GRAPH DATABASE THEORY

Comparing Graph and Relational Data Models

Sridhar Ramachandran

Contents

Introduction

Relational Data Model

The relational data model has long maintained its supremacy over other database models because of its general-purpose nature. Specifically, there are three pillars that support the relational data model:

1. **Expressive power:** It is well known that conceptual models, such as the entity-relationship model and UML class diagrams (with some limitations) can be converted to relational schemas. Such methods are integral to the subject of relational database design.
2. **Strong design guidelines, aka normalization:** A relational schema derived from a higher-level model such as an ER or UML diagram can be normalized using well-defined rules. These rules provide data model designers with useful guidelines on developing schemas.
3. **A powerful query language, aka SQL:** SQL, the query standard for relational databases, can convey any query expressible in first-order logic. The provably expressive power of SQL is a key strength of the relational model.

Graph databases

The supremacy of the relational model (and SQL) has been challenged recently by the NoSQL movement, for various reasons, most notably better performance. The ***property graph model***, which is supported by most graph databases, is one of the non-relational data models in the NoSQL movement.

This document shows that the property graph model can match relational databases in terms of its expressive power, design guidelines and query methods.

Graph Schemas

Selecting vertex labels

The Tinkerpop property-graph model can be summarized as follows. A graph has a set of vertices and a set of edges. Each edge connects an out-vertex to an in-vertex. Vertices and edges can have properties which are key-value pairs with String keys and pretty much any value that the underlying database supports.

So far, the model looks schema-less since vertices and edges can't be distinguished from other vertices and edges without knowing what the properties *mean*. However, edges have always had labels. And with Tinkerpop 3, vertices will have labels as well. The same is true with Neo4J's latest major version.

If every vertex *must* be labeled, what is the correct method to select a label? What should a label say about a vertex or an edge, from the application's perspective?

We think a vertex label should represent the ***most granular type*** of the vertex, where each "vertex type" is associated with a ***unique combination of:***

1. meaning (semantics),
2. set of property key names and value types, and
3. set of outgoing edge labels, where each label type is annotated with the possible directions of the edge (in/out/both) and cardinality.

Why so? Because labels representing vertex types give the application the most detailed information about the *behavior* of that vertex, thereby ensuring that the application can process the vertex accordingly. In other words, one should not be able to sub-divide a vertex type to get two vertex types that behave differently from the application's standpoint.

Examples of label selection

Let's go through the label-selection exercise with the classic 6-vertex tinker graph shown in the property-graph model page. Since this is a Tinkerpop-2 style graph, it doesn't have vertex labels. We'll now try to come up with the vertex labels by simply looking at the vertex behavior.

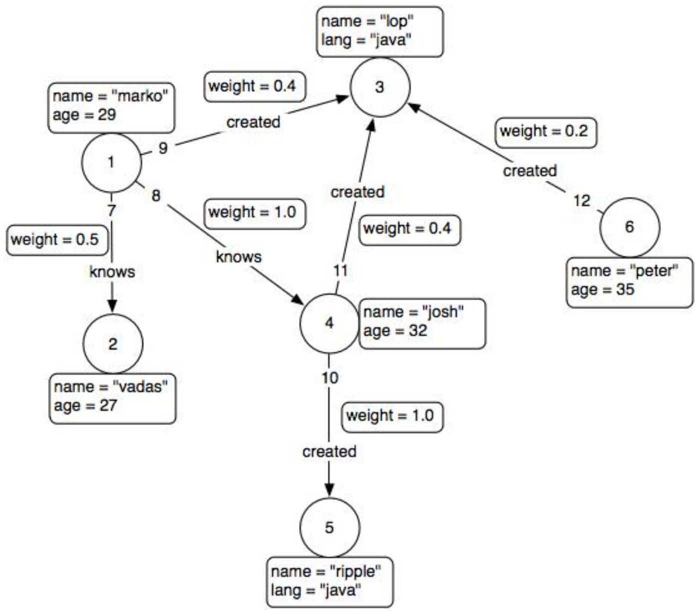


Figure 1: TinkerGraph example

If you look closely, there are two types of vertices: ones with 'name' and 'age', and ones with 'name' and 'lang'. Let us *label* the former vertex type as 'Person' and the latter vertex type as 'Software'. In other words, you have persons named 'marko', 'vadas', 'peter' and 'josh' and softwares named 'lop' and 'ripple'.

After analyzing the edge labels and direction, you could say that the 'Person' vertex type has:

- Property keys 'name' and 'age'
- Edges labeled 'knows' in the OUT direction
- Edges labeled 'created' in the OUT direction

The 'Software' vertex type has:

- Property keys 'name' and 'lang'
- Edges labeled 'created' in the IN direction

Now, an application looking at this graph automatically knows what to expect when it reads a vertex labeled 'Person' or 'Software'. We can define two different indexes on 'name', one for Person and one for Software, to *make sure that software searches don't pick up people*, or vice-versa.

The label selection process can't be fully mechanical though. For instance, a person with no friends can be thought of as a separate vertex type, because there are no adjacent 'knows' edges to such vertices. However, unless this makes sense in the context of the application or the data model, there is no point in sub-dividing the 'Person' vertex type as 'Loner' and 'Person with Friends'. The same argument goes for sub-dividing the person vertex type as the Developer' and 'Non-Developer' based on whether that person created a software.

To recap, the right way to select vertex labels for a property-graph is to first figure out the vertex types and the behaviors of each vertex type. *The totality of these behaviors is the graph schema.*

Drawing a graph schema

The best way to represent a graph schema is, of course, a graph. This is how the graph schema looks for the classic Tinkerpop graph.

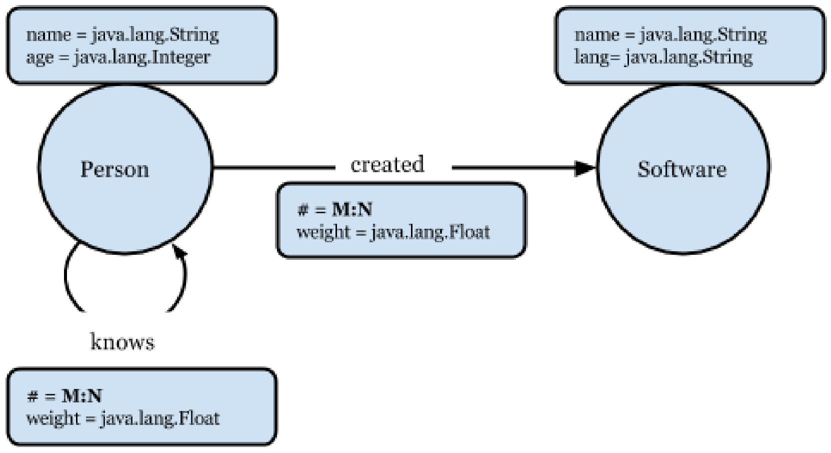


Figure 2: Example graph schema shown as a property graph

The graph schema is pretty much a property-graph. The vertices correspond to vertex types, and edges correspond to edge types. The property keys are named after the allowed property keys for that vertex type. Every property value in the schema graph contains the name of the *most-specific super-class* representing the corresponding property values of the instance graph. Optional properties can have a '?' after the class name (not shown here).

Edge properties are like vertex properties, except that there is a special property named '#' that holds the ***cardinality*** from the out to in vertex types. Common sense dictates that the cardinality is M:N, i.e., many-to-many, for both 'knows' and 'created'. One could be misled to think that some of these relationships are 1:N by looking at the 6-vertex graph. This is another reason for not fully relying on reverse-engineering methods to derive schemas.

We have gone through a similar exercise for the Grateful Dead graph. As you can see, the graph schema is very simple, although the visualization of the graph shown in the link looks complicated.

