# Chimera: A system design of dual storage and traversal-join unified query processing for SQL/PGQ

Geonho Lee
ghlee5084@kaist.ac.kr
KAIST
Republic of Korea

Jeongho Park
jhpark@graphai.io
GraphAI
Republic of Korea

Min-Soo Kim*
minsoo.k@kaist.ac.kr
KAIST
Republic of Korea

## ABSTRACT

As graphs are being used increasingly in various industries, a new standard of SQL (called SQL:2023) has incorporated SQL with Property Graph Queries (SQL/PGQ) as a core feature. While some approaches process graph queries within RDBMSs using graph view definition or materialized graph view, their performance is not good enough for interactive SQL/PGQ queries in terms of response time, throughput, and graph size. To address this problem, we propose a novel system design named *Chimera*, which features a dual-store architecture and a unified query planning called Traversal-Join(TJ). This design treats the topologies of a graph as first-class citizens rather than secondary elements overcoming the graph size limitations of the materialized graph view approach. It also generates an efficient, unified query plan that performs traversal and join in a mixed way, significantly enhancing both response time and throughput. Implemented on the open-source RDBMS, PostgreSQL, our extensive experiments with the LDBC SNB benchmark and microbenchmark show that Chimera significantly outperforms the existing approaches and GRDBMSs.

**PVLDB Reference Format:**
Geonho Lee, Jeongho Park, and Min-Soo Kim. Chimera: A system design of dual storage and traversal-join unified query processing for SQL/PGQ. PVLDB, 18(2): 279 - 292, 2024.
doi:10.14778/3705829.3705845

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/Geonho-Lee/ldbc-snb-chimera/.

## 1 INTRODUCTION

As graphs are increasingly utilized across various fields including social network services, finance, e-commerce, web, and biology, the importance of graph DBMSs for storing and efficiently processing graph queries is growing. The recent announcement of SQL:2023 [31] has placed a spotlight on graph query processing within RDBMSs. The key feature of this new standard is processing SQL with Property Graph Queries (SQL/PGQ) [15]. A property graph typically includes properties associated with each vertex and edge, making it more complex than a simple graph. Despite this complexity, property graphs are increasingly utilized due to their high expressive power. In this paper, we use the term *topologies* to refer to the vertices and edges of a graph, distinguishing these structural components from their associated *properties*.

Even before the new standard had been announced, there were already various research efforts focused on extracting property graphs from RDBMSs, managing them within RDBMSs, and processing queries using graph query languages such as Gremlin [4] and Cypher [48] within RDBMSs. The extraction methods [3, 32, 37, 52, 71] usually provide a *graph view definition* interface that allows users to define the relationship, such as FK-PK [32, 37] and join relationship [3, 52, 71], used to extract graphs from tables. The managing methods usually store a graph as relational tables with the schema designed for the graph [10, 20, 50, 62, 64, 68], or as read-optimized in-memory graph data structures that can be used with a graph processing engine [3, 30, 33, 51, 52, 70, 71]. The querying methods usually process a graph query using a pure relational engine by translating it into a SQL query [10, 20, 50, 62, 64, 68], using an external graph processing framework [3, 52], or using its own graph engine [30, 33, 51, 70, 71]. These extensive efforts have culminated in the release of the new SQL/PGQ standard, presenting a significant challenge [22, 65, 70]: *how to evolve an RDBMS into a Graph-Relational DBMS (shortly, GRDBMS) that efficiently supports SQL/PGQ.*

SQL/PGQ consists of a graph view definition language, which is used to define graph views on relational tables, and a graph pattern matching language, which is used to process graph queries on top of the graph views. We emphasize interactive queries, commonly utilized in various graph applications [65]. Examples include neighborhood queries in social networks that provide personalized trending feeds to users [17]; subgraph queries in finance that detect fraud patterns such as cycles and bipartite structures in ongoing transactions [54, 67]; and search queries in e-commerce that identify products matching user preferences using the techniques like random walks, content-based, and collaborative filtering [58, 59]. These interactive queries involve more complex operations compared to traditional relational OLTP workloads but require real-time responses by accessing specific parts of a graph with the latest information [5, 18]. Typically, a graph pattern matching query involves both graph traversal on topologies, referred to as *topology operations*, and filtering on properties, referred to as *property operations* [15, 21]. Although only a few GRDBMSs support SQL/PGQ, they are primarily divided into two approaches: *Graph Table (GT)* [10, 50, 62, 64, 68] and *Materialized Graph View (MGV)* [30, 33, 51, 70].

The GT approach stores both the topologies and properties of a graph in relational tables and additionally stores the graph's

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705845

metadata, namely the graph view definition. This approach does not necessitate changes to the storage and query processing layer of RDBMSs. For query processing, GT translates a graph query into the corresponding SQL query using the metadata. These SQL queries are then executed using standard relational operators such as join, even if the original graph query involves both property and topology operations. This approach allows the translated queries to benefit from the advanced optimization capabilities of relational query optimizers. However, a significant drawback is the necessity for costly joins between vertex and edge tables for each topology operation. Consequently, this approach often experiences performance degradation, particularly for queries that involve topology operations and require rapid response and high throughput [33, 72].

The MGV approach stores a graph in relational tables similar to the GT approach but additionally extracts the topologies from these tables as a read-optimized graph data structure known as a materialized graph view. Thus, it requires a bunch of additional layers to process topology operations on the graph data structure. While it offers advantages such as fast processing of topology operations similar to native GDBMSs, making it useful for queries like BFS/DFS and triangle counting, it also has the following three disadvantages:

- **Limited Graph Size:** The materialized graph view is typically managed as an in-memory graph data structure, imposing limits on the size of the graph that can be processed [30, 33, 51, 70].
- **Data Recency:** It is challenging to obtain the most up-to-date data solely from the materialized graph view since the view is derived from underlying tables [30, 33, 70]. Few methods like DuckPGQ [70] generate views dynamically, but this process can introduce significant delays, especially for interactive queries.
- **Inefficient Query Processing:** MGV requires separate query processing layers for topologies and properties, which can result in inefficient query plans for the queries involving both topology and property operations (*hybrid queries*) [30, 33, 70].

To address the challenges outlined above, we propose a novel system design named *Chimera* that extends an RDBMS into a GRDBMS capable of efficiently supporting SQL/PGQ. Unlike MGV that treats graph topologies as secondary objects extracted from relational tables, Chimera treats topologies as first-class citizens within its architecture. It adopts a dual-store architecture comprising separate stores for graph topologies (called *graph store*) and properties (called *relational store*). Each store has its own data format, access methods, and low-level query operators. Chimera maintains pointers from the graph store to the relational store, facilitating rapid identification of the properties of a specific vertex or edge. By storing topologies in a disk-based store rather than in main memory, it eliminates constraints on graph size imposed by memory capacity. Moreover, it employs a shared transaction manager for both stores, ensuring immediate and rigorous updates to graphs defined by SQL/PGQ — outperforming the MGV approach in terms of data recency and responsiveness. Although Chimera has separated stores for topologies and properties, it has a unified query processing layer rather than separated stacks of query processing layers for them. In general, designing a unified query processing and optimization framework for disparate stores with distinct data models poses significant challenges. To overcome this challenge, we propose a novel operator called $Traversal\text{-}Join(TJ)$ that relaxes the closedness of the algebraic operator within the relational model by allowing it to take both topologies and properties as operands. This operator is capable of performing traversal, join, or mapping between topologies and properties based on the operands involved. By leveraging the TJ operator, Chimera can formulate a *unified query plan*, referred to as the TJ plan for any given SQL/PGQ query. Since each plan has different costs depending on the order of operators including the TJ operator, we propose a new cost model that specifically considers the TJ operator and an optimization method based on the model. This allows Chimera to generate more efficient query plans compared to the MGV approach, particularly for hybrid queries. To demonstrate the effectiveness of the proposed approach, we implemented Chimera by introducing new access methods, traversal operators, and the TJ operator, while extending the capabilities of the existing query planner, optimizer, and parser within the open-source relational DBMS PostgreSQL [27].

Our major contributions can be summarized as follows:

- We propose a novel system design named Chimera that extends an RDBMS to a GRDBMS supporting SQL/PGQ.
- We propose a dual-store architecture consisting of separate graph and relational stores, along with their access methods.
- We propose a new query operator, the Traversal-Join (TJ) operator, capable of handling both topologies and properties as operands.
- We propose a unified cost-based query optimization method tailored for hybrid queries of topology and property operations.
- Through extensive experiments, we demonstrate Chimera significantly outperforms existing GRDBMSs in terms of response time, throughput, and graph size for interactive SQL/PGQ queries.

The rest of this paper is organized as follows. Section 2 introduces SQL/PGQ and the GT and MGV approaches. Section 3 presents dual store of Chimera. Section 4 and 5 present the TJ operator and query optimization, respectively. Section 6 presents the results of the evaluation. Finally, we mention related work in Section 7 and conclude this paper in Section 8.

## 2 PRELIMINARIES

### 2.1 SQL/PGQ

*2.1.1* ***Graph view definition language*** The graph view definition language defines graphs to be viewed from an RDB. A single definition query contains the declaration of a single property graph including its name, its vertex tables, and its edge tables, as shown in Figure 1. The vertex tables specify the vertices, their labels, and their properties from the a set of base tables in the RDB. For example, Figure 1(a) defines a graph named snb, which specifies three types of vertices of the labels, User, Post, and Comment, from the base tables of UserT, PostT, and CommentT. The edge tables specify the edges, their labels, and their properties using a set of src and dst column pairs from the base tables in the RDB. For example, Figure 1(a) specifies two types of edges of the labels, Likes and Replyof, using the set of column pairs, ⟨UserT(id), PostT(id)⟩ and ⟨CommentT(id), PostT(id)⟩. Figure 1(b) shows an example graph defined by Figure 1(a) from a tiny RDB like the LDBC Social Network Benchmark (SNB) dataset [1] [18].

[1]LDBC SNB is the de-facto standard graph query benchmark, and the Person label has been replaced with User for convenience of query plan representation

#### 2.1.2 ***Graph pattern matching language***

The Graph pattern matching language (GPML) performs graph pattern matching on a graph defined by the graph view definition language above. In SQL/PGQ, GPML can be inserted into the FROM clause of SQL as a sub-language using the keyword GRAPH_TABLE. GPML is extended from Conjunctive Regular Path Queries (CRPQ) [12], which includes Conjunctive Query (CQ) and Regular Path Query (RPQ). We focus on the former in this paper because typical interactive queries such as neighborhood, subgraph, and search queries mentioned in Section 1 belong to CQs.

