



Introduction to Graph Databases

A tutorial on the RDF data model and the SPARQL query language.

Paolo Pareti, University of Southampton



Overview

1. In this course we will look at RDF, one the most widespread graph data models.
2. We will briefly introduce the concepts of RDF schemas and ontologies.
3. We will learn how graph data is queried with SPARQL: the main RDF query language.
4. We will begin working through simple examples and hands-on exercises.

All the material for this course can be downloaded from the following GitHub repository:

<https://github.com/paolo7/Introduction-to-Graph-Databases>



Software Requirements

You will need this software for the hands-on part of this course.

This software is compatible with all major operating systems.

- [Java](#) (JRE), to run JAR files
- An RDF database and query engine for the hands-on part of the course. You can find installation instructions in the next two slides.
 - The recommended option is [GraphDB](#). This database requires installation, but then you will have access to a better graphical interface, and a built-in graph visualisation tool.
 - If you have problems installing GraphDB, you can also opt for [Apache Jena Fuseki](#). This lightweight database server runs as a simple executable JAR file. However, if you choose Fuseki you will not be able to experiment with the graph visualisation part of the course.



GraphDB Installation Instructions

- Request a download link from this link: <https://www.ontotext.com/products/graphdb/graphdb-free/>
 - GraphDB is free, but you need to register with an email address before downloading it.
- Once you receive the email, download GraphDB as a *desktop installation* for your operating system.
- Follow the instructions from this link that are relevant to your operating system:
<http://graphdb.ontotext.com/documentation/free/free/run-desktop-installation.html>
- Once you can access the workbench interface <http://localhost:7200/> you are good to go (there is no need for additional configurations).



Fuseki Installation Instructions

- Get a copy of the executable JAR file of Jena Fuseki
 - One such copy is attached to the material for this course. The filename is: **apache-jena-fuseki-3.14.0.zip**.
 - Or if you prefer, download it from <https://jena.apache.org/download/>.
- Unzip the file.
- Run a terminal/command prompt/power shell.
- Start the triplestore service by navigating inside the uncompressed folder and using this command:
 - `java -jar fuseki-server.jar`
- Open this address on a web browser:
 - `http://localhost:3030/index.html`
- If all was done correctly, you should see the web interface of Fuseki.



Why graphs?

Representing data as graphs is becoming more and more common.

For example, graph data has important applications in Machine Learning (under the name of *knowledge graphs*) and it is used to structure vast amount of information, as in the case of Google Knowledge Graph.

RDF is being used by many well known organisations, such as the BBC and Pinterest.

When data is rich in connections, graphs tend to be easier to work with.

- The focus is on describing information more than just storing it; on usability more than just efficiency.
- Graphs are flexible as they are not constrained by rigid relational database schemas. They can easily integrate unexpected data and adapt to change.



Why RDF?

- It is the most *standard* graph data model, as it is a recommendation from the W3C (the same organisation behind the HTML and HTTP standards).
- There are many available database implementations to choose from.
- It is well researched and optimised.
- There is a wealth of tools, libraries and APIs to work with it.

Since most graph data models are based on the same principles, much of what you will learn today about RDF can also be applied to other models too.



Graph data representation

Consider a simple SQL table representing persons, their names, and who follows whom on social media.

Table: Names	
ID	Name
:a0	Adam
:b0	Bill
:c0	Cindy
:b1	Bill

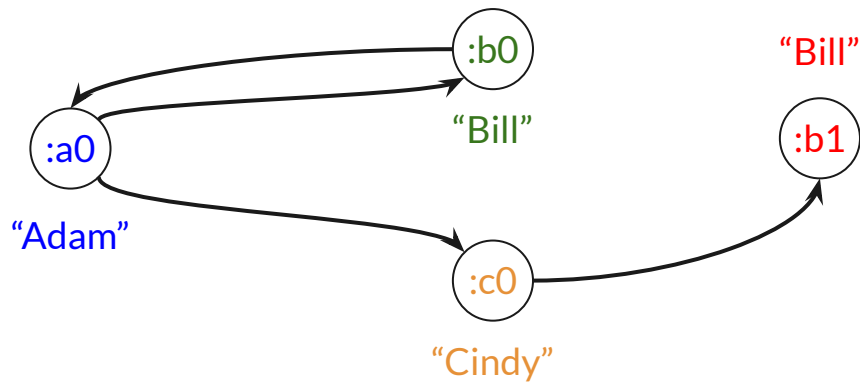
Table: Follows	
ID	Friend
:a0	:b0
:a0	:c0
:b0	:a0
:c0	:b1

Graph data representation

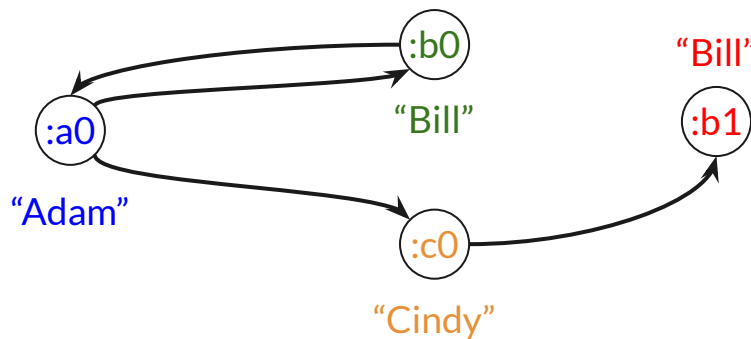
When data that is rich in connections, it is easier to think of data as a graph.

Table: Names	
ID	Name
:a0	Adam
:b0	Bill
:c0	Cindy
:b1	Bill

Table: Follows	
ID	Follows
:a0	:b0
:a0	:c0
:b0	:a0
:c0	:b1



What is a graph?



The basic components of a graph are **nodes** and **edges**.

- Nodes represent “entities”. In this example, `:a0` is a node.
- Edges represent relationships between “entities”. In this example `<:a0,:c0>` is an edge.

In practice, edges have a “direction”, and thus we say that this is a **directed** graph.

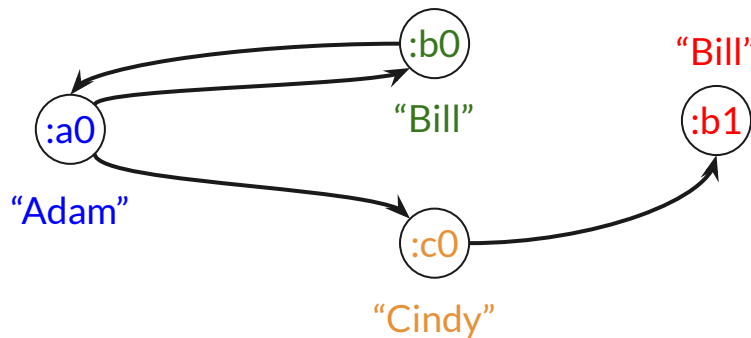
- In this example Adam follows Cindy, but Cindy does not follow adam.
- The edge `<:a0,:c0>` is in the graph, but the edge `<:c0,:a0>` is not.

What about attributes?

What are labels, such as “Adam”?

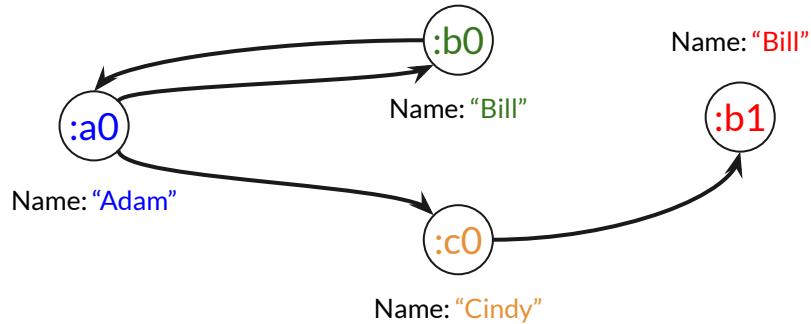
There two major approaches to model attributes are the following:

- The RDF model models everything as nodes and edges, including attributes.
- The Labelled Property Graph (e.g. Neo4J) adds attributes as a third element in the graph, distinct from nodes and edges.

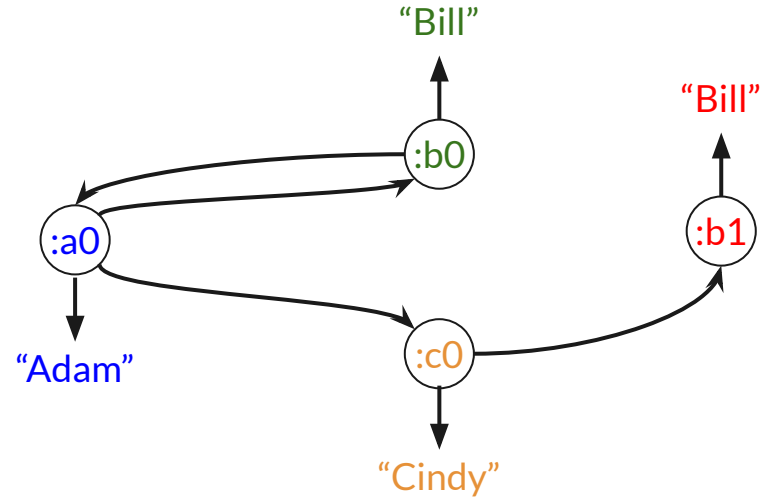


Ways to model attributes?

Labelled Property Graph model



RDF model





The Labelled Property Graph model vs RDF

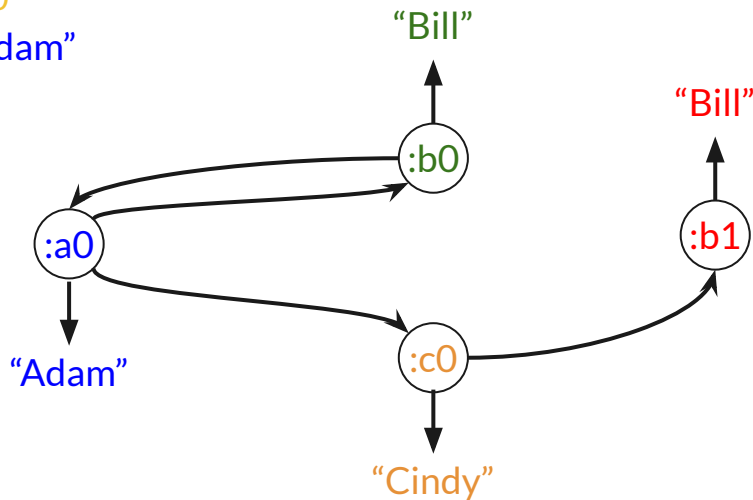
- The RDF model keeps things simple, as everything is modelled as nodes and edges.
- While the additional data structure of “labelled properties” can be convenient in some applications, it can be an unnecessary complication in others.
 - For example, what if we want to have multiple instance of the same property? E.g. What if people have multiple names?
 - What if there are properties of properties? E.g. What if someone changes name after a certain date?
 - What if we want to reference the same property from different nodes?

Ultimately, the core of all graph data models is a set of nodes and edges.

Property labels.

We need one more important component of our graph model: what does an edge mean?

- Edge $\langle :a0, :c0 \rangle$ means that person $:a0$ follows person $:c0$
- Edge $\langle :a0, \text{"Adam"} \rangle$ means that person $:a0$ has name "Adam"



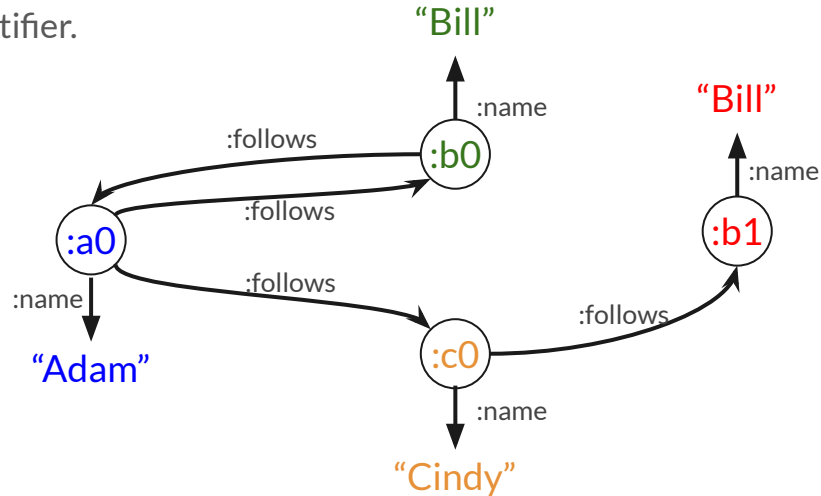
Property labels.

Nearly all graph databases give labels to edges.

In RDF, each edge is labelled with a specific property identifier.

In our example we could use these two properties:

- :follows
- :name

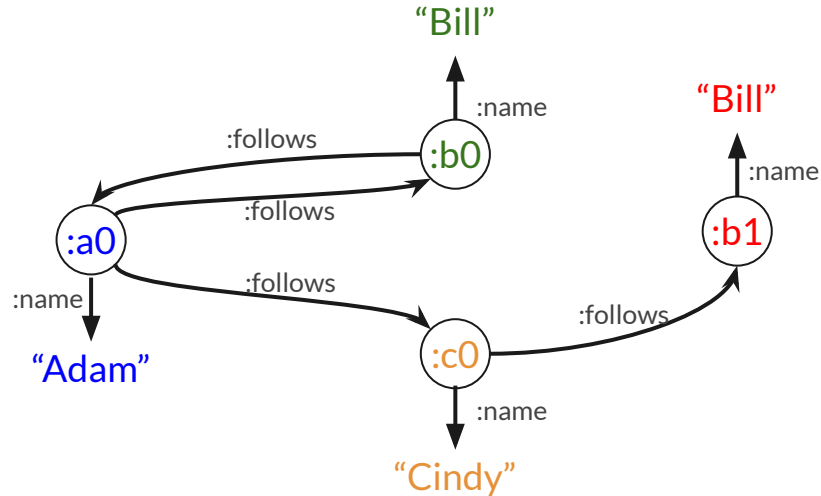


Property labels.

Now our edges are no longer a pair of values, but a **triple**.

Edges:

- `<:a0, follows, :c0>`
- `<:a0, name, "Adam">`
- `<:b0, follows, :a0>`
- `<:b0, name, "Bill">`
- ...

