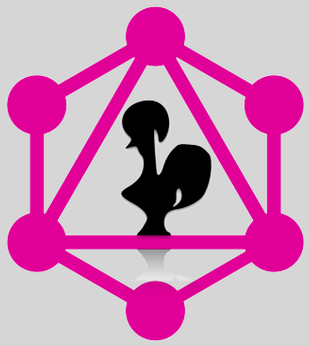


# GraphCoQL

## A mechanized formalization of GraphQL in Coq

**Tomás Díaz** Federico Olmedo Éric Tanter

Millennium Institute
Foundational Research on Data

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

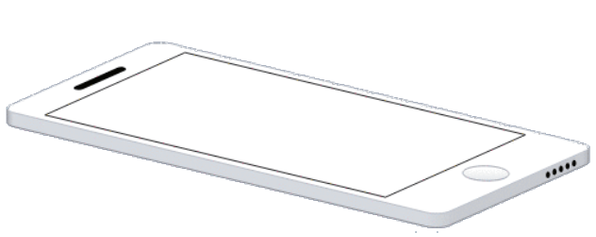
**Certified Programs and Proofs**
**New Orleans, USA – January 2020**

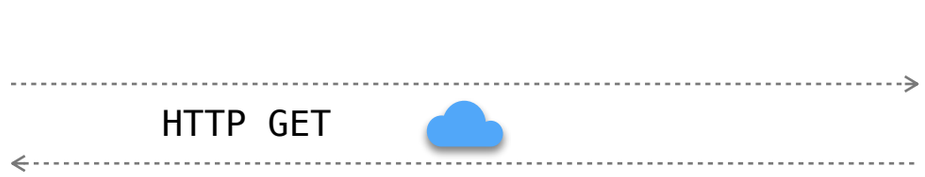
# GraphQL

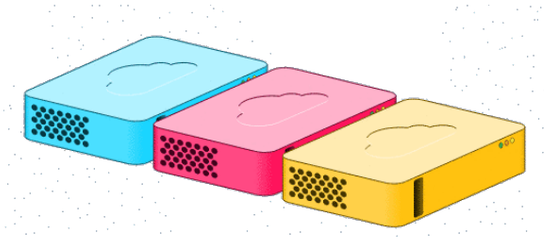
Language for specifying the interfaces of web data services and their query mechanism

# GraphQL

Language for specifying the interfaces of web data services and their query mechanism



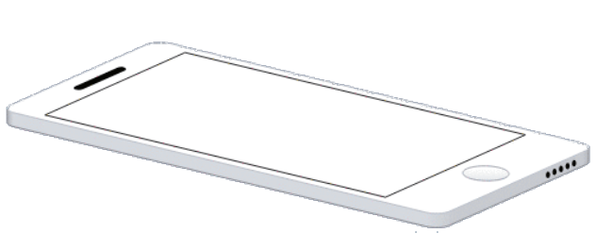


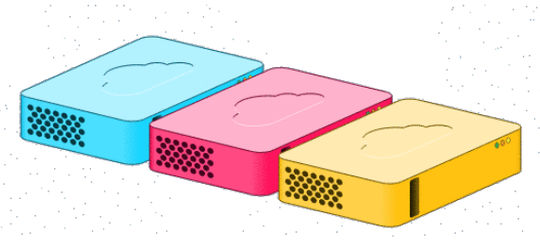
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {



}
```



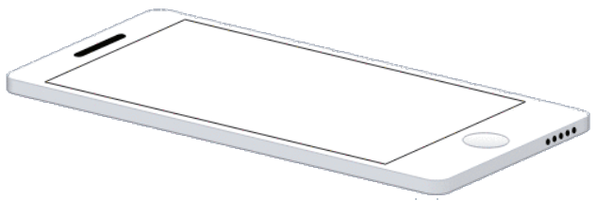
HTTP GET



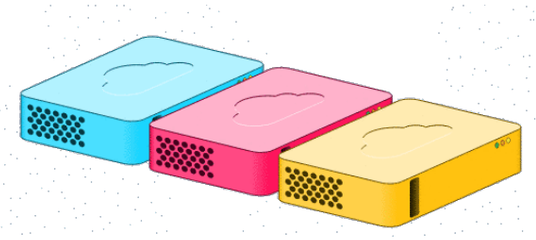
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        |



    }
}
```



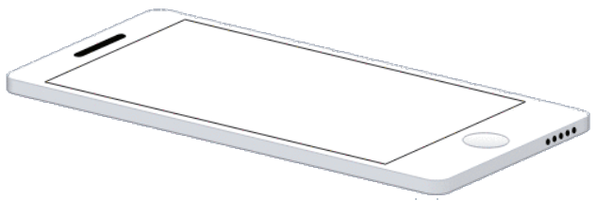
HTTP GET



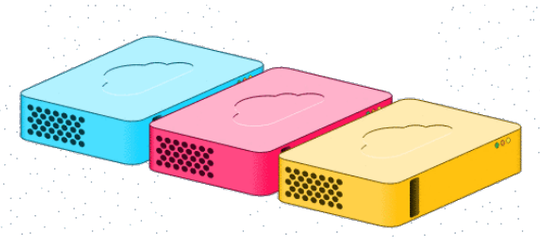
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        name



    }
}
```



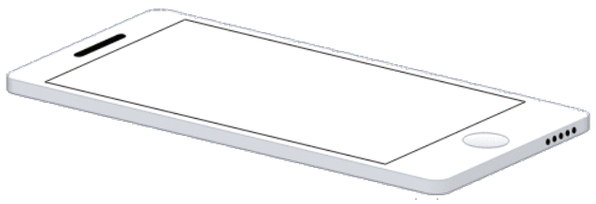
HTTP GET



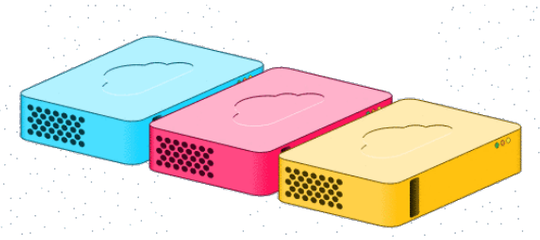
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
          title
        }
    }
}
```



