Building a Knowledge Graph With TypeDB

London, May 2022

Tomás Sabat, Vaticle
tomas@vaticle.com

Abstract

Ever since Google popularised the term in 2012, knowledge graphs have seen massive amounts of public interest. From Fortune 500 companies to universities, organisations all over the world are investing large resources into knowledge graphs.

Despite their popularity, however, a common definition of what really constitutes a knowledge graph is hard to find. This article clarifies those misconceptions, and demonstrates how and why to build a knowledge graph with TypeDB.

1 The Origin of Knowledge Graphs

The term *Knowledge Graph* actually originated back in the 1980s, when academics in the Netherlands formally described a knowledge-based system (James, 1992) that integrated knowledge from various sources to represent natural language. Ever since, the term has largely been used in the field of symbolic AI and knowledge engineering without any sort of common definition, which has led to different interpretations.

Many people consider knowledge graphs as systems that describe the real world in a graph-like format, which incorporate different domains by representing them as entities and relations (Paulheim, 2016). Some describe knowledge graphs as merely semantic representations of knowledge, without having necessarily a graph structure (Ehrlinger, Wolfram, 2016) , while others still have called any type of graph-based knowledge a knowledge graph (Färber et al, 2016).

There are also engineers who consider knowledge graphs to be platforms that distil large collections of text (using for example Apache Solr or Stanford CoreNLP) and store that into graph databases. The lack of a common definition has also led some to suggest that knowledge graphs are no different from knowledge bases or ontologies (Guarino, Giaretta, 1995; Baral, Kraus, Minker, 1991)

When Google referred to its general purpose knowledge graph that powers its search engine — "things, not strings" — it (rightfully) reinvigorated the public's interest into knowledge graphs.

2 What is a Knowledge Graph?

Despite various interpretations, a knowledge graph is at its core a knowledge-based system that consists of two components:

- *Knowledge Database*: A system that can natively represent heterogenous and highly interconnected data aggregated from various sources, and which can accurately describe its semantic structure in the form of an ontology (for practical purposes, also referred to as a *schema*).
- *Reasoner*: An engine that can perform deductive logic over the knowledge database to infer knowledge that would otherwise be hidden.

The knowledge database captures graph-shaped data and organises it into different types of entities, relations, and attributes. This kind of system represents the real world as accurately as possible, by enabling heterogenous data sources to maintain their context and semantics as they are ingested into the knowledge database. Modelling the real world in this way enables the knowledge graph to act as a bridge between humans and computers.

The description of how this data is organised is often called an ontology. These ontologies can either be generated from scratch by experts, or done automatically, from unstructured or semi-structured data sources.

The ontology of the knowledge graph is not fixed, so as the real world evolves, so should the ontology. Once the data has been populated, the knowledge database makes it easy to traverse through this graph-shaped data.

Sitting on top of the knowledge database is the reasoning engine. The reasoning engine leverages the ontology's semantically rich structure to derive new knowledge. While various forms of reasoning exist, most are based on deductive reasoning.

Deductive reasoning means that given a premise, a logical consequence can be derived from a rule based solely on the truth of the premises. This type of reasoning can be used to automatically infer new facts, based on existing data. They can be used to automatically infer new facts based on the existence of certain patterns. In a knowledge graph, rules themselves count as knowledge as they describe concrete patterns that must be true across the facts present.

3 Building a Knowledge Graph

There are various types of technologies available to build knowledge graphs. Most of these come in the form of a database software that is used to represent the data for the knowledge database. A reasoner is often added on top to provide inference capabilities.

Given the "graph" nature of a knowledge graph, some engineers lean towards using graph databases, such as labelled property graphs or RDF stores, to build their knowledge graph. However, despite being able to model graph-shaped data, these types of databases don't provide the necessary modelling constructs to accurately represent the complexity which is inherent in knowledge graphs.