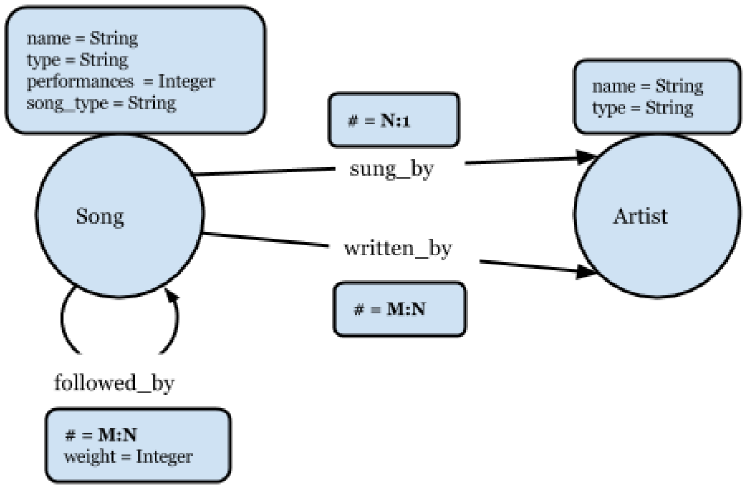


Figure 3: Grateful Dead graph schema

Our third and final example is the schema for the Kennedy family tree graph. Again, the schema is extremely simple (simplistic given recent US Supreme Court rulings).

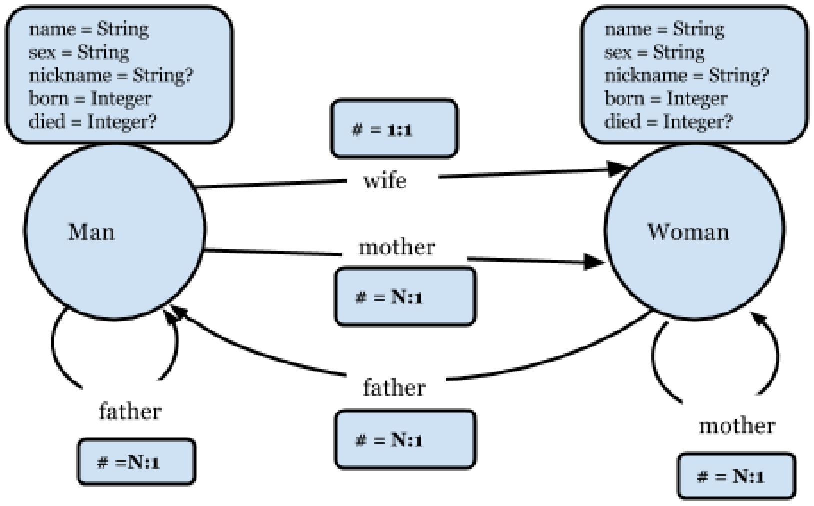


Figure 4: Family tree graph schema

Note that in the Pixy schema, the property lists are the same for 'Man' and 'Woman', but the direction of the 'wife' edge is functionally dependent on the value of the 'sex' property. This is very interesting because this means that *graph schemas could be normalized* using rules like relational databases. We will discuss this in later sections!

Summary

This section introduced the idea of schemas for property graphs and described how the schema itself can be represented as a property graph. Furthermore, it described a method to *derive* the graph schema for an existing property graph by finding the most granular division of its vertices into vertex types.

Graph schemas (or schema graphs) help application developers better understand the graph's structure.

In the next section, we will look at the problem the other way around. Can we *derive* a graph schema from a higher-level conceptual model such as an Entity-Relationship model? Could this be a systematic method to select vertex and edge labels, and property keys when designing a graph database application?

Converting ER models to graph schemas

This section will describe a general method to convert an entity-relationship model to a property-graph schema. Using this method, a database designer can develop ER models using standard conceptual modeling practices, but store the data in a graph database instead of a relational database.

ER models and diagrams

The entity-relationship model was proposed by Peter Chen in his 1976 paper titled "The Entity-Relationship Model--Toward a Unified View of Data". The ideas in this paper are taught in most database courses. This course page gives a quick description of the ER model.

Conceptual modeling is a particularly useful exercise when embarking on a project that involves a new domain. The goal of this exercise is to identify key concepts in the domain that must be captured in the data model. One of the techniques in conceptual modeling is to look at the natural language description of an application's requirements. These requirements can be analyzed to identify the entity and relationship types, using Chen's "rules of thumb" (quoted from Wikipedia):

Common noun	**Entity type**
Proper noun	Entity
Transitive verb	Relationship type
Intransitive verb	Attribute type
Adjective	Attribute for entity
Adverb	Attribute for relationship

Example

Let us consider the following requirements:

> Model a system where users create pages, which they own. Users can invite other users to look at certain pages that they own. A page can specify one or more tags which are then used to recommend other sections to the authors and invited readers.

You could analyze this requirement and come up with three entity types, viz. User, Page and Tag. The relationship types Owns, Invites and Tagged-As capture the relationships. Note that all verbs don't become relationships (like create). Similarly, the fact that invitations only apply to pages that a user owns is lost in this model.

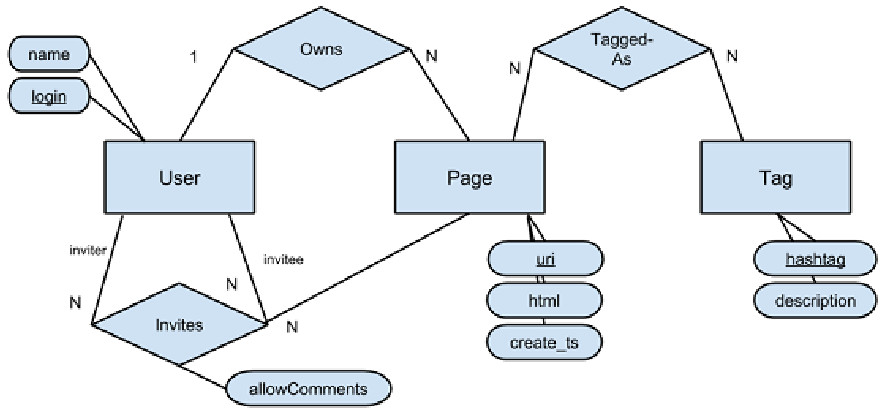


Figure 5: Example ER diagram

The square-shaped boxes show entity types, which represent sets of similar entities. The diamond-shaped boxes show relationship types, which represent sets of similar relationships. A relationship type relates two or more entity types to each other.

The diagram shows the cardinality of each entity's contribution to a relationship, such as 1:N (one to many) or N:N (many to many). The cardinality is specified using the 'look across' method. For example, a User owns N pages, and a page is owned by 1 user. There are known limitations of look-across cardinality for ternary relationships like Invites.

The diagram also shows some oval shaped attributes, like user name. These attributes must be assigned to entity or relationships types. Attributes that serve as external identifiers must be underlined.

Now, it is arguable whether Tag must be an entity or not in the final data model. But from an ER perspective, it makes sense to model tag as an entity, especially if tags are used to establish relationships across users for recommendations.

Procedure to convert an ER model to a graph schema

The procedure to convert an ER model to a relational model is well-known and discussed in the same OSU course notes that we referenced earlier. We will now go through a similar procedure the ER diagram with the above example.

Rule #1: Entity types become vertex types

Entity types such as User, Page and Tag become vertex types.

- The name of the entity type becomes the label of the vertex type.
- The associated attributes become the properties of the vertex type.

Here is an example showing User:

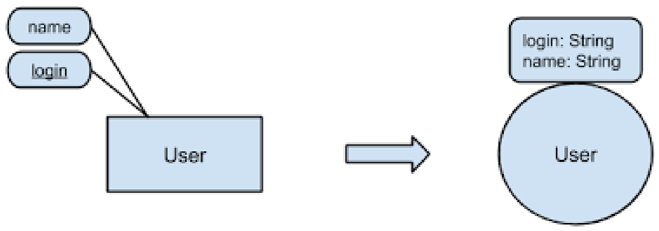


Figure 6: User entity converted to a user vertex type

Note that we are drawing a graph schema, not a graph instance. So the User type refers to any number of users in both the ER and the graph schema representation. Hence we use the term "vertex type" and not vertex. The entity-relationship model uses similar terms such as "entity types" (like User) and entities (like John Doe, the user).

Rule #2: Binary relationship types become edge types

All binary relationship types in the ER diagram can be converted to edge types in the graph schema.

- The name of the relationship type becomes the label of the edge type.
- The associated attributes become the properties of the edge type.
- The end-points of the edge type are the vertex types corresponding to the related entity types. The direction doesn't matter.