GPML consists of MATCH-WHERE-COLUMNS clauses, where the MATCH clause declares the topological patterns to be traversed in the graph, which are expressed using ASCII-art, the WHERE clause declares the filtering conditions and the COLUMNS clause declares the schema of the output table for the patterns traversed.

```
CREATE PROPERTY GRAPH snb
VERTEX TABLES (
  UserT PROPERTIES (name) LABEL User
  PostT PROPERTIES (content, creationDate) LABEL Post
  CommentT PROPERTIES (content) LABEL Comment
) EDGE TABLES (
  LikeT SOURCE KEY (src) REFERENCES UserT(id)
        DESTINATION KEY (dst) REFERENCES PostT(id)
        LABEL Likes
  ReplyofT SOURCE KEY (src) REFERENCES CommentT(id)
          DESTINATION KEY (dst) REFERENCES PostT(id)
          LABEL Replyof
);
```

**(a) Example of graph view definition language**

UserT

| ID | name |
|---|---|
| 1 | James |
| 2 | Hana |

PostT

| ID | content | creationDate |
|---|---|---|
| 3 | … | 12-19T02:20 |
| 4 | … | 12-19T02:30 |

CommentT

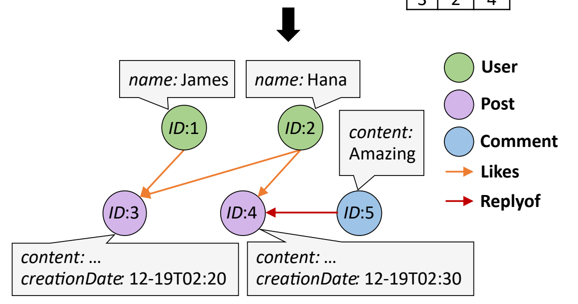
| ID | content |
|---|---|
| 5 | Amazing |

LikesT

| ID | src | dst |
|---|---|---|
| 1 | 1 | 3 |
| 2 | 2 | 3 |
| 3 | 2 | 4 |

ReplyofT

| ID | src | dst |
|---|---|---|
| 4 | 4 | 5 |



**(b) Example of graph view definition**

**Figure 1: Example of graph view in SQL/PGQ.**

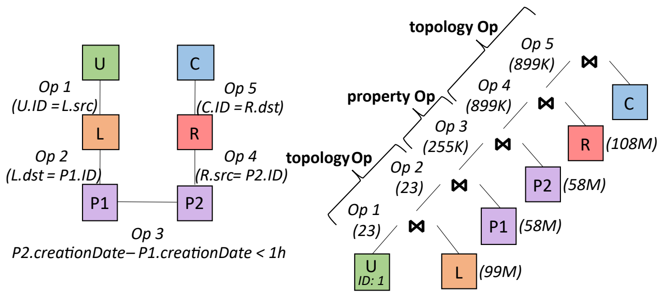
Figure 2 shows an example GPML query that recommends to the user whose ID is 1 (shortly, User 1), a set of posts that were created at a similar time to the posts that the user liked. It is a kind of content-based recommendation [39]. In Figure 2, the first expression in MATCH declares a pattern of User and Post(p1) connected by Likes where User ID is 1, and the second expression in MATCH declares a pattern of Post(p2) and Comment connected by Replyof, and the next WHERE clause declares a filtering condition between p1 and p2 where the difference in their creation time is less than 1 hour. The COLUMNS clause declares the posts(p2) with their comments as output. For Figure 1(b), this GPML query returns {(ID:4, ID:5)}, and thus, the whole SELECT-FROM query returns {(4, 1)}.

```
SELECT recommends.p2, count(recommends.c)
FROM GRAPH_TABLE (snb,
  MATCH (u:User WHERE u.ID = 1)-[l:Likes]->(p1:Post),
        (p2:Post)<-[r:ReplyOf]-(c:Comment)
  WHERE p2.creationDate - p1.creationDate < 1h
  COLUMNS (p2.ID, c.ID) recommends
);
```

**Figure 2: An example GPML query.**

## 2.2 Graph Table (GT) approach

The GT approach creates a graph virtually without changing the storage and query processing layer of RDBMSs. Examples of this approach include SQL Server graph extension [64] and Agensgraph [10]. This approach has a strong point that it can easily optimize a hybrid query, since a plan contains only relational operators. It performs joins for traversals, where the join keys are the ID column of a base vertex table (e.g., ID of UserT in Figure 1(b)) and the src (or dst) column of a base edge table (e.g., src of LikeT in Figure 1(b)). Figure 3(a) shows the join graph of the GPML query in Figure 2, and Figure 3(b) shows an example query plan generated from the join graph. In the figures, the topology operations are Op 1 ((u:User)-[l:Likes]), Op 2 ([l:Likes]->(p1:Post)), Op 4 ((p2:Post)<-[r:ReplyOf]), and Op 5 ([r:ReplyOf]-(c:Comment)), and the property operation is Op 3 (p2.creationDate - p1.creationDate < 1h).



**(a) Example of join graph** **(b) Example of query plan**

**Figure 3: Example of a query plan in the GT approach.**

In the GT approach, indexed nested loop (INL) and hash joins are commonly used for a traversal, and each join-based traversal incurs a large overhead since it requires searching an index or scanning the entire vertex/edge tables to find adjacent vertices/edges. For example, an INL join for Op 1 in Figure 3 performs an index seek on the src column of LikesT to find adjacent edges to User 1 and so incurs the overhead of O(log$|E|$). Instead, a hash join for Op 1 performs a hash lookup on LikesT after building a hash table on User 1 and so incurs the overhead of O($|E|$). Similarly, INL and hash joins for Op 2 incur the overhead of O(log$|V|$) and O($|V|$), respectively.

## 2.3 Materialized Graph View (MGV) approach

The MGV approach stores the topologies extracted from the tables as a materialized graph view. Examples of this approach include

DuckPGQ [70], GRFusion [30], and GrainDB [33]. They have the three disadvantages as described in Section 1. We explain the third disadvantage in more detail, i.e., inefficient query plans from separated stacks of processing layers for topologies and properties.

DuckPGQ [70] constructs a read-optimized graph data structure, but generates a query plan that consists of only relational operators like the GT approach (hereafter called the *GT plan*) for a CQ. It only utilizes the graph structure for a specific RPQ called bulk pathfinding to accelerate it. GrainDB [33] basically generates a GT plan, but improves it by replacing a hash join in the plan with so-called S-join [33] that utilizes the graph structure (hereafter called the *S-join plan*). GRFusion [30] splits a GPML query into two parts: the one requiring topology operations such as traversal ($\odot$) and the other one requiring property operations such as joins ($\bowtie$). It generates a sub-plan for each part and binds two sub-plans using a binding join (hereafter called the *B-join plan*).

## 3 DUAL STORE OF CHIMERA

### 3.1 Model and Architecture

In this section, we present the model and architecture of Chimera to solve the issues raised in the GT and MGV approaches. Chimera has a dual-store architecture of *relational store* for properties and *graph store* for topologies. Different from MGV, our graph store is not a read-optimized in-memory graph structure, which is more suitable for OLAP, but an updatable disk-based one, which is more suitable for OLTP. In addition, to avoid two drawbacks of MGV, the delayed update of the graph view and the binding overhead in the last step, Chimera connects both stores by maintaining pointers between vertex/edge in the graph store and the corresponding tuples in the relational store. We define the graph model of Chimera in Definition 3.1.

*Definition 3.1.* **Graph model of Chimera**

A graph G in Chimera is defined as $G = (V, E, \Psi, \Sigma, L(\cdot), B(\cdot))$.

- $V$ and $E$ are sets of vertices and edges, respectively
- $\Psi$ is a set of tuples in the vertex tables
- $\Sigma$ is a set of tuples in the edge tables
- $L(\cdot)$ is a labeling function that takes a vertex or an edge as input and returns its label as output
- $B(\cdot)$ is a bijective function that takes a vertex or an edge as input and returns a corresponding tuple as output

$V, E, L(\cdot)$, and $B(\cdot)$ are stored in the graph store, while $\Psi$ and $\Sigma$ are stored in the relational store. $B(\cdot)$ can be implemented as either a pointer from the graph store to relational store or a pointer from the relational store to graph store. We choose the former in this paper because the latter may cause a large change in the relational store for storing pointers. The pointers can be either logical or physical, where we choose the former for flexibility in update.

Figure 4 compares the architectures of MGV and Chimera. We omit the GT approach since it only adds a parser for SQL/PGQ and stores metadata for graph views, making it relatively simple. MGV has a stack of layers for graph query processing which is separated from the RDBMS. In contrast, Chimera has a common transaction manager and query processing layers for two separated stores.

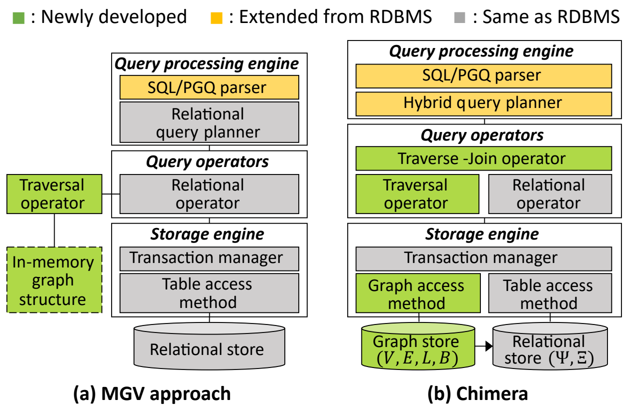


**Figure 4: Comparison of architectures.**

### 3.2 Storage format

The graph store of Chimera consists of a set of vertex records and edge records, which is similar to a native graph DBMS. Figure 5(a) shows the storage format of vertex and edge records. A vertex record has a vertex ID, a label, and pointers to the first out/in edges. Here, if the total number of labels of its out-edges is $N$, it contains the $N$ first pointers to quickly find the first edge of a specific label. Similarly, it contains the $M$ first pointers for its in-edges. An edge record has an edge ID, a label, pointers to the source and destination vertices, and pointers to the next out/in edges, like in an adjacency list. Each vertex/edge record has a pointer to the corresponding tuple in the relational store for $B(\cdot)$ and a header for transaction management, which will be explained in Section 3.5. To ensure the uniqueness of vertex/edge IDs, we employ a system counter (e.g., auto_increment). This counter is distinct from the one used for the primary key (PK), typically resulting in a different vertex/edge ID from the PK of the original tuple. For simplicity, however, the same ID is used for both the vertex/edge ID and the PK in Figure 5. Additionally, for pointers depicted as dotted arrows in Figure 5(c), we use logical addresses consisting of a page number and a page offset. These logical addresses enable flexible recycling of vertex/edge IDs, for example when updates occur. The buffer manager translates these logical addresses to physical addresses when accessing the pointers.

Figure 5(b) shows the example of pointers between vertex records and edge records for Figure 1(b), while Figure 5(c) the example of pointers between the graph store and the relational store. We show two kinds of pointers separately for an easy explanation. In Figure 5(b), User vertex records have a single out-edge pointer (i.e., $N = 1$ and $M = 0$), while Post vertex records have up to two in-edge pointers (i.e., $N = 0$ and $M = 2$). The Likes edge 3 has a next out-edge pointer to the Likes edge 2 (denoted as &L(2)), which indicates vertex 2 has two out-edges in Figure 1(b). In Figure 5(c), each vertex record has a label (e.g., User) for $L(\cdot)$ and a pointer (e.g., &UT(1)) to the corresponding tuple in the relational store for $B(\cdot)$. In this figure, there is no tuples for edges because the edges in Figure 1(b) have no properties.