Essentially, as they offer a data model that is predicated on just nodes and edges (similarly to relational databases), these databases lack fundamental modelling constructs, or are too complex to work with and struggle to natively represent contextually rich data. Moreover, most lack a built-in reasoning engine, which leads to the added complexity of having to use third-party software (see our documentation for a detailed overview of property graphs and RDF stores).

So, to build a useful knowledge graph, several key modelling constructs are needed to ensure it models the real world as accurately as possible. And since the data in a knowledge graph comes from various sources, the data will contain many types of entities, relations, and attributes. This requires a strongly-typed database which can natively represent data with a high degree of complexity.

This is where TypeDB comes in.

3.1 Entity-Relationship Modelling

TypeDB (available open-source) makes it possible to create knowledge graphs using the well-known Entity-Relationship model . This differs from traditional databases where the model needs to go through an often difficult normalisation or reification process to fit into a lower-level representation.

Using a relational model means enforcing a tabular structure on highly contextualised data. Similarly, graph databases force everything to fit into nodes and binary edges. The real world doesn't fit neatly into tables or binary edges, thus the ontology that is created ends up being too complex and loses much of its semantics.

Building a knowledge graph with the Entity-Relationship model, therefore, means that developers can represent entities, relations , and attributes, directly in TypeQL (TypeDB's query language): the way we think of a model conceptually as humans is also how we implement it in code in TypeDB.

The example below shows how a basic model in TypeQL is written:

```
define

person sub entity,
  owns name,
  plays employment:employee;
```

```
company sub entity,
  owns name,
  plays employment:employer;

employment sub relation,
  relates employee,
  relates employer;

name sub attribute,
  value string;
```

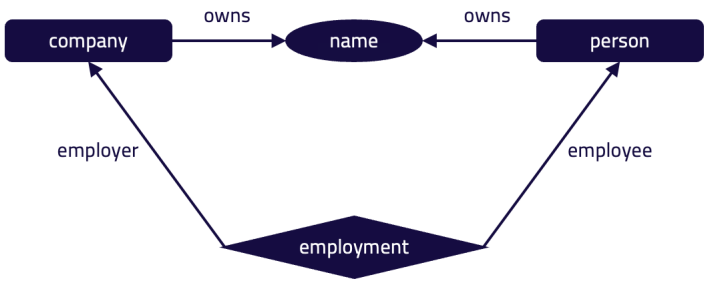

Figure 1 In our nomenclature, squares denote entities; diamonds denote relations; ovals denote attributes. This figure outlines a model with two entities — a person and a company — and both entities own a name attribute. The person plays the role of employee in the employment relation, while the company plays the role of employer.

3.1.1 Type Hierarchies

When aggregating data from various sources, very often the data can become hierarchical, making it necessary to represent type hierarchies as part of the ontology. This is why TypeDB offers the ability to model type hierarchies out of the box, unlike relational and mainstream graph databases.

Following the principles of an object-oriented type system, TypeDB ensures that all types inherit the behaviours and properties of their super-types. This makes complex data structures reusable, and data interpretation richer

through polymorphism. In the example below, a three-level entity `person` hierarchy is modelled. All of its subtypes will inherit the attributes `first-name` and `last-name` without having to re-declare these one by one:

```
define

person sub entity,
  owns first-name,
  owns last-name;

  student sub person;
    undergrad sub student;
    postgrad sub student;

  teacher sub person;
    supervisor sub teacher;
    professor sub teacher;
```

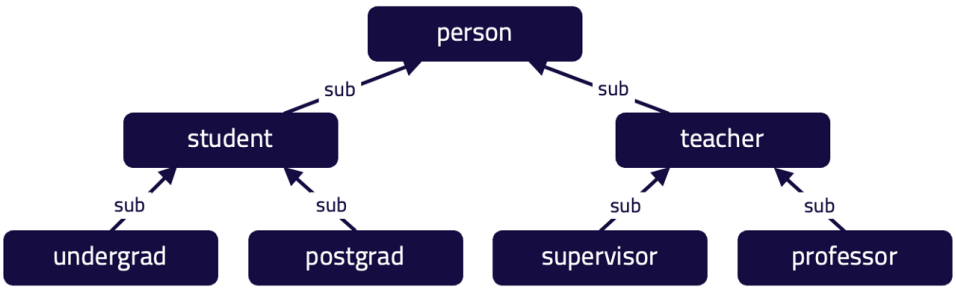

Figure 2 This type hierarchy describes an entity of type person that is subtyped by the types student and teacher. There are two types of students, undergrads and postgrads, and there are two types of teachers, supervisors, and professors.