Here is an example showing the "Owns" relationship type translated to an "owns" edge type:

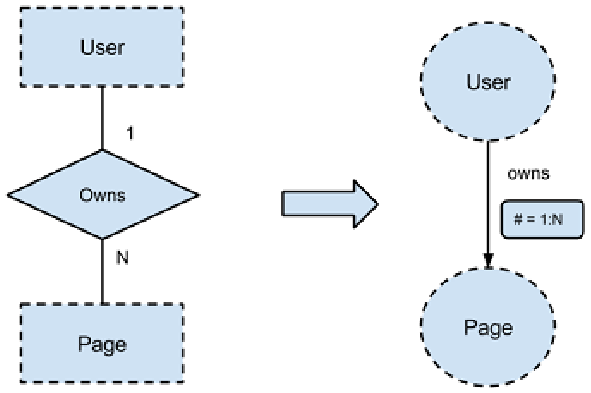


Figure 7: Owns relationship converted to an owns edge

Note that one-to-many and many-to-many binary relationships can be modeled as edges without introducing new vertices. With relational models, you would need an additional table to capture many-to-many relationships.

A minor point is that the cardinality is written as 1:N because the User (out vertex type) to Page (in vertex type) relationship is a 1:N relationship, using the look-across method. In other words, a user has N pages and a page has 1 user. If the direction of the edge were reversed, the cardinality would be N:1.

Rule #3: N-ary relationship types become vertex types

N-ary relationship types relate more than two entity types. Such relationship types become vertex types in the property-graph model.

- The name of the relationship type becomes the label of the vertex type.
- The associated attributes become the properties of the vertex type.
- The new vertex type includes edges to the vertex types corresponding to the related entity types (see example). These edge types are labeled after the role of the participating entity in the relationship. The direction doesn't matter for any of these edges.

Here is an example showing the ternary relationship Invites translated to the vertex type Invitation:

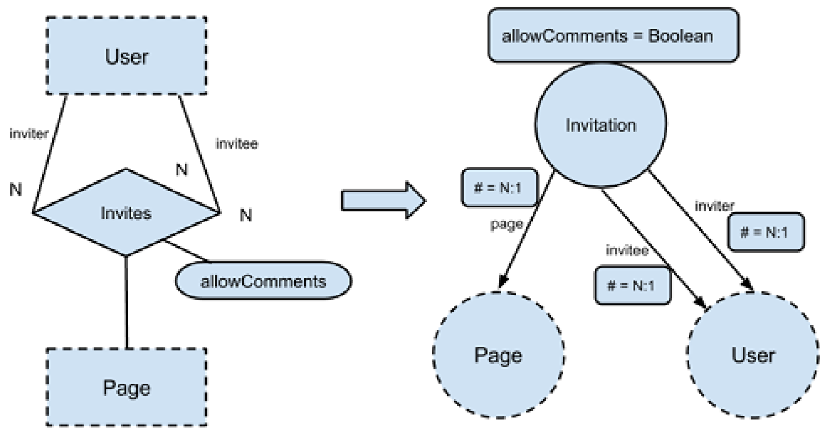


Figure 8: Invites relationship converted to an Invitation vertex type

The cardinality in the graph schema is N:1 because the Invitation to Page relationship is an N:1 relationship, using the look-across method. In other words, an invitation could be issued to 1 page, and a page (in vertex) could be part of N invitations. It is possible to just reverse some of the role types, like invitee, without affecting the overall model. In that case the cardinality will be 1:N.

We haven't shown the process for weak entity types and identifying relationship types -- but these are exactly the same as entity types and relationship types. Graph databases are more forgiving than relational databases in that they allow two vertices to have the same label and property key-value pairs. This simplifies the translation of weak entity types and identifying relationship types into the property-graph model.

Conversion example

Here is the graph schema corresponding to the example ER diagram. As you can see, this diagram provides enough information for an application developer to work with the graph database.

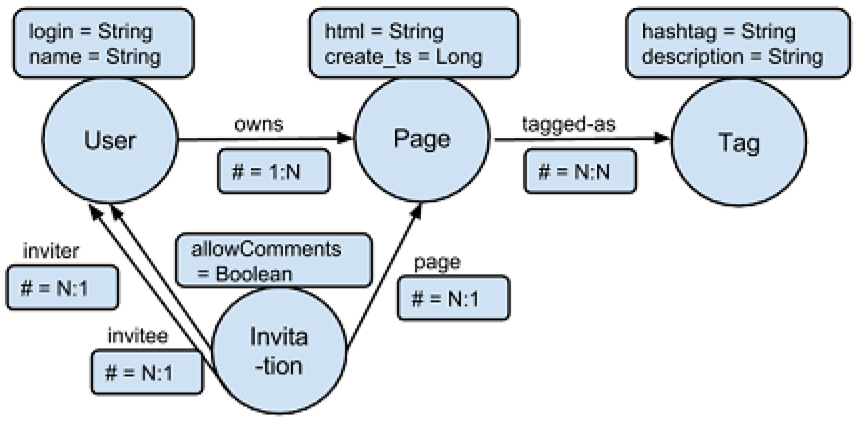


Figure 9: Graph schema for User-Page-Tag ER diagram

This is the "logical model" for the example conceptual model introduced in the first figure. We can tweak this model further by renaming the labels, changing directions of the edges, and so on. This will be the topic of the next section.

Vertices are vertices, and edges are ... edges

N-ary relationships are very common in conceptual models. For example, "Joe bought a headphone at Target" is an example of a "Bought" relationship that relates a User to a Product to a Store. Such relationships must be modeled as vertices, not edges (unless you are using hypergraphs). Hence we think it is *misleading* to think of edges as relationships and vertices as entities.

It is better to think of graphs are *visualize-able representations* of a conceptual model. We emphasize the visual nature of graphs because drawing and thinking in terms graphs is easy. For instance, you go to the Wikipedia entry for hypergraphs, you will see why visualizing hypergraphs isn't as easy as visualizing (binary) graphs.

Summary

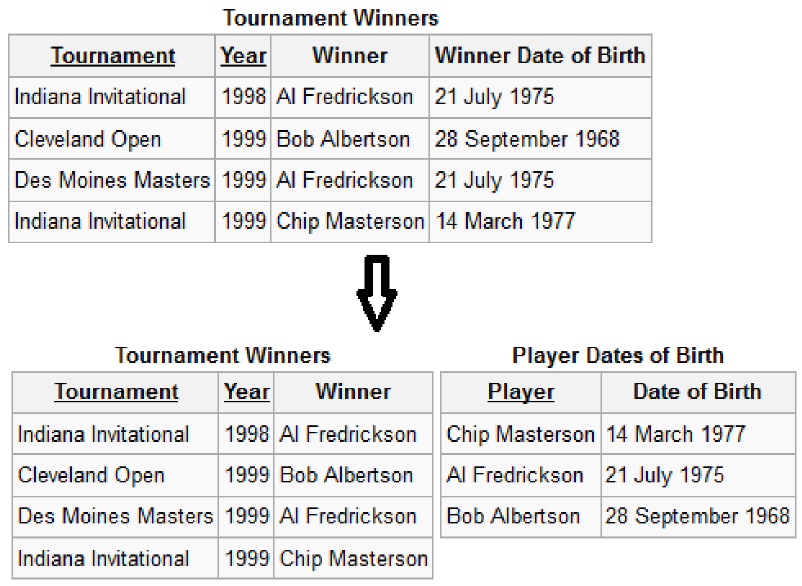
This section showed that it is possible to convert any entity-relationship model to a property-graph schema. In other words, a data architect can use standard methods to model a domain as an ER diagram and then follow this procedure to convert it to a property-graph schema. This type of a translation is not obvious for other popular NoSQL models like key-value stores and document stores.

Normalizing Graph Schemas

This section looks at how graph schemas can be manipulated and transformed to equivalent graph schemas. This is similar to the splitting and merging of tables in relational data models, typically performed to normalize or de-normalize a relational schema.

Normalization of relational databases

The goal of database normalization is make sure that relational schemas are easy to modify, easy to extend, informative to users and supportive of various query patterns. The various normal forms, such as 1NF, 2NF, and so on, define constraints that a table must satisfy to be compliant with that normal form. Although the definitions of the normal forms can be mathematical, the basic idea is break up tables with duplicate information. Here is an example from the Wikipedia page on 3NF:



Tournament Winners

Tournament	Year	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Tournament Winners

Tournament	Year	Winner
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Player Dates of Birth

Player	Date of Birth
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

The previous figure breaks up the tournament winners table into two tables, one with player details and one with the tournament details. The actual rules on "functional dependencies" and "non-prime attributes" are hard to remember, but the process of splitting and merging tables comes intuitively with experience. For example, if there was an existing table which had one row per player, we'd probably move the "date of birth" to that table.