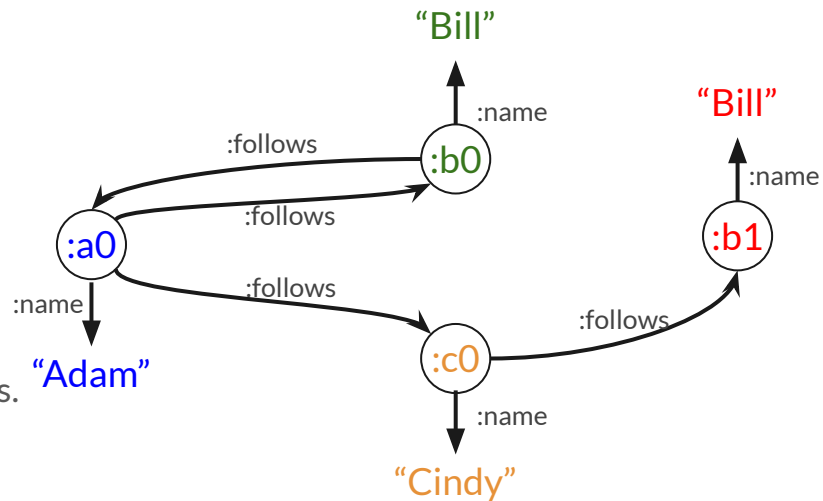


That's it! (Almost)

RDF can be seen just as a collection of triples representing edges.
Nodes are explicitly listed, as they are implicitly described in the edges.

The RDF data for the graph in the example is the following:

- `<:a0, :follows, :c0>`
- `<:a0, :follows, :b0>`
- `<:a0, :name, "Adam">`
- `<:b0, :follows, :a0>`
- `<:b0, :name, "Bill">`
- `<:c0, :name, "Cindy">`
- `<:c0, :follows, :b1>`
- `<:b1, :name, "Bill">`





Comparison with SQL tables

Graph representations can be just as efficient as relational ones.

- `<:a0,:follows,:c0>`
- `<:a0,:follows,:b0>`
- `<:a0,:name,"Adam">`
- `<:b0,:follows,:a0>`
- `<:b0,:name,"Bill">`
- `<:c0,:name,"Cindy">`
- `<:c0,:follows,:b1>`
- `<:b1,:name,"Bill">`

Table: Names	
ID	Name
:a0	Adam
:b0	Bill
:c0	Cindy
:b1	Bill

Table: Follows	
ID	Follows
:a0	:b0
:a0	:c0
:b0	:a0
:c0	:b1

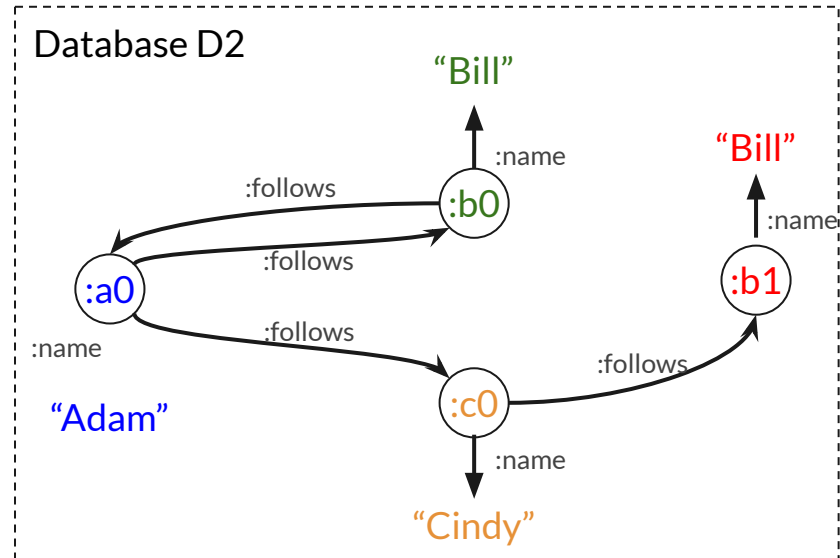
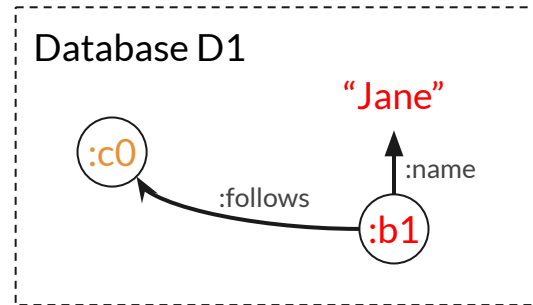
What are nodes?

Imagine integrating these two datasets from two different companies:

- Does entity **:b1** have two names?
- Does **Jane** follow **Cindy**?

Data integration can be very problematic.

RDF uses IRIs to identify nodes.
This makes data integration easier.





The building blocks of RDF: IRIs

An Internationalized Resource Identifier (IRI) is a way to assign **universally unambiguous** identifiers.

IRIs are a generalisations of URIs, which in turn are generalisation of URLs.

IRIs can be have non ASCII-characters and have a scheme different from the usual ones (e.g. **http** or **ftp**).

In practice, most IRIs are just URLs.

For example, this is an IRIs:

- <http://example.com/a0>



Namespaces

To increase readability, IRIs are often abbreviated using XML namespaces.

- <http://example.com/a0>
- <http://example.com/:b0>
- <http://example.com/:c0>
- <http://example.com/:b1>

We can simplify these IRIs by assigning a namespace, e.g. **ex** to the prefix **http://example.com/**:

- [ex:a0](http://example.com/a0)
- [ex:b0](http://example.com/:b0)
- [ex:c0](http://example.com/:c0)
- [ex:b1](http://example.com/:b1)



Namespace Conventions

In order for IRIs to act as universal identifiers, it is important to ensure that they are always used to refer to the same thing.

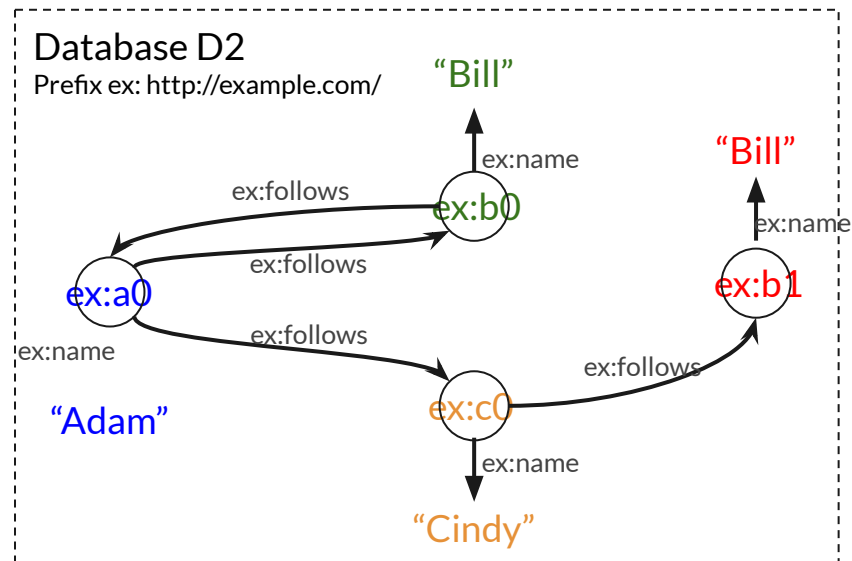
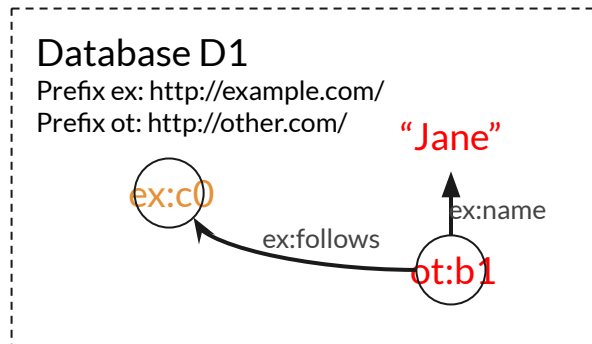
In a traditional database setting, this is no different than the use of unique identifiers.

However, creating identifiers that can work across any database requires special care:

- When creating an IRI, it should be prefixed with a domain that a person/organisation owns.
 - To create an identifier for the IT department, company **ABC** should create an IRI such as :
<http://ABC.com/IT-department>
- When reusing existing IRIs, it is important to maintain their meaning.
 - Company **ABC** should not reuse the IRI of company **XYZ**'s IT department, e.g. as <http://XYZ.com/ITdep>, to refer to its own (**ABC**'s) IT department.
 - A correct way of reusing this IRI is, for example, to specify that both IT departments are collaborating:
< <http://ABC.com/IT-department> , ex: collaboratesWith , <http://XYZ.com/ITdep> >

IRIs and Namespaces in action

- Does entity **b1** have two names?
 - No, entities **ex:b1** and **ot:b1** are clearly distinct, as they have different IRIs.
- Does **Jane** follow **Cindy**?
 - Yes, the same identifier **ex:c0** is being used in both dataset, so it refers to the same person.





A more formal definition of RDF

An **RDF graph** is a set of **RDF triples**.

Each **RDF triple** is a triple of terms $\langle s, p, o \rangle$ where:

- The 1st, 2nd, and 3rd elements are called, respectively, **subject**, **predicate** and **object**.
- Subject are **IRIs** (or **blank nodes**).
- Predicates are **IRIs**
- Objects are either **IRIs**, **Literals** (or **blank nodes**).

Literals are basic data elements, such as strings and numbers.

In our example, the names of people “Bill”, “Adam” and “Cindy” are literals.

Blank nodes are used to indicate the existence of something, without explicitly giving it an identifier.

For the purpose of this course, we can just think of blank nodes are local identifiers, not shared across databases.



Real RDF serialisations - Turtle

Several serialisations of RDF data are available. Turtle is one such serialisation optimised for human readability. This is a correct Turtle file describing the data from our example:

```
prefix ex: <http://example.com/>
```

```
ex:a0 ex:follows ex:c0 .  
ex:a0 ex:follows ex:b0 .  
ex:a0 ex:name "Adam" .  
ex:b0 ex:follows ex:a0 .  
ex:b0 ex:name "Bill" .  
ex:c0 ex:name "Cindy" .  
ex:c0 ex:follows ex:b1 .  
ex:b1 ex:name "Bill" .
```

Real RDF serialisations - RDF/XML

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ns0="http://example.com/">

  <rdf:Description rdf:about="http://example.com/a0">
    <ns0:follows>
      <rdf:Description
rdf:about="http://example.com/c0">
        <ns0:name>Cindy</ns0:name>
        <ns0:follows>
          <rdf:Description
rdf:about="http://example.com/b1">
            <ns0:name>Bill</ns0:name>
          </rdf:Description>
        </ns0:follows>
```

```
    </rdf:Description>
  </ns0:follows>

  <ns0:follows>
    <rdf:Description
rdf:about="http://example.com/b0">
      <ns0:follows
rdf:resource="http://example.com/a0"/>
        <ns0:name>Bill</ns0:name>
      </rdf:Description>
    </ns0:follows>

    <ns0:name>Adam</ns0:name>
  </rdf:Description>
</rdf:RDF>
```



Real RDF serialisations - JSON-LD

```
[
  {
    "@id": "http://example.com/a0",
    "http://example.com/follows": [
      {
        "@id": "http://example.com/c0"
      },
      {
        "@id": "http://example.com/b0"
      }
    ],
    "http://example.com/name": [
      {
        "@value": "Adam"
      }
    ]
  },
  {
    "@id": "http://example.com/b0",
    "http://example.com/follows": [
      {
        "@id": "http://example.com/a0"
      }
    ],
    "http://example.com/name": [
      {
        "@value": "Bill"
      }
    ]
  },
  {
    "@id": "http://example.com/b1",
    "http://example.com/name": [
      {
        "@value": "Bill"
      }
    ]
  },
  {
    "@id": "http://example.com/c0",
    "http://example.com/name": [
      {
        "@value": "Cindy"
      }
    ],
    "http://example.com/follows": [
      {
        "@id": "http://example.com/b1"
      }
    ]
  }
]
```



Schemas and Ontologies



RDF Schemas / Ontologies

In order to understand the data in an RDF graph we need to understand what its triples mean. This boils down to understanding the meaning of the IRIs it contains.

This meaning is typically defined both formally and informally.

A document which specifies this meaning (both the formal and informal one) is typically called a *schema* or an *ontology*.



Informal Definitions

The simplest and most effective way to communicate meaning to a human reader is through a natural language description.

For example, we could define the main relations in our example as follows:

`ex:name` This relation specifies the first name of a person.

`ex:follows` The object in this relation is a person, and the subject is one of their followers on social media.

Typically schemas/ontologies explicitly describe relations and classes (as we will see later) while specific instances such as `ex:a0` are only *implicitly* described. Although we do not have a natural language definition for `ex:a0`, we know that its first name is Adam and that it follows Bill and Cindy on social media.



Formal Definitions - RDFS

One of the key characteristics of schemas and ontologies is the ability to formally describe meaning. This allows machines to reason about the dataset and infer new facts.

For example, the following four triples provide some formal information about our dataset.

1. `ex:name rdfs:domain ex:Person .`
2. `ex:follows rdfs:domain ex:Follower .`
3. `ex:follows rdfs:range ex:Person .`
4. `ex:Follower rdfs:subclassOf ex:Person .`

You might be immediately notice that this formal description is in the form of triples. This is intentional, and it allows an RDF graph to contain both data and its own metadata.

The formal meaning in this example is derived by using [RDFS](#) terms. The RDFS specification provides standardised terms to describe RDF schemas. A more advanced way of describing formal meaning is through the [OWL](#) family of ontology languages.



Formal Definitions - Classes

1. `ex:name rdfs:domain ex:Person .`
2. `ex:follows rdfs:domain ex:Follower .`
3. `ex:follows rdfs:range ex:Person .`
4. `ex:Follower rdfs:subclassOf ex:Person .`

This formal definition describes two classes (type of entities):

`ex:Person` A person.

`ex:Follower` Someone who follows someone else on social medial.

Classes/Types are one of the most basic ways to organise information, and they can be thought of as sets. Classes are often organised in hierarchies, where the notion of a subclass corresponds to the notion of a subsets.



Formal Definitions - Reasoning

1. `ex:name rdfs:domain ex:Person .`
2. `ex:follows rdfs:domain ex:Follower .`
3. `ex:follows rdfs:range ex:Person .`
4. `ex:Follower rdfs:subClassOf ex:Person .`

These four triples are interpreted as follows:

1. Every subject in a triple with relation `ex:name` is an entity of type `ex:Person`.
2. Every subject in a triple with relation `ex:follows` is an entity of type `ex:Follower`.
3. Every object in a triple with relation `ex:follows` is an entity of type `ex:Person`.
4. Every `ex:Follower` is also a `ex:Person`.

This allows a reasoner to infer new facts from our example dataset, such as that `ex:b1` is a Person, and that `ex:c0` is not only a Person but also a Follower.



A Real-World Schema: FOAF

The Friend Of A Friend (FOAF) schema is a simple but widely used schema to describe people and their basic attributes, such as their name and their online profiles.

For example, look at the FOAF [IRI](http://xmlns.com/foaf/spec/#term_Person) for the concept of “Person”, described here:

- http://xmlns.com/foaf/spec/#term_Person

Notice the the *informal definition* of Person.

- It specifies, for example, that the term can be used both to describe real or fictional persons.

Above this informal definition you can see the *formal (aka machine readable) definition* of Person.

- It specifies the fact that persons and organisations are two separate things.

Whenever you want to describe persons, you can reuse the terms in this schema. Both people and machines can simply look up these IRIs and easily find out what you mean.



The SPARQL Query Language



Querying a graph

SPARQL is the query language for RDF graphs.

The most common common type of query follows the “SELECT” form.

Such queries have two main components:

- The SELECT component specifies which **variables** we want to include in our result
- The WHERE component specifies the **graph pattern** that we want to match on the data.

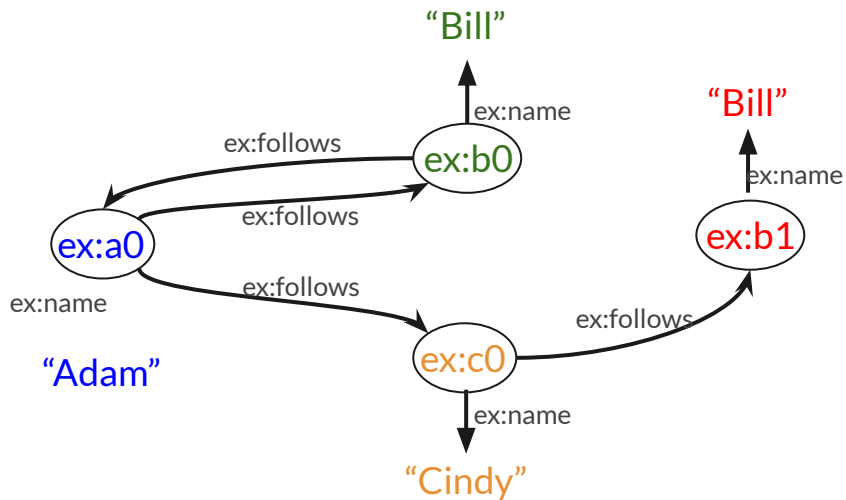
SPARQL example

Return anyone who follows someone else.

