Implementing Views for Property Graphs

Soonbo Han
University of Pennsylvania
soonbo@cis.upenn.edu

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

ABSTRACT

Property graph databases are increasingly used to integrate heterogeneous data, motivating *graph views* to abstract, simplify, and unify the data, e.g., to capture individual-level vs. organization-level relationships. This paper considers the tasks of *implementing* such views using rewriting techniques — both using existing property graph DBMSs and converting to relational RDBMSs. We consider both virtual and materialized views, ways of rewriting queries, and structures for indexing data. We also note a common use case of graph views, which involves preserving a graph except minor *local transformations*; we develop novel extensions and semantics for this. We evaluate and compare the performance of our techniques under a variety of workloads, and we compare existing graph and relational DBMS platforms.

1. INTRODUCTION

Property-graph databases [13, 6, 7] are increasingly popular for integrating data: they combine aspects of relations (nodes and edges can have sets of named properties) with those of graph data models (query languages can match complex patterns). As a result, a variety of applications, including social networks, the semantic web, provenance graphs, and knowledge graphs are using graph DBMS technology. Many of these applications involve data with significant structural complexity and multiple possible levels of abstraction.

Users of "graph data warehouses" might want to inspect or query data relationships, potentially at different levels of detail. For example, the data provenance is encoded as a PROV graph relating entities, actions, and agents [41], which might be desirable to view at different levels of detail. Knowledge graphs capture direct and transitive concept-subconcept and concept-instance relationships. Social networks capture connections among individuals, which can be abstracted to organization-organization connections. Such applications suggest the need to support multiple *views* over the graph data. Prior languages [29, 44, 23] are being unified in the GQL standard (`https://www.gqlstandards.org/`), which gives us an opportunity to target a single semantics for such views. G-CORE [6], from which GQL has been derived, enables developers to define views that integrate different property graph databases into a single output graph, and to do complex reasoning and aggregation.

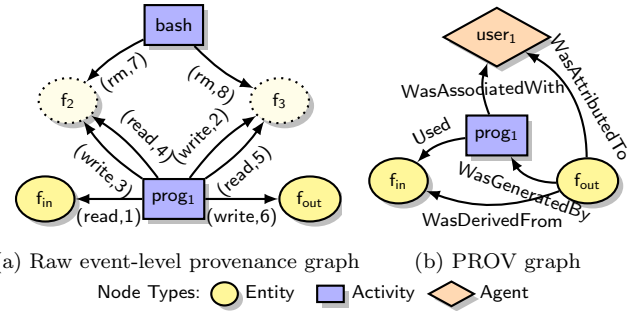
In an attempt to develop a foundation for graph views, we consider several questions: What are the underlying query reformulation [38] techniques that rewrite queries and respect the graph schema; How do we encode views and maintain them incrementally [31]; Which indexing strategies speed processing. We build upon well-understood techniques from the literature, in particular recursive Datalog and tuple-generating dependencies (TGDs), as a basis of doing graph query rewriting. To speed up execution in a relational setting, we also develop a novel *subgraph substitution relation* (SSR) indexing strategy that speeds querying over views, providing a key component to making RDBMS implementations competitive with dedicated property-graph DBMSs. Finally, we evaluate the strengths of modern property graph DBMSs, versus rewriting queries over relational DBMSs.

Furthermore, our experience with graph databases in the biomedical and social sciences suggests a common pattern in which a view performs a *set of local transformations* on the input graph (replacing a few subgraphs, preserving the rest of the graph). For example, in provenance tracking, one may want to "zoom out" [10, 11, 12] or "zoom in" in detail within a process — substituting a node for a subgraph, a subgraph for a node, etc. Similar patterns are observed in social networks. This type of usage is commonplace, but surprisingly inconvenient to express in proposed graph view languages.

EXAMPLE 1.1 . (UNIFYING PROVENANCE REPRESENTATIONS UNDER ONE MODEL) *Consider the graph in Figure 1(a), where the same program reads f_{in}, makes a series of updates to temporary files f_2 and f_3, and finally produces f_{out}. To convert to W3C's PROV [41], we may want to elide the steps involving temporary files, and add information about the user who initiates the activities. See Figure 1(b).*

Similar notions of *eliding and zooming out* on content apply equally to unified views of social networks (where, e.g., we want to convert different kinds of re-tweeting, liking, and responding to posts as a single "follows" relationship between posting user and responding user); to knowledge graph hierarchies; and even to UML and other diagrams.

As noted above, G-CORE supports views that bind to patterns over the input graph and selectively copy or construct nodes and edges into a target graph. This introduces numerous challenges, including formalization of views' underpinnings and semantics. Related concurrent work by Bonifati et al. [16] introduces *property graph transformations* using the *Graph Pattern Calculus* (GPC) [27] as an underlying



(a) Raw event-level provenance graph (b) PROV graph

Figure 1: An (a) event log-based provenance graph encodes I/O ($prog_1$ reads and writes files $f_{in}, f_2, f_3, f_{out}$); when (b) converting to PROV, we omit the temporary files, add the user, and add a variety of transitive edges.

formalism. Bonifati and collaborators develop solutions for combining multiple transformation rules, including transformation rules with recursive path expressions. They also study how to establish that the sets of rules are *consistent* with one another, leading to deterministic results; and their implementation converts sets of transformations into one unified query that can be executed in a graph DBMS.

Our work in this paper is primarily motivated by scenarios observed in collaborative projects that, like our above example, preserve the majority of the base graph *except for specific subgraph substitutions*. With path patterns in GQL, it can be extremely cumbersome to copy all nodes and edges, except for the ones to be elided. (We shall show details in Section 3). In fact, the above example is representative of a class of applications where collapsing or "zooming" has been proposed as a basic query operation [10] to hide proprietary details. These zooming steps can be applied to social or knowledge graphs, as well as provenance ones, either for visualization or reasoning. In this paper, we propose an additional (optional) construct for views, namely a *default rule* that copies all nodes and edges of an input graph *except* those that have been explicitly mapped or transformed. We also study how to implement both "traditional" views as proposed in G-CORE, as well as our extensions.