Transformation rules that produce equivalent schemas

This section lists some transformation rules that produce equivalent graph schemas. A graph schema is *equivalent* to another graph schema if the data stored in one schema, along with the applications that access it, can be ported to the other schema, and vice-versa. These rules are like splitting and merging tables in relational models.

The transformation rules in this section can be mechanically applied to any schema, and has nothing to do with its semantics. By applying a combination of these rules, you could simplify the semantics and improve the usability of your graph model.

Rule A: Renaming properties and labels

This rule consists of three transformations that result in equivalent schemas:

1. Any vertex label can be renamed, so long as the new name doesn't refer to an existing vertex label.
2. Any edge label can be renamed, so long as the new name doesn't refer to an existing edge label between the out and in vertex types.
3. Any vertex/edge property can be renamed so long the new name doesn't refer to an existing property of the vertex/edge type.

The following figure illustrates some example applications of this rule on vertex and edge labels:

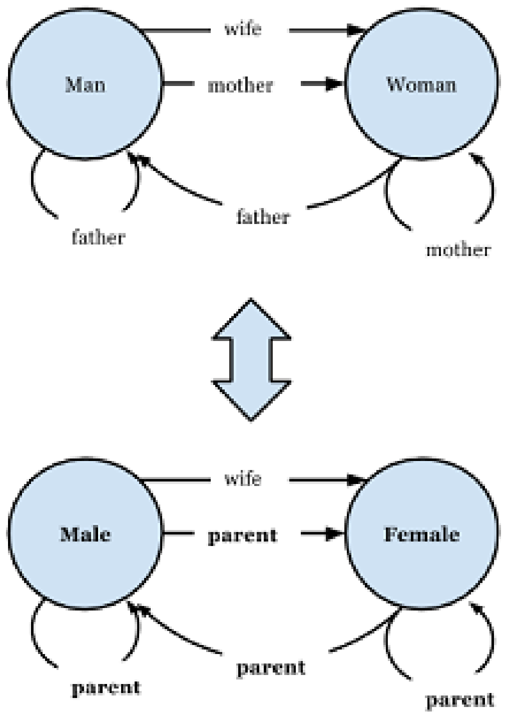


Figure 10: Renaming properties and labels

The schema shown in the top is a simple graph schema showing family relationships. This schema is transformed to the schema shown in the bottom of the figure using the following transformations:

- Vertex labels Man and Woman are renamed to Male and Female.
- Edge labels mother (2 instances), father (2 instances) are renamed to parent.

Even though it seems like some information is lost by renaming mother/father to parent, this isn't true because the vertex labels at the end points (Male/Female) have that information. This same transformation wouldn't be so obvious while looking at an instance of this graph like the Kennedy family tree.

Note that you cannot rename 'wife' to 'parent' in the bottom schema. This is because there already exists a parent edge type from Male to Female.

Rule B: Reversing edge directions

This rule states that an edge type can be reversed provided it is a self loop, or there is no edge type with the same label in the reverse direction. The cardinality of the edge type is reversed as well.

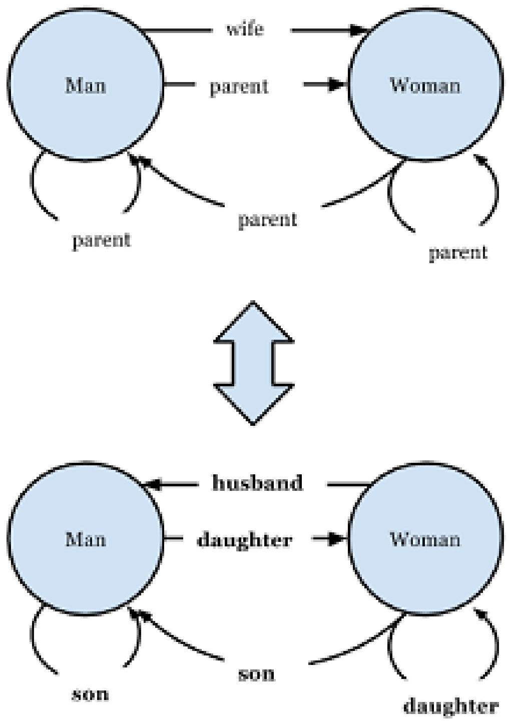


Figure 11: Reversing edge directions

The following figure illustrates an example transformation using this rule and the previous one. The transformation involves the following steps:

- The 'wife' edge is renamed to 'husband' (rule A) and then reversed.
- Each parent edge is renamed to 'son' or 'daughter' and reversed.

Note that the reversal is done in the graph instance as well as the schema. In other words, JFK Jr -parent-> JFK Sr. becomes JFK Sr. -son-> JFK Sr.

You could always rename the four 'son' and 'daughter' edge types, to 'child' using rule A. Again, no information is lost since the vertex labels are still unique.

You would, however, not be able to rename 'husband' to 'son'. You could rename 'husband' to 'daughter' (though absurd). The application will have to interpret "male daughters" as husbands. But after you rename husband to daughter, you would not be able to reverse its direction.

As you can see already, some applications of these rules may be quite hard to derive if you are thinking in terms of graph instances, rather than graph schemas.

Rule C: Property displacement

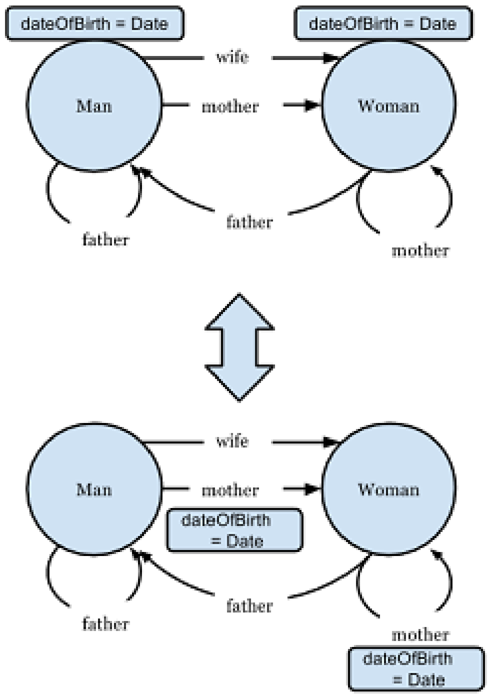


Figure 12: Property displacement

This rule states that a property on an edge type can be moved to either adjacent vertex type, provided its look-across cardinality is 1. The reverse rule states that a property in a vertex type can be moved to an adjacent edge type with look-across cardinality of 1, provided the edge always exists when the property exists.

The adjoining figure clarifies the rule, where the 'dateOfBirth' property is moved to the 'mother' relationship because there is exactly one mother relationship per Man/Woman and it is defined when the dateOfBirth is defined. If you rename 'dateOfBirth' to 'deliveryDate', one could argue that the property belongs in the edge and not the vertex.

Note that a Man's dateOfBirth can not be displaced to the wife relationship because that would mean that the dataOfBirth can not be stored unless the person is married. Similarly, the dateOfBirth in the edge type labeled 'mother' from Man to Woman in the bottom schema, can not be moved to Woman because of the cardinality restrictions in the rule.

Using this rule, you can move the properties around the schema to come up with a better-looking design. This rule is also useful in satisfying indexing requirements of various graph databases. For example, if a graph database only supports indexes on vertex properties, you could move searchable properties from the edges to vertices. Similarly, if a graph database supports vertex-centric indexes based on properties on adjacent edges/vertices, you can use this rule to bring the indexed property closer to the vertex type of interest.

Rule D: Specialization and generalization

This rule states that:

1. Any vertex type can be divided into two disjoint vertex types based on a Boolean test on the properties and adjacent edge labels of a vertex belonging to that type.
2. Any edge type can be divided into two disjoint edge types based on a Boolean test on the properties and adjacent vertex labels of an edge belonging to that type.

