) Graphix grammar for the `VertexConstructor` production.



(b) Graphix grammar for the `VertexConstructorDetail` production.

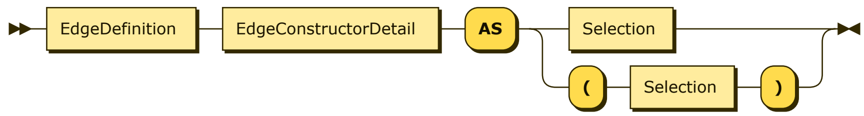


(c) Graphix grammar for the `VertexDefinition` production.

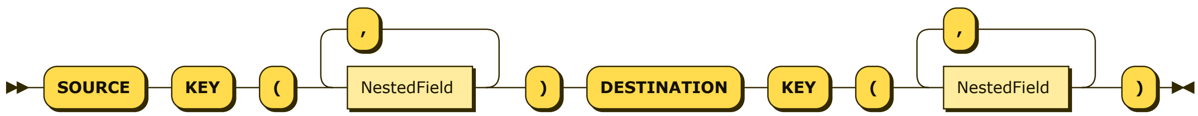
Figure 4.3: Grammar used to define a vertex (`VertexConstructor`) in Graphix.

the `GraphConstructor` production in Figure 4.2d. Modeling each part of G as piecewise functions, each `VertexConstructor` and `EdgeConstructor` production specifies some portion of our graph.

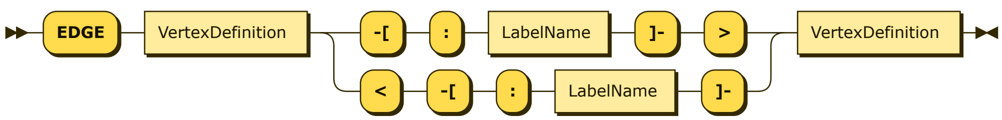
A single `VertexConstructor` determines two items: 1) $V^b \subset V$, a subset of our graph vertices that belong to the entire graph, and 2) $\lambda^{\texttt{VC}(b)}$, a labeling sub-function that belongs to the greater labeling function λ. The `VertexConstructor` production given in Figure 4.3 starts by defining a label `LabelName` b that will group / classify this working set of vertices $V^b \subset V$. We define $\lambda^{\texttt{VC}(b)} : V^b \rightarrow b$ as a constant function that maps all vertices $v \in V^b$ to our label b. As we'll see later in Chapter 5, the ASCII-art syntax of Figure 4.3c mirrors how vertex patterns are specified in queries. After defining our label, users must then specify i) the primary key associated with all vertices in V^b, and ii) a query (or dataset) that returns a collection of documents where each document *is* a vertex $v \in V^b$. We expect that every vertex has the declared primary key, though Graphix does not enforce this key constraint.



(a) Graphix grammar for the `EdgeConstructor` production.



(b) Graphix grammar for the `EdgeConstructorDetail` production.



(c) Graphix grammar for the `EdgeDefinition` production.

Figure 4.4: Grammar used to define an edge (`EdgeConstructor`) in Graphix.

At this point of our discussion, we would to draw attention to how other view-based graph mapping systems define vertex properties. In systems like Oracle Graph [54], Cytosm [63], and DuckPGQ [66] properties are explicitly enumerated. In Graphix, the query (or dataset) used to define our subset of vertices V^b *implicitly* defines the vertex properties of $v \in V^b$. A Graphix graph schema does not pre-declare the properties of its vertices (and edges), mirroring the same schema flexibility offered by AsterixDB.

A single `EdgeConstructor` determines three items: 1) $E^b \subset E$, a subset of our graph edges that belong to the entire graph, 2) $\lambda^{\texttt{EC}(b)}$, a labeling sub-function that belongs to the greater labeling function λ, and 3) $I^{\texttt{EC}(b)} \subset I$, a subset of incidence triples. The `EdgeConstructor` production in Figure 4.4 starts by defining the incidence triple set $I^{\texttt{EC}(b)}$ using the same ASCII-art syntax used to express edge patterns in queries. More specifically, we use the label b_{left} from the leading `VertexDefinition` production and the label b_{right} from the trailing `VertexDefinition` production. $I^{\texttt{EC}(b)}$ is defined as the two-way `JOIN` between the collection of left vertices V_{left}, the collection of edges E^b, and the collection of right vertices V_{right}, where

V_{left} and V_{right} are given below:

$$V_{\text{left}} = \{v \mid \lambda(v) = b_{\text{left}} \wedge v \in V\} \tag{4.2}$$

$$V_{\text{right}} = \{v \mid \lambda(v) = b_{\text{right}} \wedge v \in V\} \tag{4.3}$$

Similar to the `VertexConstructor` production, the `LabelName` in the `EdgeDefinition` production defines the label b that will group / classify this working set of edges $E^b \subseteq E$. We define $\lambda^{\texttt{EC}(b)} : E^b \rightarrow b$ as a constant function that maps all edges $e \in E^b$ to our label b. To describe *how* the aforementioned `JOIN` should be realized, users then specify three more items: i) the source key associated with all edges in E^b, ii) the destination key associated with all edges in E^b, and iii) a query that returns a collection of documents where each document is an edge $e \in E^b$. Again, we expect that every edge contains its declared source and destination keys, though Graphix (and AsterixDB) do not enforce any foreign key constraints. In the next section, we detail examples of both the `VertexConstructor` and `EdgeConstructor` productions.

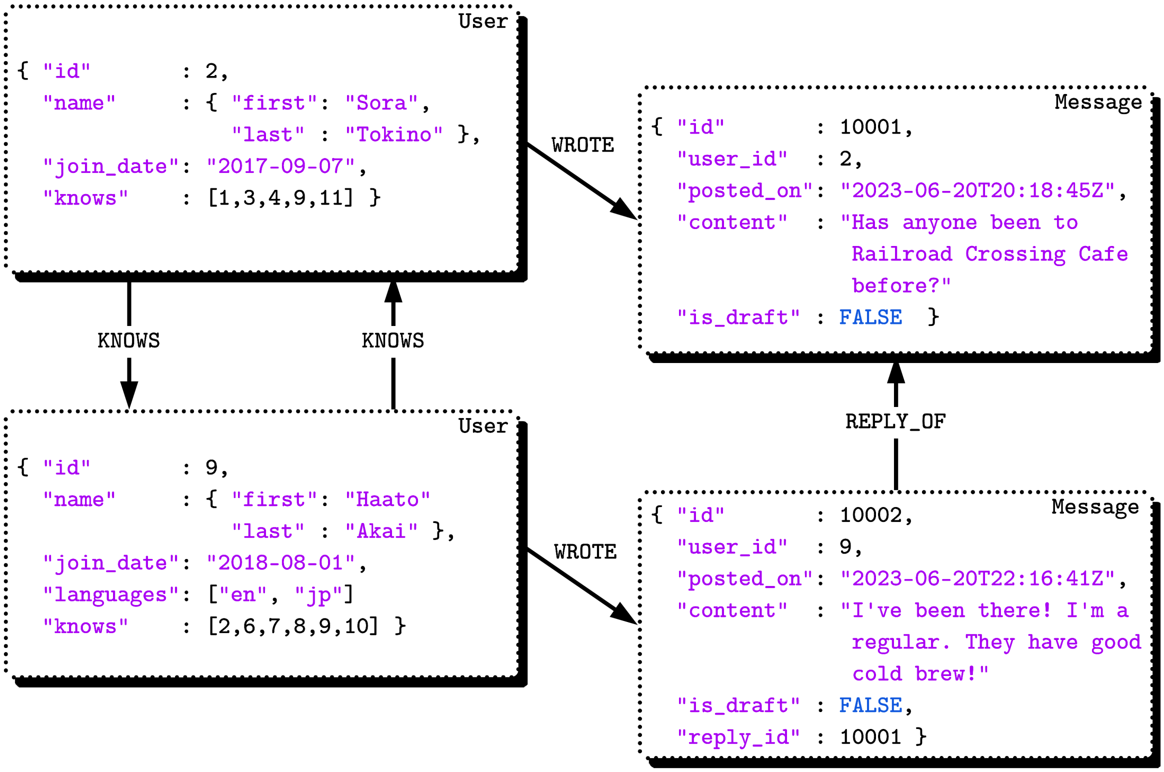
4.2.1 Social Network Example

Listing 4.1 describes the mapping of the `Users` and `Messages` datasets from Section 3.2 (an example of which is given in the replicated Figure 3.1 below for ease of reference) to the property graph `SocialNetworkGraph`, which is composed of two types of vertices and three types of edges:

1. Starting on Line 2, we define the collection of all vertices labeled `User` to be the dataset `Users`. The primary key of the `User` vertex collection is the primary key of the `Users` dataset: `id`. The properties of an individual `User` vertex are all the fields of the mapped `Users` document.

```
CREATE GRAPH SocialNetworkGraph AS
    VERTEX (:User)
        PRIMARY KEY (id)
        AS Users,
    VERTEX (:Message)
        PRIMARY KEY (id)
        AS ( FROM
                Messages m
             WHERE
                NOT m.is_draft
             SELECT
                m.* ),
    EDGE (:User)-[:KNOWS]->(:User)
        SOURCE KEY      (source_id)
        DESTINATION KEY (dest_id)
        AS ( FROM
                Users u,
                u.knows k
             SELECT
                u.id AS source_id,
                k    AS dest_id ),
    EDGE (:User)-[:WROTE]->(:Message)
        SOURCE KEY      (user_id)
        DESTINATION KEY (message_id)
        AS ( FROM
                Messages m
             SELECT
               m.user_id   AS user_id,
               m.id        AS message_id,
               m.posted_on AS posted_on ),
    EDGE (:Message)-[:REPLY_OF]->(:Message)
        SOURCE KEY      (source_id)
        DESTINATION KEY (dest_id)
        AS ( FROM
                Messages m
             SELECT
                m.id        AS source_id,
                m.reply_id  AS dest_id,
                m.posted_on AS posted_on );
```

Listing 4.1: CREATE GRAPH DDL to create a property graph view.



Duplicate of Figure 3.1. Example documents of two `Users`, two `Messages`, and their relationships in the `SocialNetworkGraph`.

2. Starting on Line 5, we define the collection of all vertices labeled `Message` to be the result of the query specified after `AS`: all `Message` documents that are not drafts. Again, the primary key and properties are taken directly from the underlying dataset: `Message`. This vertex mapping demonstrates a unique feature of Graphix when compared to other view-based graph systems: the ability to define *any* query as a vertex (or edge), not just existing stored datasets. To realize more complex vertex mappings, SQL^{++} clauses like `UNION ALL`, `JOIN`, and `GROUP BY` could be used to construct the appropriate query.
3. Starting on Line 13, we define the collection of all `KNOWS` edges to be a query that uses the `Users` dataset to return two fields: `source_id` and `dest_id`. `source_id` is defined to be the edge's source key, and `dest_id` is defined to be its destination key. No additional

properties (outside of the key fields) are defined for `KNOWS` edges. This edge mapping demonstrates a natural approach to handle relationships that are captured by arrays: we utilize the existing query language (SQL^{++}) that is purposed to handle nested data to return a normalized collection of (source key, destination key) pairs.

4. Starting on Line 22, we define the collection of all `WROTE` edges to be a query that uses the `Messages` dataset to return three fields: `user_id`, `message_id`, and `posted_on`. The source key is defined to be `user_id`, the destination key is defined to be `message_id`, and `posted_on` is defined to be an additional property of the `WROTE` edge.
5. Starting on Line 31, we define the collection of all `REPLY_OF` edges to be a query that uses the `Messages` dataset to return three fields: `source_id`, `dest_id`, and `posted_on`. The source and destination keys are defined respectively as `source_id`, `dest_id`, and `posted_on` is again defined as an additional property.

4.2.2 Multiple Dataset Example

Subsection 4.2.1 covers the most expected graph mapping examples, but the flexibility of `CREATE GRAPH` enables graphs to be defined over a wide variety of underlying, connected data. To start, we will cover multi-dataset mappings.

The `CREATE GRAPH` statement from Subsection 4.2.1 was built using two datasets that are managed internally by AsterixDB. We will now consider the case where more than one dataset maps to a single labeled vertex or edge collection. Suppose that our social network graph must include users from another organization (dubbed `OrgB` here). New message documents will populate the existing `Messages` dataset referencing these `OrgB` users. For this example, these new users exist in another AsterixDB dataverse (the `OrgB` dataverse) separate from our original datasets. Our graph mapping will now consider two new datasets: (1) `OrgB.Users`

```
CREATE DATAVERSE OrgB;

CREATE TYPE OrgB.UsersType AS {
    userId    : bigint,
    firstName : string,
    lastName  : string
};
CREATE DATASET OrgB.Users(OrgB.UsersType) PRIMARY KEY userId;

CREATE TYPE OrgB.KnowsType AS {
    startId      : bigint,
    endId        : bigint,
    creationDate : string
};
CREATE DATASET OrgB.Knows(OrgB.KnowsType)
    PRIMARY KEY startId, endId;
```

Listing 4.2: Set of DDLs to define two additional datasets: `OrgB.Users` and `OrgB.Knows`.

```
VERTEX (:User)
    PRIMARY KEY (id)
    AS ( FROM
              Users u
         SELECT
              u.id   AS id,
              u.name AS name,
              u.*
         UNION ALL
         FROM
              OrgB.Users bu
         LET
              name = { "first": bu.firstName,
                       "last" : bu.lastName }
         SELECT
              bu.userId AS id,
              name      AS name )
```

Listing 4.3: Alternative `(:User)` vertex definition which includes the newly defined dataset `OrgB.Users`†.

```
EDGE (:User)-[:KNOWS]->(:User)
    SOURCE KEY       (source_id)
    DESTINATION KEY (dest_id)
    AS ( FROM
             Users u,
             u.knows k
         SELECT
             u.id AS source_id,
             k    AS dest_id
         UNION ALL
         FROM
             OrgB.Knows bk
         SELECT
             bk.startId      AS source_id,
             bk.endId        AS dest_id,
             bk.creationDate AS creation_date )
```

Listing 4.4: Alternative `KNOWS` edge definition which includes the newly defined dataset `OrgB.Knows`.

and (2) an M:N relationship dataset between different users in `OrgB.Users`: `OrgB.Knows`. Listing 4.2 describes a set of AsterixDB DDLs to define `OrgB.Users` and `OrgB.Knows`.

We will start by redefining our `(:User)` vertex, which must now include the union of the `Users` and `OrgB.Users` dataset. We can easily accomplish this with a `UNION ALL` query in our vertex definition. The `(:User)` vertex definition is given in Listing 4.3. In contrast to SQL, SQL++ relaxes the restriction that both queries involved in the `UNION ALL` must be union-compatible (greatly simplifying the `Users` ↔ `OrgB.Users` alignment step). To align the `id` field from `Users`, we rename the `userId` field from the `OrgB.Users` dataset to `id`. To align the composite `name` field from `Users`, we bind the variable `name` to an object composed of the `firstName` and `lastName` fields from the `OrgB.Users` dataset. All remaining properties inherited from the `Users` dataset (e.g., the `birthdate` and the `languages` fields) are captured using in the "`u.*`" term of the first `SELECT` clause.

[†]The first `SELECT` clause could be replaced with "`SELECT VALUE u`", but is expressed as such for clarity.

To define our KNOWS edge to include the relationships captured in the OrgB.Knows dataset, we will express another UNION ALL query in Listing 4.4. We leverage SQL^{++} to define a mapping that handles M:N relationships from both nested data (the Users dataset and the knows array) and normalized data (the new OrgB.Knows dataset). The union-compatible relaxation from SQL^{++} is leveraged again to add the creationDate field of the OrgB.Knows dataset as a property to the KNOWS edge (a property that is not found in edges mapped from the knows array of a Users document).

4.2.3 Derived Property Example

The CREATE GRAPH statement from Subsection 4.2.1 was built using fields directly defined with the documents of the underlying datasets (Users and Messages). We will now illustrate how *computed* properties can be defined in Graphix. Suppose we want to attach a weight property ω to all KNOWS edges such that edges between users who have written many messages weigh more than edges between users who are less active posters. As we'll see in Section 5.4, we can later leverage attributes like ω to express queries like cheapest path. To compute ω, we'll use the SQL^{++} query in Listing 4.5.

The logical data independence provided to Graphix users for their graphs means that we have several options at their disposal to realize adding ω as a new property. We will look towards the least invasive option, which involves redefining the KNOWS edge definition from our CREATE GRAPH statement (although other options, e.g., creating a new dataset to hold the result of Listing 4.5, might be preferred if computing ω on-the-fly is too expensive). The new KNOWS edge definition is given in Listing 4.6. We highlight that no updates were made to the underlying data by changing the CREATE GRAPH definition.

We remark that Graphix users can also use graph queries in the bodies of vertex and edge definitions. For an example of such a pattern, see Subsection 5.4.5. If we further generalize

```
FROM
    Users u1,
    u1.knows k,
    Messages m1,
    Messages m2
WHERE
    u1.id = m1.user_id AND
    k = m2.user_id
GROUP BY
    u1.id AS source_id,
    k     AS dest_id
LET
    omega = COUNT(DISTINCT m1.id) +
            COUNT(DISTINCT m2.id)
SELECT
    source_id AS source_id,
    dest_id   AS dest_id,
    omega     AS omega;
```

Listing 4.5: SQL^{++} query to compute the weight attribute ω between two users that know each other.

```
EDGE (:User)-[:KNOWS]->(:User)
    SOURCE KEY      (source_id)
    DESTINATION KEY (dest_id)
    AS ( FROM
             Users u1,
             u1.knows k,
             Messages m1,
             Messages m2
         WHERE
             u1.id = m1.user_id AND
             k = m2.user_id
         GROUP BY
             u1.id AS source_id,
             k     AS dest_id
         LET
            omega = COUNT(DISTINCT m1.id) +
                    COUNT(DISTINCT m2.id)
         SELECT
             source_id AS source_id,
             dest_id   AS dest_id,
             omega     AS omega )
```

Listing 4.6: Alternative `KNOWS` edge definition which includes the ω weight attribute from Listing 4.5.

the concept of expressing computed attributes, we remark that Graphix's graph model can also be used to support hypernode graph structures [40] (where the vertices and edges of one graph could be used to define the vertex sets of another graph). The Graphix graph model is highly flexible, leveraging the advantages of 1) the underlying document model's self-describing nature to handle complex graph structures while also being amenable to schema evolution, and 2) the underlying query language SQL^{++}, which is able to handle schema-heterogeneity across all documents used to define vertices and edges.

Chapter 5

Query Model

A *query model* describes the constructs available for use when expressing queries. In this chapter, we will first motivate our decision to extend SQL^{++} to specify graph queries (as opposed to using an existing standard or graph language). We will then introduce two essential constructs for building graph queries: 1) pattern matching, and 2) navigation. By integrating the two aforementioned concepts with AsterixDB's current query model, users can express a rich set of graph queries that leverage modern SQL constructs (e.g., window functions, grouping sets) and SQL^{++} constructs (e.g., the `GROUP AS` clause) together with navigational pattern matching.

5.1 SQL^{++} Query Extension

When designing the query language for Graphix, special care and attention was given towards deciding *how* users should be able to specify graph queries. On one end of the solution spectrum, we could have simply used an existing graph query language. On the other end of the solution spectrum, we could have used the existing recursive features of the SQL standard to extend SQL^{++} for use in Graphix. Our desiderata for issuing graph queries on existing AsterixDB data searches for a solution somewhere in the middle: a) brevity (balancing "Turing-complete" with ease-of-use), b) maintenance (avoiding the accidental complexity users would incur by working with two different query languages), and c) synergy (being able to intuitively integrate existing SQL / SQL^{++} language features with graph query constructs).

5.1.1 SQL-1999 Recursive Queries

Recursion in SQL was introduced into the 1999 standard, and, while Turing complete, has resulted in less-than-user-friendly queries to solve basic problems like reachability. Recursion in SQL is expressed using a CTE (common table expression) that contains a reference to the CTE itself. The query body of a recursive CTE is composed of two parts: an *anchor* member and a *recursive* member. The anchor member of a recursive CTE is logically executed once, while the recursive member is executed until a least fixed-point is reached (i.e., there are no other tuples left to process). To guarantee the existence (and uniqueness) of this least fixed-point, the recursive member of a recursive CTE must be *monotonic*. Recursive-aggregate-SQL (RaSQL) [28] and recent research from Hirn and Grust [31] propose modifications that slightly relax this monotonicity constraint for more practical semantics, however, such work still goes against our desired "brevity" for common graph queries. As we'll see in the next section, if we sacrifice Turing-completeness and target graph constructs (i.e., vertices, edges, and paths), then we can express much more user-friendly queries.

We will now walk through an example. Suppose we want to see if three users are transitively connected to each other. This query can be expressed in recursive SQL as follows: Beginning on Line 2 in Listing 5.1, we start by anchoring the navigation at `$id1` and 1) grabbing the IDs for the next user to visit (`luk`), 2) initializing an array for cycle detection (`vu`) and 3) and

```sql
WITH RECURSIVE Visited AS
    ( SELECT
            u1.knows      AS luk,
            ARRAY[u1.id] AS vu,
            ARRAY[1,0,0] AS v
      FROM
            Users u1
      WHERE
            u1.id = $id1
      UNION ALL
      SELECT
            u2.knows        AS luk,
            rv.vu || u2.id AS vu,
            CASE
                WHEN u2.id = $id2
                THEN ARRAY[rv.v[0],1,rv.v[2]]
                WHEN u2.id = $id3
                THEN ARRAY[rv.v[0],rv.v[1],1]
                ELSE rv.v
            END AS v
      FROM
            Visited rv,
            Users u2
      WHERE
            u2.id = ANY(rv.luk) AND
            NOT u2.id = ANY(rv.vu) )
SELECT
    COUNT(*) > 0 AS connected
FROM
    Visited rv
WHERE
    ( SELECT
            SUM(v) = 3
      FROM
            UNNEST (rv.v) v );
```

Listing 5.1: Recursive SQL query (in PostgreSQL dialect) to find if three users are transitively connected to each other.

an output array (`v`). Subsequent iterations will execute the recursive member on Line 11, which will "traverse" to another user `u2` using the user IDs `luk` from the previous iteration. To avoid traversing over cycles, a check is specified to determine if the ID of the current user is in the visited array `vu`. If the current user has one of the IDs we are interested in, the output array is updated by performing a bitwise `OR` operation with the current output array. The results that the recursive member yields to the next iteration includes the next set of user ids, an updated visited array to include `u2`, and the status of the output array. If there any results from the recursive CTE such that the output array has a length of 3, then we know that all three users of interest have been visited at some point. Otherwise, we conclude that there exists no path that connects `$id1`, `$id2`, and `$id3`.

To get around the short-term memory restriction inherent to recursive CTEs, Listing 5.1 accumulates state from previous iterations in the `vu` and `v` arrays. Ultimately, we are only interested in the existence of a single row (one where `v` contains all "1" values). The outer `WHERE` clause and outer `COUNT(*) > 0` aggregate predicate in the `SELECT` clause tells us that we can stop as soon as find such a row, but recognizing such a pattern is non-trivial. A query optimizer would have to, at a minimum, 1) recognize that `v` is a bit vector, 2) recognize that `SUM(v) = 3` is concerned with a specific bit vector, and 3) recognize that the recursive member is performing a bitwise `OR`. Recursive SQL, while very powerful and Turing complete, requires SQL users to define hard-to-optimize constructs for graph queries (e.g., cycle prevention, edge traversal) themselves.

5.1.2 Cypher Query Language

Cypher is arguably the current leader for querying property graphs [25], though there is a growing effort to standardize [33, 24] and bridge the gap between other similar query languages [73, 7] to build a standard graph query language (known as GQL). A defining

```
MATCH
    (u1:User {id: $id1}),
    (u2:User {id: $id2}),
    (u3:User {id: $id3}),
    (u1)-[:KNOWS*]-(u2),
    (u2)-[:KNOWS*]-(u3),
    (u3)-[:KNOWS*]-(u1)
RETURN
    COUNT(*) > 0 AS connected;
```

Listing 5.2: Cypher query to find if three users are transitively connected to each other.

characteristic of all these languages are their `MATCH` clause, allowing users to specify navigational graph patterns via a user-friendly ASCII-art syntax. Recursion in a `MATCH` clause is enabled through the use of edge-labeled regular expressions between vertices in graph patterns. While not as computationally powerful as the Pregel model — or the recursive SQL-99 standard [32] — graph computations such as reachability and shortest path can be written in a much more succinct and natural manner in Cypher.

We contrast the query in the previous section (Listing 5.1) with the much easier-to-read equivalent Cypher query in Listing 5.2: We highlight two main differences between these queries:

1. In the recursive SQL query, a user has to explicitly handle (and prevent) cycles. In Cypher, cycles are implicitly pruned by forbidding traversal over duplicate edges.
2. In the recursive SQL query, a user has to specify *how* the navigation is performed. Listing 5.1 starts the navigation at `$id1`. In the Cypher query, a user does not specify a starting point, allowing the query optimizer to (more easily) choose a more appropriate starting point (say, `$id2` or `$id3`) if `$id1` is a super-node with a lot of matching records in a user's `knows` array.

The `MATCH` clause from Cypher clearly appeals to both users and query engine developers for the common task of reachability, but, as discussed in our desiderata, adopting Cypher as a second language for Graphix would require users to write (and maintain) queries in two different query languages. Furthermore, SQL is *the* de facto standard query language. Extending SQL^{++} (which extends SQL) allows the query language of Graphix to build on the decades of work that has gone into SQL. As an example, consider the SQL 2003 standard, which includes a collection of rich OLAP operations (window functions, window clauses, grouping sets, etc…). We believe that navigational graph pattern matching can and should compliment existing (and future) operations like these.

5.1.3 SQL-2023 Property Graph Queries

The ISO/IEC JTC1 SC32 working group for database languages (WG3, the same group that maintains and enhances SQL) have developed a graph pattern matching sub-language (GPML) for use in not only GQL, but also in the recently released SQL/PGQ part of the latest SQL-2023 standard [21]. SQL/PGQ enables GPML queries over relational data via a property graph view, where the results of evaluating a GPML are logically available as a table to further manipulate. Revisiting our reachability example, suppose that we built a property graph named `UserKnowsGraph` using the DDL in Listing 5.3.* Listing 5.4 illustrates a similar Cypher-like query but rooted in SQL. The result of a `GRAPH_TABLE` is a table whose structure is dictated by the trailing `COLUMNS` term.

`GRAPH_TABLE` explicitly requires SQL/PGQ users to condense the result of their query into a table before being used by the remainder of the query. Only then can we apply concepts like `JOIN`, `GROUP BY`, `OVER`, etc… Fundamentally, a SQL query revolves around structured tables, and SQL/PGQ draws a clear "line in the sand" between the relational world and the graph world. Users cannot apply constructs like `GROUP BY` and `JOIN` to graph elements without first representing the result of the GPML query as a table. To move beyond these

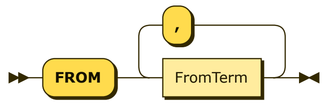
*We assume the existence of a `Knows` table to model the "user knows user" relationship. DuckPGQ requires that each vertex is defined with a single table and that each edge is defined with a single table.

```sql
CREATE PROPERTY GRAPH UserKnowsGraph
    VERTEX TABLES (
        Users
            PROPERTIES (
                id,
                name,
                join_date,
                languages
            )
            LABEL User
    )
    EDGE TABLES (
        Knows
            SOURCE KEY      (source_id) REFERENCES Users (id)
            DESTINATION KEY (dest_id) REFERENCES Users (id)
            PROPERTIES      (source_id, dest_id)
            LABEL           KNOWS
            SOURCE          User
            DESTINATION     User
    );
```

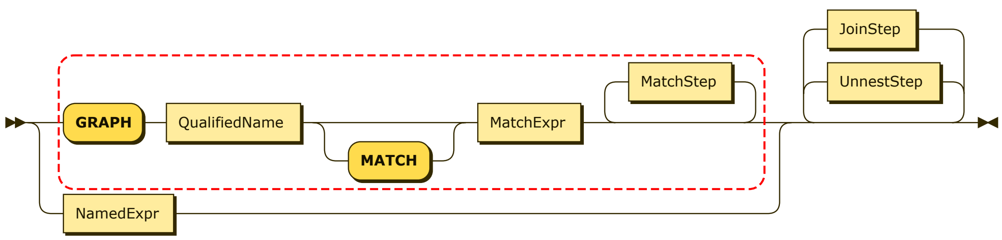
Listing 5.3: SQL/PGQ graph creation DDL in DuckPGQ dialect [66]. This graph is used in the Listing 5.4 query.

```sql
SELECT
    COUNT(*) > 0 AS connected
FROM
    GRAPH_TABLE (
        UserKnowsGraph,
        MATCH
            (u1:User WHERE u1.id = $id1),
            (u2:User WHERE u2.id = $id2),
            (u3:User WHERE u3.id = $id3),
            (u1)-[:KNOWS]-*(u2),
            (u2)-[:KNOWS]-*(u3),
            (u3)-[:KNOWS]-*(u1)
        COLUMNS (
            u1.id,
            u2.id,
            u3.id
        )
    ) AS v
```

Listing 5.4: SQL/PGQ query to find if three users are transitively connected to each other using the graph defined in Listing 5.3.



(a) SQL^{++} grammar for the `FromClause` production.



(b) SQL^{++} grammar for the `FromTerm` production.

Figure 5.1: Grammar extension used to define specify navigational graph pattern matching in a SQL^{++} `FROM` clause.

limitations, we will have to consider a different user model to map from, as we'll see in the following section.

5.1.4 gSQL^{++} `FROM` Clause Extension

We now move to gSQL^{++}, a SQL^{++} extension that enables the integration of graph pattern matching (borrowed from both Cypher and SQL/PGQ) with existing SQL and SQL^{++} constructs. In contrast to SQL/PGQ, gSQL^{++} maps from a *document* model to a graph model by extending SQL^{++}. To start, we recognize that Cypher's `MATCH` clause is more-or-less an analog to the `FROM` clause in SQL: both the `MATCH` clause and `FROM` clause specify iteration variable bindings that will be used in other clauses downstream. In SQL^{++}, the `FROM` clause is composed of one or more `FromTerm` productions. The most fundamental change that gSQL^{++} makes to SQL^{++} is therefore in the `FromTerm`. Our intent with Graphix was to make gSQL^{++} a *strict* superset of SQL^{++}. As seen in Figure 5.1b, all SQL^{++} queries are valid gSQL^{++} queries. Users follow the bottom path to express their standard SQL^{++} `FromTerm`. To express a gSQL^{++} `FromTerm`, users follow the top path (the grammar surrounded by the red dashed lines) and specify:

```
FROM
    GRAPH UserKnowsGraph
        (u1:User WHERE u1.id = $id1),
        (u2:User WHERE u2.id = $id2),
        (u3:User WHERE u3.id = $id3),
        (u1)-[:KNOWS*]-(u2),
        (u2)-[:KNOWS*]-(u3),
        (u3)-[:KNOWS*]-(u1)
SELECT
    COUNT(*) > 0 AS connected;
```

Listing 5.5: gSQL^{++} query to find if three users are transitively connected to each other.

1. the `GRAPH` keyword;
2. the name of the graph (i.e., `QualifiedName`); and
3. the graph query patterns (i.e., `MatchExpr` and zero or more `MatchStep` productions).

For completeness, we give the gSQL^{++} query to find if three users are transitively connected to each other in Listing 5.5.[†] Note that aggregation (the `COUNT(*)`) is logically performed on the graph pattern itself and not a table of the query graph pattern. At a high level, the `MatchExpr` and `MatchStep` productions specify variable bindings to graph constructs (i.e., vertices, edges, and paths). Following our design decisions from our graph model (Chapter 4), we highlight a similar point: *every* bound variable from a `FromTerm` (both gSQL^{++} and SQL^{++}) represents a document in AsterixDB's data model. This simple mapping is what makes gSQL^{++} so powerful. For both paths of Figure 5.1b, a gSQL^{++} user is able to operate on *any* bound variable as if it were a SQL^{++} variable binding. Consequently, *all* clauses and constructs from SQL^{++} are also available to use on vertices, edges, and paths. In contrast to `GRAPH_TABLE` from SQL/PGQ, users are able to directly perform operations from the parent query model on graph constructs for truly synergistic document and graph analytics.

[†]Listing 5.5 shows the use of undirected paths, which is not currently implemented at the time of writing.

5.2 Pattern Matching Queries

Having discussed the vehicle for where to include our graph query constructs, we will now delve into the foundations of modern graph query languages. We will first review the problem of pattern matching, which is central to many modern languages like SPARQL and Cypher and covers many common graph computations (e.g., neighborhood queries, diamond pattern finding, triangle counting, etc... as surveyed in [59]). We will conclude by describing pattern matching in gSQL^{++}, emphasizing the use of existing SQL clauses to cover common extensions to pattern matching queries found in other graph query languages.

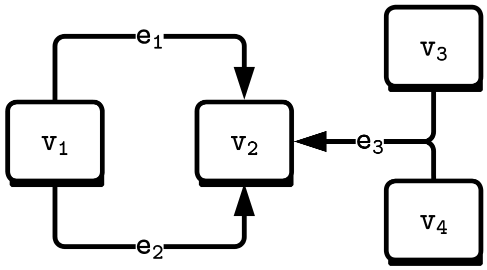
5.2.1 Graph Pattern Matching

To start, we are given a) a graph instance $G_D = (I_D,\ V_D,\ E_D)$ where I_D, V_D, and E_D are incidence triples, vertices, and edges from the graph instance respectively, and b) a query pattern $G_Q = (I_Q,\ V_Q,\ E_Q)$ where I_Q, V_Q, and E_Q are incidence triples, vertices, and edges from the query pattern respectively. The problem of pattern matching involves finding M, a set of *morphisms* (i.e., graph mappings) from the query pattern to a subgraph of the graph instance:

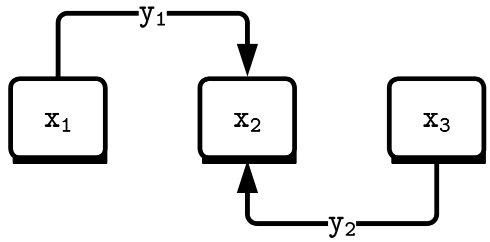
$$M = \{(m_v,\ m_e) \mid m_v : V_Q \to V_D \ \wedge \ m_e : E_Q \to E_D\} \tag{5.1}$$

The exact definition of m_v and m_e vary from system to system. We specify four classes of morphisms used for pattern matching below:

Homomorphism A morphism $m = (m_v,\ m_e)$ such that all incident vertex-edge-vertex query pattern triples imply the same incidence in the graph instance: $(v_i,\ e,\ v_j) \in I_Q \implies (m_v(v_i),\ m_e(e),\ m_v(v_j)) \in I_D$ where $v_i \in V_Q$, $e \in E_Q$, and $v_j \in V_Q$. Informally, a homomorphism defines a mapping from the query pattern to a subgraph of the graph instance that preserves edge adjacency.



(a) An example (hyper)graph instance G_D that possesses four vertices and three edges.



(b) An example graph query pattern G_Q that possesses three vertices and two edges.

Figure 5.2: An example graph instance G_D and query pattern G_Q.

Vertex Isomorphism A constrained homomorphism $m = (m_v,\ m_e)$ such that m_v is injective. Informally, a vertex isomorphism defines a homomorphism where one graph instance vertex $v_D \in V_D$ is mapped to exactly one query pattern vertex $v_Q \in V_Q$.

Edge Isomorphism A constrained homomorphism $m = (m_v,\ m_e)$ such that m_e is injective. Similar to a vertex isomorphism, an edge isomorphism defines a homomorphism where one graph instance edge $e_D \in E_D$ is mapped to exactly one query pattern edge $e_Q \in E_Q$. In a non-hypergraph setting, edge isomorphism implies vertex isomorphism, however, hyperedges (edges in a hypergraph) break this implication.

(Total) Isomorphism A constrained homomorphism $m = (m_v,\ m_e)$ such that both m_v and m_e are injective. Informally, a total isomorphism is the most restrictive, defining a homomorphism where exactly one graph instance vertex $v_D \in V_D$ is mapped to exactly one query pattern vertex $v_Q \in V_Q$ and one graph instance edge $e_D \in E_D$ is mapped to exactly one query pattern edge $e_Q \in E_Q$.

To illustrate the differences between each morphism class, we assume the graph instance G_D in Figure 5.2a and the query pattern G_Q in Figure 5.2b. Table 5.1 describes all possible homomorphisms from G_Q to subgraphs of G_D, where each row describes the subgraph being mapped to. Morphisms m_1 to m_8 are all totally isomorphic, as no graph instance vertex is mapped from more than one query pattern vertex and no graph instance edge is mapped from more than one query pattern edge. Morphisms m_9 and m_{10} describe morphisms that

m_i	x_1	x_2	x_3	y_1	y_2	Homomorphic?	Vertex Isomorphic?	Edge Isomorphic?	Totally Isomorphic?
m_1	v_1	v_2	v_3	e_1	e_3	✓	✓	✓	✓
m_2	v_1	v_2	v_4	e_1	e_3	✓	✓	✓	✓
m_3	v_1	v_2	v_3	e_2	e_3	✓	✓	✓	✓
m_4	v_1	v_2	v_4	e_2	e_3	✓	✓	✓	✓
m_5	v_3	v_2	v_1	e_3	e_1	✓	✓	✓	✓
m_6	v_3	v_2	v_1	e_3	e_2	✓	✓	✓	✓
m_7	v_4	v_2	v_1	e_3	e_1	✓	✓	✓	✓
m_8	v_4	v_2	v_1	e_3	e_2	✓	✓	✓	✓
m_9	v_3	v_2	v_4	e_3	e_3	✓	✗	✓	✗
m_{10}	v_4	v_2	v_3	e_3	e_3	✓	✗	✓	✗
m_{11}	v_1	v_2	v_1	e_1	e_2	✓	✓	✗	✗
m_{12}	v_1	v_2	v_1	e_2	e_1	✓	✓	✗	✗
m_{13}	v_1	v_2	v_1	e_1	e_1	✓	✗	✗	✗
m_{14}	v_1	v_2	v_1	e_2	e_2	✓	✗	✗	✗
m_{15}	v_3	v_2	v_3	e_3	e_3	✓	✗	✗	✗
m_{16}	v_4	v_2	v_4	e_3	e_3	✓	✗	✗	✗

Table 5.1: A table describing different morphisms from the query pattern in Figure 5.2b to subgraphs of the graph instance in Figure 5.2a.

are edge isomorphic but not vertex isomorphic (as shown by e_3 mapped from both y_1 and y_2). Morphisms m_{11} and m_{12} describe morphisms that are vertex isomorphic but not edge isomorphic (as shown by v_1 mapped from bound x_1 and x_3). The remaining morphisms (m_{13} to m_{16}) are only homomorphic, as shown by y_1 and y_2 mapped to the same graph instance edge and x_1 and x_3 mapped to the same graph instance vertex.

Languages like SPARQL, Oracle PGQL, and the GPML of GQL evaluate G_Q using homomorphism semantics, while Cypher evaluates G_Q using edge isomorphism semantics (i.e., total isomorphism in Neo4j's non-hypergraph setting). To express homomorphisms in Cypher, users can get around the more restrictive semantics by dividing G_Q into sub-patterns G_{Q_1}, G_{Q_2}, … G_{Q_N} for the underlying database to solve. By default, Graphix evaluates G_Q using total isomorphism semantics, though these semantics are explicitly tunable with the compiler flag `graphix.semantics.pattern` to generalize pattern matching to any of the other morphism classes.

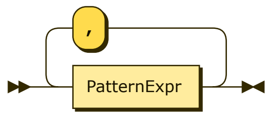
We now move toward *labeled* graphs to finally align our pattern matching problem with our graph model in Chapter 4. We extend the pattern matching problem to qualify the satisfiable morphisms based on a labeling function λ. Specifically, we define a) a graph instance $G_D = (V_D,\ E_D,\ I_D,\ \lambda_D)$ where λ_D assigns graph instance vertices and edges a single label, and b) a query pattern $G_Q = (V_Q,\ E_Q,\ I_Q,\ \lambda_Q)$ where λ_Q assigns query pattern vertices and edges a *set* of labels. The problem of labeled graph pattern matching involves finding a subset of all morphisms $M_\lambda \subset M$ from the query pattern to a subgraph of the graph instance that satisfy the labeling constraints. We describe this extension of the pattern matching problem in Equation 5.2, which many users of graph databases have grown to expect support for in modern graph query languages:

$$M_\lambda = \left\{ (m_v,\ m_e) \text{ s.t. } \begin{array}{ll} \lambda_D(m_v(v_Q)) \in \lambda_Q(v_Q) \ \wedge \ \lambda_D(m_e(e_Q)) \in \lambda_Q(e_Q) & \wedge \\ v_Q \in V_Q \ \wedge \ e_Q \in E_Q & \wedge \\ (m_v,\ m_e) \in M & \end{array} \right\} \tag{5.2}$$

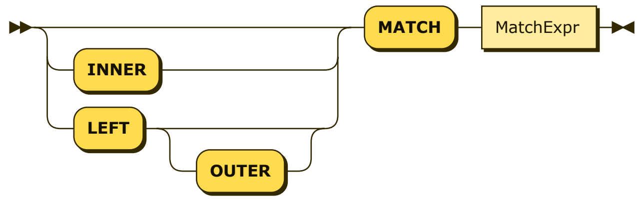
where M is defined by one of the aforementioned morphism classes (i.e., homomorphism, edge isomorphism, vertex isomorphism, and total isomorphism).

5.2.2 gSQL^{++} for Pattern Matching

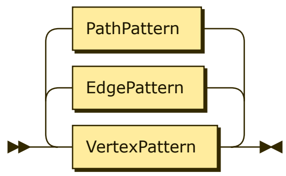
Continuing from the `FromTerm` production in Section 5.1, we will now describe the `MatchExpr` production and the optional `MatchStep` production (with `MatchStep` fully described in Section 5.4). The `MatchExpr` production allows users to describe query patterns G_Q to match against a graph G_D in Graphix. As shown in Figure 5.3, a single `MatchExpr` contains one or more pattern expressions (`PatternExpr`), which describes a series of query vertex patterns



(a) gSQL^{++} grammar for the `MatchExpr` production.



(b) gSQL^{++} grammar for the `MatchStep` production.



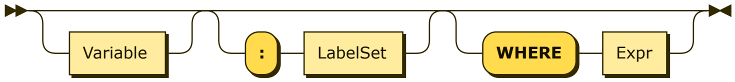
(c) gSQL^{++} grammar for the `PatternExpr` production.

Figure 5.3: Grammar used to describe the `MatchExpr`, `MatchStep`, and `PatternExpr` productions.

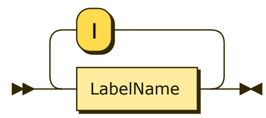
(`VertexPattern`), query edge patterns (`EdgePattern`), and query path patterns (`PathPattern`, detailed further in Section 5.3). In alignment with graph instance vertices and edges being represented as documents in the AsterixDB data model, query pattern vertices and edges (as well as paths, as seen in the next section) are "objects" (similarly defined as sets of key-value pairs) in the SQL^{++} query model. To reference graph elements for use in SQL^{++} constructs, we define an additional function in our formalism, ϑ, that maps query pattern vertices, edges, (and later paths) to "iteration variables". Iteration variables are defined in the same manner as SQL^{++}: references to an item of a result set being iterated over [14]. For pure graph pattern matching gSQL^{++} queries, this result set refers to the morphism set M.

Figure 5.4 describes the grammar for a vertex pattern $v_Q \in V_Q$. A vertex pattern is specified using parentheses, optionally containing a) a variable used to partially define the variable-assigning function ϑ for v_Q, b) a set of labels B^{v_Q} used to partially define the labeling function λ_Q such that $\lambda_Q(v_Q) = B^{v_Q}$, and c) a `WHERE` clause shorthand for further qualifying the "mapped-to" graph vertices (i.e., $m_v(v_Q)$ for some morphism $m \in M$). We note that vertex filtering can also be performed in the `WHERE` clause inline with the containing `FROM`

(a) gSQL^{++} grammar for the `VertexPattern` production.



(b) gSQL^{++} grammar for the `VertexDetail` production.



(c) gSQL^{++} grammar for the `LabelSet` production.

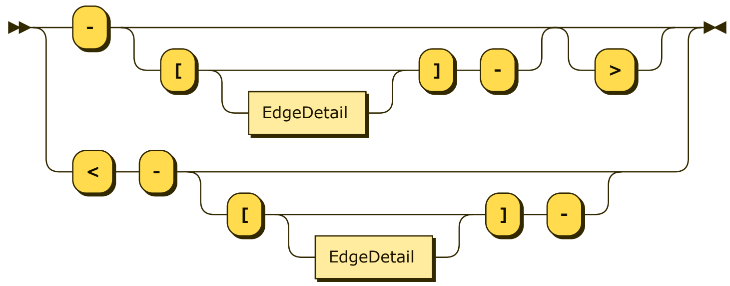
Figure 5.4: Grammar used to describe a query pattern vertex (i.e., the `VertexPattern`, `VertexDetail`, and `LabelSet` productions).

clause. To highlight the simplicity of the gSQL^{++} language extension, we will use the latter style. Finally, the absence of a label set in the context of labeled pattern matching logically denotes a query pattern vertex that can be universally matched (formally, $\lambda_Q(v_Q) = \mathcal{B}$ where $\mathcal{B}$ is the universe of labels).

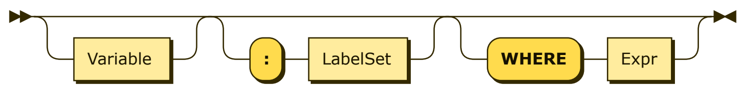
We will now describe the `VertexPattern` production through example. Consider five instances of `VertexPattern` that describe five vertex patterns:

1. `(x1:Message)`
2. `(:Message)`
3. `(x3)`
4. `(x4:Message WHERE x4.id = 10000)`
5. `(x5:User|Message)`

Item 1 defines a vertex pattern v_1 that is assigned the variable $\vartheta(v_1) =$ `x1` and is labeled as $\lambda_Q(v_1) = \{$`Message`$\}$. Item 2 defines an unnamed vertex pattern v_2 with the label $\lambda_Q(v_2) = \{$`Message`$\}$. Item 3 defines a vertex pattern v_3 that is assigned the variable $\vartheta(v_3) =$ `x3` and possesses all labels $\lambda_Q(v_3) = \mathcal{B}$. Item 4 defines a vertex pattern v_4 that is assigned the variable $\vartheta(v_4) =$ `x4`, is labeled as $\lambda_Q(v_4) = \{$`Message`$\}$, and contains a `WHERE` clause shorthand for the conjunct `x4.id = 10000`. Finally, Item 5 defines a vertex pattern v_5 that is assigned the variable $\vartheta(v_5) =$ `x5` and is labeled as $\lambda_Q(v_5) = \{$`User, Message`$\}$.



(a) gSQL^{++} grammar for the `EdgePattern` production.



(b) gSQL^{++} grammar for the `EdgeDetail` production.

Figure 5.5: Grammar used to describe a query pattern edge (i.e., the `EdgePattern` and `EdgeDetail` productions).

Figure 5.5 describes the grammar for an edge pattern $e_Q \in E_Q$. Following the grammar of a `PatternExpr` (see Figure 5.3c), we note that an edge pattern can only be specified between two vertex patterns. We refer to the left vertex pattern of an edge pattern e_Q as $\texttt{LEFT}(e_Q)$ and the right vertex pattern as $\texttt{RIGHT}(e_Q)$. An edge pattern is specified using the notation `-[]->` (denoting a left-to-right directed edge pattern), `<-[]-` (denoting a right-to-left directed edge pattern), and `-[]-` (denoting an undirected edge pattern). The syntax for each describes the existence of a triple in the incidence set I_Q:

$$\begin{aligned}
\texttt{()-[}e_Q\texttt{]->()} &\implies (\texttt{LEFT}(e_Q), e_Q, \texttt{RIGHT}(e_Q)) \in I_Q \\
\texttt{()<-[}e_Q\texttt{]-()} &\implies (\texttt{RIGHT}(e_Q), e_Q, \texttt{LEFT}(e_Q)) \in I_Q \\
\texttt{()-[}e_Q\texttt{]-()} &\implies (\texttt{LEFT}(e_Q), e_Q, \texttt{RIGHT}(e_Q)) \in I_Q \vee (\texttt{RIGHT}(e_Q), e_Q, \texttt{LEFT}(e_Q)) \in I_Q
\end{aligned} \tag{5.3}$$

As far as the edge detail goes, an edge pattern may optionally contain a) a variable used to partially define ϑ for e_Q, b) an set of labels B^{e_Q} used to partially define the labeling function λ_Q such that $\lambda_Q(e_Q) = B^{e_Q}$, and c) another `WHERE` clause shorthand for further qualifying the "mapped-to" graph edges (i.e., $m_e(e_Q)$ for some morphism $m \in M$).

We will now describe the `EdgePattern` production through example. Consider the expression `(u:User)-[w:WROTE]->(m:Message)`. This expression defines two vertex patterns v_1, v_2, and one edge pattern e. v_1 is assigned the variable $\vartheta(v_1) = \texttt{u}$, v_2 is assigned the variable $\vartheta(v_2) = \texttt{m}$, and e is assigned the variable $\vartheta(e) = \texttt{w}$. v_1 is labeled as $\lambda_Q(v_1) = \{\texttt{User}\}$, v_2 is labeled as $\lambda_Q(v_2) = \{\texttt{Message}\}$, and e is labeled as $\lambda_Q(e) = \{\texttt{WROTE}\}$. We note that the left vertex pattern of e is $\texttt{LEFT}(e) = v_1$, and the right vertex pattern of e is $\texttt{RIGHT}(e) = v_2$. The `EdgePattern` expression is directed left-to-right, therefore the triple $(v_1,\ e,\ v_2)$ exists in the incidence set: $(v_1,\ e,\ v_2) \in I_Q$.

