# GRainDB: A Relational-core Graph-Relational DBMS

Guodong Jin
jinguodong@ruc.edu.cn
Renmin University of China
China

Nafisa Anzum
nanzum@uwaterloo.ca
University of Waterloo
Canada

Semih Salihoglu
semih.salihoglu@uwaterloo.ca
University of Waterloo
Canada

## ABSTRACT

Ever since the birth of our field, RDBMSs and several classes of graph database management systems (GDBMSs) have existed side by side, providing a set of complementary features in data models, query languages, and visualization capabilities these data models provide. As a result, RDBMSs and GDBMSs appeal to different users for developing different sets of applications and there is immense value in extending RDBMSs to provide some capabilities of GDBMSs. We demonstrate *GRainDB*, a new system that extends the DuckDB RDBMS to provide graph modeling, querying, and visualization capabilities. In addition, GRainDB modifies the internals of DuckDB to provide a set of fast join capabilities, such as predefined pointer-based joins that use system-level record IDs (RID) and adjacency list-like RID indices, to make DuckDB more efficient on graph workloads.

## 1 INTRODUCTION

Ever since the birth of database management systems, relations and graphs have been the core data structures to model application data in two broad classes of DBMSs: RDBMSs and those referred to as *graph database management systems* (GDBMSs). Historically, the term GDBMS has been used to refer to several classes of DBMSs that all adopt a graph-based model, such as the CODASYL model, RDF, and most recently the property graph model, which is adopted by contemporary systems such as Neo4j [5], TigerGraph [7], GraphflowDB [20], or AvantGraph [1].

RDBMSs and GDBMSs have complementary capabilities that may appeal to users. For example, unlike graphs, the relational model is not restricted to binary relationships and easily models n-ary relationships between entities. At the same time, the relational model requires strict schematization of the data, while graph models are often semi-structured and provide more flexibility when storing sparse data. Similarly, while SQL is very suitable to express standard data analytics tasks, it is arguably not as suitable when expressing queries with recursive joins, for which the languages of GDBMSs contain specialized syntaxes. In terms of performance, RDBMSs integrate numerous techniques, such as columnar storage and vectorized processing, yet, still have shortcomings on *graph workloads* that contain many recursive and many-to-many joins, for which GDBMSs employ specialized techniques.

RDBMSs have seen wider adoption than GDBMSs in practice and emerged as the de-facto systems to store, manage, and query application data. However, given the complementary capabilities of GDBMSs over RDBMSs, there are economical and technical advantages for extending RDBMSs to natively provide some of the capabilities of GDBMSs and support efficient graph querying. Over the past two years, we have started to develop a *relational-core hybrid graph-relational system* that we call *GRainDB* at the University of Waterloo. We use the term relational-core to indicate that GRainDB extends an RDBMS at its core. Specifically, GRainDB integrates a set of storage and query processing techniques, such as predefined pointer-based joins (reviewed in Section 4.1), into the columnar DuckDB RDBMS [2, 24] to make it more efficient on graph workloads. In addition, GRainDB extends DuckDB to natively support a set of new features:

- *Hybrid graph-relational data modeling:* Users can model parts of their database also as a graph. Specifically, users can model relations as nodes and joins between these relations as edges.
- *Hybrid graph-relational querying:* Users can query tables and graphs seamlessly using an extended SQL that we call *GRQL*. GRQL has drawn from TigerGraph's GSQL [13] and Oracle's PGQL languages [28]. The FROM clause of GRQL can contain path patterns in addition to tables to express joins. Path patterns are described with a node and arrow syntax and can contain special syntax for recursive joins, e.g., the Kleene star.
- *Graph visualization:* We have built a browser frontend, similar to the frontends of GDBMSs, such as Neo4j's Bloom [6], that support node-link visualization and interactive exploration of the part of the database that is modeled as a graph.

Our work is similar in spirit to graph systems built on top of some commercial RDBMSs, such as IBM Db2 [27] and SAP Hana [25]. For example, the Db2 Graph project [27] implements the Gremlin graph query language [4] on top of IBM Db2. Unlike GRainDB, these layered systems support only querying graphs and do not modify the core query processors of the underlying RDBMS.