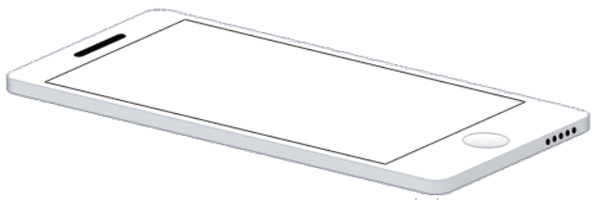
HTTP GET



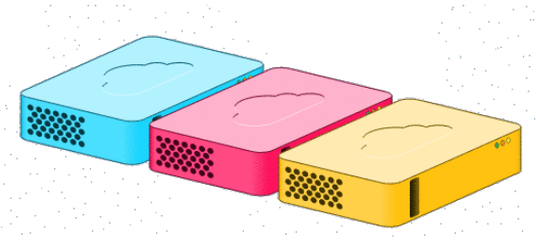
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
          title
        }
    }
}
```



HTTP GET

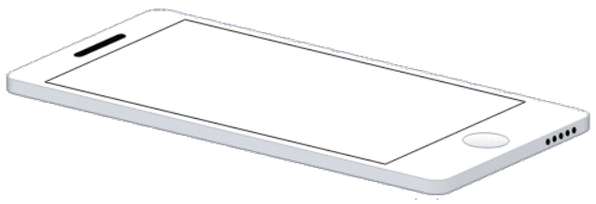


```
{

}
```

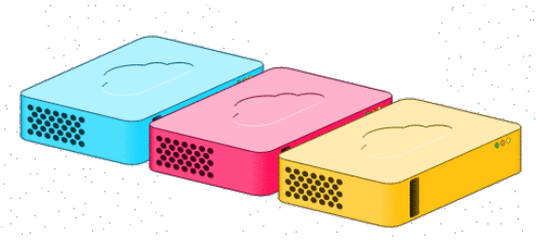
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
          title
        }
    }
}
```



HTTP GET

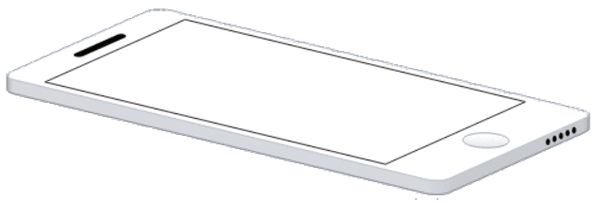


```
{
  “artist” : {
    “name” : “Tom Hanks”,

  }
}
```

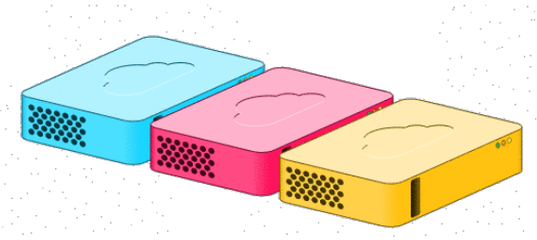
# GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
          title
        }
    }
}
```



HTTP GET



```
{
  “artist” : {
    “name” : “Tom Hanks”,
    “artworks” : [
      {
        “title” : “Toy Story”,
      },
      {
        “title” : “Forrest Gump”,
      },
      …
    ]
  }
}
```

# Industry involvement with GraphQL

2012

FACEBOOK

2015+

# First language formalization [Hartig & Pérez, WWW'18]

Paper & pencil formalization to study complexity properties.

$$[\![\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \mathtt{f}\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \mathtt{f}\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\ell\!:\!\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \ell\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \ell\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \mathtt{f}\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathtt{L}_\mathtt{T} \text{ and } \{v_1,\ldots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \mathtt{f}\!:\!\{\,[\![\varphi]\!]^{v}_G\,\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathtt{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \mathtt{f}\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathtt{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

$$[\![\ell\!:\!\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \ell\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathtt{L}_\mathtt{T} \text{ and } \{v_1,\ldots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \ell\!:\!\{\,[\![\varphi]\!]^{v}_G\,\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathtt{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \ell\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathtt{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

$$[\![\mathtt{on}\ \mathtt{t}\{\varphi\}]\!]^u_G = \begin{cases} [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathtt{O}_\mathtt{T} \text{ and } \tau(u) = \mathtt{t}, \text{ or } \mathtt{t} \in \mathtt{I}_\mathtt{T} \text{ and } \tau(u) \in \mathit{implementation}_{\mathcal{S}}(\mathtt{t}), \text{ or} \\ & \mathtt{t} \in \mathtt{U}_\mathtt{T} \text{ and } \tau(u) \in \mathit{union}_{\mathcal{S}}(\mathtt{t}) \\ \varepsilon & \text{in other case.} \end{cases}$$

$$[\![\varphi_1 \cdots \varphi_k]\!]^u_G = \mathrm{collect}([\![\varphi_1]\!]^u_G \cdots [\![\varphi_k]\!]^u_G)$$

**Figure 5: Semantics of a GraphQL query.**

# First language formalization [Hartig & Pérez, WWW'18]

Paper & pencil formalization to study complexity properties.

**Missing proofs about fundamental properties**

$$[\![\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \mathtt{f}\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \mathtt{f}\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\ell\!:\!\mathtt{f}[\alpha]]\!]^u_G = \begin{cases} \ell\!:\!\lambda(u,\mathtt{f}[\alpha]) & \text{if } (u,\mathtt{f}[\alpha]) \in \mathrm{dom}(\lambda) \\ \ell\!:\!\mathtt{null} & \text{else.} \end{cases}$$

$$[\![\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \mathtt{f}\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathtt{L}_\mathsf{T} \text{ and } \{v_1,\ldots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \mathtt{f}\!:\!\{[\![\varphi]\!]^{v}_G\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathsf{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \mathtt{f}\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathsf{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

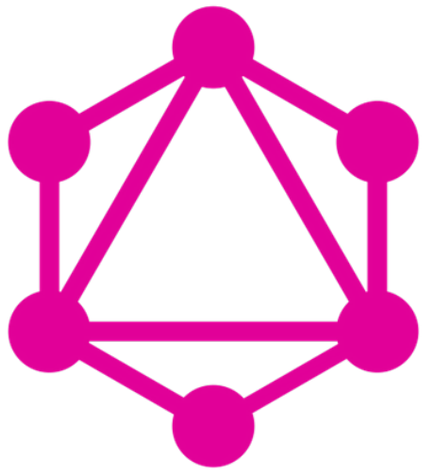
$$[\![\ell\!:\!\mathtt{f}[\alpha]\{\varphi\}]\!]^u_G = \begin{cases} \ell\!:\![\{[\![\varphi]\!]^{v_1}_G\}\cdots\{[\![\varphi]\!]^{v_k}_G\}] & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \in \mathtt{L}_\mathsf{T} \text{ and } \{v_1,\ldots,v_k\} = \{v_i \mid (u,\mathtt{f}[\alpha],v_i) \in E\} \\ \ell\!:\!\{[\![\varphi]\!]^{v}_G\} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathsf{T} \text{ and } (u,\mathtt{f}[\alpha],v) \in E \\ \ell\!:\!\mathtt{null} & \text{if } \mathit{type}_{\mathcal{S}}(\mathtt{f}) \notin \mathtt{L}_\mathsf{T} \text{ and there is no } v \in N \text{ s.t. } (u,\mathtt{f}[\alpha],v) \in E \end{cases}$$

$$[\![\mathtt{on\ t}\{\varphi\}]\!]^u_G = \begin{cases} [\![\varphi]\!]^u_G & \text{if } \mathtt{t} \in \mathtt{O}_\mathsf{T} \text{ and } \tau(u) = \mathtt{t}, \text{ or } \mathtt{t} \in \mathtt{I}_\mathsf{T} \text{ and } \tau(u) \in \mathit{implementation}_{\mathcal{S}}(\mathtt{t}), \text{ or} \\ & \mathtt{t} \in \mathtt{U}_\mathsf{T} \text{ and } \tau(u) \in \mathit{union}_{\mathcal{S}}(\mathtt{t}) \\ \varepsilon & \text{in other case.} \end{cases}$$

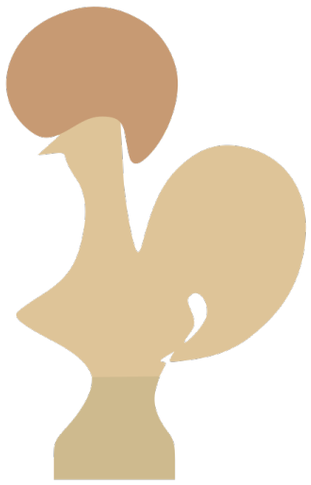
$$[\![\varphi_1 \cdots \varphi_k]\!]^u_G = \mathrm{collect}([\![\varphi_1]\!]^u_G \cdots [\![\varphi_k]\!]^u_G)$$

**Figure 5: Semantics of a GraphQL query.**

# Our contribution



**GraphCoQL**



First **mechanized** formalization of
GraphQL in the Coq proof assistant

# Schema

Describes how data is structured and queried

# Schema

Describes how data is structured and queried

object type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}
```

