The KG-ER Conceptual Schema Language

Enrico Franconi
Free University of Bozen-Bolzano, Italy
franconi@inf.unibz.it

Benoît Groz
Université Paris-Saclay, CNRS, LISN, France
groz@lisn.fr

Jan Hidders
Birkbeck, University of London
London, UK
j.hidders@bbk.ac.uk

Nina Pardal
University of Huddersfield, UK
n.pardal@hud.ac.uk

Sławek Staworko*
Relational AI, Berkeley, CA, USA
slawek.staworko@relational.ai

Jan Van den Bussche
Hasselt University, Hasselt, Belgium
jan.vandenbussche@uhasselt.be

Piotr Wieczorek
University of Wrocław, Poland
piotr.wieczorek@cs.uni.wroc.pl

Abstract

We propose *KG-ER*, a conceptual schema language for knowledge graphs that describes the structure of knowledge graphs independently of their representation (relational databases, property graphs, RDF) while helping to capture the semantics of the information stored in a knowledge graph.

1 Introduction

Knowledge graphs (KGs) have become central to many AI applications Thalhammer et al. [2022], benefiting many AI-based tasks, including NLP and reasoning Mirasdar and Bedekar [2025], data integration Liang et al. [2022], Farah et al. [2025], and semantic search Verma [2024]. KGs organize information as graphs, with nodes as entity instances and edges as relationship instances Hogan et al. [2021], Chaudhri et al. [2022]. Systems like property graphs Gheerbrant et al. [2024] and RDF Patel-Schneider et al. [2025] are built on this principle, and even relational databases can be viewed as KGs Ndefo and Franconi [2020]: they too store information about objects and their relationships, although the information may be structured in a more arbitrary manner.

To use KGs effectively, it is essential to describe their structure and semantics. Database schemas define expected data structures, e.g., specifying that a *Book* is *written* by an *Author* and has a unique *ISBN*. Conceptual modeling languages—such as ER Chen [1976], Di Battista and Lenzerini [1993], ORM2 Franconi and Halpin [2020], Halpin and Morgan [2024], and UML Fowler [2003], Berardi et al. [2005]—provide intuitive, high-level interpretations of these structures. The line between schemas and conceptual models is often blurred, as conceptual modeling languages frequently serve to define schemas with domain-relevant, intuitive names for object classes (entities) and relationships.

However, supported schema features vary across systems Keet and Fillottrani [2015] and can even be tied to the underlying data model. Consequently, one cannot assume that database schema is sufficiently expressive to fully capture the structure and the semantics of the underlying knowledge graph.

To address this deficiency we propose *KG-ER* (Section 2), a conceptual schema language that serves both as a schema capturing the structure of knowledge graphs and a modeling language to convey their semantics. KG-ER has been designed with the aim to provide modeling features particularly suited for knowledge graphs, and in particular, it supports:

*Corresponding author. †Equal contribution.

IRIS-AI 2025: *International Research and Industry Symposium on Artificial Intelligence*, Montefiascone, Italy.

- Entity types, with support for fine-grained inheritance (disjointness and totality) and expressive key constraints based on tree-patterns (acyclic conjunctive queries).
- Relationship types, of arbitrary arity, with support for multi-edge relationships, key constraints (known as determinants in TigerGraph Lee et al. [2023]), and participation constraints.
- Attributes, for entities and relationships, with support for multi-valued attributes, as well as mandatory and single-valued attributes.

We intentionally omit less commonly embraced concepts, such as cardinality constraints or inheritance between relationships; previous research Bonifati et al. [2022, 2023], Groz et al. [2022] has revealed that these features are indeed rarely used in practice. We also pragmatically limit the scope of certain concepts, such as general disjointness of entities, whose interpretation in the context of knowledge graphs leads to a complex and nuanced debate, as presented in Section 5.

If we compare KG-ER to major existing knowledge graph data models, we can observe the following. Compared to ER and EER data models it has extra features such multi-edge semantics for relationships, a more powerful notion of keys, more restricted participation constraints and it does not allow relationship hierarchies. Compared to PG-Schema it has simpler key constraints, lacks cardinality constraints and union types, although the latter can be simulated. Compared to SHACL and ShEx (which are in some ways similar and in some ways different from SHACL Ahmetaj et al. [2025]) it lacks features such as complex constraints based on regular path queries and constraints that involve nested quantifiers and boolean operators, but it does add composite keys. We argue that KG-ER because of its choice of features is eminently suitable for discussing and designing knowledge graphs stored in different logical data models such as RDF, Property Graphs and relational databases.

As evidence of the usefulness and expressiveness of KG-ER, we note that the schema of the LDBC-SNB benchmark Angles et al. [2020] can be captured using KG-ER. Naturally, KG-ER can be extended with support for additional features, if needed.

We also provide, in Sections 3 and 4 a rigorous formal semantics for KG-ER, which can be used to map KG-ER to existing schemas for knowledge graph representations, such as property graphs schemas Angles et al. [2023], schemas for RDF such as ShEx and SHACL Pareti and Konstantinidis [2022], Staworko et al. [2015], and relational schemas in various normal forms. In Appendix B, we provide examples of such translations. Because KG-ER is a conceptual schema language, it is conceivable to use it for constructing mappings and transformations between knowledge graphs stored using different representations e.g., an RDF with a SHACL to a Property Graph with a PG-Schema.

There are many benefits of using KG-ER in the context of AI practice and research alike. For AI practitioners, the KG-ER, which consists of a set of simple statements, can be easily fed into a specific AI model. In Appendix A, we illustrate precisely this benefit by verbalizing KG-ER in helping LLMs in solving a number of common database tasks (text-to-query, query optimization, and schema normalization). For AI theoreticians, the precise logical formalization of KG-ER provides a yardstick of the expressive power for a given AI model to possess, should it need to operate on the structural and semantic information about the knowledge graph.

2 KG-ER Schemas

Throughout this paper we assume pairwise disjoint sets of names of entities $\mathcal{E}nt$, relationships $\mathcal{R}el$, attributes $\mathcal{A}ttr$, and roles $\mathcal{R}ol$.

A *KG-ER schema* (or simply *schema*) is a set S of simple statements that we divide into two parts: the shape graph and the constraints. For every kind of statement we provide an intuitive natural language verbalization, and we provide precise formal semantics in Sections 3 and 4. We consider only *well-formed* schemas which satisfy certain structural conditions, stated alongside the introduced statements (using the **WF** indicator).

2.1 Shape Graphs

A shape graph of a schema consists of simple structural elements that describe the basic topology of knowledge graphs.

The shape graph part of a schema S is a collection of statements of one of the following forms (with $E \in \mathcal{E}nt$, $R \in \mathcal{R}el$, $X \in \mathcal{E}nt \cup \mathcal{R}el$, $A \in \mathcal{A}ttr$, and $B \in \mathcal{R}ol$):

ENTITY(E): *E is an entity*;
RELATIONSHIP(R): *R is a relationship*;
ATTRIBUTE(X, A): *A is an attribute of the entity/relationship X*;

ROLE(R, B, E): *E participates in R in the role B.*

WF$_1$: *All attribute and role declarations of a shape graph only use entities and relationships declared in the shape graph.* Formally, for every ATTRIBUTE(X, A) in the shape graph, it also contains ENTITY(X) or RELATIONSHIP(X), and for every ROLE(R, B, E) in the shape graph, it also contains RELATIONSHIP(R) and ENTITY(E).

Example 1. As an example consider the following set of statements that describe a simple shape graph with persons studying at universities.

ENTITY(*University*), ATTRIBUTE(*University*, *name*),
ENTITY(*Person*), ATTRIBUTE(*Person*, *fname*),
ATTRIBUTE(*Person*, *lname*), ATTRIBUTE(*Person*, *email*),
RELATIONSHIP(*studies*), ATTRIBUTE(*studies*, *year*),
ROLE(*studies*, *uni*, *University*), ROLE(*studies*, *student*, *Person*). □

For simplicity, we assume global uniqueness of attribute and role names in the schema i.e., no attribute name appears more than once (with different entities or relationships) and similarly no role name appears more than once (with different relationships or connected entities). Note that attribute names and role names are also assume not to overlap. The shape graph in Example 1 satisfies this assumption but it would violate it if it also contained ATTRIBUTE(*University*, *email*).

Relational vocabulary The shape graph of a schema S defines the finite vocabulary $\mathcal{L}_S$ that consists of the set of entity names $\mathcal{E}nt_S$, relationship names $\mathcal{R}el_S$, attribute names $\mathcal{A}tt_S$, and role names $\mathcal{R}ol_S$ used in S. For simplicity of notation, we assume a fixed shape graph, and in the remaining, we consistently use E_i, with $1 \leq i \leq n$, to range over $\mathcal{E}nt_S$, R to range over $\mathcal{R}el_S$, X to range over $\mathcal{E}nt_S \cup \mathcal{R}el_S$, A to range over $\mathcal{A}tt_S$, and B to range over $\mathcal{R}ol_S$.