```
PREFIX ex:    <http://example.com/>
```

```
SELECT ?follower
```

```
WHERE {  
    ?follower ex:follows ?someone .  
}
```



	follower	↓ ^A _Z
1	ex:a0	
2	ex:a0	
3	ex:b0	
4	ex:c0	

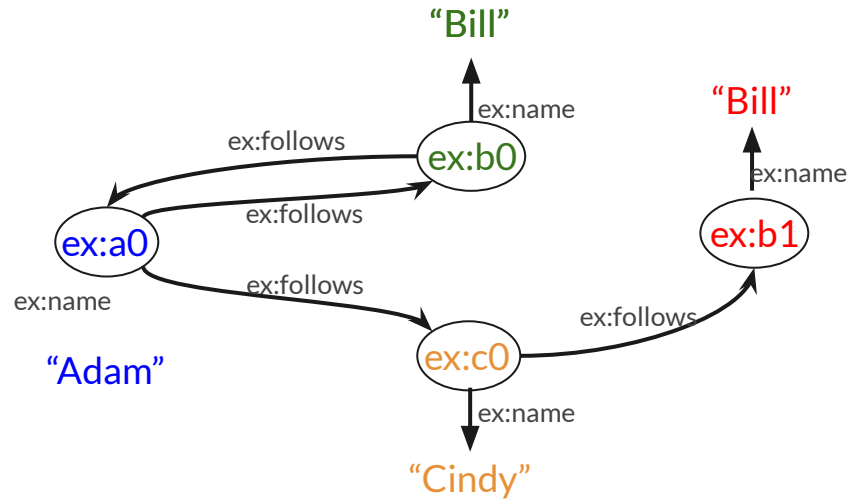
SPARQL example

Return anyone who follows someone else, without duplicates.

```
PREFIX ex:    <http://example.com/>
```

```
SELECT distinct ?follower
```

```
WHERE {  
    ?follower ex:follows ?someone .  
}
```



	follower
1	ex:a0
2	ex:c0
3	ex:b0

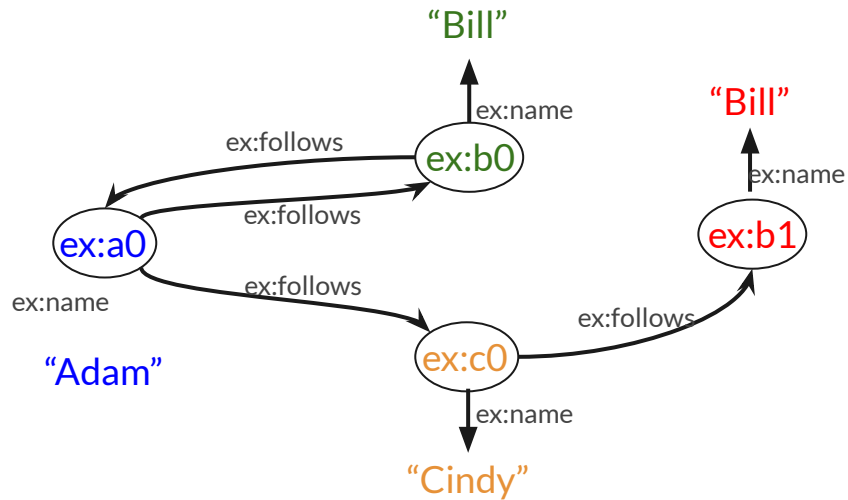
SPARQL example

Return anyone whose name is "Bill"

```
PREFIX ex:    <http://example.com/>
```

```
SELECT distinct ?follower
```

```
WHERE {  
    ?follower ex:name "Bill".  
}
```



	follower
1	ex:b0
2	ex:b1

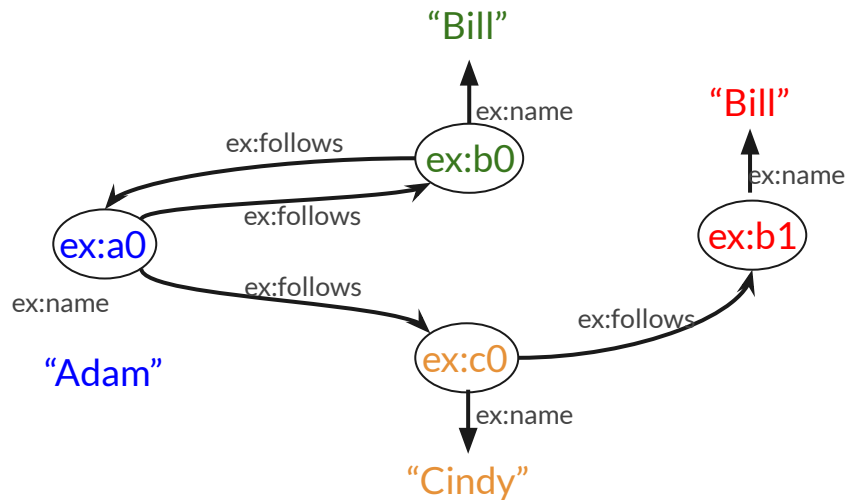
SPARQL example

Return anyone who follows someone whose name is "Bill", along with that someone.

PREFIX ex: <http://example.com/>

SELECT distinct ?follower ?someone

```
WHERE {  
  ?follower ex:follows ?someone .  
  ?someone ex:name "Bill".  
}
```



	follower ↕	someone ↕
1	ex:a0	ex:b0
2	ex:c0	ex:b1

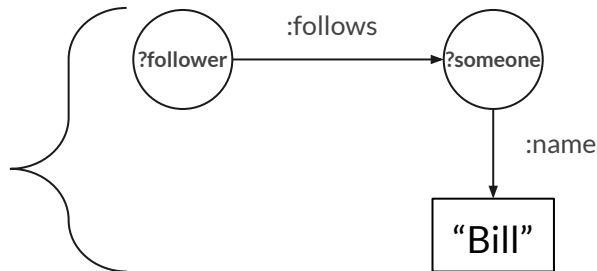
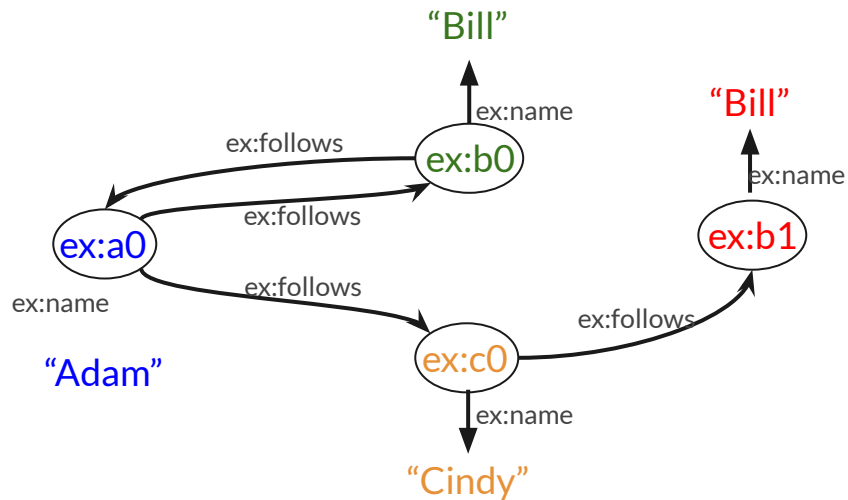
SPARQL example

Return anyone who follows someone whose name is "Bill",
along with that someone.

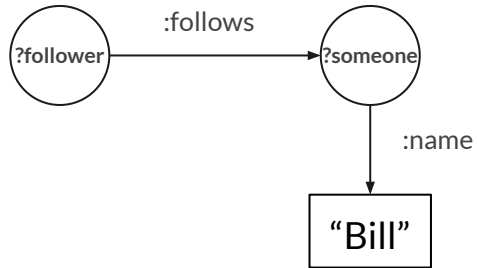
PREFIX ex: <http://example.com/>

```
SELECT distinct ?follower ?someone
```

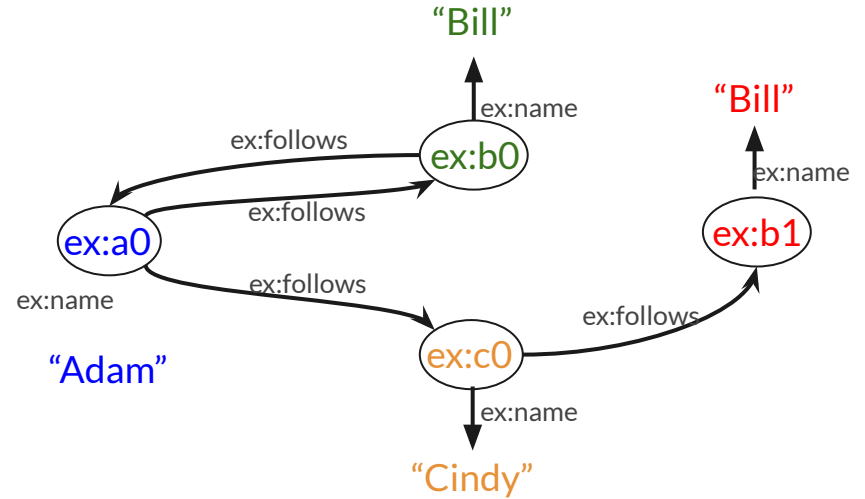
```
WHERE {  
  ?follower ex:follows ?someone .  
  ?someone ex:name "Bill".  
}
```



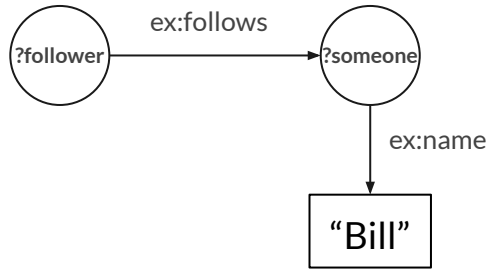
Pattern Matching



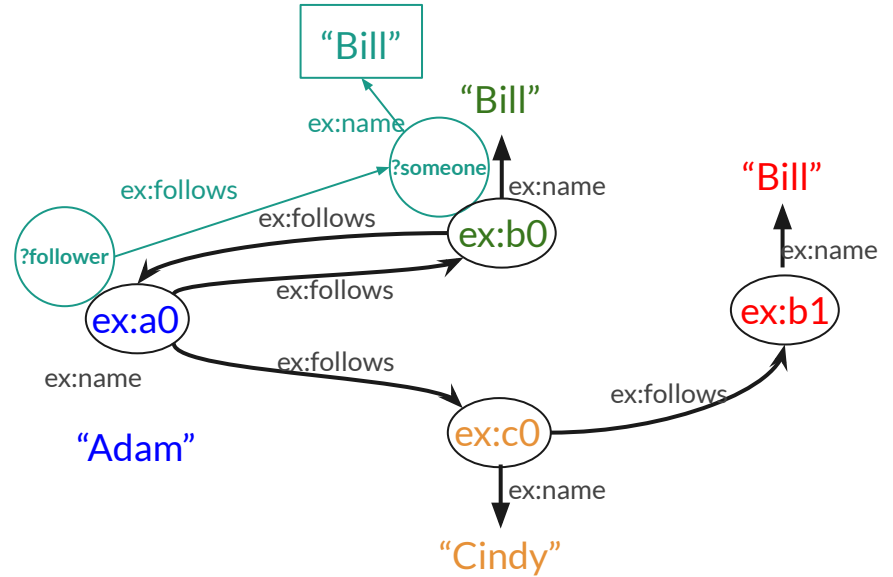
	follower ⚡	someone ⚡
1	ex:a0	ex:b0
2	ex:c0	ex:b1

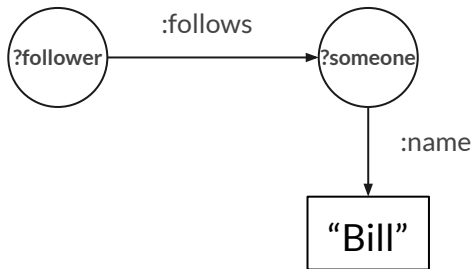


Pattern Matching

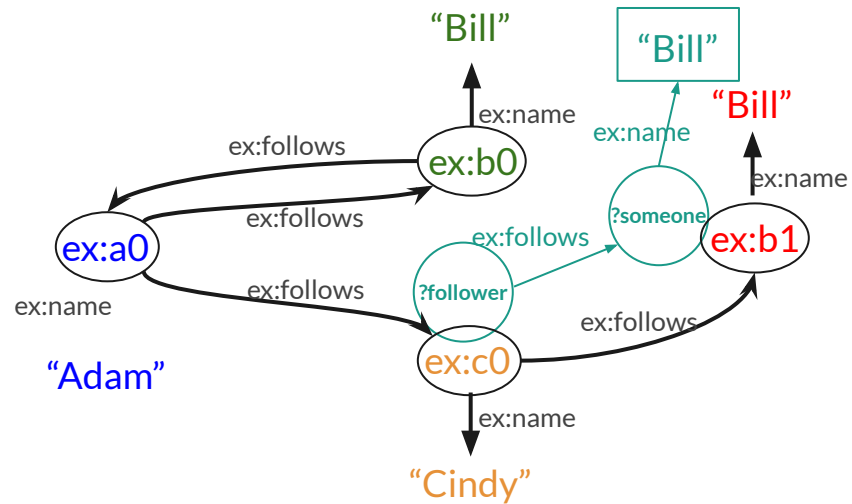


	follower ⚙	someone ⚙
1	ex:a0	ex:b0
2	ex:c0	ex:b1





2	ex:c0	ex:b1
---	-------	-------





Hands-on Exercises - Setup

If you have already setup your triplestore, you only need to run it and access the workbench.

By default, you can access the workbench here:

- <http://localhost:7200/> for GrabDB
- <http://localhost:3030/> for Jena Fuseki

If you have not setup the triplestore, please go through the installation instructions of Jena Fuseki at the beginning of this slide pack. It should only take a couple of minutes to download and run the Fuseki executable.



Hands-on Exercises - Create a dataset

Create a database and initialise it with the data from our example.

- You can find this data in the attached file *Followers.ttl*.
- You can also easily create this file yourself, using the text on the right hand side of this slide.

Followers.ttl