# Schema

Describes how data is structured and queried

object type

interface type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}

interface Movie {
  id: ID
  title: String
  year: Int
  cast: [Artist]
}
```

# Schema

Describes how data is structured and queried

object type

interface type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}

interface Movie {
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Fiction implements Movie {
  …
}

type Animation implements Movie {
  …
  style: Style
}
```

# Schema

Describes how data is structured and queried

object type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}

interface Movie {
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Fiction implements Movie {
  …
}

type Animation implements Movie {
  …
  style: Style
}
```

interface type

```
enum Role {
      ACTOR
      DIRECTOR
      WRITER
}
```

enumeration type

# Schema

Describes how data is structured and queried

object type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}
```

interface type

```
interface Movie {
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Fiction implements Movie {
  …
}

type Animation implements Movie {
  …
  style: Style
}
```

```
enum Role {
      ACTOR
      DIRECTOR
      WRITER
}
```

enumeration type

```
union Artwork = Fiction
              | Animation
              | Book
```

union type

```
type Book { … }
```

# Schema

Describes how data is structured and queried

object type

```
type Artist {
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}
```

interface type

```
interface Movie {
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Fiction implements Movie {
  …
}

type Animation implements Movie {
  …
  style: Style
}
```

```
enum Role {
      ACTOR
      DIRECTOR
      WRITER
}
```

enumeration type

```
union Artwork = Fiction
              | Animation
              | Book
```

union type

```
type Book { … }
```

```
type Query {
  artist(id:ID): Artist
  movie(id:ID): Movie
}
```

entry points for querying the dataset

# Schema

Describes how data is structured and queried

*TypeDefinition* :
*ScalarTypeDefinition*
*ObjectTypeDefinition*
*InterfaceTypeDefinition*
*UnionTypeDefinition*
*EnumTypeDefinition*
*InputObjectTypeDefinition*

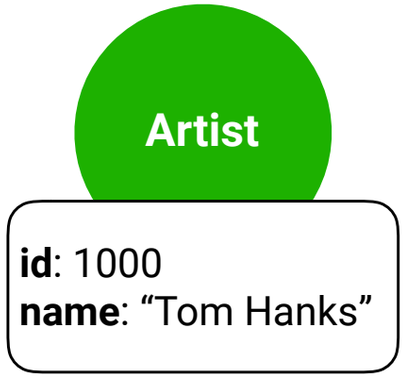
```
Inductive TypeDefinition : Type :=
| ScalarTypeDefinition (name : Name)
| ObjectTypeDefinition (name : Name)
                       (interfaces : seq Name)
                       (fields : seq FieldDefinition)
| InterfaceTypeDefinition (name : Name)
                          (fields : seq FieldDefinition)
| UnionTypeDefinition (name : Name)
                      (members : seq Name)
| EnumTypeDefinition (name : Name)
                     (members : seq EnumValue).
```

# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.

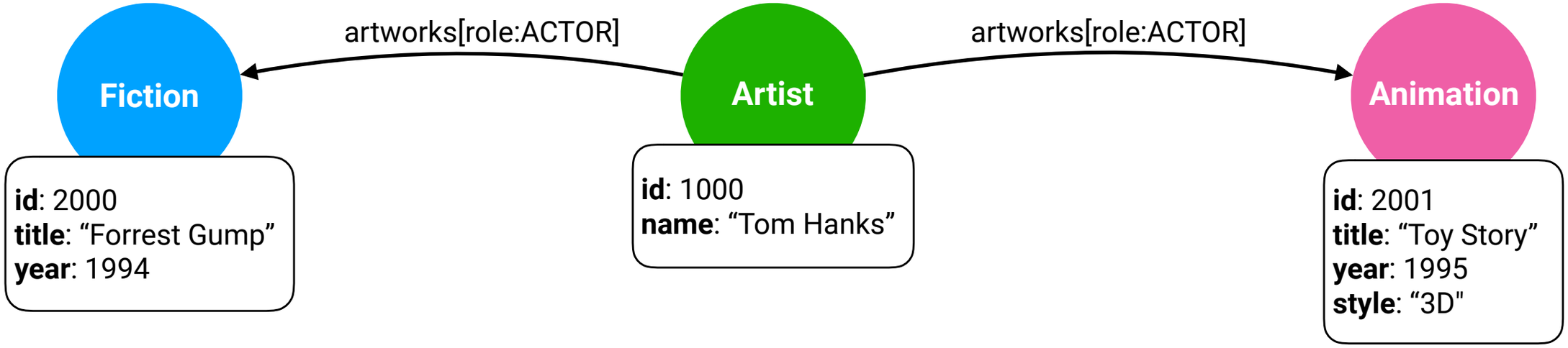
# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.