2.2 Graphical Representation and Pattens

Graphical Representation A shape graph is represented diagrammatically by using different node shapes and labels for both nodes and edges. We use rectangular nodes for entities, oval nodes for attributes, and hexagonal nodes for relationships. The node label is the name of the respective element. Additionally, role names are used as labels for the edges that connect relationships to entities.

Example 2. Graphical representations are conveniently more compact as illustrated in Figure 1, which

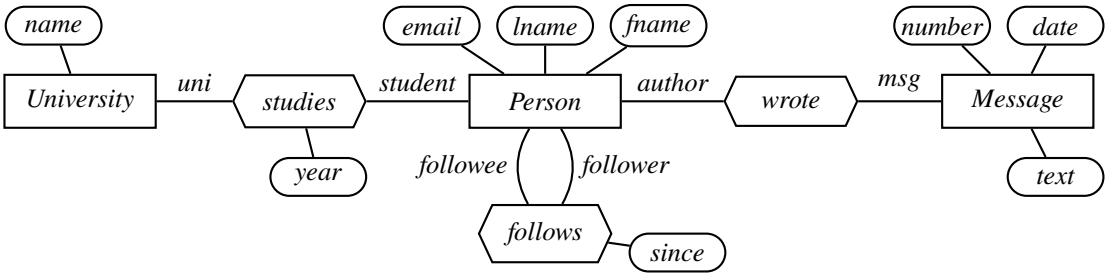

Figure 1: Shape graph inspired by the LDBC-SNB schema.

extends the shape graph in Example 1. □

Patterns We use *tree patterns*, a simple yet powerful querying metaphor, to specify the key information that constitutes the identifying information of entity and relationship instances. Formally, a *pattern* over a schema S is a term p over the set of attribute names $\mathcal{A}tt_S$ and role names $\mathcal{R}ol_S$, with the attribute names appearing only in the leaf nodes. The *arity* of a pattern is the number of its leaf nodes.

We use patterns to query knowledge graphs whose structure conforms to the shape graph, and so we only use valid patterns that navigate through the shape graph following the existing edges labeled with the given role names and may terminate at existing attributes. Patterns have a natural meaning, which is best illustrated graphically as shown on the example below.

Example 3. We illustrate patterns with graphical examples in Figure 2, which contains a fragment of the shape graph from Example 2. The simple pattern $p_1 = name$ consists of a single attribute and allows to retrieve the name of a university. A similarly simple pattern $p_2 = email$ allows to access email address of a person.

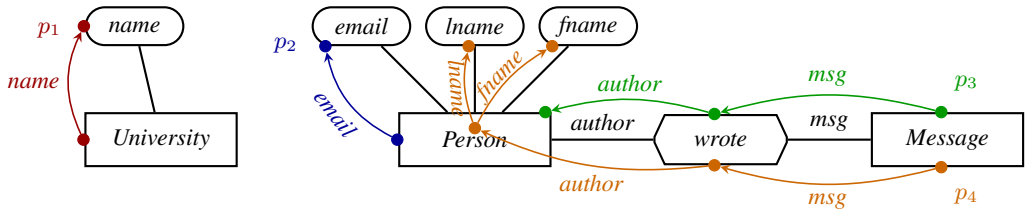

Figure 2: Examples of patterns in a fragment of LDBC-SNB shape graph.

The pattern $p_3 = msg(author)$ allows to retrieve the person that has authored a given message; this pattern retrieves the instances of the instances of *Person* entity. We are often interested in patterns that retrieve only concrete (attribute) values. As an alternative to the pattern p_3, we can use the pattern $p_4 = msg(author(fname, lname))$ that retrieves the first and last name of the author of a message. □

Note that patterns do not need to have attribute names in all leaf nodes, but those that do are called *ground* patterns. Ground patterns are particularly important because they allow to retrieve concrete data values (attribute values) and avoid using identifiers of entities and relationships that are dependent on the particular representation of a knowledge graph.

Syntactically valid patterns We only work with valid patterns that can be "embedded" in the graphical representation of the shape graph as illustrated in Figure 2. For completeness in our presentation, we formally define valid patterns as patterns rooted at an entity or relationship as follows. A pattern p *is rooted at* X *in* S iff (1) p is an attribute name A and X has attribute A; (2) p is a role name R, and either (2.a) X is a relationship with role R, or symmetrically (2.b) X is an entity that participates in a relationship in the role B; (3) $p = B(p_1, \ldots, p_k)$, with $k \geq 1$, and either (3.a) X is a relationship with an entity E participating in the role B, and each p_i is rooted at E, or symmetrically (3.b) X is an entity that participates in a relationship R in the role B, and each p_i is rooted at R.

2.3 Constraints

The constraints part of a schema S are a collection of statements that we divide into a number of categories. Again, with every statement we provide a natural language verbalization and define the formal semantics later on.

2.3.1 Participation Constraints

Participation constraints specify the minimum (0 or 1) and maximum number (1 or unbounded) of relationships instances that an entity instance participates in. They are a simple subclass of cardinality constraints, which are not used frequently in practice, and for which there is no general tool support. An analogous subclass of cardinality constraints allows to specify the minimum and maximum number of attribute values of an entity or relationship X and by abusing the naming convention we put them in the same category.

MANDATORY(X, A): *every instance of* X *must have an attribute value for* A;

SINGLE(X, A): *every instance of entity or relationship* X *must have at most one attribute value of* A;

MANDATORY(E, B, R): *every instance of the entity* E *must participate in an instance of relationship* R *through the role* B;

SINGLE(E, B, R) *every instance of the entity* E *can participate in at most one instance of the relationship* R *through the role* B.

WF$_2$: *Every participation constraint is declared for an entity or a relationship instance that is declared in the shape graph.* Formally, for every MANDATORY(X, A) and every SINGLE(X, A) the shape graph contains ATTRIBUTE(X, A), and for every MANDATORY(E, B, R) and every SINGLE(E, B, R) the shape graph contains ROLE(R, B, E).

Example 4. As an example, for the shape graph in Figure 1 we can declare the *date* attribute of the *Message* entity as mandatory and single with the pair of statements MANDATORY$(Message, date)$ and SINGLE$(Message, date)$. To declare that every *Message* must have precisely one author i.e., every instance of *Message* participates in the role of *msg* in exactly one instance of the *wrote* relationship, we can use these two statements MANDATORY$(Message, msg, wrote)$ and SINGLE$(Message, msg, wrote)$. □

2.3.2 Key Constraints

We introduce two kinds of key constraints, allowing to identify instances of entity or relationship X. Simple key requires the uniqueness of the identifying information specified with a collection of patterns $p_1, \ldots, p_k$. A stronger notion of identity key requires additionally that the identifying information is always present and unique (and so constitutes canonical identifying information). In both cases below the patterns $p_1, \ldots, p_k$ are rooted at X in the shape graph.

- KEY$(X, [p_1, \ldots, p_k])$: *no two instances of X may have the same key values obtained with $p_1, \ldots, p_k$;*
- IDENTITY$(X, [p_1, \ldots, p_k])$ *every instance of X must have precisely one tuple of key values obtained with $p_1, \ldots, p_k$, and no two instances of X may have the same key values.*

We introduce identifying keys for very specific purpose: they are used to identify instances of entities and relationships in a manner independent of the representation of a knowledge graph.

WF$_3$: *Identity keys use ground patterns only.*

Example 5. Identity keys are naturally used to identify entities. For example, in the shape graph in Figure 1 we declare the following key: IDENTITY(*Person*, [*fname*, *lname*]).

We can use simple keys to express secondary keys, e.g., Persons may have multiple email addresses, or even none, but no two persons should have the same email address, which can be expressed with KEY(*Person*, [*email*]).

The use of terms in a key allows us to capture the common concept of a weak entity. For instance, suppose that *Message* is a weak entity of *Person*, which means that the identity of a message is the combination of the identity of its author and the message number. We can express this with the identity key IDENTITY(*Message*, [*msg*(*author*), *number*]), or if we insist on using ground patterns, with IDENTITY(*Message*, [*msg*(*author*(*fname*, *lname*)), *number*]).

Keys can also be used on relationships, e.g., a person can be enrolled at a given university only once in a given year, which can be expressed with the key KEY(*studies*, [*uni*, *student*, *year*]). □

A standard assumption in graph theory is that graphs are simple, that is, they allow at most one edge connecting the same pair of nodes; this assumption is transferred quite unconsciously to knowledge graphs. However, there exist frameworks, such as property graphs and the recent RDF 1.2 Patel-Schneider et al. [2025], that allow multiple edges between the same pair of nodes. For the sake of generality, we do not restrict knowledge graphs to have at most one edge between the same pair of nodes. For instance, in the running example of the schema (Figure 1) the same person can study at the same university multiple times (although, at different years). Interestingly, keys allow us to impose the restriction of singular edges if it is required for a particular application.