### 3.3 Read access method

In this section, we present the read access methods of Chimera for the graph store of $(V, E, L(\cdot), B(\cdot))$ and the relational store of

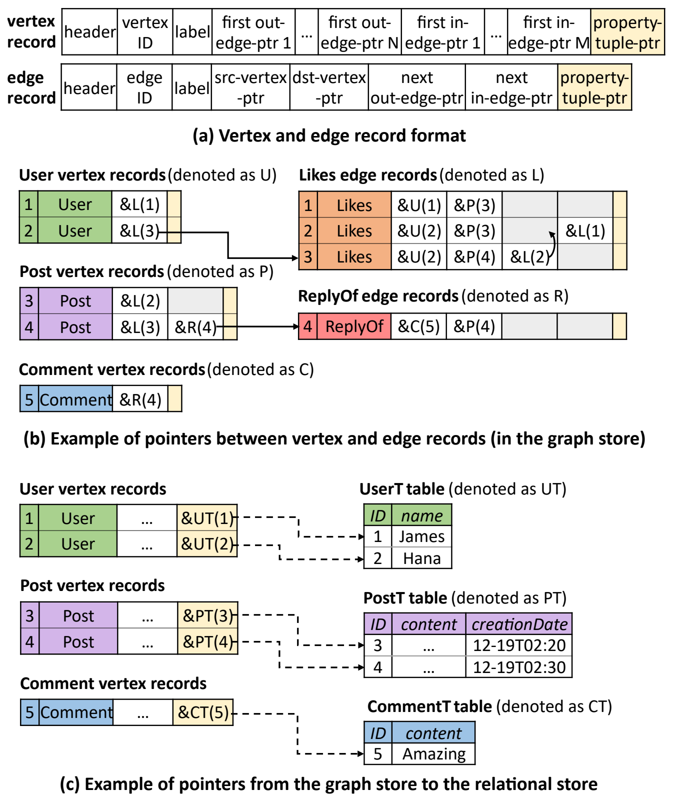

vertex record
header
vertex ID
label
first out-edge-ptr 1
...
first out-edge-ptr N
first in-edge-ptr 1
...
first in-edge-ptr M
property-tuple-ptr
edge record
header
edge ID
label
src-vertex-ptr
dst-vertex-ptr
next out-edge-ptr
next in-edge-ptr
property-tuple-ptr
(a) Vertex and edge record format
User vertex records (denoted as U)
Likes edge records (denoted as L)
Post vertex records (denoted as P)
ReplyOf edge records (denoted as R)
Comment vertex records (denoted as C)
(b) Example of pointers between vertex and edge records (in the graph store)
User vertex records
UserT table (denoted as UT)
Post vertex records
PostT table (denoted as PT)
Comment vertex records
CommentT table (denoted as CT)
(c) Example of pointers from the graph store to the relational store

Figure 5: Storage format of Chimera.

$(\Psi, \Sigma)$. The following summarizes basic scan access methods using labels. Since Chimera stores vertex and edge records separately for each label, it can find the set of records for a given label without scanning all records.

- **scanVertex**: read vertex records with a label $x$
  input = $x$ ; output = $\{p \in V \mid L(p) = x\}$
- **scanEdge**: read edge records with a label $y$
  input = $y$ ; output = $\{q \in E \mid L(q) = y\}$

The following summarizes the access methods to find the adjacent edges (or vertices) for a given vertex (or edge) by iterating over the stored edge (or vertex) pointers.

- **getAdjEdges**: read a set of out-edges (denoted as $\delta = \longrightarrow$) or a set of in-edges (denoted as $\delta = \longleftarrow$) with a label $y$ of a vertex $p$
  input = $(p, \delta, y)$ ;
  $$\text{output} = \begin{cases} \{q \in E \mid p = q.src \wedge L(q) = y\} & \text{if } \delta = \longrightarrow \\ \{q \in E \mid p = q.dst \wedge L(q) = y\} & \text{if } \delta = \longleftarrow \end{cases}$$
- **getAdjVertex**: read a dst-vertex (denoted as $\delta = \longrightarrow$) or a src-vertex (denoted as $\delta = \longleftarrow$) with a label $x$ from an edge $q$
  input = $(q, \delta, x)$ ;
  $$\text{output} = \begin{cases} \{p \in V \mid p = q.dst \wedge L(p) = x\} & \text{if } \delta = \longrightarrow \\ \{p \in V \mid p = q.src \wedge L(p) = x\} & \text{if } \delta = \longleftarrow \end{cases}$$

## 3.4 Write access method

In this section, we present the write access methods of Chimera. We present only the write access methods for insertion here and omit the remaining methods due to lack of space.

- **insertVertex**: insert a vertex record $p$ of the label $x$
- **insertEdge**: insert an edge record $q$ of the label $y$
- **insertTuple**: insert a tuple $r$ to the table $z$
  input = $(r, z)$ ; output = the tuple address of $r$

For a vertex having properties, both insertTuple and insertVertex must be executed together. Similarly, for an edge having properties, both insertTuple and insertEdge must be executed together. To ensure the atomicity, Chimera executes both insertions within the same transaction block. Both insertions are WAL logged [25] and so if a system fails after only insertTuple is executed, the system undoes the insertion for the uncommitted transaction through the stored log. Figure 6(a) shows an example of inserting a User vertex named Alex, which executes insertTuple(Alex, UserT) first to get the address of the tuple, &UT(6), creates a vertex object, and executes insertVertex using the vertex.

For insertEdge, there are usually three possible insertion positions in the adjacency list: the front, the middle, and the end. Chimera inserts it at the front for fast insertion, which requires updating the first out- or in-edge pointer of the src or dst vertex record. Thus, Chimera acquires a lock on the src or dst vertex to prevent concurrent reads on the vertex. Figure 6(b) shows an example of inserting a Likes edge from User 6 to Post 4, which consists of five steps as in the figure. In this example, Likes edge does not have a property, but if it did, the insertTuple would be performed first as in Figure 6(a) in Figure 6(b) in the same transaction block.

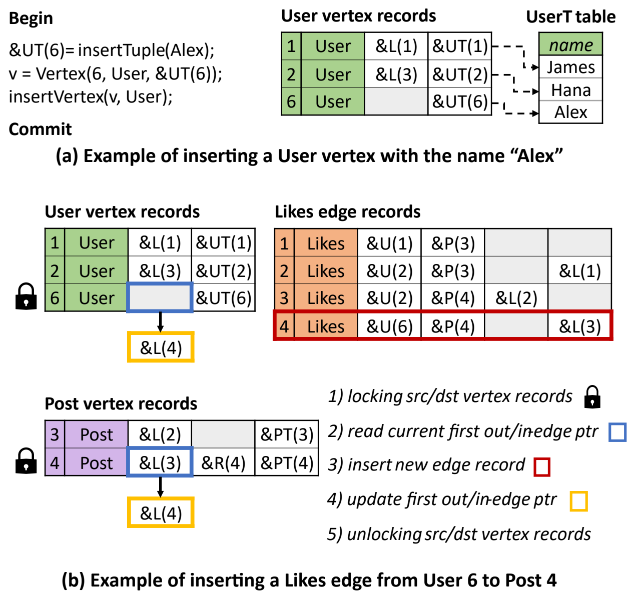

Begin
&UT(6)= insertTuple(Alex);
v = Vertex(6, User, &UT(6));
insertVertex(v, User);
Commit
User vertex records
UserT table
(a) Example of inserting a User vertex with the name "Alex"
User vertex records
Likes edge records
Post vertex records
1) locking src/dst vertex records
2) read current first out/in-edge ptr
3) insert new edge record
4) update first out/in-edge ptr
5) unlocking src/dst vertex records
(b) Example of inserting a Likes edge from User 6 to Post 4

Figure 6: Example of inserting a vertex and an edge.

## 3.5 Concurrency control

The isolation level of the concurrency control of Chimera is read committed [8], which is also the default and common one across many commercial DBMSs, including PostgreSQL [27], Oracle [49], and SQL server [44]. To ensure the read committed, we must prevent the dirty read [2] in which an uncommitted transaction's write is read by another concurrent transaction.

For doing this, we add a field of the transaction ID that inserts a record (called *insert-tid*) to the header of each vertex and edge record. Then, any transaction to write the record updates the insert-tid field with its ID, which indicates the the lifetime of the record is after the insert-tid. Similarly, an transaction to read the record checks the visibility of the record based on the lifetime. Algorithm 1 presents the pseudocode of the visibility checking for insertion. There are only two cases in which a record is visible: when the insert transaction is committed and the read transaction's ID is after the record's lifetime (Line 5), and when the insert transaction is running and the read transaction is the insert transaction itself (Line 8). Otherwise, it is invisible. Similarly, we can add the *delete-tid* field to the header and check the visibility, but omit the details.

**Algorithm 1:** Record visibility checking

```
1  Function isVisible(record, read-tid):
2      insert-txn ← getTransaction(record.insert-tid);
3      switch getStatus(insert-txn) do
4          case COMMIT do
5              if insert-tid < read-tid then
6                  return true;
7          case RUNNING do
8              if insert-tid = read-tid then
9                  return true;
10     return false;
```

## 4 TRAVERSAL-JOIN OF CHIMERA

### 4.1 Mapping operators

Due to the dual store, a query plan of Chimera needs to handle both the graph store and the relational store. Thus, each binary operator in the query plan performs one of the following three actions: (1) *traversal* taking its operands only from the graph store, (2) *join* taking its operands only from the relational store, and (3) *mapping* taking its operands from both stores. Among them, the mapping operator is defined based on $B(\cdot)$ and plays a role as a bridge between the graph and relational store. There are two mapping operators depending on the direction of mapping: graph to relational (G2R) and relational to graph (R2G). We define them in Definitions 4.1 and 4.2.

*Definition 4.1.* **G2R mapping operator**
The G2R mapping operator $\triangleright$ takes $\langle (V_1, E_1), T_1 \rangle$ as input, performs the one of the following operations depending on the the value of $(V_1, E_1)$, and returns its result tuples $T_2$.
**Case 1 ($V_1 \neq \emptyset$, $E_1 = \emptyset$) :** mapping from $V_1$ to $T_2 \subseteq T_1$ such that $T_2 = \{ B(v) \mid v \in V_1 \land B(v) \in T_1 \}$ (denoted as $V_1 \triangleright T_1$)
**Case 2 ($V_1 = \emptyset$, $E_1 \neq \emptyset$) :** mapping from $E_1$ to $T_2 \subseteq T_1$ such that $T_2 = \{ B(e) \mid e \in E_1 \land B(e) \in T_1 \}$ (denoted as $E_1 \triangleright T_1$)

*Definition 4.2.* **R2G mapping operator**
The R2G mapping operator $\triangleleft$ takes $\langle (V_1, E_1), T_1 \rangle$ as input, performs the one of the following operations depending on the the value of $(V_1, E_1)$, and returns its result vertices and edges $(V_2, E_2)$.
**Case 1 ($V_1 \neq \emptyset$, $E_1 = \emptyset$) :** mapping from $T_1$ to $V_2 \subseteq V_1$ such that $(V_2, E_2) = (\{B^{-1}(t) \mid t \in T_1 \land B^{-1}(t) \in V_1\}, E_1)$ (denoted as $V_1 \triangleleft T_1$)
**Case 2 ($V_1 = \emptyset$, $E_1 \neq \emptyset$) :** mapping from $T_1$ to $E_2 \subseteq E_1$ such that $(V_2, E_2) = (V_1, \{B^{-1}(t) \mid t \in T_1 \land B^{-1}(t) \in E_1\})$ (denoted as $E_1 \triangleleft T_1$)

In G2R and R2G mapping, $(V_1, E_1)$ is the result of read access methods such as scanVertex, scanEdge, getAdjEdges, and getAdjVertex, explained in Section 3.3. scanVertex returns only a set of vertices, and scanEdge returns only a set of edges. getAdjEdges returns only a set of edges, and getAdjVertex returns only a set of vertices, by performing a single-step traversal [57]. Therefore, we can define the mapping operators only for two cases: ($V_1 \neq \emptyset$, $E_1 = \emptyset$) or ($V_1 = \emptyset$, $E_1 \neq \emptyset$).

