Graphix: "One User's JSON is Another User's Graph"

Glenn Galvizo
University of California, Irvine
Email: ggalvizo@uci.edu

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

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00238

***Abstract*—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.**

In this paper, 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. We explain a) the graph view user model of Graphix, b) gSQL^{++}, a novel query language extension for synergistic document-based navigational pattern matching, and c) how edge hops are evaluated in a parallel fashion. We then compare queries authored in gSQL^{++} against versions in other leading query languages. Finally, we evaluate our approach against a leading native graph database, Neo4j, and show that Graphix is appropriate for operational and analytical workloads, especially at scale.

***Index Terms*—Big Data, JSON, graph analytics, graph query languages, navigational pattern matching, AsterixDB**

I. 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 [1]. 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 architecture composed of a document database `DBD` and a graph database `GDB` to analyze the relationships found in `DBD`. This architecture has several consequences: a) some form of ETL (extract-transform-load) pipeline must be developed to duplicate the data from database `DBD` to `GDB` and then maintained to ensure consistency, b) separate resources need to be allocated for both `DBD` and `GDB` (increasing the cost to own the data), c) queries are written in two query languages for two different APIs, and d) heterogeneous workloads (e.g. mixing graph and document analytics) require ad-hoc, specialized solutions. Furthermore, graph databases like Neo4j are limited by their inability to gracefully 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 paper, 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 modify their graph queries when scaling horizontally. Users *should not* have to sacrifice performance to realize all of the aforementioned points.

We find that most existing solutions satisfy at most $1/3$ 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. 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 efficient, partitioned-parallel executions, and 4) a code-complexity study and an initial performance comparison with a native graph database.

The rest of this paper is structured as follows: Section II describes related work around querying graph data. Section III reviews Apache AsterixDB, the Big Data management system used for this research, and two query languages: SQL^{++} and Cypher. Section IV introduces the graph model of Graphix, demonstrating how users can map existing document data to a non-materialized graph view. Section V details our query model and query language: gSQL^{++}. Section VI explains the implementation and architecture of Graphix. Section VII details a preliminary evaluation of Graphix's query model and

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00238

Graphix's performance against Neo4j.

II. 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 section 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.

A. Graph Processing Systems

Big Graph processing systems such as Pregel [2] and Giraph [3] 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 [4], 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 [5] and GiraphUC [6] were designed to process large graphs in an asynchronous manner. A graph processing system that used the same runtime engine as Graphix is Pregelix [7], designed to gracefully scale distributed graph algorithms for out-of-core workloads. While Big Graph processing systems have been shown to be performant and scaleable [8], 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.

B. Native Graph Databases

Native graph databases like Neo4j [9] and TigerGraph [10] were designed to challenge traditional database systems by building a new database from the ground-up (storage, execution, and user model) with graph primitives in mind. Amazon Neptune [11], 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 are supported by all three of the aforementioned systems [12]. With respect to the query model of graph databases, there are two leading query languages for the property graph model – Cypher [13] and Gremlin [14] – and one standardized language for the RDF model – namely SPARQL [15].

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 *gracefully*[1] 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.

C. 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 ([16], [17]), 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 1.

Unipop Graph [18] and Cytosm [13] 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 years. OBDA (ontology-based data access) is a related problem that pertains to converting SPARQL queries into queries against various existing data sources [19], [20], [21]. Graphix is not middleware and is more tightly coupled with 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 [22], DataStax Enterprise Graph [23], and IBM Db2 Graph [24]. 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

[1]Neo4j has the ability issue queries across a federation of multiple graphs, but the Cypher queries that Neo4j users must write are *not* shard-agnostic like TigerGraph and Amazon Neptune.

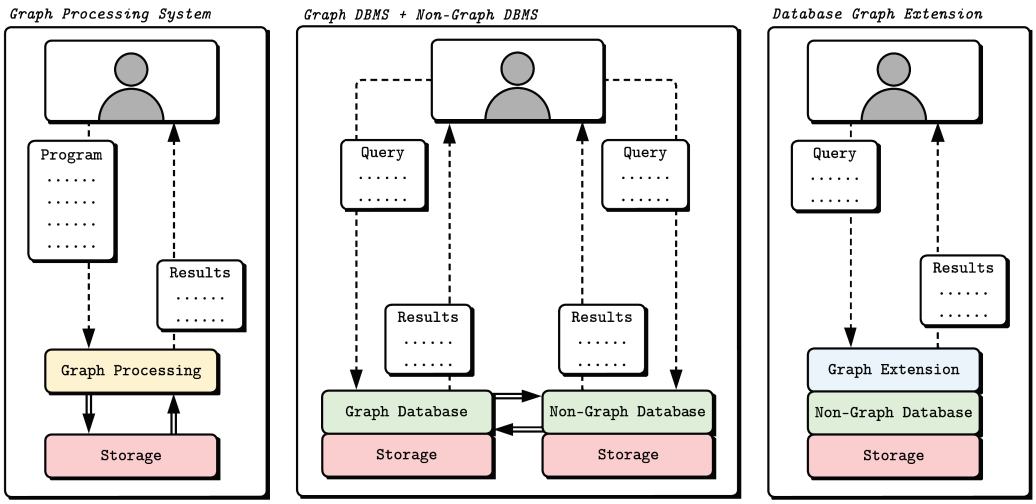


Fig. 1: Comparison between graph processing systems, graph + non-graph database systems, and database graph extensions.

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.

III. Background

In this section we give overviews of AsterixDB, of the query language of AsterixDB (SQL^{++}), and of the leading graph query language (Cypher).

A. Apache AsterixDB

AsterixDB is a Big Data management system (BDMS) designed to be a highly scalable platform for document storage, search, and analytics [25]. AsterixDB possesses a flexible, semi-structured data model that accommodates a range of use cases —from "schema-first" to "schema-never". 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 [26]. This optimized physical plan is then translated into a job that can run across all nodes in the cluster [27]. Datasets in AsterixDB are hash-partitioned across the cluster on their primary key into primary B+ tree indexes, where the data records reside, with secondary indexes being local to the primary data on each node.

```
1 FROM   Users u
2 WHERE  u.name IS NOT NULL
3 SELECT u.id AS uid,
4        ( FROM   Messages m
5          WHERE  m.user_id = u.id
6          SELECT m.id AS id ) AS mids;
```

Listing 1: SQL^{++} query that correlates two datasets in the `SELECT` clause.