Example 6. Person following another person is a singular fact, that should not be unnecessarily repeated, which is captured with the key KEY(*follows*, [*follower*,*followee*]). □

WF$_4$: *Every relationship has an identifying key.*

Note that one can use a trivial identifying key consisting of all its attributes and roles.

2.3.3 Type Hierarchy

It is a common and useful mechanism to organize entities into a hierarchy. For instance, in the LDBC-SNB schema, the *Message* entity has two subclasses, *Post* and *Comment*. These two subclasses are disjoint: a *Post* cannot be a *Comment* and vice versa. Furthermore, in this instance, the subclass relation is total: every *Message* is either a *Post* or a *Comment*. To support these features we introduce the following statements.

ISA(E_1, E_2): *E_1 is a subclass of E_2; in particular, E_1 inherits all attributes, relationships, and constraints of E_2.*

WF$_5$: *The type hierarchy formed with the ISA statements is acyclic.*

We clarify that multiple inheritance is not disallowed and moreover, the type hierarchy may consist of disconnected components. To make sure that we can distinguish between entities in the same component we introduce the following requirement.

WF$_6$: *The entity at any root of the type hierarchy has an identifying key.*

Two entities from the same component of the type hierarchy can be made disjoint.

DISJOINT(E_1, E_2): *no instance of E_1 is an instance of E_2 and vice versa.*

Finally, to express total inheritance we introduce a statement that allows to specify coverage of an entity E by its descendants $E_1, \ldots, E_k$ in the type hierarchy.

COVER$(\{E_1, \ldots, E_k\}, E)$: *any instance of E is an instance of at least one of $E_1, \ldots, E_k$.*

Example 7. The type hierarchy among the entities *Message*, *Post*, and *Comment* can be expressed with the following statements:

ISA(*Post*, *Message*), ISA(*Comment*, *Message*),
DISJOINT(*Post*, *Comment*), COVER($\{$*Post*, *Comment*$\}$, *Message*). □

3 Data Model

Recall that the shape graph of a schema S identifies the finite vocabulary $\mathcal{L}_S$ of entity names $\mathcal{E}nt_S$, relationship names $\mathcal{R}el_S$, attribute names $\mathcal{A}tt_S$, and role names $\mathcal{R}ol_S$ used in S. Also recall that, for simplicity, we work under a unique name assumption: no attribute name and no role name are used with more than a single entity or relationship.

We define knowledge graphs as relational structures over the union of two disjoint sets of elements: the entity and relationship instances $\mathcal{V} = \mathcal{V}_{\text{ent}} \cup \mathcal{V}_{\text{rel}}$ and the data values $\mathcal{D}$. Namely, a *knowledge graph* (or simply a *graph*) G over $\mathcal{L}_S$ is a function $\cdot^G$ that gives each element of $\mathcal{L}_S$ an interpretation:

$E^G \subseteq_{\text{fin}} \mathcal{V}_{\text{ent}}$ for every $E \in \mathcal{E}nt_S$, $A^G \subseteq_{\text{fin}} (\mathcal{V}_{\text{ent}} \cup \mathcal{V}_{\text{rel}}) \times \mathcal{D}$ for every $A \in \mathcal{A}tt_S$,
$R^G \subseteq_{\text{fin}} \mathcal{V}_{\text{rel}}$ for every $R \in \mathcal{R}el_S$, $B^G \subseteq_{\text{fin}} \mathcal{V}_{\text{rel}} \times \mathcal{V}_{\text{ent}}$ for every $B \in \mathcal{R}ol_S$.

Moreover, we define the sets $N_{\text{ent}} = \bigcup\{E^G \mid E \in \mathcal{E}nt_S\}$ and $N_{\text{rel}} = \bigcup\{R^G \mid R \in \mathcal{R}el_S\}$, and we require that:

1. we do not assign attributes or roles to entity or relationship instances that do not belong to any entity or any relationship respectively i.e., $A^G \subseteq (N_{\text{ent}} \cup N_{\text{rel}}) \times \mathcal{D}$ for every $A \in \mathcal{A}tt_S$ and $B^G \subseteq N_{\text{rel}} \times N_{\text{ent}}$ for every $B \in \mathcal{R}ol_S$,
2. role names are partial functions, i.e., if $B^G(r, e)$ and $B^G(r, e')$ then $e = e'$ for every $B \in \mathcal{R}ol_S$ and $r \in R^G$,
3. every relationship instance belongs to exactly one relationship in $\mathcal{R}el_S$, i.e., $R_1^G \cap R_2^G = \emptyset$ for every $R_1 \neq R_2 \in \mathcal{R}el_S$.

By $\mathcal{G}(S)$ we denote the set of all knowledge graphs over $\mathcal{L}_S$.

4 Semantics

Given a schema S and a knowledge graph G over $\mathcal{L}_S$, we define when G satisfies S by translating every statement in S into a first-order logic formula. For keys, it is important to translate patterns into formulas that extract the identifying information.

4.1 Semantics of Patterns

The translation is relatively natural but to handle correctly the direction of the role predicates we need to know if the pattern is evaluated (rooted) at an entity or a relationship. We illustrate this translation on a simple example below.

Example 8. The pattern $p_4 = \mathit{msg}(\mathit{author}(\mathit{fname}, \mathit{lname}))$ from Example 3 is rooted at *Message* and naturally translated to the following formula:

$$\varphi_{p_4}^{\mathit{Message}}(x, y_1, y_2) = \exists w, p.\ \mathit{msg}(x, w) \wedge \mathit{author}(w, p) \wedge \mathit{fname}(p, y_1) \wedge \mathit{lname}(p, y_2). \quad \square$$

Take now the simple pattern $p_0 = \mathit{student}$. Its interpretation depends on whether it is rooted at the entity *Student* or at the relationship *studies*. When rooted at *Student*, we translate it to this formula:

$$\varphi_{p_0}^{\mathit{Person}}(x, y) = \mathit{student}(y, x)$$

whereas when rooted at the relationship *studies*, we translate it to this formula:

$$\varphi_{p_0}^{\mathit{studies}}(x, y) = \mathit{student}(x, y). \quad \square$$

Formally, we translate a pattern p of arity k rooted at X to a formula $\varphi_p^X(x, y_1, \ldots, y_k)$ with the following recursive procedure: (1) For an attribute leaf $p = A$ the formula is $\varphi_p^X(x, y) = A(x, y)$; (2) For a role leaf $p = B$ we have either (2.a) X is an entity and the formula is $\varphi_p^X(x, y) = B(y, x)$ or, (2.b) X is a relationship and the formula is $\varphi_p^X(x, y) = B(x, y)$; (3) For a non-leaf pattern $p = B(p_1, \ldots, p_m)$ with $B \in \mathcal{R}ol'_S$, we have either (3.a) X is an entity that participates in a relationship R in the role B and then the formula is $\varphi_p^X(x, \bar{y}_1, \ldots, \bar{y}_m) = \exists z.\ B(z, x) \wedge \varphi_{p_1}^R(z, \bar{y}_1) \wedge \ldots \wedge \varphi_{p_m}^R(z, \bar{y}_m)$ or, (3.b) X is a relationship that participates in the role B with an entity E and then the formula is $\varphi_p^X(x, \bar{y}_1, \ldots, \bar{y}_k) = \exists z.\ B(x, z) \wedge \varphi_{p_1}^E(z, \bar{y}_1) \wedge \ldots \wedge \varphi_{p_m}^E(z, \bar{y}_m)$.

