Internet of Things

Metadata and Information Modeling

Aslak Johansen asjo@mmmi.sdu.dk

Feb 18, 2025

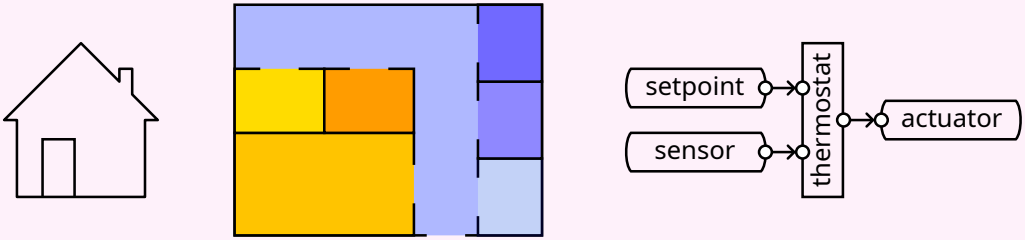
SDU

Part 0:
Introduction

Motivation ▷ A Cyber-Physical System

Consider a system consisting of

- A building with temperature sensors, actuators and setpoints per room
- A dashboard application showing a floormap colorcoded according to measured temperature
- A thermostat application which controls the temperature of each room



Motivation ▷ Connecting the Data and Control Planes

Each point (sensor, actuator and setpoint) is named through a UUID but have different access rights.

Q: How can we map the two applications to the points of a building?

1. Scatter constants throughout the code of each application
2. Introduce a per-application data structure:
   room name $\mapsto$ (sensor|actuator|setpoint) $\mapsto$ UUID
3. Share that structure

Q: What happens when we add another building?

What happens when we add another application concerned with the power consumption of the lighting of a single floor of the building?

Problem

We want to write applications for buildings.

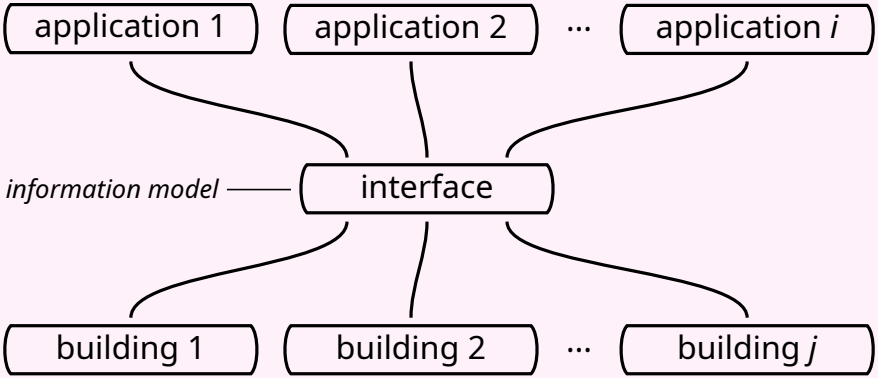
Attractive qualities:

- ***Portability*** An application can be executed on a many buildings without modification.
- ***Maintainability*** A change in a building does not translate to a need for changing an application.

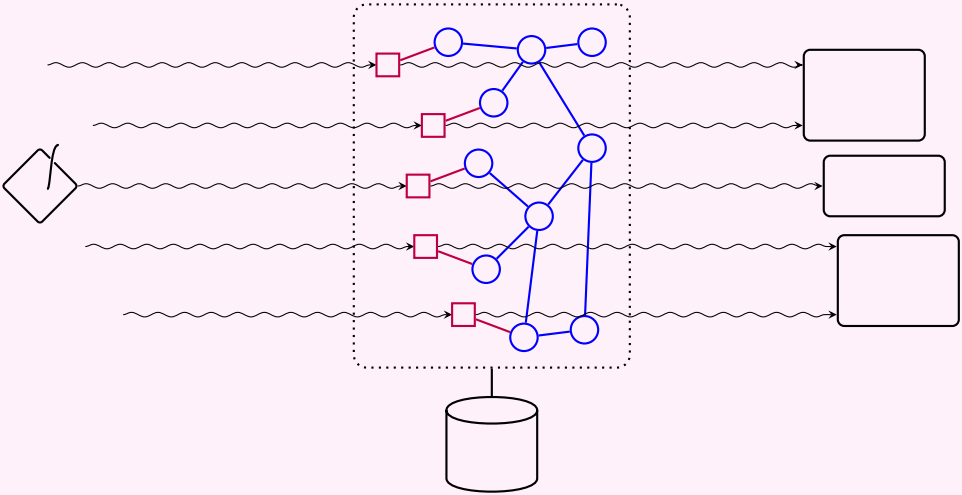
How do we accomplish this?

Approach ▷ A Narrow Waist

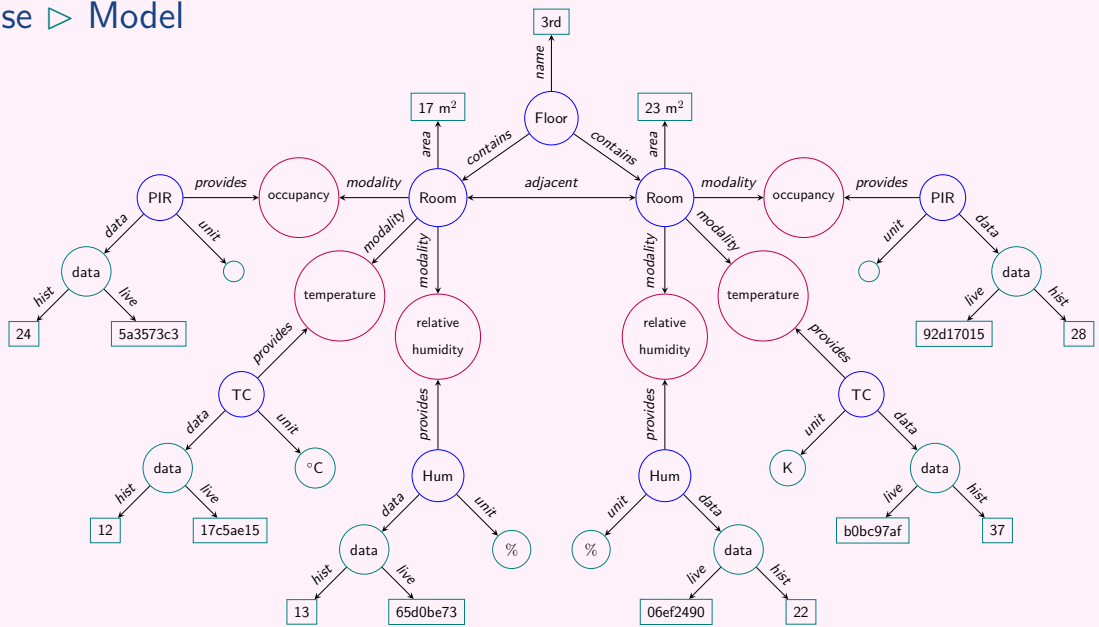
The labor intensive job of mapping out the equipment, data streams and relations between those may be shared between a portfolio of applications.



Approach ▷ Data Flow



Case ▷ Model


3rd
name
Floor
17 m²
23 m²
area
area
contains
contains
Room
adjacent
Room
modality
modality
modality
modality
modality
modality
occupancy
occupancy
temperature
temperature
relative humidity
relative humidity
PIR
PIR
provides
provides
data
data
unit
unit
hist
live
hist
live
24
5a3573c3
92d17015
28
TC
TC
provides
provides
data
data
unit
unit
hist
live
hist
live
°C
K
12
17c5ae15
b0bc97af
37
Hum
Hum
provides
provides
data
data
unit
unit
hist
live
hist
live
%
%
13
65d0be73
06ef2490
22

Case ▷ Application

Lets imagine that we have a *model* of a building and a need for an *application* that lists the absolute humidity of all rooms organized by floor.