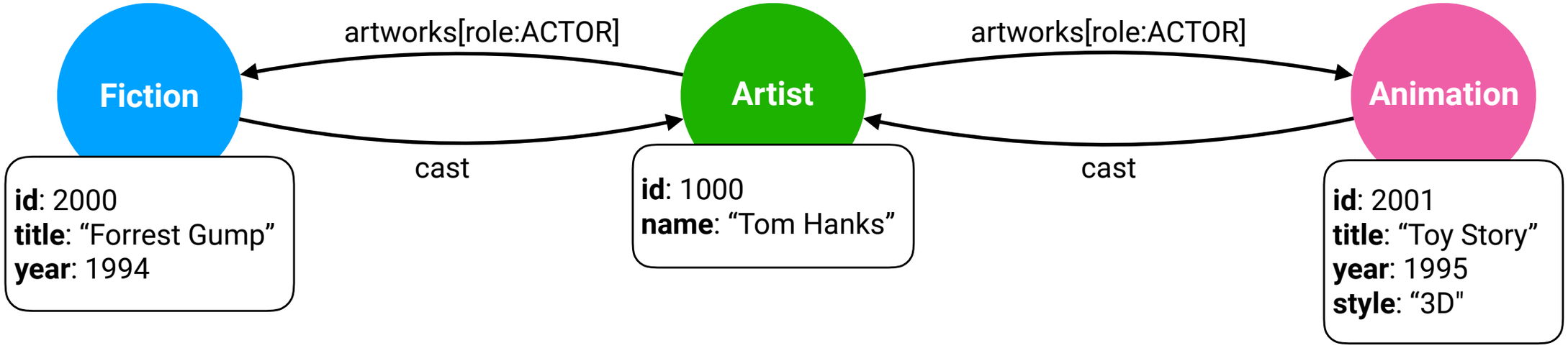
# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.



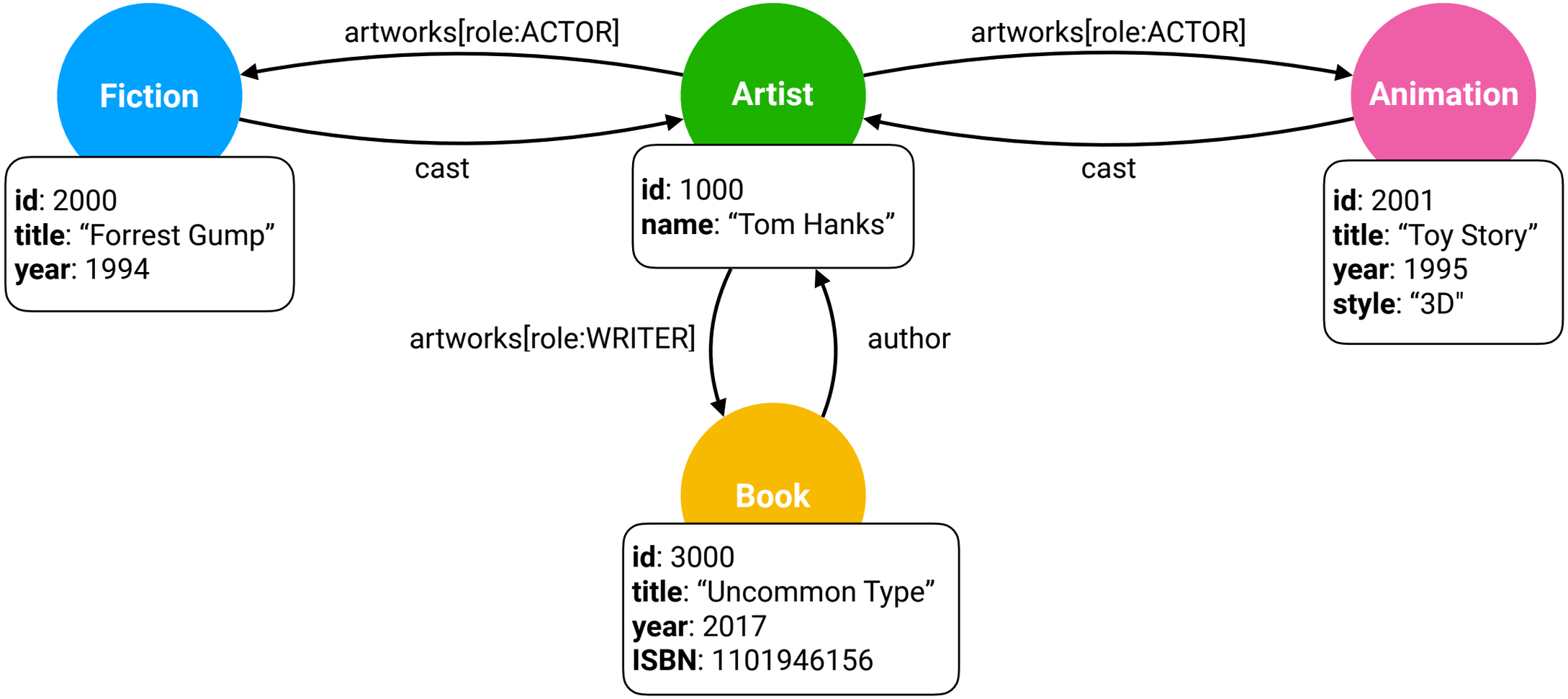
# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.