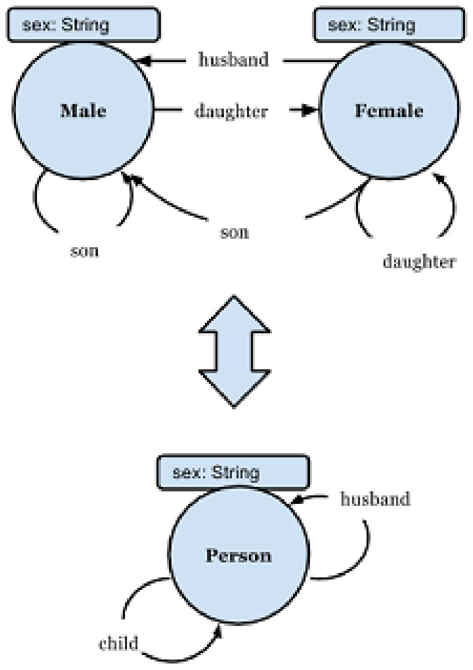


Figure 13: Generalization

In other words, if we provide a boolean function that can give a T/F result given a vertex/edge, we can use that function to divide a vertex/edge type into two different types.

The reverse rule states that:
Any vertex/edge type can be merged into another vertex/edge type provided there is a Boolean test that can distinguish its vertices/edges from the merged vertices.

The adjoining figure shows an example transformation involving the following steps:

- Male and Female are generalized as Person, because the boolean test, sex equals 'M', can distinguish Male from Female.
- After that, son and daughter edge types are generalized as child because the boolean test, sex of in-vertex equals 'M', can distinguish son from daughter.

This rule is useful in increasing the specificity, or reducing the complexity of the graph schema. As a general principle, it is better to use this rule for specialization, we.e., increasing the specificity, because that allows the different vertex and edge types to embrace different behavior in terms of properties and adjacent edges. However, there are instances where the differences between the vertex types are so minor that specialization only results in application complexity. This argument could apply to the above generalization of Male and Female to Person.

Rule E: Edge promotion

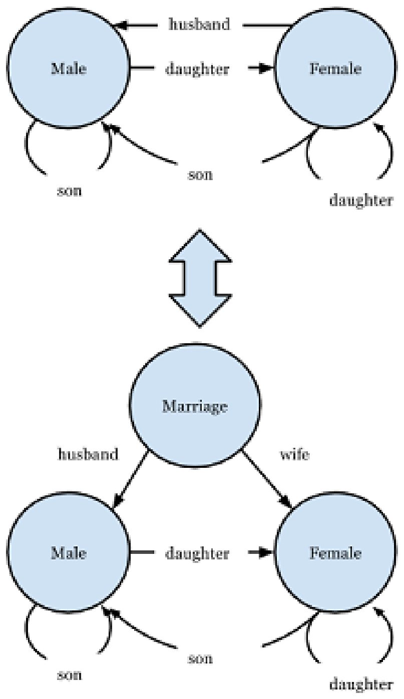


Figure 14: Edge promotion

This rule states that an edge type can be *promoted* to a vertex type by adding two "out" edge types to the end-points. The properties of the vertex type become properties of the edge type. The cardinality of the new edge types are N:1 or 1:1 depending on the look-across cardinality of the original endpoint vertex's type.

Note that the direction of the new edge types can be changed using on the rename and reverse rules resp. We only mention the "out" direction to simplify the way in which cardinality for the new edges types is derived.

The adjoining figure shows the husband edge promoted to a vertex type called 'Marriage'. The edge types 'husband' and 'wife' point to the two end-points of the vertex type.

The edge promotion rule is useful in a preparing binary relationship to become an N-ary relationship.

The reverse rule states that any vertex type with two property-less edge types, with same-side cardinality of exactly 1, can be *demoted* to an edge between the adjacent vertices. This process is useful to simplify schemas. You can use the property displacement rule (rule C) to move properties out of edges.

Rule F: Property promotion

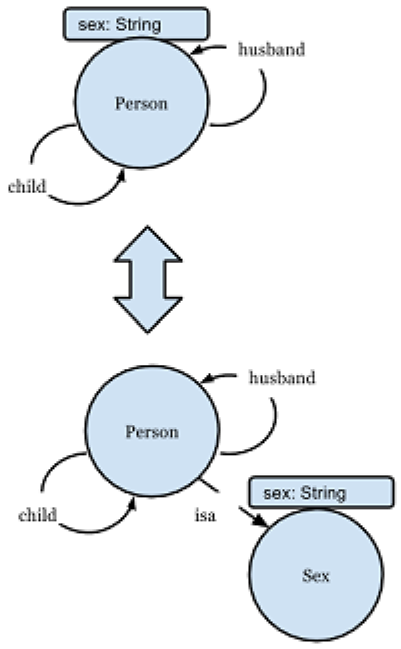


Figure 15: Property promotion

This rule states that any group of properties can be promoted to a new vertex type with those properties, provided the new vertex type has edges connecting it to all existing vertex types that include the property group. The same-side cardinality of the new edge type is 1.

The adjoining figure shows the 'sex' property converted to a new vertex type. This vertex type will have exactly two nodes corresponding to male and female. So in other words, every person in the new graph will have an outgoing 'isa' edge to one of the two new vertices.

This rule is equivalent to the splitting of a relation into two relations, as shown in the first figure of this section. Any group of properties, typically ones that repeat, can be promoted to a vertex.

While applying this rule, it is better to include all vertex types that have the same group of properties. For example, if there is a 'sex' property in a different Animal type, it is better to point that to the new Sex vertex type as well. If you have edge types with the property group, you can first promote those edge types to vertices.

The reverse of this rule is that a vertex type that has property-less edge types with same-side cardinality of 1, can be demoted to the group of properties that it holds. These properties must be added to every adjacent vertex type. This is the equivalent of the denormalization of a table in the relational model, which is useful to reduce the number of joins (or traversals in the case of graph databases).

Rule G: Property expansion

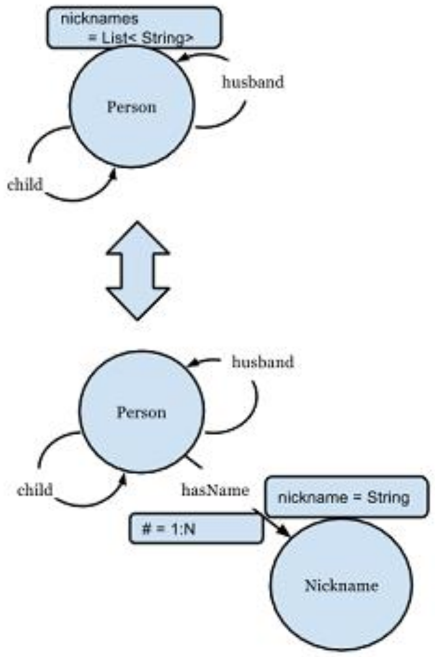


Figure 16: Property expansion

This rule states that a property of a vertex type that represents a list of values can be moved to a separate vertex type which stores each value. The new vertex type must have an "in" edge type from the existing vertex type with cardinality 1:N. The adjoining figure shows this rule applied to the nickname property which holds a list of Strings.

The reverse rule states that any vertex type with exactly one property-less edge type with look-across cardinality of exactly 1 can be removed after moving its properties to a list in the adjacent vertex type.

This is the equivalent of 1NF in the relational model. Unlike relational databases, however, many graph databases support lists as a valid type for property values. So the choice of storing nicknames as a List or a separate vertex type is up to the designer.

Summary

Rule-based schema transformations are tools that a data model designer can use to rewrite a graph schema, without losing any information in the process. In other words, a data model designer can use these rules to select the directions of edges, the names of different labels and keys, the locations of various properties, and so on. These changes don't matter from an pure information perspective, but could make a big difference in the usability and efficiency. In that sense, a data model designer can go back to Codd's original goals for normalization -- designing schemas that are easy to modify, easy to extend, informative to users and supportive of various query patterns.

One *meta-rule* for normalization

The previous section listed seven rule-based schema transformations such as renaming labels, reversing edges, promoting edges and properties to vertices, and so on. Such rule-based transformations can be mechanically applied to any graph schema, without losing any information in the process. Using these rules, a graph database designer can start with a design generated from an entity-relationship model and tweak it to get a final design.

This section describes a single *meta-rule* from which the seven previously-described rules can be derived. It also formalizes some of the ideas presented in the previous sections using set theory.