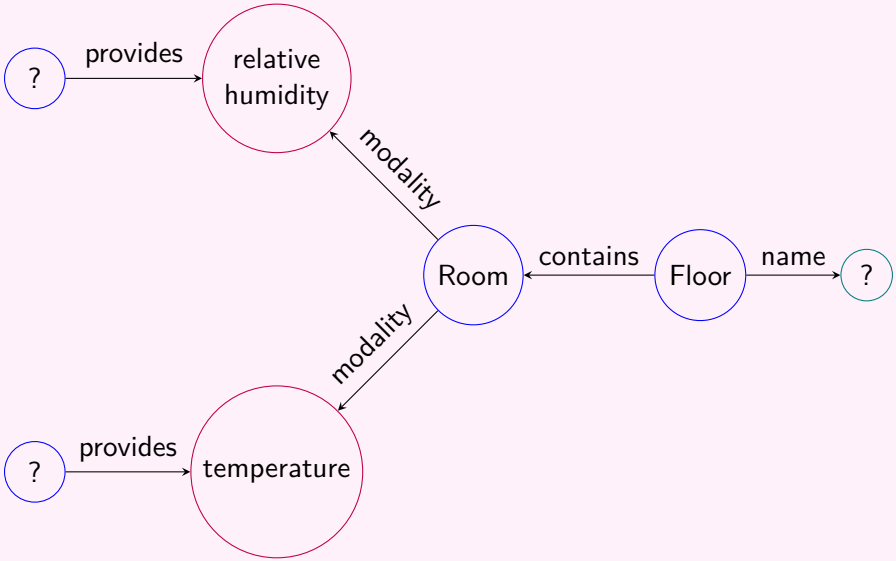
However, as we saw in the model, we don't have any absolute humidity data.

But given a temperature and a relative humidity – bot of which we do have – it is trivial to calculate the corresponding absolute humidity.

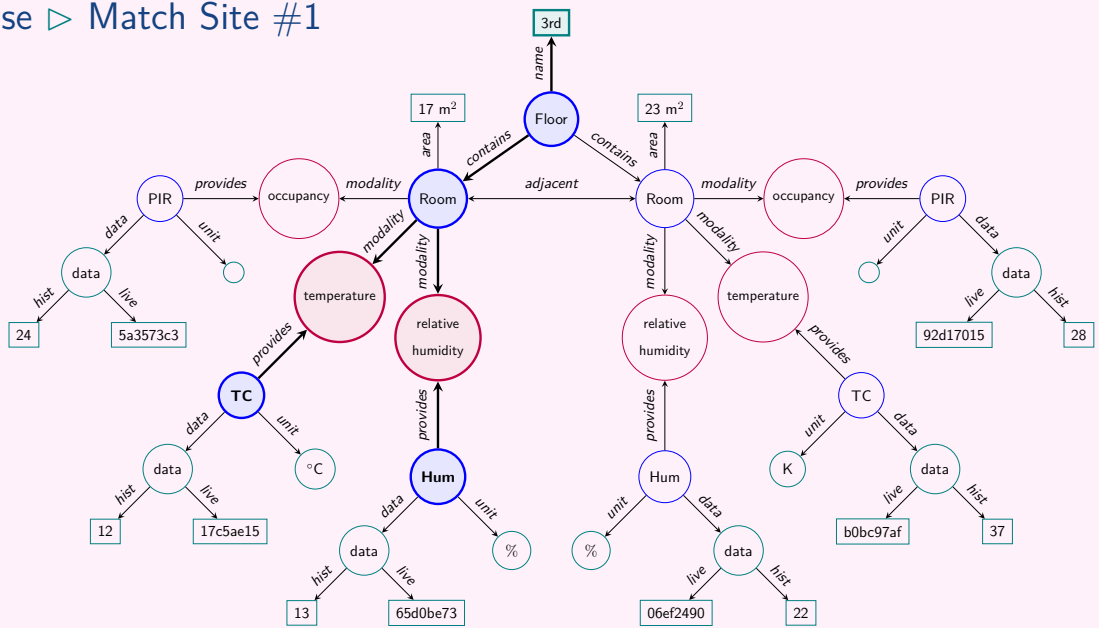
While we don't have these data *in* the model, we do have the IDs necessary for retrieving them.

We even have the option of working on *historical* or *live* data.

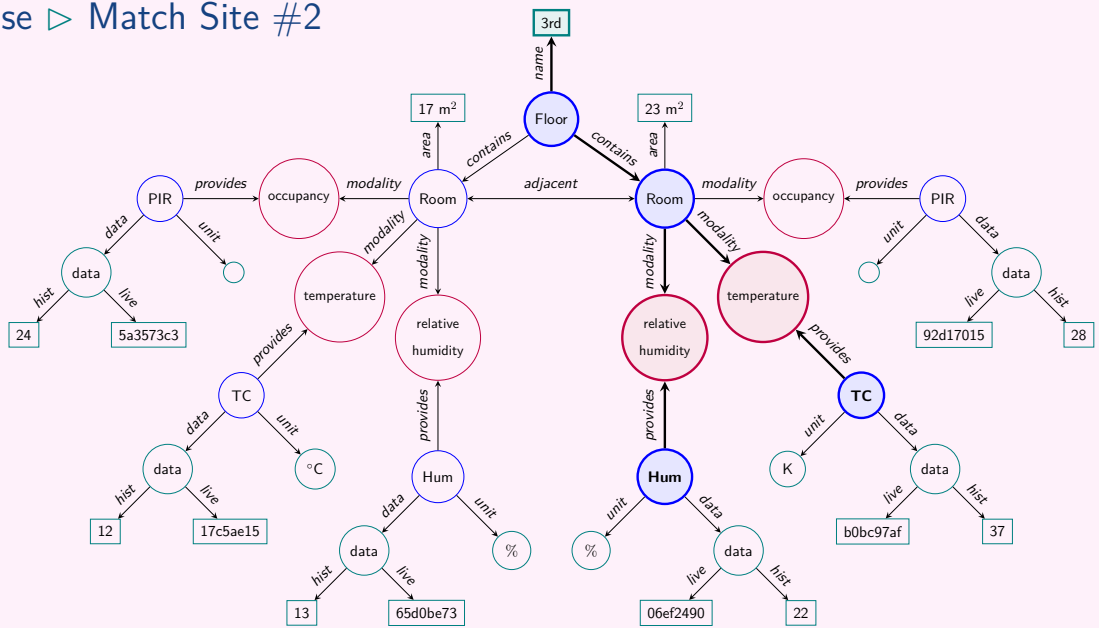
Case ▷ Query


?
provides
relative humidity
modality
Room
contains
Floor
name
?
modality
?
provides
temperature