4.2 Core Semantics of KG-ER

We map each statement s of a schema S into a FOL formula $[\![s]\!]$ that captures its meaning, as presented in Figure 3.

$$
\begin{aligned}
&[\![\textsc{Entity}(X)]\!] = \mathit{true}, \qquad [\![\textsc{Relationship}(R)]\!] = \mathit{true},\\
&[\![\textsc{Attribute}(X, A)]\!] = \forall x, y.\ A(x, y) \Rightarrow X(x),\\
&[\![\textsc{Role}(R, B, E)]\!] = \forall x, y.\ B(x, y) \Rightarrow R(x) \wedge E(y),\\
&[\![\textsc{Mandatory}(X, A)]\!] = \forall x.\ X(x) \Rightarrow \exists y.\ A(x, y),\\
&[\![\textsc{Single}(X, A)]\!] = \forall x, y, z.\ A(x, y) \wedge A(x, z) \Rightarrow y = z,\\
&[\![\textsc{Mandatory}(E, B, R)]\!] = \forall x.\ E(x) \Rightarrow \exists y.\ B(y, x),\\
&[\![\textsc{Single}(E, B, R)]\!] = \forall x, y, z.\ B(y, x) \wedge B(z, x) \Rightarrow y = z,\\
&[\![\textsc{Key}(X, [p_1, \ldots, p_k])]\!] = \forall x, y, \bar{z}_1, \ldots, \bar{z}_k.\ \psi(x, \bar{z}) \wedge \psi(y, \bar{z}) \Rightarrow x = y \text{ where}\\
&\quad \psi(x, \bar{z}) = X(x) \wedge \textstyle\bigwedge_i \varphi_{p_i}^X(x, \bar{z}_i),\\
&[\![\textsc{Identity}(X, [p_1, \ldots, p_k])]\!] = [\![\textsc{Key}(X, [p_1, \ldots, p_k])]\!] \wedge \psi^{\geq 1} \wedge \psi^{\leq 1} \text{ where}\\
&\quad \psi^{\geq 1} = \forall x.\ X(x) \Rightarrow \exists \bar{y}_1, \ldots, \bar{y}_k.\ \textstyle\bigwedge_i \varphi_{p_i}^X(x, \bar{y}_i) \text{ and}\\
&\quad \psi^{\leq 1} = \forall x.\ X(x) \Rightarrow \forall \bar{y}_1, \ldots, \bar{y}_k, \bar{z}_1, \ldots, \bar{z}_k.\ \textstyle\bigwedge_i \varphi_{p_i}^X(x, \bar{y}_i) \wedge \bigwedge_i \varphi_{p_i}^X(x, \bar{z}_i) \Rightarrow \bigwedge_i \bar{y}_i = \bar{z}_i,\\
&[\![\textsc{Isa}(E_1, E_2)]\!] = \forall x.\ E_1(x) \Rightarrow E_2(x),\\
&[\![\textsc{Disjoint}(E_1, E_2)]\!] = \forall x.\ E_1(x) \wedge E_2(x) \Rightarrow \mathit{false},\\
&[\![\textsc{Cover}(\{E_1, \ldots, E_k\}, E)]\!] = \forall x.\ E(x) \Rightarrow E_1(x) \vee \ldots \vee E_k(x).
\end{aligned}
$$

Figure 3: Core semantics of KG-ER

Now, the *core semantics* of schema S are the knowledge graphs that satisfy all its statements:

$$L_\circ(S) = \{G \in \mathcal{G}(S) \mid G \models \textstyle\bigwedge_{s \in S} [\![s]\!]\}.$$

It can be shown that schema reasoning in *KG-ER*, namely deciding entailment among graphs, is decidable in EXPTIME, by encoding entailment *KG-ER* to the *FunDL* Feature-Based Description Logics–by reifying the relationships using features as suggested in McIntyre et al. [2019]; this means that there can be concrete algorithms to reason with *KG-ER* graphs.

5 Disjointness and Identity

There are three important, but related, issues when designing a conceptual data model: (Q1) *How identifiable should entities be?* (Q2) *Do we have implicit disjointness of entities without common supertypes?* and (Q3) *For which entities is it possible to indicate disjointness?* In the data models that are used to represent knowledge graphs, these questions are answered quite differently. For example, in relational schemas and ER diagrams there is usually a very strong notion of identifiability, e.g., by requiring that all entities must have a primary key. However, in RDF and LPG schema languages there is usually no such requirement. Concerning implicit disjointness, this is usually assumed in relational schemas and ER diagrams. However, in schema languages for property graphs and RDF there is usually no such assumption. For disjointness constraints, these are in extended ER schema languages usually only allowed between entities that share a common supertype, whereas in RDF schema languages there is usually no such restriction.

For Q1 we define identifiability as the ability to identify entities in terms of an entity description that consists of the entity and a combination of associated data values. We can distinguish three different

levels of identifiability, each increasingly more demanding, but coming with practical benefits: (1) **referenceability**, which means that for all entity instances within a certain entity there is at least one entity description that identifies only that entity instance, (2) **local distinguishability**, which means that within each entity, each entity instance has exactly one such entity description, and (3) **global distinguishability**, which means that the DBMS is able to tell for two entity descriptions if they refer to the same entity or not.

In general global distinguishability is clearly preferred: it makes the instance of a schema less ambiguous and makes it easier to map it faithfully and straightforwardly to a database model such as for example a relational schema while avoiding surrogate keys or artificial identifiers. This is achieved for example in ORM2 by requiring that (R1) all entities have or inherit a well-founded identifier and (R2) entities that do not have a common supertype are assumed to be disjoint. Note that (R1) only ensures local distinguishability, but in combination with (R2) it also ensures global distinguishability. After all, given two entity descriptions it will hold that their entities have a common supertype, or not. In the first case, the DBMS can use local distinguishability in that common supertype to decide if the descriptions refer to the same entity, and in the second case it knows that they cannot refer to the same entity.

For KG-ER we have decided to require R1, but not necessarily R2, since R2 may not be possible or desirable to enforce for some domains. The core semantics $L_\circ$ satisfies R1 but not R2. However, when R2 is also required, we propose an alternative semantics $L_\perp$ with *implicit disjointness*. Namely, by θ_S we denote the conjunction of $[\![\textsc{Disjoint}(E_1, E_2)]\!]$ for any two entities $E_1, E_2 \in \mathcal{E}nt_S$ that are not directly related in the type hierarchy, i.e., do not have a common Isa-ancestor. We can now formally define the semantics with implicit disjointness as

$$L_\perp(S) = \{G \in L(S) \mid G \models \theta_S\}.$$

Finally, the issue of question Q3 is answered in KG-ER by only allowing explicit disjointness constraints between entities that have a common supertype. Under the $L_\perp$ semantics this makes sense, since disjointness constraints between entities that do not have a common supertype would be redundant. However, even under the $L_\circ$ semantics it makes sense to disallow such constraints, since they might be hard to map to efficient database schemas.

6 Conclusions and Future Work

We have presented KG-ER, an expressive modeling language with the purpose of providing a unified schema language for knowledge graphs stored in a large variety of database systems. We have built this language through diligent selection of the most useful and frequently-used features while being considerate of particular features of existing database systems used for representing knowledge graphs.

A formalized modeling language allows to ask and treat with the necessary rigor many interesting questions, some that fall into the large AI domain. Firstly, we would like to know how faithfully can we translate a KG-ER instance to existing database models (relational, property graphs, RDF) and their schemas, and conversely, can a schema of an existing database, say relational, be faithfully mapped to KG-ER? How can AI techniques, and LLMs in particular, offer assistance in those tasks? This leads to another interesting question: can KG-ER be effectively used to help to define mappings between different database models?

Acknowledgments and Disclosure of Funding

Piotr Wieczorek was supported by the National Science Centre (NCN), Poland under grant 2020/39/B/ST6/00521

References

Shqiponja Ahmetaj, Iovka Boneva, Jan Hidders, Katja Hose, Maxime Jakubowski, Jose Emilio Labra Gayo, Wim Martens, Fabio Mogavero, Filip Murlak, Cem Okulmus, Axel Polleres, Ognjen Savković, Mantas Šimkus, and Dominik Tomaszuk. Common foundations for shacl, shex, and pg-schema. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 8–21, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712746. https://doi.org/10.1145/3696410.3714694.

Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, and Norbert Martínez-Bazan et al. The LDBC social network benchmark. *CoRR*, abs/2001.02299, 2020. https://doi.org/10.48550/arXiv.2001.02299.

Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic,

Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. Pg-schema: Schemas for property graphs. *Proc. ACM Manag. Data*, 1(2), June 2023. https://doi.org/10.1145/3589778.

Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, October 2005. ISSN 00043702. https://doi.org/10.1016/j.artint.2005.05.003.

Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. Threshold queries in theory and in the wild. *Proc. VLDB Endow.*, 15(5):1105–1118, 2022. https://doi.org/10.14778/3510397.3510407.

Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. Threshold queries. *SIGMOD Rec.*, 52(1):64–73, 2023. https://doi.org/10.1145/3604437.3604452.

Vinay K Chaudhri, Chaitanya Baru, Naren Chittar, Xin Luna Dong, Michael Genesereth, James Hendler, Amit Kalyanpur, Douglas B Lenat, Juan Sequeda, Denny Vrandečić, and Kun Wang. Knowledge Graphs: Introduction, History, and Perspectives. *AI Magazine*, 43(1):17–29, 2022. https://doi.org/10.1002/aaai.12033. Provides a historical overview and different perspectives on knowledge graphs from prominent researchers in the field.

Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. https://doi.org/10.1145/320434.320440.

G. Di Battista and M. Lenzerini. Deductive entity relationship modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):439–450, June 1993. ISSN 10414347. https://doi.org/10.1109/69.224196.