Our contributions are as follows:

- A means of converting graph views into relational formalisms, for which we leverage existing techniques for computing views and composing queries over them.
- Query language extensions to support the common case of *graph transformation* with subgraph substitution; and a notion of *well-behavedness* that preserves clean semantics.
- Strategies for materializing and updating views in property graph DBMSs; and for answering queries over both materialized and virtual views using relational DBMSs.
- Indexing techniques to speed the process of answering queries over graph views in relational DBMSs.
- Experimental results showing that our methods perform well, across a variety of graph transformation workloads, and on several different DBMS platforms.

2. FORMALIZING GRAPH VIEWS

The effort to standardize graph views [21, 6] and schemas [6, 7] is still ongoing within the Linked Data Benchmark Council and ISO working groups, but we incorporate many of the core ideas and language in our implementation. In order to develop an implementation, we map from (the core of) GQL and PG-Schema to an underlying set of relational constructs and constraints — for which an array of existing techniques from the query rewriting literature can be applied. In this section, we sketch the basics of views and constraints in GQL and PG-Schema, as well as how we represent these.

2.1 Core Components of GQL Views

Views in the G-CORE proposal [6] that forms much of GQL's foundation build on *conjunctive path patterns* over graphs, in the form of the Graph Pattern Matching Language (GPML) [21]. Influenced by the Cypher language [29], GPML specifies path expressions using an "ASCII art" method in which nodes and edges are specified within parentheses and square brackets, respectively, and the direction of edges is indicated by "arrows". For example, a pattern from Figure 1 might take the form `(prog1:Activity)-[:read]->(fin:Entity)`. Within parentheses or brackets, we use a *variable name*:*label* syntax.

Aligned with G-CORE, we extend GQL to define views using the following syntax: `CONSTRUCT` *nodes-and-edges* `MATCH` *GPML-patterns* `WHERE` *predicates* syntax — analogous to the SQL clause structure. Corresponding to SQL `SELECT`, the `CONSTRUCT` clause can *output into a new graph* nodes, edges, and paths copied from bound variables from the rest of the query. Essentially, the nodes and edges in the `CONSTRUCT` clause are copied in an identity-preserving way to the graph; and as specified new nodes and edges are constructed through computed expressions. The `MATCH` clause takes the place of SQL's `FROM`, but allows for specification of regular path expressions using GPML path-patterns. The `WHERE` clause, as in other languages, allows predicates over bound variables and over paths. We illustrate more via simple examples below.

EXAMPLE 2.1. *Revisiting our provenance integration scenario, a view definition in GQL, posed over the schema of Figure 1(a) might look like this:*

```
GRAPH VIEW Prov AS (
    CONSTRUCT (f1)<-[u:Used]-(p) SET u.ts = 3
      MATCH (f1:Entity)<-[:read]-(p:Activity)
        WHERE f1.name='fin'
  UNION
    CONSTRUCT (p)<-[g:WasGeneratedBy]-(f2)
                  -[:WasAttributedTo]->(u:User)
                <-[:WasAssociatedWith]-(p),
              (f1)<-[d:WasDerivedFrom]-(f2)
                        SET u.id=1, u.name='user1'
      MATCH (f1:Entity)<-[:read]-(p:Activity)
                        -[:write]->(f2:Entity)
        WHERE f1.name='fin' and f2.name='fout'
)
```

The view creates a new graph where the first rule copies an `Entity` *node whose* `name` *property has the value 'fin',* $f1$*, from the input when* $f1$ *is connected by a* `read` *edge to an* `Activity` *bound to variable* p*. A new edge,* u*, is constructed in the view, and its 'ts' (timestamp) property is set to a literal value. The second rule in the union functions similarly. The output graph of the* `Prov` *view is shown in Figure 1(b).*

Each `CONSTRUCT-MATCH-WHERE` expression returns a set of nodes and edges; multiple graph expressions can be `UNION`ed together into a (de-duplicated) set of nodes and edges forming an output graph. In this paper, we follow the GPML specification [21], where each GQL `CONSTRUCT-MATCH-WHERE`

expression in a union has only conjunctive path steps (without negation). Thus a `CONSTRUCT` clause represents a conjunctive path query, extended with inequality/unequality between variable bindings and literals; and a view represents the union of such queries, forming a single output graph. We provide a mapping of these views to well-understood formalisms in Section 2.3.

2.2 Graph and Schema Model

We formalize the core elements of the draft GQL and PG-Schema efforts [8, 7] in a relational encoding. We capture the basic components of the property graph in a relational representation, which ultimately can be mapped to actual relations (in a relational DBMS implementation) or to the components of a property graph (in a graph DBMS implementation).

2.2.1 Property Graph Relations

We model the graph's nodes and edges using relations in a schema Σ, to which we will add various constraints to establish graph schemas.

$N(n, label)$	Labeled nodes, by ID
$E(e, n_1, n_2, label)$	Labeled edges, by ID
$NP(n, key, value)$	Node key-value properties
$EP(e, key, value)$	Edge key-value properties

where both nodes and edges may have unique IDs. Our representation can support multiple nodes with the same label as well as multi-edges between pairs of nodes.

2.2.2 Graph Schemas and Constraints

The PG-Schema [8, 7] proposal develops a set of types for nodes and edges, and establishes constraints based on these. We implement a subset of this specification: we focus on *closed types*, in which content must explicitly allowed by the type; and we limit each node or edge label to a single type. To capture types, we define several auxiliary relations:

$NT(label, nt)$	Mapping: node label to node type
$ET(label, et)$	Mapping: edge label to edge type
$NPROPS(nt, key, type)$	Labels, types for node properties
$EPROPS(et, key, type)$	Labels, types for edge properties

For a graph schema corresponding to the PROV view in Figure 1(b), we encode that **agent** and **activity** nodes have associated types:

$NT(\text{“}Agent\text{”}, agentType)$, $NT(\text{“}Activity\text{”}, activityType)$.

Additionally, we need to establish a set of constraints over the node and edge-related relations.

Keys. PG-Schema `IDENTIFIER` constraints establish that each node of a given type has an “atomic” value for the “key” property (in relational terms: 1NF and non-null), and that each value occurs uniquely among the set of all nodes of this type. We can codify this using *equality-generating dependencies* (EGDs) and *tuple-generating dependencies* (TGDs) as per prior work on graph constraints [25].