In G2R mapping, since $B(\cdot)$ can be directly implemented through the pointer from graph to relational store (i.e., *property-tuple-ptr* in Figure 5(a)), we omit the algorithm for G2R mapping. In R2G mapping, there is no pointer from relational to graph store. Thus, instead of obtaining $B^{-1}(t)$ from $t \in T_1$, we obtain $v \in V_1$ (or $e \in E_1$) that satisfies the condition $B(v) = t$ (or $B(e) = t$). In other words, it can be treated like a $\theta$-join with $B(v) = t$ (or $B(e) = t$) as $\theta$. Although various algorithms can be used for R2G mapping, we just present a basic algorithm using nested-loop join in Algorithm 2. Case 1 of R2G mapping is handled by comparing *v.property-tuple-ptr* with *&t* (address of a tuple $t$) corresponding to $B(v) = t$ (Line 2). Meanwhile, we can also improve R2G like G2R by storing and managing a pointer to $B^{-1}$.

**Algorithm 2:** R2G mapping operator

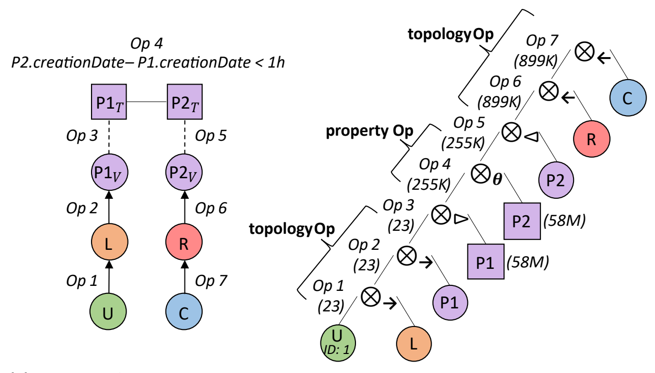
```
   Input: $(V_1, E_1)$, $T_1$                      /* left, right operand */
1  forall $v \in V_1$, $t \in T_1$ do
2      if v.property-tuple-ptr = &t then
3          $V_2, E_2 \leftarrow V_2 \cup \{ v \}, E_1$;
4  forall $e \in E_1$, $t \in T_1$ do
5      if e.property-tuple-ptr = &t then
6          $V_2, E_2 \leftarrow V_1, E_2 \cup \{ e \}$;
7  emit($V_2, E_2$);
```

### 4.2 Traversal-Join operator

Based on the mapping operators, we can define a unified operator for both traversal and join. We call this the Traversal-Join(TJ) operator. It performs traversal, join, or mapping in the same framework and so the query processing layers can generate, optimize, and execute a query plan like there is only a single store. We define it in Definition 4.3.

*Definition 4.3.* **Traversal-Join operator**
The Traversal-Join (TJ) operator $\otimes$ is a binary operator that takes $H_1, H_2$ as input, performs one of the following four cases depending on the combination of operands $(H_1, H_2)$.
**Case 1 ($H_1 \odot_\delta H_2$) :** $\delta$-direction traversal, if $H_1 = (V_1,E_1)$, $H_2 = (V_2,E_2)$
**Case 2 ($H_1 \bowtie_\theta H_2$) :** $\theta$-join, if $H_1 = T_1$, $H_2 = T_2$
**Case 3 ($H_1 \triangleright H_2$) :** G2R mapping, if $H_1 = (V_1, E_1)$, $H_2 = T_1$
**Case 4 ($H_1 \triangleleft H_2$) :** R2G mapping, if $H_1 = T_1$, $H_2 = (V_1, E_1)$

Chimera generates a query plan contains TJ operators for a SQL/PGQ query involving both topology and property operations. Then, it performs a unified query optimization by determining the optimal order of TJ operators based on a cost model (more detail in Section 5). Figure 7(b) shows an example of a query plan in Chimera for the query in Figure 2. In the figure, Ops 1, 2, 6, and 7 are Case 1 (i.e., traversal) of TJ operator, Op 4 is Case 2 (i.e., join) TJ operator, Op 3 is Case 3 (i.e., G2R mapping), and Op 5 is Case 4 (i.e., R2G mapping). Figure 7(b) is generated from Figure 7(a), and we explain the details of the TJ plan generation in Section 5.1.



(a) Example of traversal-join graph (b) Example of query plan

**Figure 7: Example of a query plan in Chimera.**

Algorithm 3 represents the pseudocode of the TJ operator, which is based on the volcano model [23], commonly used in many RDBMSs. It calls the next() function of the child operator, performs one of the four cases of operations, and feeds the result to the parent operator through the emit() function. For example, Op 1 in Figure 7(b) takes $V_1 = \{\text{U1}\}$ as the left operand and $E_2 = \{\text{L1, L2, L3}\}$ as the right operand as input. It performs traversal from a User vertex to Likes edge (Line 5) and emits $E_3 = \{\text{L1}\}$ as output (Line 9). In this case, the right operand of Op 1 is the scanEdge(Likes) without any filtering conditions, and so the intersection process in Line 8 can be omitted because $E_{adj}$ (i.e., $\{\text{L1}\}$) obtained by traversal always belongs to $E_2$ (i.e., $\{\text{L1, L2, L3}\}$). The rest examples are shown in Table 1. In Algorithm 3, the $\theta$-join operator is represented using a nested-loop join, but it can also be processed by using other join algorithms such as hash join.

In Algorithm 3 and Table 1, we explain $\delta$-direction traversal takes only a set of vertices or edges currently being visited (i.e., active vertex or edge [66]) as operands, for simplicity. However, the actual implementation of Chimera takes a set of paths for path patterns in a SQL/PGQ query. For example, due to (p2:Post)<-[r:ReplyOf]-(c:Comment) in a MATCH clause in Figure 2, Op 7 in Table 1 takes {(p2:P4)<-[r:R4]} as the left operand and {(c:C5)} as the right operand as input, and emits {(p2:P4)<-[r:R4]-(c:C5)}. The GPML query finally returns {(ID: 4, ID: 5)} taking {(p2:P4)<-[r:R4]-(c:C5)} from Op 7 due to (p2.ID, c.ID) in a COLUMNS clause.

## 5 QUERY OPTIMIZATION OF CHIMERA

### 5.1 Plan generation

In this section, we present a plan generation method of a unified query plan (called TJ plan) that consists of TJ operators. For the generation, we extend the conventional join graph (e.g., Figure 3(a)) to the Traversal-Join (TJ) graph, which is defined in Definition 5.1. Figure 7(a) shows an example of $G_{TJ}$ for the query in Figure 2. The vertices represent the labels and tables used in the query, and the edges between the vertices represent the traversal, join, and mapping operations.

*Definition 5.1.* **Traversal-Join graph**

Traversal-Join graph $G_{TJ}$ is a mixed graph consisting of vertices $(L_v, L_e, L_t)$, directed edges $(O_T)$, and undirected edges $(O_J, O_M)$.

**Algorithm 3:** TJ OPERATOR (⊗)

```
   Input: left, right                      /* left, right child operator */
   Input: $\delta, \theta$                 /* optional input */
1  $H_1, H_2 \leftarrow$ left.next(), right.next();
2  switch $H_1, H_2$ do
3      case $(V_1, E_1), (V_2, E_2)$ do    /* traversal operation */
4          forall $v_1 \in V_1$ do
5              $E_{adj} \leftarrow E_{adj} \cup$ getAdjEdges($v_1, \delta, E_2$.label);
6          forall $e_1 \in E_1$ do
7              $V_{adj} \leftarrow V_{adj} \cup$ getAdjVertex($e_1, \delta, V_2$.label);
8          $V_3, E_3 \leftarrow V_{adj} \cap V_2, E_{adj} \cap E_2$;
9          emit($V_3, E_3$);
10     case $T_1, T_2$ do                  /* join operation */
11         forall $t_1 \in T_1, t_2 \in T_2$ do
12             if $\theta(t_1, t_2) = true$ then
13                 $T_3 \leftarrow T_3 \cup \{(t_1 \cdot t_2)\}$;
14         emit($T_3$);
15     case $(V_1, E_1), T_1$ do           /* G2R mapping operation */
16         $T_2 \leftarrow$ G2Rmapping($(V_1, E_1), T_1$);
17         emit($T_2$);
18     case $T_1, (V_1, E_1)$ do           /* R2G mapping operation */
19         $(V_2, E_2) \leftarrow$ R2Gmapping($(V_1, E_1), T_1$);
20         emit($V_2, E_2$);
```

**Table 1: Example of input and output of TJ operators.**

| Op | Left operand ($H_1$) | Right operand ($H_2$) | Output |
|---|---|---|---|
| 1 | $V_1 = \{\text{U1}\}, E_1 = \emptyset$ | $V_2 = \emptyset, E_2 = \{\text{L1, L2, L3}\}$ | $V_3 = \emptyset, E_3 = \{\text{L1}\}$ |
| 2 | $V_1 = \emptyset, E_1 = \{\text{L1}\}$ | $V_2 = \{\text{P3, P4}\}, E_2 = \emptyset$ | $V_3 = \{\text{P3}\}, E_3 = \emptyset$ |
| 3 | $V_1 = \{\text{P3}\}, E_1 = \emptyset$ | $T_1 = \{\text{PT3, PT4}\}$ | $T_2 = \{\text{PT3}\}$ |
| 4 | $T_1 = \{\text{PT3}\}$ | $T_2 = \{\text{PT3, PT4}\}$ | $T_3 = \{\text{PT4}\}$ |
| 5 | $T_1 = \{\text{PT4}\}$ | $V_1 = \{\text{P3, P4}\}, E_1 = \emptyset$ | $V_2 = \{\text{P4}\}, E_2 = \emptyset$ |
| 6 | $V_1 = \{\text{P4}\}, E_1 = \emptyset$ | $V_2 = \emptyset, E_2 = \{\text{R4}\}$ | $V_3 = \emptyset, E_3 = \{\text{R4}\}$ |
| 7 | $V_1 = \emptyset, E_1 = \{\text{R4}\}$ | $V_2 = \{\text{C5}\}, E_2 = \emptyset$ | $V_3 = \{\text{C5}\}, E_3 = \emptyset$ |

