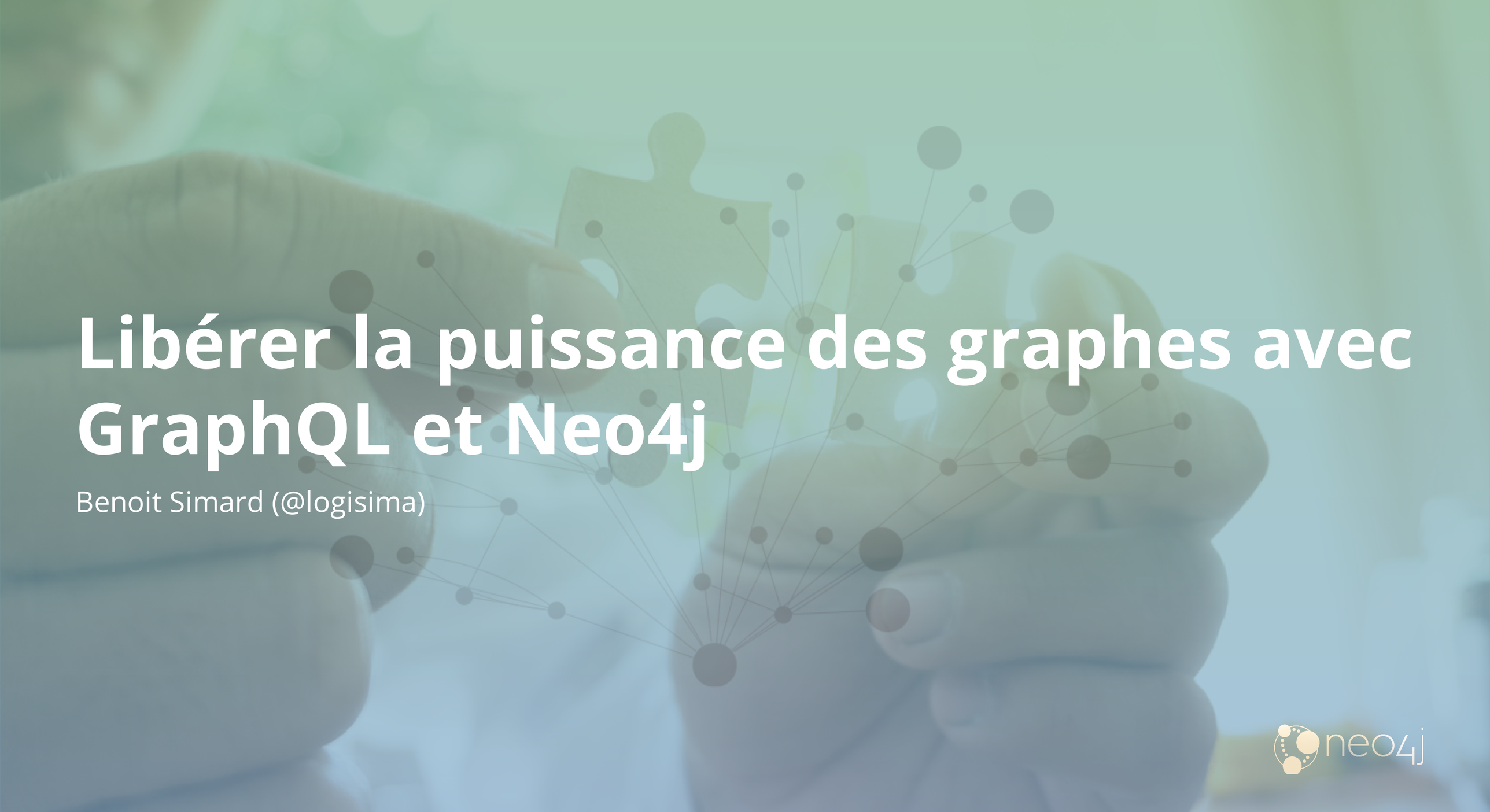
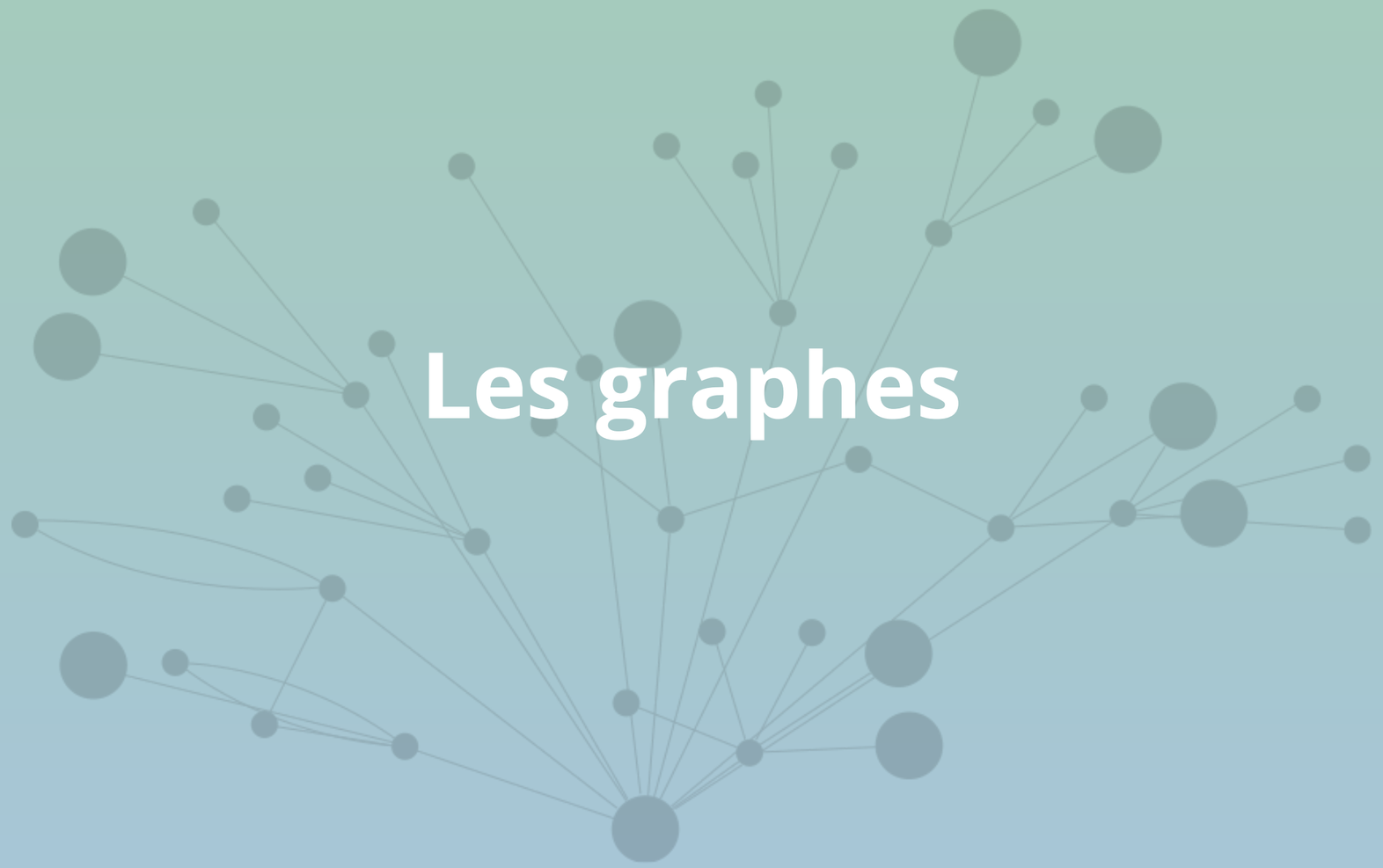