Structural constraints and foreign keys. PG-Schema also establishes constraints on which edges connect which nodes: for example, we can define *associatedType* that connects an activity to an agent, which can be expressed using a TGD.

2.3 Views as Tuple-Generating Dependencies

Suppose the DB administrator has defined a set of GQL views over a graph. To understand how to efficiently use this view (and answer queries over it), our goal is to map GQL to a more “standard” construct that is better understood from a database theory perspective, such that we can leverage prior (relational) techniques to reason about compositions of queries over views.

Unlike a standard relational view, a graph DBMS view outputs a combination of multiple kinds of entities (nodes, edges, properties) that are naturally represented in multiple relations. As a formalism that allows us to specify constraints over multiple relations at a time within a single expression, we adopt *tuple-generating dependencies* (TGDs) [9] to formalize each conjunctive GQL expression as a (partial) mapping from the input graph to output graph. TGDs are commonly used in data integration and exchange to map from one schema to another. A TGD over some schema Σ is of the form:

$$\forall \bar{X}, \bar{Y} (\phi(\bar{X}, \bar{Y}) \rightarrow \exists \bar{Z} (\psi(\bar{X}, \bar{Z})))$$

where ϕ (resp. ψ) is a conjunction of atoms in Σ (resp. Σ') over variables $\bar{X}, \bar{Y}$ (resp. $\bar{X}, \bar{Z}$). We will omit the universal quantifiers of TGDs for brevity. We do not allow existential variables in the right-hand side of TGDs, but rather require the use of Skolem functions to create new values or IDs. As a result, applying the chase [22] to such a TGD computes a standard DBMS instance as opposed to an incomplete instance [4].

Intuitively, our system will automatically convert GQL views into TGDs. From this, applying the chase procedure over the TGD allows us to compute a conjunctive expression over a set of input relations (the graph nodes, edges, properties, etc.) and output results into a set of output relations (i.e., we can materialize the view output's nodes, edges, properties, etc.). Multiple TGDs can be chased together, and their unioned results will form the **final contents of the graph view**. Suppose we designate the view's output nodes and edges as the N', E', NP', and EP' relations in the view's relational schema Σ'.

Example 2.2. *Continuing our running example, the first rule of the view definition in Example 2.1 can be translated into the TGD:*

$$\begin{aligned} & N(f_1, \text{“}Entity\text{”}) \wedge N(p, \text{“}Activity\text{”}) \\ \wedge & E(r, p, f_1, \text{“}read\text{”}) \wedge NP(f_1, \text{“}name\text{”}, \text{“}fin\text{”}) \\ \rightarrow & \exists u (N'(f_1, \text{“}Entity\text{”}) \wedge N'(p, \text{“}Activity\text{”}) \\ & \wedge E'(u, p, f_1, \text{“}Used\text{”}) \wedge EP'(u, \text{“}ts\text{”}, 3)). \end{aligned} \tag{t_1}$$

Relating our formalism to that of Bonifati et al. [16], the TGD above is equivalent to the following property graph transformation [16] using GPC-like syntax:

$$\begin{aligned} & (\texttt{p} : \texttt{Activity}) \xrightarrow[\langle \texttt{f1.name='fin'} \rangle]{\texttt{:read}} (\texttt{f1} : \texttt{Entity}) \\ & \Rightarrow (\texttt{p} : \texttt{Activity}) \xrightarrow[\langle \texttt{u.ts=3} \rangle]{\texttt{u:Used}} (\texttt{f1} : \texttt{Entity}) \end{aligned} \tag{t_2}$$

Each atom on the LHS (resp. RHS) in the TGD (t_1) corresponds to a node, an edge pattern (resp. constructor), or a property on the left (resp. right) of $\Rightarrow$ in the transformation rule (t_2). Note that our underlying mapping to TGDs has the benefit of allowing us to harness known techniques for composing TGDs with one another and with other constraints, as well as query reformulation techniques such as the chase [22, 24, 30]. However, support for full regular path expressions remains future work using our TGD-based methodology.

As in the above example, the semantics of a construction rule can be directly expressed as a TGD.

The data integration and exchange literature [28, 30] has shown that (with the addition of *Skolem functions* to create shared IDs for any new entities or values introduced in the output) we can take a TGD and break it into a set of Datalog rules (one for each atom in the right-hand side). These rules can be used to create materialized output relations for the view, equivalent to applying the chase over the TGD(s). Conversely, queries over the output view can be unfolded with these Datalog rules to produce queries over the input graph.

3. TRANSFORMATION VIEWS

For many kinds of complex graphs, views are an important intermediate step for ensuring privacy of certain subgraph structures (e.g., "zooming out" in provenance graphs [10]), or of ensuring the content is at the right level of detail for browsing or presentation (e.g., excerpting and summarizing information within huge low-level graphs, or alternatively adding detail to sparse high-level graphs). For these kinds of *graph transformations*, the view often preserves the elements of the input graph, *except* for certain subgraphs that are specifically to be replaced (with other subgraphs, with nodes, or with nothing). In the G-CORE proposal, a view *must explicitly* copy all elements from the input to the output, disallowing cases that match the components to be replaced — often making the definition cumbersome and redundant.

We argue that this is not intuitive, and the query is difficult to extend — even though the desired subgraph transformations can each be specified compactly! A more natural semantics would define a view first by matching subgraphs, and constructing corresponding output subgraphs; then adding a form of *negation*. Namely, we could union the outputs with (identity-preserving copies of) *all nodes and edges that were not mapped by subqueries of the view*.

To allow this, we extend our view formalism, to be computed in two steps or strata [4], as follows. In the first stratum, TGDs are chased and their results are unioned together to form the output graph. A *mapping table* is recorded for all nodes and edges that were bound by these subqueries, and were thus mapped to outputs. In a subsequent stratum, a *default rule* is applied which copies *all nodes* that have not been mapped, and simultaneously maps *all edges* that were not already mapped (whose node endpoints were previously mapped to the output graph, either directly in the mapping table, or by the default rule). To allow for certain nodes and edges to explicitly not be mapped, we add a `DELETE` clause that marks them to not be copied.