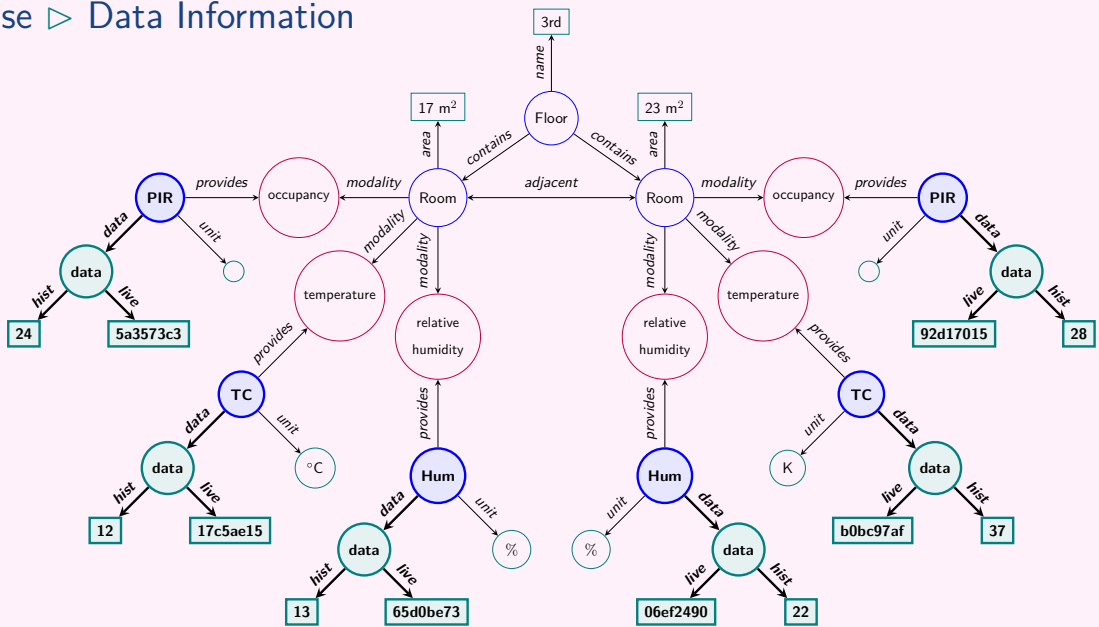
Case ▷ Match Site #1



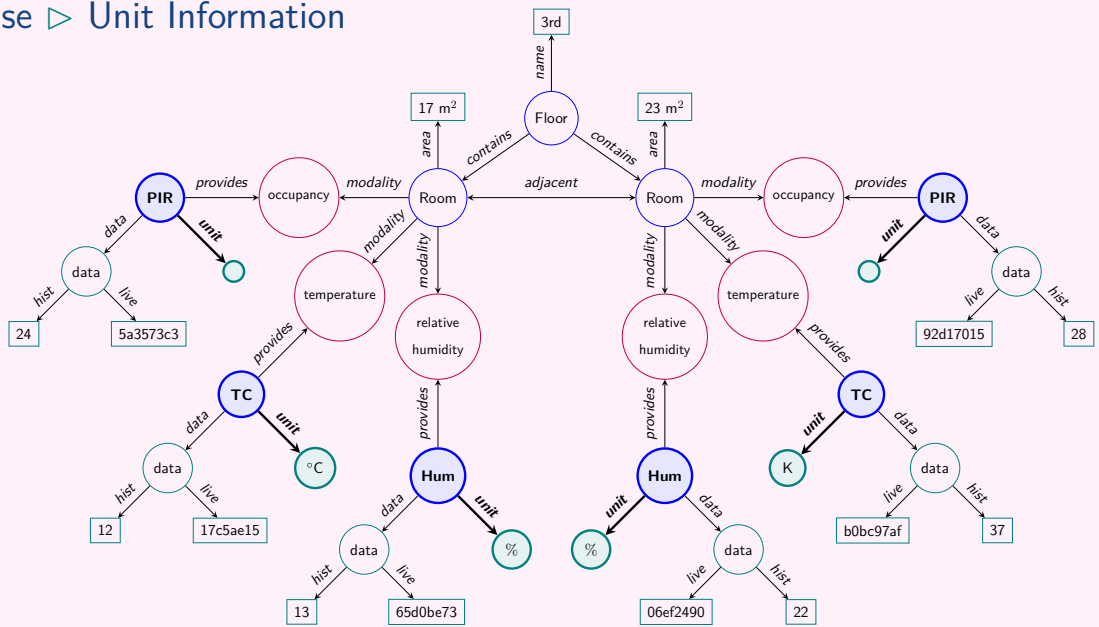
Case ▷ Match Site #2



Case ▷ Data Information



Case ▷ Unit Information



	Match Site 1	Match Site 2
Floor:	3rd	3rd
Temperature unit:	°C	K
Temperature live data:	17c5ae15	b0bc97af
Temperature historical data:	12	37
Relative humidity unit:	%	%
Relative humidity live data:	65d0be73	06ef2490
Relative humidity historical data:	13	22

Options

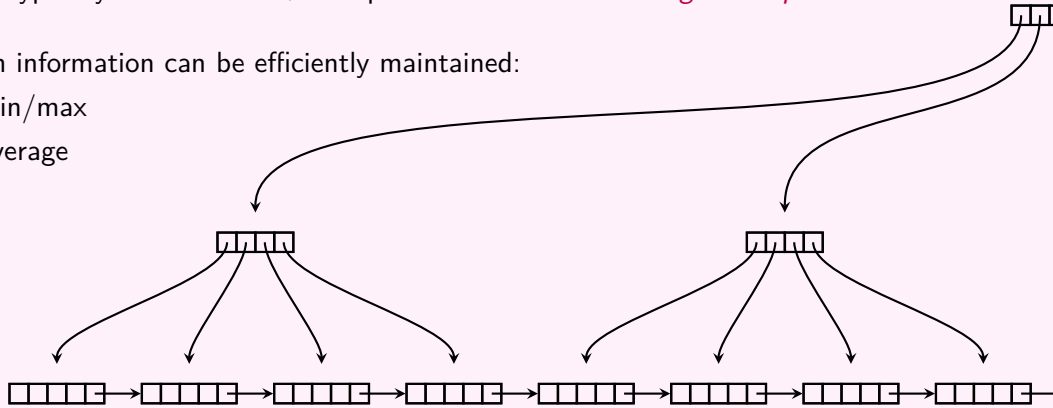
Options ▷ Timeseries Databases

A timeseries database persists a number of timeseries, each of which are *values* indexed by *timestamps*.

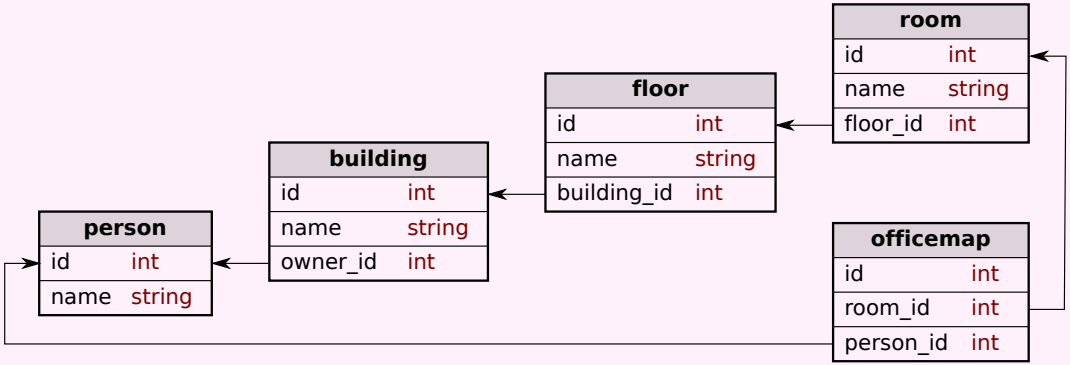
This is typically stored as a B+ tree per timeseries for fast *range lookups*.

Certain information can be efficiently maintained:

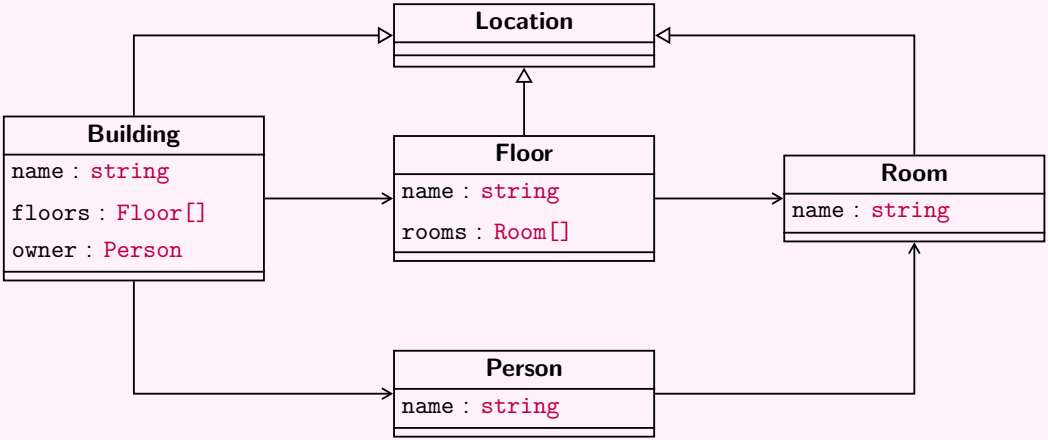
- min/max
- average



Options ▷ Relational Database



Options ▷ Object-Oriented Databases



Part 1:
RDF Ontologies

Semantics

"cec78d2a-14b8-41d6-bf46-c6a4d07574a7 is in room U182"

What is the *intension* of this statement?

- *cec78d2a-14b8-41d6-bf46-c6a4d07574a7* is a UUID.
- *cec78d2a-14b8-41d6-bf46-c6a4d07574a7* identifies some data stream.
- *U182* identifies a room.
- The data stream identified by *cec78d2a-14b8-41d6-bf46-c6a4d07574a7* originates from some equipment in the room identified by *U182*.

One of these intensions is expressed, the others are merely implied.

Ontologies

According to Wikipedia:

*"In computer science and information science, an ontology is a **formal naming and definition** of the **types, properties, and interrelationships of the entities** that really **exist in a particular domain** of discourse."*

Lets look into what this means:

- ***Formal Naming and Definition*** A formalized language is employed.
- ***Types, Properties and Interrelationships of Entities*** Things are defined through their type, parameters and relationships to other things.
- ***Entities Existing in a Particular Domain*** Ontologies have limited coverage and focus on a particular field.

  (this works because models can span multiple ontologies)

Shared Understanding

With any form of model comes some complexity that any two parties who seeks to use it as an information exchange format needs to agree on. We talk about a *shared vocabulary*.

- For an object-oriented data model, what is the notion of an attribute?
- For a relational data model, what does a table represent?

Few humans will read up on the exact definitions. Most will construct their own practical interpretations. That will usually be a fairly good match with the mental models of other developers, and when faced with a situation where this does not match we can adapt this model. Because we are flexible beings.

Machines need an exact definition, and when two machines disagree on the definitions, it results in a fault which is likely to evolve into a defect.