- $L_v$ and $L_e$ are the set of vertex and edge labels, respectively
- $L_t$ is a set of table names for the vertex and edge tables
- $O_T = \{(x, \delta, y) \mid (x \in L_v \wedge y \in L_e) \vee (x \in L_e \wedge y \in L_v)\}$ is a set of traversal operations in the $\delta$ direction between $x$ and $y$
- $O_J = \{(x, \theta, y) \mid x \in L_t \wedge y \in L_t\}$ is a set of $\theta$-join operations
- $O_M = \{(x, y) \mid (x \in (L_v \cup L_e) \wedge y \in L_t) \vee (x \in L_t \wedge y \in (L_v \cup L_e))\}$ is a set of mapping operations

Algorithm 4 represents the plan generation algorithm. It uses a dynamic programming technique [46] by enumerating plans in order of low to high degree. Here, the degree of a graph $G$ is the number of vertices, which we denote by $|G|$. During the enumeration process, the selected subplans are stored in PlanMap and the calculated costs are stored in CostMap. First, we generate scan plans for all vertex labels, edge labels, and tuple tables of degree 1 (Lines 3-5). Then, we set $i$ and $j$, the degrees of the left and right plans to be generated, by incrementing degree (Lines 6-8). Next, we find all possible pairs of subgraphs $(S_1, S_2)$ of the $G_{TJ}$ where $|S_1| = i$, $|S_2| = j$, $S_1 \cap S_2 = \emptyset$ (Line 9) and $Op = (x, y)$ connecting the two subgraphs (Line 11). Then, we generate a TJ plan with $S_1$ and

$S_2$ as left and right plans (Lines 12-17). Finally, we select the best plan based on the cost model compared to the previously generated plans (Line 19). For example, the subplan for *Op 1* in Figure 7(b) is generated when $deg = 2$, $i = 1$, $j = 1$, and $S_1 = \{U\}$, $S_2 = \{L\}$, $Op =$ (U, ⟶, L) are selected. Additional examples are shown in Table 2. For simplicity, the examples focus on generating a left-deep plan, although Algorithm 4 is also capable of generating bushy plans.

**Algorithm 4:** TJ plan generation

```
   Input: $G_{TJ} = (L_v, L_e, L_t, O_T, O_J, O_M)$
1  PlanMap /* storing subplans while bottom-up enumeration */
2  CostMap     /* storing the calculated costs for each plan */
3  for $l_v \in L_v$ do PlanMap[$l_v$] ⟵ scanVertex($l_v$);
4  for $l_e \in L_e$ do PlanMap[$l_e$] ⟵ scanEdge($l_e$);
5  for $l_t \in L_t$ do PlanMap[$l_t$] ⟵ scanTable($l_t$);
6  for $2 \le deg \le |G_{TJ}|$ do                    /* size of plan */
7      for $1 \le i < deg$ do                      /* size of left plan */
8          $j \leftarrow deg - i$;                 /* size of right plan */
9          forall $(S_1, S_2) \subset G_{TJ}$ do  /* subgraph pair $S_1, S_2$ */
10             Plan$_{S_1}$, Plan$_{S_2}$ ⟵ PlanMap[$S_1$], PlanMap[$S_2$];
11             switch $Op = (x \in S_1, y \in S_2)$ do
12                 case $Op \in O_T$ do
13                     Plan$_{new}$ ⟵ ⊗ (Plan$_{S_1}$, Plan$_{S_2}$, $Op.\delta$);
14                 case $Op \in O_J$ do
15                     Plan$_{new}$ ⟵ ⊗ (Plan$_{S_1}$, Plan$_{S_2}$, $Op.\theta$);
16                 case $Op \in O_M$ do
17                     Plan$_{new}$ ⟵ ⊗ (Plan$_{S_1}$, Plan$_{S_2}$);
18             Plan$_{old}$ ⟵ PlanMap[$S_1 \cup S_2$];
19             PlanMap[$S_1 \cup S_2$] ⟵ selectBest(Plan$_{new}$, Plan$_{old}$);
20 Function selectBest($P_{new} = \otimes(P_1, P_2)$, $P_{old}$):
21     $Cost_{P1}$ ⟵ CostMap[$P_1$];
22     $Cost_{P2}$ ⟵ CostMap[$P_2$];
23     CostMap[$P_{new}$] = calculateTJCost($Cost_{P1}$, $Cost_{P2}$);
24     return minCostPlan($P_{new}$, $P_{old}$);
```

**Table 2: Example of TJ plan generation process.**

| deg | $S_1$ | $S_2$ | op | plan$_{S_1 \cup S_2}$ |
|---|---|---|---|---|
| 2 | $\{U\}$ | $\{L\}$ | (U, ⟶, L) | $\otimes_{\rightarrow}$ |
| 3 | $\{U, L\}$ | $\{P1_V\}$ | $(L, \rightarrow, P1_V)$ | $\otimes_{\rightarrow}$ |
| 4 | $\{U, L, P1_V\}$ | $\{P1_T\}$ | $(P1_V, P1_T)$ | $\otimes_{\triangleright}$ |
| 5 | $\{U, L, P1_V, P1_T\}$ | $\{P2_T\}$ | $(P1_T, \theta, P2_T)$ | $\otimes_{\theta}$ |
| 6 | $\{U, L, P1_V, P1_T, P2_T\}$ | $\{P2_V\}$ | $(P2_T, P2_V)$ | $\otimes_{\triangleleft}$ |
| 7 | $\{U, L, P1_V, P1_T, P2_T, P2_V\}$ | $\{R\}$ | $(P2_V, \leftarrow, R)$ | $\otimes_{\leftarrow}$ |
| 8 | $\{U, L, P1_V, P1_T, P2_T, P2_V, R\}$ | $\{C\}$ | (R, ⟵, C) | $\otimes_{\leftarrow}$ |

## 5.2 Cost model

In this section, we present the cost model of the TJ operator. We consider the disk I/O and CPU computation as the measure of the cost, as in conventional disk-based DBMSs [7, 24, 38]. For buffer cache effect, we utilize the base system's disk I/O prediction function $Z(\cdot)$, which is based on a study [41] that predicts the buffer cache effect for a more accurate cost calculation. Table 3 summarizes the symbols used in our cost model.

Since the TJ operator operates in one of four cases, the cost of the TJ operator is also one of $Cost_{traversal}$, $Cost_{join}$, $Cost_{G2R}$, or $Cost_{R2G}$ depending on the combination of operands. For simplicity, we describe the cost under a nested-loop join and left-deep plan only.

**Table 3: Summary of symbols for the cost model.**

| Type | Symbol | Description |
|---|---|---|
| Cost | Cost$_{L/R}$ | Cost of the left / right subplan |
| | Cost$_{I/O}$ | Disk I/O cost |
| | Cost$_{cpu}$ | CPU computation cost |
| Function | $Z(\cdot)$ | I/O estimation function for data access |
| Statistics | $N_L$ | Number of outputs of the left plan |
| | $N_R$ | Number of outputs of the right plan |
| | $d$ | Avg. degree of a vertex in graph store |

A single traversal is performed by getting $E_{adj}$ (or $V_{adj}$), the adjacent edges (or vertices) to the left operand (Lines 4-7 in Algorithm 3), and intersecting them with $E_2$ (or $V_2$), the set of edges (or vertices) obtained from the right operand (Line 8 in Algorithm 3). Thus, we can define the $Cost_{traversal}$ as in Eq. (1).

$$Cost_{traversal} = Cost_{getAdj} + Cost_{intersect} \tag{1}$$

For $Cost_{getAdj}$, the costs of $V_1 \,\delta\, E_2$ (i.e., edge traversal) and $E_1 \,\delta\, V_2$ (i.e., vertex traversal) are different. The former cost becomes disk I/O of $Z(d)$ due to $d$ data access to get adjacent edges and $d$ CPU computations to union with $E_{adj}$ on average, for each vertex from the left operand. Similarly, the latter cost becomes disk I/O of $Z(1)$ due to a single data access to get a src or dst vertex and a single CPU computation to union with $V_{adj}$, for each edge from the left operand. Thus, we define the $Cost_{getAdj}$ as in Eq. (2).

$$Cost_{getAdj} = \begin{cases} Cost_L + N_L \times (Cost_{I/O} \times Z(d) + Cost_{cpu} \times d) & \text{if } V_1 \,\delta\, E_2 \\ Cost_L + N_L \times (Cost_{I/O} \times Z(1) + Cost_{cpu}) & \text{if } E_1 \,\delta\, V_2 \end{cases} \tag{2}$$

For $Cost_{intersect}$, the computation is min($|E_{adj}|$, $|E_2|$) for edge traversal, and min($|V_{adj}|$, $|V_2|$)) for vertex traversal based on the hash-based intersection algorithm [16]. In here, $|E_{adj}| = N_L \times d$ because there are $d$ adjacent edges for each vertex from the left operand on average, $|V_{adj}| = N_L$ because there is a single src (or dst) vertex for each edge from the left operand, and $|E_2| = N_R$ because $E_2$ is obtained from the right operand. Thus, we define the $Cost_{intersect}$ as in Eq. (3). However, as mentioned in Section 4.2, if there are no filtering conditions in the right operand, the intersection process can be omitted, in which case $Cost_{intersect} = 0$.

$$Cost_{intersect} = \begin{cases} Cost_R + Cost_{cpu} \times min(N_L \times d, N_R) & \text{if } V_1 \,\delta\, E_2 \\ Cost_R + Cost_{cpu} \times min(N_L, N_R) & \text{if } E_1 \,\delta\, V_2 \end{cases} \tag{3}$$

For $Cost_{join}$, there is a nested for-loop with a left operand as outer and a right operand as inner, and a single CPU computation for each pair of tuples, so we can define the $Cost_{join}$ as Eq. (4).

$$Cost_{join} = Cost_L + N_L \times Cost_R + N_L \times N_R \times Cost_{cpu} \tag{4}$$

For $Cost_{R2G}$, we use Eq. (4) because it is treated like a join as described in Section 4.1. For $Cost_{G2R}$, it requires disk I/O of $Z(1)$ due to a single data access to find a property tuple and a single CPU computation to union with $T_2$, for each vertex (or edge) from the left operand. Thus, we define the $Cost_{G2R}$, as Eq. (5).

$$Cost_{G2R} = Cost_L + N_L \times Cost_{I/O} \times Z(1) \quad (5)$$

In a query plan, there are other relational operators such as selection and projection in addition to TJ operators. For them, we utilize the optimization heuristics used in the query planner of the base system (i.e., PostgreSQL) such as selection push-down and projection push-down.

## 6 EXPERIMENTAL EVALUATION

In this section, we compare Chimera with the existing GT and MGV-based systems in terms of response time (i.e., elapsed time) and throughput for SQL/PGQ interactive queries using the LDBC SNB benchmark. We also evaluate the characteristics of Chimera using a microbenchmark and its storage overhead.

### 6.1 Experimental Setup

*6.1.1* ***Datasets and queries*** For experiments, we use the de-facto standard graph query benchmark LDBC SNB [18]. Its dataset consists of 10 kinds of vertex labels and 17 kinds of edge labels, where each label has up to 10 properties in the context of a social network service. We use three scale factors: SF=30 (88.8M vertices and 540.9M edges), SF=100 (282.6M vertices and 1.8B edges), and SF=300 (817.3M vertices and 5.3B edges).