The default rule contains negation, i.e., it adds all input graph nodes and edges, *except* if they have participated in a mapping in one of the components of the view. Beyond the simple `DELETE` with a default rule, we also allow `CONSTRUCT` clauses to explicitly `MAP` input nodes to corresponding output nodes, assigning identity based on *Skolem functions* [4] applied to the inputs. Multiple nodes and edges in an input subgraph can be mapped to a single output node or edge (as determined by the identity of the output node/edge); multiple bound subgraphs can be mapped to the same node or edge; and new nodes or edges can be created as needed. However, introducing `MAP` and `DELETE` operations can result in nondeterministic interactions between clauses within a transformation view, e.g., if we simultaneously delete and map a given input node. We will discuss this issue in Section 3.2.

The `SET` clause allows us to set the identities of new nodes or edges with a Skolem function. `SkolemFunc` takes a Skolem function name as its first argument; then bindings from the `MATCH` clause as additional arguments. The function guarantees a unique ID for each combination of Skolem name and input values (and values will not collide with IDs created in any other way). Multiple elements from the input graph can be mapped to the same element on the output graph (merging or "collapsing" them); multiple rules can use the same Skolem values to create shared (merged) node or edge identities, allowing us to create complex relationships in nodes and edges mapping.

3.1 Capturing Mappings and Adding Defaults

For simplicity, we refer to views that preserve most of the input graph with a default rule as *transformation views* since they apply a small number of subgraph transformations to the input graph. Recall from the previous section that we convert each `CONSTRUCT` (and, in our case, `DELETE`) clause from a view into a TGD. To capture the mappings performed within a TGD, we add an atom to the right hand side of the TGD for each input binding-to-output node pair (n, n') or edge pair (e, e'), essentially outputting that pair into a node mapping and edge mapping table:

$N^M(n, n')$	Node to node ID mapping ($n' = \perp$ if n is deleted)
$E^M(e, e')$	Edge to edge ID mapping ($e' = \perp$ if e is deleted)

where the node and edge remappings are encoded. Every input node n or edge e in N^M or E^M is paired with a new id(s) or $\perp$. As we describe later, for semantic consistency we allow for a many-to-many association between input and output IDs, but a node or edge should not be simultaneously mapped to a new id and $\perp$.

Default Rules. Any source node that is not explicitly mapped or removed by a transformation rule in a view is instead matched by a *default transformation rule* (t_4) below that copies the node to the output, maintaining its ID. To fully capture node to node ID mappings from input graph to output graph, we also use the following relations in the default rules:

$N^{DM}(n, n')$	Node:node ID mapping by default rules
$E^{DM}(e, e')$	Edge:edge ID mapping by default rules

We establish this through an identity mapping for all nodes not already mapped, and all such non-deleted node IDs are copied to the output:

$$N^M(n, n') \wedge n' \neq \perp \rightarrow N^{DM}(n, n') \qquad (t_3)$$

$$N(n, l) \wedge \neg N^M(n, n') \rightarrow N^{DM}(n, n) \wedge N'(n, l) \qquad (t_4)$$

Similarly, edges are copied by default via (t_6) below, so long as their endpoints are mapped to the output graph, they are not already mapped, and they are not explicitly marked for deletion. This will maintain referential integrity.

$$E^M(e, e') \wedge e' \neq \perp \rightarrow E^{DM}(e, e') \qquad (t_5)$$

$$E(e, s, d, l) \wedge \neg E^M(e, \perp) \wedge N^{DM}(s, s') \wedge N^{DM}(d, d') \rightarrow E'(e, s', d', l) \wedge E^{DM}(e, e) \qquad (t_6)$$

Node and edge properties copied by default mappings in a similar fashion; we omit details for brevity.

Putting It All Together. Given a view definition in our

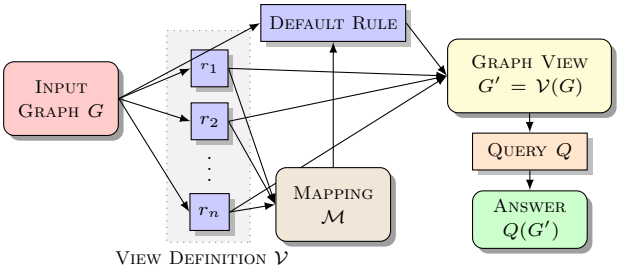


Figure 2: An overall diagram of defining a graph transformation view by a set of transformation rules $(r_1, \ldots, r_n)$ and the default rule along with a mapping relation over an input graph and querying on the view.

extended GQL language, we create a set of TGDs whose right-hand sides define the graph corresponding to the view: nodes N', edges E', as well as node-properties NP and edge properties EP. In addition, for transformation views, we create auxiliary relations N^M, E^M, and so on, representing the input graph elements that were mapped; then use the input graph minus those elements to produce the default mappings N^{DM}, E^{DM}, and so on. These latter derived relations include negation and must be computed in a second stratum, after the initial rules were run. Finally their output is unioned with N', E', and so on; forming N'', E'', and so on — which contain our final graph view. As we describe in detail in Section 4, we can either make these virtual views (unfolding queries over them), or we can materialize them directly.

3.2 Well-Behaved Views

Given that a graph view is computed as the union of multiple rules' transformations, and each rule may map input elements to output elements (possibly merging them) or may delete input elements — there is a natural question of when the rules are semantically inconsistent or ambiguous.

As illustrated schematically in Figure 2, a *graph transformation view* G' is the result of evaluating by a view definition (comprised of rules whose outputs are unioned together) over an input graph G (which may be the the output of another view). Let us denote the *mapping set* of transformation rules as $r_i \in \mathcal{V}$, including the default mapping rule. The set of mappings in the rule should produce *deterministic output* consistent with user expectations, regardless of execution order (or parallelism) among rule execution; and the output of the default rule, computed in a second stratum, should also be deterministic.

Definition 3.1 . (Well-behaved view) *A graph view $\mathcal{V}$ is said to be* well-behaved *under graph schema Σ if it satisfies two properties.*

1. *Any mappings derived by $\mathcal{V}$ applied to a graph $G \models \Sigma$ cannot perform* conflicting mappings *on the same source node or edge.*
2. *The mappings do not result in an output graph that violates schema constraints, including uniqueness of labels and keys, and referential integrity.*

We considered several notions of conflicting mappings, in expressing various types of transformations in provenance and other domains. The most useful definition is to preclude simultaneous *deletion* of a graph element while also *mapping it* to an output graph element. We use this definition below.