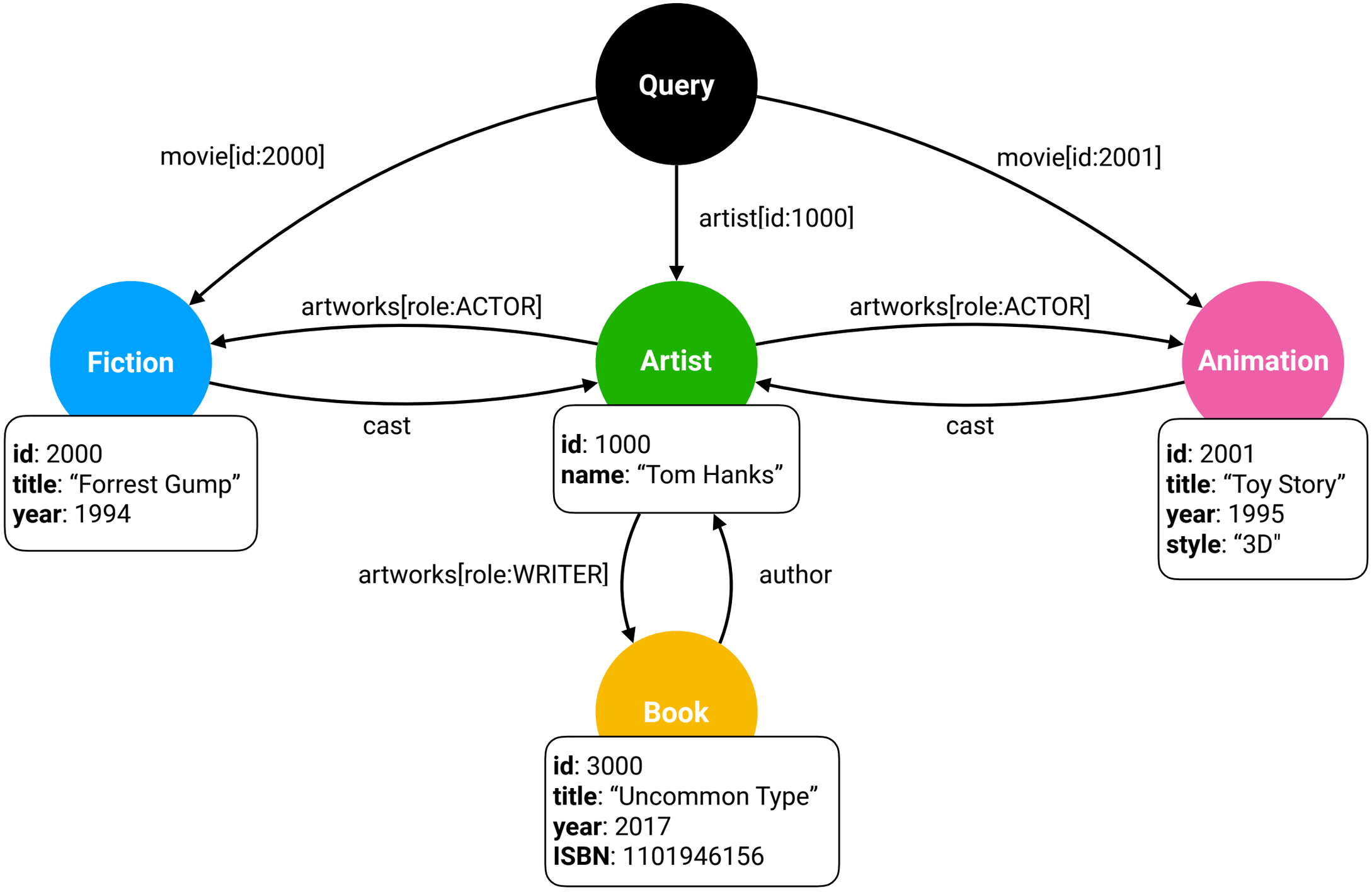
# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.



# Graph data model

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.



# Query evaluation

Queries are evaluated by traversing the graph and collecting nodes' properties

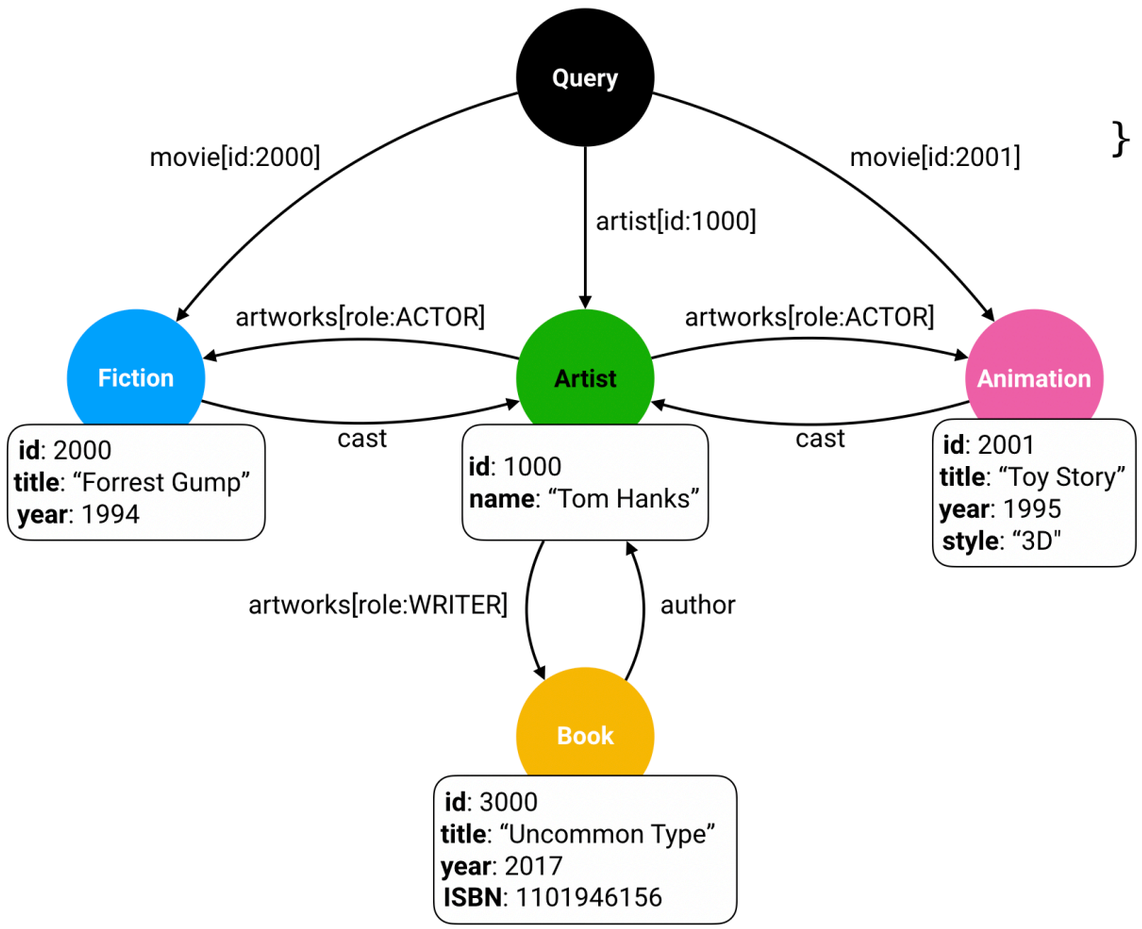
**Query**

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
            title
        }
    }
}
```

**Response (à la JSON)**