In this respect, programmers are *machine builders*. Once we let go of our creations they will have to stand on their own, with all of the model interpretations embued by us. And they don't share our ability to adapt when things go wrong.

Shared Understanding ▷ Defining Representation Models

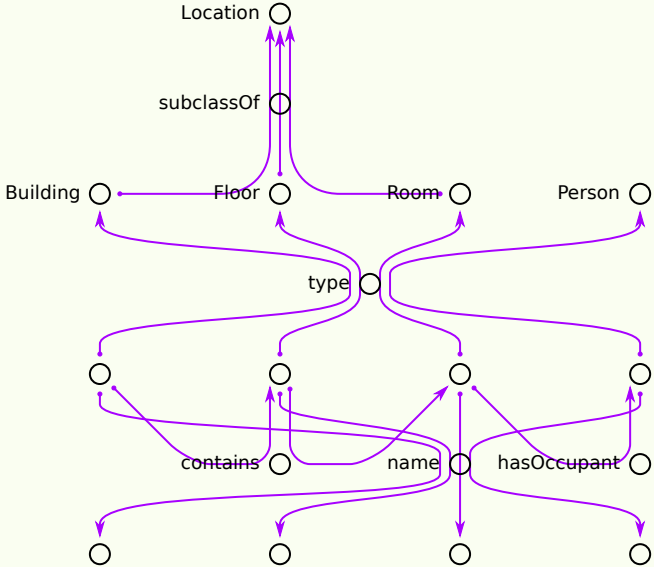
In this respect, ontologies are both a blessing and a burden. You use them to *define* your *representation* (e.g., relational, graph or object-oriented).

With ontologies, we have an extremely expressive basis for defining exactly the data models that we want, and schemas for them. We also have existing tooling for applying rules to an *offline* model.

With ontologies, we define data models using a machine readable language. That abstraction is (usually) at a far lower level than what we as humans are used to. That span represents a challenge.

- **When producing the representation model**, there are significant challenges in envisioning how it will be interpreted.
- **When producing the data adhering to the data model**, one has to consider how writing to the model affects model consistency.
- **When querying the model**, one has to think in terms of the lower abstraction unless the high-level concepts has been expanded.

Shared Understanding ▷ Example



RDF - The Resource Description Framework

An RDF ontology is defined through triples

- Triples: subject × predicate × object
- A model is a triple stores
- Namespaces
- Subclasses and subproperties

Subject	*Predicate*	*Object*
Marge	parent of	Bart
Marge	is a	Female
Bart	is a	Male
Female	subclass of	Human
Male	subclass of	Human

Subject of confusion:

- Sometimes the *schema* is referred to as the *ontology* and the data adhering to the schema as the *model*.
- However, the *ontology* is also a *model* in itself.
- At the end of the day it's all triples, and they all end up in one store.

Wherein lies the information?

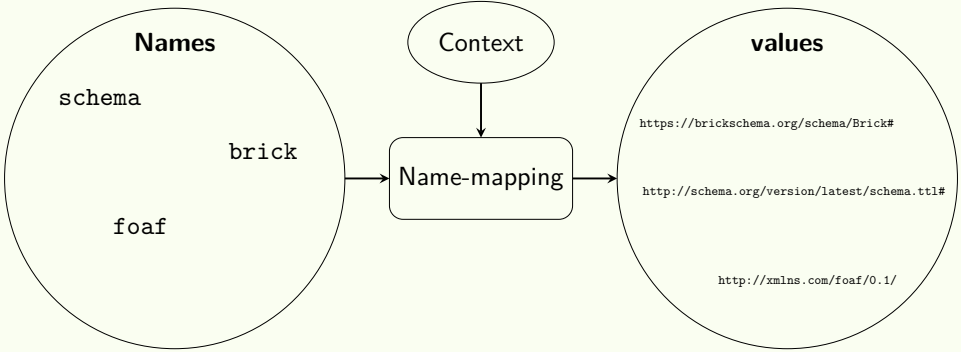
Hint: It is not in the name

In the relationships: A triple is a fact!

Resource Description Framework ▷ Namespaces

To avoid naming collisions, RDF uses *namespaces* for entities. In practice, these namespaces are simply prefixes and these are realized through *compact URIs* (abbreviated to CURI).

E.g., `foaf:Person` translates to `http://xmlns.com/foaf/0.1/Person`, and an `HTTP GET` request for this URI with an `Accept: application/rdf+xml` header should give you the definitions of that namespace.



Resource Description Framework ▷ Turtle: The Terse RDF Triple Language

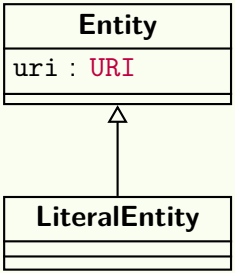
One (out of many) standardized ways of marshalling RDF.

File extension: `.ttl`

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix brick: <https://brickschema.org/schema/1.1.0/Brick#> .
@prefix demo:  <http://ontologies.sdu.dk/Demo#> .

demo:floor a brick:Floor .

demo:room a brick:Room ;
          rdfs:label "U173" ;
          brick:isLocatedIn demo:floor .
```

Resource Description Framework ▷ C# Representation

```
class Triple {
  Entity sub;  // subject
  Entity pred; // predicate
  Entity obj;  // object
}

IList<Triple> store;
```



In other words; while the *natural* representation is extremely simple, it is not very nice to work *directly* with.

That is why we have a query language!

SparQL

SparQL is by far the most popular query language for information models based on RDF and RDF-derived ontologies (like OWL).

The fundamental mechanics of SparQL operate at triple level, independently of how high abstractions have been constructed on top of RDF.

To some degree this can be bypassed by having a *reasoner* inject triples representing the relevant high-level abstractions.

SparQL ▷ SELECT Statements

```
SELECT DISTINCT ?room_name ?sensor_uuid ?setpoint_uuid ?actuator_uuid
WHERE {
    ?room     rdf:type/brick:subClassOf* brick:Room .
    ?sensor   rdf:type/brick:subClassOf* brick:Temperature_Sensor .
    ?setpoint rdf:type/brick:subClassOf* brick:Temperature_Setpoint .
    ?actuator rdf:type/brick:subClassOf* brick:Radiator_Valve_Position .

    ?sensor   brick:pointOf ?room .
    ?setpoint brick:pointOf ?room .
    ?actuator brick:pointOf ?room .

    ?room     brick:label ?room_name .
    ?sensor   brick:label ?sensor_uuid .
    ?setpoint brick:label ?setpoint_uuid .
    ?actuator brick:label ?actuator_uuid .
}
```

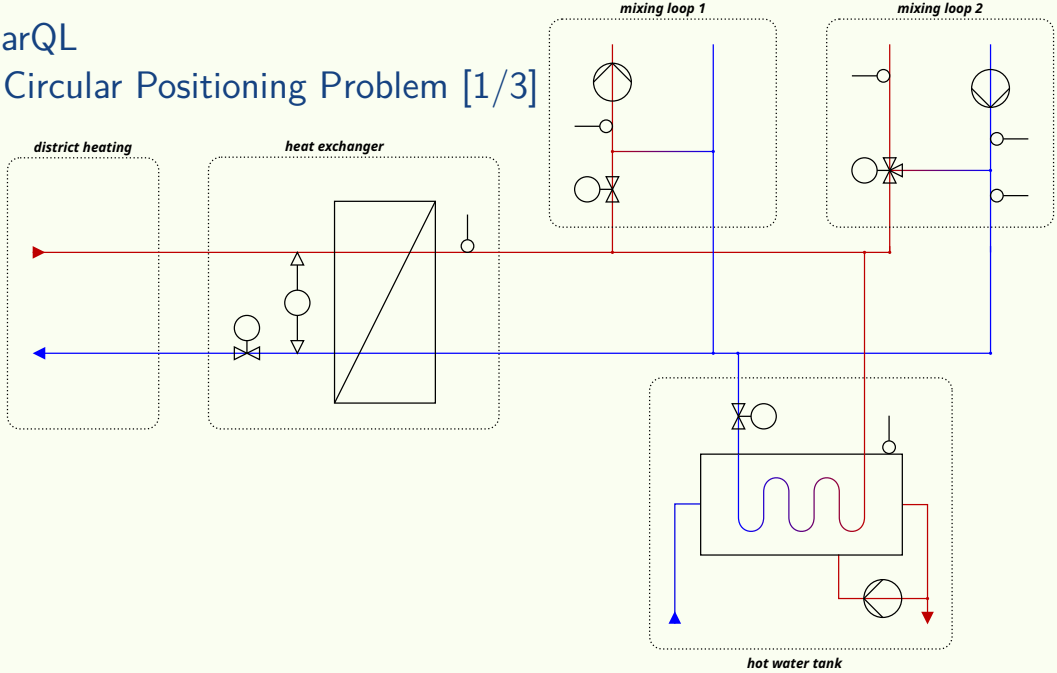
SparQL ▷ DELETE and INSERT Statements