Analogously to typechecking in a programming language, it is often desirable to know when a set of transformation rules *will not interact with regards to their mapped inputs* and thus are guaranteed to not violate the first property in Definition 3.1. We thus establish a static typechecking algorithm as a first step. We then consider cases where we may want to check at runtime if two mappings that interact may *still do so in a compatible way* and whether the output conforms to a schema.

Non-conflicting rules. A view definition consists of several transformation rules, with path patterns that can partially overlap. A key challenge is to efficiently confirm that two rules do not *conflict* with each other, i.e., that the output of one rule (perhaps a deletion) is semantically compatible with every other rule. Generally, a precisely defined schema, along with well-specified rules, should guarantee such non-conflicting rules.

3.2.1 Static Checking of Conflicting Rules

Our static compatibility checking algorithm takes a graph schema Σ and a set of transformation rules whose results are unioned together in $\mathcal{V}$ as input, and it determines whether $\mathcal{V}$ can produce conflicting mappings for the same source entities, for any graph $G \models \Sigma$.

We model this test as an SMT (satisfiability modulo theory) problem by declaring a graph G consisting of the sets of node and edge predicates for the two (possibly overlapping) subgraphs matching the `MATCH-WHERE` expressions of the two rules being checked, respectively; as well as the set of schema constraints. We color in red each node and edge that appears in the `CONSTRUCT` clause (for *mapping itself*, or *copying*), and each source node in the `MAP` clause (for *mapping*), as well as any edges connected to them. We color in blue the node, or the edge and its endpoints, included the `DELETE` clause (for *deletion*). The SMT formula will be *satisfiable if and only if* there exists a graph G conforming to Σ with colors as specified above, where a node or edge receives more than one color (i.e., both red and blue). In turn, the algorithm accepts *if and only if* the SMT formula is *unsatisfiable*, meaning there is no graph under which any mappings of the nodes or edges conflict. Our algorithm is both sound and complete (proof omitted due to the space limit).

Example 3.1. *Let us check if the first rule of the view in Example 2.1 and the following one:*

```
DELETE f2
  MATCH (f2:Entity)<-[:rm]-(p2:Activity)
```

are conflicting rules under different constraints.

1. *Suppose there are no constraints ($\Sigma = \phi$). We can imagine a graph G where an* `Entity` *node has two adjacent Activity nodes, connected by the* `rm` *and* `read` *edges, respectively. Then, the variables $f1$ and $f2$ are bound to the* `Entity` *node by the two matching patterns, and the node is colored red and blue, meaning it is simultaneously copied and deleted. The rules conflict each other.*
2. *Consider a constraint that every node can have at most one incoming or outgoing edge (easily expressed in EGDs). Then no such graph G exists, and the rules do not conflict.*

To resolve a conflict between rules, the developer can choose an order of application, by composing a view based on one rule over another view based on the other rule.

3.2.2 Runtime Checking of Schema Compliance

We use the DBMS' built-in capabilities to check the view output against the output schema at runtime, through a combination of key-foreign key constraints mapped from schema constraints enforced in the underlying DBMS; and queries to find violations to the EGDs and TGDs representing the schema.

4. IMPLEMENTING GRAPH VIEWS

We can now consider strategies for mapping from TGDs to executable queries and views.

4.1 Graph Views with Existing Graph DBMSs

We first target a property graph DBMS such as Neo4j to derive a view's output graph with Cypher. Currently available graph DBMSs do not support virtual views, so queries over the output graph have to be answered by using the materialized output graph.

GQL view to Cypher materialized instance. We rewrite views directly to Cypher to compute G' alongside G. Thus we consider two approaches to capture graph view output in Cypher:

1. (Update In-place) One naive approach is to use Cypher's CRUD operations and the APOC library to transform the input graph. However, this approach updates the input graph in-place, so it needs to be loaded again if we want to define other views.
2. (Overlay) Another approach is to have an augmented graph with additional properties on every node and edge, similar to versioning [17]. We assign a *stratum number* to each view, increased by 1 from the highest stratum of its inputs. (For a base input, the stratum number is 0.) A view at stratum number k consists of nodes and edges *created* at or before k, *deleted* after k. Views over views can be emulated by using properties *created* and *deleted*.

Querying Cypher "materialized views." In Neo4j, when a user query references a specific materialized output graph, we add predicates over the *gid*, *created*, and *deleted* properties for each variable appearing in the user query (as well as each intermediate step in a path). Direct translation from a GQL query to Cypher is straightforward, and allows queries to be posed over stored output graphs. However, it has some limitations:

- View maintenance is expensive. In the *update in-place* approach, updates to the base graph may not be feasible as the base graph is no longer available. In the *overlay* approach, incremental view maintenance is expensive as it consists of multiple queries to update portions of the graph.
- Query optimization is limited. In general, we cannot integrate extra optimization techniques, other than leveraging property graph indices.

Directly using virtual views is not feasible in any Cypher-based graph DBMS we are aware of: key building blocks such as query unfolding are not supported. Thus we instead consider a relational DBMS implementation, which we will evaluate against Neo4J.

4.2 Graph Views with Relational Languages

We consider both native-Datalog and native-SQL relational database platforms. For both cases we leverage Datalog as an intermediate representation. Our transformation rules are based on conjunctive path queries. As described in Section 2.3, each rule $r \in \mathcal{V}$ corresponds to a TGD. In turn, each TGD can be mapped to a set of Datalog rules; and the Datalog program defines the target view instance following the approach outlined in [30], corresponding to executing the *chase* [24] over the TGDs (which are all full TGDs with Skolems). Not only does this allow us to define a clean semantics for computing the instance, but query containment and rewriting are well understood with TGDs [18]. From the TGD-to-Datalog translation, we have a means of specifying N', E', NP', EP' as views.

Direct execution in Datalog DBMS (e.g., LogicBlox). For each TGD r of the form $\phi(\bar{X}, \bar{Y}) \wedge Skolems(\bar{Z}) \rightarrow \psi(\bar{X}, \bar{Z})$ in our view definition, we define a separate Datalog rule:

$$\psi_i(\bar{X}_i, \bar{Z}'_i) \text{ :- } \phi(\bar{X}, \bar{Y}) \wedge Skolems(\bar{Z})$$

where $\psi_i(\bar{X}_i, \bar{Z}'_i)$ is the ith atom. In this fashion, the set of TGDs derived from a view definition can be easily converted to a set of Datalog rules that define a set of IDBs comprising the nodes, edges, their properties, deltas, and mapping relation, as defined in Section 2.