Schemas and constraints

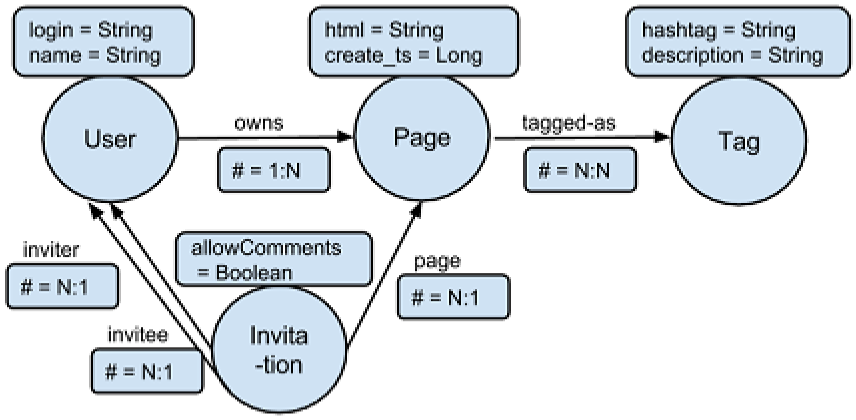


Figure 17: Example graph schema

The above figure shows an example graph schema describing constraints on the graph data model such as:

1. What are the legal labels for vertices?
2. What are the legal edge labels between two vertex types?
3. What are the legal property keys and value types at each edge or vertex type?

The reality, however, is that a graph model could have other constraints that aren't expressed in the schema. For example, the 'inviter' edge in every Invitation must be to the User who has an 'owns' edge to the 'page' edge of the Invitation. This constraint isn't captured in the above schema.

The question is: *How can we model complex constraints in a graph model?*

Graph universes, transformations and equivalence

A *graph universe* U is a set of graphs, typically an infinite set. A graph universe represents a data model in the sense that it captures every valid graph that belongs to the data model.

A graph universe U is *compatible* with a graph schema S, if every graph in the universe is compatible with S. In other words, although the graph universe is a precise description of the model, it can still be understood as a refinement of a more loosely-defined graph schema.

Redefining equivalence using transformation functions

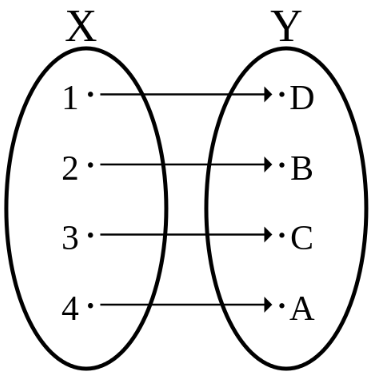


Figure 18: An invertible function

A graph transformation T is a function that takes graphs from one universe U to another universe V. In short, T : U → V.

A universe U is equivalent to a universe V if there is a transformation function T : U → V, where T is invertible. Invertible and bijective are terms to characterize functions that establish a one-to-one correspondence between two sets, which in this case are graph universes.

In other words, given any graph G ∈ U, we can use T(G) to get a graph G' ∈ V. Then we can use the inverse function T^{-1}(G') to get back G. Hence establishing equivalence of the two universes.

A programming perspective

If we are upgrading from one graph model to another, the transformation function is the *upgrade script* that we would implement to move to the new model. If we can also write a *downgrade script*, then we have two equivalent models (or universes). In other words, two graph models, represented as universes or schemas, are equivalent if they are *forward and backward compatible*.

Derived types

Consider a graph universe U that is compatible with a schema S. A vertex type in S can be called a *derived vertex type in U*, if every graph G ∈ U is such that its vertices (and adjacent edges) belonging to the vertex type can be calculated from the rest of the graph.

In other words, given any graph in the universe U, after we remove all vertices corresponding to the derived vertex type, there should be a way to calculate those vertices again. Derived edge and property types can be defined similarly. Note that all derived element types are defined in graph schemas, but are specific to graph universes that are compatible with that schema.

Meta-rule: Adding and removing derived types

Finally, here is the meta-rule behind all schema transformations:

> Given any graph universe U compatible with a schema S, we can add a derived vertex/edge/property type to produce an equivalent graph universe V compatible with the schema S ∪ {derived type}.

The reverse rule states that:

> Given any graph universe U compatible with a schema S, we can remove a derived vertex/edge/property type to produce an equivalent graph universe V compatible with the schema S - {derived type}.

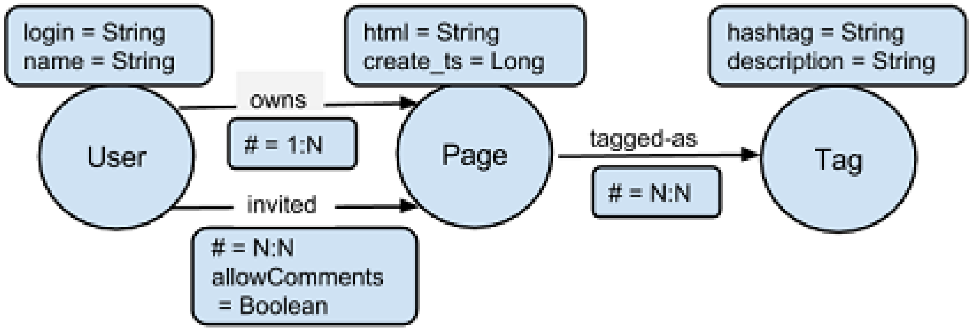


Figure 19: Modified graph schema

The 'invitee' edge type in the graph schema shown in the first figure is a derived edge type. This is because the 'invitee' edges can be calculated by going from the Invitation vertices to the Page and back to the user through 'owns' edge (reverse direction). We can simplify the original schema to the version shown in the adjoining figure by applying these rules:

1. (Meta-rule) Remove derived edge type 'invitee'
2. (Edge promotion) Demote the binary relationship Invitation to an edge called 'invited'.

As you can see, the updated schema is simpler than the original schema derived from an ER diagram.

Proving the meta-rule

The meta-rule is easy to prove because of the way derived types are defined. The transformation function to remove a derived type simply removes all elements that belong to that type. The inverse function calculates the derived types from the remaining graph. Hence the universe with the derived type is equivalent to the universe without it.

Proving the 7 rules: Renaming, Reversing, Property Displacement, ...

Consider the example of renaming an edge type. This rule was stated in the last section as:

Any edge label can be renamed, so long as the new name doesn't refer to an existing edge label between the out and in vertex types.

We can prove this in two steps:

1. Add derived edge type with the new name as a copy of the old edge type.
2. Remove the old edge type which is now derivable from the new edge type.

Of course, step 1 requires that the edge type with the new name doesn't already exist in the schema. Otherwise, all edges of the edge type can't be derived. Hence the condition "as long as the new name doesn't refer to an existing edge label."

In this manner, we can prove each rule by performing some steps to first add new derived types and then remove the existing types which become derived types themselves.

Beyond transformation rules

Thinking in terms of graph universes, derived types and transformation functions lets us do more radical transformations to our graph model. The manner with which we apply these rules or transformations depends on our overall strategy for data modeling.

One strategy is to minimize the number of implicit constraints not captured by the schema. For instance, the schema shown in the second figure doesn't have the implicit constraint on the 'invitee' edge type shown in the first figure. Generally, fewer implicit constraints means less duplication of data and less chance of bugs while updating the database. This is similar to normalization in relational databases.

A different strategy is to tune the graph for its specific querying needs. Such approaches have been popularized by "de-normalization" techniques such as dimensional modeling. For instance, we could add a "shortcut" derived edge type called 'latest' from User to Page to show the last-created page for each user. The important thing then is to ensure that any change to the rest of the graph is accurately reflected in the derived element types. The code that operates on the graph must be designed with these constraints in mind.

Summary

This section introduced set-theoretic representations of graph models called graph universes, which are more powerful than graph schemas. Secondly, this section showed that two graph universes are equivalent if there is an invertible graph-transformation function between them. Finally, this section showed that all schema transformation rules presented in the earlier section can be derived from one meta-rule that deals with adding and removing derived types.

Validating graph schemas

The last few sections have discussed how property-graph schemas can help design graph databases from ER models and refine the data model through schema manipulations. After reading this thread on the Gremlin users group, we realized that it is easy to validate graphs against schemas with Gremlin and Groovy.