Libérer la puissance des graphes avec GraphQL et Neo4j

Benoit Simard (@logisima)

neo4j



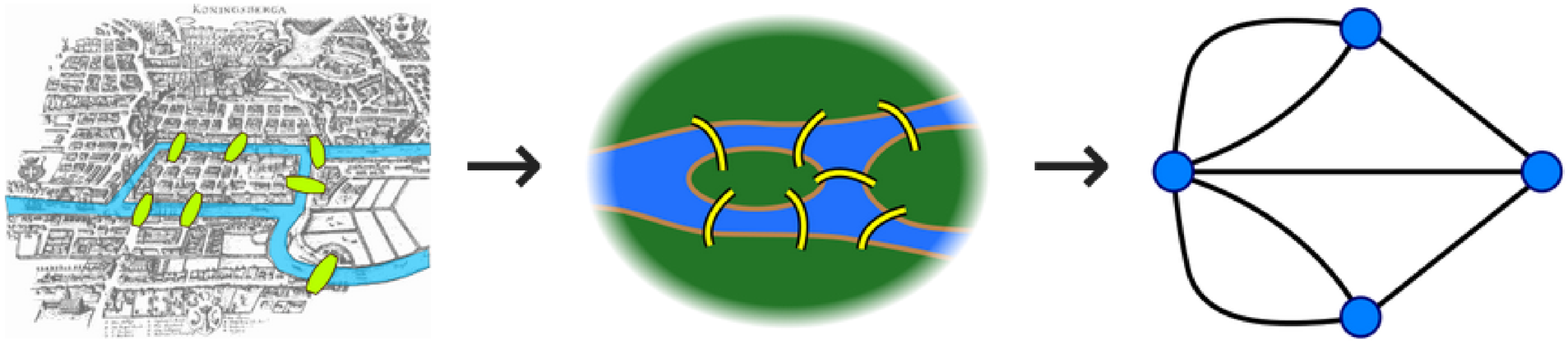
Les graphes

neo4j

Benoit Simard

- **Dépendant aux graphes**
- Libérateur de données
- Développeur Web
- Consultant chez Neo4j
- benoit@neo4j.com
- @logisima