Datalog to SQL. We leverage standard techniques to convert each Datalog rule to a corresponding SQL statement; and to define SQL views as the union of all translated rules with the same atom name. Skolem functions are implemented by user defined functions.

4.3 Answering Queries over Views

We describe how to evaluate queries not only over materialized view instances, but also to be able to treat the view as *virtual* and to use *view unfolding* to answer our queries.

Querying Datalog Program. Each `MATCH` clause is directly rewritten to the body of a conjunctive Datalog rule, except that we make use of the N', E', NP', and EP' relations as the components of the graph (instead of N, E, etc.). We further add conjunctive constraints based on the `WHERE` clause. Subsequently, we take the various components of the `CONSTRUCT` clause and generate separate Datalog rules to output a new set of edges (relation E_o), nodes (N_o), edge properties (EP_o), and node properties (NP_o) that form a graph as answer.

Querying SQL views. We can directly answer the SQL query, translated from the Datalog program above using standard techniques, over the defined SQL views using the underlying DBMS.

5. ACCELERATING QUERIES OVER VIEWS

Graph views start by finding bindings to subgraphs of the input graph, as matched by a `MATCH` pattern with any `WHERE` constraints. In a relational implementation, this matching is join-intensive. Materializing this matched result speeds up query answering.

Storing Relations for Subgraph Matching. An *access support relation* [36] is an indexing method, first proposed for object-oriented databases, that stores matches to path expressions over graph-structured data, as tuples of bindings to nodes and edges. It generally uses tuple IDs but can equivalently be thought of as encoding tuple keys. A transformation rule r of the form $\phi(\bar{X}) \rightarrow \exists \bar{Y}(\psi(\bar{X}, \bar{Y}))$ can be decomposed into

two TGDs. One is for generating an ASR:

$$\phi(\bar{X}) \rightarrow \mathbf{asr}(\bar{X}) \qquad (t_7)$$

and the other is for defining transformation over the ASR:

$$\mathbf{asr}(\bar{X}) \rightarrow \exists \bar{Y}(\psi(\bar{X}, \bar{Y})) \qquad (t_8)$$

When answering queries using ASRs, inverse rules are still employed, but in conjunction with the TGD that defines transformation over the ASRs (t_8).

5.1 Subgraph Substitution Relations (SSRs)

An ASR provides a useful way of storing bindings to input subgraphs for a given transformation rule — but it is lacking in information about the *output* subgraph produced in response which the user's query is posed over. To address this gap, we introduce the notion of the *subgraph substitution relation* (SSR). An SSR encodes local transformations made by a transformation rule on an input graph. The local transformations can be formalized as *subgraph substitution mapping* either (1) directly translated to the TGD (e.g., (t_1)) of a construction rule (with `CONSTRUCT`) in Section 2 or (2) derived from by applying the standard chase algorithm [4] to the TGD of a transformation rule (with `DELETE`) with the default rule in Section 3.

The conjunction of the atoms over the input graph (i.e., N and E) in the LHS is called *input pattern*, and that over the output graph (i.e., N' and E') in the RHS is called *output pattern*.

Creating SSRs. An SSR contains a set of subgraphs of the output graph G'. More specifically, it includes all portions originated from input subgraphs and affected (or *transformed*) by graph transformations. For the subgraph substitution mapping of the form $\phi(\bar{X}, \bar{Y}) \rightarrow \exists \bar{Z}(\psi(\bar{X}, \bar{Z}))$, the TGD for generating an SSR is of the form:

$$\phi(\bar{X}, \bar{Y}) \rightarrow \exists \bar{Z}(\mathbf{ssr}(\bar{X}, \bar{Y}, \bar{Z}))$$

Example 5.1 . (SSR) *The TGD (t_1) representing a local transformation from input pattern to output pattern can be decomposed into two TGDs:*

$$N(f_1, \text{“Entity”}) \wedge N(p, \text{“Activity”}) \wedge E(r, p, f_1, \text{“read”}) \wedge NP(f_1, \text{“name”}, \text{“fin”}) \rightarrow \exists u(\mathbf{ssr}_2(f_1, u, p)), \qquad (t_9)$$

$$\mathbf{ssr}_2(f_1, u, p) \rightarrow N'(f_1, \text{“Entity”}) \wedge N'(p, \text{“Activity”}) \wedge E'(u, p, f_1, \text{“Used”}) \wedge EP'(u, \text{“ts”}, 3). \qquad (t_{10})$$

The TGD (t_9) in the example above shows how an SSR is defined.

Using SSRs in Query Answering. The TGD (t_{10}) implies that for each tuple in an SSR, there is a subgraph matched by the output pattern in the output graph. The converse is not true in general. If the converse holds, we have the output pattern's complete view over the output graph that can be used in query rewriting.

5.2 Query Rewriting with SSRs

We present how we use SSRs in query rewriting as a variation of *optimizing queries using views* [19, 34]. We first populate a set of equivalent rewriting rules from the output pattern and the definition of an SSR derived from each rule, with covered query test defined below. Each rewriting rule is of the form:

$$\psi(\bar{X}) \leftrightarrow \mathbf{ssr}(\bar{X}, \bar{Y})$$

where ψ is a conjunctive query over G', and $\bar{Y}$ are variables not appearing in ψ.

Given a query Q over a view's output graph G', the subquery q of Q that matches the body (LHS) of a valid SSR rewriting rule is replaced by the head (RHS) of the SSR with appropriate attributes, using standard techniques from inverse rules [3]. This requires us to determine whether an SSR is sufficient to answer a path-pattern of the query comprehensively.

Whether an SSR contains all matches of some patterns in the output graph G' can be determined by the *covered query test* below with the transformation rule r corresponding to the SSR, and graph schema Σ of the input graph G.

Covered Query Test. A query Q passes the test *if and only if* there exists no $G \models \Sigma$ such that some matches of Q in G' are not stored in the SSR.