```
DELETE {
    ?actuator brick:label "ec883cad-3235-46d5-a901-7bae169dda0a" .
}
WHERE {
    ?actuator rdf:type/brick:subClassOf* brick:Radiator_Valve_Position .
    ?actuator brick:label "ec883cad-3235-46d5-a901-7bae169dda0a" .
}
```

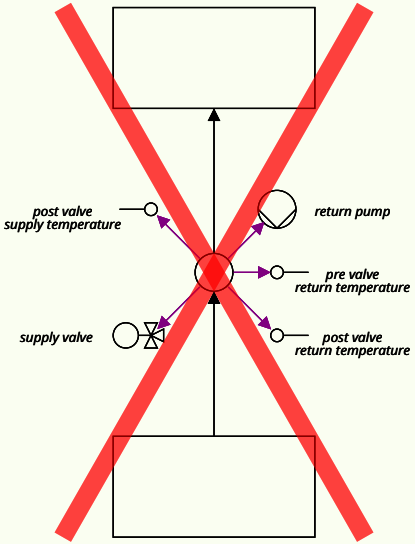
`INSERT` is similar, and it can be combined with `DELETE`.

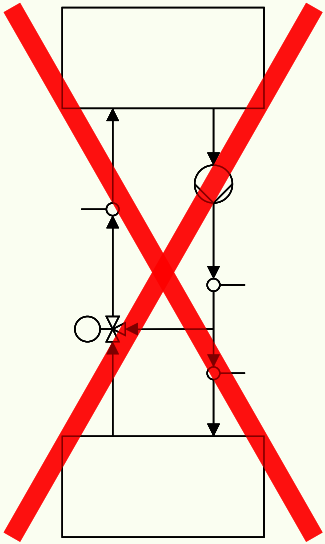
SparQL

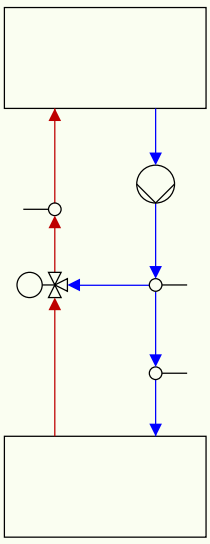
▷ Circular Positioning Problem [1/3]



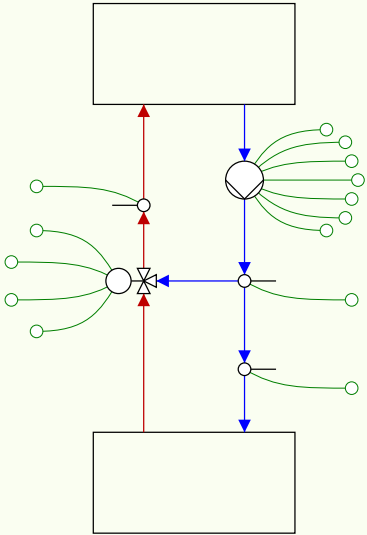
SparQL ▷ Circular Positioning Problem [2/3]







SparQL ▷ Circular Positioning Problem [3/3]



Web $\leftrightarrow$ Ontology Language (OWL)

A set of extensions to RDF.

Object-oriented'ish abstraction.

Adds formal semantics:

- ***Constraints*** e.g., on cardinality.
- ***Relationship Inferences*** transitive, symmetric, inverse ...
- ***Equivalence Testing*** whether two concepts are similar enough.
- ***Subsumption Testing*** whether one concept is more general than another.
- ...

Ontology Construction

The next couple of slides show examples of how one can construct the rules of an ontology using RDFS and OWL.

Note that this can be made quite a lot more complicated.

Ontology Construction ▷ Defining Types

We can define a type by declaring the name a subclass of something else.

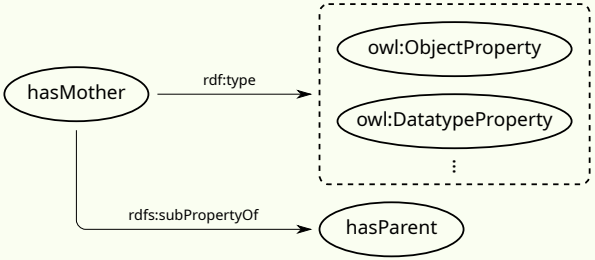
Everything that exists in the `owl:subClassOf` DAG rooted in `owl:Class` is a type.

Ontology Construction ▷ Defining Relationships

Relationships are defined through their use and their place in the DAG of types.

Their use is based on whether it points at a literal (`owl:DatatypeProperty`) or not (`owl:ObjectProperty`).

Their place is based on the `rdfs:subPropertyOf` relation.



Note: In this example `hasMother` and `hasParent` are properties.

Ontology Construction ▷ Relationship Restrictions

A relationship p can be restricted to only be allowed on some set S of subjects using `owl:domain` properties. Hereby you define the set of allowed triples of the form (S,p,*).

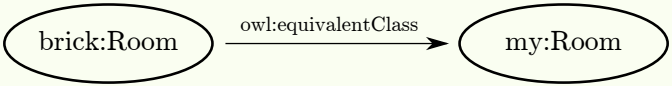
A relationship p can be restricted to only be allowed on some set O of objects using `owl:range` properties. Hereby you define the set of allowed triples of the form (*,p,O).

What happens when you combine them?

Ontology Construction ▷ Type Equivalence

We can state that two classes (e.g., types) are equivalent using `owl:equivalentType`.

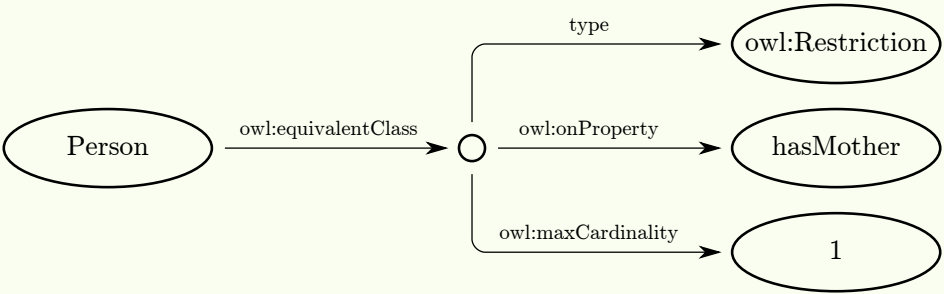
A common usecase is to bridge two ontologies in a model.



Ontology Construction ▷ Restrictions Between Type and Relationship

We can limit how many triples are allowed to follow the pattern (s, p, *).

This is known as a restriction on cardinality.



Note: In this example `hasMother` is a property.

Ontology Construction ▷ Literals

The XSD namespace defines a set of types whose values can be encoded as literals.

https://www.w3.org/2011/rdf-wg/wiki/XSD_Datatypes

Examples:

- int
- double
- boolean
- string
- dateTime

Query Execution

Query Execution ▷ Example

```
SELECT DISTINCT ?room_name ?sensor_uuid ?setpoint_uuid ?actuator_uuid
WHERE {
    ?room     rdf:type/brick:subClassOf* brick:Room .
    ?sensor   rdf:type/brick:subClassOf* brick:Temperature_Sensor .
    ?setpoint rdf:type/brick:subClassOf* brick:Temperature_Setpoint .
    ?actuator rdf:type/brick:subClassOf* brick:Radiator_Valve_Position .

    ?sensor   brick:pointOf ?room .
    ?setpoint brick:pointOf ?room .
    ?actuator brick:pointOf ?room .