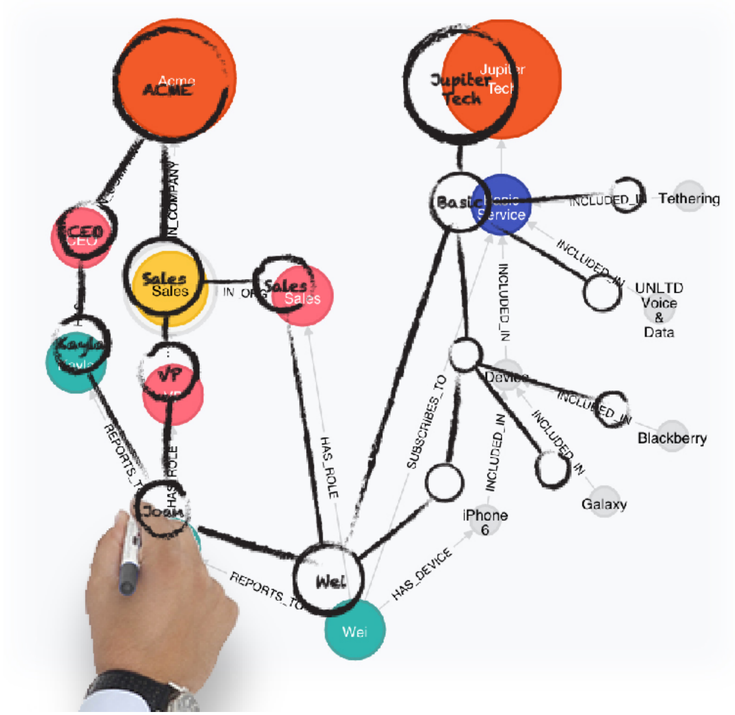
7 ponts de Königsberg



A la base de théorie des graphes par Léonard Euler au 18e

Outil de modélisation

La modélisation en graphe est ultra intuitive

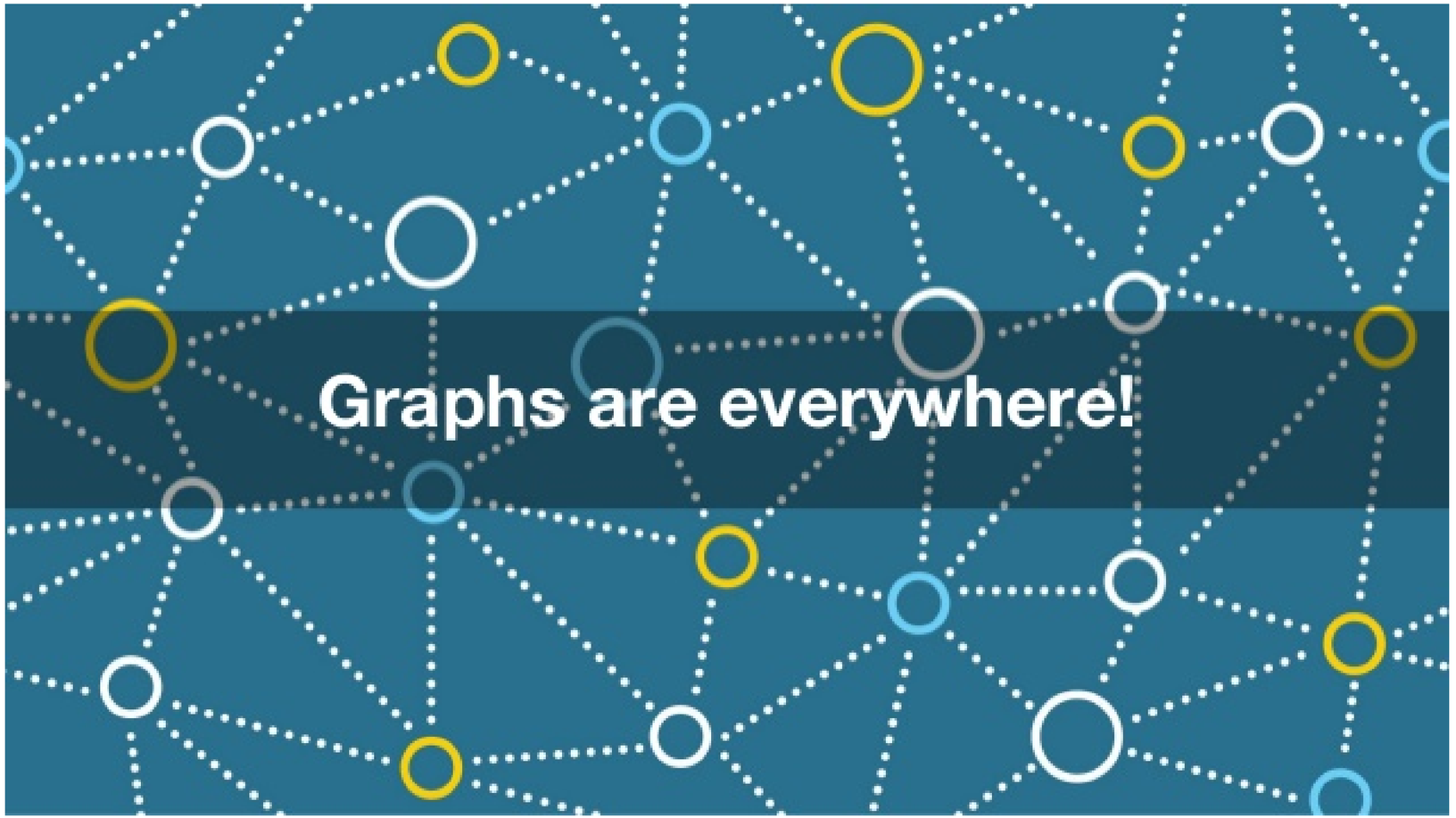


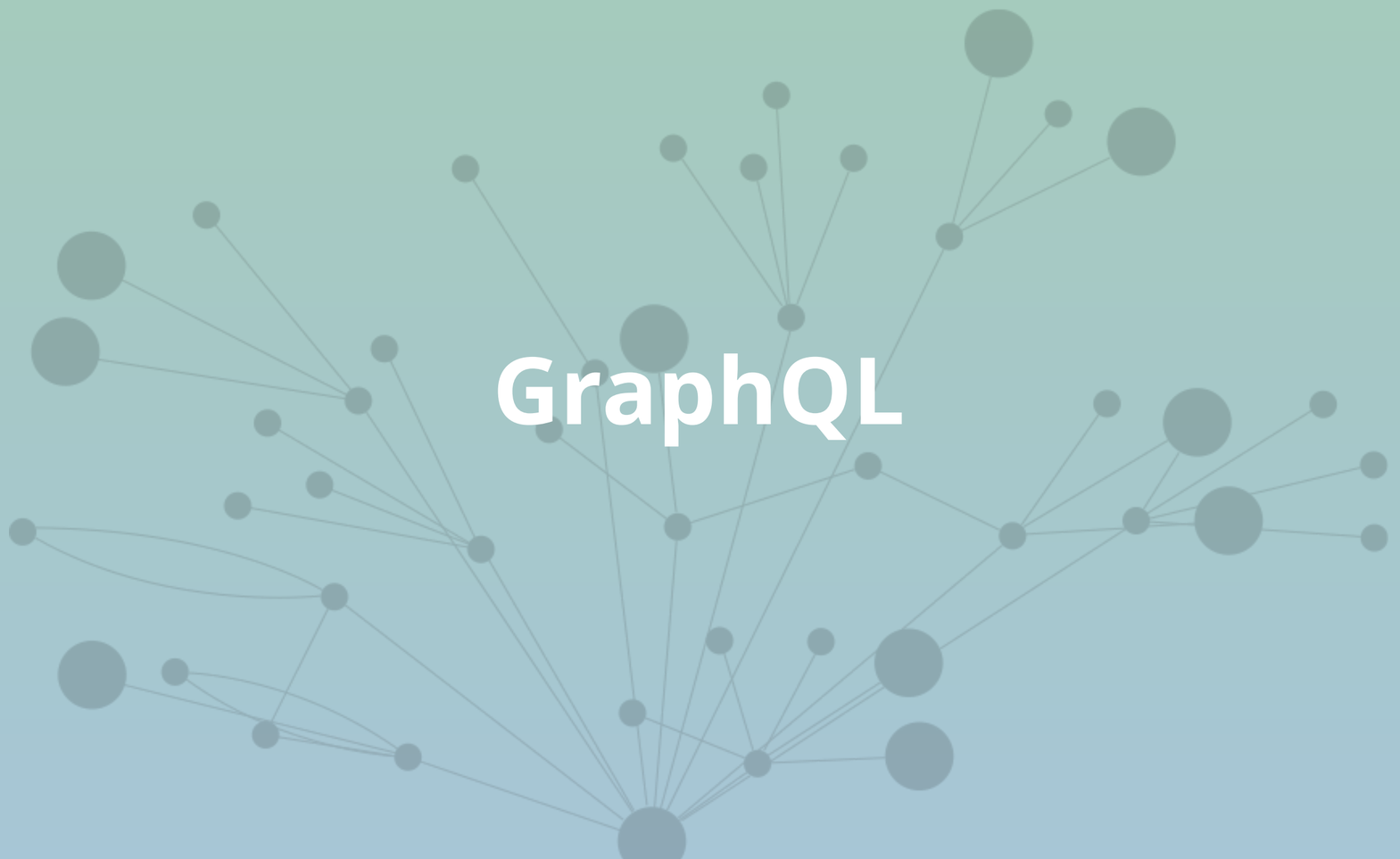
Outil mathématique et informatique

On utilise du graphe tous les jours

- File System
- GPS - chemin le plus court
- Moteur de recherche - Page Rank
- La recommandation : filtrage collaboratif
- Les indexes en base de données - Binary Search Trees
- Big Data : DisinfoLab sur Benalla avec la détection de communautés

Les graphes sont partout





GraphQL

neo4j

Qu’est ce que c’est ?

GraphQL est une spécification de langage

Un langage de requête pour vos APIS

- Développer par Facebook (iOS team)
- Langage déclaratif, où vous définissez les champs que vous souhaitez
- Auto-documenté (basé sur un schéma typé)
- Alternative à REST

Une large adoption / communauté

Quelques extraits de la documentation

- “A query language for your API, and a **server-side** runtime for executing queries by using a **type system** you define for your data”
- “GraphQL isn’t tied to **any specific database** or storage engine”
- “A GraphQL service is created by **defining types and fields** on those types, then providing **resolver** functions for each field on each type”

Prenons un exemple

Requête

```
query($id:ID = 2571) {
  Movie(movieId:$id) {
    movieId
    title
    plot
    poster
    imdbRating
    genres{
      name
    }
    directors {
      name
    }
    actors {
      name
      actedIn {
        title
      }
    }
  }
}
```

Résultat

```
{
  "Movie": {
    "movieId": "2571",
    "title": "Matrix, The",
    "plot": "A computer hacker learns from
  mysterious rebels about the true nature of his
  reality and his role in the war against its
  controllers.",
    "poster": "http://ia.media-
  imdb.com/images/M/MV5BMTkxNDYxOTA4M15BMl5BanBnXk
  FtZTgwNTk0NzQxMTE@._V1_SX300.jpg",
    "imdbRating": 8.7,
    "genres": [ { "name": "Thriller" }, {
  "name": "Sci-Fi" }, { "name": "Action" } ],
    "directors": [ { "name": " Lana Wachowski"
  }, { "name": "Andy Wachowski" } ],
    "actors": [
      {
        "name": "Keanu Reeves",
        "actedIn": [ { "title": "John Wick" }, {
  "title": "47 Ronin" }, ...]
      },
      ...
    ]
  }
```

It’s a Graph !

Votre modèle est un graphe

```
query($id:ID = 2571) {
  Movie(movieId:$id) {
    movieId
    title
    plot
    poster
    imdbRating
    genres{
      name
    }
    directors {
      name
    }
    actors {
      name
      actedIn {
        title
      }
    }
  }
}
```

Schema Definition Language

GraphQL Cheat Sheet

- **Type :** La définition de votre modèle
- **Query :** La liste des fonctions disponibles pour la lecture
- **Mutations :** La liste des fonctions disponibles pour l’écriture

Type

Movie

```
type Movie {
  movieId: ID!
  title: String
  released: Int
  plot: String
  poster: String
  imdbRating: Float
  actors: [Actor]
  recommendations(skip: Int = 0, limit: Int =
5): [Movie]
}
```

Actor

```
interface Person {
  name: ID!
}

type Actor implements Person {
  name: ID!
  actedIn(skip: Int = 0, limit: Int = 5):
[Movie]
}
```

Queries / Mutation

Movie

```
type Query {
  movieById(movieId: ID!): Movie
  movieSearch(search: String = "", skip: Int =
0, limit: Int = 10): [Movie]
}

type Mutation {
  movieMerge(movieId: ID, title: String!,
actors:[String], ...): Movie
}
```