Kirollos Farah et al. Knowledge Graphs: The Future of Data Integration and Insightful Discovery. *arXiv preprint arXiv:2502.15689*, 2025. https://doi.org/10.48550/arXiv.2502.15689. This recent preprint directly addresses knowledge graphs as the future of data integration.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., USA, 3 edition, 2003. ISBN 0321193687.

Enrico Franconi and Terry Halpin. ORM abstract syntax and semantics: normative specification. Technical report, 2020. https://gitlab.com/orm-syntax-and-semantics/orm-syntax-and-semantics-docs.

Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. GQL and SQL/PGQ: Theoretical models and expressive power. arXiv:2409.01102 [cs.DB], 2024. URL https://arxiv.org/abs/2409.01102.

Benoît Groz, Aurélien Lemay, Slawek Staworko, and Piotr Wieczorek. Inference of shape graphs for graph databases. In Dan Olteanu and Nils Vortmeier, editors, *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*, volume 220 of *LIPIcs*, pages 14:1–14:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. https://doi.org/10.4230/LIPICS.ICDT.2022.14.

Terry A. Halpin and Tony Morgan. *Information modeling and relational databases*. Morgan Kaufmann, 3rd edition, 2024.

Aidan Hogan et al. Knowledge Graphs. *Synthesis Lectures on Data, Semantics, and Networks*, 11 (1):1–257, 2021. https://doi.org/10.2200/S01083ED1V01Y202102DSN018. A comprehensive and highly cited tutorial-style book on knowledge graphs, covering various aspects from models to applications.

C. Maria Keet and Pablo Rubén Fillottrani. An analysis and characterisation of publicly available conceptual models. In Paul Johannesson, Mong Li Lee, Stephen W. Liddle, Andreas L. Opdahl, and Óscar Pastor López, editors, *Conceptual Modeling*, pages 585–593, Cham, 2015. Springer International Publishing. ISBN 978-3-319-25264-3. https://doi.org/10.1007/978-3-319-25264-3_45.

Jose Emilio Labra Gayo, Eric Prud'hommeaux, Iovka Boneva, and Dimitris Kontokostas. *Validating RDF Data*, volume 7 of *Synthesis Lectures on the Semantic Web: Theory and Technology*. Morgan & Claypool Publishers LLC, sep 2017. https://doi.org/10.2200/s00786ed1v01y201707wbe016.

Victor Lee, Phuc Kien Nguyen, and Alexander Thomas. *Graph-Powered Analytics and Machine Learning with TigerGraph*. O'Reilly Media, 2023.

Ke Liang et al. A Survey of Knowledge Graph Reasoning on Graph Types: Static, Dynamic, and Multimodal. *arXiv preprint arXiv:2212.05767*, 2022. https://doi.org/10.48550/arXiv.2212.05767.

Stephanie McIntyre, David Toman, and Grant E. Weddell. Fundl - A family of feature-based description logics, with applications in querying structured data sources. In Carsten Lutz, Uli Sattler, Cesare Tinelli, Anni-Yasmin Turhan, and Frank Wolter, editors, *Description Logic, Theory Combination, and All That - Essays Dedicated to Franz Baader on the Occasion of His 60th Birthday*, volume 11560 of *Lecture Notes in Computer Science*, pages 404–430. Springer, 2019. https://doi.org/10.1007/978-3-030-22102-7_19.

Sharayu Mirasdar and Mangesh Bedekar. Knowledge graphs for NLP: A comprehensive analysis. *The Scientific Temper*, 1(1):21–28, 2025. https://doi.org/10.58414/SCIENTIFICTEMPER.2025.16.spl-1.18.

Nonyelum Ndefo and Enrico Franconi. A study on information-preserving schema transformations. *International Journal of Semantic Computing*, 14(1):27–53, 2020. https://doi.org/10.1142/S1793351X20400024.

Paolo Pareti and George Konstantinidis. *A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs*, pages 115–144. Springer International Publishing, Cham, 2022. ISBN 978-3-030-95481-9. https://doi.org/10.1007/978-3-030-95481-9_6. URL https://doi.org/10.1007/978-3-030-95481-9_6.

Peter Patel-Schneider, Dörthe Arndt, and Enrico Franconi. *RDF 1.2 Semantics*. W3C, 2025. URL https://www.w3.org/TR/rdf12-semantics/.

Slawek Staworko, Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold R. Solbrig. Complexity and expressiveness of shex for RDF. In Marcelo Arenas and Martín Ugarte, editors, *18th International Conference on Database Theory, ICDT 2015, March 23-27, 2015, Brussels, Belgium*, volume 31 of *LIPIcs*, pages 195–211. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. https://doi.org/10.4230/LIPICS.ICDT.2015.195.

Matthias Thalhammer et al. Knowledge graphs for integrated AI. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, pages 2501–2507. ACM, 2022. https://doi.org/10.1145/3514221.3526017. This paper discusses how knowledge graphs facilitate integrated AI systems, with data integration being a foundational aspect.

Ajay Verma. Beyond the Box: Exploring Knowledge Graphs for Semantic Search, Syntactic Search, and Graph Search. In *Artificial Intelligence in Plain English (Medium)*, 2024. URL https://ai.plainenglish.io/beyond-the-box-exploring-knowledge-graphs-for-semantic-search-syntactic-search-and-graph-search-747ec75ef4f6. While from Medium, this article provides a good overview and references academic concepts regarding KGs for different types of search, including semantic.

A Appendix: verbalizing KG-ER helps LLMs

We present a number of experiments to illustrate that using a conceptual schema can help LLMs to tailor their answers for a number of natural and commonly used tasks. We performed the experiments using the OpenAI's GPT-4o model on July 4th and 5th, 2025. The prompts and answers are provided below. We point out that in our experiments multiple runs sometime returned different answers, which is expected due to the stochastic nature of LLMs. However, in all executions the observations were consistent: providing a conceptual schema improves the quality of the answers.

A.1 Text to query

We begin with a task where the LLM is provided a relational schema together with a query specification in natural language, and we ask for a corresponding SQL query.

Then, in another session, we provide a similar task where the LLM is provided with a conceptual schema in addition to the relational schema and query specification.

Prompt:
We are given a database whose schema is defined by the statements:

```
CREATE TABLE Employee(
    empl_id INT PRIMARY KEY,
    empl_name TEXT,
    phone TEXT,
    address TEXT
);

CREATE TABLE Department(
    dpt_id INT PRIMARY KEY,
    dpt_name TEXT,
    description TEXT
);

CREATE TABLE Works_in(
    employee_id INT,
    department_id INT,
    FOREIGN KEY(employee_id) REFERENCES Employee(empl_id),
    FOREIGN KEY(department_id) REFERENCES Department(dpt_id),
    PRIMARY KEY(employee_id, department_id)
);
```

Write an SQL query that counts the number of employees per department.

GPT-4o:
To count the number of employees per department, you can use the following SQL query:

```
SELECT
    d.dpt_id,
    d.dpt_name,
    COUNT(w.employee_id) AS num_employees
FROM
    Department d
LEFT JOIN
    Works_in w ON d.dpt_id = w.department_id
GROUP BY
    d.dpt_id, d.dpt_name;
```

```
### Explanation:

* **`LEFT JOIN`** ensures that departments with zero employees are still included in the result.
* **`COUNT(w.employee_id)`** counts the number of employees in each department.
* **`GROUP BY`** groups the result by department so that the count is computed per department.

This will return one row per department along with the count of employees assigned to it.
```

The query provided by the LLM is correct. It specifically observes that a LEFT JOIN is used to ensure that departments without employees are included in the result. We next issued (in another chat) the same task with an additional conceptual schema which in particular specifies that every department must contain at least one employee. We observe that the LLM is able to leverage the conceptual schema and returns a simpler query which uses an internal join rather than the left outer join:

Prompt:
We are given a database whose schema is defined by the statements:

```
CREATE TABLE Employee(
    empl_id INT PRIMARY KEY,
    empl_name TEXT,
    phone TEXT,
    address TEXT
);

CREATE TABLE Department(
    dpt_id INT PRIMARY KEY,
    dpt_name TEXT,
    description TEXT
);

CREATE TABLE Works_in(
    employee_id INT,
    department_id INT,
    FOREIGN KEY(employee_id) REFERENCES Employee(empl_id),
    FOREIGN KEY(department_id) REFERENCES Department(dpt_id),
    PRIMARY KEY(employee_id, department_id)
);
```

```
The database also satisfies the following conceptual schema:

'Employee' is an entity.
'empl_id' is an attribute of the entity 'Employee'.
'empl_name' is an attribute of the entity 'Employee'.
'address' is an attribute of the entity 'Employee'.
'Department' is an entity.
'dept_id' is an attribute of the entity 'Department'.
'dept_name' is an attribute of the entity 'Department'.
'description' is an attribute of the entity 'Department'.
'Employment' is an relationship.
Every instance of the entity 'Employee' participates in one or more
    instances of the relationship 'Employment' through the role 'works_in'.
Every instance of the entity 'Department' participates in one or more
    instances of the relationship 'Employment' through the role 'employs'.

Write an SQL query that counts the number of employees per department.
```

GPT-4o:

```
To count the number of employees per department, we need to:

1. Join the `Department` and `Works_in` tables.
2. Group the result by department.
3. Count how many employees are associated with each department.

Here is the SQL query:
```

```
SELECT
    d.dpt_id,
    d.dpt_name,
    COUNT(w.employee_id) AS num_employees
FROM
    Department d
JOIN
    Works_in w ON d.dpt_id = w.department_id
GROUP BY
    d.dpt_id, d.dpt_name;
```

```
### Explanation:

* `JOIN Department d WITH Works_in w ON d.dpt_id = w.department_id`:
    Connects departments to the employees who work in them.
* `GROUP BY d.dpt_id, d.dpt_name`: Groups the data per department.
* `COUNT(w.employee_id)`: Counts the number of employees in each department.

This query satisfies the conceptual schema's requirement
    that every department has one or more employees.
```

A.2 Query optimization

In this experiment, we investigate whether LLMs can use the information in conceptual schemas to optimize queries.