    ?room     brick:label ?room_name .
    ?sensor   brick:label ?sensor_uuid .
    ?setpoint brick:label ?setpoint_uuid .
    ?actuator brick:label ?actuator_uuid .
}
```

1+n full table scans

single full table scan per combo of potential dependant variable bindings

Query Execution ▷ Complexity

There are several approaches to designing a SparQL query execution engine:

- ***Repeated full table scans*** This is what pretty much all the library implementations do. Evaluation time grows extremely fast.
- ***Add indexes*** Many dedicated RDF DBMS add indexes, but there seem to be too much expressiveness to effectively index. Full table scans will always be the fallback. Options include:
  - Path over single edge type $\times$ path length $\mapsto$ set of entities.
  - subject $\times$ predicate $\mapsto$ set of entities.
  - subject $\times$ object $\mapsto$ set of entities.
  - predicate $\times$ object $\mapsto$ set of entities.
- ***Drop full language support*** To allow for more efficient indexes.
- ***Drop RDF support*** and only support OWL. This removes the ability for any entity to take any role in a triple, and allows for modeling the data as a graph.

Schemas

The next couple of slides will try to illustrate the fundamental problem of being unable to model everything but wanting to validate everything.

Two approaches are presented, and we touch briefly on some of the concrete options.

Schemas ▷ Open vs Closed World

Closed World Assumption

- *"the presumption that any statement that is true is also known to be true"*
- In a closed world, you require all facts to be known.
- This is not always convenient, or possible.
- It is possible to validate that all properties hold.

Open World Assumption

- *"the assumption that the truth value of a statement may be true irrespective of whether or not it is known to be true"*
- In an open world, you accept that you don't know all facts.
- A property that needs to be satisfied could reside outside the set of known facts.
- It is thus impossible to verify that such a property holds.
- It is, however, possible to check that no restrictions are violated.

Schemas ▷ The Partially Open World [1/2]

Rule: A human has a human father and a human mother.

This has some fairly unpleasant consequences in a closed world, namely that we need to model an infinite number of humans.

- It doesn't fit reality: The number of humans that have ever lived is finite.
- We may not have storage or processing power for that.

Alternatively we can replace the rule with one stating that exactly one of the following rules must hold (and extend this through the evolutionary chain to single celled organisms where we can stop the chain at the grand spark of life):

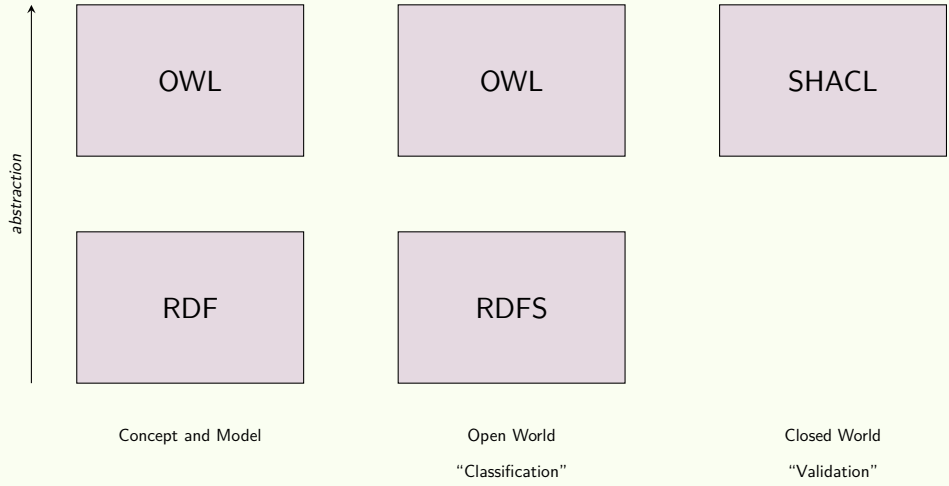
1. A human has a human father and a human mother.
2. A human has a human father and an ape mother.
3. A human has an ape father and a human mother.
4. A human has an ape father and an ape mother.

Schemas ▷ The Partially Open World [2/2]

This is overly complicated as we are unlikely to need that granularity and don't have the information necessary for populating the model readily available.

Instead, the rule is often defined in such a way that a human may have a human father and may have a human mother. Then the world assumption remains closed but we essentially allow for some openness by declaring the parents as nullable.

Schemas ▷ Overview of RDF and Friends



Validators

A validator tests the stated rules of the applied ontologies.

Examples:

- The brick:isLocatedIn relationship has as subject something that is a subclass of brick:Equipment and as object something that is a subclass of brick:Location.
- Any instance of a person:Human has exactly two parents.
- The birthdate property of an instance of a person:Human has the date type.

Without a validator triples can be added at will

- Spelling mistakes are accepted.
- Entities can be connected in any way.
  (in general, the order of definitions don't matter)
- Entities can be floating.

Reasoners

A reasoner can apply rules.

Examples:

- Given the facts that Marge is a Female and Marge is parent of Maggie a reasoner can conclude that Marge is the mother of Maggie.
- Given a fact stating that some entity is a temperature sensor and another stating that temperature sensor is a subclass of sensor a reasoner can conclude that this entity is a sensor.

Note: Each of these require a rule.

If you define such a rule, a reasoner will be able to apply it.

Reasoners ▷ Examples

Instead of:

```
?maggie person:hasName "Maggie" .
?mother person:isParentOf ?maggie .
?mother person:isA person:Female .
```

We can write:

```
?maggie person:hasName "Maggie" .
?mother person:isMotherOf ?maggie .
```

Instead of:

```
?sensor rdf:type/rdf:subClassOf* brick:Sensor .
```

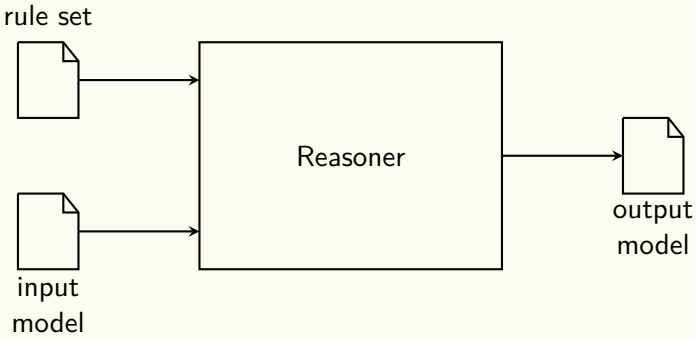
We can write:

```
?sensor rdf:type brick:Sensor .
```

Reasoners ▷ Application Time

Options for when to apply the reasoner:

1. When inserting
2. **On triple store before querying**
3. While querying



Further Reading

RDF and OWL:

- Google has plenty of information

Brick:

- Official homepage at `http://brickschema.org`

Evaluating SparQL Queries:

- ***Library***
  - ***Elixir*** the packages at `https://rdf-elixir.dev`
  - ***Python*** the `rdflib` module at `https://github.com/RDFLib/rdflib`
  - ***C#*** the `dotNetRdf` library at `https://dotnetrdf.org`
- ***Web Service***
  - Fuseki: `https://jena.apache.org/documentation/serving_data/`
  - HodDB: `https://hoddb.org`

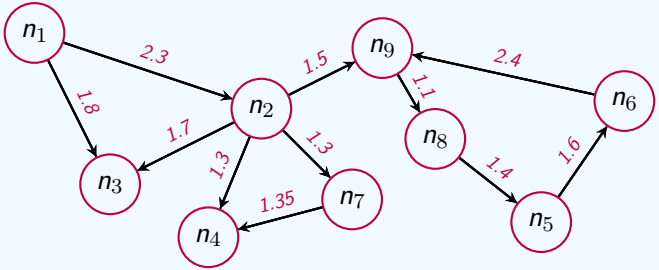
Part 2:
Property Graphs with Neo4J

Definitions

Definitions ▷ Graphs

Graphs form the basis for many applications, and are convenient for representing a wide gamut of structured data. Graphs consists of:

- ***Nodes*** | ***Vertices*** Usually illustrated as a circle or a box.
- ***Edges*** Usually illustrated as a line or arrow between two nodes.

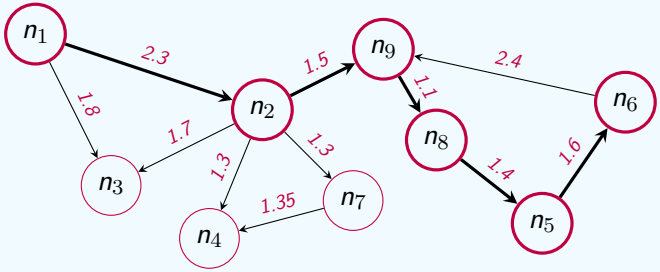


Edges can be directional and thus have a *source node* and a *destination node*. Such edges are drawn as arrows from source nodes to destination node, and the existence of such edges make the graph *directional*.

Edges may be associated with a property or weight. If all edges are associated with a weight then the graph is said to be *weighted*.

Definitions ▷ Paths

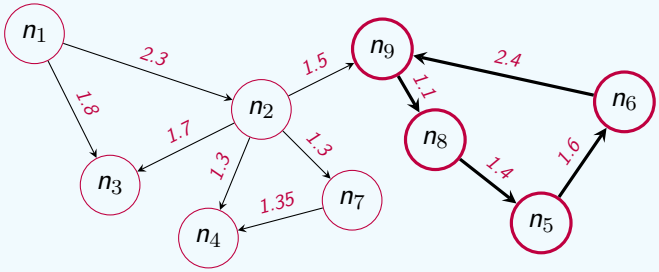
A *path* is a sequence of connected edges that connects one node to another so that one can transition from the path source to the path destination by following a sequence of such edges.



This can be used to calculate the distance between n_1 and n_6 as $2.3 + 1.5 + 1.1 + 1.4 + 1.6 = 7.9$.

Definitions ▷ Cycles

A *cycle* is a path whose source and destination node is the same.



Graphs with cycles can be difficult to work with, so this is a property that can be attractive to avoid.

A very common class of graphs is known as *direct acyclic graphs* or *DAGs* for short.

Definitions ▷ Property Graphs

Property graphs are graphs with properties on nodes and edges.

Properties are modeled as key-value stores.

Most general-purpose graphs systems (libraries/frameworks/DBMS) are property graph systems.

We will be using the *Neo4j* data model as examples.

Concrete implementation may differ, but rarely by a significant amount.

Definitions ▷ Graph Query Languages

A *graph query language* is a language defined to express a *graph pattern* that a *query execution engine* can match against a graph. The result of this is a set of *match sites* where each match site is a mapping from the graph pattern to a concrete subgraph of the queried graph.

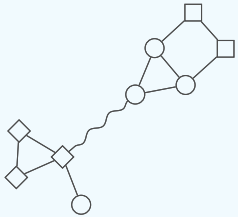
I consider a *real* graph query language to be one that features graph-idiomatic concepts, including (but not limited to):

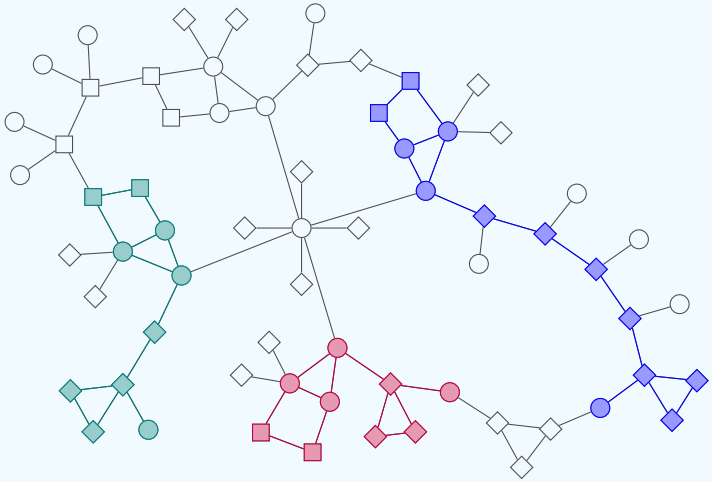
- The level of abstraction refers to *nodes*, *edges*, *properties* and *labels*.
- Paths can be matched.
- A suite of graph algorithms can be referenced.

Good examples are *(Open)Cypher* and *Gremlin*.

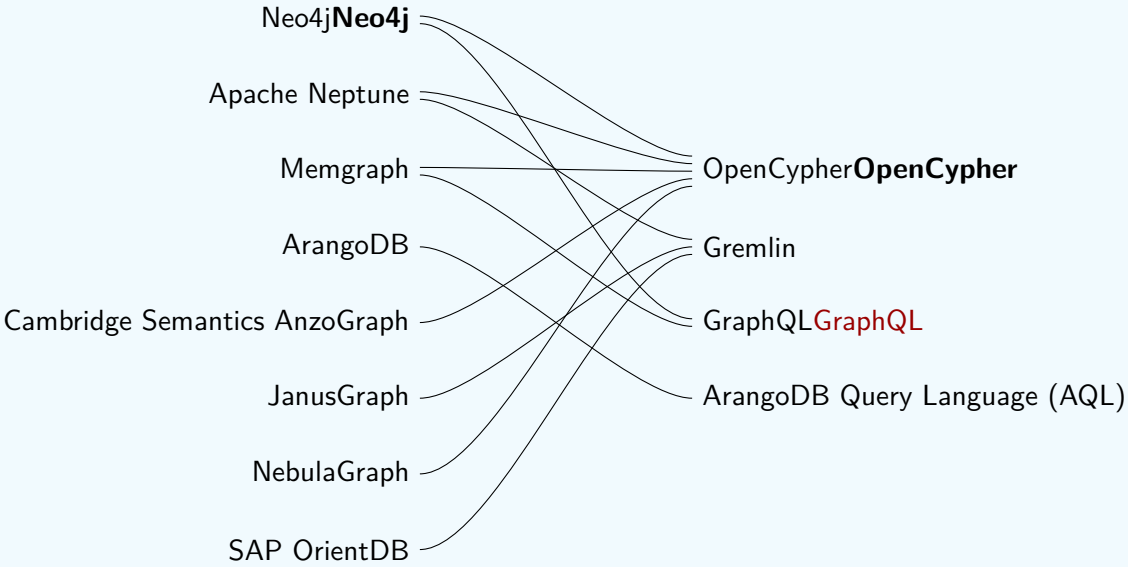
Note: This excludes *GraphQL*.

Definitions ▷ Graph Queries





Property Graph DBMS Implementations



Neo4J Data Model ▷ Nodes

A node consists of:

- A (possibly empty) set of *labels*.
  (these are essentially types)
- A key-value store containing named *properties*.
- A map from relationship type to set of *outgoing relationships*.
- A map from relationship type to set of *incoming relationships*.

```
class Node {
  ISet<string>                              labels;
  IDictionary<string, Object>               properties;
  IDictionary<string, ISet<Relationship>> outgoing;
  IDictionary<string, ISet<Relationship>> incoming;
}
```

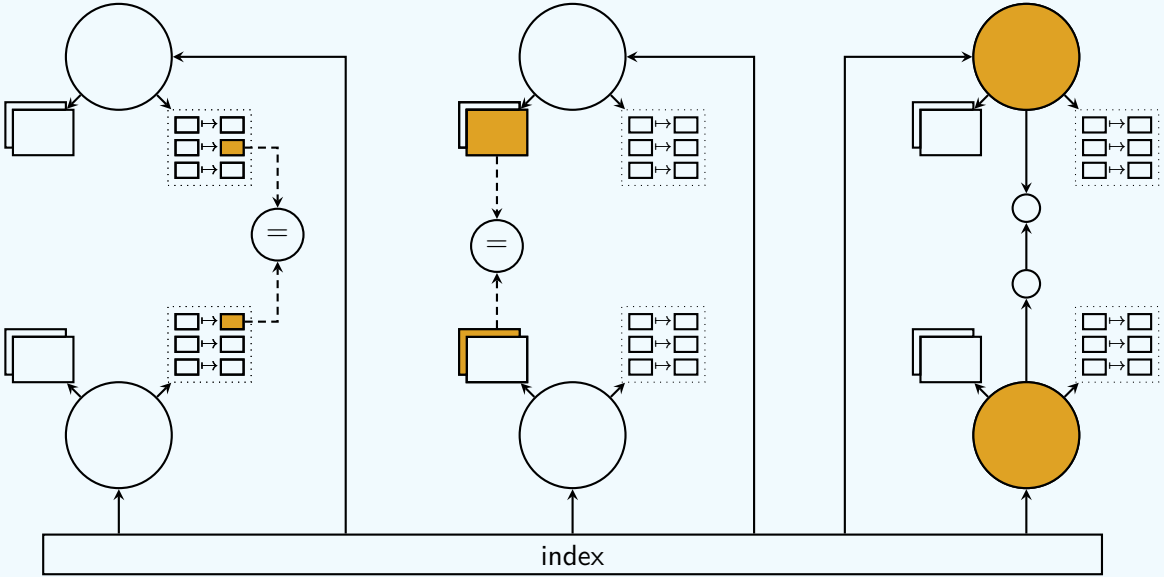
Neo4J Data Model ▷ Edges (aka Relationships)

A relationship consists of:

- A single *type*.
- A key-value store containing named *properties*.
- A *source node*.
- A *destination node*.