Actor

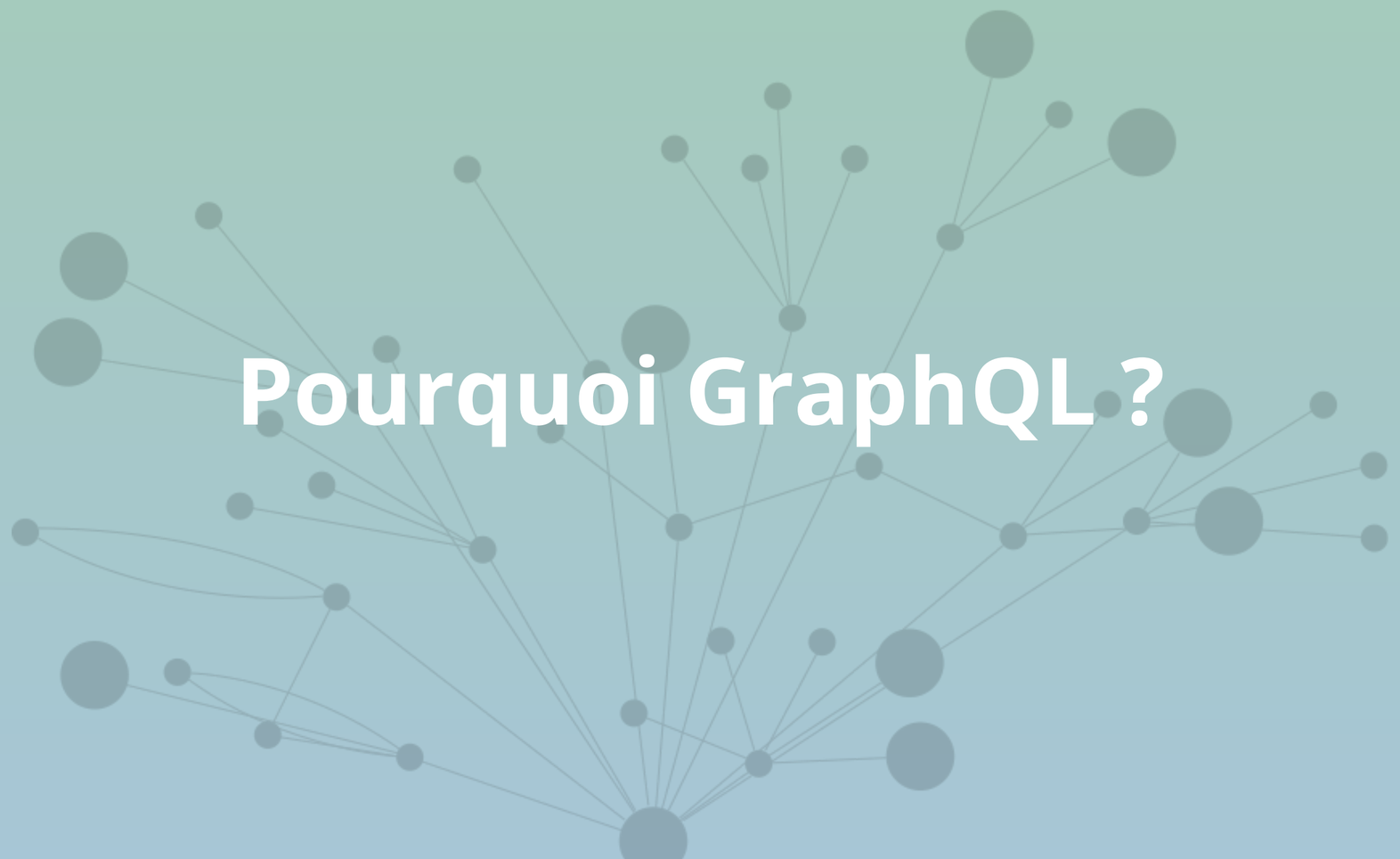
```
type Query {
  actorById(id: ID!): Actor
  actorSearch(search:String = ""): [Actor]
}

type Mutation {
  actorMerge(name:String!): Actor
}
```

Resolvers

C’est une fonction où vous codez comment récupérer la donnée

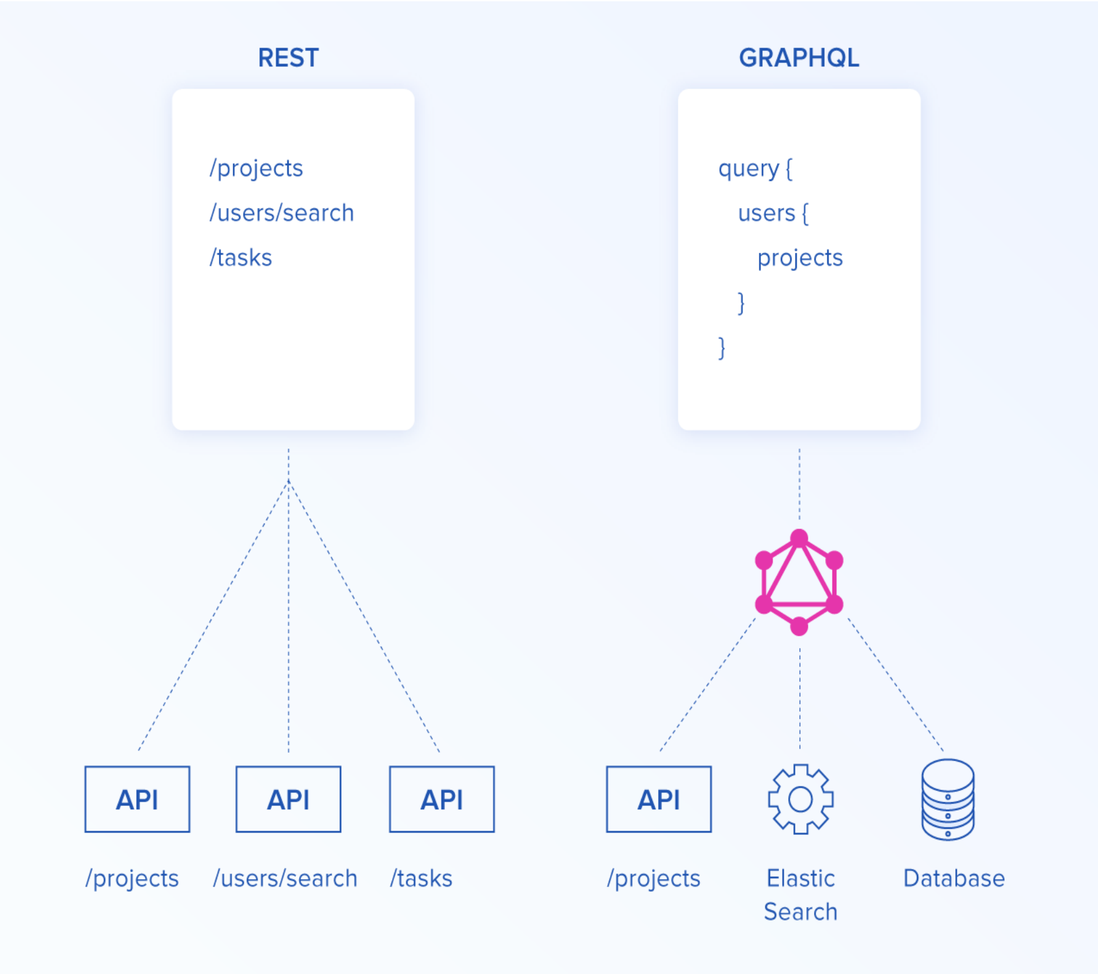
```
Movie {
  recommendations: async ( obj, params, context, resolverInfo) => {
    // Your code here
    return [];
  }
},
Query: {
  movieById: async (obj, params, context, resolverInfo) => {
    // Your code here
    return {};
  },
  movieSearch: async (root, params, context, resolverInfo) => {
    // Your code here
    return [];
  },
}
```

- **obj :** Le résultat de l’objet parent. Dans le cas de la recommandation, ce sera l’objet film.
- **params :** La liste des arguments passée aux fonctions.
- **context :** Objet partagé par tous les *resolvers* d’une même requête
- **resolverInfo :** Réservé pour les usages avancés



Pourquoi GraphQL ?

Il existe déja REST non ?