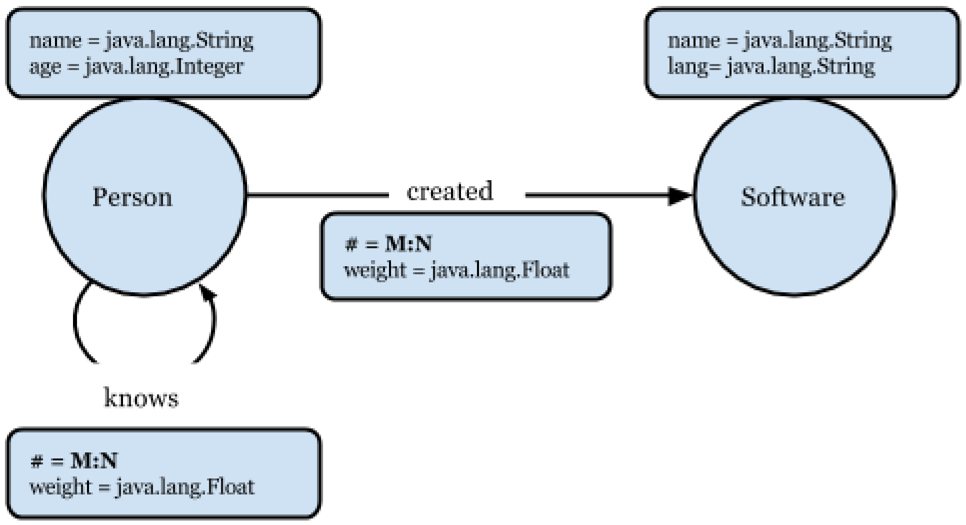


Figure 20: Tinkergraph schema

This gist on Github shows how you can take an instance graph and check to see if it is compatible with a schema graph. The schema graph has vertices and edges corresponding to vertex and edge types. Here's the code to create a schema graph inside a Gremlin shell for the classic Tinkerpop schema shown here:

```
sg = new TinkerGraph()
person = sg.addVertex()
person.setProperty('_label', 'person')
person.setProperty('name', 'java.lang.String')
person.setProperty('age', 'java.lang.Integer')

software = sg.addVertex()
software.setProperty('_label', 'software')
software.setProperty('name', 'java.lang.String')
software.setProperty('lang', 'java.lang.String')

knows = person.addEdge('knows', person)
knows.setProperty('weight', 'java.lang.Float')

created = person.addEdge('created', software)
created.setProperty('weight', 'java.lang.Float')
created.setProperty('_minIn', 1) // Someone must create the software
```

The properties have values corresponding to the Java Class of the property values in the instance graph. The property keys can end with '?' to indicate the property is optional. The edges in the schema graph can have 4 special properties, viz. _minIn, _maxIn, _minOut and _maxOut to indicate cardinality restrictions for various edge types.

Any instance graph, g, can be validated against the schema stored in sg, using the Gremlin script:

```
g.V.filter({checkVertex(it, sg) })
```

You can look at the full Github gist to see how the validation is done.

The current version of Tinkerpop doesn't support vertex labels. So the mapping from the vertex to the vertex type is specific to the graph, like this:

```
vertexType = {v, sg -> .age ? sg.V('_label', 'person').next() : sg.V('_label', 'software').next() }
```

Most graph schemas typically have a property named 'type' that would make this mapping easier. However with Tinkerpop3, this method can be standardized to use the label:

```
vertexType = {v, sg -> sg.V('label', v.label).next() }
```

Pixy: First-order logic on graph databases

The previous sections have shown that any ER model can be converted to a property graph schema, and that the schema can be normalized using rules. However, one key question remains:

Do graph databases offer the same querying capabilities as relational databases?

In other words, any data that fits in an E-R model can be stuffed into a graph database. But can data stuffed in this fashion be queried effectively? This is the subject of this section.

Background

On SQL

SQL is the query standard for relational databases. It first appeared in the 1970s and was standardized in the 80s and 90s. The theoretical foundation of SQL is relational algebra. Codd showed that relational algebra is equivalent to relational calculus, a form of first-order logic. His theorem is the bedrock of SQL's expressive power.

On first-order logic

Using relational algebra, we can write any query of the form "Find all rows from tables A, B, C, ..., matching *some predicate*", as long as the predicate can be expressed in first-order logic. Specifically, the predicate is formed using:

- various comparisons on rows and columns,
- logical operations "and" (∧), "or" (∨) and "not" (¬), and
- the universal "for every" (∀) and existential "there exists" quantifiers (∃) that operate on rows of a given table.

Let's consider tables named person, car and ticket. We could express a query like "find me people who own only BMW cars, but have at-least one speeding ticket". The predicate can be written as:

my_query(person) = (∀car, person owns car ∧ car.make = 'BMW') ∧ (∃ ticket, person has ticket)

On Gremlin

Gremlin is a standard graph traversal language. It is part of the Tinkerpop stack and works across all Blueprints-compatible databases. You can read more about Gremlin here:

1. Gremlin Wiki on Github
2. Gremlin Docs

3. The Path-o-logical Gremlin (presentation)

Gremlin is great for step-based queries. For e.g., something like "find the friend-of-a-friend of vertex v" can be written as v.out('friend').out('friend'). This style of traversal with vertices and edges isn't natural in SQL with tuples.

The declarative querying style of SQL is, however, different from Gremlin. The SQL2Gremlin tutorial goes through some examples. But you can see that the translation isn't obvious.

Pixy: First-order logic with Gremlin

Pixy is a bridge from first-order logic to Gremlin. The first-order logic of Pixy operates on vertices and edges. We can ask questions like "Find vertices and edges that match *some predicate*" where the predicate is formed by

- various comparisons on vertex and edge properties,
- logical operations "and" (∧), "or" (∨) and "not" (¬), and
- the universal "for every" (∀) and existential "there exists" quantifiers (∃) that operate on vertices and edges.

Pixy queries are expressed using Prolog rules, not SQL. Rules in Prolog are expressed as Horn clauses. Prolog like SQL has the full expressive power of first-order logic.

Let's take the predicate from the earlier discussion,

my_query(person) = (∀car, person owns car ∧ car.make = 'BMW') ∧ (∃ ticket, person has ticket)

Let's say that we represent people as vertices with outgoing edge types named 'car' and 'ticket' to vertices representing cars and tickets. Now, we could express the above predicate using Horn clauses as follows:

```
my_query(Person, Ticket) :- out(Person, 'ticket', Ticket),
                    not(not_all_bmw(Person)).

not_all_bmw(Person) :- out(Person, 'car', Car),
              property(Car, 'make', Make),
              Make <> 'BMW'.
```

Note that out and property are pre-defined predicates in Pixy. You can see that the ∃ part of the query is easy. This is a matter of finding a ticket. The ∀ part of the query is implemented using two nots. In other words, saying "every car is a BMW" is the same as saying "there is no car that isn't a BMW".

You can read the Pixy Tutorial to learn more about Pixy and how it compiles Horn clauses to Gremlin steps. The design allows you to add Gremlin steps to do things that aren't part of first-order logic, like grouping and aggregation.

ER models in Pixy

If you use an ER model as a starting point for your design, you can reconstitute the ER model from the final graph schema using Pixy. Consider the previously referenced ER model with entities named User, Page and Tag and relationships named Owns, Invites and Tagged-As.