This is a *query containment problem*: we must test if Q_1 is contained in Q_2 under a set Σ of EGDs that can express integrity constraints of base graph G. In our setting, Q_1 and Q_2 are unions of conjunctive queries with negations (UCQ$^\neg$) in the presence of node or edge deletions. Following the standard query containment testing of UCQ$^\neg$ [39, 5], we construct canonical databases from Q_1 and apply the standard chase algorithm with Σ [4]. Note that this is not a performance bottleneck since the chase algorithm is performed only over a small number of canonical databases in practice and once during the SSR creation.

6. EXPERIMENTS

In this section, we demonstrate an experimental evaluation of the most novel component of our work: end-to-end handling of transformation views. Additional experiments and workload details are available in [35] and online [1].

To compare the efficacy of our techniques for answering queries over graph views, we developed a diverse workload including four real-world graphs: a provenance graph (`PROV`), social network (`SOC`), academic citation graph (`OAG`), and a knowledge graph (`WORD`). We study the performance of transformation views in the following aspects:

- the effectiveness of materializing views and answering queries over these views.
- the performance improvements achieved by using `SSR`s in query execution.

Our graph view system was implemented as a Java middleware layer, running on the top of off-the-shelf database systems. We used OpenJDK 11.0.16 and Linux Ubuntu 22.04.1 LTS on Amazon EC2 (m5a.xlarge), with AMD EPYC 7571 CPUs at 2.2GHz and 16GB RAM. As the underlying database systems, we chose LogicBlox 4.41.0 (native-Datalog RDBMS), PostgreSQL 14.5 (SQL DBMS), and Neo4j 4.1.4 Community Edition (property graph DBMS).

We evaluate the performance of both view creation and query execution over transformation views, where subgraphs may be collapsed or merged, and where default rules are applied. In LogicBlox and PostgreSQL, the default rule copies non-transformed nodes and edges. We contrast this with the performance in Neo4j, where such nodes and edges remain unchanged. The results are shown in Table 1 and 2.

Materializing Views and `SSR`s. View materialization times (**MV** column) are influenced by several factors: the size of the base graph, the complexity of the graph pattern in the view definition, and the proportion of matches within the

View	Materialization							
	PROV$_{V_3}$		OAG$_{V_4}$		SOC$_{V_5}$		WORD$_{V_6}$	
Impl.	MV	SSR	MV	SSR	MV	SSR	MV	SSR
LB	32.22	18.71	350.36	124.92	24.87	**18.41**	**10.71**	**4.20**
PG	77.55	**11.27**	413.05	**69.79**	60.99	49.88	170.51	159.63
N4-UP	**3.46**	-	**5.93**	-	**14.94**	-	15.94	-
N4-OV	3.74	-	6.52	-	19.42	-	154.72	-

Table 1: Times for creating views with default rules (in sec).

View	Impl.	Query execution					
		Q_3(MV)	Q_3(SSR)	Q_4(MV)	Q_4(SSR)	Q_5(MV)	Q_5(SSR)
PROV	LB	0.17	**0.12**	**0.12**	**0.12**	**0.11**	**0.09**
V_3	PG	0.43	0.42	0.42	0.42	0.44	0.42
	N4-UP	0.09	-	1.23	-	1.24	-
	N4-OV	**0.06**	-	2.54	-	2.49	-
		Q_7(MV)	Q_7(SSR)	Q_8(MV)	Q_8(SSR)	Q_9(MV)	Q_9(SSR)
OAG	LB	**0.12**	**0.11**	**0.41**	**0.11**	0.10	**0.09**
V_4	PG	0.99	0.61	0.91	0.62	0.59	0.59
	N4-UP	17.00	-	19.95	-	**0.06**	-
	N4-OV	31.12	-	7.24	-	0.10	-
		Q_{12}(MV)	Q_{12}(SSR)	Q_{13}(MV)	Q_{13}(SSR)	Q_{14}(MV)	Q_{14}(SSR)
SOC	LB	**0.17**	**0.09**	0.67	**0.10**	**0.84**	**0.17**
V_5	PG	0.67	0.13	**0.66**	0.14	1.62	0.78
	N4-UP	0.54	-	0.85	-	5.38	-
	N4-OV	1.04	-	1.55	-	50.58	-
		Q_{15}(MV)	Q_{15}(SSR)	Q_{16}(MV)	Q_{16}(SSR)	Q_{17}(MV)	Q_{17}(SSR)
WORD	LB	0.10	**0.10**	**0.20**	**0.12**	**0.21**	**0.13**
V_6	PG	0.17	0.18	0.35	0.23	0.32	0.26
	N4-UP	**0.01**	-	0.48	-	0.46	-
	N4-OV	0.10	-	4.50	-	1.12	-

Table 2: Times for query execution times over views with default rules (in sec).

base graph. Additional computations, such as calculating Skolem functions and the cost associated with copying unchanged portions, also play a significant role. LogicBlox and Postgres show similar trends in their results, and Neo4j consistently takes significantly less time in all cases. This can be attributed to our Neo4J implementation's inherent advantage of not duplicating unchanged nodes and edges.

Finally, in the SSR column we note that materializing SSRs is substantially (up to 6.9x) faster than materializing views, and often comparable to Neo4J. As before, SSRs require less time to build than MV because MV copies unchanged nodes and edges while SSR does not.

Querying Views. Once a view has been materialized, querying it (**Q(MV)**) is often substantially faster than querying the raw graph by unfolding virtual views. Therefore, as for standard GQL queries: if multiple queries are to be made over the views, it makes sense to materialize.

The speedup observed in query execution due to SSRs (**Q(SSR)** vs **Q(MV)**) depend on the ratio of the size of the output graph versus the size of SSRs that encode the output subgraphs. The performance improvement by using SSRs is more noticeable in LogicBlox (up to 6.4x) than in PostgreSQL. This is because PosgreSQL already handles efficiently large graphs, with the help of built in query optimizer with indexes. LogicBlox fully takes advantage of access to smaller relations instead of a large ones. With the help of SSRs, query execution times in both LogicBlox and PostgreSQL become comparable to those in Neo4J. The speedup from SSRs becomes significant when the query involves many joins, in fact analogously to the way a native-graph DBMS often achieves efficiency by retrieving small subgraphs as a composite object.