B. SQL++ Query Language

SQL^{++} is a query language purposed for JSON, semi-structured data, while being backwards-compatible with SQL [28], [29]. This backwards compatibility enables easy adoption by existing SQL users. 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 find users with non-`NULL` names and all messages they have written. A legal way to express this query in SQL^{++} is given in Listing 1. This query in Listing 1 illustrates two features of SQL^{++} that are not present in SQL: 1) 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 in this paper. 2) In

SQL^{++}, subqueries can be used to build nested documents. In Listing 1, we use a subquery to create records containing arrays of related message IDs. The result of executing the query might yield the four results below.

```
{uid: 11, mids: [{id: 9992},{id: 9997}]}
{uid: 14, mids: []}
{uid: 27, mids: [{id: 10010}]}
{uid: 70, mids: [{id: 10524}]}
```

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 a SQL `GROUP BY` clause, which only allows reasoning over aggregate values of groups. Suppose we want to group all `Users` by their first name and return the *groups* of user IDs for groups that have less than 4 elements. We can use the SQL^{++} query in Listing 2 to realize this grouping. Executing Listing 2 might yield the two results below.

```
{name: "Gura", uids: [{id: 6}]}
{name: "Ame",  uids: [{id: 3},{id: 10}]}
```

Given that SQL^{++} is the query language used by AsterixDB, SQL^{++} also serves as the foundation for the query language of Graphix, gSQL^{++}.

C. Cypher Query Language

Cypher is arguably the current leader for querying property graphs [13], though there is a growing effort to standardize [30], [31] and bridge the gap between other similar query languages [32], [33]. A defining characteristic of Cypher is its `MATCH` clause, allowing users to specify navigational graph patterns via a user-friendly ASCII-art syntax. Recursion in Cypher is enabled through the use of regular expressions between vertices in graph patterns. While not as computationally powerful as the Pregel model – or the recursive SQL-99 standard [34] – graph computations such as reachability and shortest path can be written in a much more succinct and natural manner in Cypher.

To illustrate the simplicity of Cypher, let us start by describing a recursive SQL query to find if three users are transitively connected to each other. Beginning on Line 2 in Listing 3, we start by anchoring the navigation at `$id1` and 1) grabbing the IDs for the next user to visit (`luk`), 2) initializing an array for cycle detection (`vu`) and 3) and an output array (`v`). Subsequent iterations will execute the recursive member on Line 8, which will "traverse" to another user `u2` using the user IDs `luk` from the previous iteration. To avoid traversing over cycles, we check if the ID of our current user is in our visited array `vu`. If our current user has one of the IDs we are interested in, we update our output array accordingly by performing a bitwise `OR` operation with our current output array. The results that our recursive member yields to the next iteration includes the next set of user ids, an updated visited array to include `u2`, and the status of our output array. If we find any results from our recursive CTE such that our output array has a length of 3, then we know that we have visited all three users of interest at some point. Otherwise, we conclude that there exists no path that connects `$id1`, `$id2`, and `$id3`.

```
1 FROM     Users u
2 GROUP BY u.name.first GROUP AS g
3 HAVING   COUNT(*) < 4
4 SELECT   u.name.first AS name,
5          ( FROM g SELECT g.u.id ) AS uids;
```

Listing 2: SQL^{++} `GROUP AS` query to return groups formed by a `GROUP BY` clause.

```
1  WITH RECURSIVE Visited AS
2      ( SELECT u1.knows      AS luk
3               ARRAY[u1.id] AS vu,
4               ARRAY[1,0,0] AS v
5        FROM   Users u1
6        WHERE  u1.id = $id1
7        UNION ALL
8        SELECT u2.knows        AS luk,
9               rv.vu || u2.id AS vu,
10              CASE
11                WHEN u2.id = $id2
12                THEN ARRAY[rv.v[0],1,rv.v[2]]
13                WHEN u2.id = $id3
14                THEN ARRAY[rv.v[0],rv.v[1],1]
15                ELSE rv.v
16              END AS v
17       FROM   Visited rv,
18              Users u2
19       WHERE  u2.id = ANY(rv.luk) AND
20              NOT u2.id = ANY(rv.vu) )
21 SELECT COUNT(*) > 0 AS connected
22 FROM   Visited rv
23 WHERE  ( SELECT SUM(v) = 3
24          FROM   UNNEST (rv.v) v );
```

Listing 3: Recursive SQL query (in PostgreSQL's dialect) to find if three users are transitively connected to each other.

To get around the short-term memory restriction inherent to recursive CTEs, Listing 3 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 our 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.

We contrast the query in Listing 3 with the much easier-to-read equivalent Cypher query in Listing 4. 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 3 starts the navigation at `$u1`. In the Cypher query, a user does not specify

```
1 MATCH   (u1:User)-[:KNOWS*]-(u2:User),
2         (u2)-[:KNOWS*]-(u3:User),
3         (u1)-[:KNOWS*]-(u3)
4 WHERE   u1.id = $u1 AND
5         u2.id = $u2 AND
6         u3.id = $u3
7 RETURN  COUNT(*) > 0 AS connected;
```

Listing 4: Cypher query to find if three users are transitively connected to each other.

a starting point, allowing the query optimizer to (more easily) choose an appropriate starting point.

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 desiredata, adopting Cypher as a second language for Graphix would require users to write queries in two different query languages. Furthermore, SQL is *the* defacto 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.[2]

IV. Graph Model of Graphix

We now introduce Graphix, which was designed to work *in tandem* with existing user models. To illustrate the user-facing graph model behind Graphix, we start by modeling a social network using the document model of AsterixDB as an application might do. We then describe the social network with the graph model of Graphix, and show how to establish a mapping between Graphix and AsterixDB.

A. Social Network Example

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 other `User(s)`. Examples of these entities and relationships are given in Figure 2. We highlight two 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 `Users` documents, and 2) many-to-many relationships can be folded into a single entity, as shown by the `knows` arrays of two `Users` documents.

[2]The recently released SQL/PGQ standard closely aligns with our idea of merging SQL with graph queries, but a SQL query fundamentally revolves around a structured table. SQL/PGQ draws a clear "line in the sand" between the relational world and the graph world. Section V will show how Graphix moves beyond this model limitation.