We provide the LLM with a relational schema together with an SQL query, and we ask the LLM to optimize the query. The presented query joins two copies of `Author` to check conditions on their birth date and country of origin. However, as `book_id` is the primary key for the relationship `Authorship`, it is possible to infer that each book can have at most one author. Therefore, the query can be considerably simplified by removing unnecessary joins.

Prompt:
We are given the relational schema:

```sql
CREATE TABLE Author(
    id INT PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    birth_year INT,
    country TEXT
);

CREATE TABLE Book(
    id INT PRIMARY KEY,
    title TEXT,
    category TEXT
);

CREATE TABLE Authorship(
    book_id INT PRIMARY KEY REFERENCES Book(id),
    author_id INT REFERENCES Author(id)
);
```

Optimize the following SQL query.

```sql
SELECT b.id, b.title
FROM
    Book b JOIN
    Authorship w1 ON w1.book_id = b.id JOIN
    Author a1 ON a1.id = w1.author_id JOIN
    Authorship w2 ON w2.book_id = b.id JOIN
    Author a2 ON a2.id = w2.author_id
WHERE
    a1.birth_year < 2000 AND
    a2.country = 'France';
```

GPT-4o:

```
The SQL query aims to find books (b.id, b.title) that were co-authored by at least two
authors, where:

    One author (a1) was born before 2000, and

    Another author (a2) is from France.

However, the current query performs redundant joins and may produce duplicates due to
self-joins on the same tables. Let's both optimize it logically and simplify the structure.
```

Optimized Version

```sql
SELECT DISTINCT b.id, b.title
FROM Book b
JOIN Authorship aw1 ON aw1.book_id = b.id
JOIN Author a1 ON a1.id = aw1.author_id
WHERE a1.birth_year < 2000
  AND EXISTS (
    SELECT 1
    FROM Authorship aw2
    JOIN Author a2 ON a2.id = aw2.author_id
    WHERE aw2.book_id = b.id AND a2.country = 'France'
  );
```

```
Explanation of Optimization

    Avoids double join on Authorship and Author:

        Original query joins Authorship and Author twice to get a1 and a2.

        We replace the second join with an EXISTS subquery, which is generally
        faster and clearer.

    Uses DISTINCT to avoid duplicate rows caused by multiple matching authors.

    Preserves semantics:

        At least one author (a1) is born before 2000.

        At least one (possibly different) author (a2) is from France.

        Both authors must have authored the same book.

Bonus: Index Recommendations

For better performance on larger datasets, consider adding these indexes:

CREATE INDEX idx_author_birth_year ON Author(birth_year);
CREATE INDEX idx_author_country ON Author(country);
CREATE INDEX idx_authorship_book_author ON Authorship(book_id, author_id);

Let me know if you'd like to further optimize for a specific SQL engine (PostgreSQL, MySQL,
SQLite, etc.), or want to extend this to more complex logic (e.g., exact number of authors,
both conditions on the same author, etc.).
```

The optimized version of the query is correct, but the LLM does not infer that both conditions in the query can be checked for the single author of each book and keeps unnecessary joins. Repeated experiments generated many alternative query versions, yet none incorporated this cardinality constraint.

See additional examples of queries generated by GPT-4o below:

```sql
SELECT DISTINCT b.id, b.title
FROM Book b
JOIN Authorship w1 ON w1.book_id = b.id
JOIN Author a1 ON a1.id = w1.author_id AND a1.birth_year < 2000
JOIN Authorship w2 ON w2.book_id = b.id
JOIN Author a2 ON a2.id = w2.author_id AND a2.country = 'France';
```

```sql
SELECT b.id, b.title
FROM Book b
WHERE EXISTS (
    SELECT 1
    FROM Authorship w1
    JOIN Author a1 ON a1.id = w1.author_id
    WHERE
        w1.book_id = b.id AND
        a1.birth_year < 2000
)
AND EXISTS (
    SELECT 1
    FROM Authorship w2
    JOIN Author a2 ON a2.id = w2.author_id
    WHERE
        w2.book_id = b.id AND
        a2.country = 'France'
);
```

Next, in a new chat, we provide the LLM, in addition to the schema and query, with the verbalization of the conceptual schema containing explicit cardinality constraints.

Prompt: We are given the relational schema:

```
CREATE TABLE Author(
    id INT PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    birth_year INT,
    country TEXT
);

CREATE TABLE Book(
    id int PRIMARY KEY,
    title TEXT,
    category TEXT
);

CREATE TABLE Authorship(
    book_id INT PRIMARY KEY REFERENCES Book(id),
    author_id INT REFERENCES Author(id)
);
```

The database satisfies also the following conceptual schema:

```
'Book' is an entity.
'id' is an attribute of the entity 'Book'.
'title' is an attribute of the entity 'Book'.
'category' is an attribute of the entity 'Book'.
'year' is an attribute of the entity 'Book'.
'Author' is an entity.
'id' is an attribute of the entity 'Author'.
'first_name' is an attribute of the entity 'Author'.
'last_name' is an attribute of the entity 'Author'.
'birth_year' is an attribute of the entity 'Author'.
'country' is an attribute of the entity 'Author'.
'Authorship' is an relationship.
Every instance of the entity 'Book' participates in exactly one instance
    of the relationship 'Authorship' through the role 'is_written_by'.
Every instance of the entity 'Author' participates in zero or more instances
    of the relationship 'Authorship' through the role 'wrote'.
```

Optimize the following SQL query using the knowledge from the conceptual schema.

```
SELECT b.id, b.title
FROM Book b JOIN
    Authorship w1 ON w1.book_id = b.id JOIN
    Author a1 ON a1.id = w1.author_id JOIN
    Authorship w2 ON w2.book_id = b.id JOIN
    Author a2 ON a2.id = w2.author_id
WHERE
    a1.birth_year < 2000 AND
    a2.country = 'France';
```

GPT-4o:
We can optimize the query significantly by leveraging the conceptual schema constraint:
Every instance of the entity 'Book' participates in exactly one instance of the relationship 'Authorship'.

```
Implication:

This means:

    Each Book has only one Authorship tuple.

    Therefore, each Book has only one Author.

Current Query Problem:

The current query joins Authorship and Author twice, assuming multiple authors per book,
which contradicts the conceptual schema. It's trying to find books where:

    a1.birth_year < 2000

    and a2.country = 'France' - on the same book

But since there's only one author per book, this condition is equivalent to asking:

    Find all books whose single author has birth_year < 2000 and country = 'France'.
```

Optimized Query:

```sql
SELECT b.id, b.title
FROM Book b
JOIN Authorship w ON w.book_id = b.id
JOIN Author a ON a.id = w.author_id
WHERE
    a.birth_year < 2000 AND
    a.country = 'France';
```

```
Benefits:

    Fewer joins: goes from 5-table join to 3-table join.

    Semantically equivalent, since each book has exactly one author.

Let me know if you want this logic applied in a query optimizer explanation or rewritten
in relational algebra.
```

A.3 Schema normalization

In this subsection, we would like to check whether LLMs can use the information in conceptual schemas to normalize a relational schema originally provided in a denormalized form (a single "universal" table joining all relationships from a star-shaped schema). We first present the LLM with the original schema only, without the conceptual schema.

Prompt: We are given a database whose schema is defined by the statements:

```sql
CREATE TABLE Employment(
    employee_id INT,
    department_id INT,
    employee_name TEXT,
    department_name TEXT,
    department_desc TEXT,
    manager INT,
    PRIMARY KEY(employee_id, department_id)
);
```

Give a normalized schema for this database.