```
{
```

**Dataset**



# Query evaluation

Queries are evaluated by traversing the graph and collecting nodes' properties

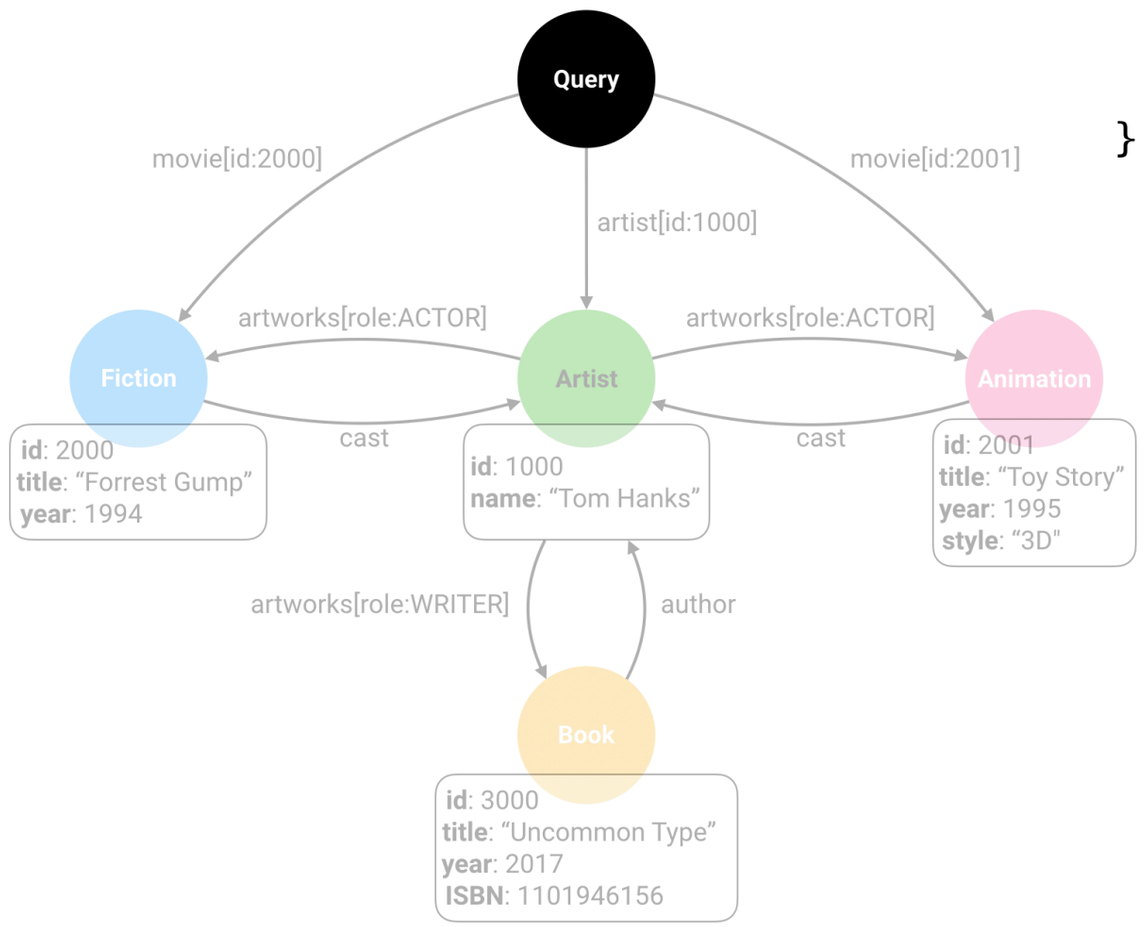
**Query**

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
            title
        }
    }
}
```

**Response (à la JSON)**

```
{
}
```

**Dataset**



# Query evaluation

Queries are evaluated by traversing the graph and collecting nodes' properties

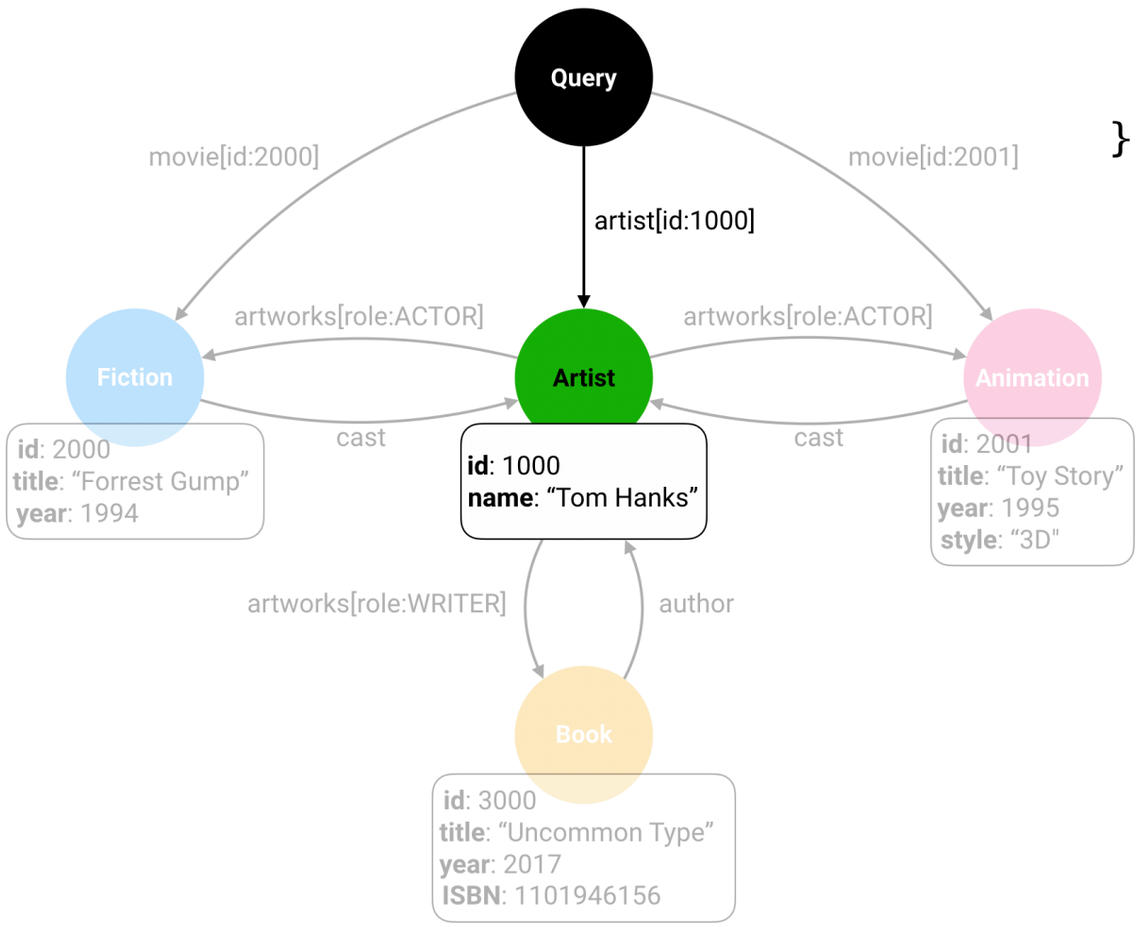
**Query**

```
query {
  artist(id:1000) {
      name
      artworks(role: ACTOR) {
          title
      }
  }
}
```