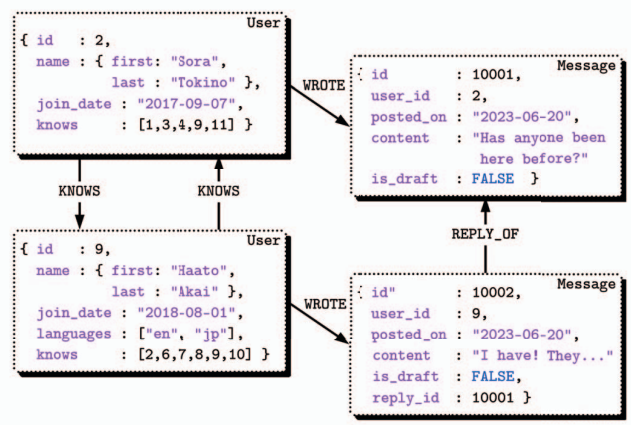


Fig. 2: Example documents of two `Users`, two `Messages`, and their relationships.

B. Mapping to Graphix

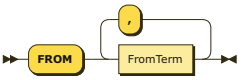
Having defined these two datasets, we will now define a mapping of these datasets to a virtual property graph that we can formulate graph queries over. In Graphix, *both* vertices and edges correspond to documents in AsterixDB. An instance of a vertex contains two sets of fields: (a) a set of fields that are denoted (but not enforced) as its *primary key*, and (b) an optional set of fields that correspond to the other properties of the vertex. An instance of an edge in Graphix is always directed, which allows us to define an edge in three distinct parts: (i) a set of fields that form a reference to a source vertex, known as the *edge source key*, (ii) a set of fields that form a reference to a destination vertex, known as the *edge destination key*, and (iii) an optional set of fields that correspond to the properties of an edge.

Listing 5 describes our mapping of the `Users` and `Messages` datasets to the property graph `SocialNetworkGraph`, which is composed of two types of vertices and three types of edges:

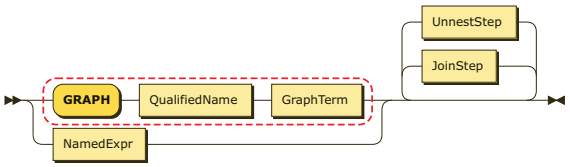
1) Starting on Line 2, we define the collection of all vertices labeled `User` to be the dataset `Users`. The primary key of the `User` vertex collection is the primary key of the `Users` dataset: `id`. The properties of an individual `User` vertex are all the fields of the mapped `Users` document.
2) 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` records 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 10, 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 de-

```
1  CREATE GRAPH SocialNetworkGraph
2  AS VERTEX (:User)
3     PRIMARY KEY (id)
4     AS Users,
5     VERTEX (:Message)
6     PRIMARY KEY (id)
7     AS ( FROM   Messages m
8          WHERE  NOT m.is_draft
9          SELECT m.* ),
10    EDGE (:User)-[:KNOWS]->(:User)
11    SOURCE KEY (source_id)
12    DESTINATION KEY (dest_id)
13    AS ( FROM   Users u
14         UNNEST u.knows k
15         SELECT u.id AS source_id,
16                k AS dest_id ),
17    EDGE (:User)-[:WROTE]->(:Message)
18    SOURCE KEY (user_id)
19    DESTINATION KEY (message_id)
20    AS ( FROM   Messages m
21         SELECT m.user_id,
22                m.id AS message_id,
23                m.posted_on ),
24    EDGE (:Message)-[:REPLY_OF]->(:Message)
25    SOURCE KEY (source_id)
26    DESTINATION KEY (dest_id)
27    AS ( FROM   Messages m
28         SELECT m.id AS source_id,
29                m.reply_id AS dest_id,
30                m.posted_on );
```

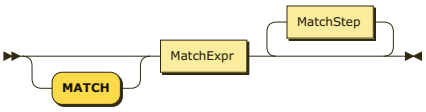
Listing 5: `CREATE GRAPH` DDL to create a property graph view.



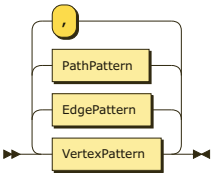
(a) SQL^{++} grammar for the `FromClause` production.



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



(c) gSQL^{++} `GraphTerm` grammar extension.



(d) gSQL^{++} `MatchExpr` grammar extension.

Fig. 3: Simplified gSQL^{++} grammar extension for the `FromTerm` production. The full grammar (describing `UnnestStep`, `JoinStep`, and `MatchStep`) is available at: https://graphix.ics.uci.edu/docs/language-reference/

fined 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 here for `KNOWS` edges. This edge mapping demonstrates a natural approach to handle relationships that are captured by arrays: we can 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 17, 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 24, 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.

We note that while Listing 5 may seem verbose, the users that author `CREATE GRAPH` statements are intended to be a subset of the users that actually query the graphs. Once Listing 5 is executed, other data analysts can simply "put on their graph glasses" and then query the existing data in-situ accordingly, i.e., as a graph.

V. Query Model of gSQL++

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).

A. SQL++ Plus Navigational Pattern Matching

We now move to gSQL^{++}, a SQL^{++} extension that enables the integration of graph pattern matching (borrowed from 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 are 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^{++}

```
1 FROM      GRAPH SocialNetworkGraph
2           (u:User)-[:WROTE]->(m:Message)
3 WHERE     m.content LIKE "%"+u.name.first+"%"
4 GROUP BY u
5 HAVING    COUNT(u) > 10
6 SELECT    u;
```

Listing 6: gSQL^{++} query to find users that have written more than 10 messages with their name.

is therefore in the `FromTerm`. Our intent with Graphix was to make gSQL^{++} a *strict* superset of SQL^{++}. As indicated by the lower path in Figure 3b, all SQL^{++} queries are valid gSQL^{++} queries. Users follow the bottom path to express a standard SQL^{++} `FromTerm`. To express a gSQL^{++} matching `FromTerm`, users follow the top path (the grammar surrounded by the red dashed lines) and specify:

1) the `GRAPH` keyword;
2) the name of the graph (i.e. `QualifiedName`); and
3) the graph query patterns (i.e. `GraphTerm`).

Users that are familiar with Cypher can optionally specify the `MATCH` keyword before a `GraphTerm` expression. Logically, after the `GraphTerm` clause is evaluated, users have a multi-set of documents to reason about. These documents can then be manipulated using the same SQL and SQL^{++} clauses that users are already familiar with: the documents can be filtered using a `WHERE` clause, `JOIN`ed with another collection of documents, aggregated through `GROUP BY`, or operated on using any other legal SQL^{++} expression.