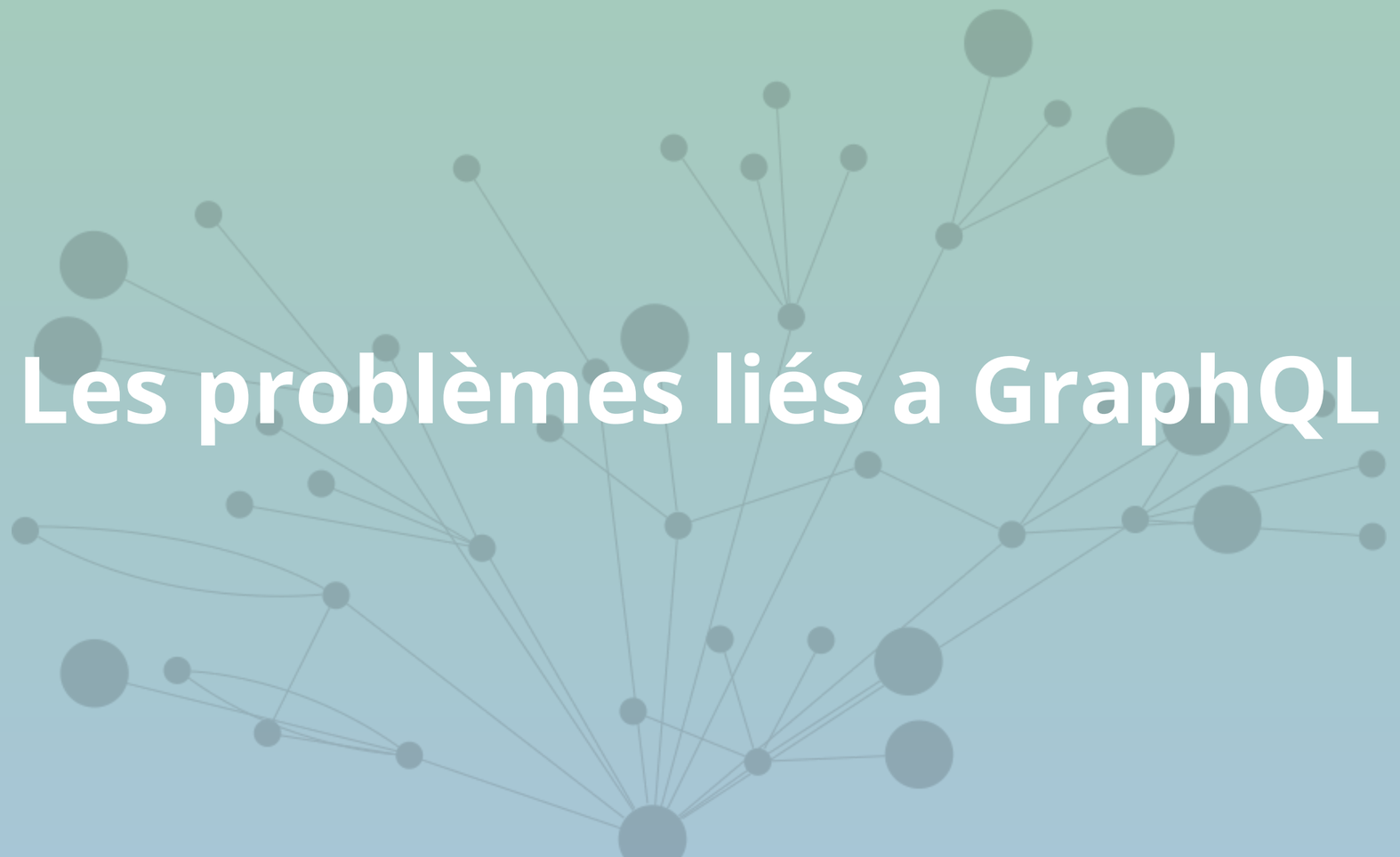
- Typage des éléments
- C’est un langage de requête : vous définissez le retour que vous souhaitez !
- Une seule requête au lieu de X : le modèle est un graphe et vos requêtes aussi

Un petit goût de savon ?

GraphQL et SOAP

- Tout est fait en **POST** dans un body sur un **unique endpoint**
- Tout est défini dans un **schéma** faisant l'interface entre le client/serveur
- Le système de typage est **fort**
- **MAIS :**
  - pas de XML
  - facile à implémenter
  - focaliser sur les données (non service)
  - *agnostique sur la technologie*

Les problèmes liés a GraphQL



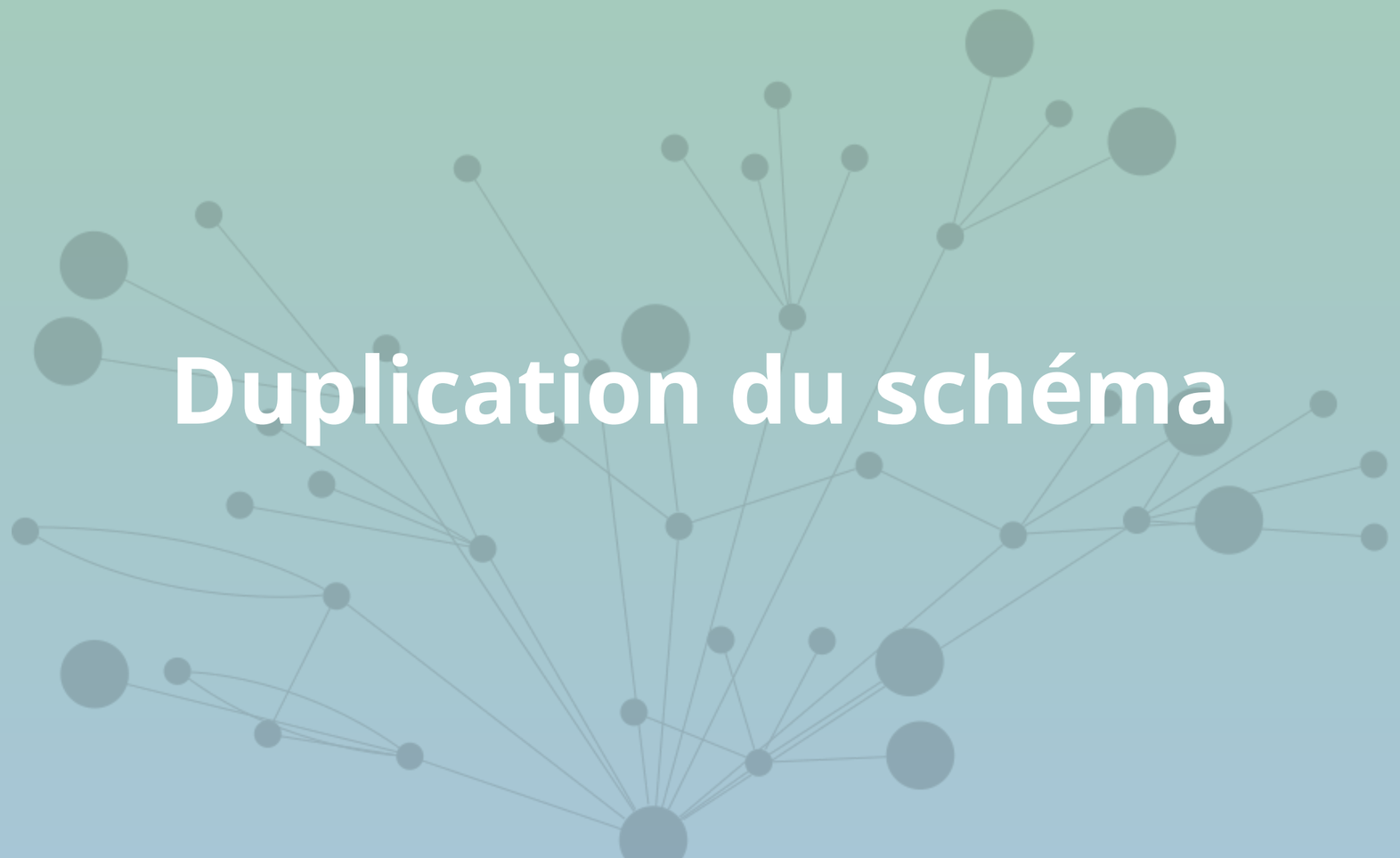
neo4j

Les 5 problèmes de GraphQL

Five Common Problems in GraphQL Apps

de Sacha Greif.

Duplication du schéma



Explication

Namely, you need one schema for your database, and another one for your GraphQL endpoint.

Your database and GraphQL API will have different schemas, which translate into different document shapes.