```
class Relationship {
  string                          type;
  IDictionary<string, Object> properties;
  Node                            src;
  Node                            dst;
}
```

Neo4J Data Model ▷ Matching Based on Property, Label and Edge



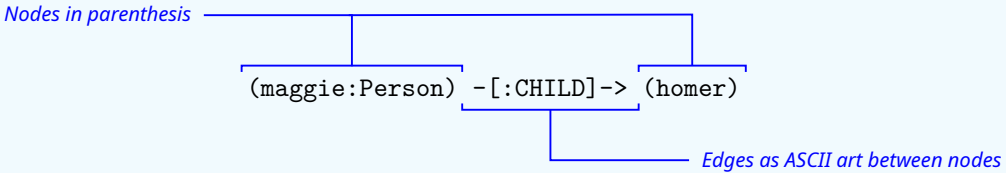
Querying with (Open)Cypher

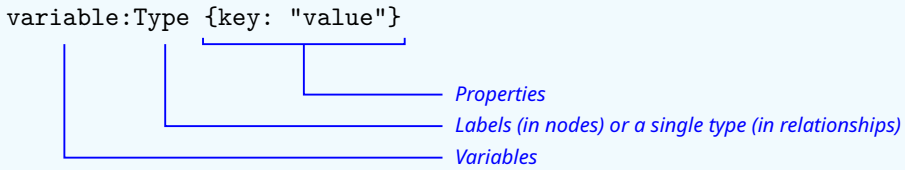
Neo4j pioneered property graph DBMS.

Initially they used the proprietary *Cypher* query language, but later opened it up as *OpenCypher*.

However, nowadays people refer to OpenCypher simply as Cyper.

Querying with (Open)Cypher ▷ Anatomy of a Match Pattern





Querying with (Open)Cypher ▷ Match Patterns

List all farther-daugther pairs:

```
MATCH
  (father:Person {sex: "male"}),
  (daughter:Person {sex: "female"}),
  (father) -[:parentOf]-> (daughter)
RETURN father.name, daughter.name
```

List all parent-child pairs of same sex:

```
MATCH
  (parent:Person),
  (child:Person),
  (parent) -[:parentOf]-> (child)
WHERE parent.sex = child.sex
RETURN parent.name, child.name
```

Querying with (Open)Cypher ▷ Query Execution Plan

How is a query evaluated?

Indexes are used to look up starting points, from which the query execution engine traverses the graph.

Tools:

- Use the `EXPLAIN` keyword to detail the produced query execution plan.
- Use the `PROFILE` keyword to get a timing profile of the query resolution.

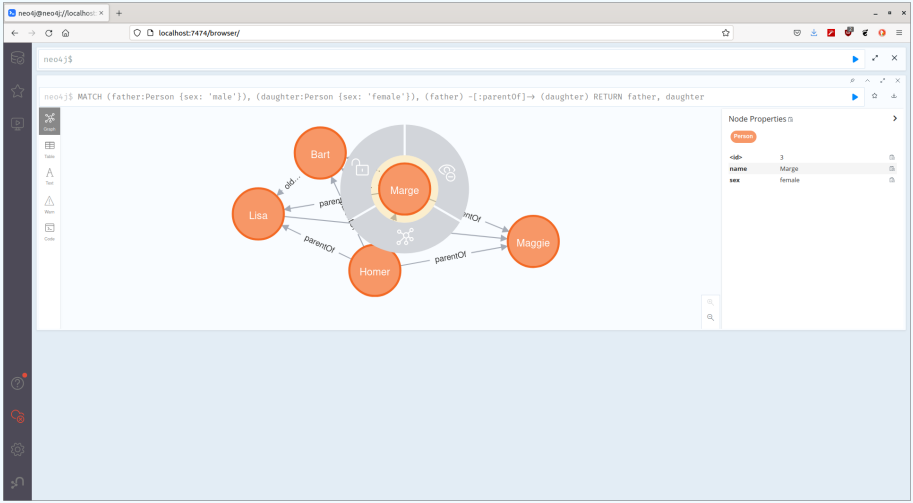
(just like you know from relational databases)

Querying with (Open)Cypher ▷ Extendability

- Libraries exist
  - Neosemantics (Neo4j RDF & Semantics toolkit)
    - Can validate your graph against constraints expressed in *SHACL*.
    - `https://neo4j.com/labs/neosemantics/`
  - APOC – Awesome Procedures On Cypher – library
    - `https://neo4j.com/docs/apoc/current/`
  - Spatial (e.g., geographical distance)
    - `https://neo4j.com/docs/cypher-manual/current/functions/spatial/`
  - GraphQL support
    - `https://neo4j.com/developer/graphql/`
- Custom functions
  - Cypher Manual:
    `https://neo4j.com/docs/cypher-manual/current/functions/user-defined/`
  - Neo4j Java Reference:
    `https://neo4j.com/docs/java-reference/current/extending-neo4j/functions/`
  - Neo4j Developer Documentation:
    `https://neo4j.com/developer/cypher/procedures-functions/`

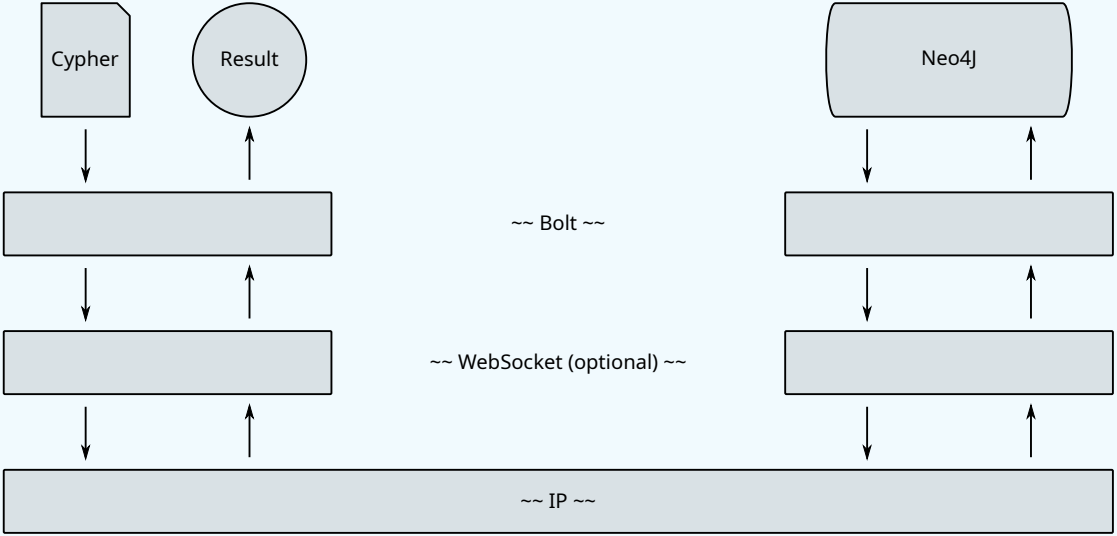
Neo4J Interaction Model

Neo4J Interaction Model ▷ Human Client



`http://localhost:7474` *or* `https://localhost:7473`

Neo4J Interaction Model ▷ Machine Client



Part -1:
Epilogue

Comparison ▷ Data

Property	RDF	Neo4J
What:	Standard	Implementation
Choice:	Multiple implementations	free and enterprise
Underlying model:	triple store	property graph
Abstraction level:	fact	property graph
Schema support:	strong	weak
Schema availability:	select implementations	enterprise only[†]
Reasoner support:	strong	theoretical
Reasoner availability:	select implementations	
Adjacency:	scanned[‡]	index-free
Schema publication:	×	

[†] A single sample constraint is available in the free version.

[‡] Index-free adjacency can be provided for high-abstraction relationships.

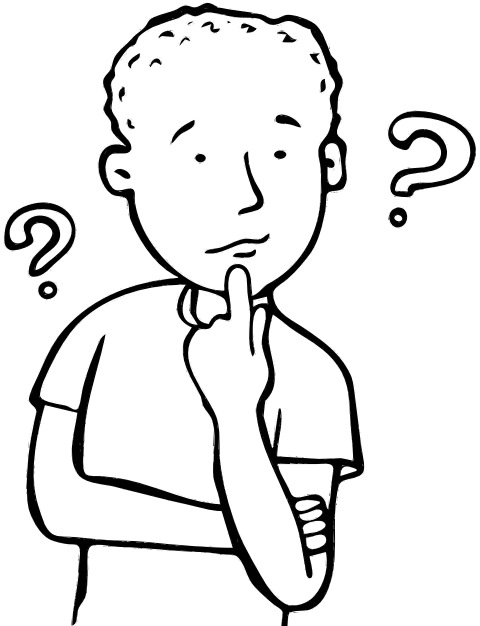
Comparison ▷ Query Language

Property	SparQL	Cypher
Model support:	RDF	Neo4J^{+}
Match unit:	triple	node/edge
Pattern unit term:	restriction	match pattern
Path directionaliy:	unidirectional	bidirectional
Restrictions of path:		×
Cross-schema queries:	×	
Evaluation speed:	varying but consistent	fast

Note 1: There are other graph query languages. *Gremlin* stands out.

Note 2: Contrary to what the name suggests, GraphQL is not a graph query language.

Questions?



https://openclipart.org/detail/238087/boy-thinking-of-question