Listing 5.6 illustrates a pattern matching query in gSQL^{++}. The graph G_D is specified using `GRAPH SocialNetworkGraph` on Line 2, with the query pattern G_Q specified on Line 3 and Line 4. The vertex patterns consist of $V_Q = \{v_u,\ v_f,\ v_m,\ v_r\}$, the edge patterns consist of $E_Q = \{e_k,\ e_w,\ e_{ro}\}$, and the incidence triple set consists of $I_Q = \{(v_u, e_k, v_f),\ (v_f, e_w, v_m),\ (v_m, e_{ro}, v_r)\}$. For simplicity, we denote the subscript of each graph element as its bound variable (e.g., $\vartheta(v_u) = \texttt{u}$). The labeling function λ_Q assigns the following vertices and edges to labels: $\lambda_Q(v_u) = \lambda_Q(v_f) = \{\texttt{User}\}$, $\lambda_Q(v_m) = \lambda(v_r) = \{\texttt{Message}\}$, $\lambda_Q(e_k) = \{\texttt{User}\}$, $\lambda_Q(e_w) = \{\texttt{WROTE}\}$, $\lambda_Q(e_{ro}) = \{\texttt{REPLY\_OF}\}$. For all morphisms M that map G_Q to subgraphs of G_D, the result of Listing 5.6 returns the application of *each* morphism $(m_v,\ m_e) \in M$ to all vertex and edge patterns in G_Q. Logically, after Line 3, a gSQL^{++} user is free to manipulate the mapped vertices and edges using the variables bound to the query pattern vertices and edges (e.g., using `u`, `m`, and `w` to reference $m_v(v_u)$, $m_v(v_m)$, and $m_e(e_w)$ respectively).

Assume that Listing 5.7 describes a result in the result set for the query in Listing 5.6. As with all SQL^{++} queries, a result in gSQL^{++} is a value in the AsterixDB data model. For our current example, we have a document whose top level fields (e.g., `"u"`, `"k"`, etc. . .) refer to the variables bound in the graph pattern. More generally, the absence of a projection in SQL^{++} and gSQL^{++} (as denoted by `SELECT *`) means that a result is a document whose top level fields are the names of *all* variables in scope. With respect to result size, the number

```
FROM
    GRAPH SocialNetworkGraph
        (u:User)-[k:KNOWS]->(f:User)-[w:WROTE]->(m:Message),
        (m)-[ro:REPLY_OF]->(r:Message)
WHERE
    u.id = 94
SELECT *;
```

Listing 5.6: gSQL^{++} query to find the replies `r` to messages `m` of a user `f` known by some other user `u` whose ID field is equal to 94.

```
{
  "u" : { "id"        : 94,
          "name"      : { "first": "Mococo", "last": "Abyssgard" },
          "join_date" : "2023-07-30",
          "knows"     : [55,90,91,92,93] },
  "k" : { "source_id" : 94,
          "dest_id"   : 91 },
  "f" : { "id"        : 91,
          "name"      : { "first": "Bijou", "last": "Koseki" },
          "join_date" : "2023-07-29",
          "knows"     : [90,91,92,93,94] },
  "w" : { "user_id"   : 91,
          "message_id": 30820,
          "posted_on" : "2023-08-14:T00:00:12Z" },
  "r" : { "id"        : 30820,
          "user_id"   : 91,
          "posted_on" : "2023-08-14:T00:00:12Z",
          "content"   : "Try the pet store down the street!",
          "reply_id"  : 30819,
          "is_draft"  : FALSE },
  "ro": { "source_id" : 30820,
          "dest_id"   : 30819,
          "posted_on" : "2023-08-14:T00:00:12Z" },
  "m" : { "id"        : 30819,
          "user_id"   : 93,
          "posted_on" : "2023-08-13:T10:02:23Z",
          "content"   : "Does anyone know where to buy dog food?",
          "is_draft"  : FALSE }
}
```

Listing 5.7: One result found in the result set of the query in Listing 5.6.

of results returned the query in Listing 5.6 is equal to the size of the morphism set $|M|$. Assuming that the morphism set initially starts off as $|M| = n_0$, we describe a sequence of updates to the graph and the size of the result set for the same query in Listing 5.6 after each update:

$t = 1$ User 91 writes one more reply to another message. Observing that a "`REPLY_OF`"-labeled edge represents a 1:1 relationship, the result size increases by one: $|M| = n_0 + 1$.

$t = 2$ User 94 adds another user to their "knows" list. Observing that a "`WROTE`"-labeled edge represents a 1:N relationship, the result size increases by the number of replies to messages that our new user has posted. For our example, suppose that this new user has posted 5 replies. The result size thus increases by 5: $|M| = n_0 + 1 + 5$.

$t = 3$ User 91 writes a top-level message that isn't a reply to any other message. This update to the graph does *not* increase the result size, as "user knows a user who posted a message" isn't a subgraph (from G_D) that matches G_Q. A subgraph must be fully matched, unless the "reply of message" query sub-pattern is specified as optional (see Subsection 5.4.1).

For more complex examples of graph pattern matching queries in gSQL^{++}, see Subsection 5.4.1 and Subsection 5.4.2.

5.3 Navigational Queries

SQL^{++} (or more generally, non-recursive SQL) can directly express each query in Section 5.2. In fact, early versions of non-recursive Graphix acted more as a "transpiler" for gSQL^{++} to SQL^{++}. Most graph query languages, however, are often characterized by another construct: the ability to express transitivity between two query pattern vertices. Specifically, we introduce the notion of a *path*: a query construct that describes the relationship between two query pattern vertices using two sequences of graph instance vertices and graph instance edges. In this section we will first detail how paths are described using regular expressions, and then explain how these regular expressions are specified in gSQL^{++}.

5.3.1 Path Finding (Navigation)

To start, we are given a graph instance $G_D = (I_D,\ V_D,\ E_D)$ where I_D is the incidence triple set, V_D is a set of graph vertices, and E_D is sets of edges. A path $p = (V_p,\ E_p)$ is a two-tuple consisting of a sequence of vertices $V_p = (v_1,\ v_2,\ \ldots\ v_n)$ and a sequence of edges $E_p = (e_1,\ e_2,\ \ldots\ e_m)$, where p possess the following properties:

1. all path vertices exist in the graph instance $v \in V_p \implies v \in V_D$;
2. all path edges exist in the graph instance $e \in E_p \implies e \in E_D$;
3. there exists at least one vertex $|V_p| > 0$;
4. there are two vertices per edge $|V_p| = |E_p| + 1$; and
5. for an edge $E_P[i]$ at position i of E_P and two vertices $V_P[i]$, $V_P[i+1]$ at positions i and $i+1$ of V_P, all graph elements are related: $(V_P[i],\ E_P[i],\ V_P[i+1]) \in I_D$.

Given two graph instance vertices $v_1 \in V_D$, $v_2 \in V_D$, the *unconstrained* problem of path "finding" involves finding all paths from v_1 (the source vertex) to v_2 (the destination vertex): $P_{v_1.v_2} = \{(V_p,\ E_p) \mid v_1 = V_p[1] \text{ and } v_2 = V_p[N]\}$. For paths containing cycles, enumerating all satisfiable paths is impossible (i.e., a longer path can always be found). Consequently, all graph query languages solve variations of the path finding problem that guarantee finite results (assuming that the graph is also finite). We list the most common variations below:[‡]

[‡]We only consider variants that preserve the paths, ruling out problem variants that ask instead about the *existence* of a path between two vertices (e.g., SPARQL's problem variant).

Any k Paths The "any k paths" problem (alt. the "reachability" problem when $k = 1$) involves finding any k-sized subset of paths $P_{v_1.v_2}$ from vertex v_1 to vertex v_2. We denote this subset as $P^{\text{ANY}(k)}_{v_1.v_2}$, where $P^{\text{ANY}(k)}_{v_1.\ v_2} \subset P_{v_1.v_2}$ and $|P^{\text{ANY}(k)}_{v_1.v_2}| = k$.

Shortest k Paths The "shortest k paths" problem involves finding some k-sized subset of paths $P^{\text{SHO}(k)}_{v_1.v_2} \subset P_{v_1.v_2}$ where the path set has exactly k paths $|P^{\text{SHO}(k)}_{v_1.v_2}| = k$ and for all paths $(V_i,\ E_i) \in P^{\text{SHO}(k)}_{v_1.v_2}$, there exists no other shorter path: $|E_i| \leq |E_j|\ \forall\ (V_j, E_j) \in (P_{v_1.v_2} \setminus P^{\text{SHO}(k)}_{v_1.v_2})$.

Cheapest k Paths The "cheapest k paths" problem is a generalization of the k shortest paths problem that adds a weight function $c : E \to \mathbb{R}$.[§] Here, we are interested in finding some subset of paths $P^{\text{CHE}(c,k)}_{v_1.v_2} \subset P_{v_1.v_2}$ where the path set is k-sized $|P^{\text{CHE}(c,k)}_{v_1.v_2}| = k$ and for all paths $(V_i,\ E_i) \in P^{\text{CHE}(c,k)}_{v_1.v_2}$, there exists no other cheaper path: $\sum_{n=0}^{|E_i|} c(E_i[n]) \leq \sum_{m=0}^{|E_j|} c(E_j[m])\ \forall\ (V_j,\ E_j) \in (P_{v_1.v_2} \setminus P^{\text{CHE}(c,k)}_{v_1.v_2})$.

All No-Repeat-Edge Paths The "all non-edge repeating paths" problem is a variant that does not quantify the entire path set, but rather *independently* qualifies each path itself. Here, we are interested in finding the subset of all paths between v_1 and v_2, $P^{\text{NRE}}_{v_1.v_2} \subset P_{v_1.v_2}$, where the edge sequence of every path contains no duplicates: $(V_i,\ E_i) \in P^{\text{NRE}}_{v_1.v_2} \implies$ every $e_i \in E_i$ is unique.

All No-Repeat-Vertex Paths The "all non-vertex repeating paths" problem is similar to the all non-edge repeating paths problem, but constrains that every *vertex* in a path is unique rather than every edge. Here, we are interested in finding the subset of all paths between v_1 and v_2, $P^{\text{NRV}}_{v_1.v_2} \subset P_{v_1.v_2}$ where the vertex sequence of every path contains no duplicates: $(V_i,\ E_i) \in P^{\text{NRE}}_{v_1.v_2} \implies$ every $v_i \in V_i$ is unique.

All No-Repeat-Anything Paths In a hypergraph setting, we may have instances where a path repeats an edge but not a vertex: $P^{\text{NRE}}_{v_1.v_2} \not\subset P^{\text{NRV}}_{v_1.v_2}$. Consequently, we define the "all non-vertex-and-edge repeating paths" problem, where both the vertex and edge sequences of every path are constrained. Here, we are interested in finding the subset

[§]As we will later see, in gSQL^{++} the weight function c has a wider domain of *all paths*: $c : P_{v_1.v_2} \to \mathbb{R}$.

of all paths between v_1 and v_2, $P^{\text{NRA}}_{v_1.v_2} \subset P_{v_1.v_2}$ where both the vertex sequence of every path contains no duplicates and the edge sequence of every path contains no duplicates: $(V_i,\ E_i) \in P^{\text{NRA}}_{v_1.v_2} \implies$ every $v_i \in V_i$ is unique $\wedge$ every $e_i \in E_i$ is unique.

Path finding queries in Cypher, by default, represent questions in the "all non-edge repeating paths" problem class. To ask a "shortest path" question, Cypher users must use special functions to change the problem class (i.e., through their `shortestPath` and `allShortestPaths` functions). To ask a "cheapest path" question, Neo4j Cypher users must call a different path function from their data science plugin that has a parameter for a weight (i.e., the `gds.shortestPath.dijkstra` function). The GPML of SQL/PGQ is slightly more unified, giving users various prefix keywords (e.g., `SHORTEST` for shortest path, `ANY` for any path, etc...) to modify the problem class, but as we'll see, gSQL^{++} can be used to express all of the aforementioned problem classes in a much more uniform manner.

We now turn to *regular path queries* (abbr. RPQs), which is another query construct found in modern graph query languages that allows users to further qualify paths p between two vertices $v_1 \in V_D$, $v_2 \in V_D$. RPQs are regular expressions over an alphabet of all edge labels in the graph instance $\mathcal{B}$. Given a regular expression r and some path $p = (V_p,\ E_p)$, let $\mathtt{LANG}(r)$ represent the language accepted by an automaton that simulates r and $\mathtt{WORD}(p)$ represent a sequence of edge labels for p (i.e., the sequence $(\lambda(E_p[1]),\ \lambda(E_p[2]),\ \ldots\ \lambda(E_p[m]))$. We are interested in finding all paths $P^{\text{RE}(r)}$ that *match* the regular expression: $p \in P^{\text{RE}(r)} \implies \mathtt{WORD}(p) \in \mathtt{LANG}(r)$. Attention must be given to the operations we permit in our regular language, otherwise evaluation becomes intractable (as illustrated in [44]). We point to the following operation set, which define how RPQs are expressed in gSQL^{++} and how RPQs were expressed in earlier versions of Cypher before moving to the GPML of the GQL standard:

Alternation Given two expressions s_1 and s_2 where $|s_1| = 1$ and $|s_2| = 1$ (i.e., s_1 and s_2 are either single symbol words or alternations of single symbol words), the regular expression $r = s_1\ |s_2$ defines a language $\mathtt{LANG}(r)$ where every word $w \in \mathtt{LANG}(r)$ has a

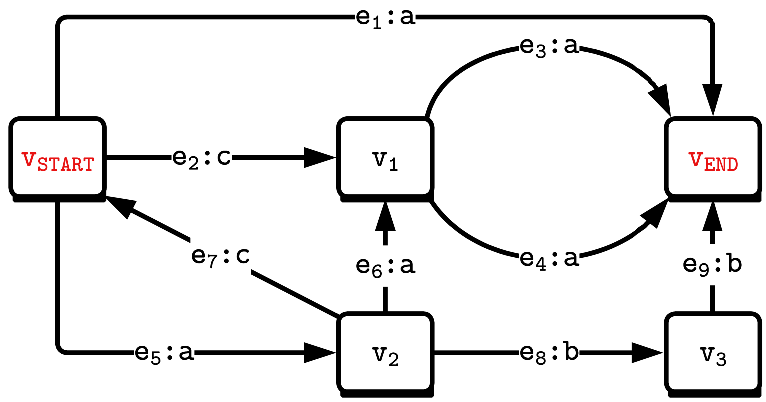


Figure 5.6: An example edge-labeled graph instance G_D that possesses five vertices and nine edges.

length of one $|w| = 1$ and the language is union of the languages associated with the operands $\texttt{LANG}(r) = \texttt{LANG}(s_1) \cup \texttt{LANG}(s_2)$. To match r here involves finding paths that consist of a single edge $e \in E_D$ where the single symbol word $\lambda(e)$ exists in $\texttt{LANG}(r)$.

Quantification Given an expression s where $|s| = 1$ (i.e., s is either a single symbol word or an alternation of single symbol words), assume the regular expression $r = s\{m, n\}$ where $m \in (\{0\} \cup \mathbb{Z})$ is a non-negative integer and $n \in \mathbb{Z}$ is an optional positive integer that is greater than or equal to m. r defines a language $\texttt{LANG}(r)$ where all words $w \in \texttt{LANG}(r)$ have a length between m and n: $w \implies m \leq |w| \leq n$. Unbounded repetition of s is denoted via $s\{m,\ \}$, where the upper limit n is excluded. We also note the following shorthand operators for quantification: a) the Kleene closure $s\texttt{*} = s\{0,\ \}$, and b) the positive closure $s\texttt{+} = s\{1,\ \}$.

Figure 5.6 describes a directed edge-labeled graph instance G_D of five vertices and nine edges. We note that the incidence triples that capture e_5 and e_7 (i.e., $(v_{\texttt{START}}, e_5, v_2)$ and $(v_2, e_7, v_{\texttt{START}})$) induce a cycle in our graph instance. If we enumerate all "non-repeating-edge" paths from vertex $v_{\texttt{START}}$ to vertex $v_{\texttt{END}}$, we get the results in Table 5.2. Now suppose we are given six RPQs to match the paths of Table 5.2 against:

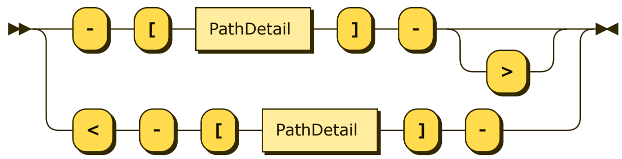
1. $r_1 = \texttt{(a|b|c)*}$
2. $r_2 = \texttt{a*}$
3. $r_3 = \texttt{(a|b)*}$
4. $r_4 = \texttt{(a|c)*}$
5. $r_5 = \texttt{a\{2,5\}}$
6. $r_6 = \texttt{b*}$

p_i	V_i (Vertex Sequence of p_i)	E_i (Edge Sequence of p_i)	$\texttt{WORD}(p_i)$
p_1	$(v_{\texttt{START}},\ v_{\texttt{END}})$	(e_1)	a
p_2	$(v_{\texttt{START}},\ v_1,\ v_{\texttt{END}})$	$(e_2,\ e_3)$	ca
p_3	$(v_{\texttt{START}},\ v_1,\ v_{\texttt{END}})$	$(e_2,\ e_4)$	ca
p_4	$(v_{\texttt{START}},\ v_2,\ v_1,\ v_{\texttt{END}})$	$(e_5,\ e_1,\ e_3)$	aaa
p_5	$(v_{\texttt{START}},\ v_2,\ v_1,\ v_{\texttt{END}})$	$(e_5,\ e_1,\ e_4)$	aaa
p_6	$(v_{\texttt{START}},\ v_2,\ v_3,\ v_{\texttt{END}})$	$(e_5,\ e_8,\ e_9)$	abb
p_7	$(v_{\texttt{START}},\ v_2,\ v_{\texttt{START}},\ v_{\texttt{END}})$	$(e_5,\ e_7,\ e_1)$	aca
p_8	$(v_{\texttt{START}},\ v_2,\ v_{\texttt{START}},\ v_1,\ v_{\texttt{END}})$	$(e_5,\ e_7,\ e_2,\ e_3)$	aca
p_9	$(v_{\texttt{START}},\ v_2,\ v_{\texttt{START}},\ v_1,\ v_{\texttt{END}})$	$(e_5,\ e_7,\ e_2,\ e_4)$	aca

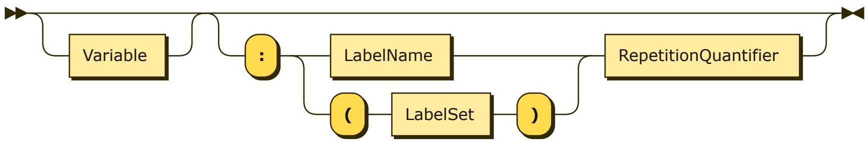
Table 5.2: A table describing all "non-repeating-edge" paths between $v_{\texttt{START}}$ and $v_{\texttt{END}}$ in the graph instance of Figure 5.6.

r_1 matches all nine paths. r_2 matches all paths that only consist of a-labeled edges: p_1, p_4, and p_5. r_3 matches paths that only consist of a-labeled edges or b-labeled edges: p_1, p_4, p_5, and p_6. r_4 matches paths containing only a-labeled edges and c-labeled edges (i.e., all paths in except for p_6). r_5 matches all paths that only consist of a-labeled edges whose word size is between 2 and 5: p_4 and p_5. Finally, r_6 matches no paths in G_D, as there exists no path from $v_{\texttt{START}}$ to $v_{\texttt{END}}$ that is only composed of b-labeled edges.

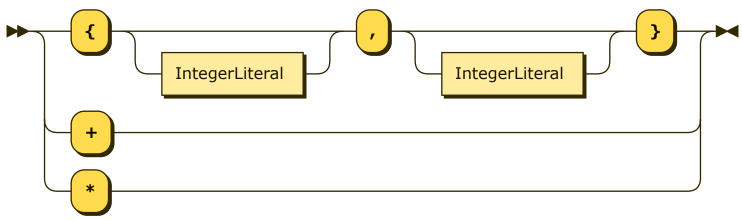
Note that concatenation is *not* a supported operation, eliminating the possibility of backtracking while trying to find a match. For the same reason, negation is not a supported operation. Such an operation set may seem too restrictive for non-trivial problems, but when combined with pattern matching (and later all of SQL^{++}), users can express a rich set of path finding queries. We will now define the problem of *navigational* pattern matching. We extend the definition for a query pattern G_Q to now include a set of regular path queries: $G_Q = (V_Q,\ E_Q,\ I_Q,\ \lambda_Q,\ R_Q)$. We expand the types of incidence triples found in I_Q to include RPQs: $I_Q \subset (V_Q \times (R_Q \cup E_Q) \times V_Q)$. We *do not* expand the domain of the labeling function λ, as RPQs are not (at least formally) graph patterns to match G_D against. In addition to finding morphisms $(m_v,\ m_e) \in M$ that map the query pattern $(V_Q,\ E_Q,\ I_Q)$ to the graph instance G_D, the problem of navigational pattern matching is also concerned with finding *all* paths $P^{\mathrm{RE}(r)}$ for *all* RPQs $r \in R_Q$ between the incident vertices of r itself (i.e.,



(a) gSQL^{++} grammar for the `PathPattern` production.



(b) gSQL^{++} grammar for the `PathDetail` production.



(c) gSQL^{++} grammar for the `RepetitionQuantifier` production.

Figure 5.7: Grammar used to describe a RPQ (i.e., the `PathPattern`, `PathDetail`, and `RepetitionQuantifier` productions).

where $(m_v(v_1),\ r,\ m_v(v_2)) \in I_Q$ for $v_1 \in V_Q \wedge v_2 \in V_Q)$. The existence of a path $p_r \in P^{\text{RE}(r)}$ for some RPQ r implies that the word of the path $\texttt{WORD}(p_r)$ exists in the accepting language $\texttt{LANG}(r)$. Modern graph languages are expected to, at a minimum, provide query constructs for solving navigational pattern matching problems.

5.3.2 gSQL^{++} for Navigation

We will now continue (and ultimately conclude) our discussion about the `MatchExpr` production by defining the `PathPattern` production. A `PathPattern` allows users to describe RPQs in conjunction with the pattern matching constructs of Section 5.2 to express navigational pattern matching queries. Figure 5.7 describes the grammar for a path pattern, used to specify RPQs r_Q in gSQL^{++}. We refer to the left vertex pattern of a path pattern as $\texttt{LEFT}(r_Q)$, and the right vertex pattern of a path pattern as $\texttt{RIGHT}(r_Q)$. Similar to an `EdgePattern`, a `PathPattern` can be left-directed (`-[]->`), right-directed (`<-[]-`), and undirected (`-[]-`). The syntax for each describes the existence of a triple in our incidence set I_Q:

$$
\begin{aligned}
\texttt{()-[}r_Q\texttt{]->()} &\implies (\texttt{LEFT}(r_Q), r_Q, \texttt{RIGHT}(r_Q)) \in I_Q \\
\texttt{()<-[}r_Q\texttt{]-()} &\implies (\texttt{RIGHT}(r_Q), r_Q, \texttt{LEFT}(r_Q)) \in I_Q \\
\texttt{()-[}r_Q\texttt{]-()} &\implies (\texttt{LEFT}(r_Q), r_Q, \texttt{RIGHT}(r_Q)) \in I_Q \vee (\texttt{RIGHT}(r_Q), r_Q, \texttt{LEFT}(r_Q)) \in I_Q
\end{aligned}
\tag{5.4}
$$

With respect to the path detail, the `PathPattern` production contains two items: 1) a variable used to partially define our variable mapping function ϑ for r_Q, and 2) a regular expression of edge labels (r_Q itself) using the `LabelSet` and `RepetitionQuantifier` productions.

Listing 5.8 illustrates a navigational pattern matching query in gSQL^{++}. The graph G_D is specified using `GRAPH SocialNetworkGraph` on Line 2. The query pattern G_Q is specified on Line 3 and Line 4. The vertex patterns consist of $V_Q = \{v_u,\ v_f,\ v_m,\ v_r\}$, the edge patterns consist of $E_Q = \{e_k,\ e_{w_1},\ e_{w_2},\ e_{ro}\}$, and the path patterns consists of $R_Q = \{r_k\}$ where the regular expression $r_k = \texttt{KNOWS*}$. The incidence triple set consists of $I_Q = \{(v_u, r_k, v_f)$, (v_u, e_{w_1}, v_m), (v_f, e_{w_2}, v_r), $(v_r, e_{ro}, v_m)\}$. For simplicity, we again denote the subscript of each graph element as its bound variable (e.g., $\vartheta(v_u) = \texttt{u}$). The labeling function λ_Q assigns the following vertices and edges to labels: $\lambda_Q(v_f) = \lambda_Q(v_u) = \{\texttt{User}\}$, $\lambda_Q(v_r) = \lambda_Q(v_m) = \{\texttt{Message}\}$, $\lambda_Q(e_{w_1}) = \lambda_Q(e_{w_2}) = \{\texttt{WROTE}\}$, $\lambda_Q(e_{ro}) = \{\texttt{REPLY\_OF}\}$.

Assume that Listing 5.9 describes a result in the result set for the query in Listing 5.8. We observe that a path $p = (V_p, E_p)$ is represented in gSQL^{++} as a document of two fields: an array-valued field `"Vertices"`, used to describe the sequence of vertices V_p, and an array-valued field `"Edges"`, used to describe the sequence of edges E_p. The first element of the vertex sequence is the document bound to `u`, while the last element of the vertex sequence is the document bound to the incident vertex `v`. As with vertex and edge instances, users can manipulate paths using the same SQL^{++} query constructs they would use for any other

```
FROM
    GRAPH SocialNetworkGraph
        (f:User)<-[k:KNOWS+]-(u:User)-[w1:WROTE]->(m:Message),
        (f)-[w2:WROTE]->(r:Message)-[ro:REPLY_OF]->(m)
WHERE
    u.id = 67 AND
    f.id = 60
SELECT
    u,
    k,
    f;
```

Listing 5.8: gSQL^{++} query to return all paths of `KNOWS` edges between two users `u`, `f` where `f` has written a reply for some message written by `u`.

```
{
  "u": { "id"         : 67,
         "name"       : { "first": "Ouro", "last": "Kronii" },
         "join_date" : "2021-08-22",
         "languages" : ["en", "kr"],
         "knows"      : [59,65,66,68,69] },
  "k": { "Vertices": [
           { "id"         : 67,
             "name"       : { "first": "Ouro", "last": "Kronii" },
             "join_date" : "2021-08-22",
             "languages" : ["en", "kr"],
             "knows"      : [59,65,66,68,69] },
           { "id"         : 59,
             "name"       : { "first": "Gura", "last": "Gawr" },
             "join_date" : "2020-09-13",
             "knows"      : [56,57,58,60,67] },
           { "id"         : 60,
             "name"       : { "first": "Amelia", "last": "Watson" },
             "join_date" : "2020-09-13",
             "knows"      : [56,57,59,60] }
         ],
         "Edges": [ { "source_id" : 67, "dest_id" : 59 },
                    { "source_id" : 59, "dest_id" : 60 } ] },
  "f": { "id"         : 60,
         "name"       : { "first": "Amelia", "last": "Watson" },
         "join_date" : "2020-09-13",
         "knows"      : [56,57,58,59] }
}
```

Listing 5.9: One result found in the result set of the query in Listing 5.8

document. The vertices of a path can be accessed using `k.Vertices` (or using the Graphix function `VERTICES(k)`) and the edges of a path can be accessed using `k.Edges` (or, using the Graphix function `EDGES(k)`). With respect to result size, the number of results returned by the query in Listing 5.8 is determined by the size of the morphism set $|M|$ *multiplied* by the number of satisfying paths $|P|$ in the graph instance G_D. Assuming that the morphism set size initially starts off as $n_0^{|M|}$ and the path set size initially starts off as $n_0^{|P|}$, we describe a sequence of updates to the graph and the size of the result set for the same query in Listing 5.8 after each update:

$t = 1$ User 60 writes another message replying to one of user 67's messages. Observing that a "`REPLY_OF`"-labeled edge represents a 1:1 relationship, the result size increases by a factor of the *path set* size $n_0^{|P|}$. The result size is now $(n_0^{|M|} + 1) \times n_0^{|P|}$.

$t = 2$ Users 60 and 67 add a mutual user to their "knows" list. The result size increases by a factor of *morphism set* size $(n_0^{|M|} + 1)$. The result size is now $(n_0^{|M|} + 1) \times (n_0^{|P|} + 1)$.

$t = 3$ User 67 adds a user to their "knows" list that isn't in user 60's "knows" list. This update does not increase the path set size, as this added edge does not yield a new path from user 67 to 60. Consequently, this update does not increase the result size.

$t = 4$ User 67 writes a new message that does not have any replies. This update to the graph also does not increase the result size, as "user posts a message" is not a subgraph (from G_D) that matches the query. A query pattern must be fully matched.

For more complex examples of navigational pattern matching queries in gSQL^{++}, see Subsection 5.4.3, Subsection 5.4.4, and Subsection 5.4.5.

5.4 Complex gSQL^{++} Examples

In this section, we will provide a series of more complex gSQL^{++} queries to illustrate the implications of defining vertices, edges, and paths as documents in SQL^{++}. These include: i) optional subgraph matching, ii) negative subgraph matching, iii) subgraph reachability, iv) shortest path finding, and v) cheapest path finding.

5.4.1 Optional Subgraph Matching

Mirroring `LEFT JOIN` in SQL (and SQL^{++}), `LEFT MATCH` in gSQL^{++} allows users to specify an optional navigational query pattern $G_Q^{\texttt{OPT}}$ attached to some mandatory query pattern $G_Q^{\texttt{REQ}}$ where one or more vertices are shared between both patterns (formally, $\exists\, v \in V_Q^{\texttt{OPT}}$ such that $v \in V_Q^{\texttt{REQ}}$). Listing 5.10 depicts a gSQL^{++} query with a `LEFT MATCH` clause which asks for all users `u1`, and *optionally* any users `u2` that `u1` knows where `u2` has posted a reply to one of `u1`'s messages. In Listing 5.10, the mandatory query pattern $G_Q^{\texttt{REQ}}$ consists of one vertex labeled `User` (specified on Line 3) and the optional query pattern $G_Q^{\texttt{OPT}}$ is specified on Line 5, Line 6, and Line 7. Note that the vertex pattern bound to the variable `u1` is shared across both $G_Q^{\texttt{REQ}}$ and $G_Q^{\texttt{OPT}}$. If we assume that executing the Listing 5.10 query yields the three results in Listing 5.12, we observe that the first two results have matched both $G_Q^{\texttt{REQ}}$ and $G_Q^{\texttt{OPT}}$ while the third result only matches $G_Q^{\texttt{REQ}}$.

To explain the semantics of `LEFT MATCH`, we can leverage gSQL^{++}'s interoperability with SQL^{++} to express an alternative to the query in Listing 5.10. Listing 5.11 details an alternative to Listing 5.10 where `LEFT JOIN` is used in place of `LEFT MATCH`. The `LEFT JOIN` condition consists of an equality of the `"id"` property (i.e., the primary key given for "`User`"-labeled vertices in the definition of G_D — more detail is given in Subsection 6.3.2) of the vertex pattern bound to the `u1` variable and the `"id"` property of the inner vertex pattern

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)
    LEFT MATCH
        (u1)-[:KNOWS]->(u2:User),
        (u2)-[:WROTE]->(m2:Message),
        (m2)-[:REPLY_OF]->(:Message)<-[:WROTE]-(u1)
SELECT DISTINCT
    u1.name.first AS u1_name,
    u2.name.first AS u2_name;
```

Listing 5.10: gSQL^{++} query to enumerate users `u1`, and optionally the users `u2` known by `u1` that have posted a reply to a message posted by `u1`.

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)
    LEFT JOIN
        ( FROM
              GRAPH SocialNetworkGraph
                  (u1i:User)-[:KNOWS]->(u2:User),
                  (u2)-[:WROTE]->(m2:Message),
                  (m2)-[:REPLY_OF]->(:Message)<-[:WROTE]-(u1i)
          SELECT
              u1i.id AS id,
              u2     AS u2 ) AS u1ku2
        ON u1ku2.id = u1.id
SELECT DISTINCT
    u1.name.first       AS u1_name,
    u1ku2.u2.name.first AS u2_name;
```

Listing 5.11: gSQL^{++} alternative to the query in Listing 5.10 that uses `LEFT JOIN` instead of `LEFT MATCH`.

```
{ "u1_name": "Mel",   "u2_name": "Choco" }
{ "u1_name": "Choco", "u2_name": "Mel" }
{ "u1_name": "Aqua" }
```

Listing 5.12: Result set for the queries in Listing 5.10 and Listing 5.11.

bound to the `u1i` variable. From our alternative query, we can infer that `LEFT MATCH` *does not* include partially matched patterns within its containing query pattern $G_Q^{\texttt{OPT}}$. For example, the subgraph from G_D "user knows a user that posted a message" does not fully match the query pattern, as such a result would not be returned in the `LEFT JOIN` right-hand subquery (starting on Line 5 and ending on Line 12).

5.4.2 Negative Subgraph Matching

Both the SQL standard and SQL^{++} do not explicitly support an "anti-`JOIN`" clause. Instead, SQL and SQL^{++} users can express a universally negative predicate that should hold for each "positive" record (typically via a "`NOT EXISTS`" subquery conjunct in the `WHERE` clause). gSQL^{++} users are expected to use this same universal logic to express "anti-navigational-query-patterns". Given a positive query pattern $G_Q^{\texttt{POS}}$ and a negative query pattern $G_Q^{\texttt{NEG}}$ where one or more vertices are shared between both patterns (formally, $\exists\, v \in V_Q^{\texttt{POS}}$ such that $v \in V_Q^{\texttt{NEG}}$), negative pattern matching involves finding the difference in matches to subgraphs of G_D. Listing 5.13 illustrates an example of negative pattern matching query where $G_Q^{\texttt{POS}}$ is specified on Line 3 and $G_Q^{\texttt{NEG}}$ is specified on Line 9. If we assume that Listing 5.15 contains a result found in the result set of the query in Listing 5.13, we observe that the resulting user has an empty `knows` list.

Listing 5.13 also illustrates a gSQL^{++} shorthand for expressing shared vertex patterns across different query patterns. In this example, the vertex pattern bound to the variable `u` is shared between the positive query pattern $G_Q^{\texttt{POS}}$ and the negative query pattern $G_Q^{\texttt{NEG}}$ even though both are specified in different query blocks. Again, we can leverage gSQL^{++}'s interoperability with SQL^{++} to express an alternative to the query in Listing 5.13. In Listing 5.14, we illustrate an alternative that does not leverage this vertex pattern sharing shorthand. In Listing 5.14, no vertex patterns are shared between the positive query pattern $G_Q^{\texttt{POS}}$ and

```
FROM
    GRAPH SocialNetworkGraph
        (u:User)-[:WROTE]->(:Message)
WHERE
    CONTAINS(m.content, u.name.first) AND
    NOT EXISTS (
        FROM
            GRAPH SocialNetworkGraph
                (u)-[:KNOWS]->(:User)
        SELECT *
    )
SELECT DISTINCT VALUE
    u;
```

Listing 5.13: gSQL^{++} query to enumerate all users `u` that a) possess an empty "knows" list and b) have posted at least one message `m` whose content contains their first name.

```
FROM
    GRAPH SocialNetworkGraph
        (u:User)-[:WROTE]->(m:Message)
WHERE
    CONTAINS(m.content, u.name.first) AND
    NOT EXISTS (
        FROM
            GRAPH SocialNetworkGraph
                (ui:User)-[:KNOWS]->(:User)
        WHERE
            ui.id = u.id
        SELECT *
    )
SELECT DISTINCT VALUE
    u;
```

Listing 5.14: gSQL^{++} alternative to the query in Listing 5.13 that explicitly performs a `JOIN` between vertex patterns.

```
{
  "id"       : 12,
  "name"     : { "first": "Aqua", "last": "Minato" },
  "join_date": "2018-08-08",
  "knows"    : []
}
```

Listing 5.15: Result found in the result set for the queries in Listing 5.13 and Listing 5.14.

negative query pattern $G_Q^{\texttt{NEG}}$. Instead, the vertex pattern bound to the variable `u` is explicitly joined with the vertex pattern bound to the variable `ui` (using SQL / SQL^{++}). gSQL^{++} gives Graphix users the ability to choose between either style.

5.4.3 Subgraph Reachability

For well-connected graphs that follow the power law [17] (i.e., the number of vertices with degree x is proportional to $x^{-\alpha}$ where α is a constant), it might be infeasible to enumerate all paths in the graph itself. In these graphs, asking instead about the existence of *any* path between two vertices may be more appropriate. The *reachability* problem is a member of the path finding class of problems that is only concerned with the existence of a path, rather than counting all satisfiable paths themselves. Both Listing 5.16 and Listing 5.17 detail navigational pattern matching reachability queries in gSQL^{++} that involve the RPQ $r =$ `KNOWS+`. The `WHERE` clause in both queries define the vertices that should be incident to the resulting paths. In Listing 5.16, we ask for the `DISTINCT` endpoints of the path pattern bound to the variable `k` and *not* the path pattern itself, reducing the overall difficulty¶ of the current path finding problem. Listing 5.17 asks for distinct endpoints as well, but does so using a `GROUP BY` clause. Both the `SELECT DISTINCT` and `GROUP BY` are query constructs from SQL / SQL^{++}, again allowing gSQL^{++} users to apply their existing knowledge on SQL duplicate elimination to the problem of vertex reachability. Graphix is able to recognize this reduction in problem difficulty for queries that contain an aggregation of incident vertex patterns to path patterns to generate a query plan that does not enumerate all paths.

For completeness, assume that the execution of the query in Listing 5.16 yields the five results in Listing 5.18. From the result set, we can conclude that a) user 56 is connected to

¶Proving the existence of some path satisfying an RPQ r is tractable with the operation set of Subsection 5.3.1, however, enumerating / counting all paths that satisfy r is at least #P-complete [43].

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)-[k:KNOWS+]->(u2:User)
WHERE
    u1.id IN [56,57,58] AND
    u2.id IN [90,91,92]
SELECT DISTINCT
    u1.id AS u1_id,
    u2.id AS u2_id;
```

Listing 5.16: gSQL^{++} query to determine the reachability via "KNOWS"-labeled edges between two groups of vertices.

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)-[k:KNOWS+]->(u2:User)
WHERE
    u1.id IN [56,57,58] AND
    u2.id IN [90,91,92]
GROUP BY
    u1.id AS u1_id,
    u2.id AS u2_id
SELECT
    u1_id,
    u2_id;
```

Listing 5.17: gSQL^{++} alternative to the query in Listing 5.16 that specifies duplicate elimination via a **GROUP BY** clause.

```
{ "u1_id": 56, "u2_id": 90 }
{ "u1_id": 56, "u2_id": 91 }
{ "u1_id": 56, "u2_id": 92 }
{ "u1_id": 57, "u2_id": 90 }
{ "u1_id": 57, "u2_id": 91 }
```

Listing 5.18: Result set for the queries in Listing 5.16 and Listing 5.17.

users 90, 91, and 92, b) user 57 is connected to users 90 and 91, c) user 58 is not connected to users 90, 91, and 92.

5.4.4 Shortest Path Finding

The *shortest path* problem is another member of the path finding class of problems (more formally defined in Subsection 5.3.1), which asks for the path (or one of the paths) that have the least amount of edges among all (satisfiable) paths. Assume that we have the graph in Figure 5.8, and that we are interested in finding three paths: i) the shortest path from vertex v_{56} to vertex v_{90}, ii) the shortest path from vertex v_{56} to vertex v_{91}, and iii) the shortest path from vertex v_{56} to vertex v_{92}. Listing 5.19 depicts how we would express such a shortest path finding query in gSQL^{++}. To start, Line 3 defines a query pattern containing the RPQ r = `KNOWS+` and vertex patterns (assigned the variables `u1` and `u2`) incident to the path pattern (assigned the variable `k`). The `WHERE` clause in the subsequent two lines define the exact vertices that should be incident to the resulting paths (`u1.id = 56` and `u2.id IN [90,91,92]`). The `GROUP BY` clause in Line 7 then aggregates *all* possible paths from `u1` to each `u2` and binds each group of paths to the variable `g`. To fetch the shortest path from `u1` to each `u2`, the subquery in Line 12 is used. Due to the `GROUP BY` clause, this subquery is logically executed for each `u2` instance. Each path `g.k` from `u1` to a `u2` instance is sorted (in ascending order) by the number of hops in `g.k`. To quantify the hops in a path, `LEN(g.k.Edges)` is used to count the number of edges a given path possesses. Once sorted, only the shortest path (or one of the shortest paths, if there are ties) is returned due to the `LIMIT 1`. Finally, the `[0]` on Line 21 is used to access the sole element of that is returned by the subquery (needed since in SQL^{++} / gSQL^{++}, subqueries always return a multiset — `ORDER BY` subqueries return an array, hence the need for the array access) [14]. As with our previous section on reachability, Graphix is able to recognize this reduction in problem difficulty to generate a query plan that avoids enumerating all paths.

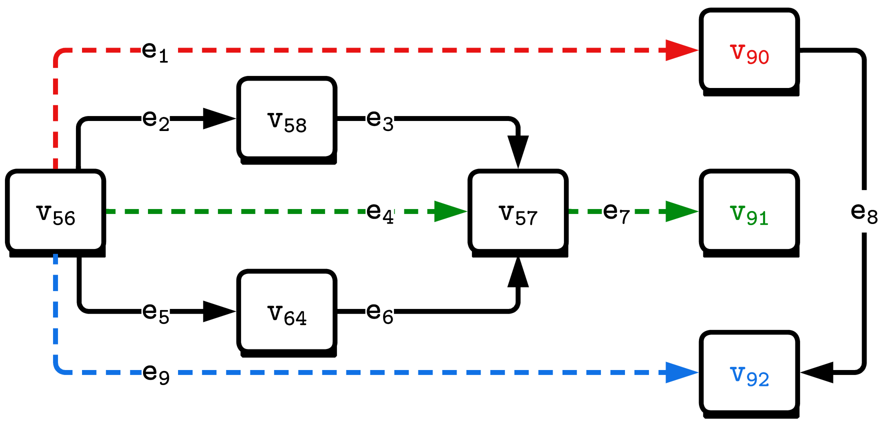


Figure 5.8: Example graph instance G_D that possesses seven vertices and eight edges. The shortest path to each endpoint (v_{90}, v_{91}, v_{92}) from vertex v_{56} is colored and dashed.

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)-[k:KNOWS+]->(u2:User)
WHERE
    u1.id = 56 AND
    u2.id IN [90,91,92]
GROUP BY
    u1.id AS u1_id,
    u2.id AS u2_id
    GROUP AS g
LET
    shortestPath = (
        FROM
            g
        SELECT VALUE
            ( FROM g.k.Vertices v SELECT VALUE v.id )
        ORDER BY
            LEN(g.k.Edges) ASC
        LIMIT
            1
    )[0]
SELECT
    u2_id,
    shortestPath;
```

Listing 5.19: gSQL^{++} query to find the shortest path of "KNOWS"-labeled edges from one user to a set of other users.

{ "u1": v_{56}, "u2": v_{90}, "k": $v_{56} \xrightarrow{e_1} v_{90}$ }
{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_2} v_{58} \xrightarrow{e_3} v_{57} \xrightarrow{e_7} v_{91}$ }
{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_4} v_{57} \xrightarrow{e_7} v_{91}$ }
{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_5} v_{64} \xrightarrow{e_6} v_{57} \xrightarrow{e_7} v_{91}$ }
{ "u1": v_{56}, "u2": v_{92}, "k": $v_{56} \xrightarrow{e_1} v_{90} \xrightarrow{e_8} v_{92}$ }
{ "u1": v_{56}, "u2": v_{92}, "k": $v_{56} \xrightarrow{e_9} v_{92}$ }

$$\Downarrow$$

{ "u1": v_{56}, "u2": v_{90}, "k": $v_{56} \xrightarrow{e_1} v_{90}$ }

{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_2} v_{58} \xrightarrow{e_3} v_{57} \xrightarrow{e_7} v_{91}$ }
{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_4} v_{57} \xrightarrow{e_7} v_{91}$ }
{ "u1": v_{56}, "u2": v_{91}, "k": $v_{56} \xrightarrow{e_5} v_{64} \xrightarrow{e_6} v_{57} \xrightarrow{e_7} v_{91}$ }

{ "u1": v_{56}, "u2": v_{92}, "k": $v_{56} \xrightarrow{e_1} v_{90} \xrightarrow{e_8} v_{92}$ }
{ "u1": v_{56}, "u2": v_{92}, "k": $v_{56} \xrightarrow{e_9} v_{92}$ }

Figure 5.9: Example records in scope before the `GROUP BY` clause (on top) and after the `GROUP BY` clause (on bottom).

To better illustrate the functionality of the Listing 5.19 subquery, consider *all* paths in Figure 5.8, which is conceptually in scope before the `GROUP BY` clause. We visualize the records in scope at the top of Figure 5.9. The `GROUP BY` generates three collections of documents, illustrated with the grouping at the bottom of Figure 5.9. Finally, the Listing 5.19 subquery executes over each group, yielding a single-element array containing the record with the shortest path for each endpoint user (i.e., the highlighted record).

5.4.5 Cheapest Path Finding

The *cheapest path* problem is the last member of the path finding class of problems we will address (more formally defined in Subsection 5.3.1), which generalizes the shortest path problem to find a path that minimizes the sum of some weight property of a path's edges. Suppose that we want to minimize the sum of some property ω for paths of `KNOWS` edges. For