In this paper, we describe our vision, progress, and ongoing work on GRainDB, and a demonstration of developing a COVID-19 contact tracing application on GRainDB. Our application aims to demonstrate: (i) the advantages of modeling and querying data seamlessly both as tables and graphs and the ability to visualize parts of the database as a graph; and (ii) the storage and processing techniques we have integrated into DuckDB to make it more efficient on graph workloads. The source code of GrainDB can be found here [3].

## 2 DEMONSTRATION DATABASE

We begin by describing a relational database that we use as our running example. We consider a COVID-19 contact tracing application in the city of Waterloo, Ontario in Canada. Figure 1 shows the schema of the database used by the application:

- `Person` contains data about people who took COVID-19 tests.
- `Contact` contains the close contact relationship between two people in the `Person` table, one of which has tested positive.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

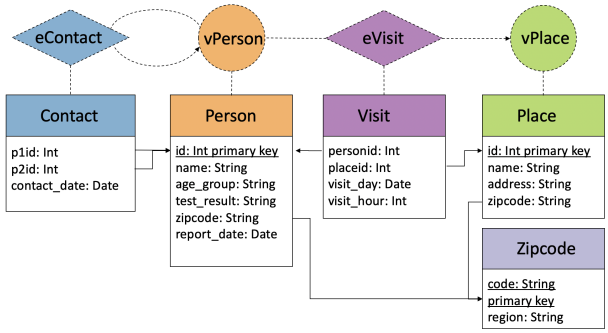



**Figure 1: Example relational database schema. Top row is the graph schema modeling part of the database also as a graph.**

- `Place` contains information about places, such as universities and restaurants, that record their visitors for contact tracing.
- `Visit` stores the records, collected by the places in the `Place` table, of who visited each place and when.
- `Zipcode` contains the set of possible zipcodes in Waterloo and is used to normalize addresses in the `Person` and `Place` tables.

Throughout the paper, we discuss modeling a part of this database also as a graph as shown in the top row of Figure 1. We model the `Person` and `Place` tables as nodes with labels `vPerson` and `vPlace`, respectively. We then model the join between a person record *pe* and a place record *pl* that *pe* visited (over the `Visit` table) as `eVisit` edges and the join between two people who contacted each other (over the `Contact` table) as `eContact` edges.

## 3 THE CASE FOR GRAINDB

We next motivate GRainDB by discussing the complementary features of RDBMSs and GDBMSs in terms of their data models, query languages, and visualization capabilities that their data models provide. We then discuss several performance features we plan to integrate into DuckDB to make it more efficient on graph workloads, which completes the description of our vision for GRainDB.

**Data Model:** Although both relations and graphs can model many application data, we identify two advantages of relations:

- *N-ary relationships:* Graphs are restricted to represent binary relations between two nodes, while relations can represent n-ary relationships for arbitrary values of *n*.
- *Normalized data:* Relations are often arguably the natural structures to model normalized data. For example, the `Zipcode` table in our example normalizes the zipcodes in `Person` and `Place` tables and may not be naturally thought of as nodes or edges.

Similarly, we identify two advantages of graph-based models:

- *Closeness to entity-relationship model:* As discussed in a user survey conducted in our group [26], some users of GDBMSs find nodes and edges closer to their mental model of entities and relationships in their data than tables.
- *Semi-structured data:* Graph-based models are often semi-structured, allowing arbitrary properties to be stored on nodes and edges without a strict schema definition, which may have advantages in applications that frequently integrate data from new sources.

**Query Languages:** SQL has emerged as the de facto language to query and manipulate data in practice. While SQL is very suitable to express many standard data preparation and analytics tasks, languages of GDBMSs contain syntax that can more easily express recursive joins. Consider asking a variable-length join query asking for people that Mahinda may have contacted through 1 to 4 degrees of contacts. Consider the the alternative graph model of our example database. For example, the Cypher language of Neo4j contains a special syntax for such recursive queries, which can be simpler than writing those queries in SQL:

```
MATCH (a:vPerson)-[:eContacted*1..4]->(b:vPerson)
WHERE a.name = Mahinda
```

Similarly, languages of GDBMSs also contain specialized syntax for expressing some queries that are unions of multiple join queries, such as queries with unlabeled nodes and edges.