LDBC SNB supports both relational and graph data models, allowing the evaluation of graph relational query languages such as SQL/PGQ, T-SQL, and SQL + Cypher alongside the relational query language SQL [1]. We used all twelve Interactive Complex queries (IC1~IC12) and seven Interactive Short queries (IS1~IS7) from the LDBC SNB benchmark, following the approach used in the previous study [33]. IC are relatively complex queries, while IS are relatively simple queries that have no property operations, and in particular, IS4 is a single vertex scan with no topology operations. All IC and IS queries involve mapping operations. We also use four additional micro-benchmark queries (MICRO-1 ~ 4) for the further analysis of the queries containing property operations.

*6.1.2* ***Systems compared*** To compare the three approaches, GT, MGV, and Chimera, themselves, we prepared Chimera-GT and Chimera-MGV by implementing the GT and MGV approaches on top of Chimera, respectively, and compared them with our proposed Chimera-TJ. Chimera-GT generates GT plans, while Chimera-MGV generates MGV plans, in particular, GRFusion's B-Join plan explained in Section 2.3. We also compare Chimera-TJ with the existing GRDBMSs including SQL Server graph extension (SQL SG) [64], Agensgraph (AG) [10], DuckPGQ [70], GrainDB [33]. We also compared two RDBMSs, Umbra [47] and DuckDB [56], since the former is well known to process graph workloads fast, and the latter is the well-known base system of all major MGV-based GRDBMSs including DuckPGQ and GrainDB. We excluded GRFusion [30] since it fails in running the benchmark. Instead, we evaluate Chimera-MGV that implements GRFusion's method as above. Table 4 summarizes the systems compared.

*6.1.3* ***HW/SW environment*** We conduct all the experiments on a server equipped with two 16-core 3.0GHz CPUs, 256GB of memory, and two U.2 SSDs of 7.68 TB (RAID 0). In our environment, we set Cost$_{I/O}$ and Cost$_{cpu}$ in the cost model to 1.0 and

**Table 4: Systems compared.**

| Systems | Architecture | Plan type | Query language | Base system |
|---|---|---|---|---|
| Umbra | RDBMS | GT plan | SQL | Umbra |
| DuckDB | | | | DuckDB |
| SQL SG | GT-based GRDBMS | | T-SQL | SQL Server |
| Agensgraph | | | SQL + Cypher | PostgreSQL |
| DuckPGQ | MGV-based GRDBMS | GT plan | SQL/PGQ | DuckDB |
| GrainDB | | S-join plan | SQL | |
| Chimera-GT | GT-based | GT plan | SQL/PGQ | PostgreSQL |
| Chimera-MGV | MGV-based | B-join plan | | |
| Chimera-TJ | TJ-based | TJ plan | | |

0.01, as used in PostgreSQL [28]. In terms of S/W, we use Umbra 30b000783, DuckDB 0.8.1 [13], SQL Server 2019 [45], Agensgraph 2.1.3 Community edition [9], DuckPGQ @7052baa [14], and GrainDB @f82a52e [34]. We set the buffer size to be 1/4 of the total RAM (i.e., 64GB), which is generally recommended [29]. In all experiments, each query was executed using a single thread.

### 6.2 Comparison of response time

Table 5 and Table 6 shows the average elapsed time for LDBC SNB queries. Due to lack of space, the experimental results for SF30 and SF300 in the Table 6 only show the top four most time-consuming queries. In the table, T.O. means time out (longer than 100,000 msec), W.A. means wrong answer (producing results that differ from those of other systems), and O.O.M. means out-of-memory. We note that Chimera-MGV and Chimera-TJ generate the same plan and so show the same performance for the queries having no property operations (e.g., IC7, IC8, and IC12). We also note that we exclude view generation time for MGV-based systems.

*6.2.1* ***Comparison among Chimera-GT / MGV / TJ***

For most of IC queries, Chimera-TJ largely outperforms Chimera-GT because the former handles traversal operations in a native manner, while the latter does them using INL joins requiring the costs of O(log$|V|$) and O(log$|E|$) for finding adjacent vertices and edges, respectively. On the other hand, some queries are simpler than other IC queries since they only involve a small number of topology operations. Thus, for such IC7, IC8, and IS queries, Chimera-GT is slightly faster than Chimera-TJ because the latter has an additional overhead of G2R mapping operations. However, the differences in elapsed time (i.e., overhead of mapping) are almost negligible, within 10 msec. We observe that Chimera-MGV performs similarly to Chimera-TJ on IC7, IC8, and IC12, which involve only topology operations and utilize native graph traversal. However, it exhibits significantly slower performance on the other queries due to generating and executing an inefficient plan that binds results from two separate sub-plans, as discussed in Section 2.3.

*6.2.2* ***Comparison with the GT-based GRDBMSs***

Chimera-TJ significantly outperforms Agensgraph and SQL SG for IC queries. The average improvement for IC queries is up to 1785× compared to SQL SG in SF100. Since Agensgraph is based on Postgres like Chimera, it shows almost the same performance trend with Chimera-GT. But, Agensgraph is slightly slower than

Table 5: Average elapsed time (msec) of queries for SF=100 (we exclude view generation time for MGV-based systems).

| Systems | SF100 | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IC1 | IC2 | IC3 | IC4 | IC5 | IC6 | IC7 | IC8 | IC9 | IC10 | IC11 | IC12 | IS1 | IS2 | IS3 | IS4 | IS5 | IS6 | IS7 |
| **Umbra** | 176 | 5332 | 776 | 1375 | 1070 | 1388 | 6.1 | 4.0 | 6511 | 327 | 147 | 7181 | 1.6 | 4.9 | 4.0 | 0.6 | 3.5 | 3.9 | 4.1 |
| **DuckDB** | 160 | 29199 | 35173 | 19271 | 4359 | 18257 | 7153 | 9932 | 30826 | 616 | 87 | 44029 | 3.7 | 37979 | 60 | 0.1 | 786 | 4202 | 28565 |
| **DuckPGQ** | 164 | 29607 | 35674 | 19214 | 4722 | 18174 | 7274 | 9884 | 31426 | 626 | 82 | 44780 | 3.7 | 37998 | 64 | 0.1 | 786 | 4205 | 28635 |
| **GrainDB** | 47 | 558 | 2303 | 8330 | T.O. | 7964 | W.A. | 54836 | W.A. | W.A. | 5.1 | 21375 | 1.3 | 352 | 118 | 0.2 | 53 | 158 | 176 |
| **GrainDB-MO** | 41 | 563 | 503 | 335 | 461 | 352 | 315 | 342 | 287 | W.A. | 12 | 755 | 1.3 | 346 | 7.1 | 0.2 | 0.8 | 160 | 188 |
| **SQL SG** | 4.5 | 48554 | 49629 | 24049 | 29770 | 4161 | 51255 | 98522 | 85639 | 26.9 | 4.2 | 59143 | 1.2 | 21384 | 268 | 0.8 | 0.1 | 135 | 21729 |
| **Agensgraph** | 5.0 | 427 | 727 | 441 | 960 | 502 | 16 | 22 | 609 | 162 | 3.9 | 2967 | 1.1 | 23 | 2.1 | 0.1 | 3.0 | 20 | 9.1 |
| **Chimera-GT** | **1.1** | 447 | 513 | 445 | 388 | 628 | **5.2** | **3.8** | 526 | 24 | 3.6 | 1191 | 0.2 | 4.3 | 0.5 | 0.1 | 0.1 | 1.7 | 1.0 |
| **Chimera-MGV** | 61 | 79707 | T.O. | 42826 | 19303 | 1045 | 5.3 | 9.2 | T.O. | 278 | 413 | **519** | 1.1 | 9.5 | 1.3 | 0.1 | 0.9 | 8.7 | 4.0 |
| **Chimera-TJ** | 4.0 | **219** | **413** | **169** | **202** | **274** | | | **279** | **24** | **3.0** | | | | | | | | |

Table 6: Average elapsed time (msec) for SF30 and SF300.

| Systems | SF30 | | | | SF300 | | | |
|---|---|---|---|---|---|---|---|---|
| | IC3 | IC5 | IC9 | IC12 | IC3 | IC5 | IC9 | IC12 |
| **Umbra** | 329 | 725 | 635 | 3205 | 26247 | 15822 | 42167 | 96793 |
| **DuckDB** | 11108 | 1347 | 8647 | 13400 | 70224 | T.O. | T.O. | T.O. |
| **DuckPGQ** | 11175 | 1424 | 8862 | 13422 | O.O.M. | | | |
| **GrainDB** | 1194 | 1110 | W.A. | 4219 | O.O.M. | | | |
| **GrainDB-MO** | 267 | 194 | W.A. | 320 | O.O.M. | | | |
| **SQL SG** | 2775 | 1135 | 6998 | 6997 | T.O. | 83542 | T.O. | T.O. |
| **Agensgraph** | 453 | 292 | 336 | 891 | 752 | 1876 | 717 | 5222 |
| **Chimera-GT** | 431 | 294 | 218 | 891 | 752 | 1109 | 748 | 2113 |
| **Chimera-MGV** | T.O. | 7202 | 33992 | **493** | T.O. | T.O. | T.O. | **1109** |
| **Chimera-TJ** | **325** | **189** | **204** | | **577** | **528** | **310** | |

Chimera-GT for some queries because it stores all properties as json [26], regardless of column type, which introduces overhead due to the conversion between JSON and literal values. SQL SG uses a specialized join operator called *batch mode adaptive join* [43] that scans the first batch of join input and selects a join algorithm depending on input characteristics at runtime. However, it usually chooses hash join, which is not suitable for interactive queries, resulting in very slow performance. For most of IS queries, there was no significant difference in elapsed time, but for IS2, IS3, and IS7, SQL SG shows very slow performance due to using hash join.

### 6.2.3 *Comparison with the MGV-based GRDBMSs*

Despite being an MGV-based GRDBMS, DuckPGQ generates GT plans [70], processing topology operations using hash joins from its base system, DuckDB. Consequently, DuckPGQ exhibits slow performance across all IC queries and some IS queries. GrainDB demonstrates significant performance enhancements over the base system, DuckDB, due to its use of S-join instead of hash join. However, GrainDB's rule-based planning method, which substitutes hash join with S-join, does not efficiently utilize generated views in most queries. Therefore, we prepared an enhanced version of GrainDB (denoted as GrainDB-MO) by manually injecting an optimal S-join plan that fully leverages materialized views and conducted additional experiments with GrainDB-MO. The results show that it performs much closer to Chimera-TJ, albeit with some overhead from S-join compared to native graph traversal.