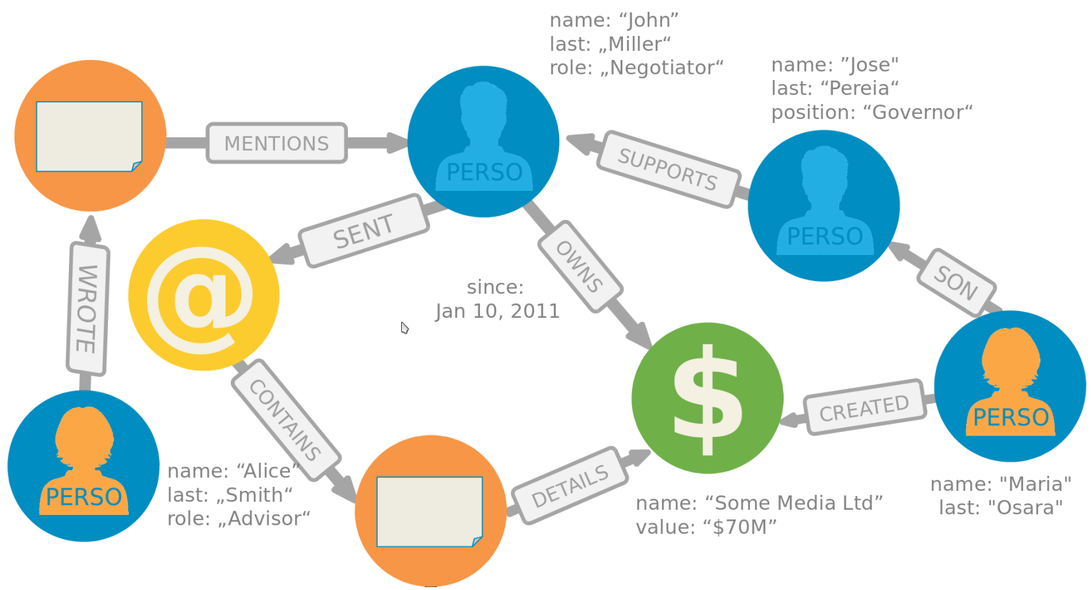
- Un schema pour la base
- Un schéma pour GraphQL
- Une fonction de transformation

Et si on les fusionne ?

Il faut une base qui gère le modèle de GraphQL

*Les bases **relationnelles** et **graphes***

Un Graphe de propriétés



Noeuds

- Les entités du model
- Peuvent avoir des `labels`
- Peuvent avoir des `propriétés`

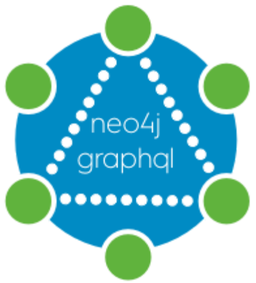
Relations

- Lient deux noeuds entre eux, avec une `direction` et un `type`
- Peuvent avoir des `propriétés`

neo4j et graphql

Première règle de GraphQL : tout commence et finit par le schéma !

https://github.com/neo4j-graphql



Permet de gérer toutes les données de la base, grâce a la définition d'un schéma GraphQL (auto-généré ou fourni).

Exemple de schéma

```
type Movie {
  movieId: ID!
  title: String
  released: Int
  plot: String
  poster: String
  imdbRating: Float
  genres: [Genre] @relation(name: "IN_GENRE", direction:"OUT")
  directors: [Director] @relation(name: "DIRECTED", direction:"IN")
  actors: [Actor] @relation(name: "ACTED_IN", direction:"IN")
  recommendations(skip: Int = 0, limit: Int = 10): [Movie] @cypher(statement:"MATCH ... RETURN m SKIP $skip LIMIT $limit")
}

type Actor {
  name: ID!
  actedIn: [Movie] @relation(name: "ACTED_IN", direction:"OUT")
}

type Director {
  name: ID!
  directed: [Movie] @relation(name: "DIRECTED", direction:"OUT")
}
```

La magie - les queries

Neo4j génère automatiquement les queries

`Movie(movieId, title, released, …, first, offset, _id, orderBy, filter): [Movie]`

```
query Nineties($released: Long, $letter: String){
  Movie(released: $released,
    filter: {
      title_starts_with: $letter,
      actors_some: { name_contains: $letter}
    }
  ) {
    title
    released
    actors(first: 3) {
      name
      born
      movies(first: 1, orderBy: title_desc) {
        title
        released
      }
    }
  }
}

# query variables
{ "released":1995, "letter":"A"}
```

- Tous les champs du type sont définis dans la query, pour un controle d'égalité
- `filter` vous permet de réaliser des filtres plus complexes
- `orderBy` pour trier vos résultats
- `first` & `offset` pour la pagination

La magie - les mutations

Neo4j génère automatiquement les mutations

vous pouvez aussi surcharger leurs comportements

```
// Pour les noeuds
createMovie(movieId: ID!, title: String, released: Int, ...) : String
updateMovie(movieId: ID!, title: String, released: Int, ...) : String
deleteMovie(movieId: ID!) : String

// Pour les relations
addMovieActors(movieId: ID!, actors:[ID]!) : String
deleteMovieActors(movieId: ID!, actors:[ID]!) : String
```

```
// query
mutation {
    createActor(name:"Chadwick Boseman", born: 1977)
}
// resultat
{ "data": {
    "createActor": "Nodes created: 1\nProperties set: 2\nLabels added: 1\n"
  }
}
```

Beaucoup de code pour pas grand chose …

This is by no means an issue exclusive to GraphQL apps, but it's true that they generally require you to write a lot of similar boilerplate code.

Des performances au rabais

neo4j

Explication

On one hand you want to take full advantage of GraphQL's graph traversal features ("show me the authors of the comments of the author of the post of..." etc.). But on the other hand, you don't want your app to become slow and unresponsive.

GraphQL

```
query($id:ID = 2571) {
  Post(id:$id) {
    title,
    text,
    comments {
      author {
        name {
          posts {
            title,
            comments  {
              author {
                name,
                ...
              }
            }
          }
        }
      }
    }
```

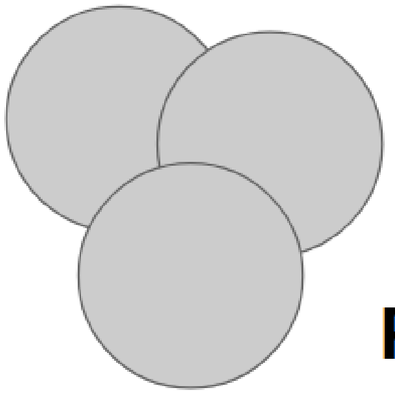
SQL

Désolé j'ai pas assez de place pour les jointures :)