**Visualization:** The output tables of queries in RDBMSs can be used to produce many useful visualizations, such as bar charts or line charts. However, these visualizations are not suitable when one wants to analyze the sequences of connections and paths that connect the entities in a database. Default frontends of GDBMSs support node-link views that are more suitable for this purpose.

Our first goal in GRainDB is to extend an RDBMS to complement it with the above features of GDBMSs. One can in principle provide these features only by building layers on top of the underlying RDBMS without modifying the internals of the system. This approach however would be sub-optimal in performance. Our second goal in GRainDB is to modify the underlying RDBMS to make it more efficient on graph workloads. This is a colloquial term to refer to workloads that contain many equality joins over many-to-many relationships that can frequently be cyclic and recursive.

**Techniques for Fast Join Capabilities:** We aim to integrate three techniques into the underlying RDBMS, the first of which is motivated by how existing GDBMSs often perform joins. The latter two are relatively recent techniques developed in the context of RDBMSs for many-to-many joins but not yet seen wide adoption.

- *Predefined pointer-based joins:* Joins in GDBMSs are pointer-based in nature and predefined to the system as edges[1]. Node records are joined with their neighbors using system-level integer IDs, which serve as pointers to look up matching neighbor IDs in an adjacency list index. This contrasts with value-based nature of joins in RDBMSs, where tables are often joined with each other using the values inside columns. Value-based joins are naturally more general as users can join arbitrary tables. Yet they can be less efficient because they typically do not use an adjacency list-like join index and may be on non-integer types, e.g., strings.
- *Factorization:* Factorization [22, 23] is a technique to avoid data replication in results of queries that contain joins over many-to-many relationships. As a simple example, consider joining a person $p_1$ with its $k$ contacts $p_{1,1}$, ..., $p_{1,k}$, which would result in $k$ tuples $(p_1, p_{1,1})$, ..., $(p_1, p_{1,k})$ that replicate the value $p_1$ $k$ times. Such replication becomes severe when queries contain further many-to-many joins. Factorization avoids such replication by

[1]The term predefined was used by Ted Codd in his Turing Lecture to describe the joins in GDBMSs of 1960s and 1970s that were based on the CODASYL model [12]

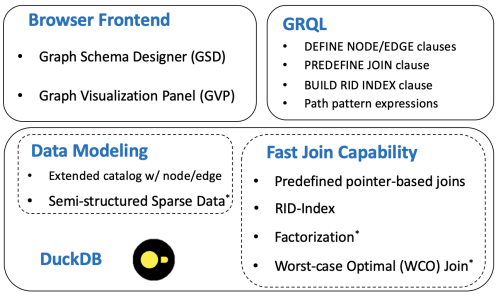



**Figure 2: GRainDB Overview. Features marked as $*$ are our ongoing and future works.**

representing results as unions of Cartesian products, e.g., as $(p1 \times \{p_{1,1}, ..., p_{1,k}\})$.

- *Worst-case Optimal (wco) Join Algorithms:* While factorization is geared toward queries with acyclic joins, the recent worst-case optimal join algorithms are geared for cyclic join queries over many-to-many joins. The seminal work of Atserias et al. [10] observed that traditional binary join plans are suboptimal on cyclic join queries, which was corrected by the recent wco join algorithms [21, 29]. Unlike binary join plans that perform joins pairs of tables-at-a-time, wco join algorithms join multiple tables one column-at-a-time.

## 4 GRAINDB OVERVIEW

Figure 2 shows the overview of GRainDB. We are developing GRainDB as an extension of DuckDB, which is a new open-source columnar read-optimized RDBMS that implements several modern query processing techniques, such as columnar storage [8], vectorized-oriented query processing [8], and morsel-driven parallelism [18]. We opted for an analytical read-optimized RDBMS as one of our goals is to efficiently support graph workloads, which are read heavy. GRainDB extends DuckDB both internally and through several new components, which we next review, highlighting our prior as well as ongoing work.

### 4.1 Prior Work: Predefined Pointer-based Joins Using Sideways Information Passing (SIP)

Our first work on GRainDB [16] focused on extending the internals of DuckDB to support predefined pointer-based joins to improve the systems' performance on many-to-many joins. We briefly review our technique here and refer readers to reference [16] for the details. We integrated predefined joins into DuckDB by extending two components of the system: (i) the physical storage and query processor; and (ii) the indexing sub-system.