```
prefix ex: <http://example.com/>
```

```
ex:a0 ex:follows ex:c0 .  
ex:a0 ex:follows ex:b0 .  
ex:a0 ex:name "Adam" .  
ex:b0 ex:follows ex:a0 .  
ex:b0 ex:name "Bill" .  
ex:c0 ex:name "Cindy" .  
ex:c0 ex:follows ex:b1 .  
ex:b1 ex:name "Bill" .
```



Hands-on Exercises - Load the dataset (GraphDB)

If you are using the GraphDB interface:

- Navigate to “Setup”, “Repositories”.
- Click “Create new repository”.
- Give it an ID, e.g. “exercisedb”.
- Select the RDFS ruleset.
- Click “Create”.
- In the “Setup”/“Repositories” page, click on the icon to the right of the dataset just created to connect to it. When using the web interface to query or modify a dataset, the dataset that is being queried/modified is the connected one.
- Navigate to “Import”/“RDF”. Click on “Upload RDF files” to start the upload process.
- Select the “Followers.ttl” file and click “Import”.

To access the query endpoint:

- Navigate to “SPARQL”.



Hands-on Exercises - Load the dataset (Fuseki)

If you are using the Fuseki interface:

- Navigate to “manage datasets” and click “add one”
- Give any name to the dataset, like “exercisedb”
- Within “manage datasets” click “upload data”.
- Select and upload “Followers.ttl”
 - You should see a confirmation that 8 triples were loaded.

To access the query endpoint:

- Navigate to the “dataset” and then “query” tab.
- Enter a query in the query editor
- Press the play arrow to run the query
- The results will be displayed below



Hands-on Exercises - Query the dataset

Try running the query on the right.

You should get two results: **ex:b0** and **ex:b1**.

```
prefix ex: <http://example.com/>

SELECT distinct ?follower

WHERE {
    ?follower ex:name "Bill" .
}
```

Visualise the Dataset (GraphDB)

- Navigate to “Explore”, “Graph overview”
- Click on “The default graph”.
- Click on one of the nodes of the triples, such as **ex:a0**.
- Click “Visual graph” from the right hand side.
- You should now see the three nodes on the right of this slide.
- You can move the nodes around.
- By clicking on a node you can see their properties (e.g. the name of the person)
- By double clicking (or hovering over the node and clicking the expand button) on a node you can expand the connections of that node.
 - Try expanding **ex:c0** to reveal the node **ex:b1**.





Visualise the Dataset (Fuseki)

If you are using Fuseki you can still try to use some online visualisation tools:

- <https://www.w3.org/RDF/Validator/> (this tool requires RDF in XML format)
- <http://www.ldf.fi/service/rdf-grapher> (this tool can use RDF in the Turtle format)

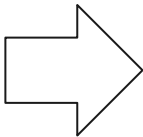
However these visualisations tools are not interactive, and their outputs might not be very legible.

Visualise the Dataset (Fuseki)

Transform the data in our example from Turtle to RDF/XML using: <http://www.easyrdf.org/converter>

prefix ex: <http://example.com/>

ex:a0 ex:follows ex:c0 .
ex:a0 ex:follows ex:b0 .
ex:a0 ex:name "Adam" .
ex:b0 ex:follows ex:a0 .
ex:b0 ex:name "Bill" .
ex:c0 ex:name "Cindy" .
ex:c0 ex:follows ex:b1 .
ex:b1 ex:name "Bill" .



```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ex="http://example.com/">

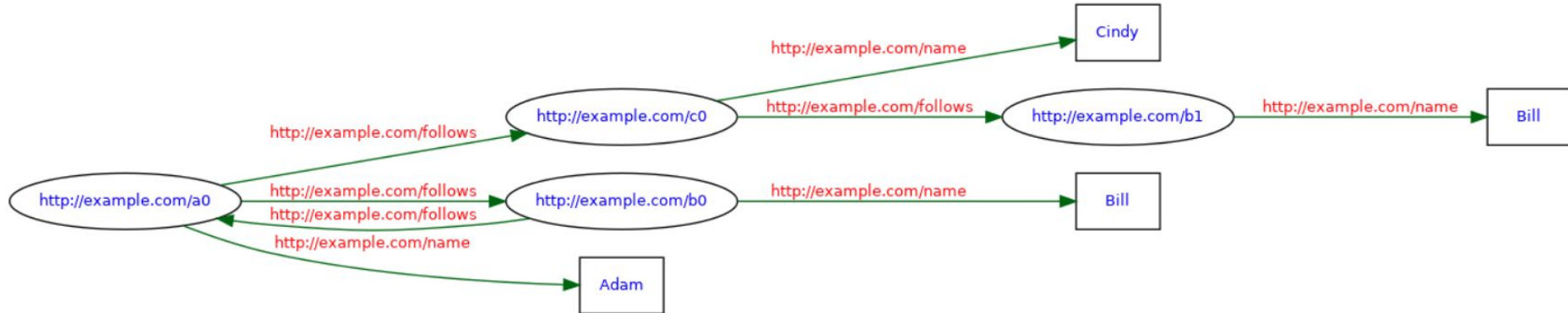
  <rdf:Description rdf:about="http://example.com/a0">
    <ns0:follows>
      <rdf:Description rdf:about="http://example.com/c0">
        <ns0:name>Cindy</ns0:name>
      <ns0:follows>
        <rdf:Description rdf:about="http://example.com/b1">
          <ns0:name>Bill</ns0:name>
        </rdf:Description>
      </ns0:follows>
    </rdf:Description>
  </ns0:follows>

  <ns0:follows>
    <rdf:Description rdf:about="http://example.com/b0">
      <ns0:follows rdf:resource="http://example.com/a0"/>
      <ns0:name>Bill</ns0:name>
    </rdf:Description>
  </ns0:follows>

  <ns0:name>Adam</ns0:name>
</rdf:Description>
</rdf:RDF>
```

Visualise the Dataset (Fuseki)

- Validate and visualise the RDF/XML data using <https://www.w3.org/RDF/Validator/>
- Select a “Graph” option under “Triples and/or Graph”, and then click “Parse RDF”





Automatic Inferences (GraphDB)

If you are using GraphDB, you might remember that when you created a database you enabled *RDFS* inferences.

We can now test this by running the following query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.com/>
select ?follower where {
    ?follower rdf:type ex:Person .
}
```

This query should yield no results, as we have not provided our database with a machine readable schema.



Automatic Inferences (GraphDB)

Now upload (into the same dataset) the schema information about our example.

You can find the schema in the attached file *FollowersSchema.ttl*.

The contents of this file are listed here on the right for your convenience:

Now try again the query from the previous slide.

- The query engine can now correctly infer that the four persons from our previous example are of class **ex:Person**.

Try and modify the query to retrieve the entities of class **ex:Follower**.

- The query engine can correctly infer that all persons except **ex:b1** are followers.

FollowersSchema.ttl

```
prefix ex: <http://example.com/>
prefix rdf:
<http://www.w3.org/1999/02/22-rdf-synta
x-ns#>

ex:name rdfs:domain ex:Person .
ex:follows rdfs:domain ex:Follower .
ex:follows rdfs:range ex:Person .
ex:Follower rdfs:subclassOf ex:Person .
```



A More in Depth Look at the Specifications

The main specification documents for RDF and SPARQL are listed below:

- SPARQL 1.1 complete list of features
 - [SPARQL 1.1 Specification](#)
- RDF 1.1 complete specification
 - [RDF 1.1 Specification](#)



The RDF Specification

Familiarise yourself with the RDF specification (<https://www.w3.org/TR/rdf11-concepts/>).

You can inspect it whenever you are unsure about some details of the RDF data model.

For example, have a look at these parts of the specification:

- Recall the basic definition of an RDF graph in [Section 3](#).
- It is not always convenient to create an IRI for every node in the graph. To avoid doing so, nodes in the graph can be identified with a *blank node*. Blank nodes can be thought of as local identifiers, as they can be referenced from inside a database, but not across databases. [Section 3.4](#) of the specification describes blank nodes.
- Literal values represent the most common datatypes, such as strings, numbers, dates and boolean values. The list of datatypes supported by RDF, and their syntax, is described in [Section 5.1](#).



The RDF Turtle Syntax

RDF triples are often shown in Turtle syntax (<https://www.w3.org/TR/turtle/>).

Turtle syntax contains more legible abbreviations for blank nodes.

In this example, `_:b` is a blank node.

```
ex:a0 ex:hasAddress _:b .
_:b ex:town "London" .
_:b ex:street "Baker Street" .
_:b ex:postCode " W1U 6TL" .
```

These four triples can be formatted in a more legible way as follows:

```
ex:a0 ex:hasAddress [
    ex:town "London" ;
    ex:street "Baker Street" ;
    ex:postCode " W1U 6TL" ] .
```



The SPARQL Specification

The SPARQL query language is rich of features.

It is important to be able to quickly navigate the SPARQL specification
(<https://www.w3.org/TR/sparql11-query/>) to find out how to construct the query you want.

Familiarise yourself with this specification as we go over some of its main parts.



SPARQL - Query Forms

<https://www.w3.org/TR/sparql11-query/#QueryForms>

Queries in SPARQL follow one of four forms.

- **SELECT**. Standard queries typically follow the SELECT form, which is used to bind variables in the query to constants in the database.
- **ASK** queries are similar to the SELECT queries. However, instead of a set of variable bindings, they return only a boolean value, to indicate whether the query has at least one match on the database. ASK queries are useful to answer yes/no questions.
- **CONSTRUCT** queries are also similar to SELECT queries, but instead of returning a set of variable mappings as a results, they use these mappings to *construct* a set of RDF triples.
- **DESCRIBE** queries ask the dataset to provide information about a specific IRI. While there is no unique way to do so, this often involves providing information about all the outgoing and incoming edges of the node.

Example: ASK

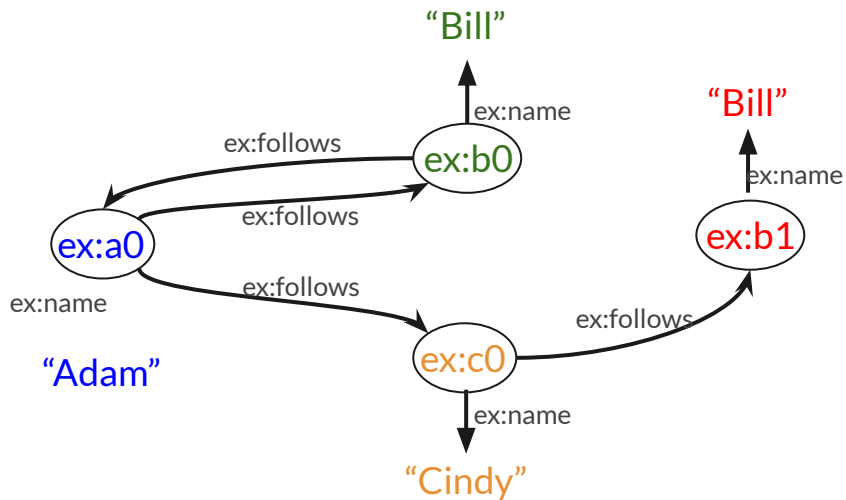
Does someone follow Cindy?

```
prefix ex: <http://example.com/>

ASK {
    ?someone ex:follows ex:c0 .
}
```

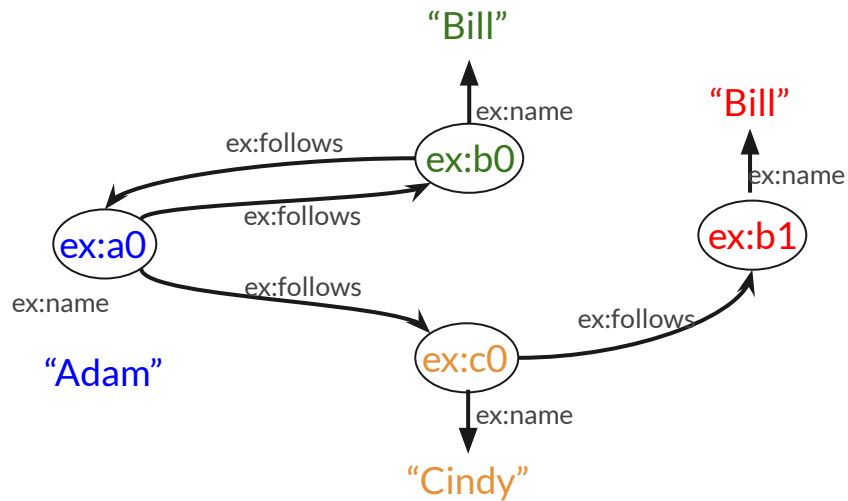
Answer:

YES



Task: ASK

Does **Bill** (**ex:b1**) follow anyone?