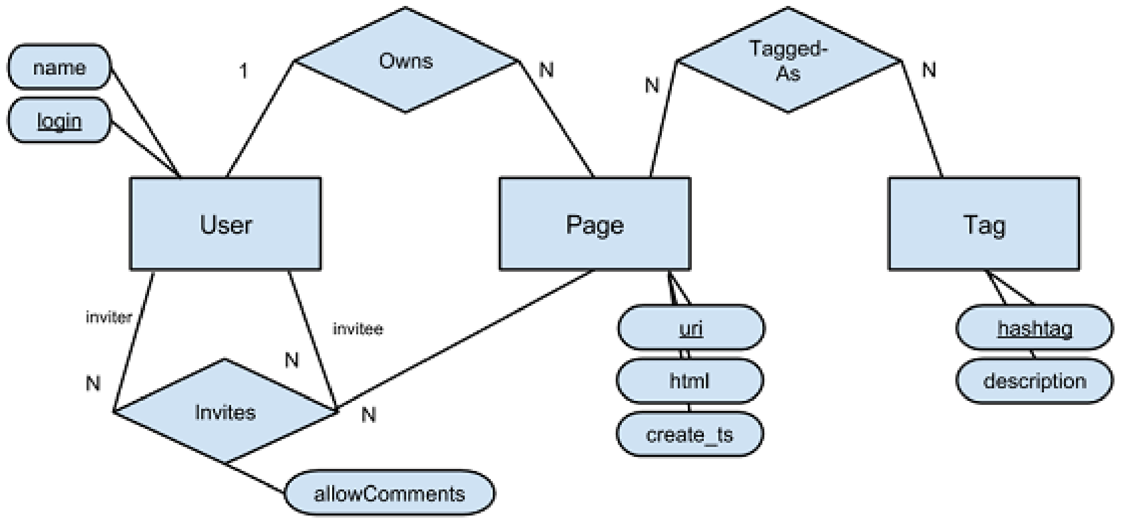


Figure 21: ER model for User-Page-Tag application

This was translated to a graph schema with four types of vertices, viz. User, Page, Tag and Invitation.

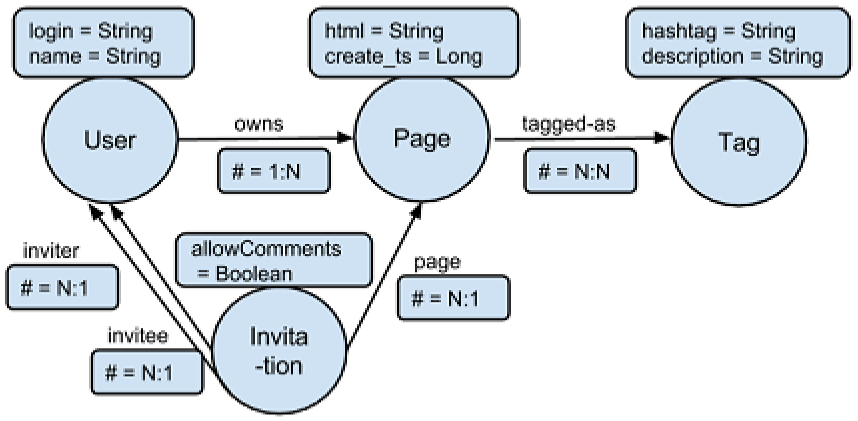


Figure 22: Graph schema for User-Page-Tag application

Now, we can reconstitute the ER model from the graph schema using Pixy with the following clauses:

```
% Entities
user(User, Name, Login) :- property(User, 'name', Name), property(User, 'login', Login).
page(Page, Uri, Html, CreateTs) :- property(Page, 'uri', Uri), ...
tag(Tag, Hashtag, Description) :- property(Tag, 'hashtag', Hashtag), ...
```

```
% Relationships
owns(User, Page) :- out(User, 'owns', Page).
taggedAs(Page, Tag) :- out(Page, 'tagged-as', Tag).
invites(Invitation, Inviter, Invitee, Page) :-
   out(Invitation, 'invitee', Invitee),
   out(Invitation, 'inviter', Inviter),
   out(Invitation, 'page', Page).
```

Every predicate corresponds to an entity or a relationship. The predicate operates on vertices, edges and properties that belong to the graph schema. Now, you get the full power of first-order logic on the ER model. In other words, any first-order predicate that applies to entities and relationships can be written as a Pixy query that uses the above clauses.

Let's take an example predicate that matches all users invited to pages tagged 'tinkerpop' created in 2014. You could express this as follows:

```
tinkerpop_invitee(User, Page) :- invites(_, _, User, Page),
   page(Page, _, _, CreateTs),
   CreateTs > 1388534400L, % Unix timestamp for 1/1/2014
   taggedAs(Page, Tag),
   tag(Tag, 'tinkerpop').
```

Note that '_' is used to represent anonymous variables.

Query requirements don't usually matter while modeling

It isn't surprising that queries in first-order logic can be compiled to Gremlin, since Gremlin is Turing-complete. The surprising thing is that Pixy converts any first-order logic query on an ER model to something that executes "efficiently" on the corresponding graph database.

By "efficiently", we mean that the Pixy/Gremlin query will always traverse edges to go from one entity/relationship to another. Edge traversal operations in graph databases are typically orders of magnitude faster than index-based joins in relational databases.

Queries on properties, will of course, need indexes for efficient querying. But as long as your starting ER model is accurate, your application will not have to simulate joins using these property indexes. In that sense, the graph schema design is independent of the query requirements.

Conclusion

In this document, we have shown that the property graph model is a theoretical strong model capable of being as flexible and powerful as the relational model. Specifically, we have shown that the three pillars of the relational model are matched by the property graph model:

1. **Expressive power:** ER diagrams can be converted to property graph schemas.
2. **Strong design guidelines, aka normalization:** Graph schemas can be normalized using the 7 normalization rules, or the 1 normalization meta-rule.
3. **A powerful query language:** Pixy, like SQL, can model queries expressible in first-order logic and convert these queries to efficient "JOIN-less" Gremlin traversals.

To summarize…

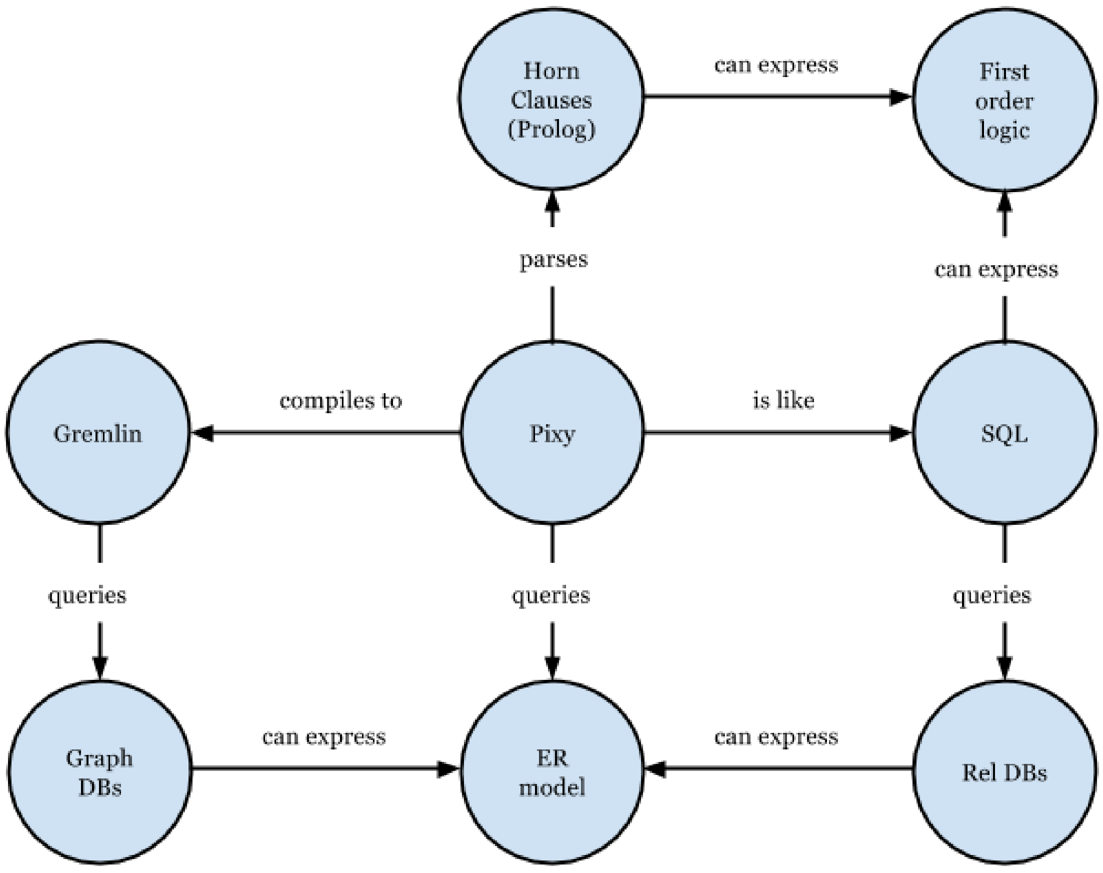


Figure 23: A graph summarizing this document