• **Physical Storage and Query Processor:** Users first predefine a primary-foreign key join from table $F$ to table $P$, where a column of $F$ has a foreign key to a column of $P$, using a `PREDEFINE JOIN` clause we added to DuckDB's SQL dialect. This performs an ALTER TABLE command that inserts an additional $RID_p$ column to $F$ that contains for each row $r_f$ in $F$ the row ID (RID) of row $r_p$ in $P$ that $r_f$ points to. RIDs are dense integer-based system-level IDs in columnar RDBMS that are used to identify the physical locations of the column values of each row. They are therefore system-level pointers, similar to node IDs in GDBMSs.

RID columns are used during query processing as follows. If a query contains a predefined join, say between `Person.id = Contact.p1id`, we replace some of the hash join and scan operators in DuckDB's default plan as follows. We replace the hash join operator evaluating `Person.id = Contact.p1id` predicate with a modified hash join operator that we call `SIPJoin`, for **s**ideways **i**nformation **p**assing join. We also replace the scan operators for `Person` and `Contact` tables with modified scan operators (explained momentarily), that scan RID values into tuples. Given the scanned RID values, `SIPJoin` performs the join on integer RIDs instead of the original `id` and `p1id` columns. In addition, `SIPJoin` can pass matching RIDs from its build side in a bitmask that can be used by the scan operator (called `ScanSJ`, for scan **s**emi**j**oin) on its probe side to perform a semijoin when scanning tuples. This ensures that only the tuples that are guaranteed to successfully join are scanned, which can decrease the scans and probes significantly.

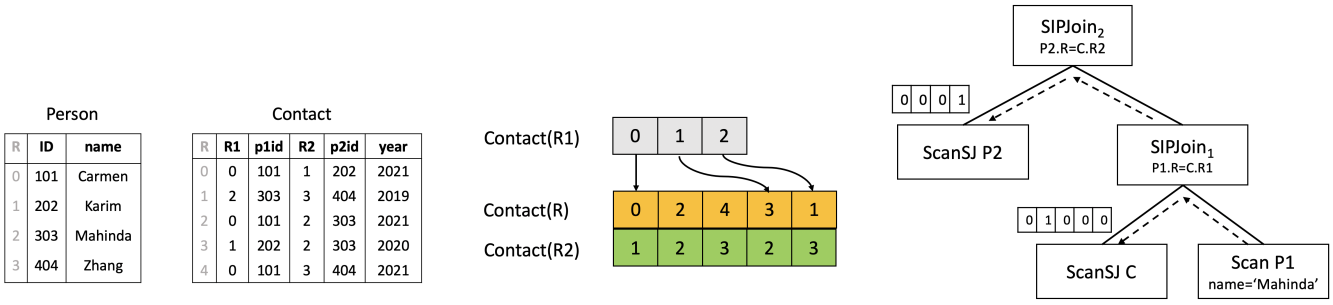
• **Indexing Sub-system:** A common way to represent many-to-many relationships between two sets of entities in relational databases is to have a table $C$ that contains two foreign keys on two other (not necessarily different) tables $P_1$ and $P_2$, as in the `Contact` table in our running example. If the join of $C$ with both $P1$ and $P2$ have been predefined to the system, users can additionally build a *RID index* on $C$ on the two extended RID columns $RID_{p1}$ and $RID_{p2}$ (using a BUILD RID INDEX command). This index is analogous to adjacency list indexes in GDBMSs and is stored in an adjacency list format. RID indexes can be used by `SIPJoin` to generate further information to pass when a query joins $P_1$ or $P_2$ with $C$.