Task: ASK

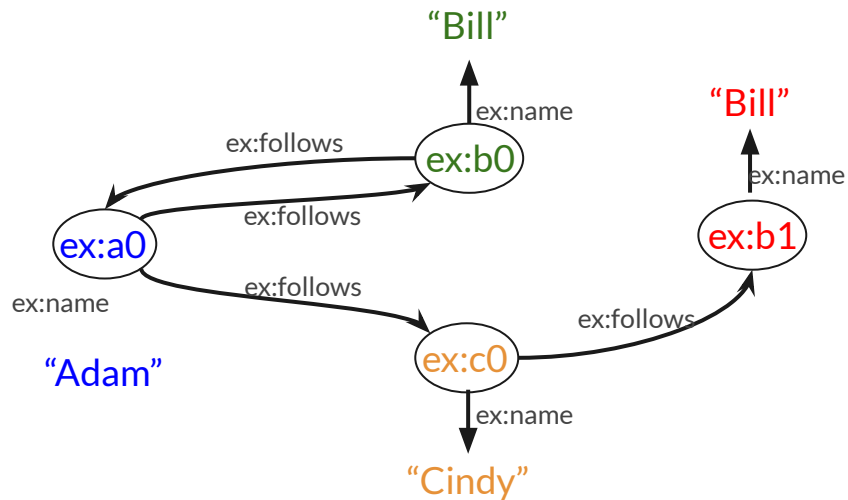
Does **Bill** (**ex:b1**) follow anyone?

```
prefix ex: <http://example.com/>

ASK {
  ex:b1 ex:follows ?someone .
}
```

Answer:

NO



Example: DESCRIBE

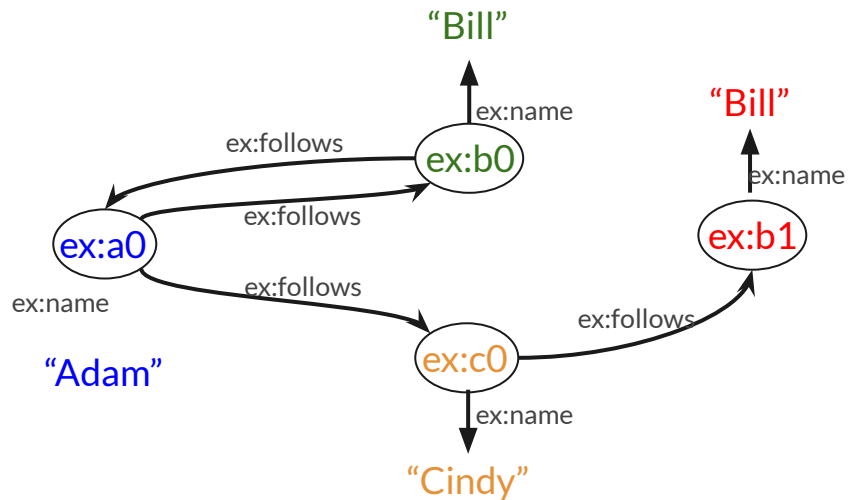
Tell me something about `ex:b0`.

```
prefix ex: <http://example.com/>
```

```
DESCRIBE ex:b0
```

Answer:

```
ex:a0 ex:follows ex:b0 .  
ex:b0 ex:follows ex:a0 .  
ex:b0 ex:name "Bill" .
```



Example: CONSTRUCT

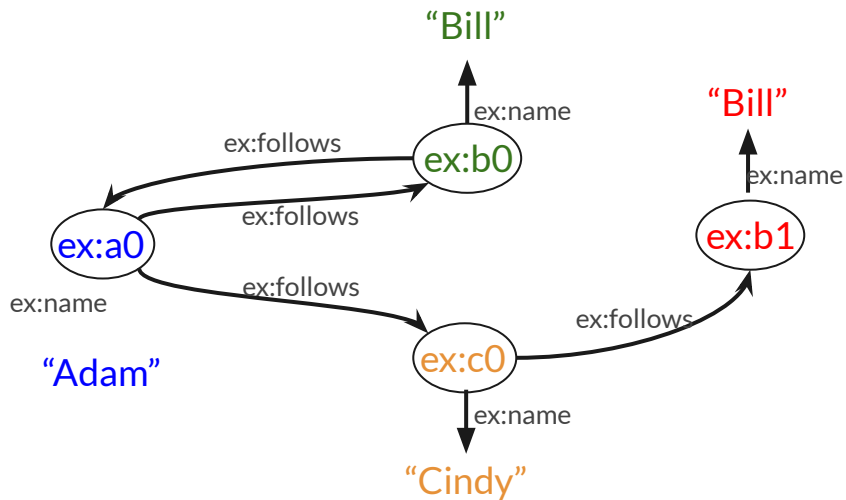
Create a dataset of 2nd degree followers. This dataset connects people with all the followers of their followers.

```
prefix ex: <http://example.com/>

CONSTRUCT {
  ?p1 ex:2ndDegreeFollower ?p3 .
}
WHERE {
  ?p1 ex:follows ?p2 .
  ?p2 ex:follows ?p3 .
}
```

Answer:

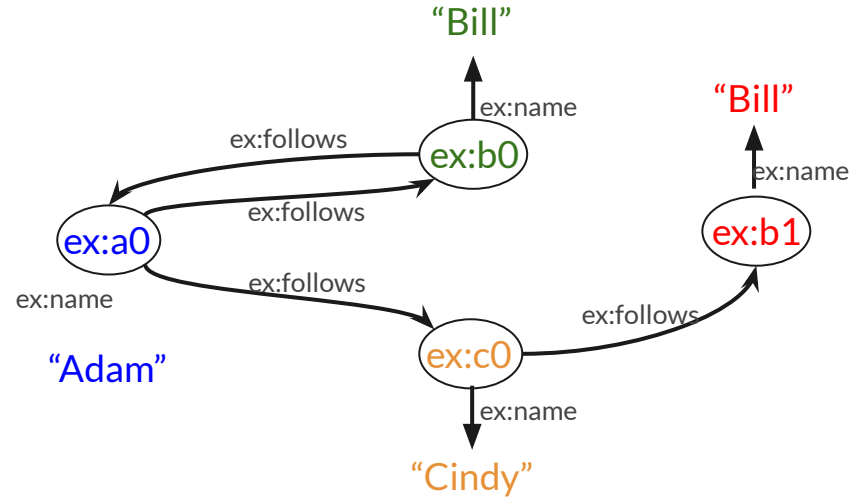
```
ex:a0 ex:2ndDegreeFollower ex:a0 .
ex:a0 ex:2ndDegreeFollower ex:b1 .
ex:b0 ex:2ndDegreeFollower ex:b0 .
ex:b0 ex:2ndDegreeFollower ex:c0 .
```



Why are **ex:a0** and **ex:b0** 2nd degree followers of themselves?

Task: CONSTRUCT

Create a dataset of 2nd degree followers of people named Bill.



Task: CONSTRUCT

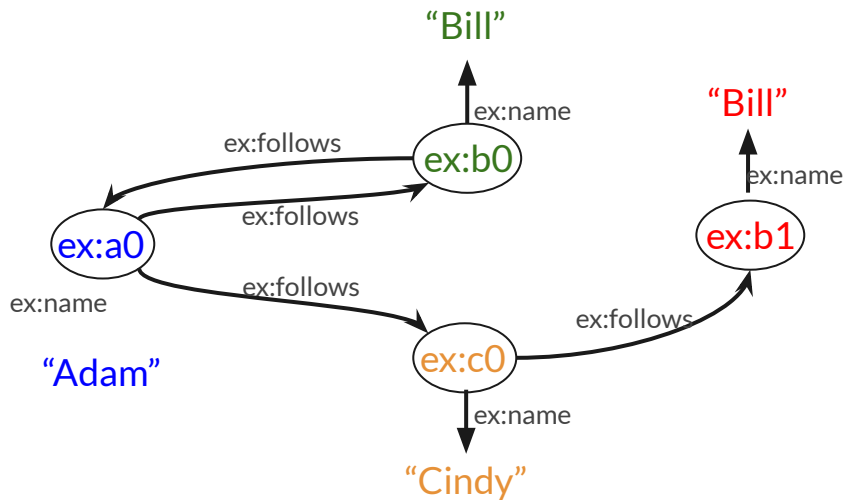
Create a dataset of 2nd degree followers of people named Bill.

```
prefix ex: <http://example.com/>

CONSTRUCT {
  ?p1 ex:2ndDegreeFollower ?p3 .
}
WHERE {
  ?p1 ex:follows ?p2 .
  ?p2 ex:follows ?p3 .
  ?p3 ex:name "Bill" .
}
```

Answer:

```
ex:a0 ex:2ndDegreeFollower ex:b1 .
ex:b0 ex:2ndDegreeFollower ex:b0 .
```





SPARQL - Advanced Graph Patterns

- The [OPTIONAL](#) operator can be used to specify optional parts of the graph pattern, which are evaluated whenever possible. However the query does not fail if the optional part is not found.
- [Negation](#) for graph patterns is done using the FILTER NOT EXISTS and the MINUS operators. You can read through Section 8 to find out more about the difference between these two forms of negation.
- Constraints on the values of literals can be specified with [FILTERS](#). These can be used to specify, for example, that a date must be later than another, or that a string must conform to a given regular expression.

Example: FILTER

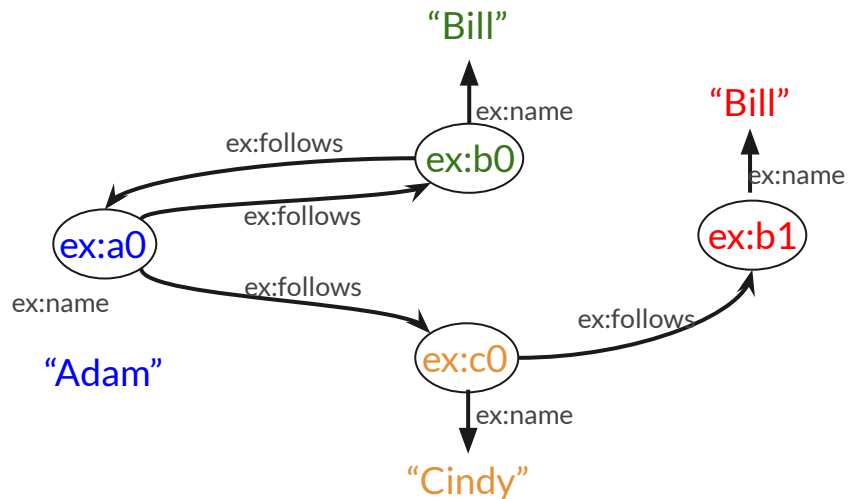
Create a dataset of 2nd degree followers where **no one** can be a second degree follower of themselves.

```
prefix ex: <http://example.com/>

CONSTRUCT {
  ?p1 ex:2ndDegreeFollower ?p3 .
}
WHERE {
  ?p1 ex:follows ?p2 .
  ?p2 ex:follows ?p3 .
  FILTER (?p1 != ?p3)
}
```

Answer:

```
ex:a0 ex:2ndDegreeFollower ex:b1 .
ex:b0 ex:2ndDegreeFollower ex:c0 .
```



Example: NEGATION

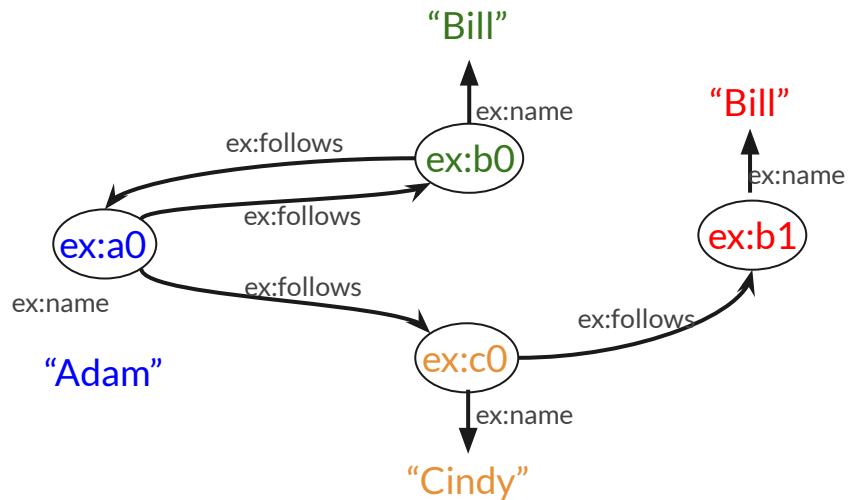
Tell me who is followed but does not follow anyone.

```
prefix ex: <http://example.com/>
```

```
SELECT ?person
WHERE {
  ?someone ex:follows ?person .
  FILTER NOT EXISTS {
    ?person ex:follows ?someoneElse .
  }
}
```

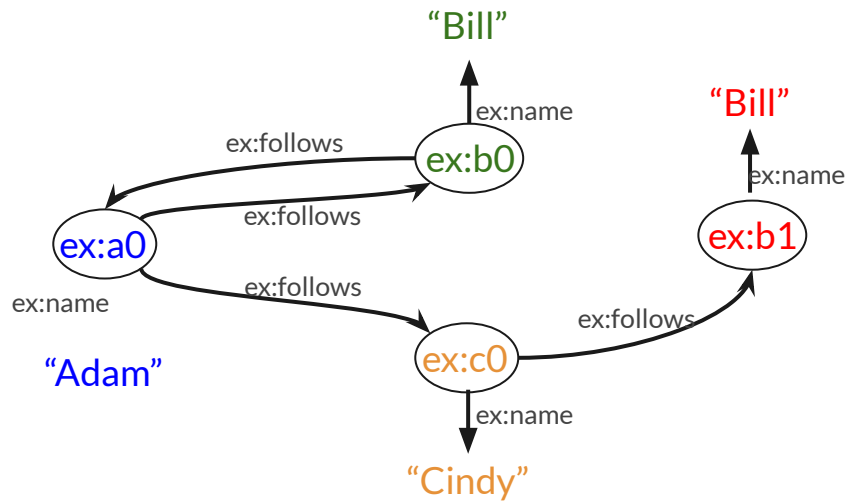
Answer:

ex:b1



Task: FILTER

Tell me who has a name which is not Bill using a filter.