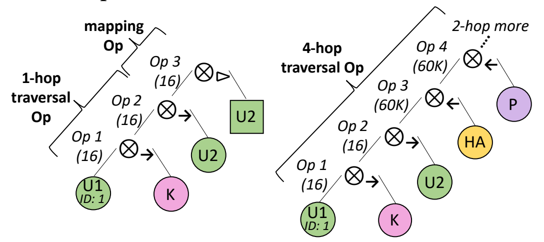
Meanwhile, as mentioned in Section 1, the MGV approach generates a materialized graph view in main memory at query processing time on the fly and thus fails to process queries for a large dataset like SF300. Table 7 shows the view generation time and size for SF100. In terms of time, DuckPGQ is faster than GrainDB because it can utilize multi-threads (32 threads) to generate views, but even so, both systems take a long time to generate views. In terms of size, DuckPGQ uses the array-based structure for views, while GrainDB uses the hashmap-based structure. But, both require a large amount of main memory and so result in out-of-memory during view generation for SF300. The above results show that the view generation is one of the major bottlenecks of the MGV approach, in particular, for interactive queries.

Table 7: View generation time and size (SF100).

| Systems | View generation | Queries | | | |
|---|---|---|---|---|---|
| | | IC2 | IC4 | IC8 | IC12 |
| GrainDB (using 1 thread) | time (msec) | 88,293 | 245,530 | 293,617 | 457,409 |
| | size (GByte) | 14.76 | 80.06 | 41.64 | 83.09 |
| DuckPGQ (using 32 threads) | time (msec) | 52,922 | 100,738 | 103,366 | 153,192 |
| | size (GByte) | 38.81 | 48.47 | 49.65 | 51.92 |

### 6.2.4 *Comparison with the RDBMSs*

Both DuckDB and Umbra are popular columnar DBMSs that use hash join as their main join operator [42]. DuckDB only uses hash joins, which makes it very slow in IC, even on IS queries, not just IC. Figure 8(a) shows the plan of Chimera-TJ for IS3, which processes Op 1 ((u1:User WHERE u.ID = 1)-[k:Knows]) in a single traversal. In contrast, DuckDB performs a hash lookup of the entire Knows table, resulting in significantly slower performance. Meanwhile, Umbra utilizes both hash join and INL join depending on a query. For IS3, it chooses an INL join with a single loop, resulting in much faster performance than DuckDB. However, when the number of loops in the INL join is large, as in the plan for IC12 in Figure 8(b), Umbra chooses hash joins for Op 3, resulting in significantly slower performance compared to Chimera-TJ, which performs native graph traversals for Op 3.



(a) Plan for IS3 (b) Plan for IC12

**Figure 8: Query plans of Chimera-TJ for IS3 and IC12.**

## 6.3 Comparison of throughput

Real workloads contain a mix of various queries such as IC, IS, and IU. Thus, we measure the throughput by running a *query mix* [1, 18] that executes eight read queries (IC2, IC4, IC8, IC12, IS1, IS3, IS4, IS5) and four write queries (IU2, IU3, IU5, IU8) with a query-specific *schedule*. We use the schedule defined by the LDBC SNB Interactive benchmark [1]. We call this query mix a *normal query mix*. In general, the throughput depends on two parameters [36]: (1) the number of threads (# connection) and (2) the workload intensity. LDBC SNB uses Time Compression Ratio (TCR) [1] to set the workload intensity, where a low TCR means a higher workload intensity. The normal query mix uses the default value that # connection = 1 and TCR = 1. We evaluate an additional query mix that # connection = 8 and TCR = 0.25 (denoted as *hard query mix*). We run each query mix following the guidelines in the LDBC Interactive benchmark [1] and evaluate throughput by measuring the number of queries (IC, IS, IU) executed per second (ops/sec). Since the MGV approach does not support transactional updates and thus cannot perform a query mix, we compare only the throughput of RDBMSs, the GT-based systems, and our Chimera. Table 8 shows the results for the query mixes.

*6.3.1* ***Result of normal query mix*** Chimera-TJ outperforms all the other systems by quickly responding to every query. The improvement is up to 779× compared to SQL SG. All systems except Chimera-TJ show the phenomenon of *delayed execution*, which means the current queries are not processed on schedule because the system is still processing the previous queries. In particular, SQL SG in Table 8 shows very low throughput due to its long response times in Table 5.

*6.3.2* ***Result of hard query mix*** Compared to the normal query mix, Umbra and SQL SG improve their throughputs due to the increased number of connections, but still show very low performance. In contrast, Agensgraph and Chimera-TJ show much higher throughputs. Chimera-TJ already achieved almost the maximum throughput for the normal query mix (about 54 ops/sec). Thus, when we increased only # connection = 8 while maintaining TCR = 1, its throughput was the same, i.e., 54 ops/sec (we omit the result). But, when we increase the workload intensity as TCR = 0.25, it improves the throughput by 4×, which is an ideal improvement. We note that DuckDB is marked as N/A because its LDBC SNB implementation does not support execution over multiple connections [63].

*6.3.3* ***Result of only IU queries*** We evaluate the throughputs of executing only IU queries, with no read queries, in the same normal and hard settings. As shown in Table 8, all systems compared process IU queries quickly, within 5 msec or less, without any delayed execution, thereby achieving the same ideal performance. Additionally, we measured their average update times in msec, where Chimera-TJ exhibits mid-level performance among the systems compared.

**Table 8: Throughput (ops/sec) for LDBC SNB query mix.**

| Query mix | Umbra | DuckDB | SQL SG | AG | Chimera-TJ |
|---|---|---|---|---|---|
| normal (mix) | 5.33 | 0.34 | 0.12 | 19.52 | **<u>54.03</u>** |
| hard (mix) | 32.04 | N/A | 0.29 | 169.64 | **<u>215.94</u>** |
| normal (IU only) | | | **<u>18.67</u>** | | |
| hard (IU only) | | | **<u>74.68</u>** | | |
| *avg update (msec)* | 3.4 | 1.0 | 4.1 | 1.2 | 2.5 |

We evaluate the detailed characteristics of query processing of three approaches: GT, MGV, and TJ, on top of Chimera for two types of synthetic queries: **Type-1) queries involving only topology operations (MICRO-1, MICRO-2)**, and **Type-2) queries involving both topology and property operations (MICRO-3, MICRO-4)**. In Type-1, we examine the performance penalty of GT due to join operations rather than native graph traversal, and in Type-2, we examine the performance penalty of MGV due to inefficient query plans. All the queries are evaluated on LDBC SNB SF100. Figure 9 shows the query template for the microbenchmark. For example, MICRO-1 finds Comments for Posts that satisfy a ID condition. In general, the performance is affected by the cardinality[2] [60] and the selectivity[3] [40]. In the query template, the cardinality is determined by ?*Label*, which is either *ReplyOf* (*n-1* relationship) or *Knows* (*n-n* relationship). The selectivity is determined by ?*topology_parameter* (denoted as $s_t$) and ?*property_parameter* (denoted as $s_p$). Chimera-GT-INL and Chimera-GT-Hash mean the results when the query optimizer chooses INL join and hash join, respectively.

```
MATCH (src)-[e:?Label]->(dst)
WHERE src.ID < ?topology_parameter
AND src.creationDate < ?property_parameter
COLUMNS (dst);
```

**Figure 9: GPML query template of microbench queries.**

In Figure 10, Chimera-TJ / MGV outperform Chimera-GT-INL for every $s_t$ selectivity. This result comes from that Chimera-TJ / MGV have O(1) and O($d$) for finding adjacent vertices and edges, respectively, but Chimera-GT-INL has O(log$|V|$ + 1) and O(log$|E|$ + $d$), where $d$ is the number of adjacent edges. In terms of cardinality, the performance gap in MICRO-1 (2.9× on average) is larger than in MICRO-2 (2.1× on average). It is because the *n-1* relationship of MICRO-1 makes $d$ smaller, leading to a larger gap between O(log$|E|$ + $d$) and O($d$). Chimera-GT-Hash shows poor performance consistently regardless of the selectivity because the costs of hash lookup for finding adjacent vertices and edges are O($|V|$) and O($|E|$), respectively [6].

[2]cardinality is the number of relationships vertices can participate in, like n-1 and n-n
[3]selectivity is the ratio of records that satisfy the condition out of entire records

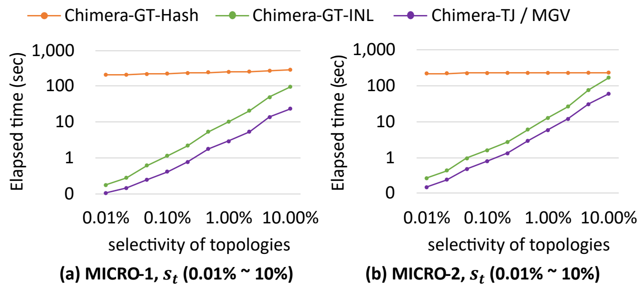


**Figure 10: Results of MICRO-1 (n-1) and 2 (n-n) ($s_p$ = 100%).**

In Figure 11, Chimera-TJ outperforms Chimera-MGV by orders of magnitude. As explained in Section 2.3, the MGV approach separates topology and property operations, and so topology operations are performed as the selectivity is 100%. As a result, Chimera-MGV shows a very poor performance regardless of selectivities. At the low selectivities under 0.2%, Chimera-GT-INL is slightly faster than Chimera-TJ about 0.1 seconds due to the overhead of the R2G mapping operation, but at the other selectivities, Chimera-TJ outperforms Chimera-GT-INL.

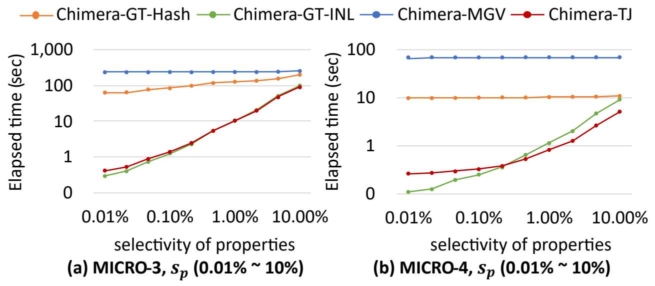


**Figure 11: Results of MICRO-3 (n-1) and 4 (n-n) ($s_t$ = 100%).**

## 6.4 Storage overhead of Chimera

Figure 12 shows the storage overhead of the GT approach (Chimera-GT), the MGV approach (DuckPGQ, GrainDB), and our approach (Chimera-TJ). GT requires two types of indexes: (1) indexes on ID column of vertex/edges tables; (2) indexes on src/dst columns of edge tables. TJ requires only the former type of indexes, resulting in lower storage overhead for indexes compared to GT. TJ does require additional space for the graph store, but the substantial performance improvements demonstrated in Sections 6.2 and 6.3 justify this investment. Furthermore, the graph store is disk-based, mitigating concerns about this overhead. On the contrary, the size of materialized graph view of MGV is comparable to that of the graph store and resides in memory which can cause an out-of-memory problem with a large dataset as shown in Figure 12(b). DuckPGQ incurs less space overhead than GrainDB due to its utilization of compression techniques [55] from the base system.

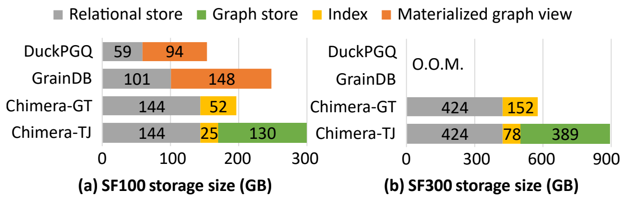


**Figure 12: Storage overhead of Chimera-TJ.**

## 7 RELATED WORK

Many studies have explored extending RDBMSs for property graphs.