7. RELATED WORK

The problem of defining, composing [33], maintaining, and indexing views has been studied for many different data models and query languages. Even within the realm of graph and semi-structured data, fundamental differences in the query language and applications have led to fundamentally different results. In the semi-structured view literature, view rewriting [42] and indexing [40] assume paths over a *rooted* graph. Graph query rewriting for SPARQL [37] addresses a model consisting of RDF triples, rather than the more general property graph model that we consider.

Within the property graph space, commercial DBMSs use a wide variety of languages, with Gremlin [43] (a functional graph transformation language) and Cypher/OpenCypher [29] (a true graph query language) as the most popular implementations. GQL/G-CORE [2, 6] seeks to unify their data models and query languages, as well as views or schemas [8, 7]. Research systems like GraphFlowDB [32] and its successor Kùzu [26] develop novel primitives for dedicated graph DBMS storage, query processing, and transaction primitives; Kùzu adopts OpenCypher as its language, and to our knowledge does not support views. Bonifati et al. [13, 15] have investigated numerous problems in answering queries using views over graphs, as well as maintaining graph view instances [14], from both theoretical and practical perspectives. This work heavily informed the GQL specification. However, they are not concerned with graph normalization and abstraction (as proposed, e.g., for provenance graphs [12, 11]), for which we propose "default rules". Kaskade [20] perhaps relates most closely in spirit to our work, in materializing graph views and rewriting queries to leverage these views. Its query language seems to be a subset of OpenCypher with a grouping clause, which returns nested relation-like output; whereas we support full graph-to-graph transforms as well as subgraph substitution. Finally, we develop RDBMS indexing schemes, in the spirit of the access support relation [36] and the t-index [40].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we investigated semantics strategies for implementing the view capabilities proposed in the emerging property graphs standard, GQL, using existing DBMS platforms. We developed novel strategies for materializing views in property graph DBMSs and for subgraph substitution relations in relational DBMSs. We considered extensions which make it easy to copy all *unmapped* nodes and edges to the view, via the default rule. We demonstrated the performance of our techniques with real-world graph datasets under various view and query workloads, and found that RDBMSs with SSRs provided superior overall performance.

In future work, we intend to investigate support for regular path expressions in views and to look at whether the fundamental ideas of SSRs — capturing the bindings of a subgraph, whereas the *structure* of the connections can be specified via a rule — might also be useful for data compression.

9. ACKNOWLEDGEMENTS

The authors would like to thank our anonymous reviewers for their feedback. We also gratefully acknowledge the support of NSF grant IIS-1910108 for portions of this work.

10. REFERENCES

[1] https://github.com/PennGraphDB/pg-view.

[2] https://www.gqlstandards.org.

[3] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, Seattle, WA, 1998.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[5] F. Afrati and V. Pavlaki. Rewriting queries using views with negation. *AI Communications*, 19(3):229–237, 2006.

[6] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, et al. G-core: A core for future graph query languages. In *SIGMOD*, pages 1421–1432. ACM, 2018.

[7] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens, et al. Pg-schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[8] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens, et al. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2423–2436, 2021.

[9] C. Beeri and M. Y. Vardi. Formal systems for tuple and equality generating dependencies. *SIAM Journal on Computing*, 13(1):76–98, 1984.

[10] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, pages 1072–1081, 2008.

[11] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1072–1081. IEEE, 2008.

[12] O. Biton, S. B. Davidson, S. Khanna, and S. Roy. Optimizing user views for workflows. In *ICDT*, pages 310–323, 2009.

[13] A. Bonifati and S. Dumbrava. Graph queries: From theory to practice. *ACM SIGMOD Record*, 47(4):5–16, 2018.

[14] A. Bonifati, S. Dumbrava, and E. J. G. Arias. Certified graph view maintenance with regular datalog. *Theory and Practice of Logic Programming*, 18(3-4):372–389, 2018.

[15] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. Querying graphs. *Synthesis Lectures on Data Management*, 10(3):1–184, 2018.

[16] A. Bonifati, F. Murlak, and Y. Ramusat. Transforming property graphs. *Proc. VLDB Endow.*, 17(11):2906–2918, Aug. 2024.

[17] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 29, 05 2002.

[18] A. Cali, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research*, 48:115–174, 2013.

[19] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200. IEEE, 1995.

[20] J. M. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun. Kaskade: Graph views for efficient graph analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 193–204. IEEE, 2020.

[21] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, et al. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258, 2022.

[22] A. Deutsch, L.Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[23] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Aggregation support for modern graph analytics in tigergraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 377–392, 2020.

[24] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. In *PODS*, pages 90–101, 2003.

[25] W. Fan and P. Lu. Dependencies for graphs. *ACM Trans. Database Syst.*, 44(2), feb 2019.

[26] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoğlu. Kùzu graph database management system. CIDR, 2023.

[27] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoc. Gpc: A pattern calculus for property graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 241–250, 2023.

[28] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W.-C. Tan. Peer data exchange. In *PODS*, pages 160–171, 2005.

[29] A. Green, M. Junghanns, M. Kießling, T. Lindaaker, S. Plantikow, and P. Selmer. opencypher: New directions in property graph querying. In *EDBT*, pages 520–523, 2018.

[30] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[31] A. Gupta and I. S. Mumick. Incremental maintenance of recursive views: A survey. 1999.

[32] P. Gupta, A. Mhedhbi, and S. Salihoglu. Columnar storage and list-based processing for graph database management systems.

[33] A. Y. Halevy. Theory of answering queries using views. *ACM SIGMOD Record*, 29(4):40–47, 2000.

[34] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[35] S. Han and Z. G. Ives. Implementation strategies for views over property graphs. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

[36] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *VLDB*, pages 290–301, San Francisco, CA, USA, 1990.

[37] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on sparql views. In *Proceedings of the 20th international conference on World wide web*, pages 655–664. ACM, 2011.

[38] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[39] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *VLDB*, volume 93, pages 171–181. Citeseer, 1993.

[40] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[41] L. Moreau, P. Missier, K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, et al. Prov-dm: The prov data model.

[42] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *ACM SIGMOD Record*, volume 28, pages 455–466. ACM, 1999.

[43] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.

[44] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.