```
WITH
    GRAPH WeightedSocialNetworkGraph AS
        VERTEX (:User)
            PRIMARY KEY (id)
            AS Users,
        EDGE (:User)-[:KNOWS]->(:User)
            SOURCE KEY      (source_id)
            DESTINATION KEY (dest_id)
            AS ( FROM
                     GRAPH SocialNetworkGraph
                         (u1:User)-[:KNOWS]->(u2:User),
                         (u1)-[:WROTE]->(m1:Message),
                         (u2)-[:WROTE]->(m2:Message)
                 GROUP BY
                     u1.id AS source_id,
                     u2.id AS dest_id
                 LET
                     omega = COUNT(DISTINCT m1.id) +
                             COUNT(DISTINCT m2.id)
                 SELECT
                     source_id AS source_id,
                     dest_id   AS dest_id,
                     omega     AS omega ),
FROM
    GRAPH WeightedSocialNetworkGraph
        (u1:User)-[k:KNOWS+]->(u2:User)
WHERE
    u1.id IN [56,57,58] AND
    u2.id IN [90,91,92]
GROUP BY
    u1.id AS u1_id,
    u2.id AS u2_id
    GROUP AS g
SELECT VALUE
    ( FROM
          g
      LET
          cost = ( FROM g.k.Edges e SELECT VALUE SUM(e.omega) )[0],
          uids = ( FROM g.k.Vertices v SELECT VALUE v.id )
      SELECT
          cost AS cost,
          uids AS uids
      ORDER BY
          ABS(cost) ASC
      LIMIT 1 )[0];
```

Listing 5.20: gSQL^{++} query to find the cheapest ω-weighted path of "KNOWS"-labeled edges between two sets of users.

two incident "`User`"-labeled vertices (v_{u_i}, v_{u_j}) to some "`KNOWS`"-labeled edge, suppose that we define ω as the total number of messages written by v_{u_i} and v_{u_j}. Listing 5.20 depicts a cheapest ω path finding query between two groups of vertices within a temporary graph in gSQL^{++}. From Line 1 to Line 23, we define a temporary graph `WeightedSocialNetworkGraph` that only exists in the context of the working query. On Line 25, we define the graph instance G_D to use with the navigational query pattern G_Q defined in Line 26. Similar to the previous two section, G_Q consists of two vertex patterns incident to one path pattern which defines the RPQ r = `KNOWS+`. The **`WHERE`** clause defines the vertices that should be incident to the resulting paths. The **`GROUP BY`** clause and subsequent subquery performs the same aggregation and filtering over each group of paths as the shortest path query in Listing 5.19, but instead sorts on the sum of edge weights as opposed to the number of edges.

We divide our further discussion of Listing 5.20 into two parts: 1) the creation of the graph `WeightedSocialNetworkGraph`, and 2) the subquery on Line 35. Starting with our **`WITH GRAPH`** clause, we point to Subsection 4.2.3 for a description of derived properties for *managed* Graphix graphs (i.e., Graphix graphs defined with the **`CREATE GRAPH`** statement). We would now like to remind the reader of an important point made in Chapter 4: we can define Graphix graph vertices and edges using *queries*. "Queries" here not only includes SQL^{++} queries, but also gSQL^{++} queries. After defining a collection of "`User`"-labeled vertices for the temporary graph, we define a collection of "`KNOWS`"-labeled edges (incident to "`User`"-labeled vertices) with a gSQL^{++} query on on Line 9. Our gSQL^{++} query builds a collection of edges / documents that:

1. performs a pattern matching query against the `SocialNetworkGraph` graph where G_Q is the pattern "users that know other users and the messages each user has posted";
2. groups each pattern matching result by the two user vertices;

3. computes an `omega` field for each group that represents the number of messages in the group itself; and
4. returns the declared source key, destination key, and the computed `omega` field.

Focusing on Line 35, we reiterate that this subquery is conceptually executed for each (`u1_id`, `u2_id`) group of paths. On Line 38, a `cost` field is defined as the `SUM` of an edge's `omega` field for some path. Each path `g.k` is then sorted in ascending order by its associated `ABS(cost)`. Finally, the cheapest path is returned using `LIMIT 1`. The use of "`ABS(cost)`" here instead of simply "`cost`" signals (i.e., hints) to Graphix there are no negative weights. Similar to the previous two sections, Graphix is able to recognize this reduction in problem difficulty to generate a query plan that does not enumerate all paths. We remark that our use of subqueries on groups in the previous two sections is a characteristic inherent to SQL^{++} itself. gSQL^{++} enjoy the benefits of issuing full gSQL^{++} *queries* over collections of vertices, edges, and paths thanks to SQL^{++}'s natural support for reasoning about groups and nested data.

Chapter 6

Implementation

Graphix was designed as an extension of AsterixDB to leverage the language features, query plan optimizer and parallel runtime engine of AsterixDB. In this chapter, we will address different layers of the AsterixDB software stack and describe the changes made to each layer. After giving a high-level overview of the Graphix architecture, we will detail our changes to the runtime layer (Hyracks) to realize semi-synchronous recursion. We will then illustrate the Graphix-specific AST-level rewrites that occur immediately after query parsing. We conclude this chapter with our loop-accommodating extensions to the optimization layer (Algebricks).

6.1 Graphix Architecture

From a software engineering standpoint, Graphix falls in the same line of Apache AsterixDB extensions as BAD (Big Active Data) [35] and Couchbase Analytics* [18]. These systems

*Couchbase Analytics chooses to use a subset of hardened AsterixDB features (e.g., no support for schema definition at the time of writing), and is not a "true" extension by definition — however, Couchbase Analytics uses the same extension machinery as Graphix.

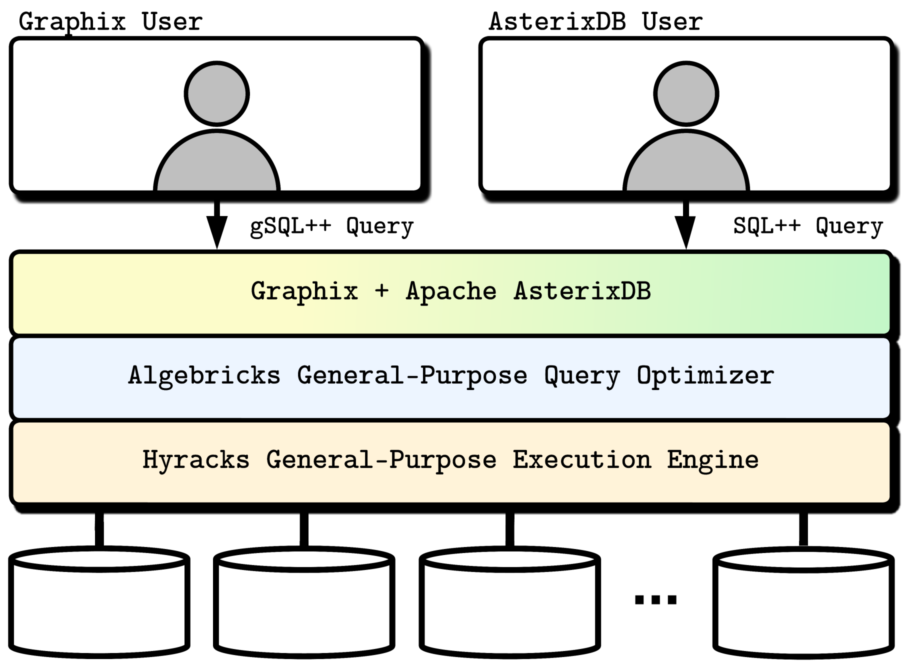


Figure 6.1: High level overview of the Graphix and AsterixDB software stack.

were designed to not only leverage the underlying runtime and optimizer layers, but also AsterixDB's data model and query language. We contrast this family of extensions with past projects that either a) only extended the Hyracks runtime engine (Pregelix) or b) only extended the Algebricks query optimizer and the Hyracks runtime engine (Apache VXQuery, Hivesterix). We also contrast Graphix with applications built on top of Apache AsterixDB: Graphix is *not* middleware, which allows it to reason about the original gSQL^{++} query in the Algebricks and Hyracks layers. A high level overview of how Graphix and the AsterixDB software stack are composed is given in Figure 6.1. We highlight the fact that existing AsterixDB users do not have to make any changes to their workflow here: both Graphix and AsterixDB users issue their queries through the same API endpoint.

Without loss of generality, we will assume the two-machine architecture in Figure 6.2 to host the social network database from Section 3.2 for the remainder of this chapter. Both AsterixDB and Graphix boast shared-nothing architectures, where clusters of machines in-

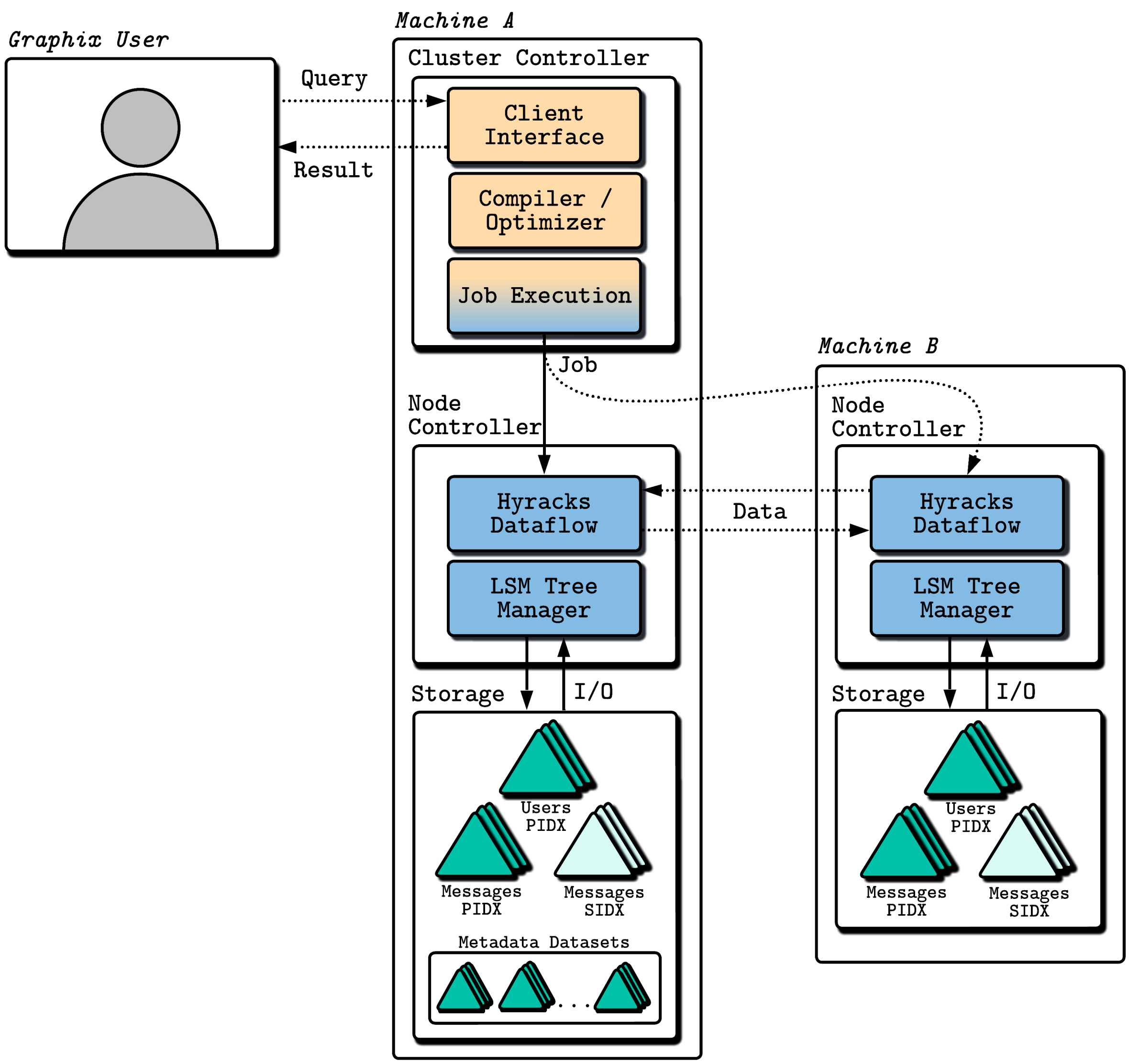


Figure 6.2: Architecture for the two-node Graphix example cluster that supports the social network database.

dependently manage their own storage and compute while relaying data across a network. In the Figure 6.2 architecture, we have two machines: machine A and machine B. Machine A hosts two processes: (1) the cluster controller process, and (2) a node controller process. The former is responsible for managing user requests and distributing work amongst the cluster, while the latter is responsible for executing work given by the cluster controller. In contrast, machine B only hosts one process: a node controller process that accepts work from the cluster controller process on machine A. To scale outward in this architecture, we would spawn more node controller processes on different machines.

With respect to storage, the node controllers on both machine A and machine B work with their own partition of the data. Machine A additionally hosts all of the Graphix and AsterixDB metadata (e.g., the schema of the datasets, the indexes built for each dataset, the graphs we have defined, etc...). The data partitioning assigned to each machine is determined using a hash of each dataset's primary key (the `id` field for the `Users` dataset, and the `id` field for the `Messages` dataset). Physically, each dataset partition is represented as an LSM-based B$^+$ tree. For this chapter, we will also consider a secondary index `messagesUserIdx` on the `user_id` field of the `Messages` dataset. Secondary indexes in AsterixDB are local to each machine, e.g., the `messagesUserIdx` on machine A will only contain index entries that point to `Message` records contained on machine A. Consequently, secondary index searches can be performed in parallel without data transfer between machines. Physically, secondary indexes are also represented as LSM-based B$^+$ trees.

6.1.1 `CREATE GRAPH` Lifecycle

To better detail the Graphix + AsterixDB architecture, we will first describe the lifecycle of a `CREATE GRAPH` statement from Section 4.2. Given a `CREATE GRAPH` request from a user to the cluster controller on machine A, the following actions are taken:

1. The `CREATE GRAPH` is lexed and parsed into an AST (abstract syntax tree) $T(\text{CG})$.
2. To enforce ACID across all metadata datasets, a metadata transaction is initialized.
3. $T(\text{CG})$ is analyzed to determine what metadata entities (e.g., datasets, dataverses, views, etc...) the `CREATE GRAPH` depends on.
4. All dependencies of $T(\text{CG})$ are then serialized and stored into a metadata dataset (`GraphDependency`) on the metadata node (i.e., machine A for our example). In both Graphix and AsterixDB, all database metadata (e.g., dataset schema, dataset indexes, view definitions, etc...) are persisted on a single designated metadata node.
5. For all `VertexConstructor` productions of $T(\text{CG})$, the following information is extracted into an array of objects (`Vertices`):

   (a) the vertex label
   (b) the primary key fields
   (c) the raw vertex body text

   For all `EdgeConstructor` productions of $T(\text{CG})$, the following information is extracted into an array of objects (`Edges`):

   (a) the label of the source vertex
   (b) the label of the destination vertex
   (c) the label of the edge
   (d) the source key fields
   (e) the destination key fields
   (f) the raw edge body text

6a. If the labels associated with `Vertices` and `Edges` are *not* valid (i.e., the vertex labels referenced by an edge are not defined), then the metadata transaction is aborted and machine A responds to the user with an error message.

6b. If the labels associated with `Vertices` and `Edges` are valid, then `Vertices` and `Edges` are serialized with the graph name and stored in a metadata dataset (`Graph`) on the metadata node. The metadata transaction is finalized, and machine A responds to the user indicating success.

After a `CREATE GRAPH` has been issued, references to the created graph will search the metadata to resolve the graph structure. Once a `DROP GRAPH` statement is issued, the corresponding entries in the `Graph` and `GraphDependency` metadata datasets are deleted (making the dropped graph no longer resolvable). Similar to other metadata entities with dependencies, if a user attempts to drop a graph that some other metadata entity depends on (or vice-versa), then Graphix will raise an error.

6.1.2 gSQL^{++} Query Lifecycle

Given a single gSQL^{++} query Q from a user to the cluster controller on machine A, the following steps and transformations are taken to execute Q in a partitioned-parallel fashion:

1. The query Q is first lexed and parsed into an abstract syntax tree $T^0(Q)$. Given that gSQL^{++} is an extension of SQL^{++}, this abstract syntax tree (AST) uses a combination of gSQL^{++} specific nodes and SQL^{++} nodes.
2. Using the `CREATE GRAPH` definition associated with the graph of $T^0(Q)$ (named in the `FROM` clause after the `GRAPH` keyword), unlabeled vertex and edge patterns are assigned labels that logically adhere to the mapping of the aforementioned `CREATE GRAPH`.
3. All of the gSQL^{++} AST nodes in $T^0(Q)$ are translated into SQL^{++} compatible AST nodes. We denote this resulting AST as $T^1(Q)$.
4. $T^1(Q)$ is transformed again through a set of AsterixDB SQL^{++} AST rewrites (e.g., `WITH` clause inlining, `GROUPING SETS`, etc. . .). For historical reasons, these AST rewrites are separate from AsterixDB's algebraic-level rewrites. We denote the final AST as $T^2(Q)$.
5. $T^2(Q)$ is then translated into an initial Algebricks query plan $P^0(Q)$. $P^0(Q)$ then undergoes a set of Graphix and AsterixDB heuristic-based logical rewrites to produce an optimized logical plan $P^1(Q)$.

6. The optimized logical plan $P^1(Q)$ then undergoes a set of Graphix and AsterixDB *physical* rewrites to produce an optimized physical plan $P^2(Q)$. $P^2(Q)$ differs from $P^1(Q)$ in that each operator in $P^2(Q)$ now has an associated physical implementation (e.g., a `JOIN` operator could be physically realized with a nested-loop algorithm, a hash-based algorithm, etc…) associated with it.
7. $P^2(Q)$ is then transformed into a Hyracks job $J(Q)$. $J(Q)$ is then expanded into a more detail graph of *activities* (e.g., a hash `JOIN` has two activities: one to build the hash table and one to probe). The activity graph of $J(Q)$ is logically divided along each blocking edge (e.g., the build phase of a hash `JOIN` must execute before the probe phase) to build another graph of *activity clusters*. This activity cluster graph is then used to define groups of activity clusters that can be run in parallel while respecting the blocking (sequencing) requirements of $J(Q)$. These groups are known as *stages*.
8. Iterating through each stage, the cluster controller process then distributes the stage instance to all node controller processes, which execute the same computation but on different partitions of the data. Once each stage has been iterated over and executed, a result is assembled and then handed back to the user by machine A.

Steps (1) to (3) are unique to Graphix, where Graphix acts (somewhat) on top of AsterixDB. Step (4) is shared by both AsterixDB and Graphix. Steps (5) to (6) are shared by both Graphix and AsterixDB, but Graphix has an additional set of rewrite rules to handle looping constructs. Steps (7) to (8) are largely decoupled from the data model of Graphix and AsterixDB, hence they are also shared by both Graphix and AsterixDB. The implementation effort behind Graphix contributes back to AsterixDB by offering Hyracks operators that can realize navigational queries, potentially enabling any future work that also requires recursion.

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

Listing 6.1: gSQL^{++} query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2`.

6.2 Hyracks Runtime Engine

Having described how gSQL^{++} queries are processed in Graphix at a high level, we will now describe how Graphix executes directed graphs of operators with cycles using a platform (Hyracks) that was purposed for executing directed *acyclic* graphs of operators. For the remainder of this chapter, we will focus on the execution of the following query (or some variation of this query) using the two-machine architecture in Figure 6.2:

Running Query Example

Given a starting user's ID (`$uid`), find all messages they have written `m1` plus all reply *chains* `r` from `m1` to messages `m2`.

We express this in gSQL^{++} with the query in Listing 6.1. This section will build upon bounded variants of Listing 6.1, detailing the intricacies of Hyracks as we a) define the additional problems that arise due to cyclic graphs in Hyracks (i.e., *liveness*, *safety*, and *mortality*), b) work towards a solution remedying these problems for a single machine Graphix cluster, and ultimately c) construct a solution that remedies these same problems in a distributed setting. We will also describe a few "paths not traveled", explaining a handful of other potential solutions we did not explore. Last but not least, in addition to the operators

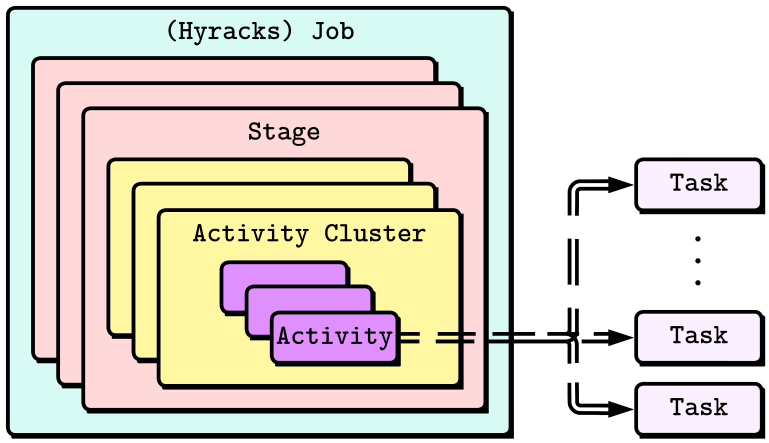


Figure 6.3: Illustration of different units of work in a Hyracks job.

required to realize recursion, we will detail two additional operators that the query optimizer, Algebricks, can leverage to optimize navigational query plans.

6.2.1 Hyracks by Example

Hyracks is the runtime engine used by AsterixDB, 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 specify 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`). Each activity later becomes instantiated as several identical tasks that are distributed to different partitions to then realize the associated Hyracks job.† Figure 6.3 illustrates the different units of work in Hyracks. A Hyracks operator may also specify blocking requirements on their activities (e.g., a hash `JOIN` operator must run the activity to build its hash table before

†AsterixDB uses Hyracks to distribute an identical set of activity instances to *all* partitions, but Hyracks itself has the potential for different distribution strategies.

running 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*. All activity developers are consequently tasked with implementing a set of methods that operate on frames. The primary carriers of data are *records*, which are contained within frames. Activities and connectors are typically written in a way that maximizes the number of records in a frame before being sent to downstream consumers, although this is not a Hyracks requirement. Hyracks was designed to be data model agnostic, thus the contents of a frame is not inherent to Hyracks. As long as all Hyracks activities and connectors in a job agree to some frame format, Hyracks will move data through a job appropriately. As we will see later, this flexibility allows us implement a rich set of features for inter-operator communication.

Single Hop Example ....

We will now describe Hyracks by example. We begin with the non-recursive query in Listing 6.2. We can answer this query using the activity graph given in Figure 6.4. For all activity diagrams in this chapter, we use the notation detailed in Table 6.1. Starting from the `PIDX SEARCH` activity on the bottom left, a search is performed for a user record (`u`) whose primary key (`id`) is equal to `$uid`. The next step involves looking for this user's messages by searching the secondary index on the `user_id` of `Messages`: `messagesUserIdx`. Because secondary indexes are local to each machine and messages are hash partitioned on `message_id`, the outbound connector of the `PIDX SEARCH` needs to broadcast the user record that `PIDX SEARCH` found to the corresponding `SIDX SEARCH` activity on *all* partitions. A secondary index entry in AsterixDB contains the primary key of the corresponding record, so to retrieve the complete `Messages` records (`m1`), a primary index search is performed. To minimize the number of index lookups, all matching primary key values (`message_id` values)

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

Listing 6.2: gSQL++ query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2` *where the length of* `r` *is equal to 1.*

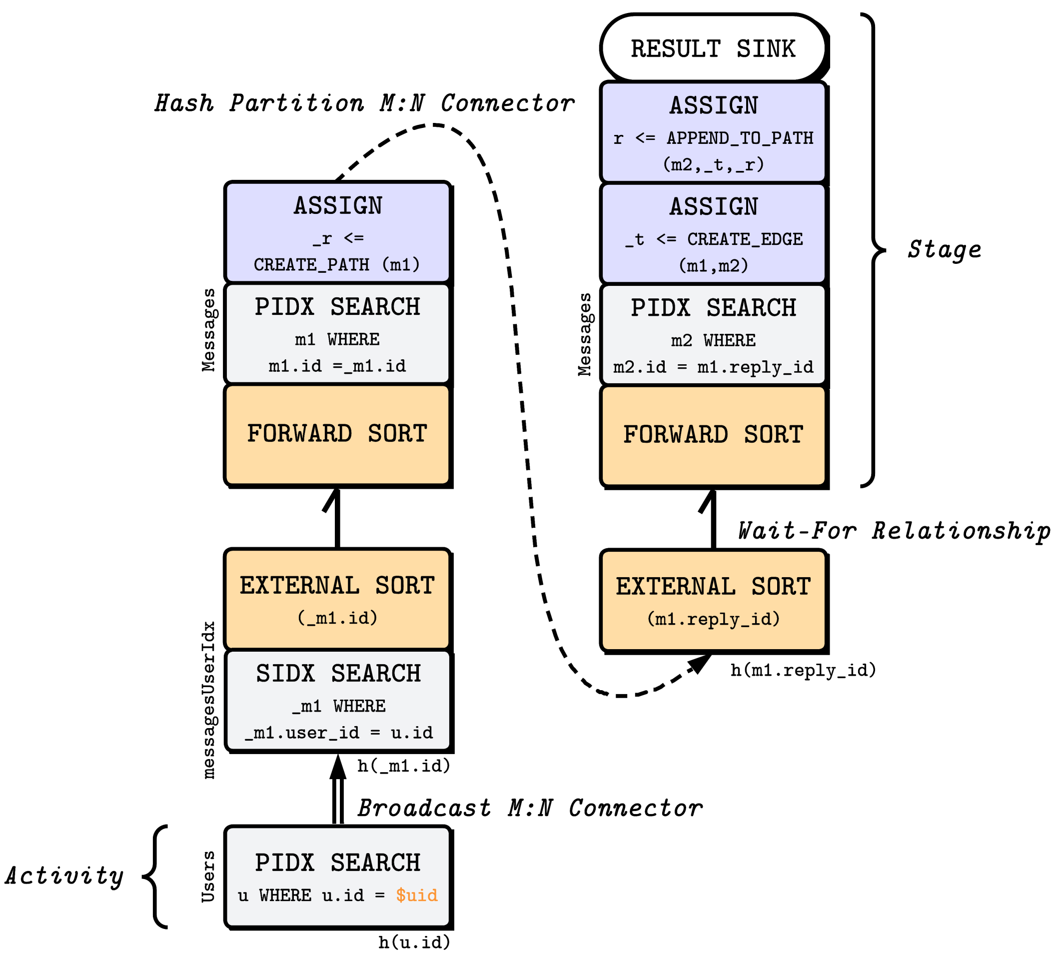


Figure 6.4: Hyracks activity graph to realize the 1-hop query in Listing 6.2.

Symbol	Symbol Name	Concept	Description
▭	Rectangle	Activity	Denotes a Hyracks activity. Rectangle colors denote different Hyracks operators (e.g., the two activities of a `SORT` operator will be the same orange color).
▭$^+$	Stacked Rectangles	Activity Group	Denotes a Hyracks activity group connected with 1:1 Hyracks connectors. Data flows from bottom to top in a pipelined manner.
$h(\ldots)$	Text Under Rectangles	Activity Partitioning	Denotes the partitioning of an activity group.
$\rightarrow$	Black Arrow	1:1 Connector	Denotes an explicit 1:1 Hyracks connector.
$\dashrightarrow$	Dashed Black Arrow	M:N Hash Partition Connector	Denotes a M:N hash-partition Hyracks connector. The $h(\ldots)$ underneath the destination activity denotes the partitioning key.
$\Rightarrow$	Double Black Arrow	M:N Broadcast Connector	Denotes a M:N broadcast Hyracks connector.
$\rightharpoonup$	Harpoon Black Arrow	Wait-For Activity Relationship	Denotes a blocking edge (i.e., a wait-for relationship) between two activities. *Does not denote data (record) transfer.*

Table 6.1: Table summarizing the notation used for all activity diagrams in this chapter.

undergo a sort before performing the actual index search [27]. External sorting in Hyracks involves two activities: (1) consuming all of the upstream input and performing a textbook external sort, followed by (2) merging and forwarding the sorted results downstream. Activity (1) must finish before activity (2), and thus the activity diagram includes a blocking edge (denoted by the harpoon arrow). Once the `Messages PIDX SEARCH` activity has been performed, Hyracks has evaluated the `(u:User)-[:WROTE]->(m1:Message)` graph pattern.

To find a path of length 1, a zero-length path (assigned the variable `_r`) is built. This zero-length path accumulates all traversed vertices and edges. As denoted by `CREATE_PATH(m1)`, `_r` includes just the starting vertex record. To evaluate a single `REPLY_OF` hop is to logically perform a self-`JOIN` with `Messages` (assigned the variable `m2`) using `m1.reply_id` and `m2.id`. This self-`JOIN` is realized by performing another `PIDX SEARCH` with `m1.reply_id` as the search key. Knowing that `Messages` is hash-partitioned on its primary key (`id`), the tuples containing ⟨`u,m1,_r`⟩ are forwarded to the appropriate machine using a) the same hash function h used to partition the `Messages` dataset, and b) the `Messages` search key: `m1.reply_id`. Again, the number of index lookups are minimized by performing a local external sort before performing the actual `PIDX SEARCH`. After performing this primary index search, the record `_t` is built to capture the traversal of the `REPLY_OF` edge. To conclude the evaluation of the `(m1)-[r:REPLY_OF{1,1}]->(m2:Message)` pattern, the previous vertex (`m2`) and the edge (`_t`) used to traverse to the previous vertex are appended to the path object (`_r`) to build a new path object (assigned the variable `r` bound to `APPEND_TO_PATH(m2, _t, _r)`). ⟨`u,m1,r,m2`⟩ is then sent back to the cluster controller process to give back to the Graphix user (shown by `RESULT SINK`).

Figure 6.5 visualizes the data transfer between machines in a Graphix cluster for the activity graph in Figure 6.4. Aside from the final result distribution, data is exchanged at two points: once at the broadcast connector originating from the `Users PIDX SEARCH` activity, and again at the hash-partition connector originating from the `CREATE_PATH(m1)` activity. All other computation (i.e., the `SORT`, the `PIDX SEARCH`, and the `ASSIGN`s) is performed locally at each partition. We conclude this section by noting that this approach meets our first design objective for navigational queries:

Design Objective 1

To realize Graphix, navigational queries should be performed in a partitioned-parallel manner.

A navigational query in Graphix should (ideally) execute faster given more machines by being able to leverage not just additional CPUs, but also increased aggregate memory (i.e., when more of the total graph fits into memory) and increased disk throughput (i.e., when the total graph can be loaded into memory faster).

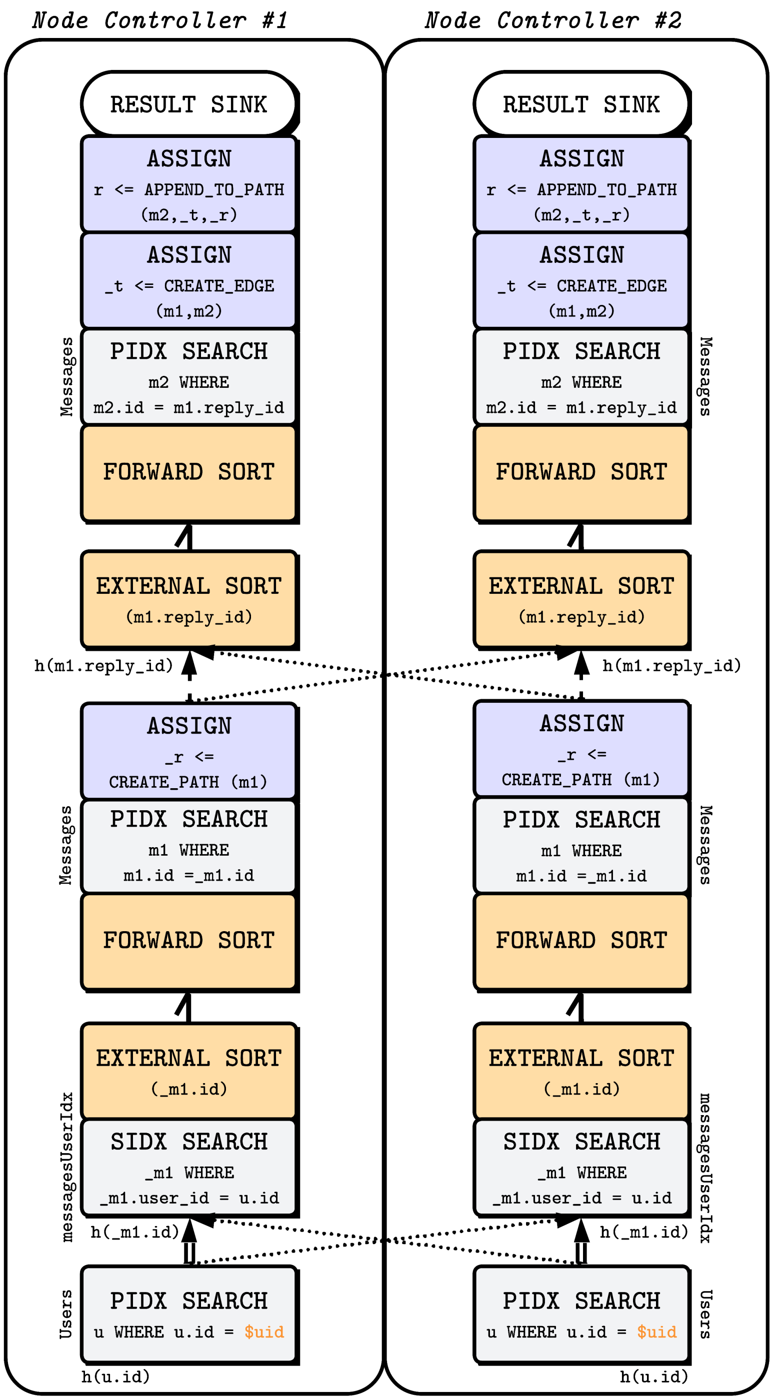


Figure 6.5: Instance of the activity graph in Figure 6.4 to realize the 1-hop query in Listing 6.2.

Three Hop Example . . . .

Having described how Hyracks can be used to evaluate a single-hop gSQL^{++} query, we will now discuss how we can extend the previous section to handle m-hop gSQL^{++} queries where m is constant (i.e., we explicitly specify the number of hops to traverse). Consider the query in Listing 6.3, where we are now interested in finding the reply chains `r` of length 3. Figure 6.6 describes an activity graph to realize our query. Aside from a few variable name changes (`r` $\leftrightarrow$ `_r0`, `_h1` $\leftrightarrow$ `m2`, `_t1` $\leftrightarrow$ `_t`, and `_r` $\leftrightarrow$ `_r1`) , the left half of the diagram remains unchanged from Figure 6.4: to find a path `r` of $m = 3$ hops, a path with a single hop ($m = 1$) must first be found. This single hop path in Figure 6.6 is assigned the variable `_r1`. From `_r1`, the same activities used to perform to first hop are used to execute the second hop and third hops. To evaluate the second hop, Hyracks must a) hash partition on the `PIDX SEARCH` key (`_h1.reply_id`), b) sort all tuples by the `PIDX SEARCH` key, c) perform the `PIDX SEARCH` to traverse to the next vertex `_h2`, d) assemble the edge record `_t2`, and finally e) *append* to the existing path to form a path of two edges and three vertices (`_r2`). The third hop repeats the exact same process, and concludes with Hyracks sending the tuple $\langle$`u, m1, r, m2`$\rangle$ to the `RESULT SINK`.

We finish this section by noting that this approach meets a second design objective for navigational queries:

> **Design Objective 2**
>
> To realize Graphix, navigation *should not* require any special auxiliary structures to be built beforehand.

To determine if a path exists between two vertices, Graphix simply perform many depth-first searches using a series of `JOIN`s (either realized as a sequence of index searches like our current example, or using hash-`JOIN`s). As a reminder, Graphix targets a range of queries that access a few vertices (i.e., highly interactive queries) to a larger fraction (i.e., analytical queries)

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

Listing 6.3: gSQL^{++} query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2` *where the length of* `r` *is equal to 3.*

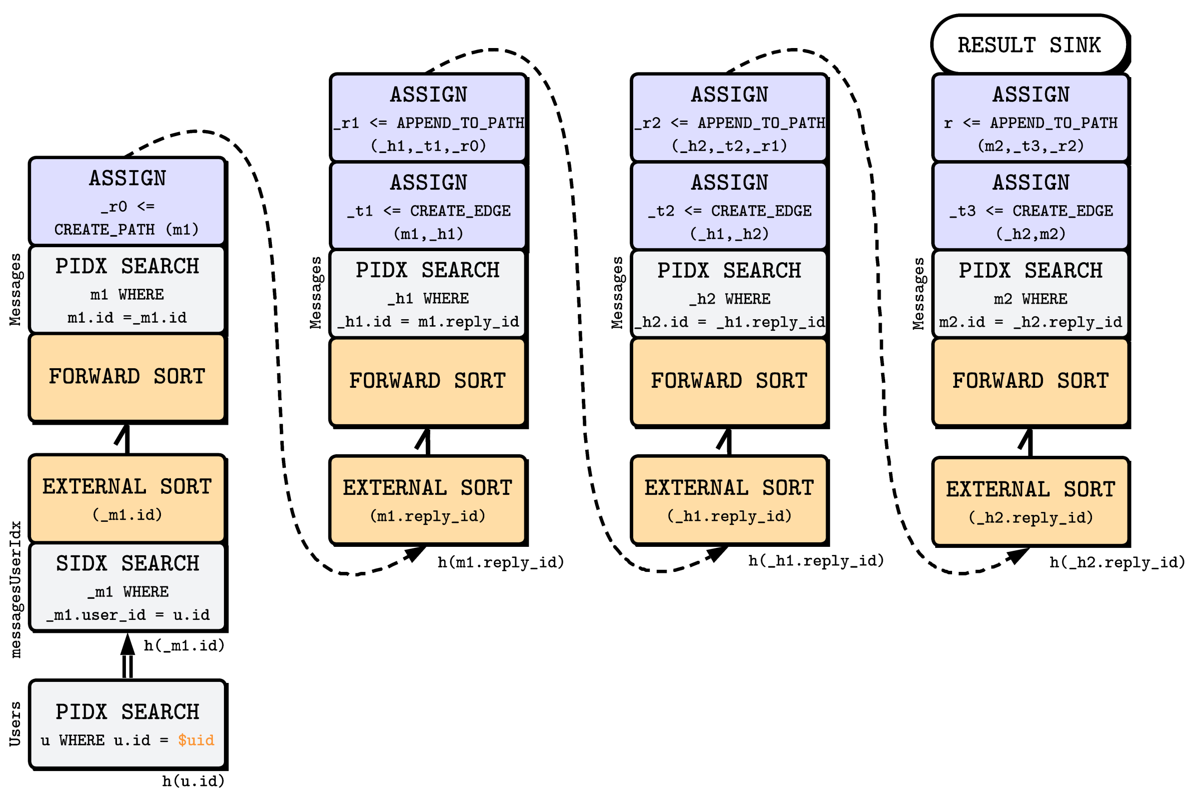


Figure 6.6: Hyracks activity graph to realize the 3-hop query in Listing 6.3.

of the entire graph. Graphix does not target queries that iterate over the *entire* graph, as such use cases are more suited toward graph processing systems like Pregelix. Immediately after a Graphix user issues a `CREATE GRAPH` (or specifies one in the `WITH` clause), they should be able to query that graph. To determine if a path exists between two vertices, Graphix should not have to scan all vertices as a preprocessing step (e.g., spending $|V|^3$ time in the case of Floyd-Warshall).

One-to-Three Hop Example ....

The previous two examples dealt with paths of fixed-length hops. In Listing 6.4, we now consider a variable-length path `r` of 1 to 3 hops. To realize our query, the activity graph in the previous section is extended to yield the intermediate paths (illustrated in Figure 6.7). More specifically, after the first hop (the activity generating `_r1`) and the second hop (the activity generating `_r2`), a `REPLICATE` activity followed by a `MATERIALIZE` activity is added. The `REPLICATE` activity duplicates its output (one connector to the subsequent `MATERIALIZE` activity, and another to the `EXTERNAL SORT` activity), while the `MATERIALIZE` activity is responsible for writing all computed records to disk for use in the last stage. The last stage includes the generation of the third hop (the activity generating `r`) as well as a sequence of `UNION ALL` activities to feed results to the `RESULT SINK`. We note that Hyracks stages contain *mutually exclusive* sets of activities. Algebricks chooses to generate a plan with a single `RESULT SINK`: even though the tuples after the first hop are not manipulated before reaching the `RESULT SINK`, using a single `RESULT SINK` means that the intermediate results must be materialized before being used in the final stage.

Figure 6.8 describes an alternative activity graph that no longer sorts before performing searches on the primary index in order to avoid the blocking needed for run generation followed by merging. If we can ensure that the activities used to calculate the first, second, and third hops are all contained within a single stage, then Hyracks does not need to needlessly

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

Listing 6.4: gSQL^{++} query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2` *where the length of `r` is between 1 and 3.*

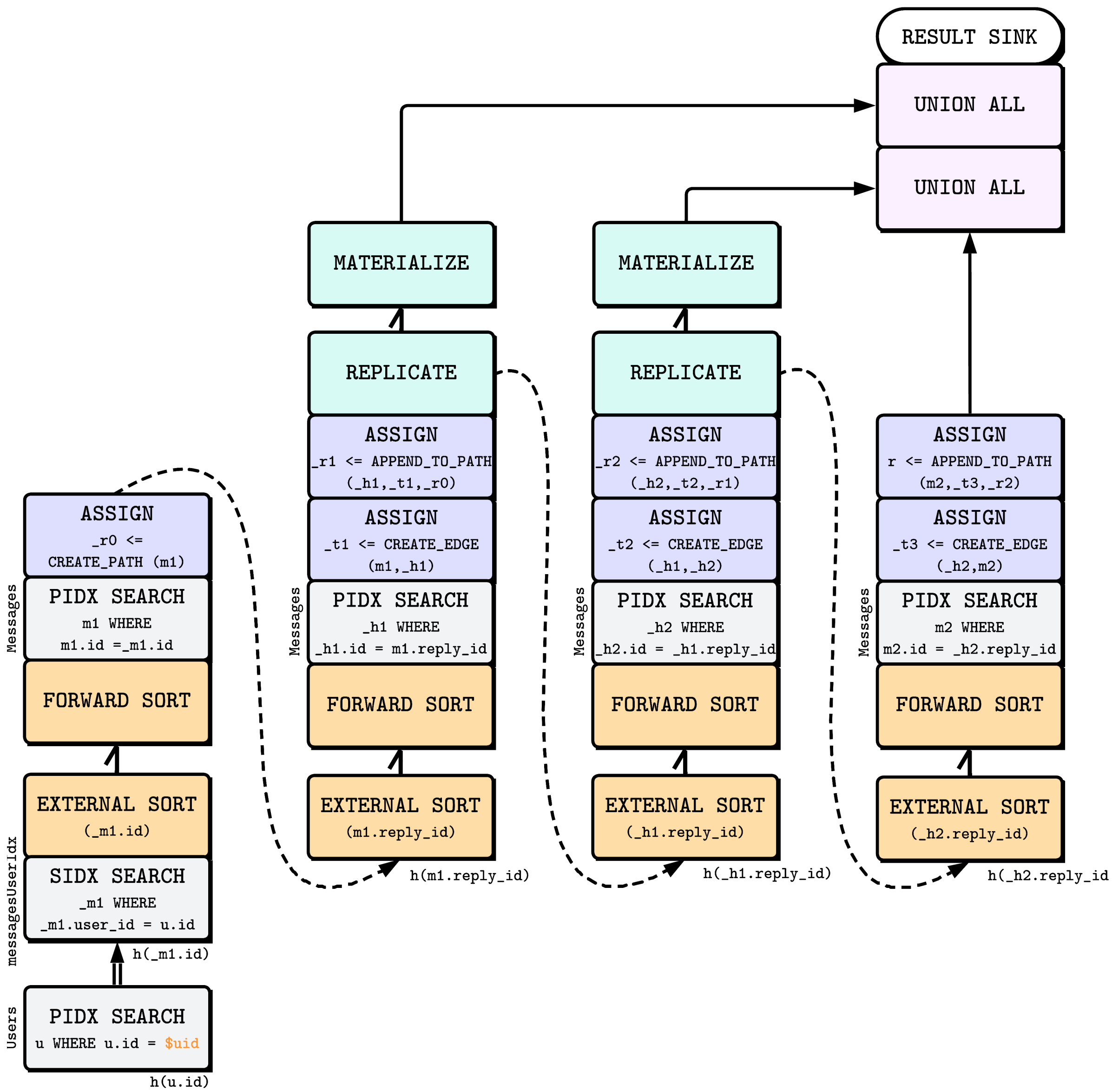


Figure 6.7: Hyracks activity graph to realize the 1-to-3-hop query in Listing 6.4.

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

Duplicate of Listing 6.4: gSQL++ query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2` *where the length of* `r` *is between 1 and 3.*

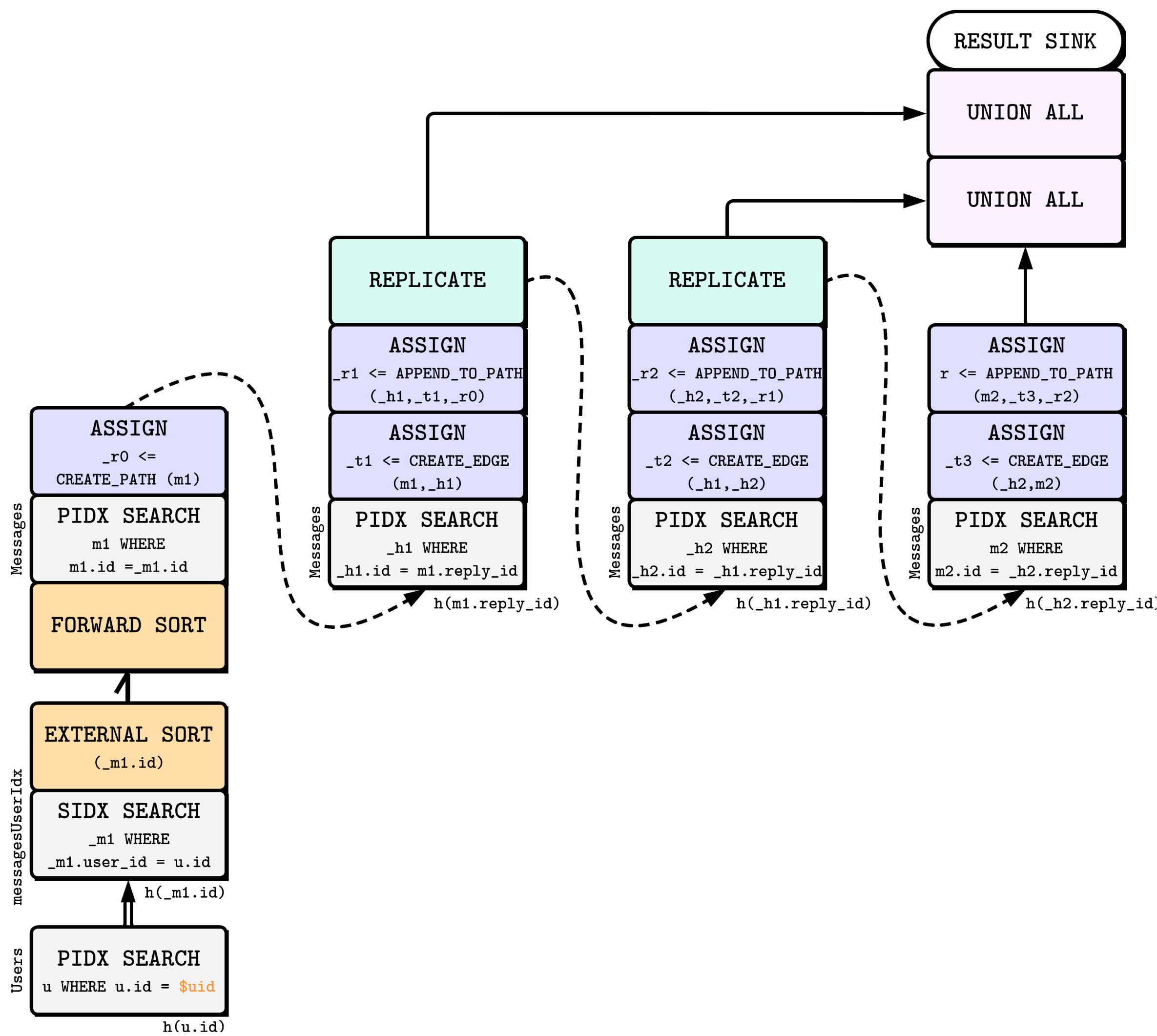


Figure 6.8: Hyracks activity graph to realize the 1-to-3-hop query in Listing 6.4 without sorting.

materialize. Figure 6.8 demonstrates an approach that meets our third design objective for navigational queries:

Design Objective 3

To realize Graphix, our implementation *should not* block globally to evaluate a single hop at a time, as each path grows *independently* per hop.

Leveraging their independence property, Graphix should be able to evaluate paths in a more pipelined manner when compared to traditional graph processing system methods (*synchronous* evaluation is common). The activities used to evaluate a hop should not contain any blocking edges, ultimately forbidding plans involving operations like aggregation, sorting, and materialization. While these restrictions ultimately limit the expressiveness of the computations that can be performed, we would like to remind the reader that we are using Hyracks as a compilation target for gSQL^{++} queries. Recursion in gSQL^{++} is only expressed in the form of navigation, so such plans will not be generated. We point to GiraphUC [30] and Mitos [29] for other systems that also leverage similar independence properties, though for more general recursion.

To further illustrate (and emphasize) how each path grows independently of one another, consider the example in Figure 6.9. In this example, the user `u` has written one message `m1` where the reply chain to both messages has a branching factor of 2 (resulting in $2^1+2^2+2^3 = 14$ total paths). Focusing on the orange filled record after the third hop $\langle \texttt{u}, \texttt{m1}_1, \texttt{\_r3}_1, \texttt{\_h3}_1 \rangle$, we observe that the path $\texttt{\_r3}_1$ is *composed* of the previous two hops:

$$\texttt{\_r3}_1 \quad = \quad \texttt{APPEND\_TO\_PATH}\,(\texttt{\_h3}_1,\ \texttt{\_t3}_1,\ \texttt{\_r2}_1) \tag{6.1}$$

$$\texttt{\_r2}_1 \quad = \quad \texttt{APPEND\_TO\_PATH}\,(\texttt{\_h2}_1,\ \texttt{\_t2}_1,\ \texttt{\_r1}_1) \tag{6.2}$$

$$\texttt{\_r1}_1 \quad = \quad \texttt{APPEND\_TO\_PATH}\,(\texttt{\_h1}_1,\ \texttt{\_t1}_1,\ \texttt{\_r0}_1) \tag{6.3}$$

$$\texttt{\_r0}_1 \quad = \quad \texttt{CREATE\_PATH}\,(\texttt{m1}_1) \tag{6.4}$$

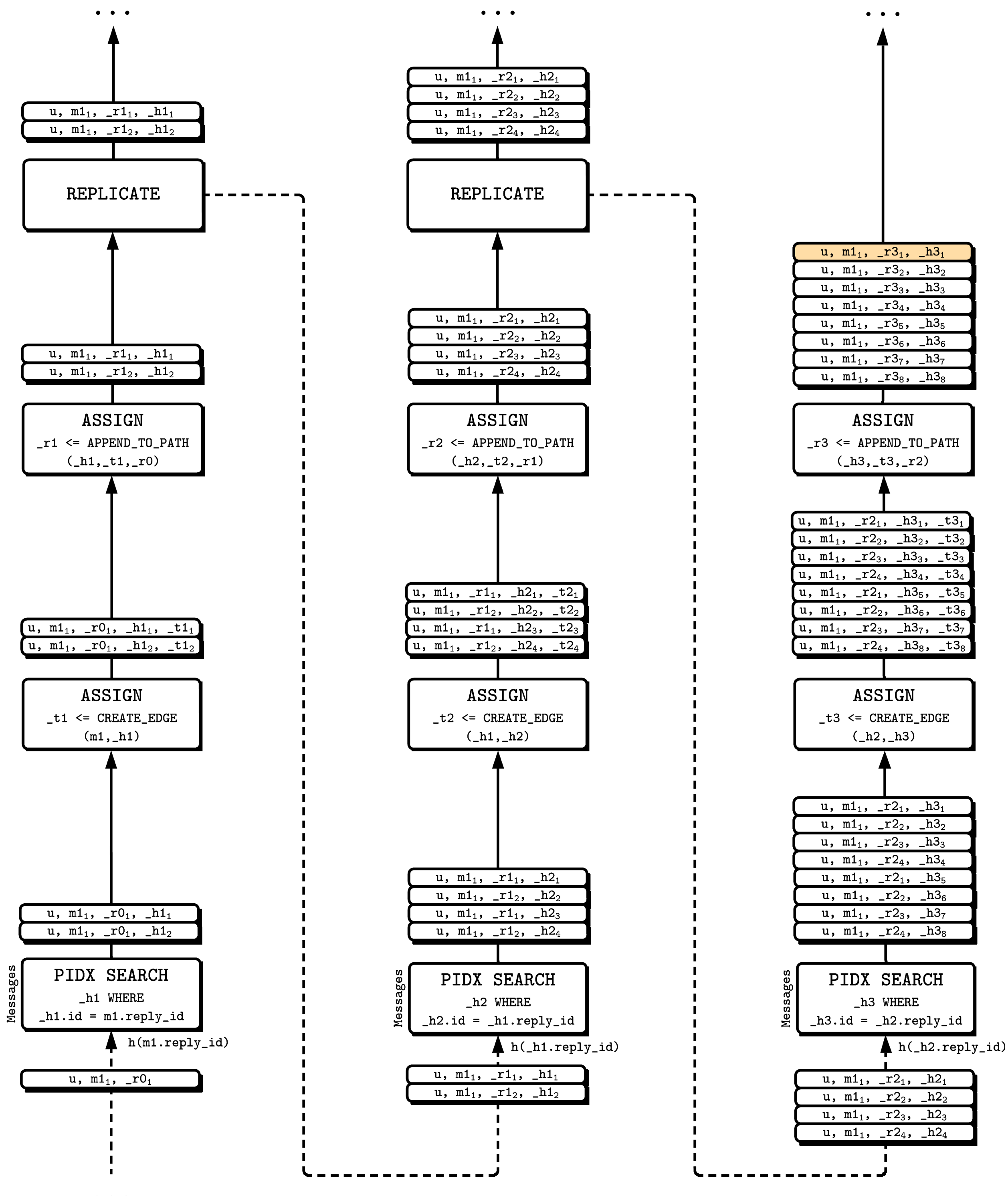


Figure 6.9: Runtime visualization of the Hyracks activity graph in Figure 6.8, demonstrating how paths grow independently of one another.

$\texttt{\_r3}$ ultimately contains all vertices and edges involved in its traversal, all of which can thus be accessed directly later downstream.[‡] As seen in Section 5.3, a path in Graphix is an object containing two array-valued fields: `Vertices` and `Edges`. The `CREATE_PATH` function returns a record with a single entry in the `Vertices` array (in Equation 6.1, $\texttt{m1}_1$) and an empty `Edges` array. The `APPEND_TO_PATH` function returns a record that *extends* the `Vertices` and `Edges` arrays of the input path record. If we factor out the use of previous paths in Equation 6.1, we get the following:

$$\texttt{\_r3}_1 = \text{ path with vertices } (\texttt{m1}_1,\ \texttt{\_h1}_1,\ \texttt{\_h2}_1,\ \texttt{\_h3}_1) \quad \text{and edges } (\texttt{\_t1}_1,\ \texttt{\_t2}_1,\ \texttt{\_t3}_1) \tag{6.5}$$

$$\texttt{\_r2}_1 = \text{ path with vertices } (\texttt{m1}_1,\ \texttt{\_h1}_1,\ \texttt{\_h2}_1) \quad \text{and edges } (\texttt{\_t1}_1,\ \texttt{\_t2}_1) \tag{6.6}$$

$$\texttt{\_r1}_1 = \text{ path with vertices } (\texttt{m1}_1,\ \texttt{\_h1}_1) \quad \text{and edges } (\texttt{\_t1}_1) \tag{6.7}$$

$$\texttt{\_r0}_1 = \text{ path with vertex } (\texttt{m1}_1) \quad \text{and zero edges} \tag{6.8}$$

6.2.2 Recursion Foundations

We will now discuss recursion for Graphix in Hyracks. As a reminder, Graphix only generates recursive Hyracks jobs that do not possess any blocking edges in-the-loop. Systems like Mitos [29] and Naiad [46] similarly enable the specification of *explicitly looping* data flows,[§] however these systems were designed for more general iterative computation (e.g., computations like PageRank and K-Means). In the context of a system like Hyracks, a recursive solution should ideally a) utilize the Hyracks computational model of operators and connectors to *explicitly* express data flow cycles, b) work with a distributed shared-nothing cluster of machines, c) respect some (aggregate) memory budget (spilling to disk when this budget is exceeded), and d) serve as a compilation target for jobs generated by Algebricks.

[‡]Graphix possesses an optimization that minimizes the vertex and edge information contained in a path object when the contents of a path are not required. See Section 6.4 for more details.

[§]We contrast explicit cyclic data flows (e.g., those generated by Graphix for use in Hyracks) with systems that issue acyclic data flows while externally managing the looping aspects. See Subsection 6.2.9 for an example of such a solution.

Consider our original query (Listing 6.1, repeated below for ease of reference) where `r` is unbounded. Recognizing the commonalities used to evaluate the first, second, and third hops, it follows that the solution involves using a `UNION ALL` and a `REPLICATE` activity to repeat the hop finding until all reply chains are found. We depict such an activity graph in Figure 6.10, where the `UNION ALL` now precedes the `PIDX SEARCH`. The `REPLICATE` activity is used to forward tuples to the `RESULT SINK` and back to the `UNION ALL` (to next perform the next hop). To handle cycles in the data, a `SELECT` activity is added before the `APPEND_TO_PATH` activity to avoid forwarding paths that repeat edges, vertices, or both (we refer back to Section 5.3 for a more formal discussion about handling cycles). All activities in the loop of Figure 6.10 are surrounded by a "lively, safe, mortal" block, which describes a hypothetical protocol implemented by each surrounded activity to solve three problems that arise due to the presence of cycles in a Hyracks graph of activities: 1) the problem of *liveness*, where no progress is being made, 2) the problem of *safety*, where activity instances deadlock due to an over-allocation of resources, and 3) the problem of *mortality*, where activity instances never terminate.

We begin our discussion with a review of the "internals" of Hyracks activities. A Hyracks activity must, at a minimum, implement the `IFrameWriter` interface.[¶] The `IFrameWriter` interface consists of the following methods, all of which are *only* called by their upstream activity.

1. `open()`, which performs any initialization (e.g., the instantiation of some objects, the allocation of memory, etc...) and subsequently calls `open()` for its own downstream activities;
2. `nextFrame(f)`, which accepts an input frame `f` and performs some computation using the contents of the frame;

[¶]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 the sake of clarity, we will consider each activity as both a producer and consumer in this section.

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

Duplicate of Listing 6.1. gSQL^{++} query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2`.

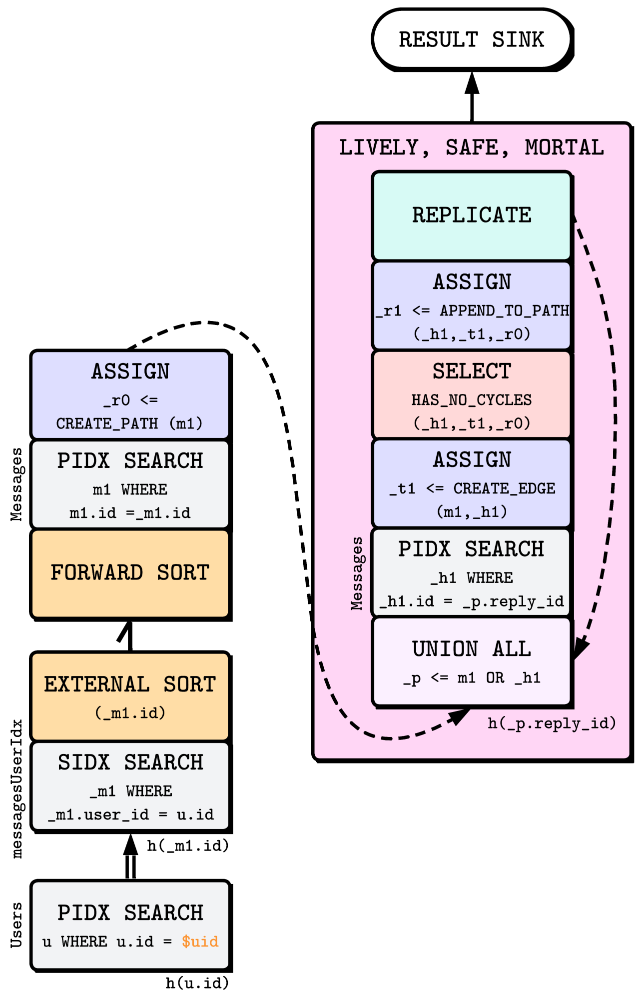


Figure 6.10: High level overview of a Hyracks activity graph to realize the query in Listing 6.1.

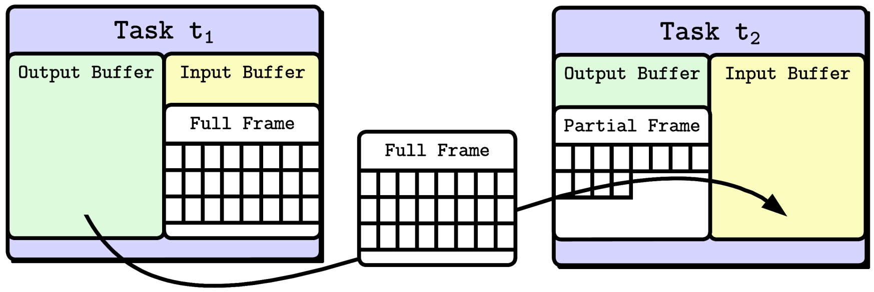


Figure 6.11: Depiction of task t_1 forwarding its output buffer to task t_2.

3. `flush()`, which eagerly forwards any partial frame data that the activity currently has buffered to its downstream activities;
4. `fail()`, which performs any required actions to fail in a "safe" manner and subsequently calls `fail()` for its own downstream activities; and
5. `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 `close()` for its own downstream activities.

The typical lifecycle of a Hyracks activity instance (i.e., a task) 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 the activity's output buffer becomes full, and finally (iii) getting its own `close()` method called and calling `close()` for its own consumers.

Now consider the two-task cluster composed of tasks t_1 and t_2. Task t_1's output is connected to task t_2's input, and task t_2's output is connected to task t_1's input. We point to Figure 6.11, which depicts an instance of t_1 directly pushing a frame to t_2 by call t_2's `nextFrame(f)` method. To reflect how most Hyracks activities implement the `IFrameWriter` interface, the activities associated with t_1 and t_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 calling its own `close()` method called). This `IFrameWriter` implementation maximizes the information held in a frame before the activity itself forwards the output buffer frame downstream, ultimately leading to better utilization of frame-transferring resources like network bandwidth. Moving back to our example in Figure 6.11, we note that task t_1 is forwarding a full frame from its output buffer to the input of t_2. Task t_2 has a partial frame, so it does not forward its output to t_1 yet.

6.2.3 Property #1: Liveness

We will now consider the *liveness* property, which describes a group of tasks that are always "making progress". In the context of navigation, liveness describes a group of tasks that will eventually generate all (satisfiable) paths. The top portion of Figure 6.12 demonstrates a violation of this liveness property: two tasks (of the same configuration as Figure 6.11) possess the potential to perform more work but will not due to their "forward when full" implementation. t_1 has a partially full output buffer that it *could* forward to t_2 but does not. Similarly, t_2 has a partially full output buffer that it *could* forward to t_1 but does not. A starting point to remedy this liveness violation involves adding the following requirements for each task within a loop: i) the ability to forward partially full output buffers, and ii) some method of invoking the former ability. We note that the former has already been implemented for all activities, as the `flush()` method (originally purposed for AsterixDB feeds), so our solution only needs to consider the latter (i.e., by invoking the `flush()` method).

At a high level, to guarantee liveness we must first implement a form of inter-task communication beyond the method calls provided by the `IFrameWriter` interface. Our solution is an in-band one (see Subsection 6.2.9 for a description of a potential alternative out-of-band solution) that leverages the `IFrameWriter` interface that each task already implements: 1) we define two classes of frames: (i) a data frame, full of tuples, and (ii) a new *message* frame

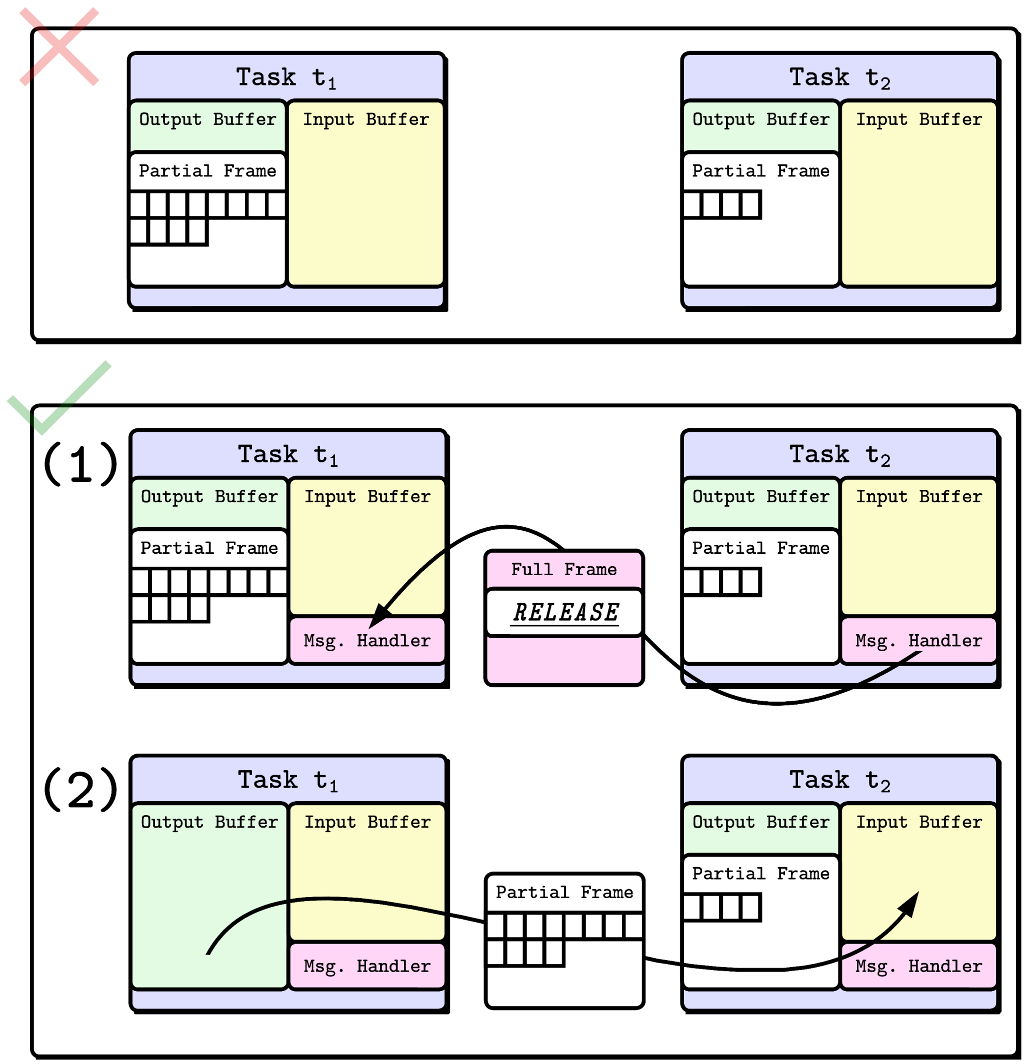


Figure 6.12: A depiction of a liveness violation (top) and a mechanism to prevent such violations (bottom).

used to pass information to other tasks downstream. 2) we non-invasively "decorate" each activity that may violate liveness (i.e., those inside of a loop) to recognize and act on these new message frames. The bottom portion of Figure 6.12 depicts our solution in action: the decorated activity instance t_2 generates and forwards a message frame containing a `RELEASE` directive to the decorated activity instance t_1 using t_1's `nextFrame(f)` method, and t_1 then calls `flush()` to forward its partial frame to the input of t_2. In order to get task t_2 to forward its partial frame, t_1 would need to send a message frame to t_2 (not depicted). Message frames can be viewed as a form of "punctuation" [71], which (in the context of stream processing) are used as signals for stream processors to release state. Note that Figure 6.12 is a partial solution: we will detail *when* and *who* generates these message frames after addressing the two other properties.

6.2.4 Property #2: Safety

Our second property of interest is the *safety* property, which (for our purposes) describes a group of tasks that will never deadlock. The top portion of Figure 6.13 demonstrates a violation of this safety property: both tasks (of the same configuration as the previous two examples) 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 Figure 6.13, task t_1 has reserved all frames within its budget and is prepared to send these frames to task t_2. t_2, however, cannot receive these frames from t_1 since t_2 has reserved all frames within its budget and is prepared to send these frames to task t_1. Neither t_1 nor t_2 is aware of the fact that their actions are causing a deadlock.

To remedy this deadlock violation, we designate (at compile time) one task within a cyclic task group to avoid acquiring "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 the ability spill to disk when full. After some point, the task associated with the designated task will then forward everything stored in its secondary buffer, repeating this buffer-and-forward process until all frames are exhausted. The bottom portion of Figure 6.13 depicts our solution in action: task t_1 is designated to store each frame sent by t_2 to its own secondary buffer. After t_2 has given all of its frames to t_1, t_1 then forwards all of the frames in its secondary buffer to task t_2. At a glance, this buffer-and-forward process may seem like Graphix is performing some global synchronization at each step of the computation. We remind the reader here of the granularity of our explanations and

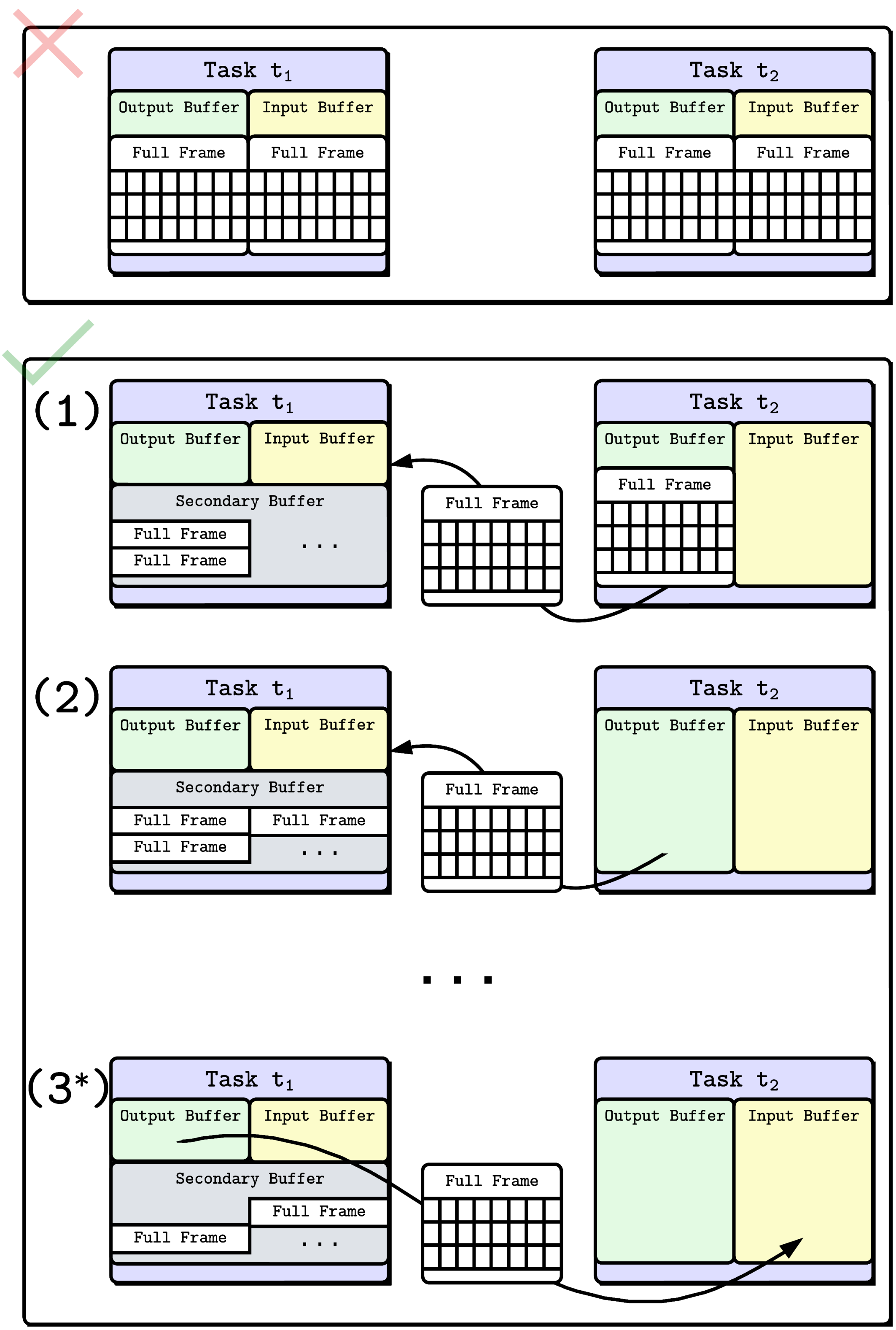


Figure 6.13: A depiction of a safety violation (top) and a mechanism to prevent such violations (bottom).

examples thus far: we have been working with *tasks* (i.e., activity instances, not activities). As we will later see, Graphix can perform this buffer-and-forward process locally *without* any need for inter-partition synchronization.

6.2.5 Property #3: Mortality

The last property we will consider is the *mortality* property, which guarantees that every task will eventually terminate (i.e., call `close()`) when there is no work left to do. For task groups with cycles, we can easily show a violation of this mortality property. The top portion of Figure 6.14 demonstrates two tasks t_1, t_2 that *could* terminate but do not. In order for task t_1 to finish, its upstream producer t_2 must call t_1's `close()` method. Conversely, task t_2 will only call the `close()` of t_1 when task t_1 calls t_2's `close()` method. Clearly, neither t_1 nor t_2 can call `close()`. This termination problem is inherent to *all* Hyracks jobs with cycles, as a task 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 a task t can only reason about the termination status of task t's immediate upstream producer (i.e., by having t's own `close()` method invoked). Our solution requires i) designating (at compile time) a task to call `close()` when there exists no tuples left to process, ii) getting all tasks within the loop to report on their status, and iii) sending the statuses of each task from (the previous item, ii) to the designated "`close()`-er" task. Graphix realizes all of the former requirements by extending the use of message frames in the liveness section (Subsection 6.2.3). The bottom portion of Figure 6.14 demonstrates a high level overview of this extension for our example two-task cluster. The first step starts with task t_2 giving the `RELEASE` directive along with an *uncolored marker*

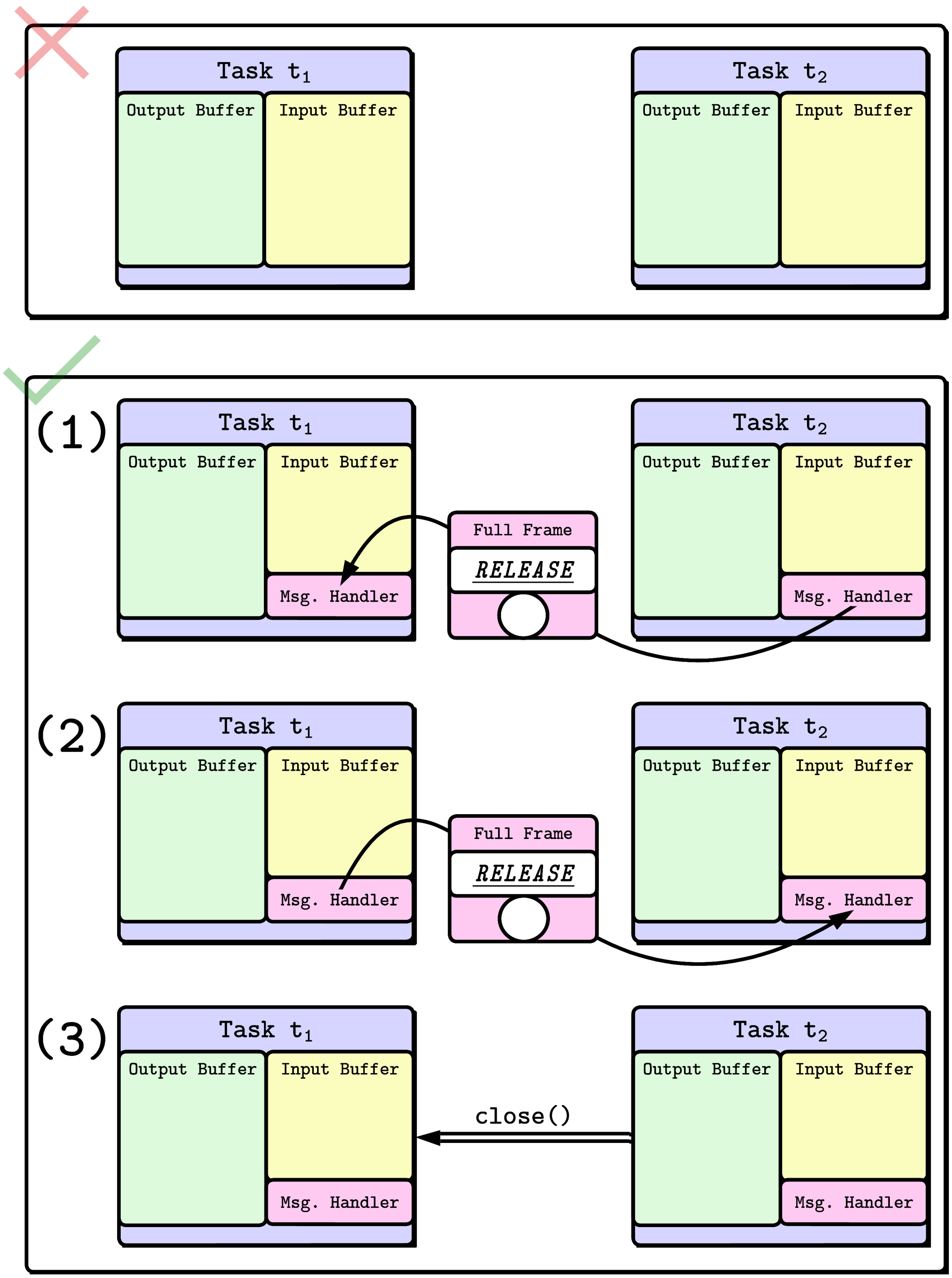


Figure 6.14: A depiction of a mortality violation (top) and a mechanism to prevent such violations (bottom).

inside a message frame to task t_1 via t_1's `nextFrame(f)` method. If task t_1 has tuples left to process, it will color in the marker. Our example shows t_1 pushing an uncolored marker frame downstream (back to t_2), denoting that t_1 has no tuples left to process. Task t_2 sees that t_1 has left the marker uncolored and therefore it calls the `close()` method of t_1. t_1 will subsequently call the `close()` method of t_2 (not depicted), terminating the computation. Our use of message frames was inspired by the use of punctuation in the FFP (flying fixed point) operator [15] for the problem of cyclic stream processing.

6.2.6 Fixed Point Operator (1-Machine)

Returning to our original query (Listing 6.1), Figure 6.15 defines an activity graph that satisfies our liveness, safety, and mortality properties. As described in previous three sections (Subsection 6.2.3, Subsection 6.2.4, and Subsection 6.2.5), the solution to each problem caused by cycles in the task graph involved (at a minimum) elevating the responsibility of some designated task. Starting at the bottom right, we define a new activity group containing a `FIXED POINT` operator that essentially elevates the responsibility of the `UNION ALL` of Figure 6.10. Instances of the `FIXED POINT` operator are responsible for: (i) generating message frames with the `RELEASE` directive and an uncolored marker to forward to their immediate downstream task (e.g., the `PIDX SEARCH`); (ii) buffering incoming frames from the `REPLICATE` task to maintain safety; and (iii) determining whether or not it is appropriate to call `close()`. Each task in the loop ($t \in T_{\texttt{LOOP}}$, where $T_{\texttt{LOOP}}$ represents a cyclic task group) have been decorated with "`MESSAGE AWARE`". A decorated activity acts as a proxy for the `open()`, `close()`, `flush()`, and `fail()` methods of t, but alters the functionality of `nextFrame(f)` upon receiving a message frame to: (a) call t's `flush()` method; (b) color the marker of the message frame if the previous `flush()` call sent any tuples downstream; and (c) forward the potentially modified message frame to its downstream consumer. We depict this task decoration in Figure 6.16. To localize the use of message frames (thus avoiding

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

Duplicate of Listing 6.1. gSQL++ query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2`.

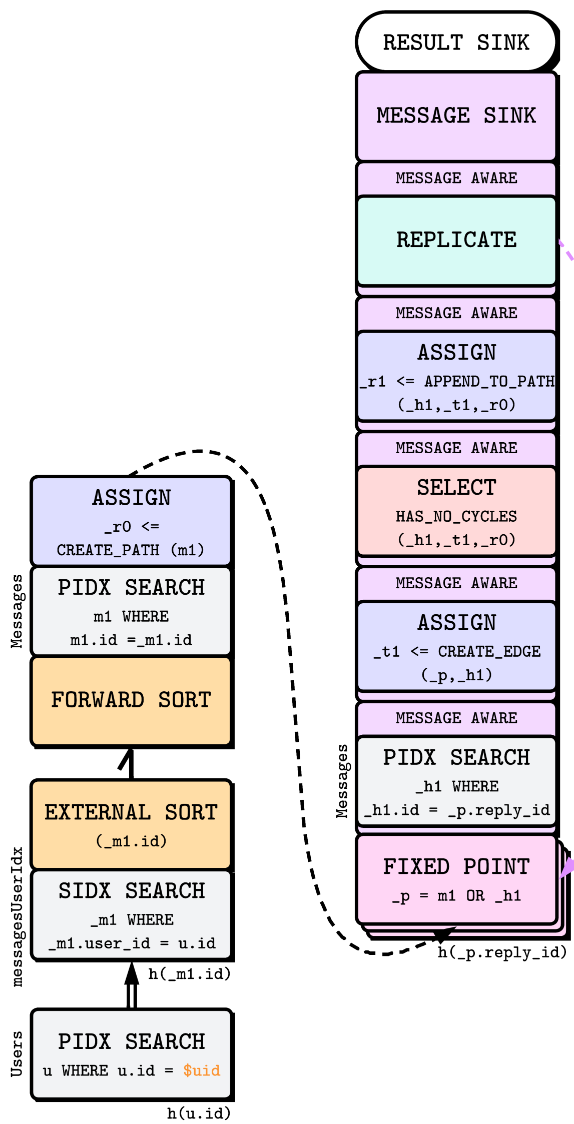


Figure 6.15: Hyracks activity graph to realize the query in Listing 6.1 that is *live*, *safe*, and *mortal*.

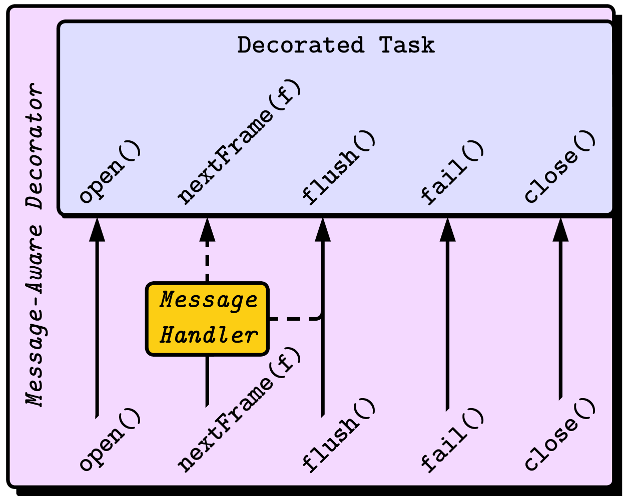


Figure 6.16: Illustration of decorating an activity instance (i.e., task) to non-invasively add message-handling functionality.

having to decorate *all* downstream activities in a plan with `MESSAGE AWARE`), a `MESSAGE SINK` activity is used to only forward data frames downstream.

Note that the "modify-and-forward" action that each decorated task performs here for message frames allows the `FIXED POINT` to reason about the status of all tasks in the loop after generating the message frames. For an instance of the activity graph in Figure 6.15, the following actions allow the `FIXED POINT` instance to reason about the status of its looping task group:

1. the `FIXED POINT` instance pushes the message frame (denoted as f_m) to the decorated `PIDX SEARCH` task (via the `PIDX SEARCH` task's `nextFrame(f)` method);
2. the decorated `PIDX SEARCH` task calls its own `flush()` method, colors the marker in f_m if `flush()` forwarded any tuples, and forwards f_m to its downstream decorated `ASSIGN` task (via the `ASSIGN` task's `nextFrame(f)` method);
3. the decorated `ASSIGN` task calls its own `flush()` method, colors the marker in f_m if `flush()` forwarded any tuples, and forwards f_m to the decorated `SELECT` task;
4. the decorated `SELECT` task calls its own `flush()` method, colors the marker in f_m if `flush()` forwarded any tuples, and forwards f_m to the decorated `ASSIGN` task;

5. the decorated `ASSIGN` task calls its own `flush()` method, colors the marker in f_m if `flush()` forwarded any tuples, and forwards f_m to the decorated `REPLICATE` task;
6. the decorated `REPLICATE` task calls its own `flush()` method, colors the marker in f_m if `flush()` forwarded any tuples, and forwards f_m to its downstream to both the `MESSAGE SINK` task (which will ultimately drop the marker frame to prevent it from reaching the `RESULT SINK`) and the `FIXED POINT`.

If `FIXED POINT` receives a message frame containing a marker that has been colored, then `FIXED POINT` knows that at least one task within the loop has generated more tuples (thus, it would be erroneous to call `close()`). In the case of 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 `FIXED POINT` can conclude that the task loop has no tuples left to process *if and only if there are no other task groups* (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.

6.2.7 Fixed Point Operator (n-Machines)

Our previous section detailed the principles for navigational queries on a single partition. One of the desideratum for Graphix, however, was to execute graph queries in a partitioned-parallel manner. In this section, we will discuss how to apply the previously described principles in a distributed setting.

We begin our discussion by reviewing the granularity of a Hyracks stage (i.e., an activity cluster): a Hyracks stage executes on a cluster of machines, where each machine executes (in parallel) computation (i.e., a task cluster) expressed using smaller predefined computation blocks (i.e., a task). Consider two activities A_A and A_B, and group of activities `FP` that

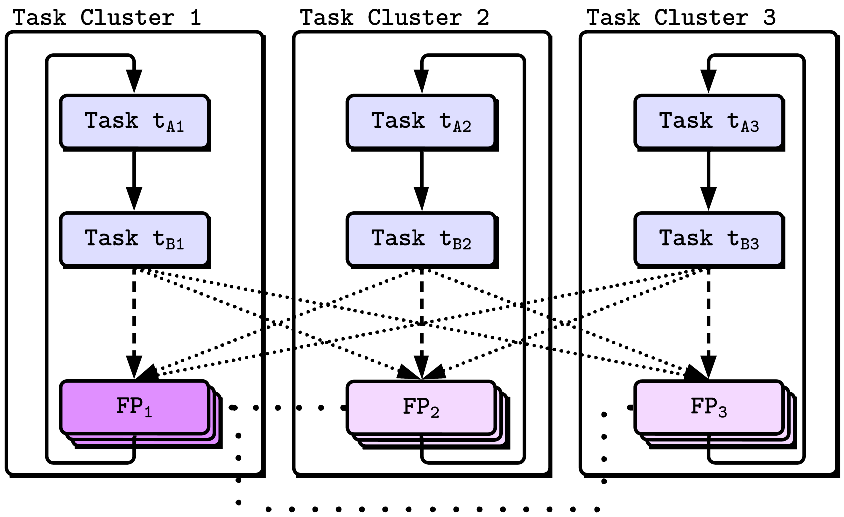


Figure 6.17: Example cyclic Hyracks activity group realized across three machines as three task clusters.

compose the `FIXED POINT` operator. The output of activity A_B connects to the input of the `FP` activity group via a hash partitioned connector, the output of the `FP` activity group connects to the input of activity A_A via a 1:1 connector, and the output of activity A_A connects to the input of activity A_B via a 1:1 connector. Given a cluster of three machines, Hyracks realizes our group of activities as three task clusters: 1) a task cluster of $\{t_{\texttt{A1}},\ t_{\texttt{B1}},\ \texttt{FP}_1\}$, 2) a task cluster of $\{t_{\texttt{A2}},\ t_{\texttt{B2}},\ \texttt{FP}_2\}$, and 3) a task cluster of $\{t_{\texttt{A2}},\ t_{\texttt{B2}},\ \texttt{FP}_2\}$. We depict these task clusters in Figure 6.17.

The liveness and safety properties of the previous section do not require coordination from tasks outside of their partition. Our mortality property, however, requires the consideration of *all* tasks across *all* task clusters to avoid premature / incorrect calls to `close()`. Similar to how we designated the `FIXED POINT` task group to manage the termination of a single task cluster, we designate one `FIXED POINT` task *group* out of all `FIXED POINT` task groups to coordinate the termination for *every* task cluster. In Figure 6.17 (and by default in Graphix), we designate the `FIXED POINT` in the first machine as the "coordinator" to manage this task cluster state. To facilitate communication between each `FIXED POINT` across machines, a custom communication channel is used between the designated FIXED POINT instance and the other FIXED POINT instances[‖] (depicted in Figure 6.17 by the sparse dotted lines).

To minimize the network chatter between machines and to reduce the message-to-data-frame ratio during runtime, message frames *do not* travel across the network at partitioned or broadcast connectors. Instead, each FIXED POINT task group (with the exception of the designated coordinator task group) only receives message frames for tasks local to the FIXED POINT's specific task cluster. For example, the FIXED POINT $\mathtt{FP}_3$ in Figure 6.17 only needs to manage tasks $t_{\mathtt{A3}}$ and $t_{\mathtt{B3}}$. $\mathtt{FP}_3$ then summarizes and transmits the status of tasks $t_{\mathtt{A3}}$ and $t_{\mathtt{B3}}$ to the coordinator FIXED POINT $\mathtt{FP}_1$. Similarly, $\mathtt{FP}_2$ manages tasks $t_{\mathtt{A2}}$ and $t_{\mathtt{B2}}$ (transmitting a summarized status of $t_{\mathtt{A2}}$ and $t_{\mathtt{B2}}$ to the coordinator FIXED POINT $\mathtt{FP}_1$). Using the statuses transmitted by $\mathtt{FP}_2$ and $\mathtt{FP}_3$ (as well as its own status on tasks $t_{\mathtt{A1}}$ and $t_{\mathtt{B1}}$), the coordinator $\mathtt{FP}_1$ is able to reason about and act on the termination status of *every* task of the activity group.

Figure 6.18 demonstrates the individual processes that compose the FIXED POINT operator distributed across our original two machine setup. We give an overview each block of Figure 6.18 below:

Recursive Task The task contained within a looping task cluster. In the activity graph of Figure 6.15, the instances of the decorated REPLICATE activity refer to the recursive input tasks.

Anchor Task The task used to initialize the start of the loop. In the activity graph of Figure 6.15, the instances of the ASSIGN (_ro <= CREATE_PATH(m1)) activity refer to the anchor input tasks.

[‖] In practice, this communication channel between different FIXED POINT task groups 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.

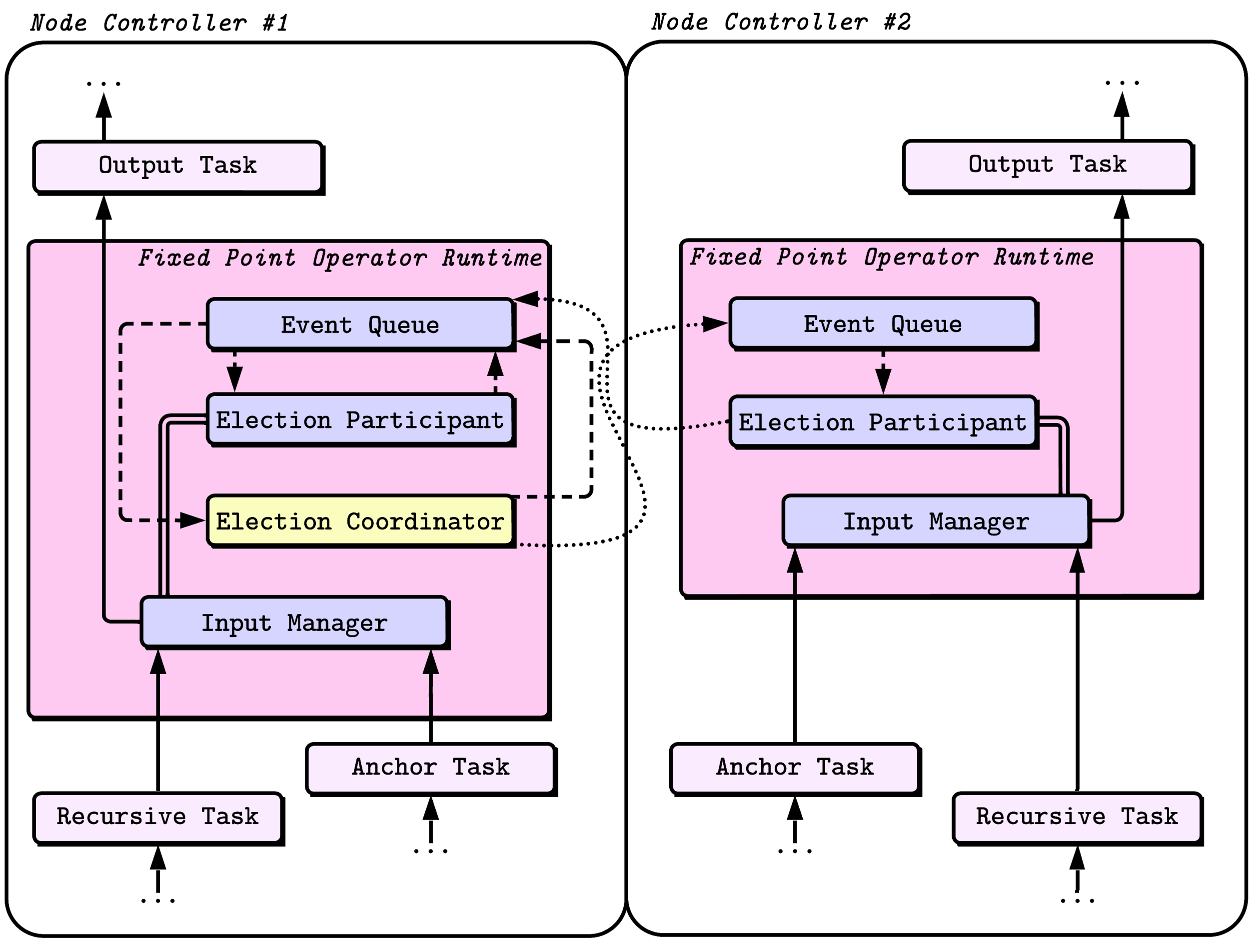


Figure 6.18: Internal processes of the `FIXED POINT` operator, realized as a set of activities (which are then realized as a set of tasks).

Output Task The immediate downstream task of the `FIXED POINT` task group. In the activity graph of Figure 6.15, the instances of the decorated `PIDX SEARCH` activity refer to the output tasks.

Input Manager The task that performs the `UNION ALL` of data frames and manages the transmission (and eventual receipt) of message frames. This task captures all the functionality of the previous section to preserve the liveness and safety properties. The input manager task is also used by the election participant task to call the `close()` method of the output task.

Event Queue The task that (a) listens for "events" from the election coordinator, and (b) listens for events from the election participants (only applicable to the designated coordinator `FIXED POINT` task group). Each received event at this event queue is buffered to be read on-demand by the election processes.

Election Participant The task that (a) collects the status of its local task cluster (via the input manager task), (b) transmits this aforementioned status to the election coordinator's event queue, and (c) listens for coordinator-related events (e.g., when to call `close()`) via the election participant.

Election Coordinator The task that works in tandem with every election participant process across each machine to guarantee that each looping task cluster has no tuples left to offer downstream.

To scale the `FIXED POINT` operator outward (and generalize to larger Hyracks/AsterixDB clusters of size n), Graphix simply duplicates the *Fixed Point Operator Runtime* task set in node controller #2 (abbrv. NC2) of Figure 6.18 to NC3, NC4, ..., NCn.

We now move to the actions the coordinator `FIXED POINT` must take to inform each "participant" `FIXED POINT` that it can call `close()`. Figure 6.19 depicts the algorithm the election coordinator task performs (illustrated as a state machine). The coordinator begins at the `STARTING` state, where it take no action until its local anchor input is exhausted (i.e., when the anchor input task calls the `close()` method of the input manager belonging to the same task group as the election coordinator task). The next state is the `WAITING` state. To move from the `WAITING` state, each election participant must first send the `REQ` event to the coordinator. This `REQ` event is given to the coordinator by a participant when the participant itself observes that there are no tuples (we detail the participant state machine 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 local task cluster *for some instant*. Calling `close()` now would be erroneous, however, because task clusters give tuples to other task clusters *asynchronously*. We can easily visualize an example where a participant sends `REQ` to the coordinator, only to receive tuples immediately after transmitting its status. To

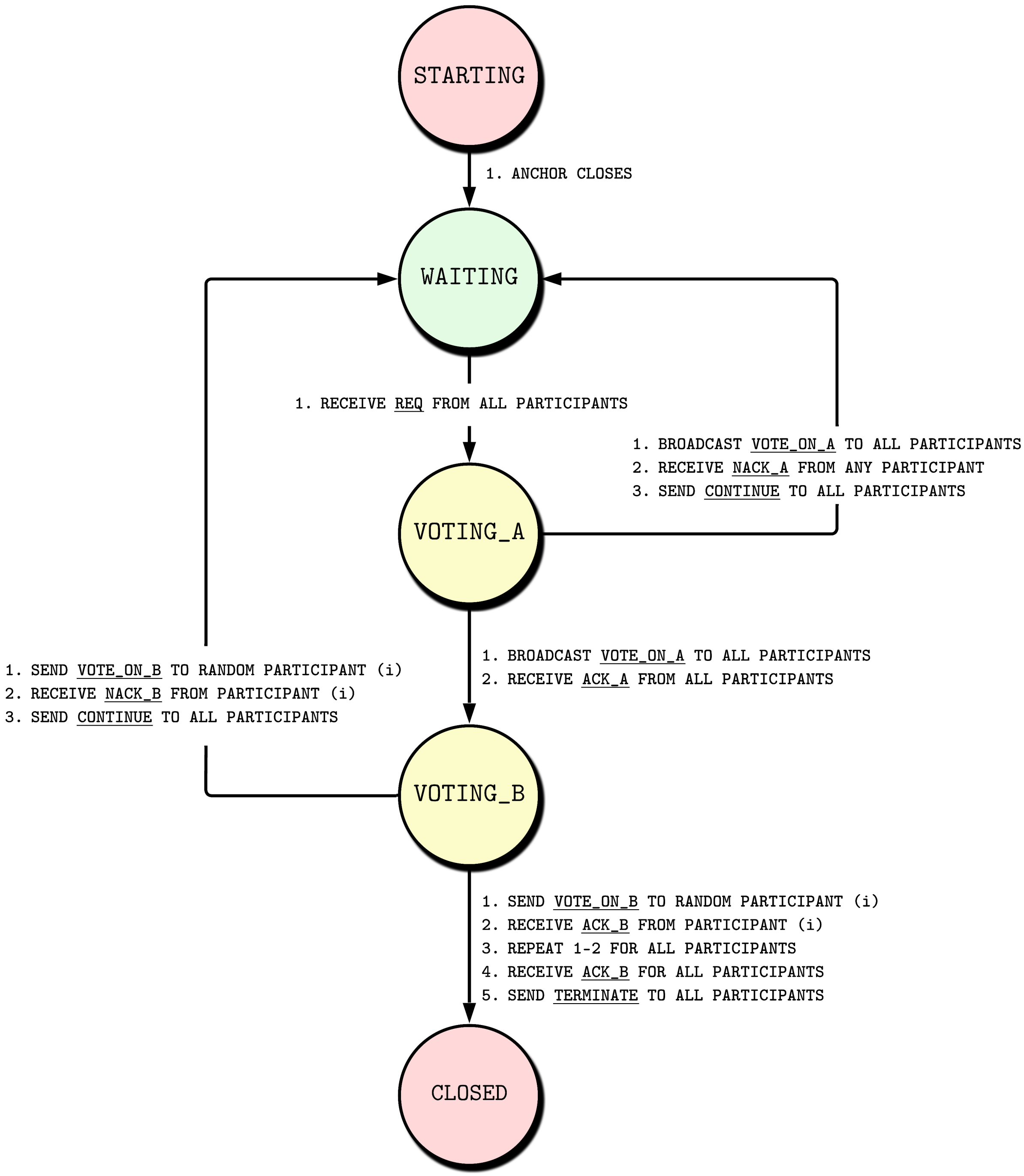


Figure 6.19: State machine representing the algorithm the `FIXED POINT` coordinator executes to terminate its associated set of looping task clusters.

handle this asynchronous nature, the status checking that each task cluster performs is serialized in order to guarantee correctness [70, 45]. Skipping ahead to the `VOTING_B` state to `CLOSED` transition, we see the election coordinator process coordinating the serialization of this status checking to reach the `CLOSED` state:

1. a participant is drawn (without replacement) from all participants and sent a `VOTE` event to re-transmit its message frame and check the status of its local task cluster;
2. the participant replies with an `ACK` (denoting that the participant has observed no tuples locally);
3. the previous two steps are repeated for all participants; and
4. the `TERMINATE` event is sent to all participants.

If any participant replies with a `NACK` (denoting that the participant has observed tuples locally), our coordinator sends the `CONTINUE` event to each participant while transitioning back to the `WAITING` state. The coordinator then waits until each participant transmits a new `REQ` event to perform another election.

To minimize the impact of serializing the task cluster status checking, the status checking is divided into two phases: the A phase and the B phase. During the A phase, the coordinator is in the `VOTING_A` state. Instead of serializing the status checking, the coordinator *broadcasts* the `VOTE_ON_A` event to each participant. The purpose of this 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 local task cluster also contains the `RELEASE` directive to flush the buffers of its corresponding tasks. A participant during the A phase responds with either `ACK_ON_A` or `NACK_ON_A`. When all participants vote with `ACK_ON_A`, the coordinator moves to the B phase which executes the aforementioned serialized status checking.

To conclude our discussion on the `FIXED POINT` operator, we describe the algorithm that every election participant task performs in Figure 6.20. A participant begins in the `STARTING` state, where it does not perform any election related actions until its local anchor input task calls the `close()` method of the input manager task. The next state is the `OBSERVING` state, where we expect a participant to spend the majority of its runtime (relative to the processing of the loop). To transition out of the `OBSERVING` state into the `WAITING` state, the election participant task works with the input manager task to push a message frame with the `RELEASE` directive and an uncolored marker. Using the aforementioned "modify-and-forward" actions of every decorated task in the loop, the message frame will eventually arrive back to the input manager task. If the marker is colored in, the participant stays in the `OBSERVING` state and pushes a new message frame with an uncolored marker. If the marker is not colored in, the participant transitions to the `WAITING` state. Note that a local task cluster never has more than one message frame in circulation (illustrating another design point to minimize the message-to-data-frame ratio).

Once a participant is in the `WAITING` state, the participant sends a `REQ` event to the coordinator. While in this state, the participant does not transmit any message frames. Only upon receiving the `VOTE_ON_A` response event from the election coordinator can the participant move to the next state: `VOTING_A`. In the `VOTING_A` state, a participant will repeat its status checking procedure (i.e., sending the message frame with a `RELEASE` directive and an uncolored marker). If a participant receives an uncolored marker in return, the participant sends the `ACK_A` event to the election coordinator. If a participant receives a *colored* marker in return, the participant sends the `NACK_A` event to the election coordinator and eagerly returns back to the `OBSERVING` state (denoted by the dashed line). An eager transition back to `OBSERVING` is meant to increase the throughput of tuples, as the participant is already aware that the coordinator is going to respond with `CONTINUE`. The self-loop "`RECEIVE CONTINUE`" transitions for the `OBSERVING` and `WAITING` states address two cases where a participant receives the `CONTINUE` response from the coordinator after an eager transition out of the `VOTING_A` state.

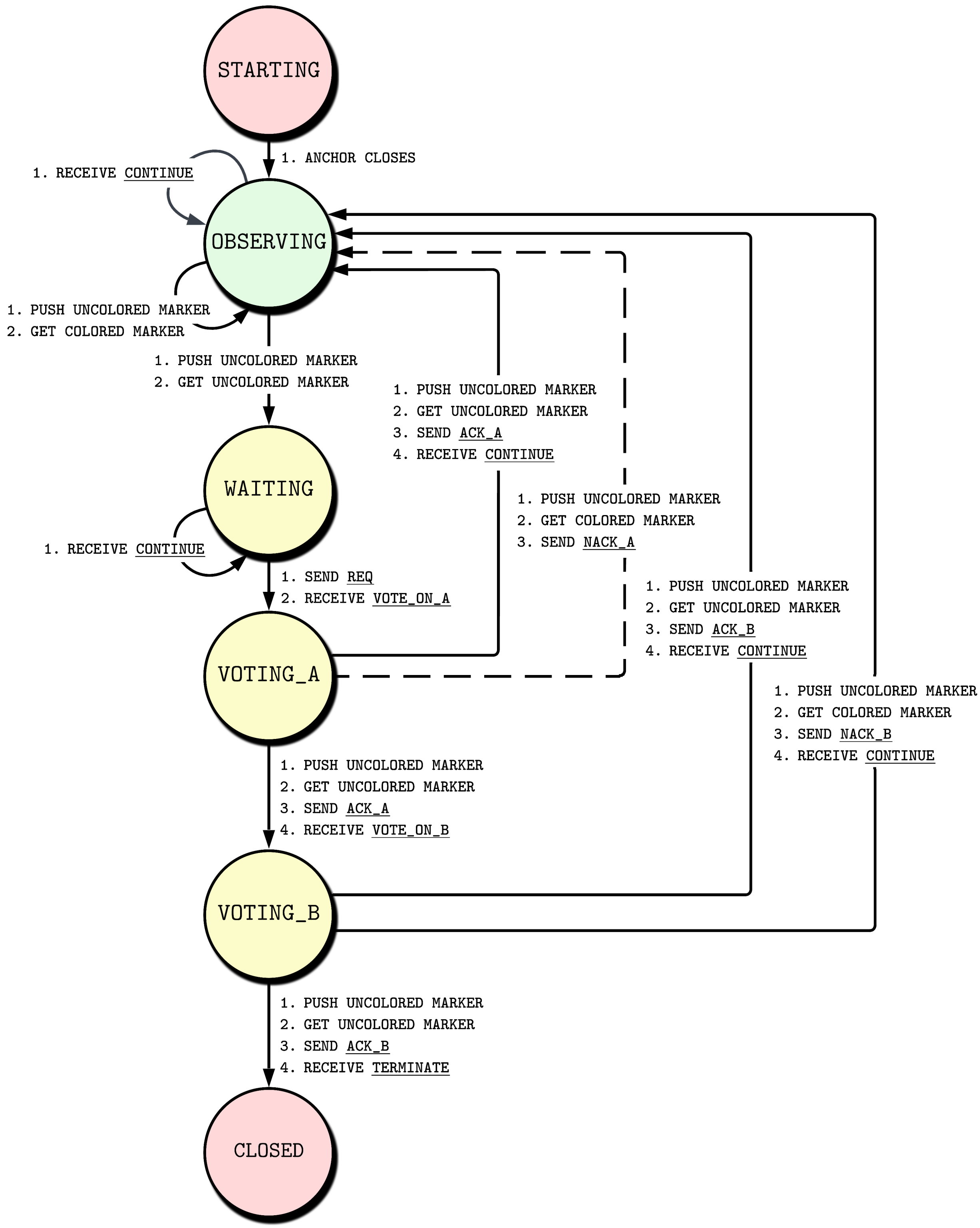


Figure 6.20: State machine representing the algorithm that every `FIXED POINT` participant executes to work with its coordinator to call `close()`.

When a participant receives the `VOTE_ON_B` event, it again repeats its status checking procedure. If a participant receives a colored marker, the participant replies with `NACK_B` and waits** for the subsequent `CONTINUE` event to transition back to the `OBSERVING` state. When a participant receives an uncolored marker in return, the participant sends the `ACK_B` event to the election coordinator. Once all participants have sent `ACK_B` back to the coordinator, the coordinator will broadcast `TERMINATE` to every participant. Finally, each participant will call its downstream task's `close()` method to terminate the loop computation.

6.2.8 Additional Hyracks Operators

In addition to the aforementioned `FIXED POINT` and `MESSAGE SINK` operators, Graphix also provides two additional operators to optimize navigation: i) the `PERSISTENT BUILD JOIN` operator, used to evaluate edge hops using hybrid hash join principles, and ii) the `TOP K` operator, used to bound the number of paths yielded by a looping activity group. In this section, we detail both operators.

Persistent Build `JOIN` (PBJ) Hyracks Operator . . . .

Each activity graph for navigation thus far has assumed the existence of a primary index on the `JOIN` field used to evaluate each edge hop (e.g., the primary index of the `id` field for the `Messages` dataset). Graphix, however, should work independently of how the underlying data is represented. For non-indexed data, we initially considered a nested-loop `JOIN` for each edge hop for each path, but such a direction would not benefit from the plethora of research that has gone into hash `JOIN`s over the years [36]. Knowing that an edge hop is

**In contrast to the previous A phase, a participant does not wait for other participants after sending `NACK_B` in the B phase. Thus, it simply waits to receive the inevitable `CONTINUE` event to simplify our state machine.

always realized as an equi-`JOIN` (see Section 6.3 for more details), we instead decided to use and extend the optimized hybrid-hash `JOIN` Hyracks operator.

Given the equi-`JOIN` $R \bowtie S$, a hash `JOIN` requires two activities: 1) the *build* activity, which scans R (or S) to build a hash table to load into memory, and 2) the *probe* activity, which iterates through all of S (or R) to probe the hash table and evaluate the `JOIN` itself. To handle data volumes larger than memory, Hyracks provides a hybrid-hash `JOIN` operator that extends the previous activities to leverage the disk when appropriate. More specifically, hybrid-hash `JOIN` will "partition" (not to be confused with partitioning across machines in a cluster) R and S according to some hash function while operating the two aforementioned activities, with some partitions being spilled to disk [62]. Once the probe activity has exhausted all of its input (i.e., when the `close()` method is called for the probe), hybrid-hash `JOIN` will recursively build, probe, partition, and spill until every probe tuple has been considered. The Hyracks hybrid-hash `JOIN` operator in particular contains several optimizations to be more robust to skew [37], making this `JOIN` operator a good candidate for evaluating vertices with a high degree.

In order to use hybrid-hash `JOIN` for path navigation, our operator must be able to forward any spilled tuples when a message frame containing the `RELEASE` directive is received *and* not have to rebuild the hash table after forwarding. In the context of the existing Hyracks hybrid-hash `JOIN` operator, we note that hash table used for the initial probe is discarded to maximize the memory available to perform the spilled-tuple-`JOIN`ing. For navigational queries in Graphix, we provide an extended version of the optimized hybrid-hash `JOIN` that *persists* the initial hash table in memory. All Hyracks operators must adhere to a memory budget [39], thus we add an additional parameter α to the hybrid-hash `JOIN`. Given a memory budget M, α determines the ratio of memory dedicated to (i) persisting the hash table in memory ($\alpha \cdot M$) vs. (ii) performing the spilled-tuple-`JOIN`ing action ($(1-\alpha) \cdot M$). By default, we set $\alpha = 0.5$ (though potential future work involves finding a more optimal ratio).

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

Duplicate of Listing 6.1. gSQL++ query to find a) a specific user u, b) messages m1 written by u, c) messages m2 that m1 replied to, and d) reply chains r from m1 to m2.

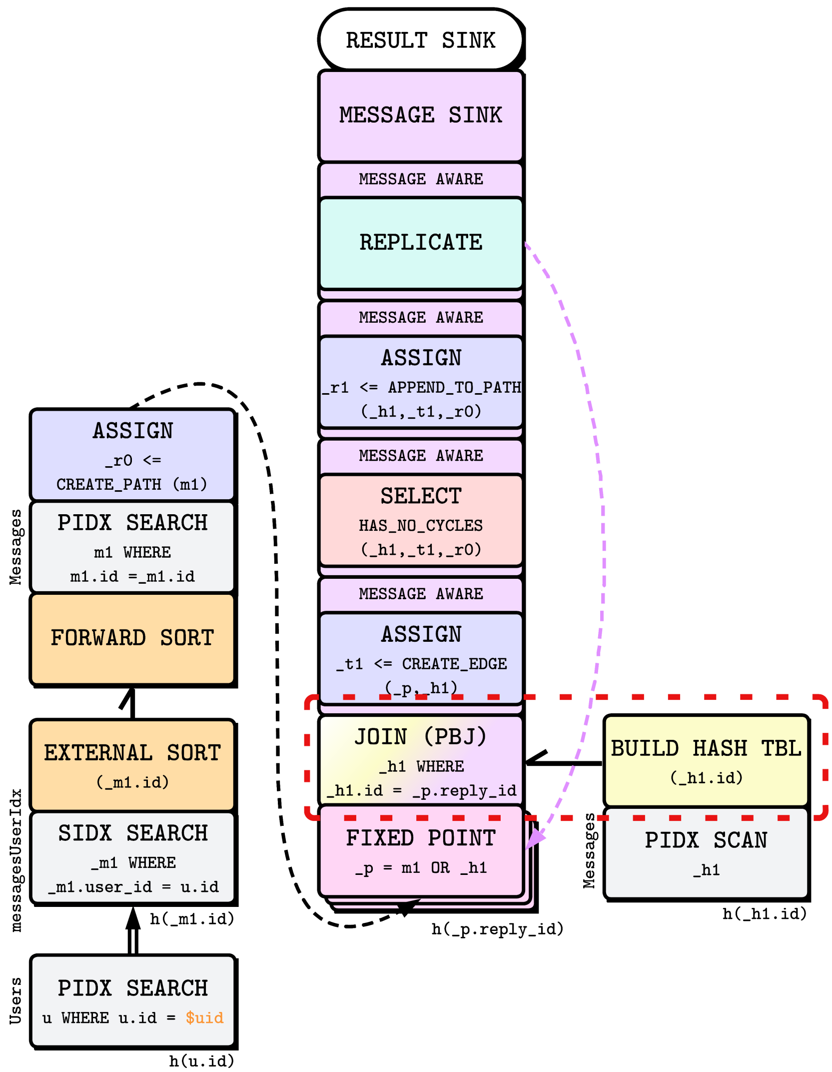


Figure 6.21: Hyracks activity graph to realize the query in Listing 6.1 using PBJ operator (whose constituent activities are surrounded by the red dotted box) to evaluate navigational edge hops.

Figure 6.21 depicts an activity graph using the previously specified operator (denoted as persistent build `JOIN`, or "PBJ" for short). In the stage prior to navigation, Graphix will first scan the entire `Messages` dataset to "partition", build the hash table, and spill `Messages` tuples if necessary. In the same stage, Graphix will (in parallel) run the `PIDX SEARCH` → `SIDX SEARCH` → `EXTERNAL SORT` activity group. Once both of the aforementioned activity groups have finished, the next stage starts to perform the navigation itself. When the `JOIN` activity of PBJ receives a message frame with the `RELEASE` directive, it calls the `close()` method of the original Hyracks optimized hybrid-hash `JOIN` to yield all spilled tuples and maintain the liveness property.

Hash Partitioned Top-k Hyracks Operator. . . . .

Recall from our discussion on subgraph reachability (Subsection 5.4.3) that enumerating all paths will yield a massive number of results for large enough graphs. For the majority of queries, however, most users are concerned with a select few (k) paths between two vertices v_1, v_2. As discussed in Subsection 5.4.4 and Subsection 5.4.5, users can express which k paths they are interested in using some monotonically increasing weight function (realized in Graphix using a SQL^{++} `GROUP BY` ... `GROUP AS` clause). In this section we introduce the `TOP K` operator, used to supplement the evaluation of shortest k paths, cheapest k paths, and any k paths (i.e., reachability) queries.

At a high level, the `TOP K` operator constructs a hash-distributed LSM-based B$^+$ tree where (a) the sort key is composed of the endpoint vertices of a path, and (b) the payload is the weight associated with the path. For each tuple in a frame (where a tuple contains endpoint vertices (v_1, v_2), a path p, and the weight associated with a path $c(p)$), a `TOP K` instance will first look up the endpoint vertices (v_1, v_2) in the B$^+$ tree. If there exists no entry in the B$^+$ tree with these endpoints, `TOP K` will forward the tuple downstream and store the tuple

$\langle v_1, v_2, c(p)\rangle$ in the tree. If an entry with the key (v_1, v_2) is found in the B$^+$ tree, `TOP K` takes one of three actions:

1. If `TOP K` finds fewer than k other entries with the search key (v_1, v_2), it forwards the tuple downstream and stores the tuple $\langle v_1, v_2, c(p)\rangle$.
2. If `TOP K` finds exactly k existing entries with the search key (v_1, v_2), `TOP K` must then determine if the working tuple has a lower $c(p)$ value than any other entry in the B$^+$ tree. If `TOP K` finds that the working tuple has a higher $c(p)$ value, then `TOP K` does not forward the working tuple downstream.
3. If `TOP K` finds exactly k existing entries with the search key (v_1, v_2) and `TOP K` finds that the working tuple has a lower $c(p)$ value than any of the matching k B$^+$ entries, `TOP K` will a) delete the B$^+$ entry with the largest $c(p)$ value, b) forward the working tuple downstream, and c) store the tuple $\langle v_1, v_2, c(p)\rangle$.

We note that a tuple yielded by a `TOP K` instance might not necessarily be in the final set of k tuples containing the cheapest paths, as a cheaper path may be discovered later in the process. These false positives need to be filtered out by other operators downstream.

Consider the shortest path query in Listing 6.5, which asks for the shortest path of `REPLY_OF` edges between two vertices `m1` and `m2`. Figure 6.22 depicts an activity graph using the `TOP K` operator to realize the Listing 6.5 query. In the `ASSIGN` immediately above the `APPEND_TO_PATH ASSIGN` activity, `leng` (i.e., the weight of a path `_r1`) is computed by determining the path length `LEN(_r1.Edges)`. The outbound connector of this `ASSIGN` activity then hash partitions its output on the B$^+$ tree search key (`m1.id`, `_h1.id`) to distribute the work across all task clusters. At the `TOP K` activity, paths `_r1` will be selectively filtered out according to the criteria above. All paths outside the loop (after the `MESSAGE SINK` activity) are given to 1) an `EXTERNAL SORT` activity, followed by 2) a `FORWARD SORT` activity (to merge and forward the results of the `SORT` activity), followed by 3) a (pre-clustered) `GROUP BY` activity sequence to ultimately group all *filtered* paths by their endpoints `m1` and `m2`. The activity group attached

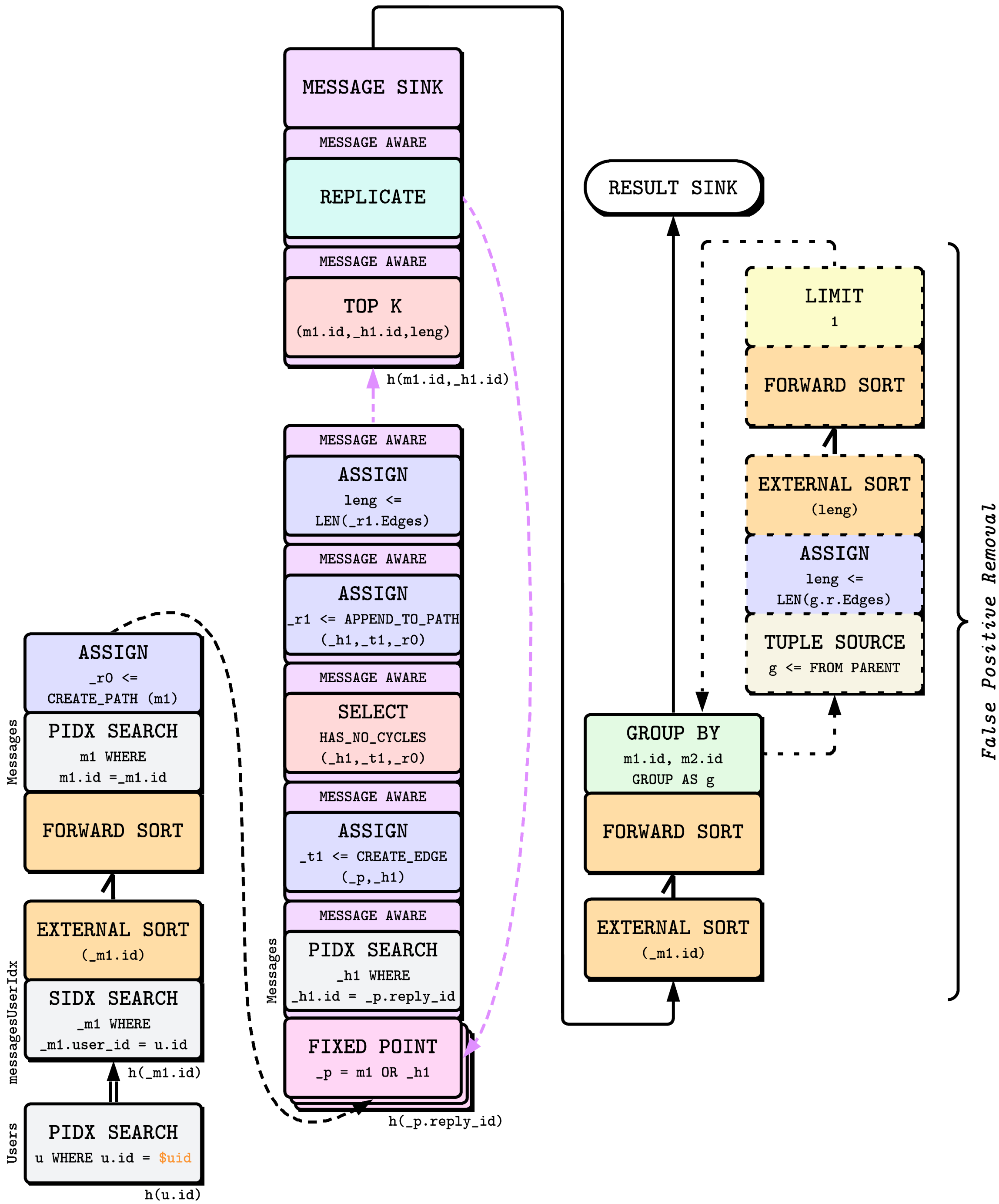


Figure 6.22: Hyracks activity graph to realize the shortest path query in Listing 6.5 without enumerating all paths.