Task: FILTER

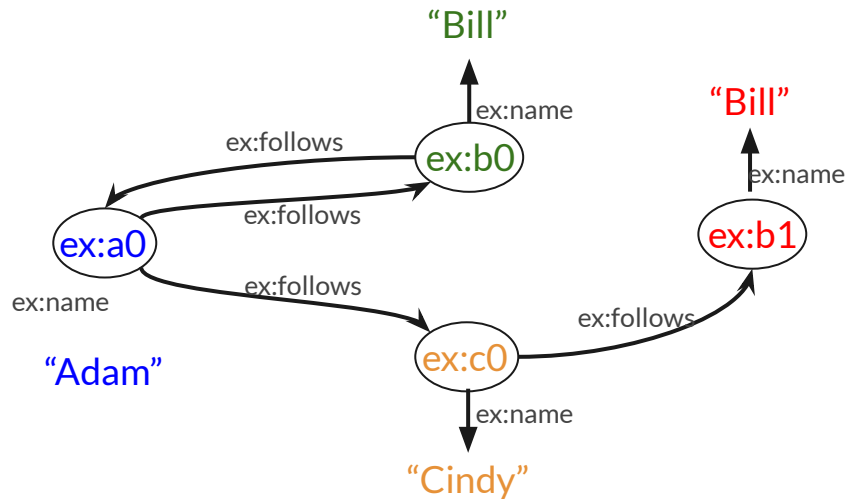
Tell me who has a name which is not Bill using a filter.

```
prefix ex: <http://example.com/>
```

```
SELECT ?person
WHERE {
  ?person ex:name ?name .
  FILTER (?name != "Bill")
}
```

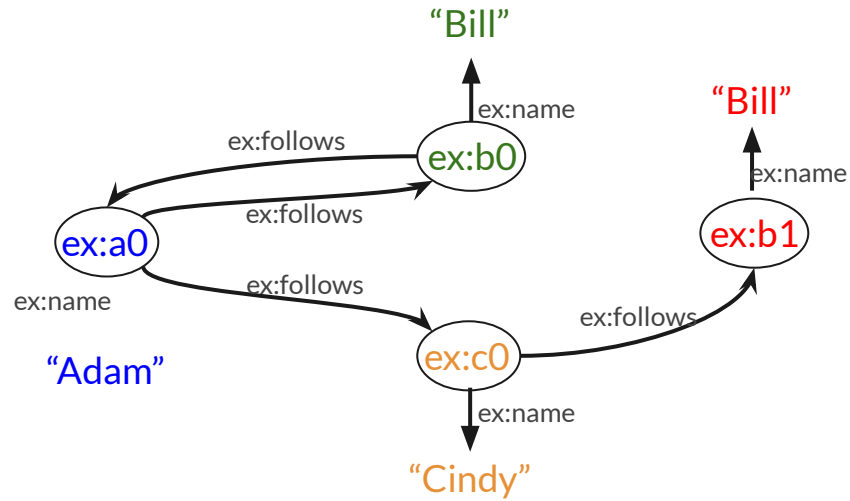
Answer:

```
ex:a0
ex:c0
```



Task: NEGATION

Tell me who has does not have a name which is Bill using negation.



Task: NEGATION

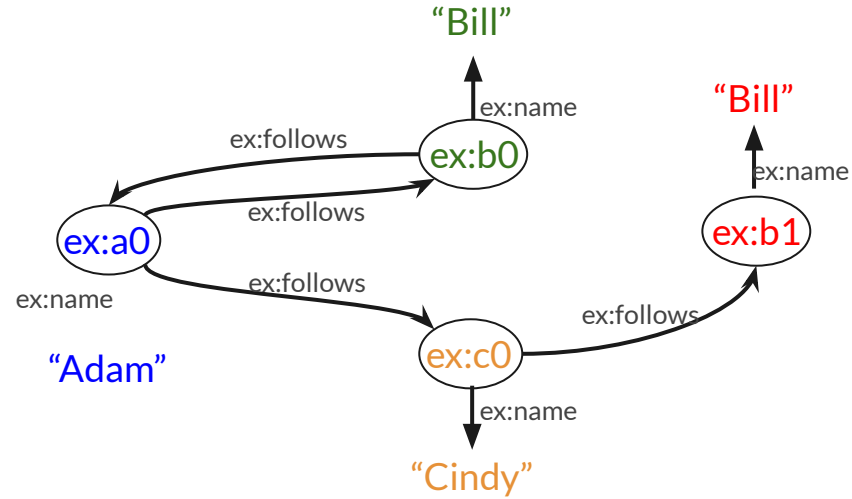
Tell me who has does not have a name which is Bill using negation.

```
prefix ex: <http://example.com/>
```

```
SELECT ?person
WHERE {
  ?person ex:name ?name .
  FILTER NOT EXISTS {
    ?person ex:name "Bill" .
  }
}
```

Answer:

```
ex:a0
ex:c0
```



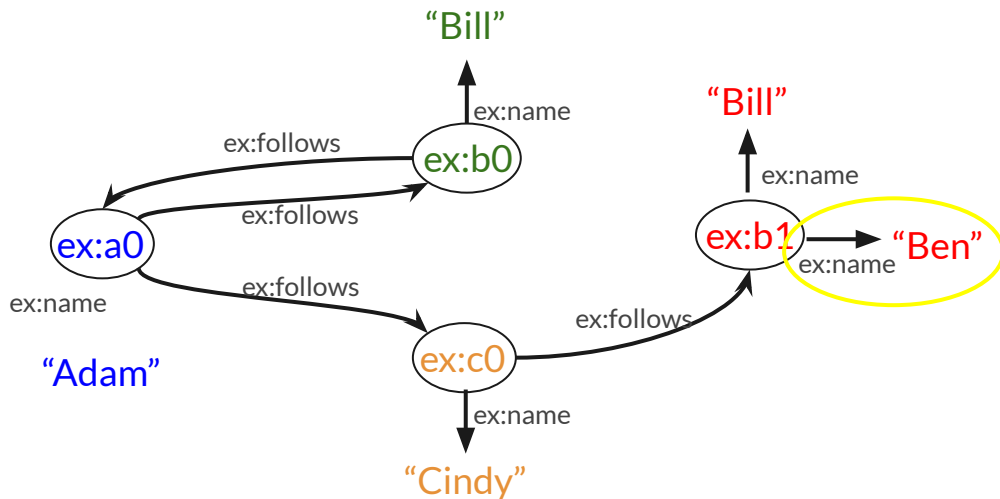
Task: Query Comparison

You might have noticed a slightly different wording of the previous two tasks:

- 1) “who has a name which is not Bill”
- 2) “who has does not have a name which is Bill”

Consider adding a second name for **ex:b1**, as in the following example. Would the results of the previous two queries differ?

Optional task: try creating another dataset with the additional name for **ex:b1** and verify the results of the previous two queries.



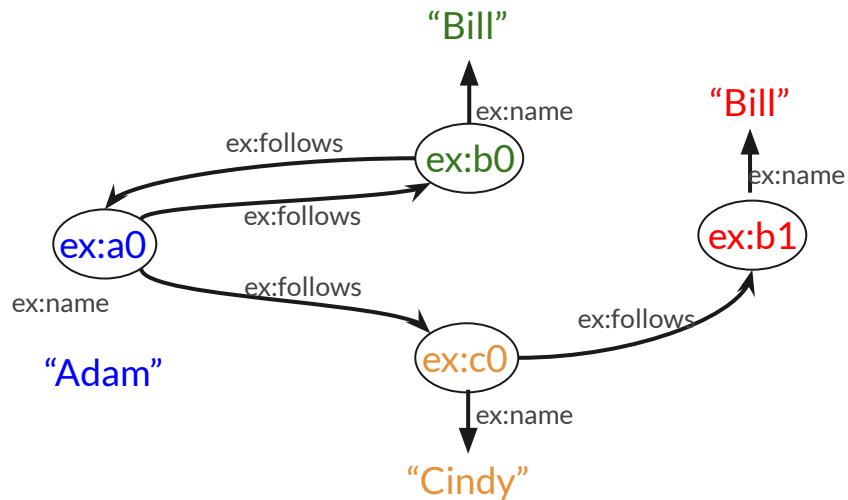
Example: OPTIONAL

Give me the name of everyone,
and optionally who they follow.

```
prefix ex: <http://example.com/>
```

```
SELECT *  
WHERE {  
  ?person ex:name ?name .  
  OPTIONAL {  
    ?person ex:follows ?someone .  
  }  
}
```

Answer:



	person	name	someone
1	ex:a0	Adam	ex:c0
2	ex:a0	Adam	ex:b0
3	ex:c0	Cindy	ex:b1
4	ex:b0	Bill	ex:a0
5	ex:b1	Bill	



SPARQL - Solution Modifiers

- You can limit the number of answers using the [LIMIT](#) modifier.
- You can [ORDER](#) the results in some ascending or descending order.
- You can [aggregate](#) the results by counting them, summing values, averaging, etc.

Solution Modifiers

Give me the all the named individuals, in ascending order of how many people they follow.

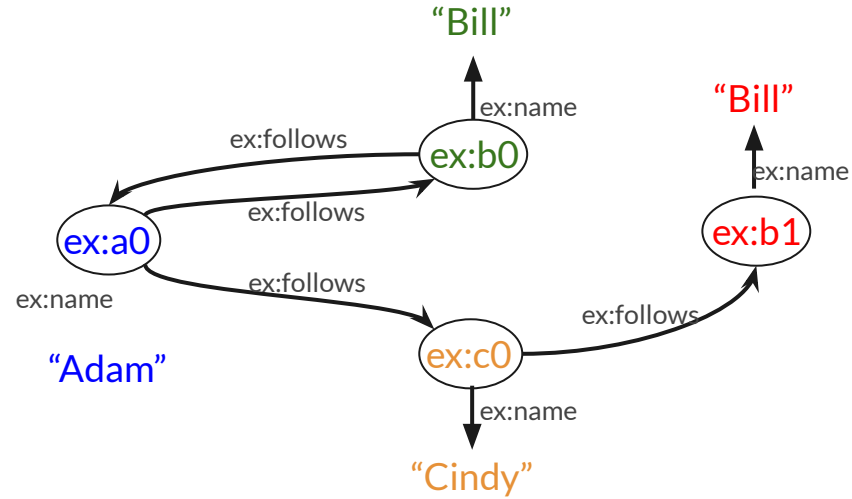
```
prefix ex: <http://example.com/>
```

```
SELECT ?person COUNT(?someone) AS ?numFollowed
WHERE {
  ?person ex:name ?name .
  OPTIONAL { ?person ex:follows ?someone . }
} GROUP BY ?person ORDER BY ASC(?numFollowed)
```

Why do we need an
OPTIONAL here?

Answer:

	person	numFollowed
1	ex:b1	"0"^^xsd:integer
2	ex:b0	"1"^^xsd:integer
3	ex:c0	"1"^^xsd:integer
4	ex:a0	"2"^^xsd:integer





Hands-on Exercise: Trucks and Warehouses



Exercise Overview

In this exercise we will work with a more complex dataset.

- We will start by loading the dataset.
- We will perform simple queries to explore and understand the dataset better.
- We will work our way towards more complex queries capable of computing valid routes on a road network subject to multiple constraints.



Exercise Setup

We can reuse the same dataset from our previous exercise.

Load the attached dataset (also copied here to the right for your convenience):

- *TrucksAndWarehouses.ttl*

There is no need to understand what the triples mean, at this stage.

```
prefix ex: <http://example.com/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
ex:OwnTruck rdf:type ex:Truck .
ex:TruckAlpha rdf:type ex:Truck .
ex:TruckBeta rdf:type ex:Truck .
ex:OwnTruck rdfs:label "My Truck" .
ex:TruckAlpha rdfs:label "Adam's Truck" .
ex:TruckAlpha ex:ownedBy ex:a0 .
ex:TruckBeta rdfs:label "Cindy's Truck" .
ex:TruckBeta ex:ownedBy ex:c0 .
ex:warehouse1 rdf:type ex:Warehouse .
ex:warehouse2 rdf:type ex:Warehouse .
ex:warehouse3 rdf:type ex:Warehouse .
ex:warehouse4 rdf:type ex:Warehouse .
ex:warehouse5 rdf:type ex:Warehouse .
ex:warehouse1 rdfs:label "Amsterdam's Depot" .
ex:warehouse2 rdfs:label "Berlin's Store" .
ex:warehouse3 rdfs:label "Vienna's Storehouse" .
ex:warehouse4 rdfs:label "Zurich's Silo" .
ex:warehouse5 rdfs:label "Paris' Warehouse" .
ex:warehouse5 ex:connectedTo ex:warehouse4 .
ex:warehouse4 ex:connectedTo ex:warehouse5 .
ex:warehouse5 ex:connectedTo ex:warehouse1 .
ex:warehouse1 ex:connectedTo ex:warehouse5 .
ex:warehouse2 ex:connectedTo ex:warehouse1 .
ex:warehouse1 ex:connectedTo ex:warehouse2 .
ex:warehouse3 ex:connectedTo ex:warehouse2 .
ex:warehouse2 ex:connectedTo ex:warehouse3 .
ex:warehouse4 ex:connectedTo ex:warehouse3 .
ex:warehouse3 ex:connectedTo ex:warehouse4 .
ex:warehouse4 ex:connectedTo ex:warehouse1 .
ex:warehouse1 ex:connectedTo ex:warehouse4 .
ex:OwnTruck ex:hasLocation ex:i1 .
ex:i1 ex:warehouse ex:warehouse5 .
ex:i1 ex:day 1 .
ex:TruckAlpha ex:hasLocation ex:i2 .
ex:i2 ex:warehouse ex:warehouse2 .
ex:i2 ex:day 1 .
ex:TruckAlpha ex:hasLocation ex:i3 .
ex:i3 ex:warehouse ex:warehouse1 .
ex:i3 ex:day 2 .
ex:TruckAlpha ex:hasLocation ex:i4 .
ex:i4 ex:warehouse ex:warehouse5 .
ex:i4 ex:day 3 .
ex:TruckAlpha ex:hasLocation ex:i5 .
ex:i5 ex:warehouse ex:warehouse4 .
ex:i5 ex:day 4 .
ex:TruckBeta ex:hasLocation ex:i6 .
ex:i6 ex:warehouse ex:warehouse4 .
ex:i6 ex:day 1 .
ex:TruckBeta ex:hasLocation ex:i7 .
ex:i7 ex:warehouse ex:warehouse3 .
ex:i7 ex:day 2 .
ex:TruckBeta ex:hasLocation ex:i8 .
ex:i8 ex:warehouse ex:warehouse3 .
ex:i8 ex:day 3 .
ex:TruckBeta ex:hasLocation ex:i9 .
ex:i9 ex:warehouse ex:warehouse3 .
ex:i9 ex:day 4 .
```



Task 1

One of the first steps in inspecting and RDF databases is to find out they type of classes that it describes. Create and run a query to retrieve all the classes from the dataset.



Task 1

One of the first steps in inspecting and RDF databases is to find out they type of classes that it describes. Create and run a query to retrieve all the classes from the dataset.

Note: the solution for each task is shown in the slide after the task. You should not skip ahead until you have tried to solve the task yourself.



Task 1 - Solution

One of the first steps in inspecting and RDF databases is to find out they type of classes that it describes. Create and run a query to retrieve all the classes from the dataset.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?class
WHERE {
    ?instance rdf:type ?class .
}
```



Task 2

Let's find out more about the entities of one particular class.

Query the dataset to retrieve all instances of class ex:Warehouse.



Task 2 - Solution

Let's find out more about the entities of one particular class.

Query the dataset to retrieve all instances of class `ex:Warehouse`.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.com/>
SELECT ?instance
WHERE {
    ?instance rdf:type ex:Warehouse .
}
```



Task 3

We can now find out more about a specific instance.