- **Extraction methods:** GraphBuilder [32], Table2Graph [37], GraphGen [71], R2GSync [3], and Ringo [52] provide an interface to define relationships to extract a property graph from tables.
- **Managing methods:** SQLGraph [62], Grail [19], SQL Server graph extension [64], Agensgraph [10], and Oracle PGQL [50] store graphs as tables with a specialized schema design. Thus, they require data migration from the tables to the graphs. On the other hand, IBM db2 graph [68] and Cytosm [61] avoid such data migration by keeping the existing schema of tables and only storing additional metadata for graphs. GrainDB [33, 35], GRFusion [30], and DuckPGQ [70] stores graphs as read-optimized graph data structure such as CSR. They cannot process a query when the CSR generated does not fit in the main memory.
- **Querying methods:** SQLGraph [62], Grail [19], SQL Server graph extension [64], Agensgraph [10], Oracle PGQL [50], IBM db2 graph [68] and Cytosm [61] can process hybrid queries involving topology and property operations in the GT approach.

## 8 CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a novel method called Chimera for extending an RDBMS to a GRDBMS that efficiently supports interactive SQL/PGQ queries. It adopts a dual-store architecture that manages both graph topologies and relational properties as first-class citizens, enabling immediate update and efficient access without graph size limitations. It also has the unified query operator (i.e., TJ operator) that handles both topologies and properties to perform traversal, join, or mapping operations. We have implemented Chimera atop an open-source RDBMS with significant effort, resulting in Chimera outperforming the state-of-the-art GRDBMSs by up to 1,785× in terms of response time and by up to 779× in terms of throughput, all achieved without encountering out-of-memory issues in the LDBC SNB benchmark at SF300.

Native GDBMSs, while specifically optimized for graph traversal and offering dedicated query languages like Cypher [48], Gremlin [4], and GSQL [69], face challenges such as a steep learning curve, limited general-purpose use, and less mature ecosystems [53, 65]. RDBMSs, on the other hand, are widely adopted for their versatility, familiarity, and advanced features like security, concurrency, and indexing, but struggle with graph query performance [11, 33]. Therefore, it is crucial to balance between leveraging specialized capabilities and avoiding the pitfalls of reinventing the wheel when designing a GRDBMS. The current version of Chimera successfully improves the performance of graph queries while retaining many features of RDBMSs, such as security, concurrency, and indexing, thanks to its common layers. Nevertheless, aspects like fault tolerance and high availability may need to be reinvented within Chimera's graph store, which will be a focus of our future work.

## ACKNOWLEDGEMENT

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. RS-2024-00347471, RS-2023-00281635) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW).

## REFERENCES

[1] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, et al. 2020. The LDBC social network benchmark. *arXiv preprint arXiv:2001.02299* (2020).

[2] ANSI. 1992. ANSI X3.135-1992, American National Standard for Information Systems - Database Language - SQL. https://www.iso.org/standard/16663.html. Accessed: 2024-02-19.

[3] Nafisa Anzum and Semih Salihoglu. 2021. R2GSync and edge views: practical RDBMS to GDBMS synchronization. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–9.

[4] Apache. 2024. Tinkerpop gremlin. https://tinkerpop.apache.org/gremlin.html. Accessed: 2024-02-19.

[5] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[6] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2014. Main-memory hash joins on modern processor architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2014), 1754–1766.

[7] Andrey Balmin, Tom Eliaz, John Hornibrook, Lipyeow Lim, Guy M Lohman, David Simmen, Min Wang, and Chun Zhang. 2006. Cost-based optimization in DB2 XML. *IBM Systems Journal* 45, 2 (2006), 299–319.

[8] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[9] Bitnine. 2024. Bitnine AgensGraph. https://hub.docker.com/r/bitnine/agensgraph. Accessed: 2024-02-19.

[10] Inc Bitnine Global. 2024. Agensgraph. https://bitnine.net/agensgraph. Accessed: 2024-02-19.

[11] Yijian Cheng, Pengjie Ding, Tongtong Wang, Wei Lu, and Xiaoyong Du. 2019. Which category is better: benchmarking relational and graph database management systems. *Data Science and Engineering* 4 (2019), 309–322.

[12] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. *ACM SIGMOD Record* 16, 3 (1987), 323–330.

[13] CWI. 2024. DuckDB. https://github.com/duckdb/duckdb/releases/tag/v0.8.1. Accessed: 2024-02-19.

[14] CWI. 2024. DuckPGQ. https://github.com/cwida/duckpgq-extension/releases/tag/v0.8.1. Accessed: 2024-02-19.

[15] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[16] Bolin Ding and Arnd Christian König. 2011. Fast Set Intersection in Memory. *Proceedings of the VLDB Endowment* 4, 4 (2011).

[17] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. 2013. Sharding social networks. In *Proceedings of the sixth ACM international conference on Web search and data mining*. 223–232.

[18] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[19] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. 2015. The Case Against Specialized Graph Analytics Engines.. In *CIDR*.

[20] Apache Software Foundation. 2024. Apache AGE. https://github.com/apache/age. Accessed: 2024-02-19.

[21] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A pattern calculus for property graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 241–250.

[22] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL. In *The 26th International Conference on Database Theory, 2023*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1–1.

[23] Goetz Graefe. 1994. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

[24] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[25] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[26] PostgreSQL Global Development Group. 2024. JSONB type. https://www.postgresql.org/docs/current/datatype-json.html Accessed: 2024-02-19.

[27] PostgreSQL Global Development Group. 2024. PostgreSQL. https://www.postgresql.org/. Accessed: 2024-02-19.

[28] PostgreSQL Global Development Group. 2024. Resource Consumption. https://www.postgresql.org/docs/current/runtime-config-query.html Accessed: 2024-02-19.

[29] PostgreSQL Global Development Group. 2024. Resource Consumption. https://www.postgresql.org/docs/current/runtime-config-resource.html Accessed: 2024-02-19.

[30] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. 2018. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 1789–1792.

[31] ISO/IEC. 2023. ISO/IEC CD 9075-16.2: Information technology - Database languages SQL - Part 16: SQL Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html. Accessed: 2024-02-19.

[32] Nilesh Jain, Guangdeng Liao, and Theodore L Willke. 2013. Graphbuilder: scalable graph etl framework. In *First international workshop on graph data management experiences and systems*. 1–6.

[33] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GRainDB: A Relational-core Graph-Relational DBMS. In *12th Conference on Innovative Data Systems Research, CIDR*. 9–12.

[34] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2024. GrainDB. https://github.com/graindb/graindb/tree/master. Accessed: 2024-02-19.

[35] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. *Proc. VLDB Endow.* (2022).

[36] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. 2014. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1219–1230.

[37] Sangkeun Lee, Byung H Park, Seung-Hwan Lim, and Mallikarjun Shankar. 2015. Table2graph: A scalable graph construction from relational tables using map-reduce. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*. IEEE, 294–301.

[38] Jonathan Lewis and Thomas Kyte. 2006. *Cost-based Oracle fundamentals*. Springer.

[39] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. 2011. Content-based recommender systems: State of the art and trends. *Recommender systems handbook* (2011), 73–105.

[40] Clifford A Lynch. 1988. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values.. In *VLDB*. 240–251.

[41] Lothar F Mackert and Guy M Lohman. 1989. Index scans using a finite LRU buffer: A validated I/O model. *ACM Transactions on Database Systems (TODS)* 14, 3 (1989), 401–424.

[42] Amine Mhedhbi and Semih Salihoğlu. 2022. Modern techniques for querying graph-structured relations: foundations, system implementations, and open challenges. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3762–3765.

[43] Microsoft. 2024. Adaptive Joins. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing?view=sql-server-2017#batch-mode-adaptive-joins Accessed: 2024-02-19.

[44] Microsoft. 2024. SQL Server. https://www.microsoft.com/en-us/sql-server/sql-server-2019. Accessed: 2024-02-19.

[45] Microsoft. 2024. SQL Server. https://hub.docker.com/_/microsoft-mssql-server. Accessed: 2024-02-19.

[46] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*. 930–941.

[47] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[48] Opencypher. 2024. Cypher. https://opencypher.org. Accessed: 2024-02-19.

[49] Inc Oracle corporation. 2024. Oracle. https://www.oracle.com/database/. Accessed: 2024-02-19.

[50] Inc Oracle corporation. 2024. Oracle PGQL. https://oracle-base.com/articles/23c/sql-property-graphs-and-sql-pgq-23c. Accessed: 2024-02-19.

[51] Inc Oracle corporation. 2024. Oracle PGX. https://docs.oracle.com/en/database/oracle/property-graph/23.2/spgdg/executing-pgql-queries-graph-server-pgx.html. Accessed: 2024-02-19.

[52] Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. 2015. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1105–1110.

[53] Jaroslav Pokornỳ. 2015. Graph databases: their power and limitations. In *Computer Information Systems and Industrial Management: 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*. Springer, 58–69.

[54] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[55] Mark Raasveldt. 2024. Lightweight Compression in DuckDB. https://duckdb.org/2022/10/28/lightweight-compression.html. Accessed: 2024-02-19.

[56] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[57] Marko A Rodriguez and Peter Neubauer. 2012. The graph traversal pattern. In *Graph data management: Techniques and applications*. IGI global, 29–46.

[58] 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. *The VLDB journal* 29 (2020), 595–618.

[59] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[60] Il-Yeol Song, Mary Evans, and Eun K Park. 1995. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering* 3, 4 (1995), 427–459.

[61] Benjamin A Steer, Alhamza Alnaimi, Marco ABFG Lotz, Felix Cuadrado, Luis M Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–6.

[62] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1887–1901.

[63] Gábor Szárnyas. 2024. Run DuckDB queries in parallel. https://github.com/ldbc/ldbc_snb_interactive_v1_impls/pull/219. Accessed: 2024-02-19.

[64] SQL Server Team. 2017. Graph Data Processing with SQL Server 2017 and Azure SQL Database. https://cloudblogs.microsoft.com/sqlserver/2017/04/20/graph-data-processing-with-sql-server-2017/. Accessed: 2024-02-19.

[65] Yuanyuan Tian. 2023. The world of graph databases from an industry perspective. *ACM SIGMOD Record* 51, 4 (2023), 60–67.

[66] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From" think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[67] Yuanyuan Tian, Wen Sun, Sui Jun Tong, En Liang Xu, Mir Hamid Pirahesh, and Wei Zhao. 2019. Synergistic graph and sql analytics inside ibm db2. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1782–1785.

[68] 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 *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 345–359.

[69] TigerGraph. 2024. *GSQL Query Language*. TigerGraph Inc. https://docs.tigergraph.com/gsql-ref/current/querying/ Accessed: 2024-02-19.

[70] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4034–4037.

[71] Konstantinos Xirogiannopoulos and Amol Deshpande. 2017. Extracting and analyzing hidden graphs from relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 897–912.

[72] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* (2020).