3.2 N-ary Relations

In the real world, relations aren't just binary connections between two things. That's why it's often necessary to capture three or more things related with each other at once. Representing them as separate binary relationships would

lead to a loss of information, which is what happens in graph databases. TypeDB, on the other hand, can naturally represent an arbitrary number of things as one relation, for example, without needing to reify the model.

In the example below, the n-ary relation `cast` connects three different entities: a `person` entity, a `character` entity, and a `movie` entity:

```
match

$person isa person, has name "Leonardo";
$character isa character, has name "Jack";
$movie isa movie, has name $movie-name;
(actor: $person, character: $character, movie: $movie) isa cast;
get $movie-name;
```

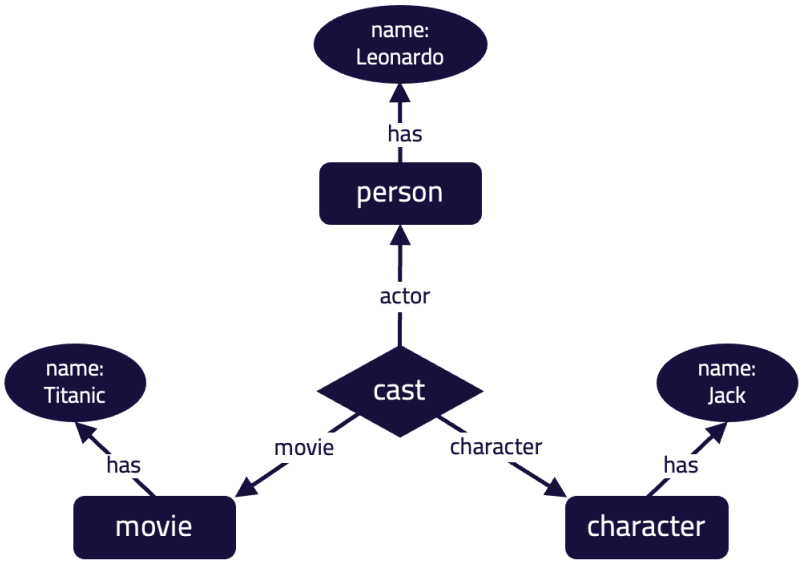

Figure 3 This is an example of an n-ary relation, specifically a ternary relation, of type cast. This relation relates to three entities: the movie "Titanic" plays the role of movie, the person "Leonardo" plays the role of actor, and the character "Jack" plays the role of character in this cast relation.

3.3 Nested Relations

Relations are concepts that describe the association between two or more things. Sometimes, those things are relations themselves, which means modelling a relation that directly refers to another relation — nested relations.

Graph databases don't allow relations to connect to other relations directly, since that would require making a binary edge that points to a binary edge. The only way to achieve this is via *reification*, which is transforming a binary edge into a node so that another binary edge can now point to it. This workaround is a code smell that gets very messy very quickly.