Query the dataset to describe entity **ex:warehouse1**



Task 3 - Solution

We can now find out more about a specific instance.

Query the dataset to describe entity **ex:warehouse1**

```
PREFIX ex: <http://example.com/>
```

```
DESCRIBE ex:warehouse1
```




Task 4

It looks like the dataset describes warehouses connected to each other through some roads. In order to find out more about these connections, let's construct all the triples that have **ex:connectedTo** as predicate.



Task 4 - Solution

It looks like the dataset describes warehouses connected to each other through some roads. In order to find out more about these connections, let's construct all the triples that have **ex:connectedTo** as predicate.

```
PREFIX ex: <http://example.com/>
CONSTRUCT {
    ?w1 ex:connectedTo ?w2 .
}
WHERE {
    ?w1 ex:connectedTo ?w2 .
}
```



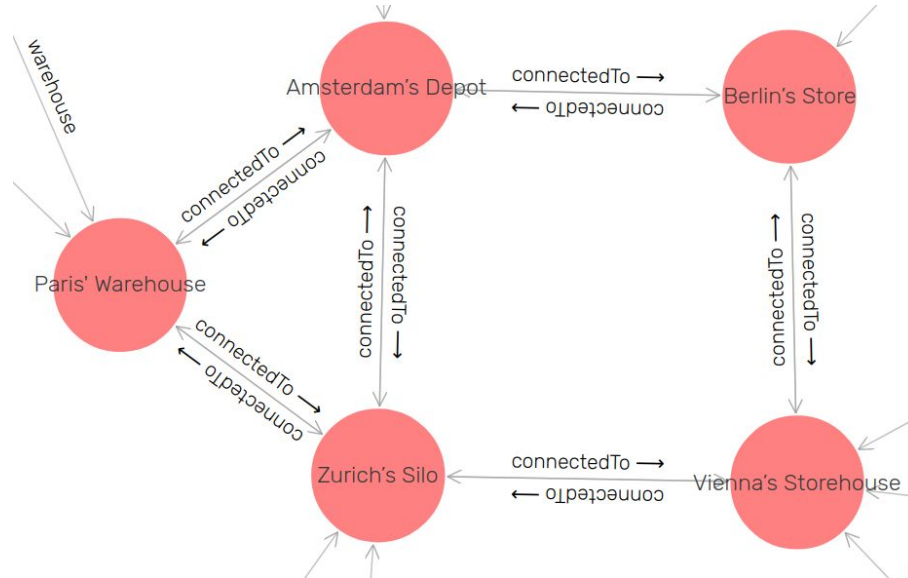
Task 5

In the previous task you have extracted a subset of the dataset that describes the connections between warehouses. Visualise these connections using the visualisation tool presented earlier in this course.

- For GraphDB, you can just start by focussing on one of the warehouses in the visualisation tool, and then expand its connections.
- If you are using Fuseki and the online visualisation tools, try to visualise only the triples you have constructed using the previous query. If you try to visualise the whole dataset the output might be too chaotic.

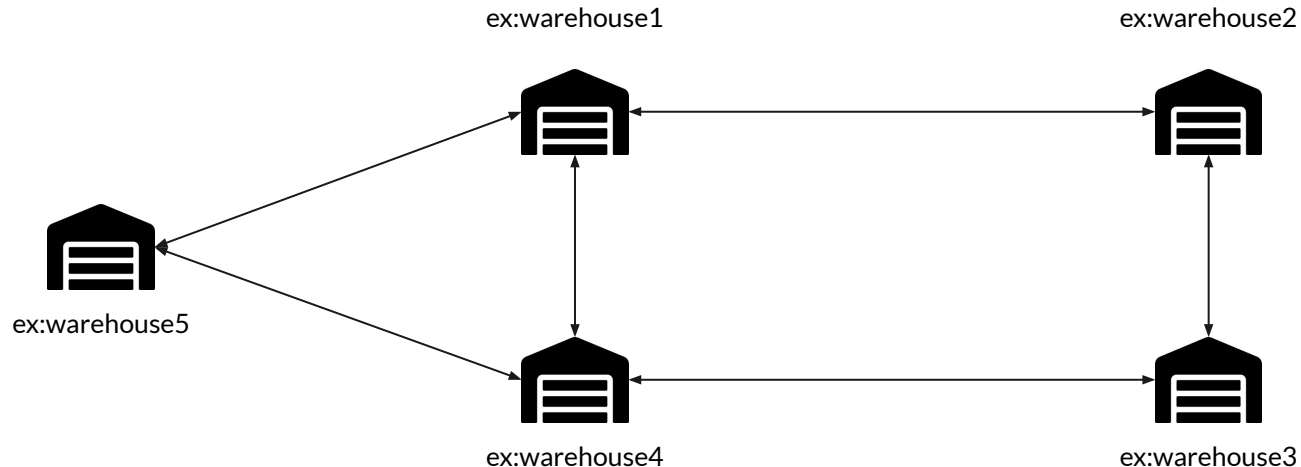
Task 5 - Solution (GraphDB)

In the previous task you have extracted a subset of the dataset that describes the connections between warehouses. Visualise these triples using the visualisation tool presented earlier in this course.



Task 5 - Solution (simplified view)

In the previous task you have extracted a subset of the dataset that describes the connections between warehouses. Visualise these triples using the visualisation tool presented earlier in this course.





Task 6

Query all the instances of class **ex:Truck**



Task 6 - Solution

Query all the instances of class `ex:Truck`

```
PREFIX ex: <http://example.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?truck
WHERE {
    ?truck rdf:type ex:Truck .
}
```



Task 7

There are only 3 trucks described in the dataset.

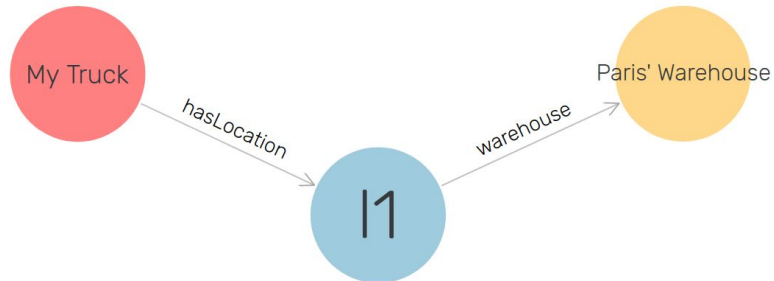
Use the describe function to find out more about **ex:OwnTruck**.

Task 7 - Solution (and sample visualisation)

There are only 3 trucks described in the dataset.

Use the describe function to find out more about `ex:OwnTruck`.

```
PREFIX ex: <http://example.com/>  
DESCRIBE ex:OwnTruck
```





Task 8

Let's assume that today is day number 1 (in a more realistic example, this could be a full date or timestamp). The data from the previous task shows that **ex:OwnTruck** today is located in warehouse 5.

Now use the describe function to find out more about **ex:TruckAlpha**.



Task 8 - Solution

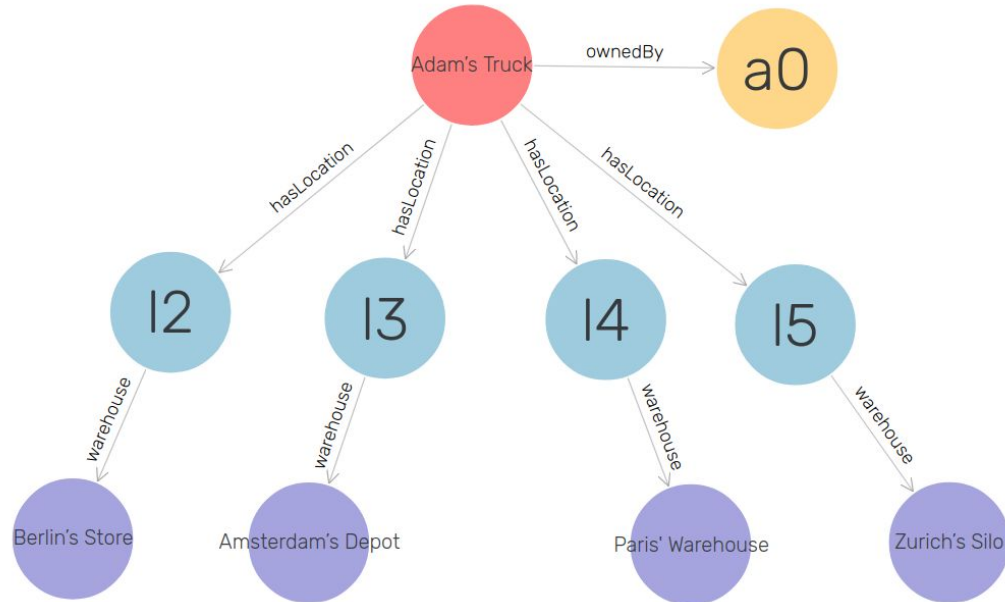
Let's assume that today is day number 1 (in a more realistic example, this could be a full date or timestamp). The data from the previous task shows that **ex:OwnTruck** today is located in warehouse 5.

Now use the describe function to find out more about **ex:TruckAlpha**.

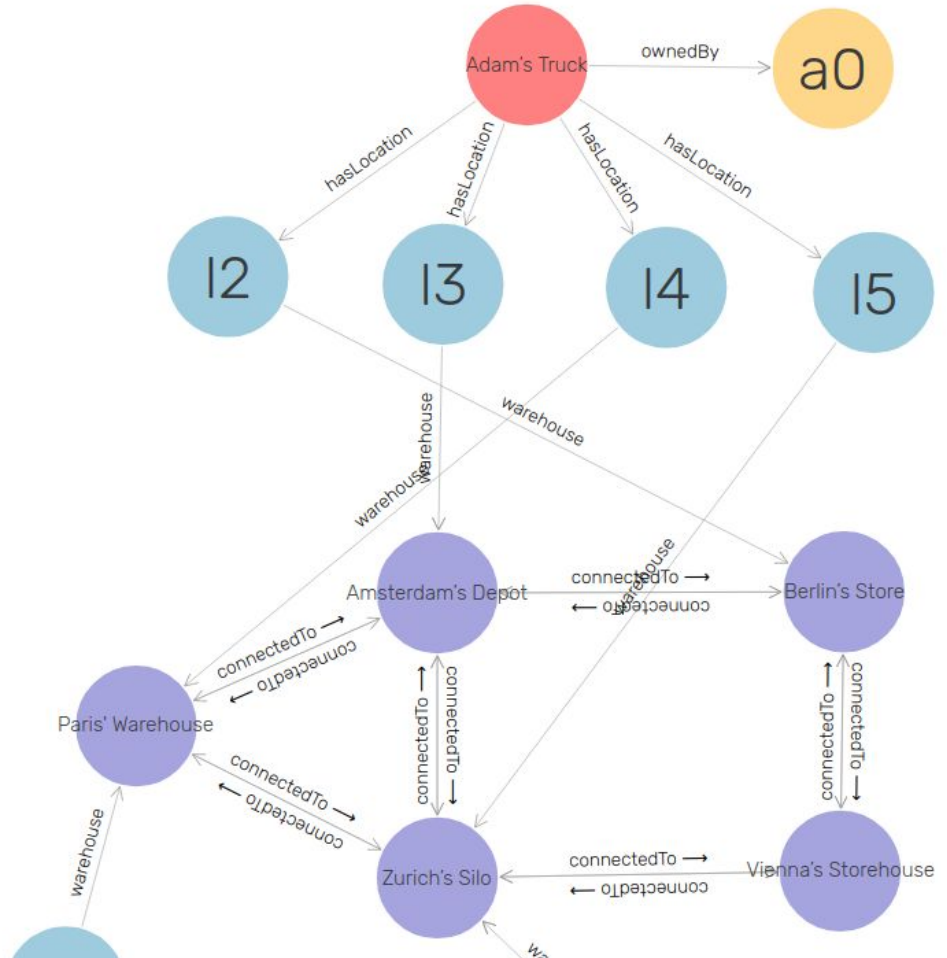
```
PREFIX ex: <http://example.com/>
```

```
DESCRIBE ex:TruckAlpha
```

Task 8 - Solution (sample visualisation)



Task 8 - Solution (sample visualisation)





Task 9

The data from the previous task shows that **ex:TruckAlpha** today is located in warehouse 2. Moreover, we also have location information about where this truck will be in the next few days. More specifically, on day 2 it will be in warehouse 1, on day 3 it will be in warehouse 5, and finally on day 4 it will be in warehouse 4.

Query the database to find out where each truck is on day 1.



Task 9 - Solution

The data from the previous task shows that **ex:TruckAlpha** today is located in warehouse 2. Moreover, we also have location information about where this truck will be in the next few days. More specifically, on day 2 it will be in warehouse 1, on day 3 it will be in warehouse 5, and finally on day 4 it will be in warehouse 4.

Query the database to find out where each truck is on day 1.

```
PREFIX ex: <http://example.com/>
SELECT ?truck ?warehouse
WHERE {
    ?truck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
}
```



Task 10

Now query where each truck will be on day 2.