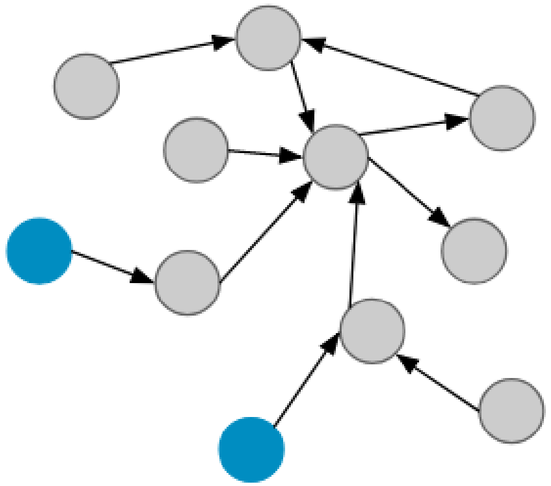
Killer feature de GraphQl

Au sens propre et figuré

Le paradigme des bases de données graphe

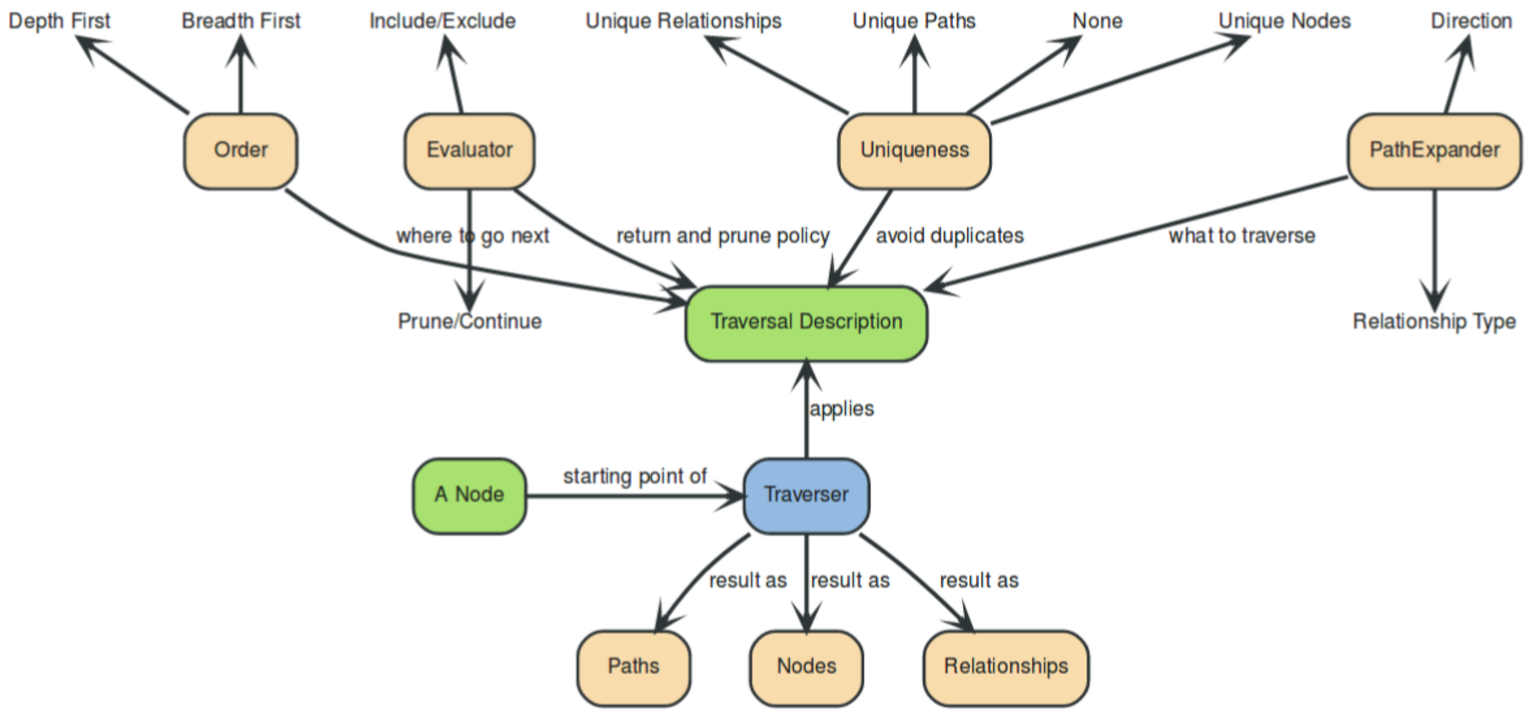


Relational



Graph

Les traversés en graphe



C'est un peu barbar mais il y a plus simple, attendez un peu :)

Des requêtes en nombres et parfois inutiles

neo4j

Explication

Imagine a list of posts, each of which has a user attached to it. You now want to display 10 of these posts, along with the name of their author.

- Une requête pour avoir la liste des `posts`
- Une requête **par** `post` pour récupérer les auteurs

Cette unique requête GraphQL fait 11 appels à la base !!!

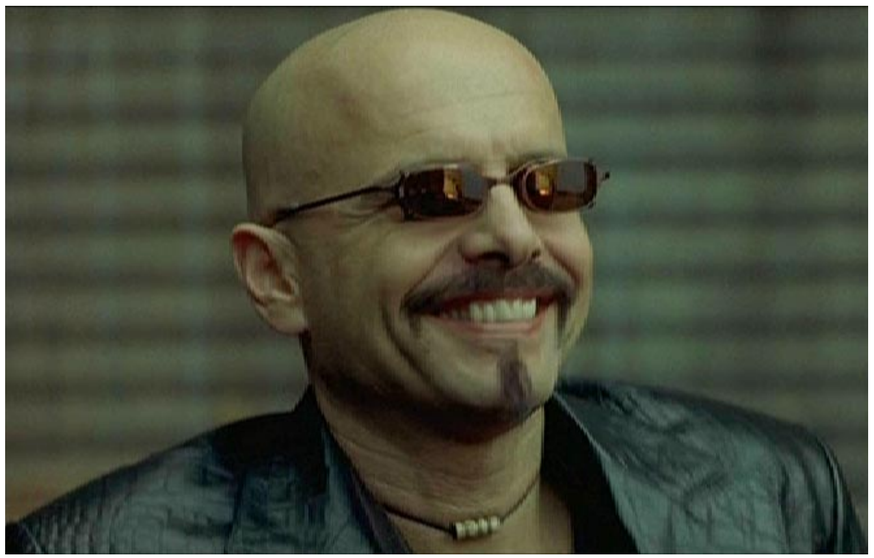
Une solution ?

Dataloader

Batche les requêtes et met en place un système de cache sur les objets basés sur leurs `ID`.

Et si on pouvait générer qu'une seule requête ?

Et laisser la base se débrouiller toute seule, c'est son job non ?