B. gSQL++ Through Example Queries

To illustrate the expressiveness of gSQL^{++}, let us consider several queries on our graph from Subsection IV-B. For our first query, we want to find users that have written more than 10 messages whose content includes their name. We express this first gSQL^{++} query in Listing 6, where our first line specifies the name of our graph `SocialNetworkGraph` in the `FROM` clause. The next line is our `GraphTerm`, which specifies the query pattern "users `u` that have written messages `m`". This `FROM` clause and `GraphTerm` expression is equivalent to the following `FROM` and `JOIN` clauses in SQL^{++}:

```
FROM Messages m JOIN Users u ON m.user_id = u.id
```

For the rest of the query shown in Listing 6, we can operate on the bound variables `m` and `u` as if they came from the `FROM` and `JOIN` clauses above. Starting on Line 3, we specify the "message with a user's name" condition. For all users and messages, we then `GROUP BY` users on Line 4 and filter out groups with 10 or less messages using the `HAVING` clause on Line 5.

For our second query, we are interested in finding out which messages socially isolated users are currently engaging with. For all week-old (or less) messages `m1` written by users `u` that do not know any other users, we want to find all *top-level* messages `m2` that `m1` is a reply to. "Top-level" in the context of this query means that we want to recursively follow the reply-of chain. To find these top-level messages, we utilize the concept of a *path*. A path in gSQL^{++} is a sequence of one or more edges and is specified using a regular expression of edge labels. If two messages `m1` and `m2` are connected using a path `r` of one or more `REPLY_OF` edges, we capture this sequence of edges using the pattern expression below. Note the use of the positive closure quantifier `+` after the `REPLY_OF` edge label.

```
(m1:Message)-[r:REPLY_OF+]->(m2:Message)
```

An instance of a path can logically be thought of as a single document containing two array-valued fields: `Vertices` and `Edges`. Users are free to manipulate these arrays using SQL^{++}. An example of a two-hop path of `REPLY_OF` edges is given in Listing 7.

Having described the notation for a path, we express our second query in gSQL^{++} in Listing 8. Line 2 again identifies messages and their authors. Line 3 asks for all messages `m2` that `m1` is a direct or transitive reply to. If there are multiple paths between the same `m1` and `m2` vertex, then *all paths* are

```
{
  Vertices: [
    { id         : 10423,
      user_id    : 4,
      posted_on  : "2023-06-21"
      content    : "Thanks! I'll check it out!",
      is_draft   : FALSE,
      reply_of   : 10420 },
    { id         : 10420,
      user_id    : 8,
      posted_on  : "2023-06-20"
      content    : "I've been there! They..."
      is_draft   : FALSE,
      reply_of   : 10419 },
    { id         : 10419,
      user_id    : 4,
      posted_on  : "2023-06-20",
      content    : "Has anyone been to Rail...",
      is_draft   : FALSE } ],
  Edges: [
    { source_id : 10423,
      dest_id   : 10420,
      posted_on : "2023-06-21" },
    { source_id : 10420,
      dest_id   : 10419,
      posted_on : "2023-06-20" } ]
}
```

Listing 7: JSON document describing a two-hop path of `REPLY_OF` edges in the graph `SocialNetworkGraph`.

```
1 FROM    GRAPH SocialNetworkGraph
2         (u:User)-[w:WROTE]->(m1:Message),
3         (m1)-[r:REPLY_OF+]->(m2:Message)
4 LET     lw = CURRENT_DATE()-DURATION("P7D")
5 WHERE   w.posted_on > lw AND
6         m2.reply_id IS UNKNOWN AND
7         NOT EXISTS (
8           FROM    GRAPH SocialNetworkGraph
9                   (u)-[:KNOWS]->(:User)
10          SELECT *
11        )
12 SELECT DISTINCT m2;
```

Listing 8: gSQL^{++} query to find the top-level messages that socially isolated users are engaging with.

```
1  FROM     GRAPH SocialNetworkGraph
2           (u1:User)-[k:KNOWS+]->(u2:User)
3  WHERE    u1.id = $u1 AND
4           u2.id IN $D
5  GROUP BY u2 GROUP AS g
6  SELECT   u2.id,
7           ( FROM     g
8             SELECT   g.k
9             ORDER BY LEN(g.k.Edges) ASC
10            LIMIT    1
11          )[0] AS shortest_path;
```

Listing 9: gSQL^{++} query to find the shortest paths from one user to a collection of other users.

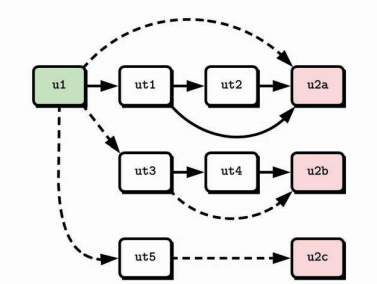


Fig. 4: Example graph of `Users` and `KNOWS` edges. The dashed lines represent the shortest paths from u_1 to u_{2_a}, u_{2_b}, and u_{2_c}.

available for the user to manipulate. We are only interested in the top-level messages for our query, so we specify that the `reply_id` field of `m2` should be `UNKNOWN` (i.e. `NULL` or absent from `m2` entirely) in the `WHERE` clause. Finally we reach a correlated anti-`JOIN` query on Line 7, which expresses the condition that user `u` does not know any other user.

The last query that we will ask involves finding a shortest path from a single user `$u1` to each user in a set of other users `$D`. This third gSQL^{++} query is expressed in Listing 9. On Line 2, the path `k` between two users `u1` and `u2` is represented using the path expression `:KNOWS+`. Our navigation is anchored using the subsequent `WHERE` clause, which qualifies the primary keys of both `u1` and `u2`. The `GROUP BY` clause in Line 5 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`, we use the subquery in Line 7. Due to the `GROUP BY` clause, this subquery is logically executed for each `u2` instance: in ascending order, we sort each path `g.k` from `u1` to a `u2` instance by the number of hops in `g.k` and choose the shortest path (or one of the shortest paths, if there are ties). To quantify the hops in a path, `LEN(g.k.Edges)` is used to count the number of edges a given path possesses. Finally, the `[0]` on the last line is used to access the sole element returned by the subquery. In SQL^{++} / gSQL^{++}, subqueries will always return a multiset. `ORDER BY` subqueries return an array, hence the need for the array access [29]. By utilizing the existing `GROUP BY` ... `GROUP AS` clauses of SQL^{++}, gSQL^{++} is able to naturally express a rich set of navigation constraints in a novel manner not found in any other existing graph query language.

```
1 {u1: u1, u2: u2a, k: u1 -KNOWS-> u2a}
2 {u1: u1, u2: u2a, k: u1 -KNOWS-> ut1 -KNOWS-> u2a}
3 {u1: u1, u2: u2a, k: u1 -KNOWS-> ut1 -KNOWS-> ut2 -KNOWS-> u2a}
4 {u1: u1, u2: u2b, k: u1 -KNOWS-> ut3 -KNOWS-> u2b}
5 {u1: u1, u2: u2b, k: u1 -KNOWS-> ut3 -KNOWS-> ut4 -KNOWS-> u2b}
6 {u1: u1, u2: u2c, k: u1 -KNOWS-> ut5 -KNOWS-> u2c}
```