Example 1. *Figure 3 shows example instances of the* `Person` *and* `Contact` *tables when two joins are predefined:* `Person.id = Contact.p1id` *and* `Person.id = Contact.p2id`*. This results in an extended* `Contact` *table that contains two RID columns,* `R1` *and* `R2`*, that contain the RIDs of the person tuples that match* `p1id` *and* `p2id` *columns, respectively. For example the second tuple $t_2$ in* `Contact` *has* `R1` *value 2 because $t_2$.*`p1id` *references 303 (Mahinda's ID), which has an (implicit) RID value 2 in the* `Person` *table. Now consider the following SQL query:*

```
SELECT P2.name FROM Person P1, Contact C, Person P2
WHERE P1.id = C.p1id AND P2.id = C.p2id
      AND P1.name = 'Mahinda'
```

*This query asks for the* `name` *of the first-degree contacts of Mahinda. Figure 3c is an example plan in GRainDB for this query using* `SIPJoin` *and* `ScanSJ` *operators. For example, the bottom* `SIPJoin` *receives the RID of the tuple $t_2$ from* `Person` *where t2.R1 is 2 (corresponding to Mahinda's RID). Using the RID index, the RID of the only matching tuple from* `Contact` *is generated (with RID 1) and passed to the probe side* `ScanSJ` *as a bitmask.* `ScanSJ` *then performs a semijoin and scans only the tuples whose RIDs are in the bitmask.*

Our prior work [16] presents extensive experiments demonstrating the performance benefits of enhancing DuckDB with predefined joins on both graph and relational workloads with many-to-many joins, e.g., the LDBC social network benchmark [9]. Predefined




**(a) Example tables.** **(b) Example RID index.** **(c) Example join plan.**

**Figure 3: (a) Example instances of `Person` and `Contact` tables. The `Contact` table contains the two RID columns, `R1` and `R2` which are materialized after two joins to the `Person` table predefined on `p1id` and `p2id` columns, respectively. (b) The RID index on `Contact`. (c) An example join plan for the query in Example 1 that uses `SIPJoin` and `ScanSJ` operators.**

joins is purely a relational query processing optimization that is independent of GRainDB's graph modeling and querying capabilities. That said, as we next discuss, when users model parts of their database as a graph, we automatically predefine the joins implied by the modeled edges and build the necessary RID indexes in DuckDB. For example, in our running example, of which tables are shown in Figure 3a, users map the table `Person` to a node label `vPerson`, and define an edge label `eContact` from `vPerson` to `vPerson`. The edge label is mapped from the table `Contact` through two equality joins between `Person.ID` and `Contact.p1id`, and between `Person.ID` and `Contact.p2id`, respectively. Automatically, GRainDB predefines these two joins, which adds two RID columns `R1` and `R2` to `Contact` table for `p1id` and `p2id`, respectively. Additionally, GRainDB creates RID indexes on these two extended RID columns (`R1` and `R2`).

After the modeling, given a query, we perform a rule-based query optimization based on DuckDB's query plan. Specifically, we recursively traverse DuckDB's default logical plan and find each join operator that evaluates a predefined join. Upon finding these joins, we replace them with our SIPJoin operator, and also replace corresponding probe-side Scan operators, to which bitmasks are passed, with ScanSJ.

## 4.2 Graph Modeling, Querying, and Visualization

As part of this demonstration paper, we have extended GRainDB with two components to provide graph modeling, querying, and visualization capabilities: (i) GRQL; and (ii) a browser frontend.

*4.2.1 GRQL.* We added further extensions to SQL to define: (i) nodes and edges; and (ii) path patterns that can seamlessly exist in the FROM clause with tables to describe joins over the defined nodes and edges. We refer to this extended SQL as GRQL.

**Node and edge label definitions:** We added two commands: `DEFINE NODE LABEL` and `DEFINE EDGE LABEL`, that form our relation-to-graph transformation language. The former simply maps any relation in the database to a node label, e.g., the `Person` table to a `vPerson` node label. The latter defines an edge between any two (not necessarily distinct) node labels `V` and `W`, say mapping relations $R_V$ and $R_W$, respectively, in two possible ways: (i) as a direct primary key-foreign key join between $R_V$ and $R_W$; or (ii) as two primary key foreign key joins, one from $R_V$ to a "relationship" table `E` and the other from $R_W$ to `E`. For example, the following command defines an `eContact` edge label from `vPerson` to `vPerson` nodes through two joins to the `Contact` table:

```
DEFINE EDGE LABEL eContact FROM vPerson V TO vPerson W
ON V.id = Contact.p1id AND W.id = Contact.p2id
```

Edge definition commands predefine the joins in the edge definition to DuckDB and build the necessary RID indexes. We store the defined vertices and edges in DuckDB's catalog and use it when compiling path patterns to query plans, as we next explain.