GPT-4o:

```
CREATE TABLE Employee(
    employee_id INT PRIMARY KEY,
    employee_name TEXT
);

CREATE TABLE Department(
    department_id INT PRIMARY KEY,
    department_name TEXT,
    department_desc TEXT,
    manager INT -- could also be a foreign
                -- key to Employee if manager is an employee
);

CREATE TABLE Employment(
    employee_id INT,
    department_id INT,
    PRIMARY KEY(employee_id, department_id),
    FOREIGN KEY (employee_id) REFERENCES Employee(employee_id),
    FOREIGN KEY (department_id) REFERENCES Department(department_id)
);
```

We removed irrelevant summaries and explanations from the answer returned by the LLM. The relational schema provided by the LLM is correct if we expect `manager` to be the unique manager of a department, but would not fit some other cases. The unicity and referential integrity constraints on the input (denormalized) relational schema cannot clarify the relationship between and its manager. We next issued (in another chat) the same task with an additional conceptual schema which in particular specifies that we aim to model a relationship between an employee and its possibly multiple managers.

Prompt:
We are given a database whose schema is defined by the statements:

```
CREATE TABLE Employment(
    employee_id INT,
    department_id INT,
    employee_name TEXT,
    department_name TEXT,
    department_desc TEXT,
    manager INT,
    PRIMARY KEY(employee_id, department_id)
);
```

```
The database also satisfies the following conceptual schema:

'Employee' is an entity.
'employee_id' is an attribute of the entity 'Employee'.
'employee_name' is an attribute of the entity 'Employee'.
'Department' is an entity.
'department_id' is an attribute of the entity 'Department'.
'department_name' is an attribute of the entity 'Department'.
'description' is an attribute of the entity 'Department'.
'Employment' is an relationship.
'Management' is an relationship.
Every instance of the entity 'Employee' participates in one or more
    instances of the relationship 'Employment' through the role 'works_in'.
Every instance of the entity 'Department' participates in one or more
    instances of the relationship 'Employment' through the role 'employs'.
Every instance of the entity 'Employee' participates in one or more
    instances of the relationship 'Management' through the role 'managed_by'.
Every instance of the entity 'Employee' participates in one or more
    instances of the relationship 'Management' through the role 'manages'.

Give a normalized schema for this database.
```

GPT-4o:

```
-- Employee Entity
CREATE TABLE Employee (
    employee_id INT PRIMARY KEY,
    employee_name TEXT
);

-- Department Entity
CREATE TABLE Department (
    department_id INT PRIMARY KEY,
    department_name TEXT,
    department_desc TEXT
);

-- Employment Relationship (works_in)
CREATE TABLE Employment (
    employee_id INT,
    department_id INT,
    PRIMARY KEY (employee_id, department_id),
    FOREIGN KEY (employee_id) REFERENCES Employee(employee_id),
    FOREIGN KEY (department_id) REFERENCES Department(department_id)
);

-- Management Relationship (manages)
CREATE TABLE Management (
    employee_id INT,     -- Employee being managed
    department_id INT,   -- Context of department
    manager_id INT,      -- Manager employee
    PRIMARY KEY (employee_id, department_id),
    FOREIGN KEY (employee_id, department_id)
        REFERENCES Employment(employee_id, department_id),
    FOREIGN KEY (manager_id) REFERENCES Employee(employee_id)
);
```

```
###  Notes:

* We **split** `Employment` and `Management` into separate tables
    to reflect their roles as relationships.
* `Management` is modeled with a foreign key to `Employment`
    to ensure the context of the department is preserved.
* Each `employee_id` can be managed by a `manager_id` **within a department**.
```

Again, we removed irrelevant summaries and explanations from the answer returned by the LLM. The LLM is able to leverage the conceptual schema and returns a normalized schema that allows an employee to have multiple managers. We observe that the LLM chose to add a department attribute to the management relationship, an information that we had not explicitly represented in our conceptual schema.

B Appendix: translations from KG-ER to other schema formalisms

In this section, we show how we can express the KG-ER conceptual schema of our running example (inspired by LDBC-SNB) into other schema formalisms.

B.1 Our running example

We first recap and complete our running example. We start from the shape graph from Figure 1 (recalled below), then add the constraints in the paper (except the type hierarchy that we omit from this discussion for simplicity), and finally add or strengthen a few constraints to obtain a well-formed schema:

The constraints in our example are the following:

Participation: We simply collect the participation constraints from our running example, and for simplicity require that every attribute but *Person(email)* is single-valued:

- $\textsc{Mandatory}(\mathit{Message}, \mathit{date})$
- $\textsc{Single}(\mathit{Message}, \mathit{date})$ and, more generally $\textsc{Single}(X, A)$ for every X, A except the pair $(X = \mathit{Person}, A = \mathit{email})$ since we want to allow persons to have multiple emails but want the other attributes to be single-valued.

In addition we have:

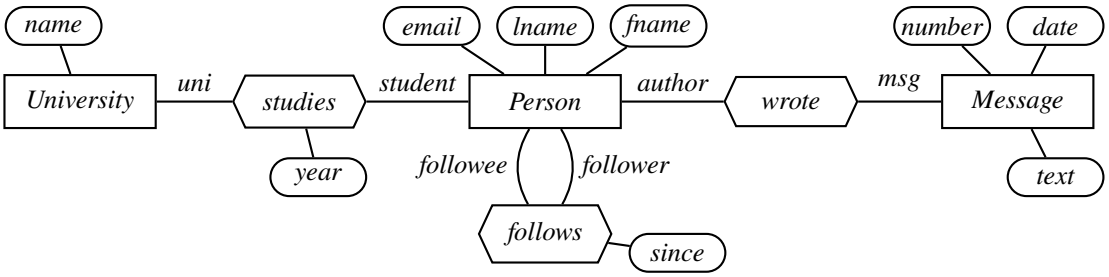

Figure 4: Shape graph inspired by the LDBC-SNB schema.

- MANDATORY(*Message*, *msg*, *wrote*)
- SINGLE(*Message*, *msg*, *wrote*).

Keys: We not only collect the participation constraints from our running example, but also complete and strengthen keys in order to satisfy $\mathbf{WF}_4$ and $\mathbf{WF}_6$ (every relationship – resp. "root" entity – has an identifying key):

- IDENTITY(*University*, [*name*])
- IDENTITY(*Person*, [*fname*, *lname*])
- KEY(*Person*, [*email*])
- IDENTITY(*Message*, [*msg*(*author*(*fname*, *lname*)), *number*])
- IDENTITY(*studies*, [*uni(name)*, *student(fname, lname)*, *year*])
- IDENTITY(*follows*, [*follower(fname, lname)*, *followee(fname, lname)*])
- IDENTITY(*wrote*, [*author(fname,lname)*, *msg(number)*])

When translating this KG-ER schema to other schema formalisms we will for simplicity consider that every attribute has a textual data type.

B.2 KG-ER to relational schemas

In this section we present a translation of our running example KG-ER schema into a relational schema, expressed using SQL.

The translation is similar in essence to the translation from an Entity-Relationship conceptual schema to an SQL relational schema. Every entity type is mapped to a distinct relation. In our example, this is the case for *University*, *Person*, *Message*. In general, every relationship type is also mapped to a distinct relation, but some relationships can be absorbed into the participating entities depending on the constraints: *studies* and *follows* are expressed using a relation, but *wrote* is absorbed into the *Message* relation. Single-valued attributes such as *Person(fname)* are inserted into the relation which represents their parent entity or relationship, whereas a distinct relation is created for each set-valued attribute such as *Person(email)*, containing the attribute and the identifying key of its parent attribute. The constraint KEY(*Person*, [*email*]) which prevents two persons from having any common email is expressed in our relational schema by the unicity constraint on *Emails(email)*. On our example, all KG-ER constraints translate directly into simple SQL integrity constraints. But triggers could have been required to express more complex KG-ER constraints, for instance if the patterns involved had connected distant entities through a long path.

```
CREATE TABLE University(
    name text,
    PRIMARY KEY (name)
);

CREATE TABLE Person(
    fname text,
    lname text,
    PRIMARY KEY (fname, lname)
);

-- Relation that records the set-valued attribute email:
CREATE TABLE Emails(
    email text,
    fname text,
    lname text,
```

```
    FOREIGN KEY (fname, lname) REFERENCES Person(fname, lname),
    PRIMARY KEY (email, fname, lname),
    UNIQUE (email)
);

/*
Relation that records the `Message` entity
together with its identifying relationship `works`:
*/
CREATE TABLE Message(
    author_fname text,
    author_lname text,
    number text,
    date text NOT NULL,
    msg_text text,
    FOREIGN KEY (author_fname, author_lname) REFERENCES Person(fname, lname),
    PRIMARY KEY (author_fname, author_lname, number)
);

CREATE TABLE Studies(
    university_name text,
    student_fname text,
    student_lname text,
    year text,
    FOREIGN KEY (university_name) REFERENCES University(name),
    FOREIGN KEY (student_fname, student_lname) REFERENCES Person(fname, lname),
    PRIMARY KEY (university_name, student_fname, student_lname, year)
);

CREATE TABLE Follows(
    follower_fname text,
    follower_lname text,
    followee_fname text,
    followee_lname text,
    since text,
    FOREIGN KEY (follower_fname, follower_lname) REFERENCES Person(fname, lname),
    FOREIGN KEY (followee_fname, followee_lname) REFERENCES Person(fname, lname),
    PRIMARY KEY (follower_fname, follower_lname, followee_fname, followee_lname)
);
```

B.3 KG-ER to SHACL

We present here a mapping of the running example KG-ER schema into SCHACL core, so without using SPARQL. This maps entities to nodeshapes, relationships to nodeshapes (unless they are identifying relationships such as *wrote*, which are mapped to properties) and attributes to properties.