Listing 10: Example records in scope after the `FROM` clause but before the `GROUP BY` clause of Listing 9.

```
1 {u1: u1, u2: u2a, k: u1 -KNOWS-> u2a}
2 {u1: u1, u2: u2a, k: u1 -KNOWS-> ut1 -KNOWS-> u2a}
3 {u1: u1, u2: u2a, k: u1 -KNOWS-> ut1 -KNOWS-> ut2 -KNOWS-> u2a}

1 {u1: u1, u2: u2b, k: u1 -KNOWS-> ut3 -KNOWS-> u2b}
2 {u1: u1, u2: u2b, k: u1 -KNOWS-> ut3 -KNOWS-> ut4 -KNOWS-> u2b}

1 {u1: u1, u2: u2c, k: u1 -KNOWS-> ut4 -KNOWS-> u2c}
```

Listing 11: Example groups in scope (for use in the `SELECT` clause subquery) after `GROUP BY` clause of Listing 9.

To better illustrate the functionality of Line 7's subquery, suppose that the query logically matches the paths in Listing 10 from user u_1 to three separate users $\{u_{2_a}, u_{2_b}, u_{2_c}\} \in$ `$D`. A visual representation of all possible paths from u_1 to all users in `$D` is given in Figure 4. The `GROUP BY` in the outer query generates three collections of documents, illustrated by the grouping in Listing 11. Finally, the 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 records in Listing 11).

VI. Implementation of Graphix

Graphix was designed as an extension of AsterixDB. In addition to extending AsterixDB's query language, Graphix also extends AsterixDB's query optimizer [26] and AsterixDB's parallel runtime engine [27]. Given a single gSQL^{++} query Q, 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 *lowered* 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 SQL^{++} AST rewrites (e.g. `WITH` clause inlining, `GROUPING SETS`, etc...). For historical reasons, these AST rewrites are separate

from our 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 detailed graph of *activities* [27] (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 requirements of $J(Q)$. These groups are known as *stages*.
8) Iterating through each stage, the cluster controller process then distributes a 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 handed back to the user.

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 (e.g., using index-nested-loop-`JOIN` to traverse edges, factoring out loop invariants, etc...). 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.

A. Execution of Graph Queries

We will now take a look at how edge hops are physically executed for traversing (non-materialized) Graphix graphs in a partitioned-parallel manner. More specifically, we will be looking at how Graphix would execute the query in Listing 12. Graphix extends Hyracks to leverage the concepts in this section in order to support recursive gSQL^{++} queries. These Hyracks extensions include a) a new in-band message passing paradigm to characterize the progress of a looping activity graph instance and b) a novel `FIXED POINT` Hyracks operator for coordinating the termination of parallel task clusters. These

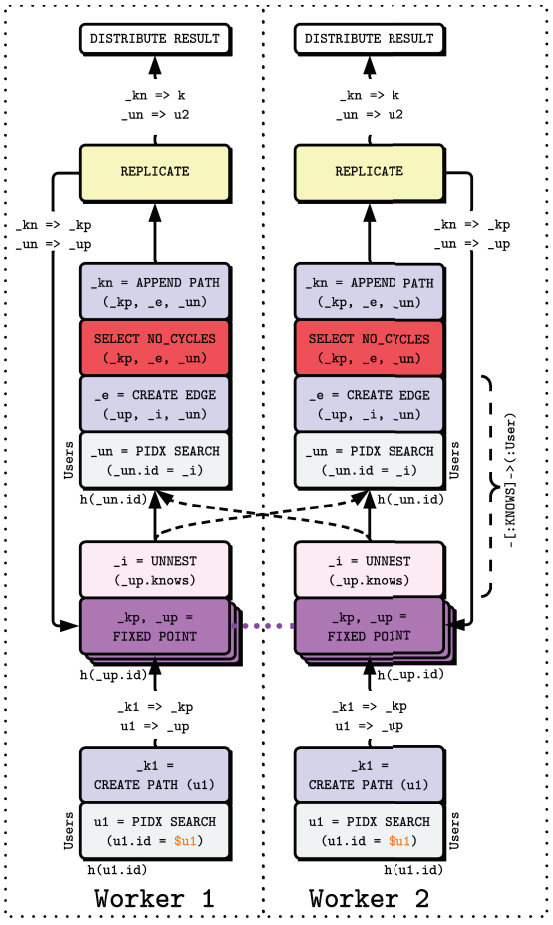


Fig. 5: Two-worker execution plan to realize the query in Listing 12 using index-nested-loop-`JOIN`s to traverse edges.

```
1 FROM    GRAPH SocialNetworkGraph
2         (u1:User)-[k:KNOWS+]->(u2:User)
3 WHERE   u1.id = $u1
4 SELECT k, u2;
```

Listing 12: gSQL^{++} query to find all paths composed of `KNOWS` edges from a single user `u1` to all other connected users `u2`.

extensions are described in more detail in Chapter 5 of [35] and a forthcoming paper [36].

Suppose that we translate and execute the query in Listing 12 on a two-worker Graphix cluster, where the dataset corresponding to the `User` vertex set, `Users`, is hash-partitioned across both workers on the `Users` primary key: the `id` field. Figure 5 describes a potential execution plan where two workers perform the same job on their different partitions of data. The support for explicitly cyclic plans, the `FIXED POINT` operator, and the path functions (i.e., `CREATE PATH`, `CREATE EDGE`, and `APPEND PATH`) are unique to Graphix.

Similar to how recursive SQL queries are structured, Graphix-based navigation in Hyracks is composed of two parts: 1) an *anchor* data flow, which describes how starting ver-

tices are found, and 2) a *recursive* data flow, which describes how graph edges are traversed. The bottom two operators of Figure 5 describe the anchor data flow, while the remaining operators (up to the REPLICATE operator) describe the recursive data flow. Starting from the bottom operators, we begin our execution by finding our starting user u1. We perform a search on the primary index such that the primary key of Users is equal the starting ID $u1. A starting path _k1 of one vertex (u1) and zero edges is then created in the next operator. The 2-tuple ⟨_k1, u1⟩ is then forwarded to the FIXED POINT operator to anchor the navigation.