```
FROM
    GRAPH SocialNetworkGraph
        (u:User)-[:WROTE]->(m1:Message),
        (m1)-[r:REPLY_OF+]->(m2:Message)
WHERE
    u.id = $uid
GROUP BY
    m1.id AS m1_id,
    m2.id AS m2_id
    GROUP AS g
LET
    shortestPath = (
        FROM
            g
        SELECT VALUE
            g.r
        ORDER BY
            LEN(g.r.Edges) ASC
        LIMIT
            1
    )[0]
SELECT
    m1_id,
    m2_id,
    shortestPath;
```

Listing 6.5: gSQL^{++} query to find the a) messages `m1` written by a specific user `u`, b) messages `m2` that `m1` replied to, and c) the *shortest* reply chain `r` from `m1` to `m2`.

to the right of the `GROUP BY` is a *subplan* that corresponds to the `shortestPath` subquery in Listing 6.5. As demonstrated in the shortest path example of the query model section (see Subsection 5.4.4), the subplan corresponding to the `shortestPath` subquery executes once per group. If `TOP K` does not yield any false positives for some group g, the `TUPLE SOURCE` activity in the subplan iterating over g will yield a single tuple to its downstream activity (the `ASSIGN`). If `TOP K` did, however, yield false positives for some group g, the subsequent `EXTERNAL SORT`, `FORWARD SORT`, and `LIMIT` activities in the subplan performs the removal of false positives.

6.2.9 "Paths Not Traveled" (Alternatives)

Having detailed the approach Graphix takes to handle navigational queries in Hyracks, this section will briefly describe a few "paths not taken" (i.e., solutions that we did not use). Recall that the `FIXED POINT` operator is responsible for a) forwarding a message frame with the `RELEASE` directive downstream to get all tasks within the loop to invoke their own `flush()` method, b) buffering all tuples that arrive at its input, and c) calling `close()` method of its downstream task to terminate the loop. This section describes two alternative approaches for maintaining liveness, safety, and mortality: i) the use of a *manager* process that directly communicates with each task in the loop, and ii) the use of an external *recursion manager* for circumventing the liveness, safety, and mortality properties that are inherent to cyclic activity graphs.

Figure 6.23 describes a potential alternative activity graph to our original activity graph in Figure 6.15 which would a) decorate each activity in-the-loop with a `MANAGER AWARE` decorator (not to be confused with our original `MESSAGE AWARE` decorator), b) introduce a new `MANAGER` process that *directly* communicates to each decorated activity, and c) sit in the middle of the `REPLICATE` and `UNION ALL` activities. To maintain the liveness property, the `MANAGER` process would directly inform each activity in-the-loop (via the `MANAGER AWARE` decorator) to call the decorated activity's `flush()` method. To maintain the safety property, the `MANAGER` process would maintain a secondary buffer to hold the incoming tuples from the `REPLICATE` activity. To maintain the mortality property, the `MANAGER` process would be responsible for calling the `close()` method of the `UNION ALL` activity after correctly reasoning that there exists no tuples left to process. This Figure 6.23 solution is reminiscent of how data flow systems like Naiad [47] track progress. While Naiad is able to minimize the chatter its `MANAGER` process uses to track progress, Naiad ultimately requires explicit management of each and every activity in-the-loop. We contrast this potential solution to our original

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

Duplicate of Listing 6.1. gSQL++ query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2`.

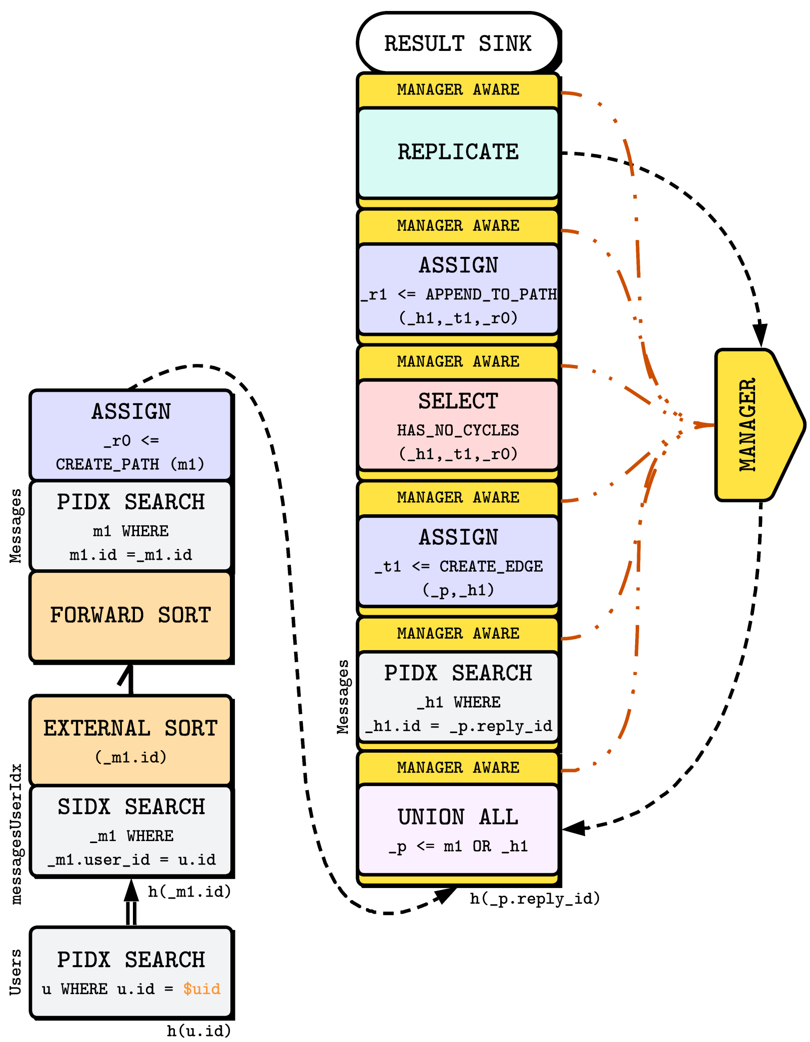


Figure 6.23: Potential alternative Hyracks activity graph to realize the query in Listing 6.1 that relies on some manager process directly communicating with each task in-the-loop.

solution in Figure 6.15, where the `FIXED POINT` operator is *agnostic* of any computation that occurs in the loop.

Figure 6.24 describes a potential alternative activity graph to our original activity graph in Figure 6.15 which would a) cut the original Hyracks *job* (i.e., the operator graph) along the operator subgraph to compute the path, b) execute each path hop *synchronously* as a single job via an external recursion manager, and c) terminate when this external recursion manager observes no work. At a high level, Figure 6.24 can also be thought of as the "bulk-synchronous-parallel" (BSP) potential alternative to recursion in Graphix. Note that the activity graph to execute each edge hop here is acyclic. Consequently, liveness, safety, and mortality (with respect to a single Hyracks job) do not need to be addressed. Recursion is realized in Figure 6.24 via an external recursion manager. This external recursion manager is responsible for issuing the same Hyracks job until a least fixed point is reached. Figure 6.24 is the approach adopted by Pregelix [13], a graph processing system that acts on top of Hyracks to realize graph computations like PageRank. We argue, however, that Figure 6.24 assumes too *little* about the specific problem of path navigation in Graphix. We contrast this alternative solution to our original solution in Figure 6.15, which leverages the fact that path navigation in a shared-nothing cluster of machines does not require Hyracks to synchronize for each and every path hop.

We conclude this discussion of recursion in Hyracks with a few high level characteristics of our `FIXED POINT` operator and message passing solution:

Semi Synchronous Evaluation Task clusters across different node controllers operate asynchronously with different partitions of the data. Paths in Graphix grow independently of one another. Synchronization is only required for the final phase (i.e., phase B) of termination.

Minimally Invasive Design Existing pipelineable Hyracks activities did not need to be rewritten to be used in a cyclic activity graph. Engineering-wise, the message aware

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

Duplicate of Listing 6.1. gSQL++ query to find a) a specific user `u`, b) messages `m1` written by `u`, c) messages `m2` that `m1` replied to, and d) reply chains `r` from `m1` to `m2`.

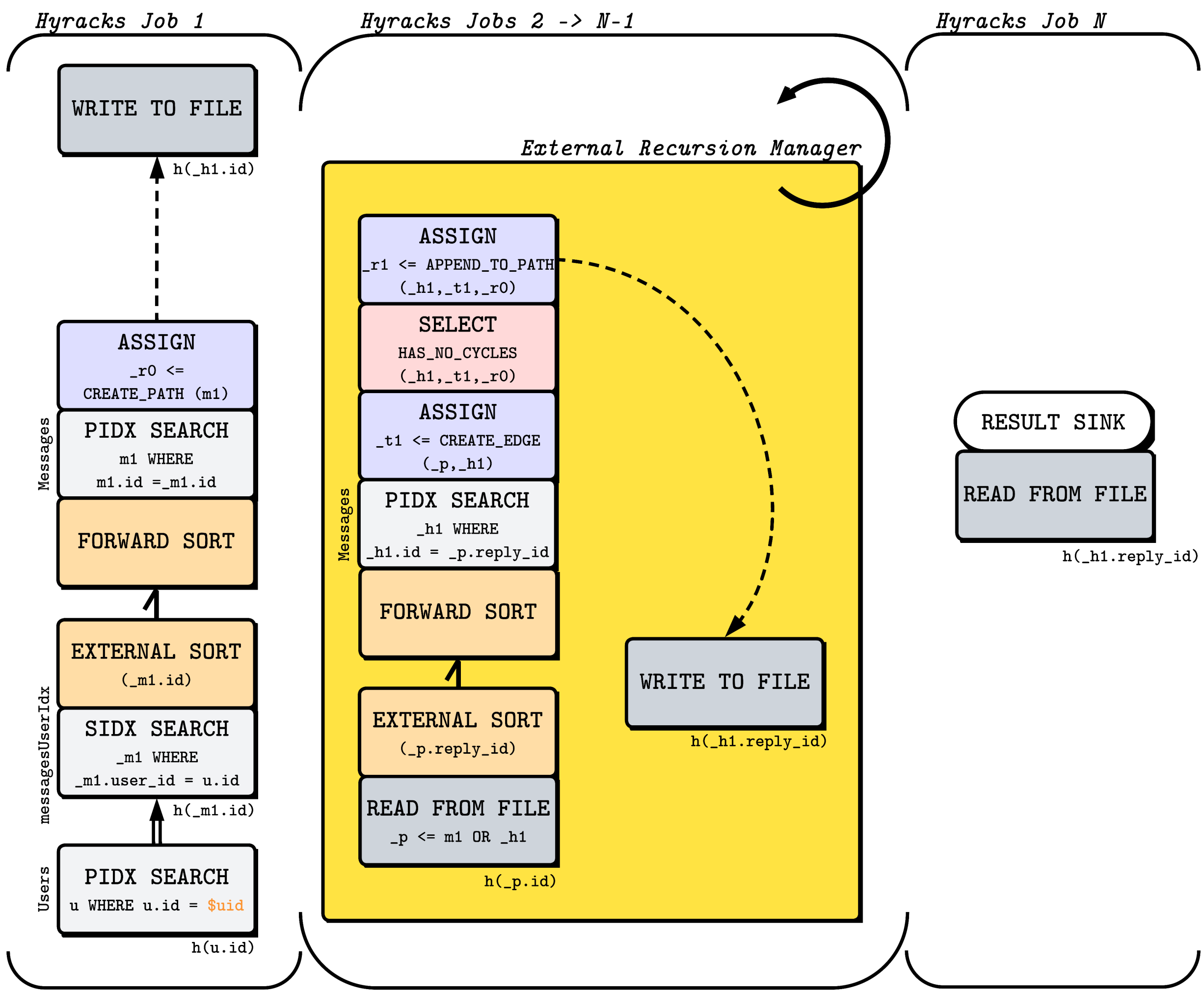


Figure 6.24: Potential alternative Hyracks activity graph to realize the query in Listing 6.1 using an external recursion manager plus a bulk-synchronous-parallel evaluation style.

decorator could be generalized to solve other problems that require in-band message passing.

Message-Sparse Protocol Only the `FIXED POINT` is allowed to generate message frames. Only one message is ever in transit for a single task cluster. Messages do not travel across the network (between different node controllers).

Loosely Coupled Design No sole "process" possesses more responsibility than necessary. In the spirit of Hyracks, tasks in-the-loop (that are not in the `FIXED POINT` task group) do not manage information about other tasks. Participant `FIXED POINT` instances are unaware of the computation in the loop and do not manage information about other participants.

6.3 Abstract Syntax Tree Rewriter

As outlined earlier, the abstract syntax tree (AST) rewriter in Graphix serves to rewrite the gSQL^{++} AST $T^0(Q)$ generated immediately parsing (via a JavaCC generated parser) into a SQL^{++}-compatible AST $T^1(Q)$. We denote the transformation of $T^0(Q)$ to $T^1(Q)$ as "lowering". The SQL^{++}-compatible AST $T^1(Q)$ then undergoes the same set of AST-level rewrites as a standard SQL^{++} query to generate an AST $T^2(Q)$ which then is translated into an Algebricks logical plan $P^0(Q)$. In this section, we will bridge our query model discussion (Chapter 5) and our graph model discussion (Chapter 4). We divide this section into two parts: a) a high level overview of the sets of SQL^{++} and gSQL^{++} AST rewrites, and b) a description of the gSQL^{++} to SQL^{++} AST lowering.

6.3.1 gSQL^{++} AST Rewriting

An AST rewrite rule accepts an AST $T(Q)$ as input and (potentially) modifies $T(Q)$ in place to create $T'(Q)$. We contrast the AST rewriter to Algebricks in the next section, where rules are executed by a rule controller (adding an additional layer of abstraction). Rules are implemented using a visitor design pattern, which defines operational logic for each AST node.

At a high level, the gSQL^{++} $T^0(Q)$ to $T^1(Q)$ rewrites consist of the following:

1. Verifying that all `MatchExpr` AST nodes define well-formed query patterns in the gSQL^{++} query model. This verification includes:
   (a) ensuring that all vertex labels, edge labels, and RPQ (regular path query) symbols exist in the graph being queried (note that in contrast to Neo4j, the universe of all labels is known at compilation time for Graphix);
   (b) checking that the minimum m and maximum n bounds of an RPQ (if defined) satisfy $0 \leq m \leq n$; and
   (c) forbidding the reuse of edge and path variables (i.e., all edges and paths have a one-to-one correspondence with a triple in the query pattern incident set).
2. Fetching the description of the graph being queried $G_D = (V_D,\ E_D,\ I_D,\ \lambda_D)$ using the name defined in the corresponding `FromTerm` AST node (i.e., the name after the `GRAPH` token). Implementation-wise, this action involves a metadata query to the metadata node or a fetch from the metadata cache (local to the cluster controller process itself). If the graph being queried is a temporary one (i.e., one defined in a `WITH GRAPH` clause), this step can be skipped as G_D is already fully defined.
3. Resolving vertex labels, edge labels, edge directions, RPQ symbols, and path pattern directions using:
   (a) the description of the graph being queried G_D;

(b) the labels assigned to each vertex and edge pattern (via the labeling function λ_Q);

(c) the symbols of each path pattern RPQ (via the R_Q set); and

(d) the directions of each edge / path pattern (via the incidence triple set I_Q).

This resolution is currently an exhaustive process: all combinations of labels, symbols, and directions are considered before being pruned according to the graph schema dictated by the I_D incidence set description. For example, an unlabeled edge pattern between two vertex "`User`"-labeled patterns cannot possess the label "`REPLY_OF`". Similarly, an undirected edge pattern $e_Q \in E_Q$ labeled "`WROTE`" connecting a "`User`"-labeled vertex pattern on the left and a "`Message`"-labeled vertex pattern on the right can only be directed from left-to-right because a `(:Message)` cannot "write" a `(:User)` according to the definition of I_D. Future work involves a more "bottom-up" approach to schema resolution (I_Q and λ_Q) that avoids enumerating all possibilities.

4. Rewriting all shared vertex patterns (e.g., the vertex sharing in the negative pattern matching example in Subsection 5.4.2) to a) not share vertex patterns, and b) to be explicitly correlated in SQL^{++} via a `JOIN`. An example of this AST rewrite is given in Figure 6.25.
5. Finally lowering the gSQL^{++} AST $T^0(Q)$ to use SQL^{++} AST nodes $T^1(Q)$ for use in the SQL^{++} AST rewrite set (described more in the next section).

Once all gSQL^{++} AST nodes have been factored out, the AST $T^1(Q)$ undergoes the same set of AST rewrites as a normal SQL^{++} query would immediately after SQL^{++} parsing in AsterixDB. These include (but are not limited to) (i) `WINDOW` function and expression rewrites, (ii) `GROUPING SETS` rewrites, (iii) `RIGHT OUTER JOIN` rewrites, (iv) the user-defined-function (UDF) and view inlining rewrites, and (v) the `WITH` clause inlining rewrite. After the SQL^{++} rewrites ($T^1(Q)$ to $T^2(Q)$), the AST is finally translated into an Algebricks logical plan. We contrast the two step rewrite process $T^0(Q) \to T^1(Q) \to T^2(Q)$ to the one step rewrite process that avoids the SQL^{++} AST rewrites altogether (i.e., $T^0(Q) \to T^2(Q)$). Defining $T^1(Q)$ as an intermediate AST not only decreases the engineering maintenance cost (i.e.,

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)-[:WROTE]->(:Message)
WHERE
    NOT EXISTS (
        FROM
            GRAPH SocialNetworkGraph
                (u1)-[:KNOWS]->(:User)
        SELECT *
    )
SELECT *;
```