The resulting SCHACL schema fails to model fully the following constraints:

- Identity(*University*, [*name*])
- Identity(*Person*, [*fname*, *lname*])
- Key(*Person*, [*email*])
- Identity(*Message*, [*msg*(*author*(*fname*, *lname*)), *number*])
- Identity(*studies*, [*uni(name)*, *student(fname, lname)*, *year*])
- Identity(*follows*, [*follower(fname, lname)*, *followee(fname, lname)*])

Our schema features two unique keys on a single attribute: Identity(*University*, [*name*]) and Key(*Person*, [*email*]). Such keys can in our case be expressed in SHACL using *maxCount* on the reversed edge, thanks to the global uniqueness of attribute names (Section 2.1). Without this global uniqueness assumption, the unique keys on a single attribute could still be expressed in core SHACL using *qualifiedMaxCount* on *qualifiedValueShape* over the reversed edge, thus checking for instance that each name has a single parent entity of class University (though it may then have other parent entities of other classes). On the other hand, it seems it may not be possible to express composite unique keys in SHACL core [Labra Gayo et al., 2017, End of Section 7.11], so we did not translate the other key constraints such as Identity(*Person*, [*fname*, *lname*]), though we could have expressed them using SPARQL.

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/> .
```

```
#################################################################
# University
#################################################################
ex:UniversityShape
    a sh:NodeShape ;
    sh:targetClass ex:University ;
    sh:property [
        sh:path ex:name ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] .

#################################################################
# Unicity of Name globally (hence of University names)
#################################################################
ex:NameTargetShape
    a sh:NodeShape ;
    sh:targetObjectsOf ex:name ;  # all nodes that are the object of :name
    sh:property [
        sh:path [ sh:inversePath ex:name ] ;
        sh:maxCount 1 ;
    ] .

#################################################################
# Person
#################################################################
ex:PersonShape
    a sh:NodeShape ;
    sh:targetClass ex:Person ;
    sh:property [
        sh:path ex:fname ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:lname ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:email ;
        sh:datatype xsd:string ;
        sh:minCount 0 ;    # Optional
    ] ;
    sh:property [
        sh:path ex:studies ;
        sh:node ex:StudiesRelShape ;
        sh:minCount 0 ;
    ] ;
    sh:property [
        sh:path ex:follows ;
        sh:node ex:FollowsRelShape ;
        sh:minCount 0 ;
    ] ;
    sh:property [
        sh:path ex:wrote ;
        sh:node ex:MessageShape ;
        sh:minCount 0 ;
    ] .

#################################################################
# Unicity of emails globally (hence of Person's emails)
#################################################################
ex:EmailTargetShape
    a sh:NodeShape ;
    sh:targetObjectsOf ex:email ;  # all nodes that are the object of :email
    sh:property [
```

```
        sh:path [ sh:inversePath ex:email ] ;
        sh:maxCount 1 ;
    ] .

#################################################################
# Message (Weak Entity)
#################################################################
ex:MessageShape
    a sh:NodeShape ;
    sh:targetClass ex:Message ;
    sh:property [
        sh:path ex:number ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:date ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:text ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:writtenBy ;
            # derived from identifying relationship "wrote"
        sh:class ex:Person ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] .

#################################################################
# studies relationship node (with attribute "year")
#################################################################
ex:StudiesRelShape
    a sh:NodeShape ;
    sh:property [
        sh:path ex:year ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:university ;
        sh:class ex:University ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] .

#################################################################
# follows relationship node (with attribute "since")
#################################################################
ex:FollowsRelShape
    a sh:NodeShape ;
    sh:property [
        sh:path ex:since ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] ;
    sh:property [
        sh:path ex:followee ;
        sh:class ex:Person ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
```

```
    ] .
```

B.4 KG-ER to ShEx

We present here a mapping of the running example KG-ER schema into ShEx. The mapping is broadly speaking the same as the one for SHACL. So we again map entities to nodeshapes, relationships to nodeshapes (unless they are identifying relationships such as *wrote*, which are mapped to properties) and attributes to properties. The result again does not fully represent all constraints of the KG-ER schema, and fails to represent the same list of constraints. Also here it is as far as we are aware not straightforward to represent in general such constraints correctly in the current version of ShEx at the time of writing, although extensions are under development that will make this possible, and in particular to express the key constraints.

```
PREFIX ex: <http://example.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

# --------------------
# Entity: University
# --------------------
ex:UniversityShape {
  ex:name xsd:string          # exactly 1 name
}

# --------------------
# Entity: Person
# --------------------
ex:PersonShape {
  ex:fname xsd:string ;       # exactly 1
  ex:lname xsd:string ;       # exactly 1
  ex:email xsd:string* ;      # 0..* emails
  ex:studies @ex:StudiesRelShape* ;   # 0..* studies relationships
  ex:follows @ex:FollowsRelShape* ;   # 0..* follows relationships
  ex:wrote @ex:MessageShape*      # 0..* messages written
}

# --------------------
# Entity: Message (weak entity)
# Identified relative to a Person through 'wrote'
# --------------------
ex:MessageShape {
  ex:number xsd:string ;      # partial key (relative to Person)
  ex:date   xsd:string ;
  ex:text   xsd:string
}

# --------------------
# Relationship: studies (Person <-> University, with attribute year)
# --------------------
ex:StudiesRelShape {
  ex:year xsd:string ;
  ex:university @ex:UniversityShape
}

# --------------------
# Relationship: follows (Person <-> Person, with attribute since)
# --------------------
ex:FollowsRelShape {
  ex:since xsd:string ;
  ex:followee @ex:PersonShape
}

# --------------------
# Relationship: wrote (Person <-> Message)
# --------------------
# In ShEx we just model this via the property ex:wrote in PersonShape.
# Each Message is connected back to its Person.
```

B.5 KG-ER to PG-Schema

In this section we present a translation of our running example KG-ER schema into a PG-Schema graph type as presented in Angles et al. [2023]. The translation basically maps entities to node types and (binary) relationship to edge types. The exceptions are the node type `emailType` and the edge type with label `hasEmail`, which model the multi-valued attribute *email* of entity *Person*. The KG-ER schema constraints map straightforwardly to similarly named constraints. Note that in some cases, such as for MANDATORY(*Message*, *msg*, *wrote*) and SINGLE(*Message*, *msg*, *wrote*), they are combined into a single PG-Schema constraint.

```
CREATE GRAPH TYPE socialStudentGraphType STRICT {
   // Node types
  (universityType: University {name STRING),
  (personType: Person {lname STRING, fname STRING}),
  (emailType: EMail {address STRING}),
  (messageType: Message {number STRING, date STRING, text STRING}),

  // Edge types
  (:personType)-[:hasEMail]->(:emailType),
  (:personType)-[:studies {year: STRING}]->(:universityType),
  (:personType)-[:follows {since: STRING}]->(:personType),
  (:personType)-[:wrote]->(:messageType),

  // Node identifiers
  FOR (u:universityType)
    IDENTIFIER u.name,
  FOR (p:personType)
    IDENTIFIER p.fname, p.lname,
  FOR (m:messageType)
    IDENTIFIER p.fname, p.lname, m.number WITHIN (p) -[:wrote]-> (m),

  // Edge identifiers
  FOR ()-[s:studies]->()
    IDENTIFIER u, p, s.year WITHIN (p)-[s:studies]->(u),
  FOR ()-[f:follows]->()
    IDENTIFIER p1.fname, p1.lname, p2.fname, p2.lname
    WITHIN (p1)-[f]->(p2),
  FOR ()-[w:wrote]->()
    IDENTIFIER p.fname, p.lname, m.number WITHIN (p)-[w]->(m),

  // Key and participation constraints
  FOR (m:messageType)
    MANDATORY SINGLETON w WITHIN ()-[w:wrote]->(m),
  FOR (p:personType)
    EXCLUSIVE e WITHIN (p)-[:hasEMail]->(e),
}
```