**Path patterns:** Path patterns visually describe joins between the tables that were defined in node and edge definitions using node and arrow syntax. Our syntax draws directly from existing languages of GDBMSs, such as Cypher, GSQL, or PGQL. Path patterns can include syntax to describe fixed-length joins, such as `(a:vPerson)-[:eContact]->(b:vPerson)`, or two types of recursive joins: (i) variable-length joins that provide a minimum and maximum lengths on the joins/paths; or (ii) transitive closure of a join using the Kleene star syntax. For example, in the pattern `(a:vPerson)-[:eContact*2..5]->(b:vPerson)`, variable $b$ denotes 2 to 5 degree contacts of nodes matching variable $a$.

Path patterns can appear in the FROM clause to form queries that seamlessly query graphs and relations.

Example 2. *Consider a query that asks for the top 50 highest risk people $b$ who have the most pathways to someone $a$ infected with the virus, where $a$ lives in one of the zipcodes in high risk regions of Waterloo. We define a pathway as a 1 to 3-length contact path from $a$ to $b$. This query can be expressed in GRQL as follows:*

```
SELECT b.name, count(*) as numPathways
FROM (a:vPerson)-[:eContact*1..3]->(b:vPerson), zipcode
WHERE a.test_result='+'  AND  a.zipcode=zipcode.code AND
      zipcode.region IN HighRiskZipcodes
ORDER BY numPathways DESC LIMIT 50
```

*The omitted* `HighRiskZipcodes` *is an inner (pure) SQL query that computes the highest risk zipcodes based on the number of infections*

*in that zipcode. The query seamlessly joins a path pattern with the* `Zipcode` *table.*

To compile queries with path patterns, we modified DuckDB's parser to generate an ast that recognizes path patterns. In addition, we modified DuckDB's planner, which takes the output ast of the parser and generates an initial logical plan, to rewrite fixed-length paths as default join trees and variable-length paths as recursive common table expressions (CTEs). We did not modify the rest of the pipeline: as before DuckDB generates a default optimized plan from the initial plan and if any part of the plan can benefit from predefined joins, we replace necessary hash join and scans (possibly inside the subplans evaluating recursive CTEs) with `SIPJoin` and `ScanSJ` operators.

*4.2.2 GRainDB Browser Frontend.* We extended GRainDB with a browser frontend, which contains two visual components: (1) an interactive Graph Schema Designer (GSD); and (2) an interactive Graph Visualization Panel (GVP), shown in Figure 4. GSD allows users to model their graphs visually and interactively instead of typing our node and edge definition commands. On the left panel of GSD users have access to the schema of their relational database. Users drag and drop any table's schema to define a node label. Users then draw edges between the defined node labels to define edges. The joins defining the edges are described by selecting join columns and the edge table (if any) in drop down menus.

GVP is a frontend to ask queries in GRQL to GRainDB and visualize the results both as tables and as graphs when possible. Specifically, if projections in the SELECT clause contain variables that were used to describe nodes and edges in path patterns, GVP can visualize these outputs in an interactive node link diagram, as shown in Figure 4b. Users can click on nodes to expand the nodes to their neighborhoods, which are obtained by further GRQL queries to GRainDB.

## 4.3 Ongoing and Future Work

Our implementation of predefined joins inside DuckDB and GRQL and our browser frontend partially fulfills our goal of developing an extended RDBMS that is highly performant on graph workloads and one that provides a set of graph modeling, querying, and visualization capabilities to users. We next describe our ongoing and future work.

**Factorization:** In ongoing work we are modifying DuckDB's query processor to process queries in a factorized format instead of blocks of flat tuples. Several works have proposed approaches to integrate factorization into query processors both in the context of RDBMSs and GDBMSs. The FDB system [11] describes a relational query processor that consists of operators that take as input tries and output tries. This architecture materializes full outputs and is in conflict with DuckDB's vectorized and pipelined architecture. In a recent work done in our group, we described an implementation of factorized query processor, called *list-based processor* in the context of the GraphflowDB GDBMS [15], that is also developed in our group. Although this is a pipelined processor that uses operators similar to DuckDB's, it is designed assuming an in-memory system and implements a limited form of factorization where only the last edges of paths can be factorized. We are investigating ways to develop a query processor that is as general as FDB but uses DuckDB's vectorized and pipelined processor.