$$\Downarrow$$

```
FROM
    GRAPH SocialNetworkGraph
        (u1:User)-[:WROTE]->(:Message)
WHERE
    NOT EXISTS (
        FROM
            GRAPH SocialNetworkGraph
                (u1_inner:User)-[:KNOWS]->(:User)
        WHERE
            u1_inner.id = u1.id
        SELECT *
    )
SELECT *;
```

Figure 6.25: Example of the shared vertex pattern AST rewrite transformation. The modified lines are highlighted in green.

"main branch" AsterixDB maintainers do not have to replicate their work specifically for Graphix), but also mirrors the implementation philosophy found in the Hyracks section of this chapter and (to be found) in the Algebricks section of this chapter: to incorporate and reuse as much of AsterixDB as possible.

6.3.2 gSQL^{++} Lowering to SQL^{++}

We now move to the final rules of the gSQL^{++} AST $T^0(Q)$ to $T^1(Q)$ rewrite set, which performs the actual lowering. We acknowledge that the specific problem of translating *bounded* graph queries into SQL has been studied extensively before in the context of the RDF model and SPARQL, with early attempts using correlated sub-queries to express the connection between vertices [76]. To help solve the problem of translating gSQL^{++} queries into SQL^{++} queries, we borrow SPARQL to SQL translating methods from Elliott et al. [22] to create SQL^{++} ASTs that are more amenable to optimization (i.e., easier for us to reason about at the Algebricks layer). Our desiderata for this section includes (i) avoiding superfluous query nesting, and (ii) minimizing the number of `JOIN` operations.

To start, consider the gSQL^{++} query and its equivalent SQL^{++} query in Figure 6.26. The first subquery from Line 2 to Line 7 of the SQL^{++} query (bottom) refers to the `(m:Message)` vertex pattern in the gSQL^{++} query (top). This first subquery comes from the `Message` vertex body in the `CREATE GRAPH` of Listing 4.1 and is bound to the variable `m`. The second subquery from Line 8 to Line 13 of the SQL^{++} query refers to the `<-[w:WROTE]-` edge pattern in the gSQL^{++} query. This second subquery comes from the `(:User)-[:WROTE]->(:Message)` edge body in the `CREATE GRAPH` DDL and is bound to the variable `w`. The `Users u` after the previous two subqueries refers to the `(u:User)` vertex pattern. Similar to the first and second subqueries, the "`Users`" from `Users u` comes from the definition of a `(:User)` vertex in the `CREATE GRAPH` DDL. To correlate each term in the `FROM` clause, two conjuncts are added to

```
CREATE GRAPH SocialNetworkGraph AS
    VERTEX (:User)
        PRIMARY KEY (id)
        AS Users,
    VERTEX (:Message)
        PRIMARY KEY (id)
        AS ( FROM
                Messages m
            WHERE
                NOT m.is_draft
            SELECT
                m.* ),
    EDGE (:User)-[:KNOWS]->(:User)
        SOURCE KEY      (source_id)
        DESTINATION KEY (dest_id)
        AS ( FROM
                Users u,
                u.knows k
            SELECT
                u.id AS source_id,
                k    AS dest_id ),
    EDGE (:User)-[:WROTE]->(:Message)
        SOURCE KEY      (user_id)
        DESTINATION KEY (message_id)
        AS ( FROM
                Messages m
            SELECT
               m.user_id   AS user_id,
               m.id        AS message_id,
               m.posted_on AS posted_on ),
    EDGE (:Message)-[:REPLY_OF]->(:Message)
        SOURCE KEY      (source_id)
        DESTINATION KEY (dest_id)
        AS ( FROM
                Messages m
            SELECT
                m.id        AS source_id,
                m.reply_id  AS dest_id,
                m.posted_on AS posted_on );
```

Duplicate of Listing 4.1. `CREATE GRAPH` DDL describing the `SocialNetworkGraph`.

```
FROM
    GRAPH SocialNetworkGraph
        (m:Message)<-[w:WROTE]-(u:User)
WHERE
    u.id = 70
SELECT
    m.id          AS mid,
    w.message_id AS w_mid,
    w.user_id     AS w_uid,
    u.id          AS u_id;
```

$$\Downarrow$$

```
FROM
    ( FROM
          Messages m
      WHERE
          NOT m.is_draft
      SELECT
          m.* ) m,
    ( FROM
          Messages m
      SELECT
          m.user_id    AS user_id,
          m.id         AS message_id,
          m.posted_on AS posted_on ) w,
    Users u
WHERE
    u.id = 70 AND
    m.id = w.message_id AND
    u.id = w.user_id
SELECT
    m.id          AS mid,
    w.message_id AS w_mid,
    w.user_id     AS w_uid,
    u.id          AS u_id;
```

Figure 6.26: Example of a gSQL^{++} query being lowered into an equivalent SQL^{++} query.

the `WHERE` clause: `m.id = w.message_id` and `u.id = w.user_id`. The access to the `id` field of `m` is defined using the `PRIMARY KEY` line of the `(:Message)` vertex definition. The access to the `message_id` and `user_id` fields of `w` is defined from the `DESTINATION KEY` and `SOURCE KEY` lines of the `(:User)-[:WROTE]->(:Message)` edge definition respectively. Finally, the access to the `id` field of `u` is defined from the `PRIMARY KEY` line of the `(:User)` vertex definition.

Now consider another translation of the same gSQL^{++} query (from Figure 6.26) to a *nearly* equivalent SQL^{++} query in Figure 6.27. In comparison to SQL^{++} query in Figure 6.26, the SQL^{++} query of Figure 6.27 a) removes the nesting of `m`, b) pushes the `NOT m.is_draft` condition to the outer `WHERE` clause, and c) rewrites `w` and `u` subqueries as objects built fields from `m`. The latter rewrite (i.e., replacing subqueries in the `FROM` clause with objects) is legal if Graphix can guarantee that no properties of `w` or `u` are required. For example, if the `SELECT` clause was `SELECT u` or `SELECT u.name`, then Graphix would not be able to rewrite the `u` term of the `FROM` clause in this way. We also note that the changes Graphix make preserve `w` and `u` as variables available for the rest of the query to use. This preservation allows Graphix to minimize the total number of edits it makes to the original gSQL^{++} query. The gSQL^{++} AST lowering rewrite does *not* modify any conditions found in the original `WHERE` clause (e.g., the `u.id = 70` conjunct) nor does the lowering rewrite modify the `GROUP BY`, `SELECT`, `ORDER BY`, or `LIMIT` clauses. The lowered SQL^{++} query in Figure 6.27 is equivalent to its initial gSQL^{++} query if (and only if):

1. the `id` field used in `u.id` is the primary key of the `Messages` dataset;
2. the `id` field used in `m.id` is the primary key of `Users`; and
3. every `user_id` field of a record in the `Messages` dataset points to an existing record in the `Users` dataset (i.e., there exists no dangling foreign keys).

While the first two points can potentially be inferred from the metadata about each dataset, AsterixDB (and Graphix) cannot infer the last point without evaluating the `JOIN` between `m` and `u`. Graphix provides a compiler flag `graphix.evaluation.minimize-joins` (disabled by

```
FROM
    GRAPH SocialNetworkGraph
        (m:Message)<-[w:WROTE]-(u:User)
WHERE
    u.id = 70
SELECT
    m.id         AS mid,
    w.message_id AS w_mid,
    w.user_id    AS w_uid,
    u.id         AS u_id;
```

$$\Downarrow$$

```
FROM
    Messages m
LET
    w = { "user_id"   : m.user_id,
          "message_id": m.id,
          "posted_on" : m.posted_on },
    u = { "id"        : m.user_id }
WHERE
    u.id = 70 AND
    NOT m.is_draft AND
    m.id = w.message_id AND
    u.id = w.user_id
SELECT
    m.id         AS mid,
    w.message_id AS w_mid,
    w.user_id    AS w_uid,
    u.id         AS u_id;
```

Figure 6.27: Example of a gSQL^{++} query being lowered into a *nearly* equivalent SQL^{++} query.

default) to give Graphix users the option to factor out as many `JOIN`s as possible at the cost of a potentially different result set (for the aforementioned edge cases above).

Figure 6.26 and Figure 6.27 illustrates gSQL^{++} lowering where the initial gSQL^{++} query can be directly lowered into an *equivalent* SQL^{++} query. Figure 6.28 depicts another gSQL^{++} lowering example for a navigational query that has no SQL^{++} equivalent. To represent Line 2 and Line 3 of the gSQL^{++} query in Figure 6.28, the gSQL^{++} AST rewriter divides Line 3 into two parts: i) a non-recursive "anchor" subquery used to initialize navigation, and ii) a *recursive* subquery that references the vertex and evaluates each edge hop. The bottom left of Figure 6.28 represents the anchor subquery translation, where Graphix builds zero-length paths using the definition body for the starting vertex pattern. This zero-length path, `ro`, is constructed using the private Graphix function `CREATE_PATH`. In the `SELECT` clause of the anchor subquery, Graphix logically yields (a) the starting vertex `m1 AS start_vertex`, (b) the initial path `ro AS this_path`, and (c) the working vertex `m1 AS this_vertex` to the subquery on the right.

The bottom right of Figure 6.28 represents the recursive subquery translation, where Graphix `JOIN`s the previous iteration (depicted as `<PREV_ITER> prev`), the edge pattern (bound to the variable `roe` here), and the next vertex pattern (bound to the variable `nm` here). To "grow" the path, Graphix uses the private Graphix function `APPEND_TO_PATH`, which appends the edge `roe` and the next vertex `nm` to the previous path `prev.this_path`. In the `SELECT` clause of the recursive subquery, Graphix logically yields (a) the starting vertex `prev.start_vertex` (the exact same vertex bound in the anchor subquery), (b) the extended path `ro AS this_path`, and (c) the vertex Graphix just traversed to, `nm AS this_vertex`, to the next iteration of the same recursive subquery. Note the application of the same AST rewriter techniques for non-navigational gSQL^{++} queries to anchor and recursive subqueries in Figure 6.28 (e.g., representing the `REPLY_OF` edge as an expression from `prev`). Navigational ASTs benefit

```
FROM
    GRAPH SocialNetworkGraph
        (m1:Message)-[ro:REPLY_OF+]->(m2:Message)
SELECT *;
```

$\Downarrow$ (anchor subquery)

```
FROM
    Messages m1
LET
    ro = CREATE_PATH(m1)
WHERE
    NOT m1.is_draft
SELECT
    m1 AS start_vertex,
    ro AS this_path,
    m1 AS this_vertex;
```

$\Downarrow$ (recursive subquery)

```
FROM
    <PREV_ITER> prev,
    Messages nm
LET
    pn = prev.this_vertex,
    roe = {
        "source_id": pn.id,
        "dest_id":
            pn.reply_id,
        "posted_on":
            pn.posted_on
    },
    ro = APPEND_TO_PATH(
        prev.this_path,
        roe,
        nm
    )
WHERE
    prev.m.id = roe.source_id
    AND roe.dest_id = nm.id
SELECT
    prev.start_vertex,
    ro AS this_path,
    nm AS this_vertex;
```

Figure 6.28: gSQL^{++} query to find all messages `m1` and their reply chains `ro` to other messages `m2`, followed by two "SQL^{++}-like" subqueries that represent the translation of the path pattern `ro`. The non-SQL^{++} features are surrounded by angle brackets.

```
FROM
    <LOWERED_PATH_PTRN> p
LET
    m1 = p.start_vertex,
    ro = p.this_path,
    m2 = p.this_vertex
SELECT *;
```

Listing 6.6: "SQL^{++}-like" translation of the gSQL^{++} query in Figure 6.28. The non-SQL^{++} features are surrounded by angle brackets.

from the same AST rewrites as ASTs generated from SQL^{++} queries by considering the non-iterative computation inside (and outside) the loop.

Given the anchor and recursive ASTs of a `PathPattern` expression, Graphix encloses both ASTs in a special "black-box" AST node that exposes a) the variable bound to the source vertex pattern (`p.start_vertex`), b) the variable bound to the path pattern (`p.this_path`), and c) the variable bound to the last visited vertex pattern (`p.this_vertex`). Listing 6.6 depicts how the gSQL^{++} query in Figure 6.28 is translated in a manner that exposes `m1`, `ro`, and `m2` from the "black-box" AST node <`LOWERED_PATH_PTRN`> (bound to the variable `p`). Similar to the second gSQL^{++} lowering example of Figure 6.27, Listing 6.6 demonstrates how Graphix localize the path pattern translation. The translated query encapsulates the anchor and recursive subqueries, allowing the downstream clauses (e.g., the `GROUP BY`, the `SELECT`, etc. . .) to treat `m1`, `ro`, and `m2` like any other SQL^{++} variable.

6.4 Algebricks Query Optimizer

Algebricks is a data-model agnostic query optimizer used by AsterixDB. With respect to the AsterixDB stack, Algebricks sits between the previously discussed AST rewriter and the Hyracks runtime engine. In this section we will bridge the two layers: we will describe how the constructs from a gSQL^{++} AST are further optimized (beyond AST rewrites) and later assembled into a Hyracks job, ultimately concluding the description of the Graphix implementation.

We begin with an overview of Algebricks. Similar to Hyracks, Algebricks expects a directed *acyclic* graph of operators as input. Given an Algebricks plan and a set of Algebricks rules, Algebricks (specifically, an Algebricks rule controller) will invoke each rule for each operator in the plan. Algebricks rule developers implement the `IAlgebraicRewriteRule` interface, which is composed of two methods:

1. `rewritePre(plan, context)`, which is invoked for some subgraph of the entire query plan during its pre-order traversal; and
2. `rewritePost(plan, context)`, which is invoked for some subgraph of the entire query plan during its post-order traversal.

We contrast the `IAlgebraicRewriteRule` interface above with the `IFrameWriter` interface that Hyracks operator developers have to implement. By design, Hyracks operators consider frames in isolation (without the need to consider other operators). On the other hand, Algebricks rules works in units of *query plans*. The "DAG" assumption of a query plan is present in ∼150 Algebricks rules implemented in AsterixDB. Furthermore, many of these existing rules (e.g., predicate pushdown, secondary index insertion, etc. . .) are rules that a recursive computation can benefit from. Our reasons to keep Algebricks query plans acyclic are twofold: (i) to avoid the costly engineering effort involved in assessing and rewriting these rules (and imposing new requirements for future Algebricks rules), and (ii) to apply existing (and future) rules to our recursive computation.

Similar to our gSQL^{++} AST rewriting section, Graphix divides a cyclic computation into two parts: 1) an *anchor* member (a computation that is performed once), and 2) a *recursive* member (a computation that is performed until a least fixed-point is reached). We introduce four new Algebricks operators to model the ASTs found in Subsection 6.3.2:

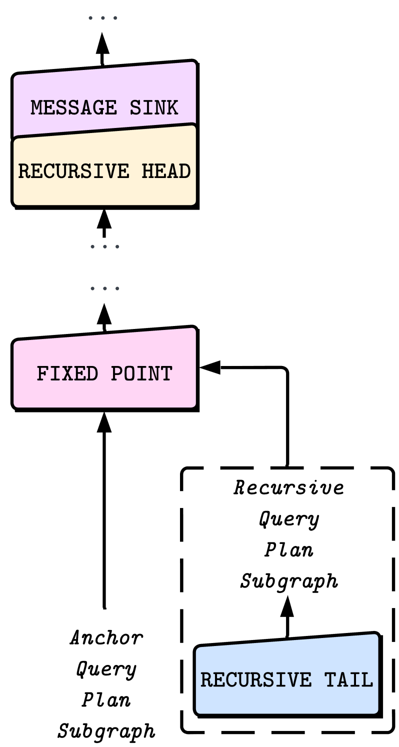


Figure 6.29: Query plan depicting the use of four new Algebricks operators to realize a recursive query.