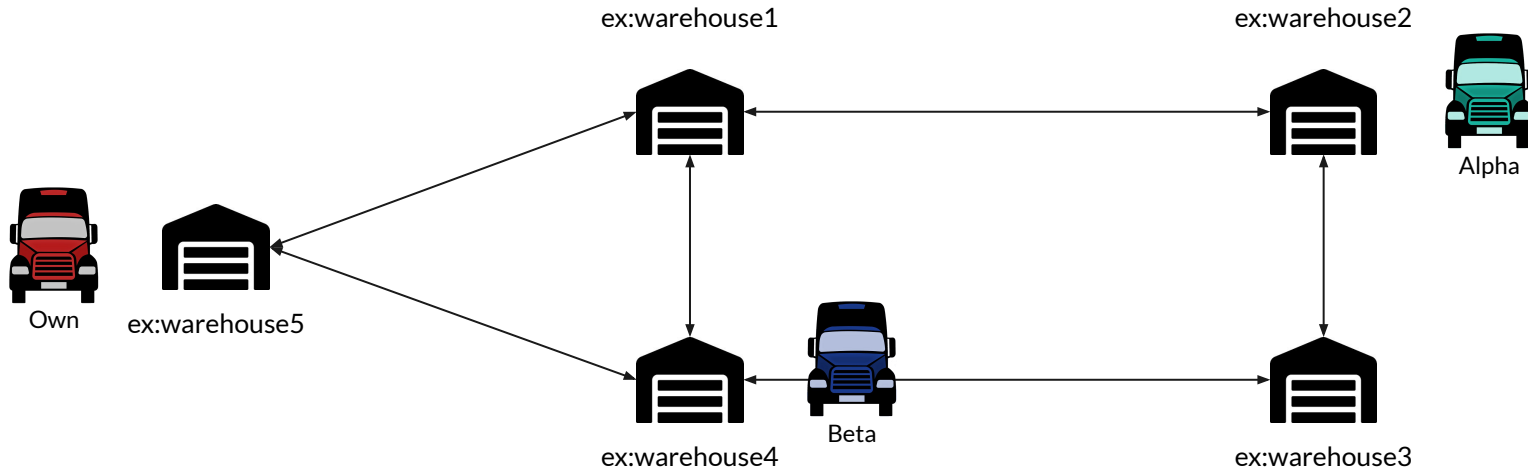
Task 10 - Solution

Now query where each truck will be on day 2.

```
PREFIX ex: <http://example.com/>
SELECT ?truck ?warehouse
WHERE {
    ?truck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 2 .
}
```

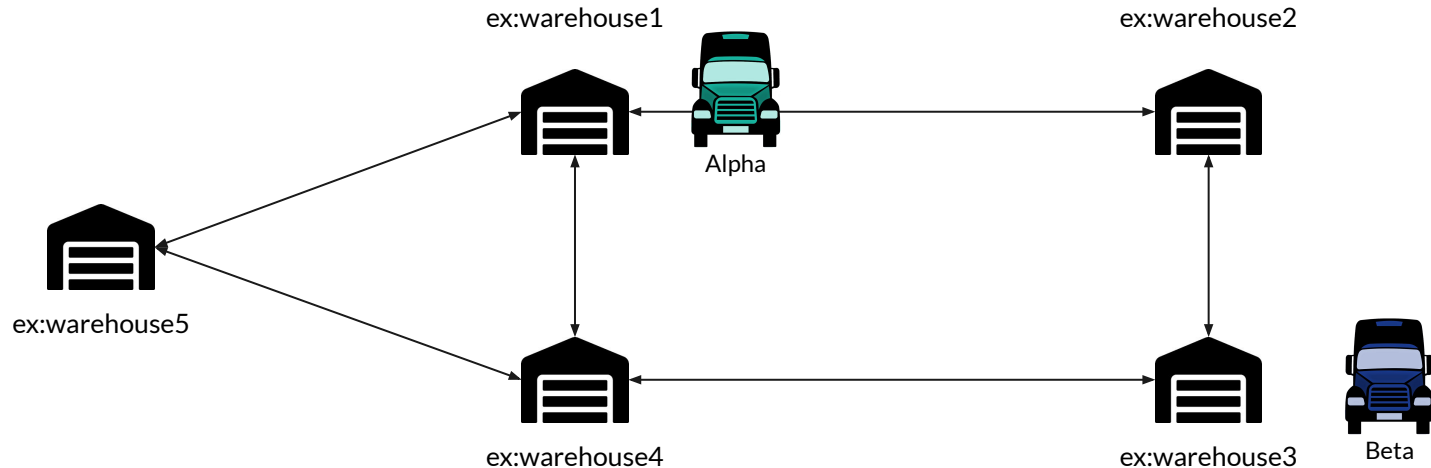
Summary Schematic Visualisation

Position of trucks on day 1.



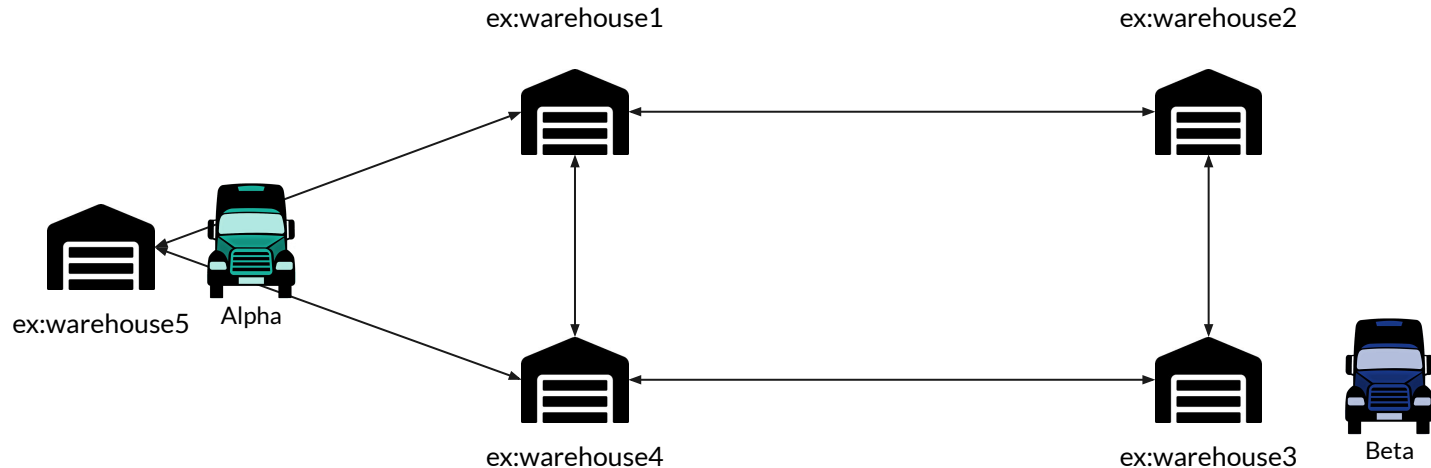
Summary Schematic Visualisation

Position of trucks on day 2.



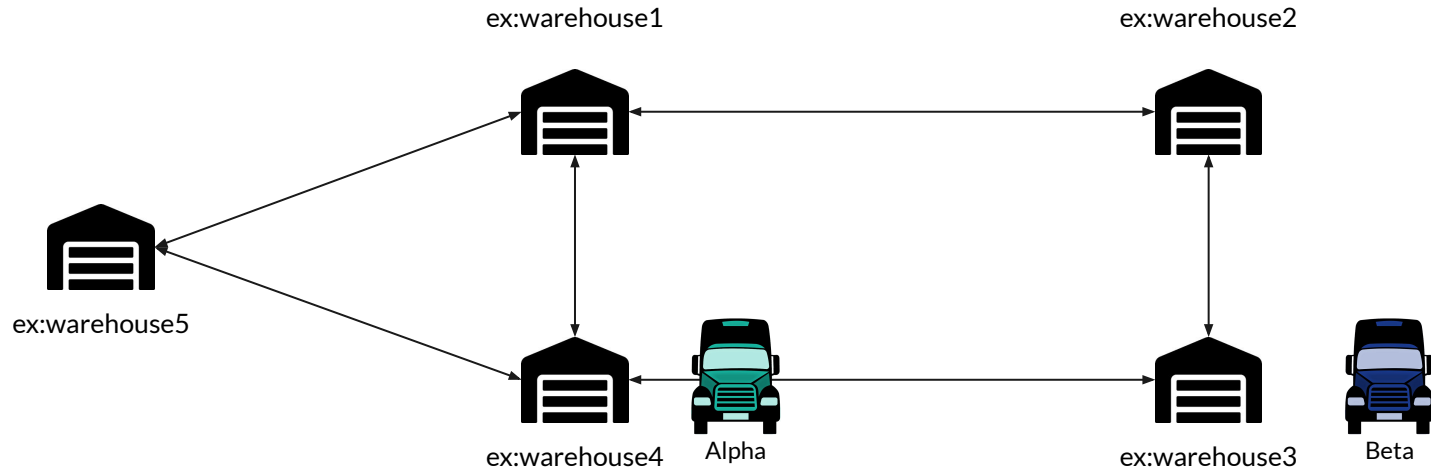
Summary Schematic Visualisation

Position of trucks on day 3.



Summary Schematic Visualisation

Position of trucks on day 4.





Task 11


As expected, we do not have information about where **ex:OwnTruck** will be on day 2.

In fact, we are told that our job is to schedule a path for our truck to reach **ex:warehouse2** on day 3.

Unfortunately, we cannot just drive our truck there directly, but we need to follow two main rules:

1. It takes one day to move a truck from a warehouse to a directly connected warehouse.
2. On any given day, only one truck can be in a warehouse.

Let's start by querying which warehouses are connected to the one where **ex:OwnTruck** currently is.



Task 11 - Solution

As expected, we do not have information about where **ex:OwnTruck** will be on day 2.

In fact, we are told that our job is to schedule a path for our truck to reach **ex:warehouse2** on day 3.

Unfortunately, we cannot just drive our truck there directly, but we need to follow two main rules:

1. It takes one day to move a truck from a warehouse to a directly connected warehouse.
2. On any given day, only one truck can be in a warehouse.

Let's start by querying which warehouses are connected to the one where **ex:OwnTruck** currently is.

```
PREFIX ex: <http://example.com/>
SELECT ?nextWarehouse
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?nextWarehouse .
}
```



Task 12

In order to find out where we could move our truck tomorrow we also need to take into account other trucks.

Extend the query from the previous task to exclude warehouses where other trucks are already scheduled to arrive on day 2.



Task 12 - Solution

In order to find out where we could move our truck tomorrow we also need to take into account other trucks.

Extend the query from the previous task to exclude warehouses where other trucks are already scheduled to arrive on day 2.

```
PREFIX ex: <http://example.com/>
SELECT ?warehouse2
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
}
```



Task 13

We can now see that the only warehouse that we can reach on day 2 is warehouse 4.

Let's extend again the query from the previous task to find out which warehouses we can reach on day 3.

Hint: you can start with the same contents of the `WHERE` clause from the previous query, and just add more lines after it to deal with day 3.



Task 13 - Solution

We can now see that the only warehouse that we can reach on day 2 is warehouse 4.

Let's extend again the query from the previous task to find out which warehouses we can reach on day 3.

Hint: you can start with the same contents of the `WHERE` clause from the previous query, and just add more lines after it to deal with day 3.

```
PREFIX ex: <http://example.com/>
SELECT ?warehouse3
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo ?warehouse3 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
}
```



Task 14

And now extend the previous query again to find out where our track could be on day 4.



Task 14 - Solution

And now extend the previous query again to find out where our track could be on day 4.

```
PREFIX ex: <http://example.com/>
SELECT ?warehouse4
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo ?warehouse3 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
    ?warehouse3 ex:connectedTo ?warehouse4 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse4 .
        ?otherLocation ex:day 4 .
    }
}
```



Task 15

As we can see, our truck can reach both warehouses 2 and 5 by day 4.

All that is left to do now is to find out a complete itinerary for our path.

Query the list of warehouses that our truck should travel to in order for it to arrive in **ex:warehouse2** on day 4.

Hint: this new query will be very similar to the previous one. You will have to modify the `SELECT` clause to return all the intermediate warehouses.

Task 15 - Solution

As we can see, our truck can reach both warehouses 2 and 5 by day 4.

All that is left to do now is to find out a complete itinerary for our path.

Query the list of warehouses that our truck should travel to in order for it to arrive in **ex:warehouse2** on day 4.

Hint: this new query will be very similar to the previous one. You will have to modify the `SELECT` clause to return all the intermediate warehouses.

```
PREFIX ex: <http://example.com/>
SELECT DISTINCT ?warehouse ?warehouse2 ?warehouse3 ?warehouse4
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo ?warehouse3 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
    ?warehouse3 ex:connectedTo ?warehouse4 .
    FILTER (?warehouse4 = ex:warehouse2)
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse4 .
        ?otherLocation ex:day 4 .
    }
}
```



Task 15 - Alternative Solution

Instead of using a filter, we can also use the name of `ex:warehouse2` in the query.

```
PREFIX ex: <http://example.com/>
SELECT DISTINCT ?warehouse ?warehouse2 ?warehouse3
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo ?warehouse3 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
    ?warehouse3 ex:connectedTo ex:warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ex:warehouse2 .
        ?otherLocation ex:day 4 .
    }
}
```




Extra Tasks

Well done! You have created a complex query that can compute valid fixed-length itineraries on any road network while factoring in the movements of any number of other trucks.

This query assumes that the truck must move every single day. In practice, however, we might find a shorter path by allowing our truck to stay in the same warehouse for more than one day.

Extra Task 1: can you extend the query to give your truck the option to remain in the same warehouse?

Extra Task 2: can you change the query to return the labels of the warehouses, instead of their IRIs?

Hint: you can find an easy solution by looking at [SPARQL property paths](#), and in particular to the question mark operator.

Extra Task 1 - Solution

You can see here a possible solution query to the extra task.

By running this query we can find out another possible path to move **ex:OwnTruck** to **ex:warehouse2**, namely by staying one extra day in **ex:warehouse5**, and then moving to **ex:warehouse1** and then **ex:warehouse2**.

```
PREFIX ex: <http://example.com/>
SELECT DISTINCT ?warehouse ?warehouse2 ?warehouse3 ?warehouse4
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo? ?warehouse2 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo? ?warehouse3 .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
    ?warehouse3 ex:connectedTo? ?warehouse4 .
    FILTER (?warehouse4 = ex:warehouse2)
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse4 .
        ?otherLocation ex:day 4 .
    }
}
```

Extra Task 2 - Solution

You can see another solution to the bonus task which returns the names of the warehouses instead of their IRIs.

The results of this query are more human understandable. The two suggested paths are the following:

1. Stay in Paris for one extra day, then drive to Amsterdam and Berlin.
2. Drive to Zurich, then Amsterdam, then Berlin.

```
PREFIX ex: <http://example.com/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?warehouseLabel ?warehouse2Label ?warehouse3Label ?warehouse4Label
WHERE {
    ex:OwnTruck ex:hasLocation ?location .
    ?location ex:warehouse ?warehouse .
    ?warehouse rdfs:label ?warehouseLabel .
    ?location ex:day 1 .
    ?warehouse ex:connectedTo? ?warehouse2 .
    ?warehouse2 rdfs:label ?warehouse2Label .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse2 .
        ?otherLocation ex:day 2 .
    }
    ?warehouse2 ex:connectedTo? ?warehouse3 .
    ?warehouse3 rdfs:label ?warehouse3Label .
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse3 .
        ?otherLocation ex:day 3 .
    }
    ?warehouse3 ex:connectedTo? ?warehouse4 .
    ?warehouse4 rdfs:label ?warehouse4Label .
    FILTER (?warehouse4 = ex:warehouse2)
    FILTER NOT EXISTS {
        ?otherTruck ex:hasLocation ?otherLocation .
        ?otherLocation ex:warehouse ?warehouse4 .
        ?otherLocation ex:day 4 .
    }
}
```



Summary

- We have learnt the basics of graph databases with RDF and SPARQL, and we have familiarised ourselves with their official specifications, which are useful reference documents.
- We gained some insights into how data can be modelled into graphs, and how we can describe its meaning both formally and informally using schemas.
- We have worked our way through a non-trivial example and experimented with complex SPARQL queries.

Further reading:

- SPARQL is an expressive query language which contains many more useful features than the ones we covered in this course, for example it can be used to modify a dataset using [SPARQL Update](#) queries.
- [Ontologies](#) and inference rules can be a powerful tool to enrich/validate a dataset, and to simplify queries.