The next operator in Figure 5 is the FIXED POINT operator, which possesses two operator inputs and one operator output. At a high level, the FIXED POINT operator is akin to a UNION ALL operator (see [35] and [36] for a more in-depth explanation) and is used to "merge" two data flows (i.e., the anchor data flow and the recursive data flow) into one. To ensure that the recursive data flow is a) lively (progress is being made), b) safe (resources are never over-allocated), and c) mortal (the data flow eventually terminates), FIXED POINT operator instances will communicate various events to each other using an out-of-band channel (as denoted by the purple dotted line). The output of the FIXED POINT operator is a 2-tuple ⟨_kp, _up⟩, where _kp and _up represent a path and a user from a previous iteration respectively.

To get the users known by the user _up, the knows array is UNNESTed to produce Users foreign keys _i. Because the Users dataset is hash partitioned across multiple workers, Hyracks needs to make sure that each ⟨_kp, _up⟩ tuple is sent to the data shard that might contain the user _un whose id field is equal to the tuple's _i. Using the same hash function that was used to partition the Users dataset (denoted as h in our diagram), each worker will hash the PIDX SEARCH field _i and possibly forward some of the tuples to the other worker across the network. The UNNEST and PIDX SEARCH operators define the traversal across a KNOWS edge to a User vertex in our conceptual SocialNetworkGraph graph. In the absence of an index on the id field of the Users dataset, a hybrid hash JOIN approach (not depicted) would be used instead of the depicted index nested loop JOIN approach.

Using the fields _up, _i, and _un, an edge _e is created and used to check whether or not adding the edge _e to the previous path _kp induces a cycle. If adding _e to _kp does not induce a cycle, then a new path _kn is constructed. The 2-tuple ⟨_kn, _un⟩ is then duplicated via the REPLICATE operator and sent to two places: 1) downstream (up in Figure 5) to the DISTRIBUTE RESULT operator (with _kn and _un being renamed to k and u2 respectively), and 2) back to the FIXED POINT operator (with _kn and _un being renamed to _kp and _up respectively) to repeat another KNOWS edge traversal. Once all simple paths between $u1 and every other user have been enumerated, our execution terminates. Moving beyond Figure 5, Graphix is able to compile and evaluate gSQL^{++} queries about conceptual graphs over their underlying existing data using *any* AsterixDB cluster configuration of *any* size and partitioning.

VII. Preliminary Evaluation

To illustrate the potential of Graphix, we describe two sets of preliminary analysis: 1) a code-complexity analysis of gSQL^{++}, Cypher, PostgreSQL, SPARQL, and TigerGraph GSQL queries, and 2) a preliminary performance evaluation comparing the current version of Graphix with a leading graph database that presents a similar graph user model: 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 competitive performance for many queries in this section). Nonetheless, we report our "proof-of-concept" findings below.

A. Code Complexity Analysis

For our code-complexity evaluation, we quantify the "effort" of *authoring* a query. Inspired by work from Vashistha [37] as well as Goretity and Reguly [38], we are interested in determining the volume (V), difficulty (D), and effort (E) of a query using the query's *operators* (N_1 = the total number of operators, η_1 = the distinct number of operators) and the query's *operands* (N_2 = the total number of operands, η_2 = the distinct number of operands) [39]. Take the following SQL query snippet: "SELECT a, b, c". We would consider SELECT as an *operator* and the fields a, b, and c to be the *operands* of the "SELECT operator". By counting the number of operators and operands in a query, we define volume as $V = (N_1+N_2) \cdot \log_2(\eta_1+\eta_2)$, difficulty as $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$, and effort as $E = DV$. Intuitively, queries with higher V values are more verbose than queries with lower V values. Queries with higher D values are more "difficult-to-understand" than queries with lower D values. Queries with higher E values require more developer "effort" that queries with lower E values. High E-valued queries suggest that a) the query is verbose, b) the query is difficult to author, or c) some combination of the two.

The workload chosen for our analysis was a subset of the LDBC interactive workload [40], which describes a set of operational queries about a social network. For the LDBC interactive queries IS-1, IS-2, IS-3, IC-1, IC-2, and IC-3, we compare our gSQL^{++} implementation against the same query authored in a) SPARQL, b) Cypher, c) SQL (in PostgreSQL's dialect), and d) TigerGraph GSQL. All query implementations analyzed (aside from those in gSQL^{++}) were taken from the LDBC's reference implementation repository. Our measurements for each authored query can be found at https://github.com/graphix-asterixdb/benchmark.

Figure 6 illustrates the distribution of 1) normalized volume V, 2) normalized difficulty D, and 3) normalized effort E for all studied query implementations. On average, the queries that required the most effort to author were in GSQL. This result is not surprising, given that GSQL "queries" are more akin to procedural "programs". Queries authored in SPARQL had the 2nd highest E value on average. We attribute SPARQL's high average E value to the target data model of SPARQL: RDF. Property graphs, the target model of Graphix and the

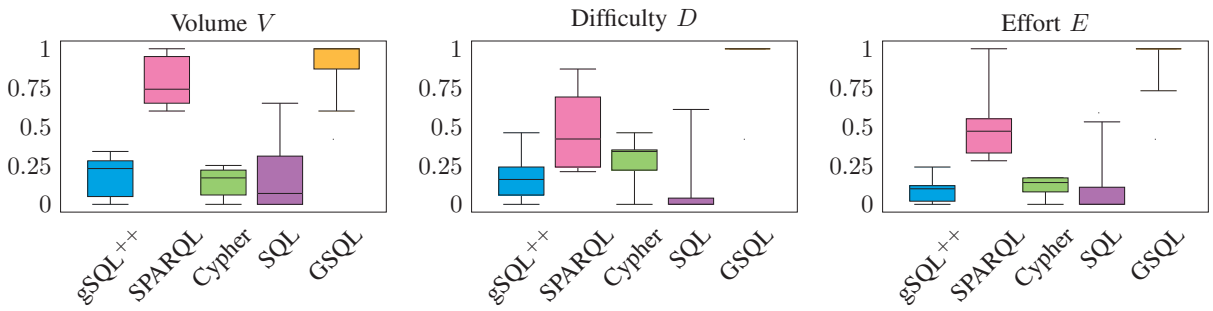


Fig. 6: Distribution of normalized Halstead metrics (volume V, difficulty D, effort E) for several implementations of queries from the LDBC interactive social network benchmark.