1. `MESSAGE SINK`, used to generate the corresponding `MESSAGE SINK` Hyracks operator.
2. `FIXED POINT`, used to logically the model the union of the anchor member and the recursive member.
3. `RECURSIVE HEAD`, used to logically forward tuples[††] to the `RECURSIVE TAIL` operator.
4. `RECURSIVE TAIL`, used to logically provide tuples for the recursive member to use.

All four of these operators are used in the gSQL^{++} AST translation to an Algebricks query plan. We depict how each operator is assembled in an Algebricks query plan in Figure 6.29. We highlight the absence of cycles, allowing existing Algebricks rules to retain their "DAG" assumption of a query plan. Once a plan for a navigational query is optimized, Algebricks will realize the cycle of Hyracks operators by replacing the `RECURSIVE HEAD` operator with a

[††]In the context of Algebricks, each operator provides a set of logical properties (e.g., schema, used variables, etc. . .). The `RECURSIVE HEAD` operator forwards the logical properties it receives from its upstream operator to the `RECURSIVE TAIL` operator.

REPLICATE Hyracks operator that feeds into the downstream operator of the RECURSIVE TAIL operator.

We conclude this section by highlighting three specific rules for navigational queries: (i) the rule to recognize the applicability of an index for edge traversal, (ii) the rule to minimize the information in a path during runtime, and (iii) the rule to determine the path finding problem class (e.g., reachability, shortest path, cheapest path).

Edge Traversal Rule As mentioned in Subsection 6.2.8, the default `JOIN` method used to perform an edge hop during navigation is PBJ (persistent-build `JOIN`). For navigational queries that only require access to a handful of vertices, however, performing an INLJ (index nested loop `JOIN`) is more appropriate. The *Edge Traversal Rule* is an "adapter" rule for AsterixDB's existing *Join Access Method Rule*, which determines the applicability of an index for use in evaluating a `JOIN`. In order to use INLJ for edge hops, Graphix additionally requires that a) the path pattern is annotated with the `indexnl` hint, or b) the `graphix.evaluation.prefer-indexnl` compiler flag is raised. Potential future work involves leveraging the cost based optimizer of AsterixDB (realized as an Algebricks rule) to automate this decision.

Path Minimization Rule For queries containing path patterns whose bound variable is not used downstream, the cost associated with maintaining path objects at runtime can be reduced. The *Path Minimization Rule* is used to recognize when path objects are not required outside of path navigation. If the variable bound to a path pattern is not used beyond the `RECURSIVE HEAD` operator, then the *Path Minimization Rule* will a) represent vertices in a path object using the vertex's primary key, and b) represent edges in a path object using the edge's source and destination keys. Potential future work involves leveraging the `TOP K` operator to avoid the maintenance of a runtime path altogether.

Navigational Problem Class Rule The initial Algebricks plan for a navigational query will always enumerate all paths. The *Navigational Problem Class Rule* is used to recognize when a reachability or shortest/cheapest path query is being expressed. This rule starts by locating a `FIXED POINT` operator and either a) a `DISTINCT` operator, or b) a `GROUP BY` operator. If a `DISTINCT` operator is found, the following conditions must hold for the *Navigational Problem Class Rule* to insert a `TOP K` operator into the recursive query plan subgraph:

1. The `DISTINCT` key must contain functionally dependent variables attached to the incident vertex patterns *but not* a functionally dependent variable attached to the path pattern. For example, the path pattern expression `(u1:User)-[k:KNOWS+]->(u2:User)` must have a `DISTINCT` clause like `DISTINCT u1,u2` *but not* `DISTINCT u1,k,u2`.

If a `GROUP BY` operator is found, the following conditions must hold for the *Navigational Problem Class Rule* to insert a `TOP K` operator into the recursive query plan subgraph:

1. The `GROUP BY` key must contain functionally dependent variables attached to the incident vertex patterns *but not* a functionally dependent variable attached to the path pattern. For example, the path pattern expression `(u1:User)-[k:KNOWS+]->(u2:User)` must have a `GROUP BY` clause like `u1,u2` *but not* `GROUP BY u1,k,u2`.
2. If there exists no subquery operating on the variable bound to the group, then Graphix changes the problem class from "all paths" to "reachability".
3. If there exists a subquery operating on the group variable, it must contain (i) a `LIMIT` clause whose value is a constant integer (defining the k for `TOP K`), and (ii) an `ORDER BY` clause whose expression is functionally dependent on the path pattern variable and whose expression is guaranteed to be non-negative. The `ORDER BY` expression defines the weight variable used for the `TOP K` operator. Graphix will then change the problem class from "reachability" to "cheapest path".

Additional future work with respect to Graphix + Algebricks is to incorporate more rules that take loops into account (e.g., loop unrolling, magic sets) [58, 61, 34, 19]. In the context of planning path queries specifically, we point to work from Yakovets, Godfrey, and Gryz [74].

Chapter 7

Evaluation

In this chapter, we describe a set of experiments that measure the end-to-end query performance of Graphix against a leading graph database, Neo4j.* We reiterate that Graphix is meant to operate on existing JSON data with latent graph structure. Graphix was not designed with the sole purpose of executing graph queries in the smallest amount of time (although we do observe some competitive performance for many queries in this chapter). Nonetheless, we report our findings in this chapter.

7.1 Experimental Setup

All experiments in this chapter are based on the LDBC social network database (abbrv. LDBC SNB) [6], which is summarized visually in Figure 7.1. To model this database as a Graphix graph, we first modeled the database as a collection of (AsterixDB) documents as an application would do. This approach more closely reflects the intended use case of Graphix: to *augment* (not replace) existing AsterixDB instances. Our AsterixDB representation of

*TigerGraph, a distributed graph database, was initially also considered for comparison, but their free "community" edition is limited to 50 GB graphs on a single node.

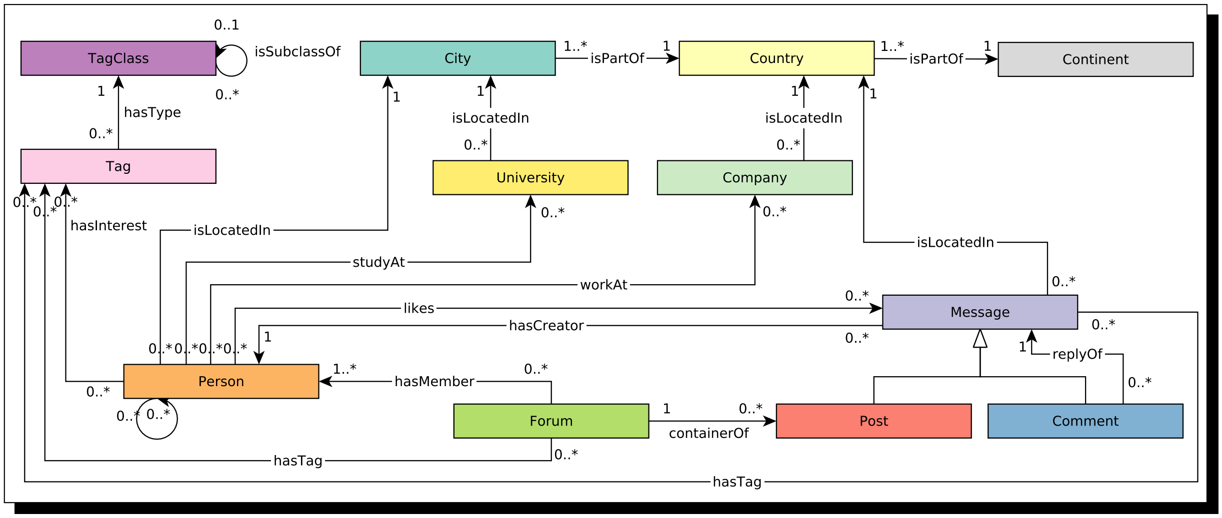


Figure 7.1: Entities and their relationships in the LDBC social network database (copied from [6]).

the Figure 7.1 database consists of 14 datasets that leverage nested objects and arrays when appropriate. In particular, we highlight the following non-1NF features of our AsterixDB datasets:

1. The "hasTag" M:N relationship between a `Forum` entity and a `Tag` entity is folded into `Forum` side via an array of `Tag` (non-enforced) foreign key references. This same folding is performed for the "hasTag" M:N relationship between a `Message` entity and a `Tag`.
2. The "workAt" M:N relationship between a `Company` entity and a `Person` entity is folded into the `Person` side via an array of objects. Each object in this array contains a `Company` (non-enforced) foreign key reference and an attribute about the relationship itself (e.g., the year the person joined the company). This same kind of folding is performed for the "studyAt" M:N relationship.
3. The `Post` and `Comment` entities are both captured in the `Messages` dataset. To distinguish between `Post` and `Comment` documents, a Boolean flag `isPost` is included in each `Message` document.

Each AsterixDB dataset was declared using the primary key given in the LDBC benchmark specification. Once the JSON documents representing the social network were loaded into

AsterixDB, the Graphix graph was then defined using a `CREATE GRAPH` statement. To give Graphix the ability to evaluate edge hops using an index-nested-loop-`JOIN`s approach, secondary indexes were created for each foreign key reference. We give the full set of DDLs used for evaluation in Section A.1.

The results reported in this chapter used AWS EC2 `t2.2xlarge` instances, each with (i) 32 GB of memory, (ii) 8 vCPUs, and (iii) EBS `gp3` SSDs at 3000 IOPS. We compared a Neo4j instance (version 5.13) on a single AWS instance against Graphix clusters of various sizes ($n = \{1,\ 2,\ 4,\ 8,\ 16,\ 32\}$). With respect to data itself, 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. Our evaluation consists of two LDBC scale factors:

SF=1 raw data size ≃1 GB, 3.7 million vertices, 10.2 million edges — this scale factor was used to evaluate the archetypal *in-core* scenario, where a single machine can fit the entire graph into memory; and

SF=100 raw data size ≃100 GB, 312.0 million vertices, 1.1 billion edges — this scale factor was used to evaluate the archetypal *out-of-core* scenario, where a single machine cannot fit the entire graph into memory (and consequently must work with its disk and/or other machines).

The workload for our experiments is composed of read-only queries from: a) the LDBC interactive workload [23], which is a "graph-based parallel" to the TPC-C benchmark for relational-based on-line transaction processing, and b) the LDBC business intelligence workload [65], which is a "graph-based parallel" to the TPC-H benchmark for relational analytics. Queries were issued to each system remotely using a separate AWS node in the same region, with our results reporting the end-to-end response time (i.e., starting from the time the query was issued to the time all results were received). The driver used to issue Neo4j queries was written in Python and uses the `neo4j.GraphDatabase.driver` class from the `neo4j=5.4.0` package. The driver used to issue Graphix queries was also written in Python and uses the `requests=2.28.2` package to issue POST requests to the query service REST API endpoint of the Graphix cluster controller. All Neo4j queries were directly copied from the official LDBC SNB GitHub repositories. All gSQL^{++} queries are given in Section A.2. All artifacts used for the experiments in this paper can be found at: https://github.com/graphix-asterixdb/benchmark.

7.2 Operational `IS-X` Queries

In this section, we detail our first set of experiments comparing Neo4j against Graphix clusters of varying size for the LDBC SNB "interactive short" queries (abbrv. `IS-X`). All `IS-X` queries anchor on a specific vertex (either a `Message` or `Person`) via its primary key and traverse a small portion of the graph from its anchor. Consequently, we observed that `IS-X` queries executed faster than queries in the other two query suites (i.e., the "interactive complex" and "business intelligence" suite). Each `IS-X` query was given a deadline of 3 minutes (after which the query was terminated), though *nearly* all results are in the sub-second range.

Figure 7.2 and Figure 7.3 both illustrate several plots comparing the median execution time of all `IS-X` queries for scale factors `SF=1` and `SF-100` respectively. The execution times for Figure 7.2 and Figure 7.3 are also given in Table 7.1 and Table 7.2 respectively. Starting with Figure 7.2, we observe that Neo4j consistently ran faster than Graphix for all `IS-X` runs at `SF=1`. These results are not surprising — Neo4j is better equipped for handling these types of low-latency in-core graph queries when compared to Graphix for two reasons:

Query Plan Cache Neo4j caches its query plans to avoid compiling the same query again, while Graphix has no such cache (although an AsterixDB query plan cache is actively

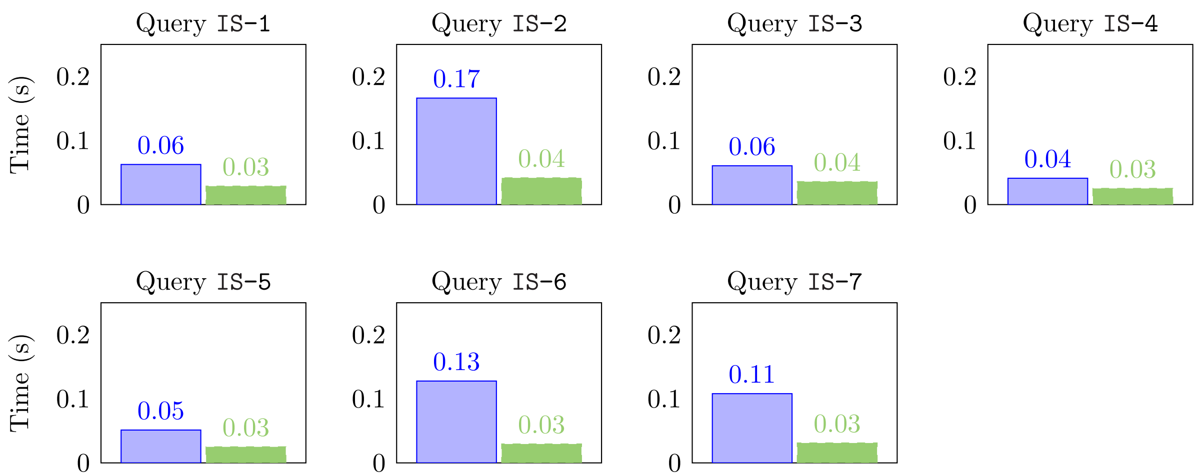


Figure 7.2: Several plots showing a Graphix cluster of n=1 (in blue) against a Neo4j instance (in green) for the `IS-X` query suite at `SF=1`.

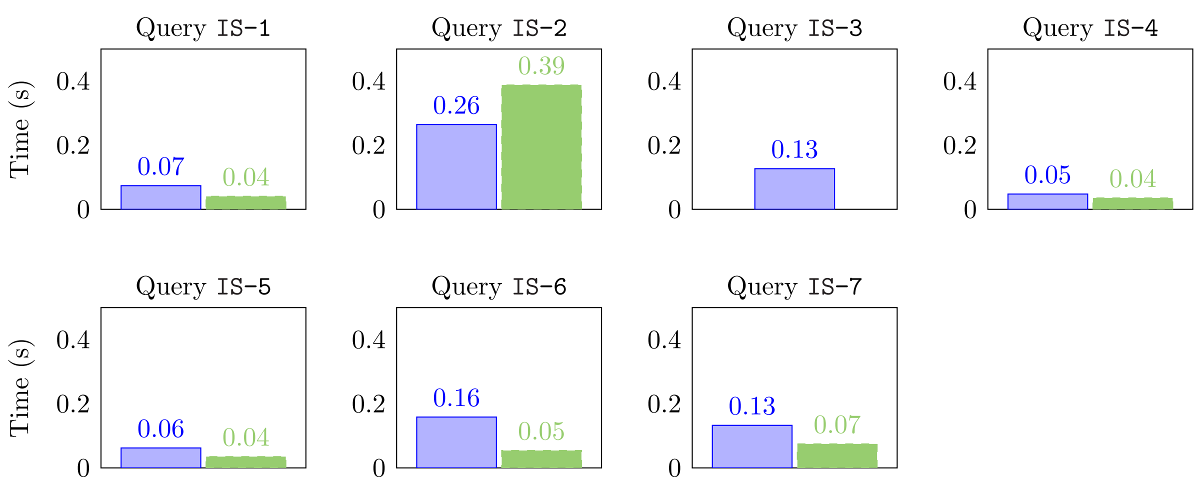


Figure 7.3: Several plots showing a Graphix cluster of n=1 (in blue) against a Neo4j instance (in green) for the `IS-X` query suite at `SF=100`. Neo4j did not consistently finish query `IS-3` in under 3 minutes.

being developed). Even when Graphix is given the same query twice in a row, it will currently needlessly repeat the AST rewriting, the Algebricks optimization, and the Hyracks job distribution.

Network Protocol Neo4j uses a custom binary network protocol (Bolt [48]) to communicate between a client and a server, while Graphix only exposes a REST API (which uses HTTP between the client and server). Using a specialized network protocol here allows Neo4j to reduce the number of bytes sent between a client and a server.

For these short queries and graph size, we note that parallelism is not beneficial. Increasing the size of a Graphix cluster (as observed in Table 7.1) *increases* the query execution time. We attribute this to i) the overhead of distributing Hyracks stages to every node controller and/or ii) the overhead of collecting the query result from every node controller.[†] Nevertheless, applications that do not require sub 100 ms response times for short-read queries (in the manner of `IS-X`) for in-memory graphs would benefit from using Graphix.

Moving to Figure 7.3, we observe similar Graphix performance for all `IS-X` queries at `SF=100`. Neo4j, however, runs `IS-2` *slower* than Graphix clusters of $n \leq 4$. Furthermore, Neo4j is *not* able to run `IS-3` consistently under 3 minutes. We observed a few runs of Neo4j executing `IS-3` under the 3 minute timeout, though Neo4j was unable to consistently run below this timeout for our experiments. For `IS-3` in particular, we ascribe this Neo4j inconsistency to the `-[:KNOWS]->` edge being traversed. The `KNOWS`-edge degree distribution of `(:Person)` vertices follow the power law, where a few vertices possess significantly more `KNOWS` edges than the rest of the population. Both Graphix and Neo4j evaluate edges for `IS-X` queries using an index-nested-loop-`JOIN`. Graphix, however, places a `SORT` operator on the `JOIN` key before the `PIDX SEARCH` operator to minimize the total number of index lookups. Neo4j does

[†]The impact of the `FIXED POINT` operator is most likely minimal here (for the `IS-X` queries) as only `IS-2` and `IS-6` are recursive. Furthermore, the edges that `IS-2` and `IS-6` traverse (i.e., the `REPLY_OF` edges) do not branch.

not perform any such sort, resulting in more random I/O for high degree vertices. For short read queries, Graphix is able to achieve consistent performance for small and large graphs.

7.3 Operational `IC-X` Queries

In this section, we detail our second set of experiments comparing Neo4j against Graphix clusters of varying size for the LDBC SNB "interactive complex" queries (abbrv. `IC-X`). The `IC-X` queries differ from the `IS-X` queries in that `IC-X` queries traverse a larger portion of the graph. All `IC-X` queries specify a) a starting (anchor) `Person` vertex via its primary key, and b) a small number of "destination" vertices. The latter is generally specified for `IC-X` queries using some low selectivity predicate on the destination vertex itself (e.g., where the destination vertex was created between some date range), though queries `IC-13` and `IC-14` explicitly specify a single destination vertex via its primary key. We observed that `IC-X` queries typically took longer to complete than the `IS-X` suite of queries but executed faster than the analytical queries of the "business intelligence" suite (in the next section). For our experiments, we gave each `IC-X` query a deadline of 30 minutes (after which, the query would be terminated).

Figure 7.4 and Figure 7.5 both illustrate several plots comparing the median execution time of all `IC-X` queries for scale factors `SF=1` and `SF-100`, respectively, against the size of a Graphix cluster. Neo4j does not scale horizontally, therefore its results in all plots are depicted as a straight line. Queries `IC-2`, `IC-7`, and `IC-8` are only displayed at $n = 1$ as they do not benefit from parallelism. The execution times for Figure 7.4 and Figure 7.5 are also given in Table 7.1 and Table 7.2 respectively. For Figure 7.4, we draw similar conclusions for the `IC-X` query suite (compared to the `IS-X` query suite): Neo4j outperforms Graphix for low latency queries over small (in-memory) graphs. The exception for Figure 7.4 is query `IC-12`, which Neo4j does not consistently execute under the 30 minute timeout for both

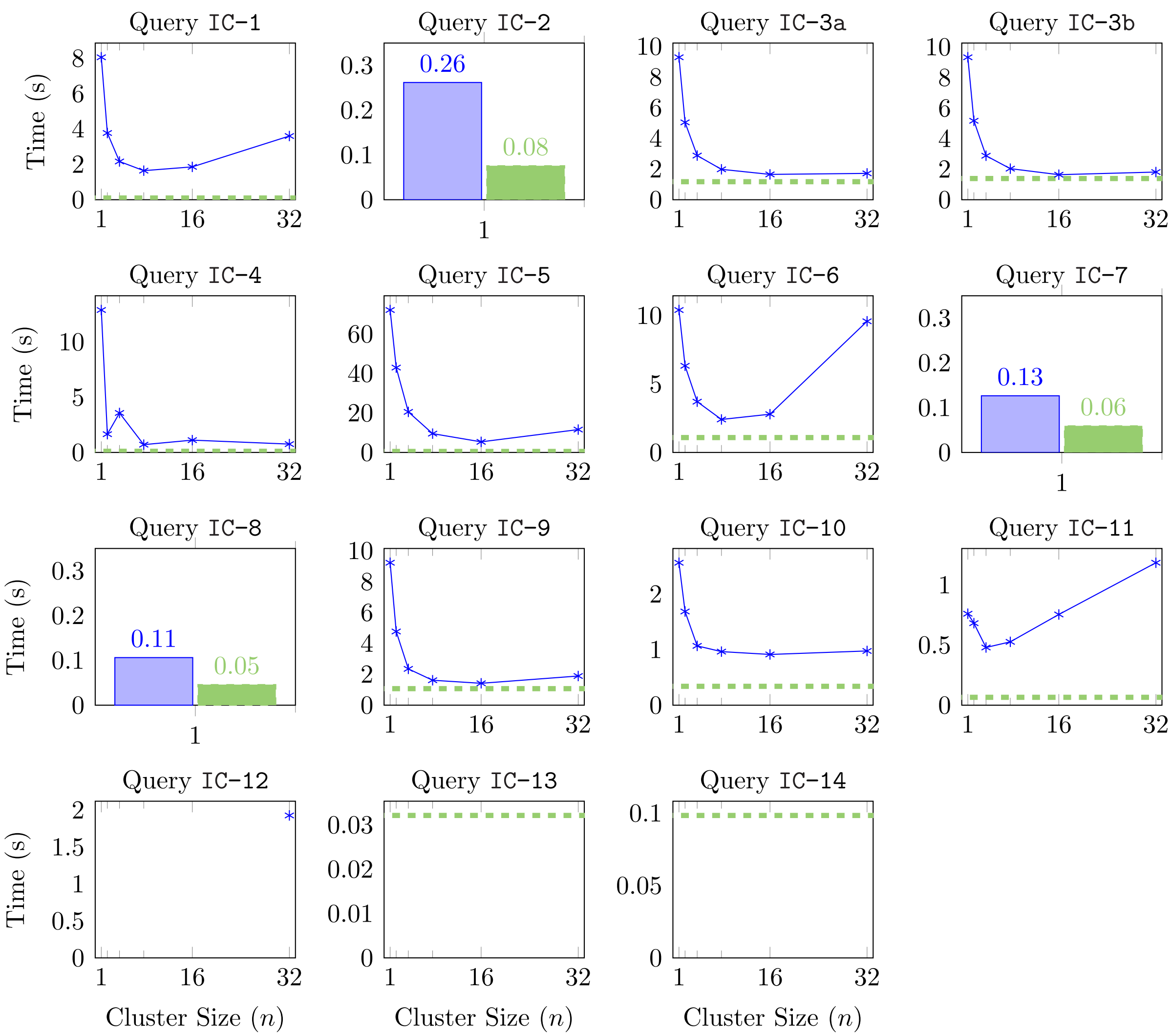


Figure 7.4: Several plots showing a Graphix cluster of varying size (in blue) against a Neo4j instance (in green) for the `IC-X` query suite at `SF=1`. Queries `IC-2`, `IC-7`, and `IC-8` are shown at $n = 1$. Neo4j and Graphix (for $n < 32$) were not able to consistently execute query `IC-12` underneath the 30 minute timeout. Graphix was unable to execute `IC-13` and `IC-14` in under 30 minutes.

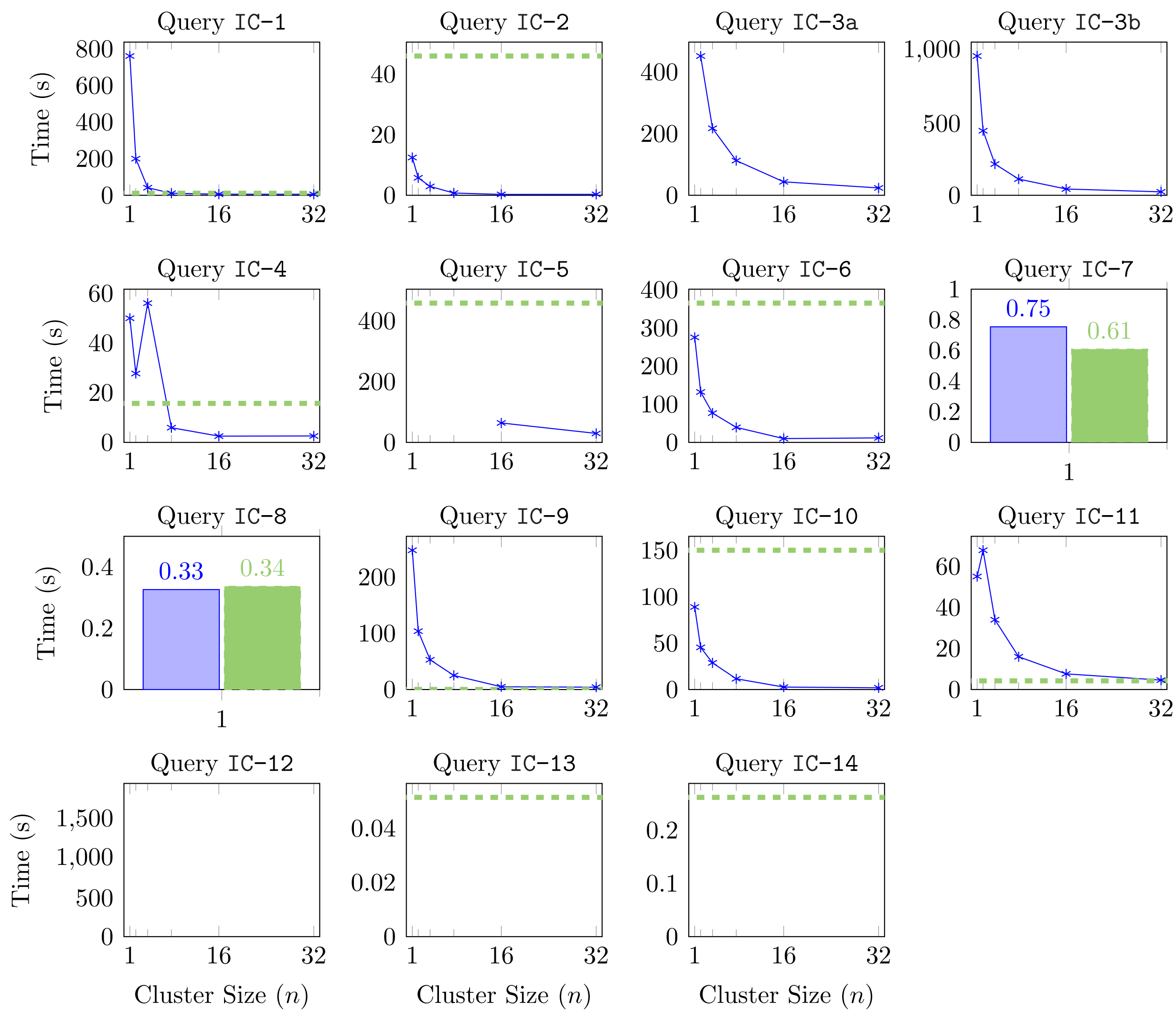


Figure 7.5: Several plots showing a Graphix cluster of varying size (in blue) against a Neo4j instance (in green) for the `IC-X` query suite at `SF=100`. Queries `IC-7` and `IC-8` are shown at $n = 1$. Neo4j and Graphix were not able to execute query `IC-12` underneath the 30 minute timeout. Graphix was unable to execute `IC-13` and `IC-14` in under 30 minutes. Neo4j was unable to finish queries `IC-3a` and `IC-3b` in under 30 minutes.

`SF=1` and `SF=100`. Only at $n = 32$ is Graphix able to consistently execute this query. Both Neo4j and Graphix in our evaluation start from the anchor `(:Person)` vertex and walk from there, though LDBC suggests that walking backwards from the destination might be a better evaluation strategy for some `(:Person)` vertices [6]. Increasing Graphix to $n = 32$ illustrates a "brute-force" approach to not-as-(consistently)-efficient evaluation strategies that Neo4j cannot utilize: the approach of adding more machines.

Graphix at larger values of n generally performs better than Graphix at smaller values of n if the query executes for longer than one second (queries `IC-3a`, `IC-3b`, `IC-5`, `IC-9`, and `IC-10`). Queries `IC-1`, `IC-4`, and `IC-6` illustrate data points that warrant further investigation. Queries `IC-1` and `IC-6` show an initial negative correlation of execution time and n, but a positive correlation for $n > 4$. We do not observe this same upward trend at `SF=100`. Query `IC-4` demonstrates a strange increase in execution time at $n = 4$ (relative to the previous data point, $n = 2$), a pattern that is also observed for `SF=100`.

We now bring our attention to queries `IC-13` and `IC-14`, two queries for which Graphix was unable to record *any* runs under the 30 minute timeout. Neo4j is able to significantly outperform Graphix for these queries due to the bidirectional BFS (breadth-first-search) approach that Neo4j takes to evaluate shortest paths:

Bidirectional BFS At a high level, Graphix evaluates all navigational queries using BFS: given some starting set of vertices, Graphix branches out from the starting vertices in parallel until the destination vertices are reached. For query plans that include the `TOP K` operator (see Subsection 6.2.8), Graphix can (loosely) bound the length of the paths it enumerates; however, power law degree distributions (e.g., the `KNOWS` edges connecting `(:Person)` vertices together) imply the existence of significantly more shorter paths than longer paths [16]. If Graphix is given an explicit destination vertex (e.g., queries `IC-13` and `IC-14`), Graphix does not leverage this information to potentially avoid having to traverse the massive number of shorter paths before reaching the

Query	Neo4j (n=1)	Graphix (n=1)	Graphix (n=2)	Graphix (n=4)	Graphix (n=8)	Graphix (n=16)	Graphix (n=32)
`IS-1`	28.7 ms	62.6 ms	62.5 ms	61.9 ms	75.7 ms	80.0 ms	94.3 ms
`IS-2`	41.7 ms	166.2 ms	183.2 ms	269.0 ms	388.5 ms	471.5 ms	605.6 ms
`IS-3`	35.8 ms	60.6 ms	76.2 ms	119.5 ms	183.3 ms	231.0 ms	299.3 ms
`IS-4`	25.2 ms	41.1 ms	42.1 ms	40.5 ms	52.1 ms	57.1 ms	68.0 ms
`IS-5`	25.0 ms	51.3 ms	53.4 ms	65.1 ms	90.6 ms	106.9 ms	131.2 ms
`IS-6`	29.5 ms	127.8 ms	135.9 ms	184.6 ms	266.5 ms	335.2 ms	464.9 ms
`IS-7`	31.0 ms	108.1 ms	129.8 ms	216.0 ms	332.7 ms	418.3 ms	495.2 ms
`IC-1`	98.0 ms	8.1 s	3.8 s	2.2 s	1.6 s	1.9 s	3.6 s
`IC-2`	75.9 ms	261.9 ms	185.8 ms	181.1 ms	216.4 ms	243.5 ms	279.4 ms
`IC-3a`	1.2 s	9.3 s	5.0 s	2.9 s	2.0 s	1.7 s	1.7 s
`IC-3b`	1.4 s	9.3 s	5.2 s	2.9 s	2.0 s	1.6 s	1.8 s
`IC-4`	104.8 ms	12.9 s	1.7 s	3.6 s	706.6 ms	1.1 s	756.3 ms
`IC-5`	504.1 ms	72.2 s	43.0 s	20.5 s	9.5 s	5.4 s	11.6 s
`IC-6`	1.1 s	10.4 s	6.3 s	3.7 s	2.4 s	2.8 s	9.6 s
`IC-7`	59.5 ms	126.6 ms	118.7 ms	169.0 ms	259.9 ms	317.2 ms	386.3 ms
`IC-8`	46.0 ms	106.3 ms	111.9 ms	150.4 ms	220.6 ms	286.7 ms	449.0 ms
`IC-9`	1.1 s	9.2 s	4.8 s	2.3 s	1.6 s	1.4 s	1.9 s
`IC-10`	339.8 ms	2.6 s	1.7 s	1.1 s	964.7 ms	913.7 ms	976.6 ms
`IC-11`	65.5 ms	760.9 ms	682.1 ms	480.0 ms	527.0 ms	754.6 ms	1.2 s
`IC-12`	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	1.9 s
`IC-13`	32.1 ms	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)
`IC-14`	98.4 ms	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)

Table 7.1: Table comparing the median execution times of `IS-X` and `IC-X` queries at scale factor `SF=1` for Neo4j and different Graphix cluster configurations. Neo4j and Graphix (for $n < 32$) were not able to consistently execute query `IC-12` underneath the 30 minute timeout. Graphix was unable to execute `IC-13` and `IC-14` in under 30 minutes.

Query	Neo4j (n=1)	Graphix (n=1)	Graphix (n=2)	Graphix (n=4)	Graphix (n=8)	Graphix (n=16)	Graphix (n=32)
`IS-1`	40.4 ms	73.8 ms	75.7 ms	72.8 ms	77.9 ms	83.3 ms	93.3 ms
`IS-2`	387.7 ms	264.4 ms	304.3 ms	341.4 ms	399.0 ms	475.2 ms	610.1 ms
`IS-3`	>3 min (`T/O`)	126.7 ms	111.9 ms	145.0 ms	196.5 ms	242.9 ms	312.3 ms
`IS-4`	35.5 ms	47.6 ms	49.8 ms	48.6 ms	52.8 ms	58.2 ms	72.2 ms
`IS-5`	35.1 ms	62.2 ms	62.6 ms	75.0 ms	93.5 ms	111.2 ms	131.1 ms
`IS-6`	54.2 ms	158.6 ms	167.3 ms	208.5 ms	281.0 ms	347.8 ms	464.4 ms
`IS-7`	74.8 ms	132.7 ms	152.3 ms	231.0 ms	335.8 ms	417.5 ms	496.0 ms
`IC-1`	11.2 s	760.8 s	200.0 s	41.4 s	11.3 s	5.9 s	5.5 s
`IC-2`	46.0 s	12.5 s	5.8 s	2.9 s	743.9 ms	295.2 ms	312.9 ms
`IC-3a`	>30 min (`T/O`)	>30 min (`T/O`)	451.8 s	217.2 s	112.6 s	43.6 s	23.8 s
`IC-3b`	>30 min (`T/O`)	968.5 s	448.9 s	217.2 s	112.8 s	43.7 s	24.1 s
`IC-4`	15.7 s	50.0 s	27.7 s	56.1 s	5.9 s	2.5 s	2.6 s
`IC-5`	458.5 s	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	64.0 s	29.7 s
`IC-6`	365.0 s	275.3 s	132.2 s	76.8 s	39.4 s	10.0 s	11.9 s
`IC-7`	608.6 ms	754.4 ms	466.0 ms	443.1 ms	460.7 ms	334.6 ms	382.8 ms
`IC-8`	336.7 ms	326.2 ms	266.4 ms	312.8 ms	374.3 ms	313.0 ms	451.6 ms
`IC-9`	163.0 ms	247.9 s	103.9 s	52.9 s	25.1 s	4.9 s	4.2 s
`IC-10`	150.2 s	89.0 s	45.4 s	28.7 s	11.6 s	2.7 s	1.9 s
`IC-11`	4.2 s	55.1 s	67.8 s	34.0 s	16.0 s	7.6 s	4.6 s
`IC-12`	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)
`IC-13`	51.4 ms	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)
`IC-14`	261.9 ms	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)	>30 min (`T/O`)

Table 7.2: Table comparing the median execution times of `IS-X` and `IC-X` queries at scale factor `SF=100` for Neo4j and different Graphix cluster configurations. Neo4j did not consistently finish query `IS-3` in under 3 minutes. Neo4j and Graphix were not able to execute query `IC-12` underneath the 30 minute timeout. Graphix was unable to execute `IC-13` and `IC-14` in under 30 minutes. Neo4j was unable to finish queries `IC-3a` and `IC-3b` in under 30 minutes.

destination. Neo4j, on the other hand, is able to perform a *bidirectional* BFS: given a starting vertex and destination vertex, Neo4j walks from both endpoints until the traversals intersect [50]. For queries `IC-13` and `IC-14` in particular, Neo4j is able to perform well using this strategy with both small and large graphs (as we will see next).

Future work with respect to Graphix involves implementing bidirectional BFS in Graphix to handle queries with more than one anchor point.

Figure 7.5 describes our evaluation of the `IC-X` queries for `SF=100`. Queries `IC-7` and `IC-8` are only displayed at $n = 1$ as they do not benefit from parallelism. For all other queries, Graphix benefits significantly from horizontal scaling. Queries `IC-1`, `IC-4`, `IC-9`, and `IC-11` perform worse than Neo4j on Graphix with $n = 1$, but larger clusters of Graphix ($n \geq 8$) are able to either outperform Neo4j (e.g., `IC-1`, `IC-4`) *or* perform on-par with Neo4j (e.g., `IC-9`, `IC-11`). Queries `IC-2`, `IC-6`, and `IC-10` execute *faster* than Graphix at $n = 1$ when compared to Neo4j, with Graphix executing these same queries even faster on larger values of n. Neo4j is unable to record any runs for query `IC-3a` and is unable to consistently execute `IC-3b` under the 30 minute timeout. Graphix, on the other hand, is able to execute `IC-3a` and `IC-3b` consistently in under 30 minutes *and* is able to achieve better performance as n increases. For all queries except `IC-8` (which already executes in sub-second time), `IC-12`, `IC-13`, and `IC-14`, Graphix is able to leverage parallel processing to lower the execution time to under one minute.

We will now address the `IC-X` queries where Graphix does not perform favorably for `SF=100`: `IC-5` (for $n < 16$), `IC-12`, `IC-13`, and `IC-14`. Graphix was unable to consistently record runs underneath 30 minutes for all values of n for query `IC-12`, and was unable to record any runs underneath 30 minutes for queries `IC-13` and `IC-14`. Graphix at $n < 16$ was not able to record any runs underneath 30 minutes for query `IC-5`. Graphix at $n \geq 16$ is not only able to execute query `IC-5`, but executes it $7\times$ faster at $n = 16$ and $15\times$ faster at $n = 32$ as compared to Neo4j. To execute queries `IC-5` and `IC-12` for Big Data, choosing the correct

JOIN tree and physical operators is essential. Integrating AsterixDB's cost based optimizer with Graphix would help generate query plans that are better suited to each individual query. We observe that Neo4j executes queries IC-13 and IC-14 in sub-second time in spite of the larger data. To execute IC-13 and IC-14 for Big Data in Graphix, again, would involve implementing some form of bidirectional BFS.

7.4 Analytical BI-X Queries

In this section, we detail our third and final set of experiments, comparing Neo4j against Graphix clusters of varying size for the LDBC social network benchmark "business intelligence" queries (abbrv. BI-X). In contrast to the IS-X and IC-X query suites from the previous section, a larger *fraction* of the graph (i.e., not the entire graph) is accessed in the BI-X suite of queries. For our experiment, the BI-X queries were expected to to take the longest to complete (relative to the IS-X and IC-X query suites). The deadline for each BI-X query was set to 5 hours, after which the query would be terminated.

Figure 7.6 and Figure 7.7 both illustrate several plots comparing the median execution time of all BI-X queries for scale factors SF=1 and SF-100, respectively, against the size of a Graphix cluster. Again, Neo4j does not scale out, so its results in all plots are depicted as a straight line. The execution times for Figure 7.6 and Figure 7.7 are also given in Table 7.3 and Table 7.4 respectively. Queries BI-4 (as well as BI-6, BI-11, and BI-12 for Neo4j only) are not included in our results due to mistakes[‡] that occurred during the benchmarking process. We leave BI-6, BI-11, and BI-12 in for Graphix to characterize how Graphix performs with larger values of n. Query BI-15 was not considered for evaluation due to its requirement for nested recursion (which Graphix does not support at the time of writing). Nevertheless,

[‡]Queries BI-4, BI-6, and BI-12 were accidentally left out of the query set during the benchmarking process. Due to time constraints, we were unable to rerun our experiments again to include these three missing queries.

both `BI-4` and `BI-15` are listed with the rest of the `BI-X` queries in Section A.2 to demonstrate the Graphix query model.

Starting with Figure 7.6, we note that Neo4j does not dominate Graphix for the majority of the `BI-X` queries at `SF=1` (though Neo4j does still outperform Graphix for many queries here at this low scale factor). For queries `BI-1`, `BI-2a`, `BI-10a`, `BI-16a`, `BI-18`, `BI-19a`, and `BI-19b`, Graphix performs worse than Neo4j for low values of n ($4 \lesssim n$). For Graphix clusters with larger n values, Graphix executes these queries faster (or on-par with) Neo4j. Graphix is unable to outperform Neo4j for the remainder of the queries here (for `SF=1`), though Graphix at higher values of n still (generally) perform better than Graphix at lower values of n. One outlier of interest (for Graphix) is query `BI-17`. This query involves two Kleene-closure RPQs (e.g., `-[:REPLY_OF*]->`) and 11 edge patterns, making `BI-17` one of the more difficult queries to evaluate. Graphix at $n = 1$ executes `BI-17` in 35 min and exhibits a significant decrease in running time for $2 \leq n \leq 8$. Beyond $n > 8$, however, Graphix executes `BI-17` slower than Graphix with $n = 2$. One suspect for this large variation is the set of `FIXED POINT` operators used to evaluate each RPQ. Other `BI-X` queries containing an unbounded RPQ (`BI-3`, `BI-9`, `BI-12`, `BI-19{a|b}`) also exhibit a similar increase in execution time for increasingly larger values of n (though still lower in execution time when compared to $n = 1$). Queries `BI-20{a|b}` also specify unbounded RPQs, but instead demonstrate a (nearly) positive correlation up to $n = 4$ and $n = 8$ respectively, after which Graphix executes `BI-20a` and `BI-20b` faster with larger cluster sizes. Future work involves characterizing the effect of n on determining termination for different workloads.

We now move to Figure 7.7, where Graphix at all values of n demonstrates better performance than Neo4j for five of the displayed `BI-X` queries. With respect to the single-worker comparison, Graphix was able to outperform Neo4j for queries `BI-1`, `BI-2a`, `BI-8a`, `BI-9`, and `BI-18`. We observed a few runs of Neo4j executing `BI-2a` and `BI-8a` under the 5 hr timeout, though Neo4j was unable to consistently run below this timeout for our experiments. To explain why Graphix is able to execute many of these queries faster than Neo4j with the same hardware, we will focus on a few key points used by Graphix to evaluate queries `BI-1` and `BI-2`{`a`|`b`}:

Vertex Layout Query `BI-1` involves two scans of all vertices with the `Message` label. Neo4j stores *all* of its vertices in one physical file, meaning that a scan of all `Message` vertices requires also scanning all non-`Message`-labeled vertices [56]. In contrast, the `Message` vertices in Graphix have a 1:1 mapping with a single AsterixDB dataset (and therefore, a single collection partition per worker). As a consequence, Graphix has to scan less vertex data.

Intermediate Result Size Query `BI-2`{`a`|`b`} involves the evaluation of three edges and two aggregations. Neo4j chooses to enumerate all possible mappings in its query plan *before* performing its aggregation. Graphix on the other hand, recognizes that the two aggregations are independent from one another. It is then able to reduce the intermediate result size by interleaving the aggregation and the edge evaluations (leading to better performance).

Hybrid Hash Joins When evaluating an edge, Graphix has several parallel `JOIN` algorithms at its disposal. In the case of query `BI-2`{`a`|`b`}, Graphix evaluates all three edges with hybrid hash `JOIN`s. We contrast this `JOIN` algorithm with Neo4j's edge evaluation approach, which is akin to an index-nested-loop `JOIN`, which is not as performant for `JOIN`s that are not selective [27].

Queries `BI-2b`, `BI-3`, `BI-5` and `BI-7`, and `BI-8b` are another demonstration of how Graphix is able to outperform Neo4j when the cluster size n is increased, despite Neo4j executing these two queries faster than Graphix with $n = 1$. On the other hand, `BI-10`{`a`|`b`} `BI-17`, and `BI-20`{`a`|`b`} are examples of Neo4j executing queries that Graphix (for all values of n) is unable to execute in under 5 hours. Neither Neo4j nor Graphix are able to execute queries `BI-16b` and `BI-19`{`a`|`b`} under 5 hours. We suspect that the majority of the `BI-X` queries that Graphix was unable to execute could be remedied by modifying the query plan. `BI-18` was a query that we used for benchmarking the earlier (and non-recursive version) of Graphix. For this earlier benchmark, `BI-18` was a query that Neo4j executed significantly faster than Graphix for all values of n. After refactoring the query to change the `JOIN` order and supplying query hints to change the physical `JOIN` operator, Graphix was able to beat Neo4j's execution time for `BI-18`. We suspect that a cost-based approach to determining this `JOIN` order and `JOIN` physical operator would help Graphix here not only to execute many of these queries under the 5 hour timeout but also to potentially best Neo4j in execution time.

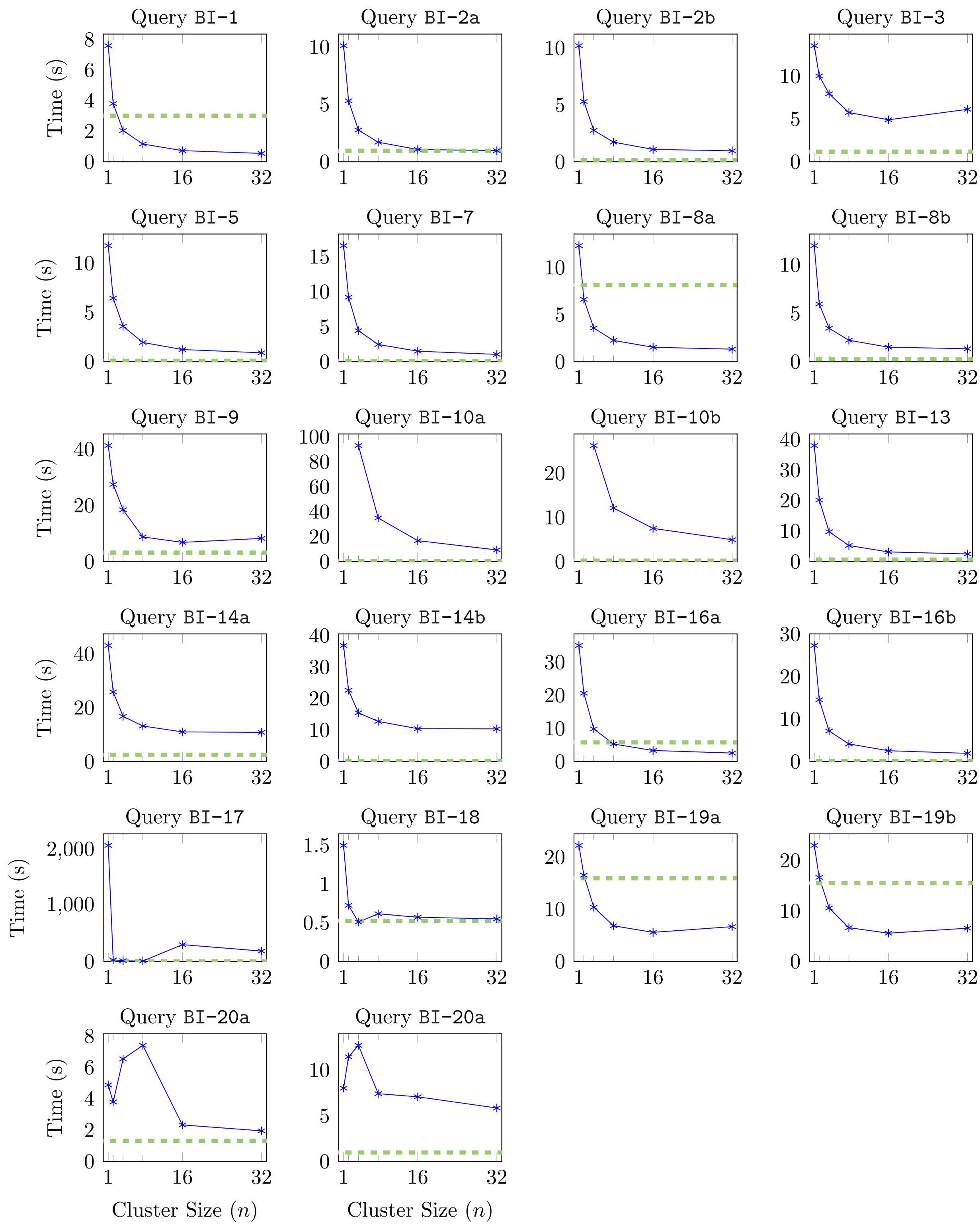


Figure 7.6: Several plots showing a Graphix cluster of varying size (in blue) against a Neo4j instance (in green) for the `BI-X` query suite at `SF=1`. Graphix was unable to consistently execute queries `BI-10a` and `BI-10b` in under 5 hours at $n \leq 2$.