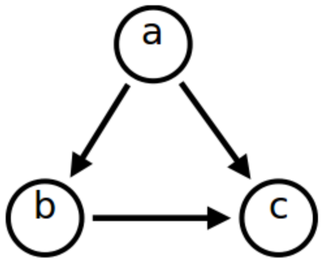


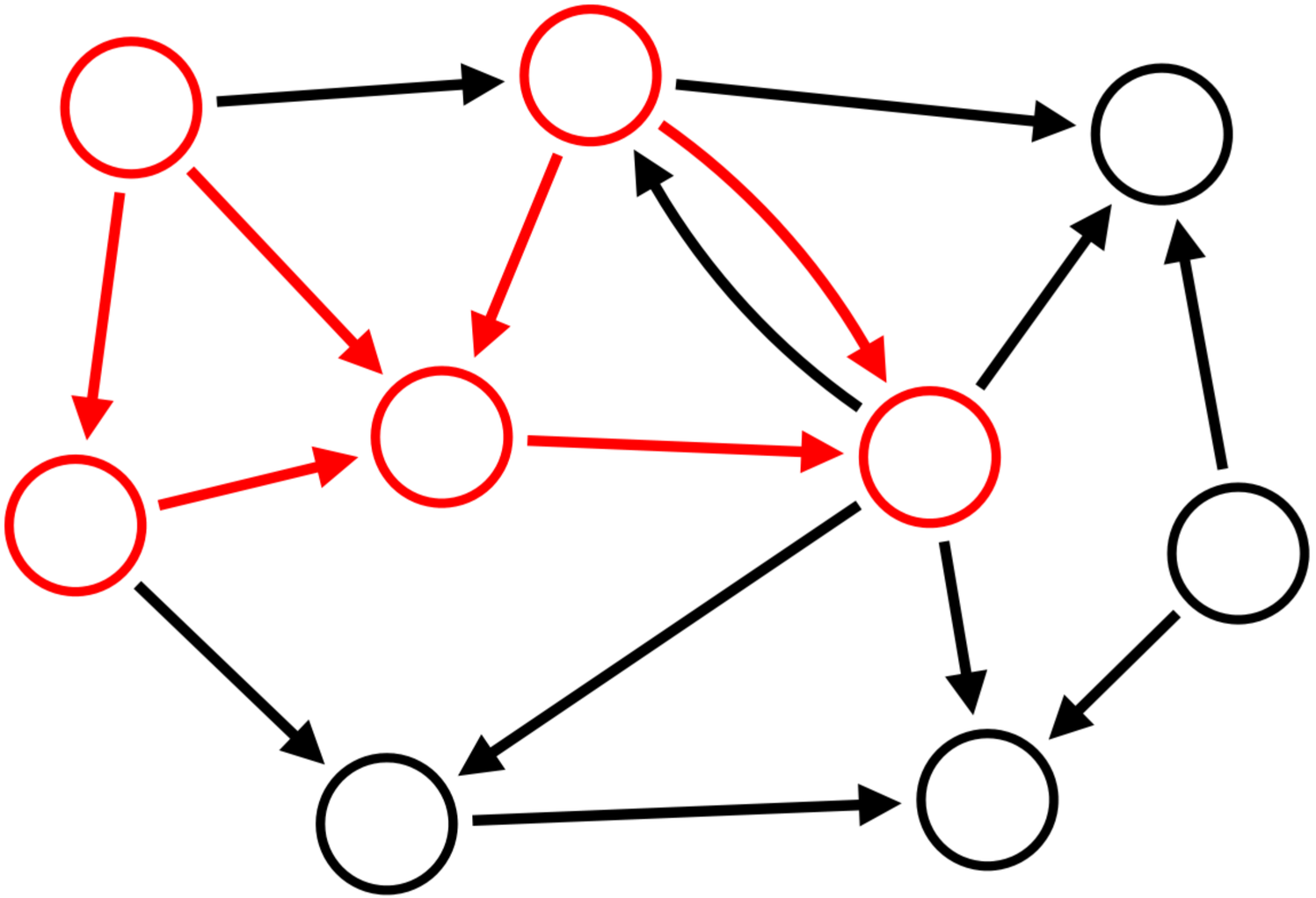
Une requête en **Cypher**

Cypher

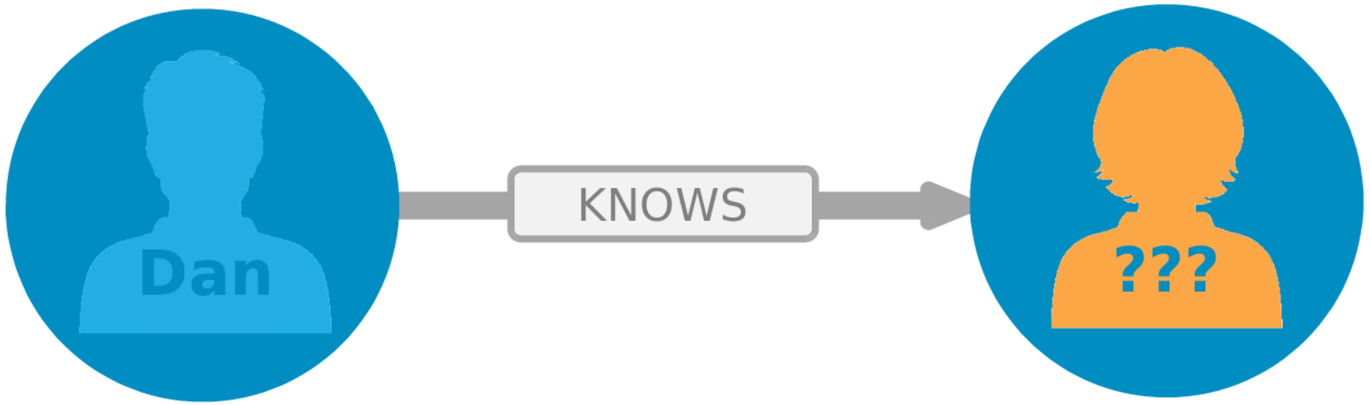
- SQL du Graph
- Langage déclaratif
- Développer par Neo4j en 2013
- open-source depuis 2016 (opencypher)
- Implémenter dans Neo4j, SAP Hana, Redis, AgensGraph, ...
- Est en cours standardisation (https://gql.today/)

Cypher - une histoire pattern





Cypher - ASCIIART



NODE NODE

```
MATCH   (:Person { name:"Dan"} ) -[:KNOWS]-> (who:Person)
```

LABEL PROPERTY LABEL

```
RETURN who
```

Cypher - Example

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->()<-[:ACTED_IN]-(cocoActors) AND tom <> cocoActors
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Cypher - GraphQL

Same, same … but different

GraphQL

```
{
  Movie(title: "River Runs Through It, A") {
    title
    year
    imdbRating
    actors {
      name
    }
  }
}
```

Cypher

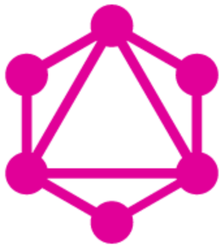
```
MATCH (movie:Movie {title:"River Runs Through
It, A"})
RETURN
  movie {
    .title ,
    .year ,
    .imdbRating,
    actors: [
      (movie)<-[ACTED_IN]-(movie_actors:Actor) |
      movie_actors {
        .name
      }
    ]
  } AS movie
SKIP 0
```

GRAND Stack

neo4j

GRAND Stack

http://grandstack.io/



GraphQL

React

Apollo

Neo4j Database

Apollo

“A set of tools designed to leverage GraphQL and work together to create a great workflow”

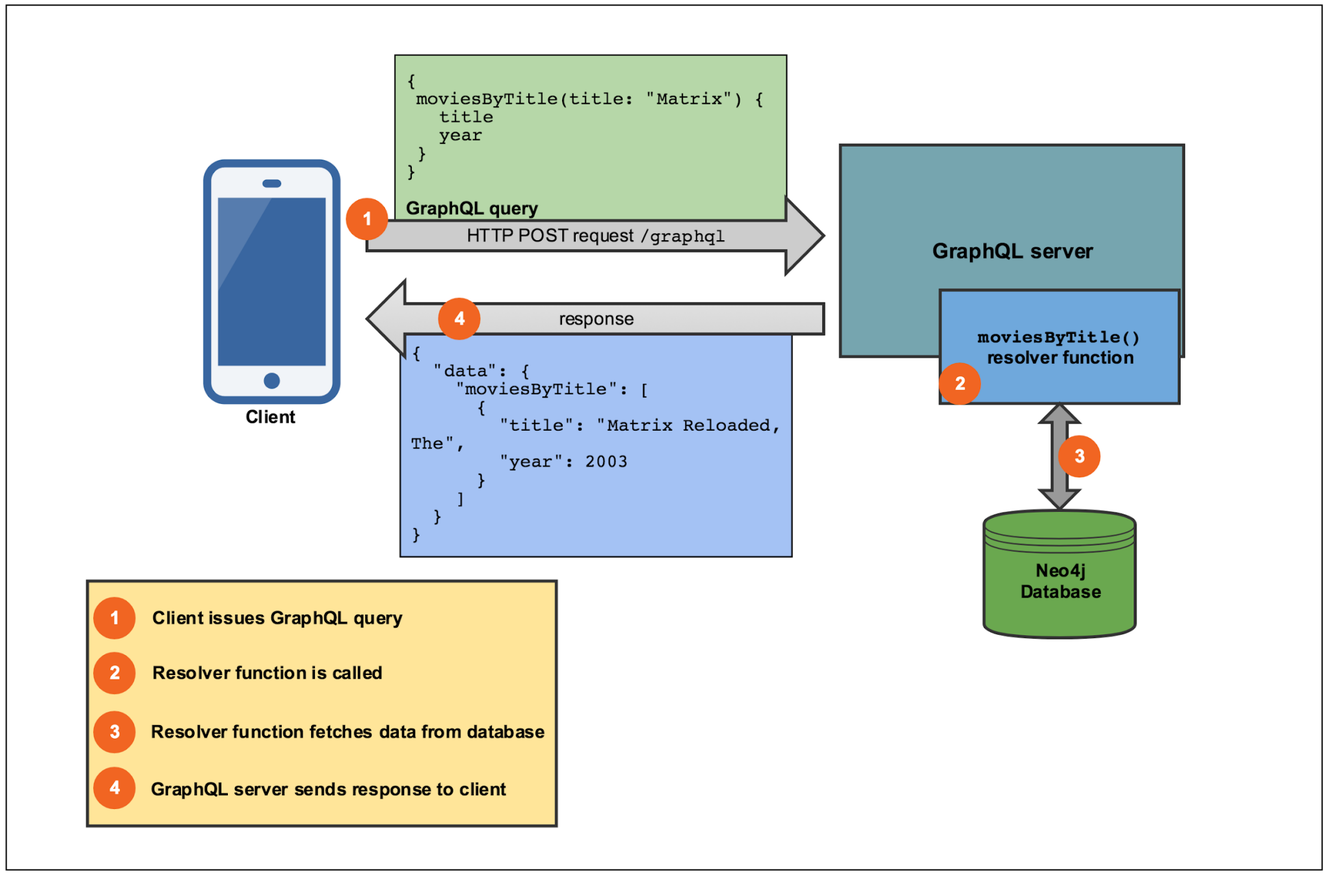
Client-side tooling

- Intégration avec les framework frontend
- Politique de cache
- Génération de code

Server-side tooling

- Création du schema avec ses resolvers
- Mocking
- Monitoring

Architecture



Kit de démarrage

```
git clone https://github.com/grand-stack/grand-stack-starter.git
```

Merci

neo4j

Des questions ?