LDBC social network benchmark, define vertices, edges, *and* properties. In contrast, the RDF model only defines vertices and edges. A "property" in RDF is modeled using a vertex and an edge to the containing entity. Consequently, we observed high verbosity (high V values) when compared to queries authored in gSQL^{++}, Cypher, and SQL.

On average, SQL queries required the least amount of effort to author – however Figure 6 shows a wide distribution of E values. SQL queries for the LDBC interactive workload were either incredibly easy to author, or incredibly difficult / verbose to author. Queries that required paths (`IC-1` and `IC-3`) yielded large V and D values when implemented in SQL. Queries implemented in Cypher had the tightest distribution of E values. Given that the LDBC benchmark is tailored towards graph-based analysis, it should come as no surprise that the effort required to write graph queries in Cypher is more "predictable" than writing graph queries in SQL. Figure 6 show that queries authored in gSQL^{++} are able to leverage the simplicity of SQL (or more accurately, SQL^{++}) and the graph constructs from Cypher to ensure low and consistent query-authoring effort.

B. Preliminary Performance Analysis

1) Experimental Setup: For our performance experiments, we used AWS EC2 `i2.2xlarge` instances, each with (i) 32 GB of memory, (ii) 8 vCPUs, and (iii) EBS `gp3` SSDs. Our evaluation compares a Neo4j instance (version 5.4.0) on a single AWS node against Graphix clusters of various sizes.[3] The workloads chosen for the experiments in this section were a) the LDBC business intelligence social network benchmark [41], which is a "graph-based parallel" to the TPC-H benchmark for relational analytics, and b) the LDBC interactive workload [40], the operational counterpart to the business intelligence benchmark. With respect to the structure of the social network graph, LDBC's data generator produces networks that adhere to the Homophily principle (i.e. persons with similar interests and behavior know each other) and with vertex degrees similar to Facebook. A scale-factor of `SF=100` (raw data size $\simeq$100 GB, 312.0 million vertices, 1.1 billion edges) was used to evaluate the archetypal out-of-core scenario, where a single machine cannot operate on the graph entirely in memory. All artifacts used for the experiments in this paper can be found at: https://github.com/graphix-asterixdb/benchmark.

2) Experimental Results: Figure 7 shows eight plots that compare Neo4j on one node (green) against Graphix clusters (blue) of increasing size where appropriate. For brevity, only eight queries are discussed here: two from the interactive-short workload, three from the interactive-complex workload, and three from the business intelligence workload. A more comprehensive set of results (i.e., nearly all LDBC social network benchmark queries) can be found in [35].

The first two plots of Figure 7 compare Neo4j and Graphix on a single node with queries `IS-1` and `IS-2` from the (short) interactive workload. Both `IS-1` and `IS-2` anchor on some starting vertex to subsequently traverse a small portion of the graph. `IS-1` requires a single edge hop from an anchor vertex, while `IS-2` requires a) traversing all one-hop neighbors from the anchor vertex, and b) recursively navigating to a source vertex from the previously traversed-to neighbors. As shown in Figure 7, Graphix on a single node is able to perform roughly on-par with Neo4j for `IS-1` while outperforming Neo4j for `IS-2`.

The next three plots of Figure 7 compare Neo4j and Graphix with queries `IC-1`, `IC-2`, and `IC-8` from the (complex) interactive workload. These queries anchor on some starting vertex and traverse a larger portion of the graph to some set of destination vertices. As shown in Figure 7, Neo4j outperforms Graphix for `IC-1` at $n = 1$, $n = 2$, and $n = 4$. Neo4j employs a bidirectional BFS to evaluate `IC-1`, whereas Graphix performs a BFS from the anchor vertex. Neo4j, on average, will traverse fewer edges than Graphix for `IC-1` due to Neo4j's bidirectional BFS strategy. If a Graphix user wants comparable performance to Neo4j for `IC-1`, they would have to increase their Graphix cluster size to $n = 8$ at a minimum. If a Graphix user wants *better* performance than Neo4j for `IC-1`, they would then increase their Graphix cluster to more than $n = 8$ nodes. A Neo4j user *might* be able to achieve faster execution times

[3]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. No other distributed database has implementations for the LDBC social network interactive and business intelligence benchmarks.

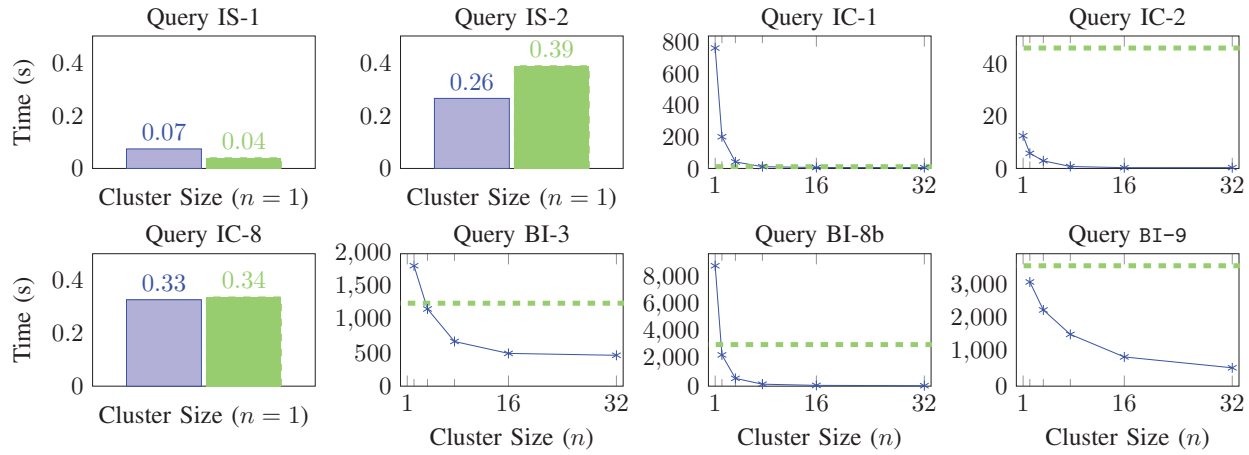


Fig. 7: Several plots showing a Graphix cluster of varying size (in blue) against a Neo4j instance (in green) for a variety of queries from the LDBC social network benchmark.

with multiple nodes, but (as discussed earlier) users must also rewrite their existing queries to accommodate their cluster scale-out.