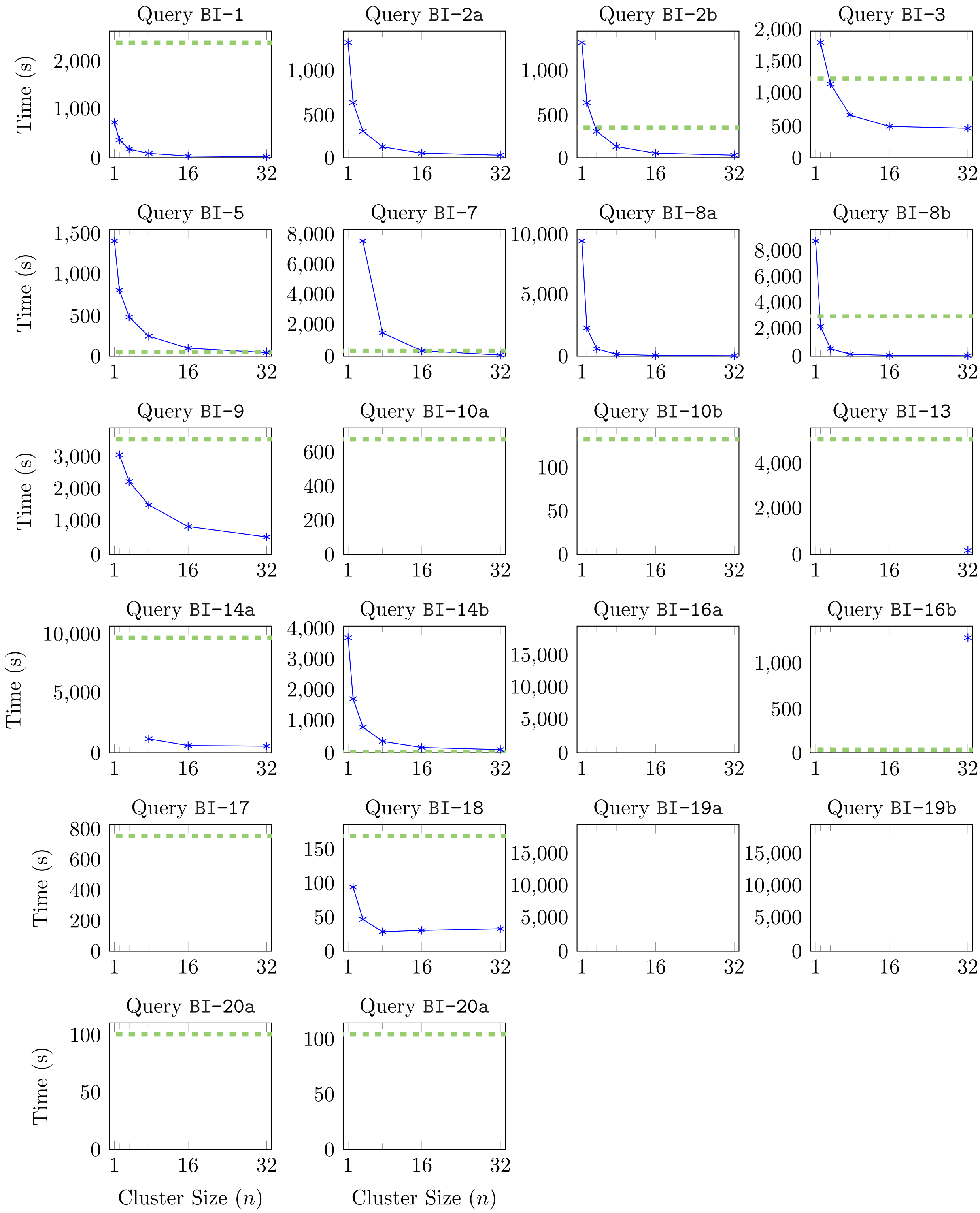


Figure 7.7: Several plots showing a Graphix cluster of varying size (in blue) against a Neo4j instance (in green) for the `BI-X` query suite at `SF=100`. Both Neo4j and Graphix (for all n) were unable to finish queries `BI-16a`, `BI-19a`, and `BI-19b` in under 5 hours. Neo4j was additionally unable to finish queries `BI-2a` and `BI-8a` in under 5 hours. Graphix was additionally unable to finish queries `BI-10{a|b}`, `BI-17`, and `BI-20{a|b}` in under 5 hours.

Query	Neo4j (n=1)	Graphix (n=1)	Graphix (n=2)	Graphix (n=4)	Graphix (n=8)	Graphix (n=16)	Graphix (n=32)
BI-1	3.0 s	7.6 s	3.8 s	2.0 s	1.2 s	727.9 ms	547.3 ms
BI-2a	959.0 ms	10.1 s	5.3 s	2.8 s	1.7 s	1.1 s	970.0 ms
BI-2b	136.8 ms	10.2 s	5.3 s	2.8 s	1.7 s	1.1 s	964.4 ms
BI-3	1.2 s	13.6 s	10.0 s	7.9 s	5.8 s	4.9 s	6.1 s
BI-5	90.7 ms	11.8 s	6.5 s	3.6 s	2.0 s	1.2 s	905.5 ms
BI-6	N/A	28.3 min	16.7 min	499.3 s	307.5 s	145.7 s	77.2 s
BI-7	86.2 ms	16.6 s	9.2 s	4.4 s	2.5 s	1.5 s	1.0 s
BI-8a	8.1 s	12.3 s	6.6 s	3.6 s	2.3 s	1.5 s	1.3 s
BI-8b	259.5 ms	12.0 s	6.0 s	3.5 s	2.2 s	1.5 s	1.3 s
BI-9	3.2 s	41.3 s	27.5 s	18.4 s	8.8 s	6.9 s	8.3 s
BI-10a	366.6 ms	>5 hr (T/O)	>5 hr (T/O)	93.0 s	35.0 s	16.8 s	9.4 s
BI-10b	221.7 ms	>5 hr (T/O)	>5 hr (T/O)	26.3 s	12.1 s	7.5 s	5.0 s
BI-11	N/A	4.4 s	3.8 s	2.2 s	1.4 s	1.2 s	1.2 s
BI-12	N/A	195.8 s	22.7 s	12.6 s	7.1 s	5.9 s	6.5 s
BI-13	676.2 ms	38.0 s	20.2 s	9.8 s	5.3 s	3.2 s	2.5 s
BI-14a	2.5 s	43.1 s	25.9 s	16.8 s	13.2 s	11.0 s	10.8 s
BI-14b	120.9 ms	36.8 s	22.6 s	15.5 s	12.7 s	10.4 s	10.4 s
BI-16a	5.8 s	35.1 s	20.7 s	9.9 s	5.3 s	3.3 s	2.6 s
BI-16b	114.5 ms	27.3 s	14.5 s	7.2 s	4.2 s	2.6 s	1.9 s
BI-17	711.6 ms	35.0 min	28.7 s	14.6 s	7.9 s	305.0 s	188.9 s
BI-18	524.6 ms	1.5 s	725.7 ms	509.8 ms	616.6 ms	571.4 ms	550.1 ms
BI-19a	16.0 s	22.2 s	16.5 s	10.4 s	6.8 s	5.6 s	6.7 s
BI-19b	15.5 s	23.0 s	16.7 s	10.6 s	6.7 s	5.6 s	6.6 s
BI-20a	1.3 s	4.9 s	3.8 s	6.5 s	7.4 s	2.3 s	2.0 s
BI-20b	979.8 ms	8.0 s	11.4 s	12.7 s	7.4 s	7.1 s	5.8 s

Table 7.3: Table comparing the median execution times of BI-X queries at scale factor SF=1 for Neo4j and different Graphix cluster configurations. Neo4j does not have any values for queries BI-6, BI-11, and BI-12. Graphix was unable to consistently execute queries BI-10a and BI-10b in under 5 hours at $n \leq 2$.

Query	Neo4j (n=1)	Graphix (n=1)	Graphix (n=2)	Graphix (n=4)	Graphix (n=8)	Graphix (n=16)	Graphix (n=32)
BI-1	40.6 min	748.6 s	373.8 s	183.1 s	91.4 s	35.6 s	19.9 s
BI-2a	>5 hr (T/O)	22.5 min	649.2 s	311.2 s	129.4 s	53.5 s	29.5 s
BI-2b	355.2 s	22.5 min	647.6 s	309.8 s	133.0 s	52.9 s	29.6 s
BI-3	20.9 min	>5 hr (T/O)	30.4 min	19.5 min	676.2 s	495.9 s	467.2 s
BI-5	48.3 s	23.9 min	820.0 s	485.3 s	247.9 s	99.6 s	43.7 s
BI-6	N/A	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-7	349.4 s	>5 hr (T/O)	>5 hr (T/O)	127.8 min	25.8 min	356.0 s	73.4 s
BI-8a	>5 hr (T/O)	160.2 min	39.4 min	611.5 s	153.8 s	60.8 s	34.8 s
BI-8b	51.2 min	148.2 min	38.5 min	576.1 s	147.2 s	59.2 s	33.6 s
BI-9	60.1 min	>5 hr (T/O)	52.0 min	38.0 min	25.8 min	874.4 s	549.8 s
BI-10a	674.3 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-10b	132.4 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-11	N/A	20.1 min	430.4 s	243.7 s	133.1 s	83.0 s	67.7 s
BI-12	N/A	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-13	86.1 min	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	177.8 s
BI-14a	164.3 min	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	19.9 min	625.8 s	578.0 s
BI-14b	33.6 s	62.6 min	29.4 min	839.5 s	375.0 s	176.6 s	109.1 s
BI-16a	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-16b	38.5 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	21.9 min
BI-17	753.6 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-18	169.0 s	>5 hr (T/O)	94.1 s	46.7 s	28.6 s	30.5 s	33.1 s
BI-19a	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-19b	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-20a	100.4 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)
BI-20b	104.1 s	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)	>5 hr (T/O)

Table 7.4: Table comparing the median execution times of BI-X queries at scale factor SF=100 for Neo4j and different Graphix cluster configurations. Neo4j does not have any values for queries BI-6, BI-11, and BI-12. Both Neo4j and Graphix (for all n) were unable to finish queries BI-16a, BI-19a, and BI-19b in under 5 hours. Neo4j was additionally unable to finish queries BI-2a and BI-8a in under 5 hours. Graphix was additionally unable to finish queries BI-6, BI-10b, BI-12, BI-17, BI-20a, and BI-20b in under 5 hours.

Chapter 8

Conclusion

In this thesis we have introduced Graphix, an Apache AsterixDB extension that takes a view-based approach to perform ad-hoc, partitioned-parallel, and synergistic graph plus document analytics on JSON data in-situ. In contrast, current solutions (reviewed in Chapter 2) fall short on either i) the "in-situ" aspects (e.g. native graph databases), ii) the "partitioned-parallel" (e.g. graph databases like Neo4j), iii) the "ad-hoc" aspects (e.g. graph processing systems), or iv) the "synergistic" aspects (e.g. existing database graph extensions). This thesis has detailed (a) an example social network database in AsterixDB (Chapter 3), (b) how users can define a graph on top of existing AsterixDB data (Chapter 4), (c) how users can query the graphs that they define (Chapter 5), (d) what goes on "underneath-the-hood" to realize Graphix queries (Chapter 6), and (e) a performance evaluation versus a native graph database (Chapter 6). We conclude this thesis here with: 1) a summary of this thesis, and 2) potential future work for Graphix.

8.1 Conclusion

Chapter 2 reviewed several existing solutions for managing large graph data. Big Graph processing systems have been shown to be highly performant and scalable, but their "think like a vertex" paradigm still requires users to develop a *program*. Graph databases allow users to reason about their data like a graph, but require users of existing non-graph-databases to build ETL pipelines to copy their data over to the chosen graph database. We concluded this review chapter with database graph extensions, which focus on translating queries for a graph data model into the query model understood by an existing system. Graphix is a graph extension for AsterixDB.

Chapter 3 described the running example for this thesis: a social network in AsterixDB. AsterixDB is a Big Data management system (BDMS) that is designed to be a highly scalable platform for document storage, search, and analytics. The semi-structured data model provided by AsterixDB allows users to a) reason about their data with rich(er) concepts than a traditional relational model (e.g., arrays, nested objects), and b) flexibly specify a range of dataset type definitions between *schema-first* to *schema-never*. We concluded this chapter by discussing SQL^{++}, AsterixDB's query language that is purposed for querying semi-structured data while also being backwards compatible with SQL.

Chapter 4 explained the graph user model of Graphix. A Graphix graph a) is directed, b) is vertex and edge labeled, c) permits parallel edges, and d) associates properties with each vertex and edge. Furthermore, a Graphix graph is a hypergraph which relaxes the constraint that each edge associates exactly two vertices. A managed Graphix graph is defined using a `CREATE GRAPH` DDL, and an unmanaged Graph graph is defined using the `WITH` clause. Vertices and edges in Graphix are AsterixDB documents (either materialized or non-materialized), which allows Graphix to utilize SQL^{++} and even gSQL^{++} subqueries to define the vertices and edges of a graph. This chapter concluded by giving examples of the `CREATE GRAPH` DDL to handle 1) the social network example, 2) multi-dataset mappings, and 3) derived properties.

Chapter 5 detailed the query model of Graphix. Existing approaches to issuing ad-hoc graph queries involve a) the SQL-1999 recursive CTE (which result in less-than-user-friendly queries for simple computations like reachability), b) the Cypher query language (which forces users of existing data to adopt a new query language just for graph data), and c) the recent SQL-2023 SQL/PGQ (which draws a clear "line in the sand" between the relational world and the graph world). gSQL^{++}, the query language for Graphix, is a minimal extension to SQL^{++} that allows users to bind variables in the `FROM` clause directly to graph query constructs. gSQL^{++} specifically allows users to specify navigational pattern matching queries, where users can bind vertex patterns, edge patterns, and path patterns to iteration variables that are semantically indistinguishable from other non-graph SQL^{++} variables. We concluded this chapter by illustrating the implications of defining vertices, edges, and paths as documents in SQL^{++} with i) optional subgraph matching, ii) negative subgraph matching, iii) subgraph reachability, iv) shortest path finding, and v) cheapest path finding.

Chapter 6 presented the implementation underlying Graphix. This chapter led with an architectural overview of Graphix, detailing the lifecycle of a `CREATE GRAPH` statement and a gSQL^{++} query. All gSQL^{++} queries undergo two rewriting steps (one at the AST layer after parsing, another at the query plan optimization layer) that are designed to reuse and incorporate as much of AsterixDB as possible. After a gSQL^{++} query is optimized, Graphix translates the query plan into a job that AsterixDB's runtime engine, Hyracks, will distribute across the cluster. To realize Graphix, Hyracks needed to be extended in order to run recursive execution plans. We detailed the three essential properties to realize semi-synchronous partitioned-parallel recursion in Hyracks: 1) liveness, 2) safety, and 3) mortality. After explaining how Graphix performs recursion, we detailed two operators (`PBJ` and `TOP K`) to potentially optimize path traversals.

Chapter 7 evaluated Graphix against a native graph database, Neo4j. Specifically, we measured how performant a no-ETL + in-situ approach to graph queries for existing data is against a database tailored for graphs (modeling the scenario where a user performed the costly ETL and was then subsequently able to query their graph). In general, we observed that Graphix is generally able to leverage larger cluster sizes to execute queries faster than smaller sized Graphix clusters. We found that Graphix is able to perform on par with (and even outperform) Neo4j for many queries on larger graphs, however the `JOIN` order and `JOIN` physical operator impacts the performance of Graphix. Furthermore, we saw that Neo4j benefits from a bidirectional BFS technique for shortest path finding that can be orders of magnitude faster than Graphix.

8.2 Future Work

Future work with respect to Graphix can be divided into three categories: i) implementation, ii) evaluation, and iii) exploration. Starting with query model features, Graphix currently does not implement multi-label query patterns (e.g., `(:User|Message)`) and undirected path patterns (e.g., `-[:KNOWS*]-`). Graphix also does not currently allow for nested recursion, though gSQL^{++} does allow such queries to be expressed (see `BI-15` in Section A.2). The recent introduction of SQL/PGQ and Neo4j's push to migrate Cypher towards the GQL standard suggests that gSQL^{++} should perhaps also revisit how its navigational query patterns should be expressed. For example, Graphix currently provides users with the option to modify the pattern matching semantics via the `graphix.semantics.pattern` compiler option at the query level. SQL/PGQ and GQL, however, allow users to modify the pattern matching semantics for each individual query *pattern* via a keyword prefix (e.g., `WALK`, `TRAIL`, etc. . .), ultimately allowing the user to express potentially easier-to-read queries. On the subject of path navigation in Hyracks, hierarchical queries expressed using Oracle's `CONNECT BY` [53] could potentially be realized in AsterixDB by leveraging the recursion implementation of Graphix.

Potential future work for evaluating Graphix performance includes specifically characterizing the `FIXED POINT` operator (e.g., recording the number of voting periods before a loop is terminated, measuring the number of messages exchanged between each participant), characterizing the "endgame" of a recursive computation (e.g., how full each frame is during the last iterations of a loop), and determining the relationship between data size / graph structure and the size of a Graphix cluster. Chapter 7 would also benefit from a comparison against a native partitioned-parallel graph database like TigerGraph to compare speedup factors (as a function of the cluster size) for Graphix.

With respect to *exploration*, Chapter 7 demonstrated the need for cost-based optimization to determine the `JOIN` order and `JOIN` physical operators and a bidirectional BFS strategy to adequately handle power-law graphs. For graphs with a larger average diameter (potentially resulting in longer average paths), potential future work could involve leveraging the `TOP K` operator to remove the use of path objects in operations like transitive closure. Exploring alternative `JOIN` operators like worst-case-optimal-`JOIN` [51] to handle large intermediate results in between `JOIN` operations could be another piece of potential future work for Graphix. To realize iterative full-graph algorithms like PageRank, potential future work could involve leveraging Pregelix to work in tandem with Graphix to handle a larger set of use cases. Finally, in the area of visual exploratory analysis / development, a Graphix user interface project is currently being developed to act as a "Neo4j Browser"-esque parallel for Graphix (see `https://github.com/graphix-asterixdb/visualizer`).

Bibliography

[1] G. Abuoda, D. Dell'Aglio, A. Keen, and K. Hose. Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches (Extended Version), 2022.

[2] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A Scalable, Open Source BDMS. *Proceedings of the VLDB Endowment*, 7:1905–1916, 2014.

[3] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage Management in AsterixDB. *Proceedings of the VLDB Endowment*, 7(10):841–852, June 2014.

[4] Amazon. Amazon Neptune: Serverless Graph Database Designed for Superior Scalability and Availability. Available at `https://aws.amazon.com/neptune/`.

[5] R. Angles. The Property Graph Database Model. In *Alberto Mendelzon Workshop on Foundations of Data Management*, 2018.

[6] R. Angles, J. B. Antal, A. Averbuch, P. A. Boncz, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J.-L. Larriba-Pey, N. Martínez-Bazan, J. Marton, M. Paradies, M.-D. Pham, A. Prat-Pérez, M. Spasic, B. A. Steer, G. Szárnyas, and J. Waudby. The LDBC Social Network Benchmark. *ArXiv*, abs/2001.02299, 2020.

[7] R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-CORE: A Core for Future Graph Query Languages. *Proceedings of the 2018 International Conference on Management of Data*, 2017.

[8] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), sep 2017.

[9] R. Angles, H. Thakkar, and D. Tomaszuk. Mapping RDF Databases to Property Graph Databases. *IEEE Access*, 8:86091–86110, 2020.

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

[11] V. Borkar, Y. Bu, E. P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A Data Model-Agnostic Compiler Backend for Big Data Languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 422–433, New York, NY, USA, 2015. Association for Computing Machinery.

[12] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, page 1151–1162, USA, 2011. IEEE Computer Society.

[13] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) Graph Analytics on a Dataflow Engine. *Proceedings of the VLDB Endowment*, 8:161–172, 2014.

[14] D. Chamberlin. *SQL++ for SQL Users: A Tutorial.* Couchbase Incorporated, 2018.

[15] B. Chandramouli, J. Goldstein, and D. Maier. On-the-Fly Progress Detection in Iterative Stream Queries. *Proceedings of the VLDB Endowment*, 2(1):241–252, aug 2009.

[16] F. Chung and L. Lu. The Average Distances in Random Graphs with Given Expected Degrees. *Proceedings of the National Academy of Sciences*, 99(25):15879–15882, 2002.

[17] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4):661–703, nov 2009.

[18] Couchbase. Couchbase Analytics Service, Parallel Data Management Enabling Complex Analytial Queries. Available at `https://docs.couchbase.com/server/current/analytics/introduction.html`.

[19] S. Dar, R. Agrawal, and H. Jagadish. Optimization of Generalized Transitive Closure Queries. In *[1991] Proceedings. Seventh International Conference on Data Engineering*, pages 345–354, 1991.

[20] DataStax. DataStax Enterprise Graph: A Distributed Cassandra Graph Database Optimized for Enterprise Applications. Available at `https://www.datastax.com/products/datastax-graph`.

[21] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoč, M. Wu, and F. Zemke. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2246–2258, New York, NY, USA, 2022. Association for Computing Machinery.

[22] B. Elliott, E. Cheng, C. Ogbuji, and Z. M. Özsoyoglu. A Complete Translation from SPARQL into Efficient SQL. In *International Database Engineering and Applications Symposium*, 2009.

[23] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[24] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoč. A Researcher's Digest of GQL. In F. Geerts and B. Vandevoort, editors, *26th International Conference on Database Theory (ICDT 2023)*, volume 255 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[25] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[26] J. E. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[27] G. Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, Apr. 2011.

[28] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo. RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-Aggregate-SQL on Spark. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 467–484, New York, NY, USA, 2019. Association for Computing Machinery.

[29] G. E. Gévay, T. Rabl, S. Breß, L. Madai-Tahy, J.-A. Quiané-Ruiz, and V. Markl. Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1428–1439, 2021.

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

[31] D. Hirn and T. Grust. A Fix for the Fixation on Fixpoints. In *Conference on Innovative Data Systems Research*, 2023.

[32] ISO Central Secretary. Information Technology — Database Languages — SQL — Part 2: Foundation (SQL / Foundation). Standard ISO/IEC 9075-2:1999, International Organization for Standardization, Geneva, CH, 1999.

[33] ISO/IEC. Graph Query Language GQL Standard. Available at `https://www.gqlstandards.org`.

[34] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaida. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 681–697, New York, NY, USA, 2020. Association for Computing Machinery.

[35] S. Jacobs, M. Y. S. Uddin, M. J. Carey, V. Hristidis, V. J. Tsotras, N. Venkatasubramanian, Y. Wu, S. Safir, P. Kaul, X. Wang, M. A. Qader, and Y. Li. A BAD Demonstration: Towards Big Active Data. *Proceedings of the VLDB Endowment*, 10:1941–1944, 2017.

[36] S. Jahangiri. *Managing Complex Join Queries in Big Data Management Systems*. PhD Thesis, University of California, Irvine, Irvine, CA, December 2022. Available at `https://escholarship.org/uc/item/2hv8408v`.

[37] S. Jahangiri, M. J. Carey, and J.-C. Freytag. Design Trade-Offs for a Robust Dynamic Hybrid Hash Join. *Proceedings of the VLDB Endowment*, 15(10):2257–2269, jun 2022.

[38] G. Jin, N. Anzum, and S. Salihoglu. GRainDB: A Relational-core Graph-Relational DBMS. In *12th Conference on Innovative Data Systems Research, CIDR*, pages 9–12, 2022.

[39] T. Kim, A. Behm, M. Blow, V. Borkar, Y. Bu, M. J. Carey, M. Hubail, S. Jahangiri, J. Jia, C. Li, C. Luo, I. Maxon, and P. Pirzadeh. Robust and Efficient Memory Management in Apache AsterixDB. *Software: Practice and Experience*, 50(7):1114–1151, 2020.

[40] M. Levene and A. Poulovassilis. The Hypernode Model and its Associated Query Language. In *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990. 'Next Decade in Information Technology'*, pages 520–530, 1990.

[41] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab : A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment*, 2012.

[42] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[43] W. Martens and T. Trautner. Evaluation and Enumeration Problems for Regular Path Queries. In B. Kimelfeld and Y. Amsterdamer, editors, *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, volume 98 of *LIPIcs*, pages 19:1–19:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[44] A. O. Mendelzon and P. T. Wood. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, page 185–193, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

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

[46] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[47] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, Iterative Data Processing with Timely Dataflow. *Communications of the ACM*, 59(10):75–83, sep 2016.

[48] Neo4J. Bolt Protocol Documentation. Available at `https://neo4j.com/docs/bolt/current/`.

[49] Neo4j. Neo4j, the Graph Data Platform. Available at `https://neo4j.com`.

[50] Neo4j. Shortest Path Planning. Available at `https://neo4j.com/docs/cypher-manual/current/appendix/tutorials/shortestpath-planning/`.

[51] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-Case Optimal Join Algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[52] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *A Computing Research Repository*, abs/1405.3631, 2014.

[53] Oracle. Hierarchical Queries. Available at `https://docs.oracle.com/cd/B13789_01/server.101/b10759/queries003.htm`.

[54] Oracle. Oracle Spatial and Graph: Spatial and Graph Analytic Services and Data Models that Support Big Data Workloads. Available at `https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html`.

[55] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C recommendation, W3C, Jan. 2008. https://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[56] J. Rocha. Understanding Neo4j's Data on Disk. Available at `https://neo4j.com/developer/kb/understanding-data-on-disk/`.

[57] M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. *Proceedings of the 15th Symposium on Database Programming Languages*, 2015.

[58] Y. Sagiv. Optimizing Datalog Programs. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Principles of Database Systems '87, page 349–362, New York, NY, USA, 1987. Association for Computing Machinery.

[59] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, dec 2017.

[60] S. Salihoglu and M. T. Özsu. Response to "Scale Up or Scale Out for Graph Processing". *IEEE Internet Computing*, 22:18–24, 09 2018.

[61] M.-C. Shan and M.-A. Neimat. Optimization of Relational Algebra Expressions Containing Recursion Operators. In *Proceedings of the 19th Annual Conference on Computer Science*, CSC '91, page 332–341, New York, NY, USA, 1991. Association for Computing Machinery.

[62] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, aug 1986.

[63] B. A. Steer, A. Alnaimi, M. A. B. F. G. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne. Cytosm: Declarative Property Graph Queries Without Data Migration. *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017.

[64] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1887–1901, New York, NY, USA, 2015. Association for Computing Machinery.

[65] G. Szárnyas, J. Waudby, B. A. Steer, D. Szakállas, A. Birler, M. Wu, Y. Zhang, and P. Boncz. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proceedings of the VLDB Endowment*, 16(4):877–890, 2022.

[66] D. ten Wolde, T. Singh, G. Szárnyas, and P. Boncz. DuckPGQ: Efficient property Graph Queries in an Analytical RDBMS. In *Proceedings of the Conference on Innovative Data Systems Research*, jan 2023.

[67] Y. Tian. The World of Graph Databases from An Industry Perspective. *ACM SIGMOD Record*, 51:60 – 67, 2022.

[68] Y. Tian, E. L. Xu, W. Zhao, M. H. Pirahesh, S. J. Tong, W. Sun, T. Kolanko, M. S. H. Apu, and H. Peng. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 345–359, New York, NY, USA, 2020. Association for Computing Machinery.

[69] TigerGraph. TigerGraph: The World's Fastest and Most Scaleable Graph Platform. Available at `https://www.tigergraph.com`.

[70] R. W. Topor. Termination Detection for Distributed Computations. *Information Processing Letters*, 18(1):33–36, 1984.

[71] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15:555–568, 2003.

[72] Unipop. Unipop Graph: Analyze Data from Multiple Sources Using the Power of Graphs. Available at `https://github.com/unipop-graph/unipop`.

[73] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a Property Graph Query Language. In *International Workshop on Graph Data Management Experiences and Systems*, 2016.

[74] N. Yakovets, P. Godfrey, and J. Gryz. Query Planning for Evaluating SPARQL Property Paths. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1875–1889, New York, NY, USA, 2016. Association for Computing Machinery.

[75] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big Graph Analytics Platforms. *Foundations and Trends in Databases*, 7(1–2):1–195, jan 2017.

[76] F. Zemke. Converting SPARQL to SQL. Technical Report, W3C, 10 2006.

Appendix A

Benchmark Detail

In this appendix, we detail i) the DDLs and SQL^{++} transformation queries to model the LDBC social network benchmark CSV set in a more natural JSON representation for AsterixDB, and ii) the queries in gSQL^{++} used to realize the LDBC interactive and business intelligence workloads in Graphix.

A.1 Graphix DDLs

create-s1.sqlpp: AsterixDB script that defines the LDBC social network benchmark entities and relationships in AsterixDB (with a document model). . . . .

```
DROP DATAVERSE   SNB.Native IF EXISTS;
CREATE DATAVERSE SNB.Native;
USE              SNB.Native;

CREATE TYPE MessageType AS {
    id              : bigint,
    imageFile       : string?,
    creationDate    : datetime?,
    locationIP      : string,
    browserUsed     : string,
    language        : string?,
    content         : string?,
    length          : int,
    creatorId       : bigint,
    forumId         : bigint?,
    placeId         : bigint,
```

```
    replyOfMessageId : bigint?,
    isPost           : boolean,
    tags             : [bigint]
};
CREATE TYPE ForumType AS {
    id           : bigint,
    title        : string,
    creationDate : datetime,
    moderatorId  : bigint?,
    tags         : [bigint]
};
CREATE TYPE PersonType AS {
    id           : bigint,
    firstName    : string,
    lastName     : string,
    gender       : string,
    birthday     : date,
    creationDate : datetime,
    locationIP   : string,
    browserUsed  : string,
    placeId      : bigint,
    language     : [string],
    email        : [string],
    universities : [{
        organizationId : bigint,
        classYear      : int
    }],
    companies    : [{
        organizationId : bigint,
        workFrom       : int
    }]
};
CREATE TYPE KnowsType AS {
    startId      : bigint,
    endId        : bigint,
    creationDate : datetime
};
CREATE TYPE LikesType AS {
    personId     : bigint,
    messageId    : bigint,
    creationDate : datetime
};
CREATE TYPE PersonTagType AS {
    personId     : bigint,
    tagId        : bigint,
    creationDate : datetime
};
CREATE TYPE ForumPersonType AS {
    forumId  : bigint,
    personId : bigint,
    joinDate : datetime
};
CREATE TYPE TagType AS {
    id         : bigint,
    name       : string,
    url        : string,
    tagClassId : bigint
};
CREATE TYPE TagClassType AS {
    id           : bigint,
    name         : string,
    url          : string,
    isSubclassOf : bigint?
};
CREATE TYPE OrganizationType AS {
    id      : bigint,
    name    : string,
    url     : string,
```

```
    placeId : bigint
};
CREATE TYPE LocationType AS {
    id          : bigint,
    name        : string,
    url         : string,
    containerId : bigint?
};

CREATE DATASET Messages (MessageType)          PRIMARY KEY id;
CREATE DATASET Forums (ForumType)              PRIMARY KEY id;
CREATE DATASET Persons (PersonType)            PRIMARY KEY id;
CREATE DATASET Knows (KnowsType)               PRIMARY KEY startId, endId;
CREATE DATASET Likes (LikesType)               PRIMARY KEY personId, messageId;
CREATE DATASET PersonTag (PersonTagType)       PRIMARY KEY personId, tagId;
CREATE DATASET ForumPerson (ForumPersonType)   PRIMARY KEY forumId, personId;
CREATE DATASET Tags (TagType)                  PRIMARY KEY id;
CREATE DATASET TagClasses (TagClassType)       PRIMARY KEY id;
CREATE DATASET Universities (OrganizationType) PRIMARY KEY id;
CREATE DATASET Companies (OrganizationType)    PRIMARY KEY id;
CREATE DATASET Cities (LocationType)           PRIMARY KEY id;
CREATE DATASET Countries (LocationType)        PRIMARY KEY id;
CREATE DATASET Continents (LocationType)       PRIMARY KEY id;
```

transform-a1.sqlpp: Query used to define the posts in the Messages dataset. Each dataset represents an AsterixDB external dataset defined with the CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.Post p,
    SNB.FromDatagen.PostHasCreatorPerson phcp,
    SNB.FromDatagen.PostIsLocatedInCountry pilic
LET
    tags = (
        FROM
            SNB.FromDatagen.PostHasTagTag phtt
        WHERE
            phtt.PostId = p.id
        SELECT VALUE
            phtt.TagId
    ),
    forumId = (
        FROM
            SNB.FromDatagen.ForumContainerOfPost fcop
        WHERE
            fcop.PostId = p.id
        SELECT VALUE
            fcop.ForumId
    )[0]
WHERE
    p.id = phcp.PostId AND
    p.id = pilic.PostId
SELECT
    p.id                      AS id,
    p.imageFile               AS imageFile,
    DATETIME(p.creationDate)  AS creationDate,
    p.locationIP              AS locationIP,
    p.browserUsed             AS browserUsed,
    p.language                AS language,
    p.content                 AS content,
    p.length                  AS length,
    phcp.PersonId             AS creatorId,
    forumId                   AS forumId,
    pilic.CountryId           AS placeId,
```

```
    /* replyOfMessageId does not exist for Posts. */
    TRUE                       AS isPost,
    tags                       AS tags;
```

`transform-a2.sqlpp`: Query used to define the comments in the `Messages` dataset. Each dataset represents an AsterixDB external dataset defined with the CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.Comment c,
    SNB.FromDatagen.CommentHasCreatorPerson chcp,
    SNB.FromDatagen.CommentIsLocatedInCountry cilic
LET
    tags = (
        FROM
            SNB.FromDatagen.CommentHasTagTag chtt
        WHERE
            chtt.CommentId = c.id
        SELECT VALUE
            chtt.TagId
    ),
    replyOfCommentId = (
        FROM
            SNB.FromDatagen.CommentReplyOfComment cpoc
        WHERE
            cpoc.Comment1Id = c.id
        SELECT VALUE
            cpoc.Comment2Id
    )[0],
    replyOfPostId = (
        FROM
            SNB.FromDatagen.CommentReplyOfPost crop
        WHERE
            crop.CommentId = c.id
        SELECT
            VALUE crop.PostId
    )[0]
WHERE
    c.id = chcp.CommentId AND
    c.id = cilic.CommentId
SELECT
    c.id                                                AS id,
    /* imageFile does not exist for Comments. */
    DATETIME(c.creationDate)                            AS creationDate,
    c.locationIP                                        AS locationIP,
    c.browserUsed                                       AS browserUsed,
    c.content                                           AS content,
    c.length                                            AS length,
    chcp.PersonId                                       AS creatorId,
    /* forumId does not exist for Comments. */
    cilic.CountryId                                     AS placeId,
    IF_MISSING_OR_NULL(replyOfPostId, replyOfCommentId) AS replyOfMessageId,
    FALSE                                               AS isPost,
    tags                                                AS tags;
```

`transform-b.sqlpp`: Query used to define the `Forums` dataset. Each dataset represents an AsterixDB external dataset defined with the CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
```

```
    SNB.FromDatagen.Forum f
LET
    tags = (
        FROM
            SNB.FromDatagen.ForumHasTagTag fhtt
        WHERE
            fhtt.ForumId = f.id
        SELECT VALUE
            fhtt.TagId
    ),
    moderatorId = (
        FROM
            SNB.FromDatagen.ForumHasModeratorPerson fhmp
        WHERE
            fhmp.ForumId = f.id
        SELECT VALUE
            fhmp.PersonId
    )[0]
SELECT
    f.id                         AS id,
    f.title                      AS title,
    DATETIME(f.creationDate)     AS creationDate,
    moderatorId                  AS moderatorId,
    tags                         AS tags;
```

transform-c.sqlpp: Query used to define the Persons dataset. Each dataset represents an AsterixDB external dataset defined with the CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.Person p,
    SNB.FromDatagen.PersonIsLocatedInCity pilic
LET
    universities = (
        FROM
            SNB.FromDatagen.PersonStudyAtUniversity psau
        WHERE
            psau.PersonId = p.id
        SELECT
            psau.UniversityId AS organizationId,
            psau.classYear    AS classYear
    ),
    companies = (
        FROM
            SNB.FromDatagen.PersonWorkAtCompany pwac
        WHERE
            pwac.PersonId = p.id
        SELECT
            pwac.CompanyId AS organizationId,
            pwac.workFrom  AS workFrom
    )
WHERE
    p.id = pilic.PersonId
SELECT
    p.id                         AS id,
    p.firstName                  AS firstName,
    p.lastName                   AS lastName,
    p.gender                     AS gender,
    DATE(p.birthday)             AS birthday,
    DATETIME(p.creationDate)     AS creationDate,
    p.locationIP                 AS locationIP,
    p.browserUsed                AS browserUsed,
    pilic.CityId                 AS placeId,
    SPLIT(p.email, ';')          AS email,
```

```
    SPLIT(p.language, ';')   AS language,
    universities             AS universities,
    companies                AS companies;
```

transform-d.sqlpp: Query used to define the `Knows` dataset. The `PersonKnowsPerson` dataset represents an AsterixDB external dataset defined with the `PersonKnowsPerson` CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.PersonKnowsPerson pkp
SELECT
    DATETIME(pkp.creationDate) AS creationDate,
    pkp.Person1Id              AS startId,
    pkp.Person2Id              AS endId
UNION ALL
FROM
    SNB.FromDatagen.PersonKnowsPerson pkp
SELECT
    DATETIME(pkp.creationDate) AS creationDate,
    pkp.Person2Id              AS startId,
    pkp.Person1Id              AS endId;
```

transform-e.sqlpp: Query used to define the `Likes` dataset. Each dataset represents an AsterixDB external dataset defined with the CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.PersonLikesComment plc
SELECT
    DATETIME(plc.creationDate) AS creationDate,
    plc.PersonId               AS personId,
    plc.CommentId              AS messageId
UNION ALL
FROM
    SNB.FromDatagen.PersonLikesPost plp
SELECT
    DATETIME(plp.creationDate) AS creationDate,
    plp.PersonId               AS personId,
    plp.PostId                 AS messageId;
```

transform-f.sqlpp: Query used to define the `PersonTag` dataset. The `PersonHasInterestTag` dataset represents an AsterixDB external dataset defined with the `PersonHasInterestTag` CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.PersonHasInterestTag phit
SELECT
    phit.PersonId               AS personId,
    phit.InterestId             AS tagId,
    DATETIME(phit.creationDate) AS creationDate;
```

`transform-g.sqlpp`: Query used to define the `ForumPerson` dataset. The `ForumHasMemberPerson` dataset represents an AsterixDB external dataset defined with the `ForumHasMemberPerson` CSV collection generated by the LDBC's social network graph generator. . . . .

```
FROM
    SNB.FromDatagen.ForumHasMemberPerson fhmp
SELECT
    fhmp.ForumId                AS forumId,
    fhmp.PersonId               AS personId,
    DATETIME(fhmp.creationDate) AS joinDate;
```

`create-s3a.sqlpp`: AsterixDB script used to load the "dynamic" entities of the LDBC social network graph into AsterixDB as managed datasets. . . . .

```
LOAD  DATASET SNB.Native.Messages
USING localfs (
    ("path"="$DATA_PATH/Messages.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.Forums
USING localfs (
    ("path"="$DATA_PATH/Forums.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.Persons
USING localfs (
    ("path"="$DATA_PATH/Persons.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.Knows
USING localfs (
    ("path"="$DATA_PATH/Knows.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.Likes
USING localfs (
    ("path"="$DATA_PATH/Likes.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.PersonTag
USING localfs (
    ("path"="$DATA_PATH/PersonTag.adm"),
    ("format"="adm")
);
LOAD  DATASET SNB.Native.ForumPerson
USING localfs (
    ("path"="$DATA_PATH/ForumPerson.adm"),
    ("format"="adm")
);
```

`create-s3b.sqlpp`: AsterixDB script used to transform the each "static" entity's CSV collection (generated from the LDBC's social network graph generator) to directly populate the corresponding AsterixDB managed datasets. . . . .

```
USE SNB.Native;

INSERT INTO Tags (
    FROM
```

```
        SNB.FromDatagen.Tag t,
        SNB.FromDatagen.TagHasTypeTagClass thttc
    WHERE
        thttc.TagId = t.id
    SELECT
        t.*,
        thttc.TagClassId AS tagClassId
);
INSERT INTO TagClasses (
    FROM
        SNB.FromDatagen.TagClass tc
    LET
        isSubclassOf = (
            FROM
                SNB.FromDatagen.TagClassIsSubclassOfTagClass tcisotc
            WHERE
                tcisotc.TagClass1Id = tc.id
            SELECT VALUE
                tcisotc.TagClass2Id
        )[0]
    SELECT
        tc.*,
        isSubclassOf AS isSubclassOf
);

INSERT INTO Universities (
    FROM
        SNB.FromDatagen.Organisation o,
        SNB.FromDatagen.OrganisationIsLocatedInPlace oilip
    WHERE
        o.`type` LIKE 'University' AND
        oilip.OrganisationId = o.id
    SELECT
        o.id          AS id,
        o.name        AS name,
        o.url         AS url,
        oilip.PlaceId AS placeId
);
INSERT INTO Companies (
    FROM
        SNB.FromDatagen.Organisation o,
        SNB.FromDatagen.OrganisationIsLocatedInPlace oilip
    WHERE
        o.`type` LIKE 'Company' AND
        oilip.OrganisationId = o.id
    SELECT
        o.id          AS id,
        o.name        AS name,
        o.url         AS url,
        oilip.PlaceId AS placeId
);

INSERT INTO Cities (
    FROM
        SNB.FromDatagen.Place p,
        SNB.FromDatagen.PlaceIsPartOfPlace pipop
    WHERE
        p.`type` = 'City' AND
        pipop.Place1Id = p.id
    SELECT
        p.id           AS id,
        p.name         AS name,
        p.url          AS url,
        pipop.Place2Id AS containerId
);
INSERT INTO Countries (
    FROM
        SNB.FromDatagen.Place p,
```

```
        SNB.FromDatagen.PlaceIsPartOfPlace pipop
    WHERE
        p.`type` = 'Country' AND
        pipop.Place1Id = p.id
    SELECT
        p.id           AS id,
        p.name         AS name,
        p.url          AS url,
        pipop.Place2Id AS containerId
);
INSERT INTO Continents (
    FROM
        SNB.FromDatagen.Place p
    WHERE
        p.`type` = 'Continent'
    SELECT
        p.id   AS id,
        p.name AS name,
        p.url  AS url
);
```

create-s4.sqlpp: AsterixDB script that defines a set of indexes for each foreign key of the previously defined AsterixDB datasets. . . . .

```
USE SNB.Native;

CREATE INDEX messageForumIdIndex   ON Messages ( forumId );
CREATE INDEX messageCreatorIdIndex ON Messages ( creatorId );
CREATE INDEX messagePlaceIdIndex   ON Messages ( placeId );
CREATE INDEX messageReplyOfIndex   ON Messages ( replyOfMessageId );
CREATE INDEX messageTagsIndex      ON Messages (
    UNNEST tags
) EXCLUDE UNKNOWN KEY;

CREATE INDEX forumPersonPersonIdIndex ON ForumPerson ( personId );
CREATE INDEX forumModeratorIdIndex    ON Forums ( moderatorId );
CREATE INDEX forumTagIndex            ON Forums (
    UNNEST tags
) EXCLUDE UNKNOWN KEY;

CREATE INDEX knowsEndPersonIndex ON Knows ( endId );

CREATE INDEX personPlaceIdIndex      ON Persons ( placeId );
CREATE INDEX personUniversitiesIndex ON Persons (
    UNNEST universities
    SELECT organizationId
) EXCLUDE UNKNOWN KEY;
CREATE INDEX personsCompaniesIndex   ON Persons (
    UNNEST companies
    SELECT organizationId
) EXCLUDE UNKNOWN KEY;

CREATE INDEX personTagTagIdIndex ON PersonTag ( tagId );
CREATE INDEX likesMessageIdIndex ON Likes ( messageId );

CREATE INDEX tagTagClassIdIndex        ON Tags ( tagClassId );
CREATE INDEX tagClassesSubclassOfIndex ON TagClasses ( isSubclassOf );

CREATE INDEX universitiesPlaceIdIndex  ON Universities ( placeId );
CREATE INDEX companiesPlaceIdIndex     ON Companies ( placeId );
CREATE INDEX citiesContainerIdIndex    ON Cities ( containerId );
CREATE INDEX countriesContainerIdIndex ON Countries ( containerId );
```

create-s5.sqlpp: Graphix script used to define the `SNBGraph` graph with the aforementioned datasets. . . . .