TypeDB, however, does allow nested relations, making it possible to model data in its most natural form. In the example below, the relation of type `marriage` is assigned to the variable `$mar`, which is then used to connect it to a `city` through the relation `located`:

```
match

$alice isa person, has name "Alice";
$bob isa person, has name "Bob";
$mar ($alice, $bob) isa marriage;
$city isa city;
($mar, $city) isa location;
```

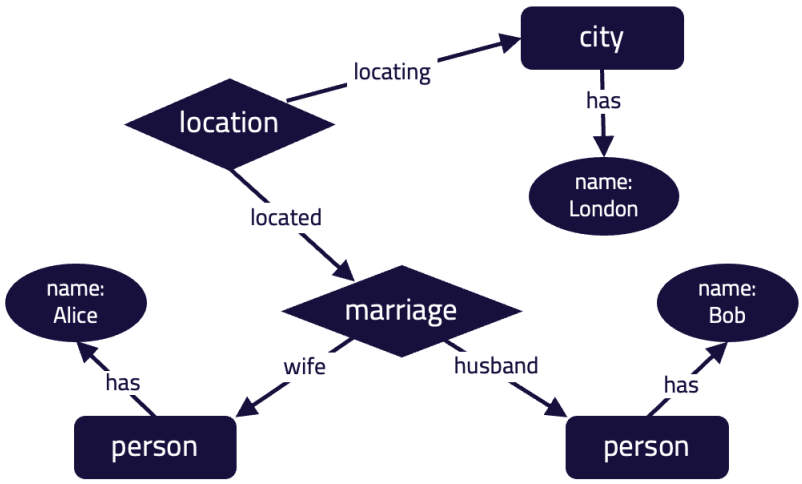

Figure 4 In this figure, the person "Alice" plays the role of wife and the person "Bob" plays the role of husband in a marriage relation. Marriage is a nested relation, as it also plays the role of located in a location relation, where the city "London" plays the role of locating in that same relation

3.4 Automated Reasoning

TypeDB offers a built-in native reasoning engine that was designed from scratch to infer new knowledge directly from the database, and provides full explainability to give the necessary trust for the results it returns. This differs to mainstream graph databases, which don't offer native reasoning capabilities; while third-party plug-ins exist, these can't reason over all data in real time and are too cumbersome to use in production.

TypeDB performs deductive reasoning over persisted data at query runtime from user defined logical rules. These rules extend the expressivity of the model as it enables the knowledge graph to derive new conclusions, when certain logical conditions in the database are satisfied.

There are many uses for rule based reasoning in TypeDB, such as the need for reusable computation blocks, inferences that are always up-to-date, safety against contradictions, and reducing query complexity by taking care of highly permutative operations.

TypeDB allows any arbitrary rule to be written, as long as the conditions

can be expressed in a TypeQL pattern. The following is just a simple example of a rule, which creates a transitivity in the `location` relation:

```
define

rule transitive-location:
when {
  (located: $x, locating: $y) isa location;
  (located: $y, locating: $z) isa location;
} then {
  (located: $x, locating: $z) isa location;
};
```

This means that given this data:

```
$camden isa borough, has name "Camden;
$london isa city, has name "London";
$uk isa country, has name "UK";
(located: $camden, locating: $london) isa location;
(located: $london, locating: $uk) isa location;
```

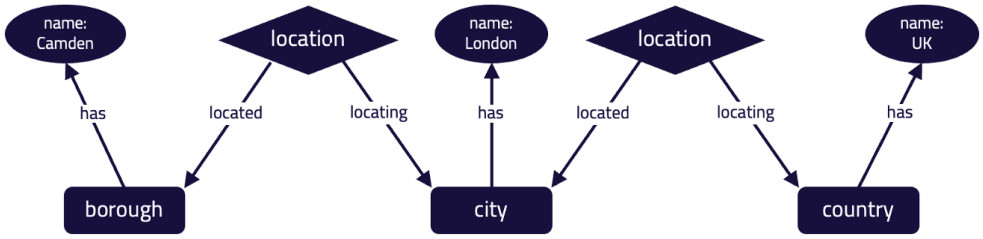

Figure 5 In this example, the borough "Camden" is located in the city "London", which is located in the country "UK".