With respect to queries `IC-2` and `IC-8`, Graphix at $n = 1$ either outperforms Neo4j (`IC-2`) or performs on par with Neo4j (`IC-8`). For query `IC-2`, Graphix experiences near-linear speedup with increasing values of n up until $n = 32$. In contrast to `IC-1`, Neo4j and Graphix traverse the same number of edges for both `IC-2` and `IC-8`. Both Graphix and Neo4j evaluate edges for queries `IC-2` and `IC-8` 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 not perform such a sort, resulting in more random I/O for high degree vertices (like those in `IC-2`).

The last three queries in Figure 7 compare Neo4j and Graphix clusters of varying size with queries `BI-3`, `BI-8b`, and `BI-9` from the business intelligence workload. In contrast to queries from the interactive workload, the business intelligence queries do not anchor on a starting vertex. Queries `BI-3`, `BI-8b`, and `BI-9` require the traversal of several edges from a set of starting vertices plus some form of aggregation. As a general trend across all business intelligence queries here, Graphix is able to execute such queries faster with larger values of n. Neo4j outperforms Graphix at $n = 1$ and $n = 2$ for query `BI-3` and at $n = 1$ for query `BI-8b`. Graphix for all other values of n, however, outperforms Neo4j. Neo4j traverses edges with an index-nested-loop `JOIN` approach, which is not as performant for high out-degree vertices (i.e., `JOIN`s that are not selective [42]). Graphix, on the other hand, is able to traverse edges with `JOIN` algorithms that are better equipped to handle high out-degree vertices (e.g., hybrid hash `JOIN`s). Furthermore, the `JOIN` algorithms used by Graphix operate in parallel on different partitions of data. Graphix has the capability to utilize more compute and disk I/O to further accelerate the traversal of multiple edges.

VIII. Conclusion

In this paper we have introduced Graphix, an Apache AsterixDB extension that takes a view-based approach to perform ad-hoc, (shard agnostic) partitioned-parallel, and synergistic graph + document analytics on JSON data in-situ. In contrast, current solutions fall short on either i) the "in-situ" aspects (e.g. native graph databases), ii) the (shard agnostic) "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 paper has detailed the user model of Graphix: a) the graph view user model, b) the `CREATE GRAPH` DDL, and c) how SQL^{++} with a navigational pattern matching extension can support a rich set of graph queries. Graphix leverages the AsterixDB stack (Algebricks and Hyracks) to execute these graph queries on a cluster of workers. As shown by our preliminary evaluation, Graphix allows users to express queries with low effort that will ultimately leverage horizontal scaling to handle both analytical and operational workloads. Note that although this work was done within the context of AsterixDB, many of the concepts explored here could be applied to other systems with data-parallel execution engines. We invite readers to try Graphix at https://graphix.ics.uci.edu.

Acknowledgment

We would like to acknowledge several individuals from UCSD: Yannis Papakonstantinou for his review and comments on early versions of this paper, and Amarnath Gupta and Subhasis Dasgupta for their input on the query model and experimental setup sections. We would also like to thank UCI student Sushrut Borkar for his help writing the queries used in our evaluation. This research was supported in part by NSF awards IIS-1838248, IIS-1954962, and CNS-1925610, by the HPI Research Center in Machine Learning and Data Science at UC Irvine, and by the Donald Bren Foundation (via a Bren Chair).

REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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*, vol. 8, pp. 161–172, 2014.

[8] D. Yan, Y. Bu, Y. Tian, and A. Deshpande, "Big Graph Analytics Platforms," *Foundations and Trends in Databases*, vol. 7, no. 1–2, p. 1–195, jan 2017. [Online]. Available: https://doi.org/10.1561/1900000056

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

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

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

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

[13] 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.

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

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

[16] 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*, 2015, pp. 1887–1901.

[17] H. Jiewen, D. Abadi, and K. Ren, "Scalable SPARQL querying of large RDF graphs," *PVLDB*, vol. 4, pp. 1123–1134, 08 2011.

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

[19] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati, "Linking data to ontologies," *J. Data Semant.*, vol. 10, pp. 133–173, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:1325494

[20] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev, "Ontology-based data access: a survey," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18. AAAI Press, 2018, p. 5511–5519.

[21] E. Botoeva, D. Calvanese, B. Cogrel, M. Rezk, and G. Xiao, "OBDA beyond relational DBs: A study for MongoDB," *Description Logics*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:1418349

[22] 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.

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

[24] 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*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 345–359. [Online]. Available: https://doi.org/10.1145/3318464.3386138

[25] 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*, vol. 7, pp. 1905–1916, 2014.

[26] 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*, ser. SoCC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 422–433. [Online]. Available: https://doi.org/10.1145/2806777.2806941

[27] 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*, ser. ICDE '11. USA: IEEE Computer Society, 2011, p. 1151–1162. [Online]. Available: https://doi.org/10.1109/ICDE.2011.5767921

[28] 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*, vol. abs/1405.3631, 2014. [Online]. Available: http://arxiv.org/abs/1405.3631

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

[30] ISO/IEC, "Graph Query Language GQL Standard," Available at https://www.gqlstandards.org.

[31] 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 *26th International Conference on Database Theory (ICDT 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), F. Geerts and B. Vandevoort, Eds., vol. 255. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 1:1–1:22. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/17743

[32] 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.

[33] 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.

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

[35] G. Galvizo, "Graphix: View the (JSON) World Through Graph-Tinted Glasses," PhD Thesis, University of California, Irvine, Irvine, CA, December 2023.

[36] G. Galvizo and M. J. Carey, "Hyracks Unchained: Realizing Semi-Synchronous Recursion in Apache AsterixDB," In preparation.

[37] A. Vashistha, "Measuring query complexity in SQLShare workload," 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:53451506

[38] Á. Goretity and I. Reguly, "Query complexity in modern database DSLs," *ACM Transactions on Information Systems*, vol. 1, no. 1, 2021.

[39] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[40] 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*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 619–630. [Online]. Available: https://doi.org/10.1145/2723372.2742786

[41] 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*, vol. 16, no. 4, pp. 877–890, 2022.
[42] G. Graefe, "Modern B-Tree Techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, p. 203–402, Apr. 2011. [Online]. Available: https://doi.org/10.1561/1900000028