```
USE SNB.Native;

DROP   GRAPH SNBGraph IF EXISTS;
CREATE GRAPH SNBGraph AS
    VERTEX (:Message)
        PRIMARY KEY (id)
        AS Messages,
    VERTEX (:Forum)
        PRIMARY KEY (id)
        AS Forums,
    VERTEX (:Person)
        PRIMARY KEY (id)
        AS Persons,
    VERTEX (:Tag)
        PRIMARY KEY (id)
        AS Tags,
    VERTEX (:TagClass)
        PRIMARY KEY (id)
        AS TagClasses,
    VERTEX (:University)
        PRIMARY KEY (id)
        AS Universities,
    VERTEX (:Company)
        PRIMARY KEY (id)
        AS Companies,
    VERTEX (:City)
        PRIMARY KEY (id)
        AS Cities,
    VERTEX (:Country)
        PRIMARY KEY (id)
        AS Countries,
    VERTEX (:Continent)
        PRIMARY KEY (id)
        AS Continents,

    EDGE (:Message)-[:REPLY_OF]->(:Message)
        SOURCE KEY      (id)
        DESTINATION KEY (replyOfMessageId)
        AS (
            FROM
                Messages m
            SELECT
                m.id               AS id,
                m.replyOfMessageId AS replyOfMessageId
        ),
    EDGE (:Message)-[:HAS_CREATOR]->(:Person)
        SOURCE KEY      (id)
        DESTINATION KEY (creatorId)
        AS ( FROM Messages SELECT id, creatorId ),
    EDGE (:Message)-[:IS_LOCATED_IN]->(:Country)
        SOURCE KEY      (id)
        DESTINATION KEY (placeId)
        AS ( FROM Messages SELECT id, placeId ),
    EDGE (:Message)-[:HAS_TAG]->(:Tag)
        SOURCE KEY      (id)
        DESTINATION KEY (tagId)
        AS (
            FROM
                Messages m,
                m.tags tagId
            SELECT
                m.id  AS id,
                tagId AS tagId
        ),
```

```
    EDGE (:Forum)-[:CONTAINER_OF]->(:Message)
        SOURCE KEY      (forumId)
        DESTINATION KEY (id)
        AS (
            FROM
                Messages m
            WHERE
                m.isPost
            SELECT
                m.id      AS id,
                m.forumId AS forumId
        ),
    EDGE (:Forum)-[:HAS_MODERATOR]->(:Person)
        SOURCE KEY      (id)
        DESTINATION KEY (moderatorId)
        AS ( FROM Forums SELECT id, moderatorId ),
    EDGE (:Forum)-[:HAS_MEMBER]->(:Person)
        SOURCE KEY      (forumId)
        DESTINATION KEY (personId)
        AS ( FROM ForumPerson fp SELECT VALUE fp ),
    EDGE (:Forum)-[:HAS_TAG]->(:Tag)
        SOURCE KEY      (id)
        DESTINATION KEY (tagId)
        AS (
            FROM
                Forums f,
                f.tags tagId
            SELECT
                f.id  AS id,
                tagId AS tagId
        ),
    EDGE (:Person)-[:KNOWS]->(:Person)
        SOURCE KEY      (startId)
        DESTINATION KEY (endId)
        AS ( FROM Knows k SELECT VALUE k ),
    EDGE (:Person)-[:HAS_INTEREST]->(:Tag)
        SOURCE KEY      (personId)
        DESTINATION KEY (tagId)
        AS ( FROM PersonTag pt SELECT VALUE pt ),
    EDGE (:Person)-[:IS_LOCATED_IN]->(:City)
        SOURCE KEY      (id)
        DESTINATION KEY (placeId)
        AS ( FROM Persons SELECT id, placeId ),
    EDGE (:Person)-[:STUDY_AT]->(:University)
        SOURCE KEY      (id)
        DESTINATION KEY (organizationId)
        AS (
            FROM
                Persons p,
                p.universities u
            SELECT
                p.id             AS id,
                u.organizationId AS organizationId,
                u.classYear      AS classYear
        ),
    EDGE (:Person)-[:WORK_AT]->(:Company)
        SOURCE KEY      (id)
        DESTINATION KEY (organizationId)
        AS (
            FROM
                Persons p,
                p.companies c
            SELECT
                p.id             AS id,
                c.organizationId AS organizationId,
                c.workFrom       AS workFrom
        ),
    EDGE (:Person)-[:LIKES]->(:Message)
```

```
        SOURCE KEY      (personId)
        DESTINATION KEY (messageId)
        AS ( FROM Likes l SELECT VALUE l ),
    EDGE (:Tag)-[:HAS_TYPE]->(:TagClass)
        SOURCE KEY      (id)
        DESTINATION KEY (tagClassId)
        AS ( FROM Tags SELECT id, tagClassId ),
    EDGE (:TagClass)-[:IS_SUBCLASS_OF]->(:TagClass)
        SOURCE KEY      (id)
        DESTINATION KEY (isSubclassOf)
        AS ( FROM TagClasses SELECT id, isSubclassOf ),
    EDGE (:University)-[:IS_LOCATED_IN]->(:City)
        SOURCE KEY      (id)
        DESTINATION KEY (placeId)
        AS ( FROM Universities SELECT id, placeId ),
    EDGE (:Company)-[:IS_LOCATED_IN]->(:Country)
        SOURCE KEY      (id)
        DESTINATION KEY (placeId)
        AS ( FROM Companies SELECT id, placeId ),
    EDGE (:City)-[:IS_PART_OF]->(:Country)
        SOURCE KEY      (id)
        DESTINATION KEY (containerId)
        AS ( FROM Cities SELECT id, containerId ),
    EDGE (:Country)-[:IS_PART_OF]->(:Continent)
        SOURCE KEY      (id)
        DESTINATION KEY (containerId)
        AS ( FROM Countries SELECT id, containerId );
```

A.2 Graphix Queries (in gSQL^{++})

short-1.sqlpp: SNB query IS-1 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:IS_LOCATED_IN]->(city:City)
SELECT
    person.firstName                                    AS firstName,
    person.lastName                                     AS lastName,
    UNIX_TIME_FROM_DATE_IN_MS(person.birthday)          AS birthday,
    person.locationIP                                   AS locationIp,
    person.browserUsed                                  AS browserUsed,
    city.id                                             AS cityId,
    person.gender                                       AS gender,
    UNIX_TIME_FROM_DATETIME_IN_MS(person.creationDate)  AS creationDate;
```

short-2.sqlpp: SNB query IS-2 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
LET
    topMessages = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person:Person WHERE person.id = $personId),
```

```
                (person)<-[:HAS_CREATOR]-(message:Message)
        SELECT VALUE
            message.id
        ORDER BY
            message.creationDate DESC
        LIMIT
            10
    )
FROM
    topMessages tm,
    GRAPH SNB.Native.SNBGraph
        (message:Message)-[:REPLY_OF*]->(post:Message),
        (post)-[:HAS_CREATOR]->(originalPoster:Person)
WHERE
    tm /*+indexnl*/ = message.id AND
    post.isPost
SELECT
    message.id                                          AS messageId,
    COALESCE(message.content, message.imageFile)        AS messageContent,
    UNIX_TIME_FROM_DATETIME_IN_MS(message.creationDate) AS messageCreationDate,
    post.id                                             AS originalPostId,
    originalPoster.id                                   AS originalPostAuthorId,
    originalPoster.firstName                            AS originalPostAuthorFirstName,
    originalPoster.lastName                             AS originalPostAuthorLastName
ORDER BY
    messageCreationDate DESC,
    messageId DESC
LIMIT
    $limit;
```

short-3.sqlpp: SNB query IS-3 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[knows:KNOWS]->(friend:Person)
SELECT
    friend.id                                         AS personId,
    friend.firstName                                  AS firstName,
    friend.lastName                                   AS lastName,
    UNIX_TIME_FROM_DATETIME_IN_MS(knows.creationDate) AS friendshipCreationDate
ORDER BY
    friendshipCreationDate DESC,
    friend.id ASC;
```

short-4.sqlpp: SNB query IS-4 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (message:Message WHERE message.id = $messageId)
SELECT
    UNIX_TIME_FROM_DATETIME_IN_MS(message.creationDate) AS messageCreationDate,
    COALESCE(message.content, message.imageFile)        AS messageContent;
```

short-5.sqlpp: SNB query IS-5 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (message:Message WHERE message.id = $messageId),
        (message)-[:HAS_CREATOR]->(person:Person)
SELECT
    person.id        AS personId,
    person.firstName AS firstName,
    person.lastName  AS lastName;
```

short-6.sqlpp: SNB query IS-6 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (message:Message WHERE message.id = $messageId),
        (message)-[:REPLY_OF*]->(post:Message),
        (post)<-[:CONTAINER_OF]-(forum:Forum),
        (forum)-[:HAS_MODERATOR]->(moderator:Person)
WHERE
    post.isPost
SELECT
    forum.id            AS forumId,
    forum.title         AS forumTitle,
    moderator.id        AS moderatorId,
    moderator.firstName AS moderatorFirstName,
    moderator.lastName  AS moderatorLastName;
```

short-7.sqlpp: SNB query IS-7 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE** and the compiler option graphix.semantics.pattern was set to "edge-isomorphism". . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (message:Message WHERE message.id = $messageId),
        (message)-[:HAS_CREATOR]->(messageAuthor:Person),
        (message)<-[:REPLY_OF]-(comment:Message),
        (comment)-[:HAS_CREATOR]->(replyAuthor:Person)
LET
    isKnows = EXISTS (
        FROM
            GRAPH SNB.Native.SNBGraph
                (replyAuthor)-[:KNOWS]->(messageAuthor)
        SELECT
            1
    )
SELECT DISTINCT
    comment.id                                         AS commentId,
    comment.content                                    AS commentContent,
    UNIX_TIME_FROM_DATETIME_IN_MS(comment.creationDate) AS commentCreationDate,
    replyAuthor.id                                     AS replyAuthorId,
    replyAuthor.firstName                              AS replyAuthorFirstName,
    replyAuthor.lastName                               AS replyAuthorLastName,
    isKnows                                            AS isKnows
ORDER BY
    commentCreationDate DESC,
    replyAuthorId ASC;
```

`complex-1.sqlpp`: SNB query `IC-1` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[p:KNOWS{1,3}]->(otherPerson:Person),
        (otherPerson)-[:IS_LOCATED_IN]->(locationCity:City)
LET
    companies = (
        FROM
            otherPerson.companies opc,
            GRAPH SNB.Native.SNBGraph
                (company:Company)-[:IS_LOCATED_IN]->(companyCountry:Country)
        WHERE
            opc.organizationId /*+indexnl*/ = company.id
        SELECT
            company.name        AS companyName,
            company.workFrom    AS workFrom,
            companyCountry.name AS countryName
    ),
    universities = (
        FROM
            otherPerson.universities opu,
            GRAPH SNB.Native.SNBGraph
                (university:University)-[:IS_LOCATED_IN]->(universityCity:City)
        WHERE
            opu.organizationId /*+indexnl*/ = university.id
        SELECT
            university.name      AS universityName,
            university.classYear AS classYear,
            universityCity.name  AS cityName
    )
WHERE
    otherPerson.firstName = $firstName
GROUP BY
    person.id,
    otherPerson.id,
    otherPerson,
    locationCity,
    companies,
    universities
SELECT
    otherPerson.id                                        AS friendId,
    otherPerson.lastName                                  AS friendLastName,
    MIN(LEN(EDGES(p)))                                    AS distanceFromPerson,
    UNIX_TIME_FROM_DATE_IN_MS(otherPerson.birthday)       AS friendBirthday,
    UNIX_TIME_FROM_DATETIME_IN_MS(otherPerson.creationDate) AS friendCreationDate,
    otherPerson.gender                                    AS friendGender,
    otherPerson.browserUsed                               AS friendBrowserUsed,
    otherPerson.locationIP                                AS friendLocationIp,
    otherPerson.email                                     AS friendEmails,
    otherPerson.speaks                                    AS friendLanguages,
    locationCity.name                                     AS friendCityName,
    universities                                          AS friendUniversities,
    companies                                             AS friendCompanies
ORDER BY
    distanceFromPerson ASC,
    otherPerson.lastName ASC,
    otherPerson.id ASC
LIMIT
    $limit;
```

complex-2.sqlpp: SNB query IC-2 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS]->(friend:Person),
        (friend)<-[:HAS_CREATOR]-(message:Message)
WHERE
    message.creationDate < $maxDate
SELECT
    friend.id                                           AS personId,
    friend.firstName                                    AS personFirstName,
    friend.lastName                                     AS personLastName,
    message.id                                          AS messageId,
    COALESCE(message.content, message.imageFile)        AS messageContent,
    UNIX_TIME_FROM_DATETIME_IN_MS(message.creationDate) AS messageCreationDate
ORDER BY
    messageCreationDate DESC,
    messageId ASC
LIMIT
    $limit;
```

complex-3.sqlpp: SNB query IC-3 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId)-[:KNOWS{1,2}]->(otherPerson:Person),
        (otherPerson)<-[:HAS_CREATOR]-(m1:Message)-[:IS_LOCATED_IN]->(countryX:Country),
        (otherPerson)<-[:HAS_CREATOR]-(m2:Message)-[:IS_LOCATED_IN]->(countryY:Country),
        (otherPerson)-[:IS_LOCATED_IN]->(city:City)
LET
    endDate = $startDate + DURATION(CONCAT("P", TO_STRING($durationDays), "D"))
WHERE
    (m1.creationDate BETWEEN $startDate AND endDate) AND
    (m2.creationDate BETWEEN $startDate AND endDate) AND
    countryX.name = $countryXName AND
    countryY.name = $countryYName AND
    city.containerId != countryX.id AND
    city.containedId != countryY.id
GROUP BY
    person.id,
    otherPerson
    GROUP AS g
LET
    xCount = ARRAY_COUNT((FROM g SELECT DISTINCT g.m1.id)),
    yCount = ARRAY_COUNT((FROM g SELECT DISTINCT g.m2.id))
SELECT
    otherPerson.id        AS personId,
    otherPerson.firstName AS personFirstName,
    otherPerson.lastName  AS personLastName,
    xCount                AS xCount,
    yCount                AS yCount,
    xCount + yCount       AS `count`
ORDER BY
    `count` DESC,
    personId ASC
LIMIT
    $limit;
```

complex-4.sqlpp: SNB query `IC-4` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS]->(:Person)<-[:HAS_CREATOR]-(post:Message),
        (post)-[:HAS_TAG]->(tag:Tag),
        (person)-[:KNOWS]->(:Person)<-[:HAS_CREATOR]-(post2:Message)
LET
    endDate = $startDate + DURATION(CONCAT("P", TO_STRING($durationDays), "D"))
WHERE
    post.isPost AND
    post2.isPost AND
    (post.creationDate BETWEEN $startDate AND endDate) AND
    (post2.creationDate BETWEEN $startDate AND endDate) AND
    tag.id NOT IN post2.tags
GROUP BY
    tag.name AS tagName
SELECT
    tagName                  AS tagName,
    COUNT(DISTINCT post.id)  AS postCount
ORDER BY
    postCount DESC,
    tagName ASC
LIMIT
    $limit;
```

complex-5.sqlpp: SNB query `IC-5` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**. There exists a semantically equivalent **LEFT MATCH** variant for this query, however the listing below was used for the benchmark. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId)-[:KNOWS{1,2}]->(otherPerson:Person),
        (otherPerson)<-[h:HAS_MEMBER]-(forum:Forum)
    LEFT JOIN
        (
            FROM
                SNB.Native.Messages p
            WHERE
                p.isPost
            SELECT
                p.forumId   AS forumId,
                p.creatorId AS creatorId
        ) post ON post.forumId = forum.id AND post.creatorId = otherPerson.id
WHERE
    h.joinDate > $minDate
GROUP BY
    forum
SELECT
    forum.title          AS forumTitle,
    COUNT(DISTINCT post) AS postCount
ORDER BY
    postCount DESC,
    forum.id ASC
LIMIT
    $limit;
```

complex-6.sqlpp: SNB query IC-6 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS{1,2}]->(:Person)<-[:HAS_CREATOR]-(post:Message),
        (post)-[:HAS_TAG]->(tag:Tag),
        (post)-[:HAS_TAG]->(otherTag:Tag)
WHERE
    tag.name = $tagName AND
    post.isPost
GROUP BY
    otherTag
SELECT
    otherTag.name         AS tagName,
    COUNT(DISTINCT post) AS postCount
ORDER BY
    postCount DESC,
    tagName ASC
LIMIT
    $limit;
```

complex-7.sqlpp: SNB query IC-7 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)<-[:HAS_CREATOR]-(message:Message),
        (message)<-[likes:LIKES]-(friend:Person)
LET
    isNew = friend.id NOT IN person.knows
GROUP BY
    friend,
    isNew
    GROUP AS g
LET
    likeInfo = (
        FROM
            g
        SELECT
            g.likes.creationDate,
            g.message
        ORDER BY
            g.likes.creationDate DESC,
            g.message.id ASC
        LIMIT
            1
    )[0],
    latency = GET_DAY_TIME_DURATION(likeInfo.creationDate - likeInfo.message.creationDate)
SELECT
    friend.id                                                          AS personId,
    friend.firstName                                                   AS personFirstName,
    friend.lastName                                                    AS personLastName,
    UNIX_TIME_FROM_DATETIME_IN_MS(likeInfo.creationDate)               AS likeCreationDate,
    likeInfo.message.id                                                AS messageId,
    COALESCE(likeInfo.message.content, likeInfo.message.imageFile)     AS messageContent,
    MS_FROM_DAY_TIME_DURATION(latency) / 60000.0                       AS minutesLatency,
    isNew                                                              AS isNew
ORDER BY
    likeCreationDate DESC,
    personId ASC
LIMIT
```

```
    $limit;
```

complex-8.sqlpp: SNB query `IC-8` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**...........................

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)<-[:HAS_CREATOR]-(:Message)<-[:REPLY_OF]-(comment:Message),
        (comment)-[:HAS_CREATOR]->(commentAuthor:Person)
WHERE
    NOT comment.isPost
SELECT
    commentAuthor.id                                   AS personId,
    commentAuthor.firstName                            AS personFirstName,
    commentAuthor.lastName                             AS personLastName,
    UNIX_TIME_FROM_DATETIME_IN_MS(comment.creationDate) AS commentCreationDate,
    comment.id                                         AS commentId,
    comment.content                                    AS commentContent
ORDER BY
    commentCreationDate DESC,
    commentId ASC
LIMIT
    $limit;
```

complex-9.sqlpp: SNB query `IC-9` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**...........................

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS{1,2}]->(otherPerson:Person),
        (otherPerson)<-[:HAS_CREATOR]-(message:Message)
WHERE
    message.creationDate < $maxDate
SELECT DISTINCT
    otherPerson.id                                     AS personId,
    otherPerson.firstName                              AS personFirstName,
    otherPerson.lastName                               AS personLastName,
    message.id                                         AS messageId,
    COALESCE (message.content, message.imageFile)      AS messageContent,
    UNIX_TIME_FROM_DATETIME_IN_MS(message.creationDate) AS messageCreationDate
ORDER BY
    messageCreationDate DESC,
    messageId ASC
LIMIT
    $limit;
```

complex-10.sqlpp: SNB query `IC-10` for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**........................

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS]->(:Person)-[:KNOWS]->(friend:Person),
        (friend)-[:IS_LOCATED_IN]->(city:City)
LET
```

```
    friendPosts = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (friend)<-[:HAS_CREATOR]-(post:Message WHERE post.isPost)
        SELECT VALUE
            post.id
    ),
    commonPosts = (
        FROM
            friendPosts fp,
            GRAPH SNB.Native.SNBGraph
                (commonPost:Message)-[:HAS_TAG]->(:Tag)<-[:HAS_INTEREST]-(person)
        WHERE
            fp /*+indexnl*/ = commonPost.id
        SELECT DISTINCT VALUE
            fp
    ),
    commonPostCount = ARRAY_COUNT(commonPosts),
    commonInterestScore = commonPostCount - (ARRAY_COUNT(friendPosts) - commonPostCount)
WHERE
    ((GET_MONTH(friend.birthday) = $month AND GET_DAY(friend.birthday) >= 21) OR
     (GET_MONTH(friend.birthday) = (3 % 12) + 1 AND GET_DAY(friend.birthday) < 22)) AND
    NOT EXISTS (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person)-[:KNOWS]->(friend)
        SELECT
            1
    )
SELECT
    friend.id           AS personId,
    friend.firstName    AS personFirstName,
    friend.lastName     AS personLastName,
    commonInterestScore AS commonInterestScore,
    friend.gender       AS personGender,
    city.name           AS personCityName
ORDER BY
    commonInterestScore DESC,
    personId ASC
LIMIT
    $limit;
```

complex-11.sqlpp: SNB query IC-11 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**.......................

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person WHERE person.id = $personId),
        (person)-[:KNOWS{1,2}]->(otherPerson:Person),
        (otherPerson)-[w:WORK_AT]->(company:Company),
        (company)-[:IS_LOCATED_IN]->(country:Country)
WHERE
    w.workFrom < $workFromYear AND
    country.name = $countryName
GROUP BY
    person.id,
    otherPerson.id,
    otherPerson,
    company.name AS organizationName,
    w.workFrom   AS organizationWorkFromYear
SELECT
    otherPerson.id          AS personId,
    otherPerson.firstName   AS personFirstName,
    otherPerson.lastName    AS personLastName,
```

```
    organizationName         AS organizationName,
    organizationWorkFromYear AS organizationWorkFromYear
ORDER BY
    organizationWorkFromYear ASC,
    personId ASC,
    organizationName DESC
LIMIT
    $limit;
```

complex-12.sqlpp: SNB query IC-12 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**.

```
FROM
    GRAPH SNB.Native.SNBGraph
        (:Person)-[k:KNOWS WHERE k.startId = $personId]->(friend:Person),
        (friend)<-[:HAS_CREATOR]-(comment:Message)-[:REPLY_OF]->(post:Message),
        (post)-[:HAS_TAG]->(tag:Tag)-[:HAS_TYPE]->(tc:TagClass),
        (tc)-[:IS_SUBCLASS_OF*]->(tagClass:TagClass)
WHERE
    NOT comment.isPost AND
    post.isPost AND
    tagClass.name = $tagClassName
GROUP BY
    friend
    GROUP AS g
LET
    tagNames = (FROM g SELECT DISTINCT VALUE g.tag.name)
SELECT
    friend.id                   AS personId,
    friend.firstName            AS personFirstName,
    friend.lastName             AS personLastName,
    tagNames                    AS tagNames,
    COUNT(DISTINCT comment.id) AS replyCount
ORDER BY
    replyCount DESC,
    personId ASC
LIMIT
    $limit;
```

complex-13.sqlpp: SNB query IC-13 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**.

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person1:Person)-[k:KNOWS+]->(person2:Person)
WHERE
    person1.id = $person1Id AND
    person2.id = $person2Id
GROUP BY
    person1.id,
    person2.id
    GROUP AS g
LET
    shortestPathLength = (
        FROM
            g
        SELECT VALUE
            LEN(EDGES(g.k))
        ORDER BY
            LEN(EDGES(g.k)) ASC
        LIMIT
```

```
            1
    )[0]
SELECT VALUE
    COALESCE(shortestPathLength, -1);
```

complex-14.sqlpp: SNB query IC-14 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**........................

```
WITH
    GRAPH Complex14Graph AS
        VERTEX (:Person)
            PRIMARY KEY (id)
            AS SNB.Native.Persons,
        EDGE (:Person)-[:KNOWS]->(:Person)
            SOURCE KEY      (startId)
            DESTINATION KEY (endId)
            AS (
                FROM
                    SNB.Native.Messages m1,
                    SNB.Native.Messages m2,
                    SNB.Native.Knows k
                WHERE
                    k.startId = m1.creatorId AND
                    k.endId = m2.creatorId AND
                    (m1.replyOfMessageId = m2.id OR
                     m2.replyOfMessageId = m1.id)
                GROUP BY
                    m1.creatorId AS startId,
                    m2.creatorId AS endId
                    GROUP AS g
                LET
                    w1 = (
                        FROM
                            g
                        WHERE
                            g.m1.isPost OR g.m2.isPost
                        SELECT VALUE
                            COUNT(*)
                        )[0],
                    w2 = (
                        FROM
                            g
                        WHERE
                            NOT g.m1.isPost OR NOT g.m2.isPost
                        SELECT VALUE
                            COUNT(*)
                        )[0] * 0.5
                SELECT
                    startId AS startId,
                    endId   AS endId,
                    w1 + w2 AS weight
            )
FROM
    GRAPH Complex14Graph
        (person1:Person)-[k:KNOWS+]->(person2:Person)
WHERE
    person1.id = $person1Id AND
    person2.id = $person2Id
GROUP BY
    person1.id,
    person2.id
    GROUP AS g
LET
    cheapestPath = (
```

```
        FROM
            g
        SELECT
            (FROM VERTICES(g.k) kv SELECT VALUE kv.id)           AS ids,
            (FROM EDGES(g.k) ke SELECT VALUE SUM(ke.weight))[0] AS cost
        ORDER BY
            ABS(cost) ASC
        LIMIT
            1
    )[0]
SELECT
    cheapestPath.ids  AS personIdsInPath,
    cheapestPath.cost AS pathWeight
ORDER BY
    pathWeight DESC;
```

bi-1.sqlpp: SNB query BI-1 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
LET
    totalMessages = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (inner_m:Message)
        WHERE
            inner_m.creationDate < $datetime AND
            inner_m.content IS NOT NULL
        SELECT VALUE
            COUNT(inner_m.id)
    )[0]
FROM
    GRAPH SNB.Native.SNBGraph
        (message:Message)
LET
    year           = GET_YEAR(message.creationDate),
    isComment      = NOT message.isPost,
    lengthCategory = CASE
        WHEN LENGTH(message.content) < 40
        THEN 0
        WHEN LENGTH(message.content) < 80
        THEN 1
        WHEN LENGTH(message.content) < 160
        THEN 2
        ELSE 3
    END
WHERE
    message.creationDate < $datetime AND
    message.content IS NOT NULL
GROUP BY
    year,
    isComment,
    lengthCategory
SELECT
    year                                 AS year,
    isComment                            AS isComment,
    lengthCategory                       AS lengthCategory,
    COUNT(*)                             AS messageCount,
    AVG(LENGTH(message.content))         AS averageMessageLength,
    SUM(LENGTH(message.content))         AS sumMessageLength,
    COUNT(*) * 100.0 / totalMessages     AS percentageOfMessages
ORDER BY
    year DESC,
    isComment ASC,
    lengthCategory ASC;
```

bi-2.sqlpp: SNB query BI-2 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (tagClass:TagClass WHERE tagClass.name = $tagClass),
        (tagClass)<-[:HAS_TYPE]-(tag:Tag)
LET
    countWindow1 = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (m1:Message)-[:HAS_TAG]->(tag)
        WHERE
            m1.creationDate BETWEEN $date AND ($date + DURATION("P100D"))
        SELECT VALUE
            COUNT(m1.id)
    )[0],
    countWindow2 = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (m2:Message)-[:HAS_TAG]->(tag)
        LET
            startDate = ($date + DURATION("P100D")),
            endDate = ($date + DURATION("P200D"))
        WHERE
            m2.creationDate BETWEEN startDate AND endDate
        SELECT VALUE
            COUNT(m2.id)
    )[0]
SELECT
    tag.name                          AS tagName,
    countWindow1                      AS countWindow1,
    countWindow2                      AS countWindow2,
    ABS(countWindow1 - countWindow2)  AS diff
ORDER BY
    diff DESC,
    tagName ASC
LIMIT
    $limit;
```

bi-3.sqlpp: SNB query BI-3 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(person:Person),
        (person)<-[:HAS_MODERATOR]-(forum:Forum)-[:CONTAINER_OF]->(post:Message),
        (post)<-[:REPLY_OF*]-(message:Message),
        (message)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE]->(tagClass:TagClass)
WHERE
    country.name = $country AND
    tagClass.name = $tagClass AND
    post.isPost
GROUP BY
    forum     AS forum,
    person.id AS personId
SELECT
    forum.id                                         AS forumId,
    forum.title                                      AS title,
    UNIX_TIME_FROM_DATETIME_IN_MS(forum.creationDate) AS creationDate,
    personId                                         AS personId,
    COUNT(DISTINCT message.id)                       AS messageCount
ORDER BY
    messageCount DESC,
```

```
    forumId ASC
LIMIT
    $limit;
```

bi-4.sqlpp: SNB query BI-4 for Graphix in gSQL^{++}. This query was not used in the benchmark due to a bug that was found after evaluation. Nonetheless, we list the (now corrected) BI-4 query below to demonstrate the gSQL^{++} query model. . . . .

```
LET
    topForums = (
        FROM
            (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (country:Country)<-[:IS_PART_OF]-(c:City),
                        (c)<-[:IS_LOCATED_IN]-(member:Person),
                        (member)<-[:HAS_MEMBER]-(forum:Forum)
                WHERE
                    forum.creationDate > $date
                GROUP BY
                    forum,
                    country
                SELECT
                    forum         AS forum,
                    country       AS country,
                    COUNT(member) AS memberCount
            ) AS t
        GROUP BY
            t.forum
        SELECT VALUE
            t.forum.id
        ORDER BY
            MAX(t.memberCount) DESC
        LIMIT
            100
    )
FROM
    topForums tf,
    GRAPH SNB.Native.SNBGraph
        (person:Person)<-[:HAS_MEMBER]-(forum2:Forum)
LET
    messages = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person)<-[:HAS_CREATOR]-(message:Message),
                (message)-[:REPLY_OF*]->(post:Message)<-[:CONTAINER_OF]-(:Forum)
        WHERE
            post.isPost
        SELECT
            message.id
    )
WHERE
    tf = forum2.id
GROUP BY
    person
    GROUP AS g
LET
    messageCount = ARRAY_COUNT((FROM g, g.messages gm SELECT DISTINCT gm))
SELECT
    person.id                                         AS personId,
    person.firstName                                  AS personFirstName,
    person.lastName                                   AS personLastName,
    UNIX_TIME_FROM_DATETIME_IN_MS(person.creationDate) AS creationDate,
    messageCount                                      AS messageCount
```

```
ORDER BY
    messageCount DESC,
    personId ASC
LIMIT
    $limit;
```

bi-5.sqlpp: SNB query BI-5 for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**. There exists a semantically equivalent **LEFT MATCH** variant for this query, however the listing below was used for the benchmark...

```
FROM
    GRAPH SNB.Native.SNBGraph
        (tag:Tag WHERE tag.name = $tag)<-[:HAS_TAG]-(m:Message),
        (m)-[:HAS_CREATOR]->(person:Person),
    SNB.Native.Likes liker,
    SNB.Native.Messages comment
WHERE
    liker.messageId /*+indexnl*/ = m.id AND
    comment.replyOfMessageId /*+indexnl*/ = m.id
GROUP BY
    person.id AS personId
    GROUP AS g
LET
    messageCount = COUNT(DISTINCT m.id),
    likeCount = (FROM g SELECT VALUE COUNT(DISTINCT g.liker.personId))[0],
    replyCount = (FROM g SELECT VALUE COUNT(DISTINCT g.comment.id))[0]
SELECT
    personId,
    replyCount,
    likeCount,
    messageCount,
    (messageCount + 2 * replyCount + 10 * likeCount) AS score
ORDER BY
    score DESC,
    personId ASC
LIMIT
    $limit;
```

bi-6.sqlpp: SNB query BI-6 for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **TRUE**. There exists a semantically equivalent **LEFT MATCH** variant for this query, however the listing below was used for the benchmark...

```
FROM
    (
        FROM
            GRAPH SNB.Native.SNBGraph
                (tag:Tag WHERE tag.name = $tag),
                (tag)<-[:HAS_TAG]-(message1:Message),
                (message1)-[:HAS_CREATOR]->(person1:Person)
            LEFT JOIN
                (
                    FROM
                        SNB.Native.Likes p2lm,
                        SNB.Native.Messages p2m
                    WHERE
                        p2lm.personId = p2m.creatorId
                    SELECT
                        p2lm.messageId AS likedMessage,
                        p2m.id         AS createdMessage,
                        p2m.creatorId  AS id
```

```
                    ) p2 ON p2.likedMessage = message1.id
                LEFT JOIN
                    (
                        FROM
                            SNB.Native.Likes l
                        SELECT VALUE
                            l
                    ) p3 ON p3.messageId = p2.createdMessage
                GROUP BY
                    person1.id AS person1Id,
                    p2.id      AS person2Id
                SELECT
                    person1Id,
                    person2Id,
                    COUNT(p3.personId) AS popularityScore
    ) t
GROUP BY
    t.person1Id AS person1Id
SELECT
    person1Id              AS personId,
    SUM(t.popularityScore) AS authorityScore
ORDER BY
    authorityScore DESC,
    person1Id ASC
LIMIT
    $limit;
```

bi-7.sqlpp: SNB query BI-7 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (tag:Tag)<-[:HAS_TAG]-(:Message)<-[:REPLY_OF]-(comment:Message),
        (comment)-[:HAS_TAG]->(relatedTag:Tag)
WHERE
    tag.name = $tag AND
    NOT comment.isPost AND
    NOT EXISTS (
        FROM
            GRAPH SNB.Native.SNBGraph
                (comment)-[:HAS_TAG]->(tag)
        SELECT
            1
    )
GROUP BY
    relatedTag
SELECT
    relatedTag.name   AS tagName,
    COUNT(comment.id) AS commentCount
ORDER BY
    commentCount DESC,
    tagName ASC
LIMIT
    $limit;
```

bi-8.sqlpp: SNB query BI-8 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
LET
    personScores = (
        FROM
```

```
            (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (person:Person)-[:HAS_INTEREST]->(tag:Tag)
                WHERE
                    tag.name = $tag
                SELECT
                    person.id AS id,
                    100       AS score
                UNION ALL
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (message:Message)-[:HAS_TAG]->(tag:Tag),
                        (message)-[:HAS_CREATOR]->(person:Person)
                WHERE
                    tag.name = $tag AND
                    message.creationDate > $startDate AND
                    message.creationDate < $endDate
                GROUP BY
                    person.id AS id
                SELECT
                    id                 AS id,
                    COUNT(message.id) AS score
        ) AS t
        GROUP BY
            t.id
        SELECT
            t.id         AS id,
            SUM(t.score) AS score
    )
FROM
    GRAPH SNB.Native.SNBGraph
        (p1:Person)
    LEFT MATCH
        (p1)-[:KNOWS]->(p2:Person)
    JOIN
        personScores ps1 ON ps1.id = p1.id
    LEFT JOIN
        personScores ps2 ON ps2.id = p2.id
GROUP BY
    p1.id     AS id1,
    ps1.score AS score
SELECT
    id1             AS id,
    score           AS score,
    SUM(ps2.score) AS friendsScore
ORDER BY
    score + friendsScore DESC,
    id ASC
LIMIT
    $limit;
```

bi-9.sqlpp: SNB query BI-9 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to TRUE. . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (person:Person)<-[:HAS_CREATOR]-(post:Message),
        (post)<-[:REPLY_OF*]-(message:Message)
WHERE
    post.isPost AND
    (post.creationDate BETWEEN $startDate AND $endDate) AND
    (message.creationDate BETWEEN $startDate AND $endDate)
GROUP BY
```

```
    person
SELECT
    person.id                   AS personId,
    person.firstName            AS firstName,
    person.lastName             AS lastName,
    COUNT(DISTINCT post.id)     AS threadCount,
    COUNT(DISTINCT message.id) AS messageCount
ORDER BY
    messageCount DESC,
    personId ASC
LIMIT
    $limit;
```

bi-10.sqlpp: SNB query BI-10 for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **FALSE** and the compiler option `graphix.semantics.pattern` was set to `"edge-isomorphism"`. . . . .

```
LET
    expertCandidates = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person:Person)-[k:KNOWS{1,$maxPathDistance}]->(epc:Person)
        WHERE
            person.id = $personId
        GROUP BY
            person.id AS personId,
            epc.id    AS expertCandidatePersonId
        HAVING
            MIN(LEN(EDGES(k))) BETWEEN $minPathDistance AND $maxPathDistance
        SELECT VALUE
            expertCandidatePersonId
    )
FROM
    expertCandidates ec,
    GRAPH SNB.Native.SNBGraph
        (epc:Person)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country:Country),
        (epc)<-[:HAS_CREATOR]-(message:Message),
        (message)-[:HAS_TAG]->(:Tag)-[:HAS_TYPE]->(tagClass:TagClass),
        (message)-[:HAS_TAG]->(tag:Tag)
WHERE
    ec = epc.id AND
    country.name = $country AND
    tagClass.name = $tagClass
GROUP BY
    epc.id AS personId,
    tag    AS tag
SELECT
    personId                AS personId,
    tag.name                AS name,
    COUNT(DISTINCT message) AS messageCount
ORDER BY
    messageCount DESC,
    tag.name ASC,
    personId ASC
LIMIT
    $limit;
```

bi-11.sqlpp: SNB query BI-11 for Graphix in gSQL^{++}. For the benchmark, the compiler flag `graphix.evaluation.prefer-indexnl` was set to **FALSE** and the compiler option `graphix.semantics.pattern` was set to `"homomorphism"`. . . . .

```
FROM
    (
        FROM
            GRAPH SNB.Native.SNBGraph
                (a:Person)-[k1:KNOWS]->(b:Person)-[k2:KNOWS]->(c:Person),
                (a)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country:Country),
                (b)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
                (c)-[:IS_LOCATED_IN]->(:City)-[:IS_PART_OF]->(country),
                (c)-[k3:KNOWS]->(a)
        WHERE
            country.name = $country AND
            a.id < b.id AND
            b.id < c.id AND
            (k1.creationDate BETWEEN $startDate AND $endDate) AND
            (k2.creationDate BETWEEN $startDate AND $endDate) AND
            (k3.creationDate BETWEEN $startDate AND $endDate)
        GROUP BY
            a.id AS aid,
            b.id AS bid,
            c.id AS cid
        SELECT
            1
    ) AS g
SELECT
    COUNT(g) AS gCount;
```

bi-12.sqlpp: SNB query BI-12 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
LET
    personMessages = (
        FROM
            SNB.Native.Persons p
            LEFT JOIN
                (
                    FROM
                        GRAPH SNB.Native.SNBGraph
                            (message:Message)-[:REPLY_OF*]->(post:Message)
                    WHERE
                        message.content IS NOT NULL AND
                        message.length < $lengthThreshold AND
                        message.creationDate > $startDate AND
                        post.language IN $languages AND
                        post.isPost
                    SELECT
                        message.id,
                        message.creatorId
                ) postMessages ON postMessages.creatorId = p.id
        GROUP BY
            p.id AS personId
        SELECT
            personId               AS personId,
            COUNT(postMessages.id) AS messageCount
    )
FROM
    personMessages pm
GROUP BY
    pm.messageCount
SELECT
    pm.messageCount    AS messageCount,
    COUNT(pm.personId) AS personCount
ORDER BY
    personCount DESC,
    messageCount DESC;
```

bi-13.sqlpp: SNB query BI-13 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE** and the compiler option graphix.semantics.pattern was set to "edge-isomorphism". . . . .

```
LET
    zombies = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (country:Country)<-[:IS_PART_OF]-(:City)<-[:IS_LOCATED_IN]-(zombie:Person)
            LEFT MATCH
                (zombie)<-[:HAS_CREATOR]-(message:Message)
        WHERE
            zombie.creationDate < $endDate AND
            country.name = $country AND
            (message IS UNKNOWN OR zombie.creationDate < $endDate)
        GROUP BY
            zombie.id           AS zombieId,
            zombie.creationDate AS zombieCreationDate
        LET
            yearDiff  = GET_YEAR($endDate) - GET_YEAR(zombieCreationDate),
            monthDiff = GET_MONTH($endDate) - GET_MONTH(zombieCreationDate),
            months    = 12 * yearDiff + monthDiff + 1
        HAVING
            (COUNT(message.id) / months) < 1
        SELECT VALUE
            zombieId
    )
FROM
    zombies z,
    GRAPH SNB.Native.SNBGraph
        (zombie:Person)
LET
    zombieLikeCount = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (zombie)<-[:HAS_CREATOR]-(:Message)<-[:LIKES]-(likerZombie:Person),
            zombies lz
        WHERE
            likerZombie.id = lz AND
            likerZombie.creationDate < $endDate
        SELECT VALUE
            COUNT (DISTINCT likerZombie.id)
    )[0],
    totalLikeCount  = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (zombie)<-[:HAS_CREATOR]-(:Message)<-[:LIKES]-(likerPerson:Person)
        WHERE
            likerPerson.creationDate < $endDate
        SELECT VALUE
            COUNT(DISTINCT likerPerson.id)
    )[0],
    zombieScore = CASE
        WHEN totalLikeCount = 0
        THEN 0.0
        ELSE zombieLikeCount / totalLikeCount
    END
WHERE
    z = zombie.id
SELECT
    zombie.id       AS zombieId,
    zombieLikeCount AS zombieLikeCount,
```

```
    totalLikeCount  AS totalLikeCount,
    zombieScore     AS zombieScore
ORDER BY
    zombieScore DESC,
    zombieId ASC
LIMIT
    $limit;
```

bi-14.sqlpp: SNB query BI-14 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
FROM
    (
        FROM
            GRAPH SNB.Native.SNBGraph
                (co1:Country)<-[:IS_PART_OF]-(ci1:City)<-[:IS_LOCATED_IN]-(p1:Person),
                (co2:Country)<-[:IS_PART_OF]-(ci2:City)<-[:IS_LOCATED_IN]-(p2:Person),
                (p1)-[:KNOWS]->(p2)
        LET
            c1 = EXISTS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (p1)<-[:HAS_CREATOR]-(:Message)-[:REPLY_OF]->(m:Message),
                        (m)-[:HAS_CREATOR]->(p2)
                SELECT
                    1
            ),
            c2 = EXISTS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (p1)<-[:HAS_CREATOR]-(:Message)<-[:REPLY_OF]-(m:Message),
                        (m)-[:HAS_CREATOR]->(p2)
                SELECT
                    1
            ),
            c3 = EXISTS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (p1)-[:LIKES]->(:Message)-[:HAS_CREATOR]->(p2)
                SELECT
                    1
            ),
            c4 = EXISTS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (p1)<-[:HAS_CREATOR]-(:Message)<-[:LIKES]-(p2)
                SELECT
                    1
            ),
            c1Score   = SWITCH_CASE(c1, TRUE, 4, FALSE, 0),
            c2Score   = SWITCH_CASE(c2, TRUE, 1, FALSE, 0),
            c3Score   = SWITCH_CASE(c3, TRUE, 10, FALSE, 0),
            c4Score   = SWITCH_CASE(c4, TRUE, 1, FALSE, 0),
            c1c2Score = c1Score + c2Score,
            c3c4Score = c3Score + c4Score
        WHERE
            co1.name = $country1 AND
            co2.name = $country2
        SELECT DISTINCT
            p1.id                 AS person1Id,
            p2.id                 AS person2Id,
            ci1.name              AS city,
            c1c2Score + c3c4Score AS score
    ) AS s
```

```
GROUP BY
    s.city
    GROUP AS g
LET
    result = (FROM g SELECT VALUE g.s ORDER BY g.s.score DESC LIMIT 1)[0]
SELECT VALUE
    result
ORDER BY
    result.score DESC,
    result.person1Id ASC,
    result.person2Id ASC;
```

`bi-15.sqlpp`: SNB query `BI-15` for Graphix in gSQL^{++}. This query was not used in the benchmark due to Graphix's current lack of physical support for nested recursion (at the time of writing). Nonetheless, we list the `BI-15` query below to demonstrate the gSQL^{++} query model. . . . .

```
WITH
    GRAPH BI15Graph AS
        VERTEX (:Person)
            PRIMARY KEY (id)
            AS SNB.Native.Persons,
        EDGE (:Person)-[:KNOWS]->(:Person)
            SOURCE KEY      (startId)
            DESTINATION KEY (endId)
            AS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (personA:Person)-[:KNOWS]->(personB:Person)
                    LET
                        w1 = (
                            FROM
                                GRAPH SNB.Native.SNBGraph
                                    (personA)<-[:HAS_CREATOR]-(comment:Message),
                                    (comment)-[:REPLY_OF]->(post:Message),
                                    (post)-[:HAS_CREATOR]->(personB),
                                    (post)<-[:CONTAINER_OF]-(forum:Forum)
                            WHERE
                                NOT comment.isPost AND
                                post.isPost AND
                                forum.creationDate BETWEEN $startDate AND $endDate
                            SELECT VALUE
                                COUNT(comment)
                        )[0],
                        w2 = (
                            FROM
                                GRAPH SNB.Native.SNBGraph
                                    (personA)<-[:HAS_CREATOR]-(post:Message),
                                    (post)<-[:REPLY_OF]-(comment:Message),
                                    (comment)-[:HAS_CREATOR]->(personB),
                                    (post)<-[:CONTAINER_OF]-(forum:Forum)
                            WHERE
                                NOT comment.isPost AND
                                post.isPost AND
                                forum.creationDate BETWEEN $startDate AND $endDate
                            SELECT VALUE
                                COUNT(comment)
                        )[0],
                        w3 = (
                            FROM
                                GRAPH SNB.Native.SNBGraph
                                    (personA)<-[:HAS_CREATOR]-(c1:Message),
                                    (c1)-[:REPLY_OF]->(c2:Message),
```

```
                                    (c2)-[:HAS_CREATOR]->(personB),
                                    (c2)-[:REPLY_OF+]->(post:Message),
                                    (post)<-[:CONTAINER_OF]-(forum:Forum)
                            WHERE
                                NOT c1.isPost AND
                                NOT c2.isPost AND
                                post.isPost AND
                                forum.creationDate BETWEEN $startDate AND $endDate
                            SELECT VALUE
                                COUNT(c1)
                        )[0],
                        w4 = (
                            FROM
                                GRAPH SNB.Native.SNBGraph
                                    (personA)<-[:HAS_CREATOR]-(c2:Message),
                                    (c2)<-[:REPLY_OF]-(c1:Message),
                                    (c1)-[:HAS_CREATOR]->(personB),
                                    (c2)-[:REPLY_OF+]->(post:Message),
                                    (post)<-[:CONTAINER_OF]-(forum:Forum)
                            WHERE
                                NOT c1.isPost AND
                                NOT c2.isPost AND
                                post.isPost AND
                                forum.creationDate BETWEEN $startDate AND $endDate
                            SELECT VALUE
                                COUNT(c1)
                        )[0]
                    SELECT
                        personA.id                              AS startId,
                        personB.id                              AS endId,
                        1.0 / (w1 + w2 + (0.5 * (w3 + w4)) + 1) AS weight
            )
FROM
    GRAPH BI15Graph
        (person1:Person)-[k:KNOWS+]->(person2:Person)
LET
    weight = ARRAY_SUM((FROM EDGES(k) ke SELECT VALUE ke.weight))
WHERE
    person1.id = $person1Id AND
    person2.id = $person2Id
GROUP BY
    person1.id AS person1Id,
    person2.id AS person2Id
SELECT
    SUM(weight) AS totalWeight
LIMIT
    1;
```

bi-16.sqlpp: SNB query BI-16 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
LET
    mc1 = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person1:Person)<-[:HAS_CREATOR]-(message1:Message)-[:HAS_TAG]->(tag:Tag)
        WHERE
            tag.name = $tagA AND
            GET_YEAR(message1.creationDate) = 2012 AND
            (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (person1)<-[:KNOWS]-(person2:Person),
                        (person2)<-[:HAS_CREATOR]-(message2:Message),
```

```
                        (message2)-[:HAS_TAG]->(tag)
                WHERE
                    GET_DAY(message2.creationDate) = GET_DAY($dateA)
                SELECT VALUE
                    COUNT(DISTINCT person2.id)
            )[0] < $maxKnowsLimit
        GROUP BY
            person1.id AS id
        SELECT
            id                          AS id,
            COUNT(DISTINCT message1.id) AS messageCount
    ),
    mc2 = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (person1:Person)<-[:HAS_CREATOR]-(message1:Message)-[:HAS_TAG]->(tag:Tag)
        WHERE
            tag.name = $tagB AND
            GET_YEAR(message1.creationDate) = 2012 AND
            (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (person1)<-[:KNOWS]-(person2:Person),
                        (person2)<-[:HAS_CREATOR]-(message2:Message),
                        (message2)-[:HAS_TAG]->(tag)
                WHERE
                    GET_DAY(message2.creationDate) = GET_DAY($dateB)
                SELECT VALUE
                    COUNT(DISTINCT person2.id)
            )[0] < $maxKnowsLimit
        GROUP BY
            person1.id AS id
        SELECT
            id                          AS id,
            COUNT(DISTINCT message1.id) AS messageCount
    )
FROM
    (
        FROM
            mc1
        SELECT
            mc1.id           AS id,
            mc1.messageCount AS messageCountA,
            0                AS messageCountB
        UNION ALL
        FROM
            mc2
        SELECT
            mc2.id           AS id,
            0                AS messageCountA,
            mc2.messageCount AS messageCountB
    ) AS t
GROUP BY
    t.id
SELECT
    t.id               AS id,
    SUM(messageCountA) AS messageCountA,
    SUM(messageCountB) AS messageCountB
ORDER BY
    messageCountA + messageCountB DESC,
    t.id ASC
LIMIT
    $limit;
```

bi-17.sqlpp: SNB query BI-17 for Graphix in gSQL++. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE** and the compiler option graphix.semantics.pattern was set to "homomorphism". . . . .

```
FROM
    GRAPH SNB.Native.SNBGraph
        (tag:Tag)<-[:HAS_TAG]-(message1:Message)-[:HAS_CREATOR]->(person1:Person),
        (message1)-[:REPLY_OF*]->(post1:Message)<-[:CONTAINER_OF]-(forum1:Forum),
        (forum1)-[:HAS_MEMBER]->(person2:Person)<-[:HAS_CREATOR]-(comment:Message),
        (comment)-[:HAS_TAG]->(tag),
        (forum1)-[:HAS_MEMBER]->(person3:Person)<-[:HAS_CREATOR]-(message2:Message),
        (comment)-[:REPLY_OF]->(message2)-[:REPLY_OF*]->(post2:Message),
        (post2)<-[:CONTAINER_OF]-(forum2:Forum)
LET
    delta_d = DURATION(CONCAT("P", TO_STRING($delta), "H"))
WHERE
    post1.isPost AND
    post2.isPost AND
    NOT comment.isPost AND
    forum1.id != forum2.id AND
    person2.id != person3.id AND
    tag.name = $tag AND
    message2.creationDate > message1.creationDate + delta_d AND
    NOT EXISTS (
        FROM
            GRAPH SNB.Native.SNBGraph
                (forum2)-[:HAS_MEMBER]->(person1)
        SELECT
            1
    )
GROUP BY
    person1.id AS person1Id
SELECT
    person1Id                   AS person1Id,
    COUNT(DISTINCT message2.id) AS messageCount
ORDER BY
    messageCount DESC,
    person1Id ASC
LIMIT
    $limit;
```

bi-18.sqlpp: SNB query BI-18 for Graphix in gSQL++. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **TRUE**. . . . .

```
LET
    idPairs = (
        FROM
            GRAPH SNB.Native.SNBGraph
                (tag:Tag WHERE tag.name = $tag)<-[:HAS_INTEREST]-(person:Person),
                (person)-[:KNOWS]->(mutualFriend:Person)
        SELECT
            person.id       AS personId,
            mutualFriend.id AS friendId
    )
FROM
    idPairs idp1,
    idPairs idp2
LET
    mutualFriendId = idp1.friendId,
    person1Id = idp1.personId,
    person2Id = idp2.personId
WHERE
    idp1.friendId = idp2.friendId AND
```

```
    person1Id != person2Id AND
    NOT EXISTS (
        FROM
            SNB.Native.Knows k
        WHERE
            k.startId /*+indexnl*/ = person1Id AND
            k.endId /*+indexnl*/ = person2Id
        SELECT VALUE
            1
    )
GROUP BY
    person1Id,
    person2Id
SELECT
    person1Id                          AS person1Id,
    person2Id                          AS person2Id,
    COUNT(DISTINCT mutualFriendId) AS mutualFriendCount
ORDER BY
    mutualFriendCount DESC,
    person1Id ASC,
    person2Id ASC
LIMIT
    $limit;
```

bi-19.sqlpp: SNB query BI-19 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
WITH
    GRAPH BIGraph19 AS
        VERTEX (:Person)
            PRIMARY KEY (id)
            AS SNB.Native.Persons,
        EDGE (:Person)-[:KNOWS]->(:Person)
            SOURCE KEY      (startId)
            DESTINATION KEY (endId)
            AS (
                FROM
                    (
                        FROM
                            SNB.Native.Messages m1,
                            SNB.Native.Messages m2
                        WHERE
                            m1.replyOfMessageId = m2.id
                        SELECT
                            m1.creatorId AS startId,
                            m2.creatorId AS endId,
                            m1.isPost    AS m1IsPost,
                            m2.isPost    AS m2IsPost
                        UNION ALL
                        FROM
                            SNB.Native.Messages m1,
                            SNB.Native.Messages m2
                        WHERE
                            m2.replyOfMessageId = m1.id
                        SELECT
                            m1.creatorId AS startId,
                            m2.creatorId AS endId,
                            m1.isPost    AS m1IsPost,
                            m2.isPost    AS m2IsPost
                    ) AS m12,
                    SNB.Native.Knows k
                WHERE
                    k.startId = m12.startId AND
                    k.endId = m12.endId
```

```
                GROUP BY
                    k.startId AS aid,
                    k.endId AS bid
                SELECT
                    aid             AS startId,
                    bid             AS endId,
                    1.0 / COUNT(*)  AS weight
            )
FROM
    SNB.Native.Persons person1A,
    GRAPH BIGraph19
        (person1B:Person)-[k:KNOWS+]->(person2B:Person),
    SNB.Native.Persons person2A
WHERE
    person1A.placeId = $city1Id AND
    person1B.placeId = $city2Id AND
    person1A.id = person1B.id AND
    person2A.id = person2B.id
GROUP BY
    person1B.id AS id1,
    person2B.id AS id2
    GROUP AS g
LET
    cheapestPathWeight = (
        FROM
            g
        LET
            cost = (FROM EDGES(g.k) ke SELECT VALUE SUM(ke.weight))[0]
        SELECT VALUE
            cost
        ORDER BY
            ABS(cost) ASC
        LIMIT
            1
    )[0]
SELECT
    id1                AS person1id,
    id2                AS person2id,
    cheapestPathWeight AS totalWeight
ORDER BY
    person1id ASC,
    person2id ASC;
```

bi-20.sqlpp: SNB query BI-20 for Graphix in gSQL^{++}. For the benchmark, the compiler flag graphix.evaluation.prefer-indexnl was set to **FALSE**. . . . .

```
WITH
    GRAPH BIGraph20 AS
        VERTEX (:Person)
            PRIMARY KEY (id)
            AS SNB.Native.Persons,
        EDGE (:Person)-[:KNOWS]->(:Person)
            SOURCE KEY      (startId)
            DESTINATION KEY (endId)
            AS (
                FROM
                    GRAPH SNB.Native.SNBGraph
                        (personA:Person)-[:KNOWS]->(personB:Person),
                        (personA)-[saA:STUDY_AT]->(:University)<-[saB:STUDY_AT]-(personB)
                GROUP BY
                    personA.id AS aid,
                    personB.id AS bid
                    GROUP AS g
                LET
```

```
                    weight = (
                        FROM
                            g
                        SELECT VALUE
                            MIN(ABS(g.saA.classYear - g.saB.classYear)) + 1
                    )[0]
                SELECT
                    aid    AS startId,
                    bid    AS endId,
                    weight AS weight
            )
FROM
    GRAPH BIGraph20
        (person2A:Person WHERE person2A.id = $person2Id)<-[k:KNOWS+]-(person1A:Person),
    GRAPH SNB.Native.SNBGraph
        (person1B:Person)-[:WORK_AT]->(company:Company)
WHERE
    person1A.id = person1B.id AND
    company.name = $company
GROUP BY
    person1A.id AS id1,
    person2A.id AS id2
    GROUP AS g
LET
    cheapestPath = (
        FROM
            g
        LET
            cost = (FROM EDGES(g.k) ke SELECT VALUE SUM(ke.weight))[0]
        SELECT VALUE
            cost
        ORDER BY
            ABS(cost) ASC
        LIMIT
            1
    )[0]
SELECT
    id1                AS person1Id,
    cheapestPath.cost AS totalWeight
ORDER BY
    totalWeight ASC,
    person1Id ASC
LIMIT
    1;
```