Even though `camden` and `uk` are not directly connected, TypeDB will infer that since `camden` is located in `london`, that also means that `camden` is located in the `uk` with this query:

```
(located: $camden, locating: $uk) isa location;
```

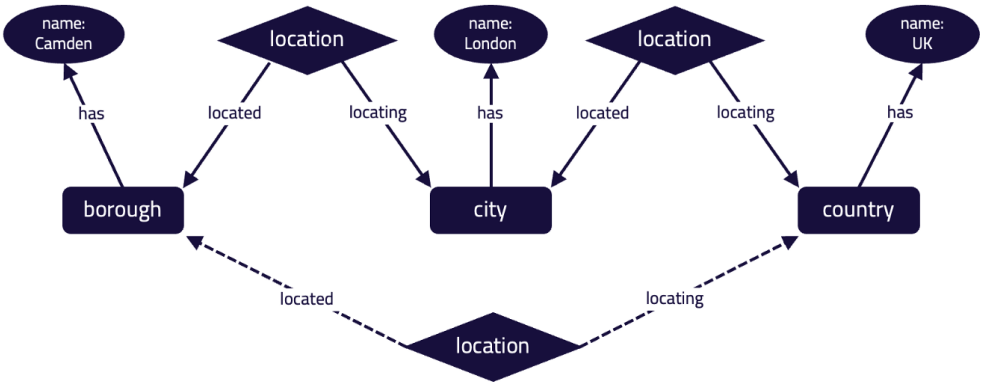

Figure 6 Using automated reasoning, TypeDB can infer a relationship (dotted line) between the borough "Camden" and the country "UK", even though they're not directly connected.

Any pattern expressed in TypeQL can serve as a condition in a TypeDB rule. This allows for all sorts of complex rules, from representing tax systems, hypothesis generation for drug discovery, or uncovering hidden connections in cybersecurity knowledge graphs. Moreover, like functions in programming, rules can chain onto one another, creating abstractions of behaviour at the data level.

4 What Are Knowledge Graphs Used for?

Today, many organisations in different industries use the open source and commercial versions of TypeDB to build knowledge graphs. Some of the knowledge graphs built with TypeDB include:

- **Life Sciences:** AstraZeneca built a biomedical knowledge graph with TypeDB to predict new disease targets for drug discovery. They ingest heterogeneous biomedical data such as genes, proteins, compounds, and diseases. As they need to work with multiple ontologies, they leverage TypeDB's reasoning engine to rectify IDs across multiple merged ontologies (learn more in this article).
- **Cyber Security:** A Fortune 100 company built a knowledge graph to prosecute cyber criminals on their networks. They integrated diverse

private and public data sources to identify malicious behaviour, and leveraged TypeDB's reasoning engine to identify criminals and collect the evidence to prosecute them.

- **Robotics:** TNO, the Dutch national research institute, uses TypeDB as the knowledge graph to power search and rescue autonomous vehicles. The success of these vehicles (which includes the famous SPOT robot from Boston Dynamics) depends on how closely to reality the knowledge graph can model a physical environment. The reasoning engine is then used to infer the relative position of physical objects in order to automatically navigate between rooms in a building (learn more in this article and this paper).

- **E-commerce:** A multi-national e-commerce enterprise built a knowledge graph with TypeDB to power the recommendation engine on its website. They represent their entire product catalogue, which includes thousands of products, in one ontology, using modelling constructs such as type hierarchies and n-ary relations to accurately represent thousands of product categories.

- **Data Governance:** A Fortune 100 company built a knowledge graph with TypeDB to capture the complexity inherent in their organisation. This meant aggregating disparate and unconnected data silos across organisational units, some acquired through M&A or poor data governance. The result was thousands of separate tables that no one really understood. Given this complexity, creating a knowledge graph in TypeDB to overcome this data silo problem was essential to unlocking their data's real potential and drive better business decisions.

5 Conclusion

TypeDB's expressive data model allows it to serve as a knowledge database that captures context and semantics when storing knowledge, while its built-in reasoner enables it to discover new facts by creating logical abstractions of data. Therefore, it meets our definitions of a true Knowledge Graph, and is built with commercial usability in mind.