**Response (à la JSON)**

```
{
  "artist" : {
                                    }
                                  }
```

**Dataset**



# Query evaluation

Queries are evaluated by traversing the graph and collecting nodes' properties

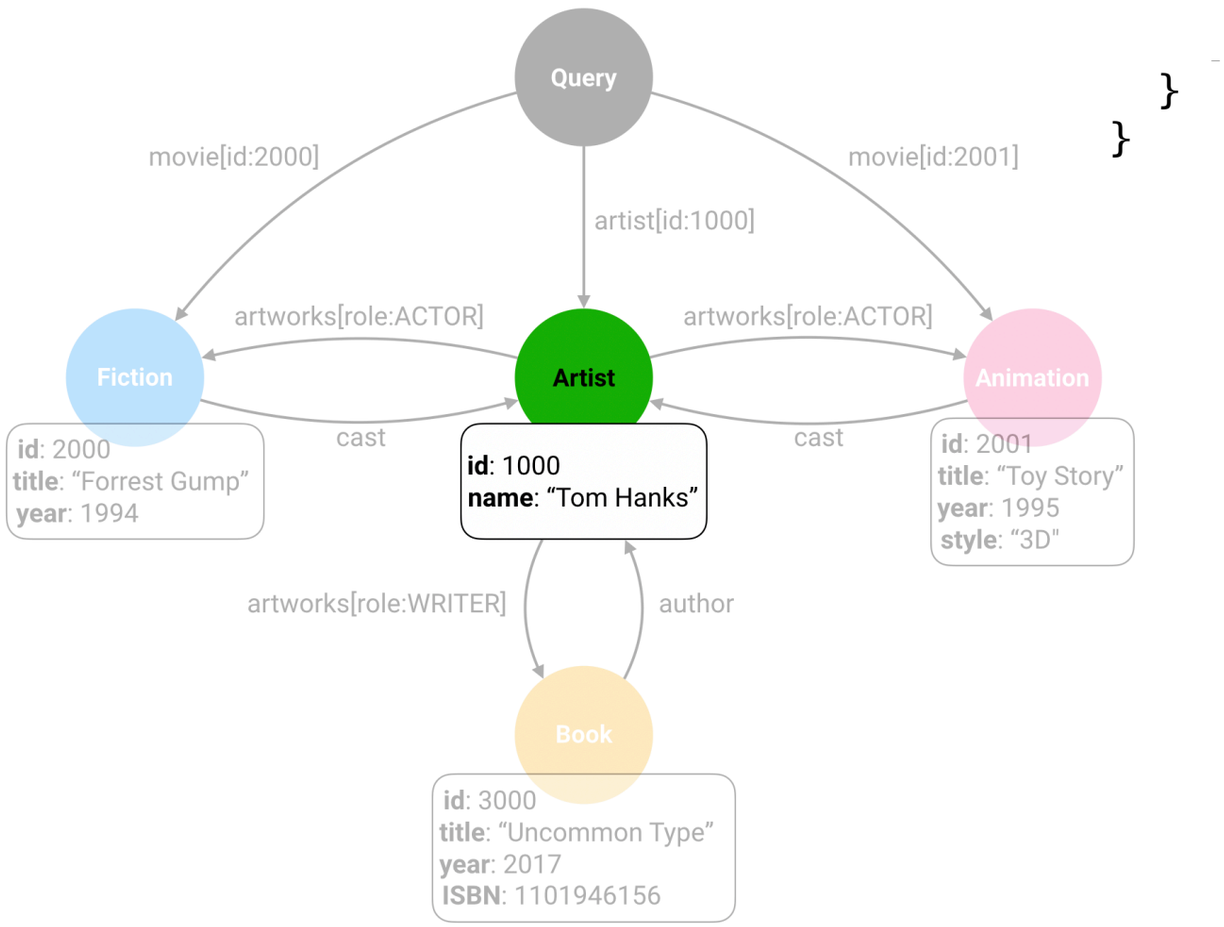
**Query**

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
            title
        }
    }
}
```

**Response (à la JSON)**

```
{
  "artist" : {
    "name" : "Tom Hanks",
```

**Dataset**




```
  }
}
```

# Query evaluation

Queries are evaluated by traversing the graph and collecting nodes' properties

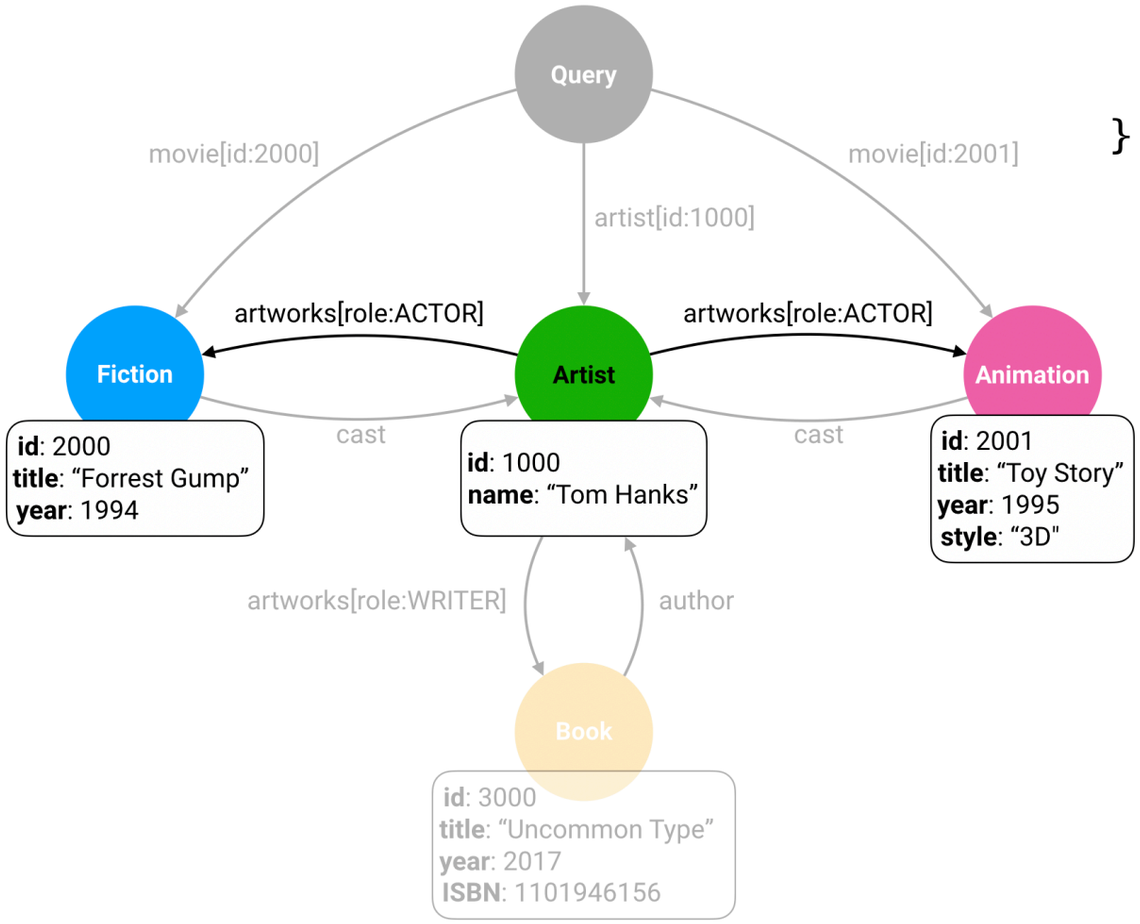
**Query**