**WCO Joins:** Several works have implemented wco join algorithms in the context of both RDBMSs and GDBMSs. In the context of GraphflowDB, we have proposed techniques that assume the existence of presorted indices and and use fast merge-sort like join operators [17, 19, 20]. This has a performance advantage during query evaluation but slows down updates. Reference [14] instead integrates wco joins in the Umbra system using hash-indexes that are built on the fly. As future work, we plan to integrate WCO joins into DuckDB's predefined joins using an approach that sorts RID indices on the fly and uses merge-sort like operators and study the tradeoffs of this design compared to prior approaches.

**Semi-structured Sparse Data** Another challenge is how to integrate support for semi-structured attributes in DuckDB. A simple but not performant approach would be to add a default column to each table that stores a blob that encodes unstructured properties and parse this blob during query processing. Whether this capability can be provided more efficiently and without making the system very complex is an interesting future direction.

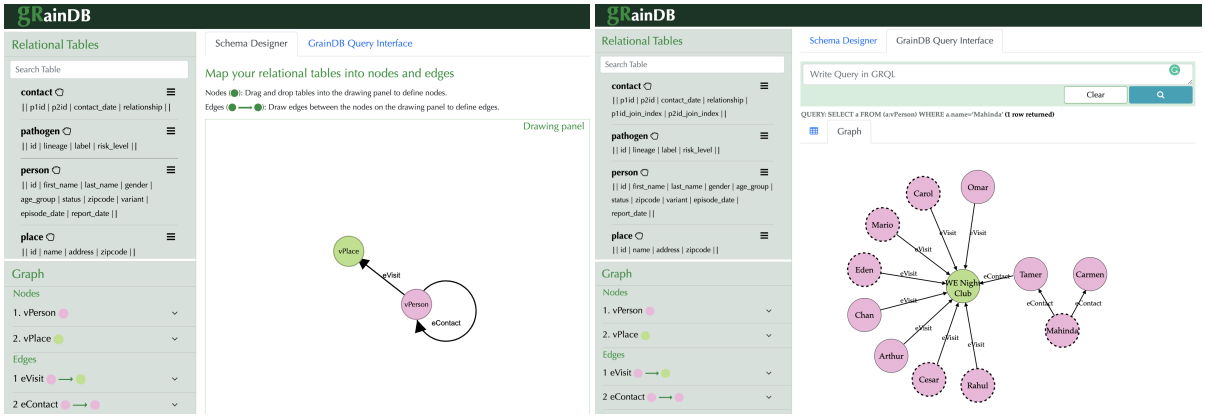
# 5 DEMONSTRATION SCENARIOS

Our demonstration illustrates three aspects of GRainDB: (i) the ease of development of an application using GRainDB's hybrid graph relational modeling capability and GRQL language; (ii) the benefits of enhancing an RDBMS with graph visualization capabilities; and (iii) GRainDB's query plans that perform our SIP- and semijoin-based evaluation of predefined joins.

**GRainDB Application Development:** Our first scenario walks attendees through the steps of developing a reporting application for Waterloo Public Health Services (WPHS) using GRainDB. The application runs routine queries at the end of each day to generate daily reports for managing the pandemic at WPHS. The queries generating the reports will include those that are easily written in SQL, as well as those that can benefit from GRQL's path patterns.

Using the Graph Schema Designer, the attendees will first visually model the `Person` and `Place` relations as nodes and then define the `eContact` and `eVisit` edges between them (without going into the details of the predefined joins and created RID indices in the underlying DuckDB). We will then go to the query panel in GRainDB's browser frontend to show attendees the queries written both in pure SQL and GRQL to highlight the differences and ease of using specialized syntax for graph patterns. For example, one report will generate the list of high risk individuals based the number of pathways they may have to infected people. We will show that this can easily be done with the GRQL query in Example 2 that seamlessly joins recursive path patterns with tables and uses selections on inner SQL queries.

**GRainDB Graph Visualization:** Our next scenario demonstrates the benefits of visualizing parts of a relational database in the form of a graph. Our setup will be a contact tracing agent at WPHS who has been given a report on an individual Mahinda who has tested positive but reported no one else during their interview who was positive and has not visited a place recently. The agent was asked to analyze how the virus may have spread to Mahinda. The agent uses the graph visualization capability of GRainDB's GVP. On the frontend we will ask a simple GRQL query to select




**(a) GRainDB Schema Designer (GSD).** **(b) GRainDB Visualization Panel (GVP).**

**Figure 4: GRainDB Browser Frontend.**

Mahinda's record as a node as in Figure 4b. This displays Mahinda's record as a node and immediately gives the attendees a graph view on the database. The attendees will click on Mahinda's node to see their direct contacts, Carmen and Tamer, neither of whom are infected (and no place nodes, since Mahinda has not visited places recently). Attendees will expand Carmen and Tamer to see their neighborhoods. When expanding Tamer's neighborhood, they will notice that Tamer has recently visited a night club, which upon further expansion will reveal many infected visitors. Through this exploratory graph analysis (but performed on top of an RDBMS), the attendees will formulate the hypothesis that Tamer may be an asymptomatic person who has spread the virus to Mahinda.

**GRainDB's Predefined Joins:** The goal of our last scenario is to demonstrate GRainDB's query plans that implement sip-based predefined joins. We first describe attendees how GRainDB predefines the implicit joins defined for `eContact` and `eVisit` edges and builds the RID indices in the underlying DuckDB. The attendees will then issue and profile queries that contain these predefined joins by turning our predefined join optimization on and off through a flag we have in GRainDB. Turning the optimization off returns DuckDB's default plan, i.e., without replacing hash joins and scans with `SIPJoin` and `ScanSJ` operators. Our queries will include fixed-length path queries as well as recursive ones. We will compare the vanilla DuckDB plans and GRainDB plans and show the bitmasks that are passed from `SIPJoin` operators to `ScanSJ` operators.

## 6 ACKNOWLEDGMENTS

Guodong Jin's work is supported by NSFC grant No. U1711261.

## REFERENCES

[1] 2021. AvantGraph. http://avantgraph.io
[2] 2021. DuckDB. https://duckdb.org
[3] 2021. GRainDB. https://github.com/graindb/graindb
[4] 2021. Gremlin Query Langage. https://tinkerpop.apache.org/
[5] 2021. Neo4j. http://neo4j.com
[6] 2021. Neo4j Bloom. https://neo4j.com/product/bloom/
[7] 2021. TigerGraph. http://tigergraph.com
[8] Daniel Abadi, Peter Boncz, Stavros Harizopoulos Amiato, Stratos Idreos, and Samuel Madden. 2013. *The design and implementation of modern column-oriented database systems.* Now Hanover, Mass.
[9] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, et al. 2020. The LDBC social network benchmark. *CoRR* (2020).
[10] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42 (2013).
[11] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A Query Engine for Factorised Relational Databases. *PVLDB* 5, 11 (2012).
[12] Edgar F Codd. 1982. Relational database: a practical foundation for productivity. *Commun. ACM* 25, 2 (1982).
[13] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2018. GSQL 2.0: Seamless Querying of Relational and Graph Databases. (2018).
[14] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB* 13, 12 (2020).
[15] Pranjal Gupta, Amine Mhedhbi, , and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *PVLDB* 14, 11 (2021).
[16] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. *PVLDB* 15 (2022).
[17] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD*.
[18] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.
[19] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2021. A+ Indexes: Lightweight and Highly Flexible Adjacency Lists for Graph Database Management Systems. In *ICDE*.
[20] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-Case Optimal Joins. *TODS* (2021).
[21] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *JACM* 65, 3 (2018).
[22] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45 (2016).
[23] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *TODS* 40, 1 (2015).
[24] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *SIGMOD*.
[25] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The graph story of the SAP HANA database. *BTW*.
[26] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDBJ* 29, 2 (2020).
[27] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *SIGMOD*.
[28] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*.
[29] Todd L Veldhuizen. 2012. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR* (2012).