```
query {
    artist(id:1000) {
        name
        artworks(role: ACTOR) {
            title
        }
    }
}
```

**Response (à la JSON)**

```
{
  "artist" : {
    "name" : "Tom Hanks",
    "artworks" : [
      {
        "title" : …
      },
      {
        "title" : …
      },
    ]
  }
}
```

**Dataset**



# Query evaluation - Peculiarities

Query evaluation is not compositional

# Query evaluation - Peculiarities

Query evaluation is not compositional

```
query {
    artist(id:1000) {
        name
    }
    artist(id:1000) {
        artworks(role: ACTOR) {
            title
        }
    }
}
```

# Query evaluation - Peculiarities

Query evaluation is not compositional

```
query {
    artist(id:1000) {
        name
    }
    artist(id:1000) {
        artworks(role: ACTOR) {
            title
        }
    }
}
```

# Query evaluation - Peculiarities

Query evaluation is not compositional

```
query {
    artist(id:1000) {
        name
    }
    artist(id:1000) {
        artworks(role: ACTOR) {
            title
        }
    }
}
```

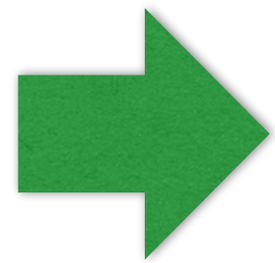
```
{
  “artist” : {
    “name” : “Tom Hanks”,
  },

  “artist” : {
    “artworks” : [
      {
        …
      },
      {
        …
      },
    ]
  }
}
```

# Query evaluation - Singularities

Query evaluation is not compositional

Selections are “factored-out” in between the recursive calls

```
query {
    artist(id:1000) {
        name
    }
    artist(id:1000) {
        artworks(role: ACTOR) {
            title
        }
    }
}
```



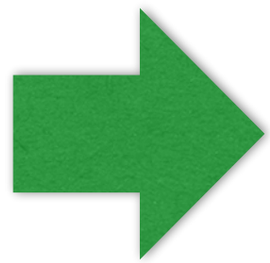
```
{
  “artist” : {
    “name” : “Tom Hanks”,
    “artworks” : [
      {
        …
      },
      {
        …
      },
    ]
  }
}
```

# Query evaluation - Singularities

Query evaluation is not compositional

Selections are “factored-out” in between the recursive calls

```
query {
    artist(id:1000) {
        name
    }
    artist(id:1000) {
        artworks(role: ACTOR) {
            title
        }
    }
}
```



```
{
  “artist” : {
    “name” : “Tom Hanks”,
    “artworks” : [
      {
        …
      },
      {
        …
      },
    ]
  }
}
```

This makes reasoning significantly harder

# Application

# Normalization [H&P, WWW'18]

# Normalization [H&P, WWW'18]

Queries admit a **normal form** that can be evaluated purely compositionally and significantly simplifies reasoning

# Normalization [H&P, WWW'18]

Queries admit a **normal form** that can be evaluated purely compositionally and significantly simplifies reasoning

**But....**

- 👎 Normalization procedure not provided
- 👎 No correctness proof

# Query normalization

# Query normalization

- Certified normalization algorithm

```
Theorem normalized_query_is_in_nf :
 ∀ (φ : query) (s : wfGraphQLSchema),
    is_in_normal_form s (normalize s φ).
```

```
Theorem normalize_preserves_semantics :
  ∀ (φ : query) (s : wfGraphQLSchema) (g : conformedGraph s),
    eval_query (normalize s φ) g s = eval_query φ g s.
```

# Query normalization

- Certified normalization algorithm

```
Theorem normalized_query_is_in_nf :
 ∀ (φ : query) (s : wfGraphQLSchema),
    is_in_normal_form s (normalize s φ).
```

```
Theorem normalize_preserves_semantics :
  ∀ (φ : query) (s : wfGraphQLSchema) (g : conformedGraph s),
    eval_query (normalize s φ) g s = eval_query φ g s.
```

- Simplified evaluation for queries in normal form

```
Theorem simpl_eval_correctness :
 ∀ (φ : query) (s : wfGraphQLSchema) (g : conformedGraph s),
    is_in_normal_form s φ ->
    eval_query φ g s = simpl_eval_query φ g s.
```

# Formalization evaluation and details

# Evaluation

**Effectivity**

Uncovered two issues in H&P formalization:

- Flawed definition of normal form
- Incomplete set of equivalence rules for normalization

# Evaluation

**Effectivity**

Uncovered two issues in H&P formalization:

- Flawed definition of normal form
- Incomplete set of equivalence rules for normalization

**Faithfulness**

Validated with a series of examples from different sources:

- Examples (41) from the SPEC validation section*
- Star Wars example from GraphQL reference implementation
- Example used in H&P

* https://graphql.github.io/graphql-spec/June2018/#sec-Validation

# Conclusion

## Contribution

- First mechanized formalization of GraphQL in the Coq proof assistant
- Certified query normalization algorithm
- Uncover issues in initial formalization [H&P, WWW18]

# Conclusion

## Contribution

- First mechanized formalization of GraphQL in the Coq proof assistant
- Certified query normalization algorithm
- Uncover issues in initial formalization [H&P, WWW18]

## Future work

- Further GraphQL features
- Extraction (certified reference implementation)
- More general data models

# Conclusion

## Contribution

- First mechanized formalization of GraphQL in the Coq proof assistant
- Certified query normalization algorithm
- Uncover issues in initial formalization [H&P, WWW18]

## Future work

- Further GraphQL features
- Extraction (certified reference implementation)
- More general data models

**Thanks!**