Tracce di esami, esempi e confronti di linguaggi NoSQL, SPARQL e Datalog

Raffaele D'Anna e GPT-3.5

September 17, 2024

1 Introduzione

Questo documento presenta un confronto tra diversi esempi di linguaggi di query utilizzati in database NoSQL, SPARQL e Datalog. Gli esempi forniti mettono in evidenza le differenze sintattiche tra MongoDB, Cypher, SPARQL e Datalog. Inoltre, sono anche presenti tracce di esami con relativi svolgimenti e possibili soluzioni.
Ovviamente, non è detto che tutto sia corretto, quindi è consigliabile approfondire ulteriormente e verificare le informazioni prima di trarre conclusioni definitive. È inoltre possibile che ci siano più soluzioni valide per un determinato esercizio, e la scelta della soluzione può dipendere da diversi fattori

2 Database e Linguaggi

- **Database Documentali:** MongoDB
- **Database Grafici:** Neo4j (Cypher)
- **Database RDF:** SPARQL
- **Database Logici:** Datalog

3 Esami svolti

- Esame del 26/06/2024
- Esame del 26/07/2024
- Esame del 12/09/2024

4 Database Documentali

4.1 MongoDB

Nota:

- Utilizzare $set quando si vuole assegnare un valore fisso a un campo.
  Esempio: Assegna direttamente un valore specificato a un campo. Con $set impostiamo il prezzo a 1.10 per tutti i prodotti nella categoria "Elettronica". Il prezzo di tutti i prodotti "Elettronica" diventerà esattamente 1.10.

- Utilizzare $mul quando si vuole modificare il valore attuale di un campo con un moltiplicatore.
  Esempio: Moltiplica il valore attuale di un campo per un fattore specificato. Con $mul aumentiamo il prezzo del 10% attraverso la seguente istruzione: { "prezzo": 1.10 }, che moltiplicherebbe il prezzo corrente per 1.10 (aggiungendo così il 10%).

Esercizio 1

Un sistema informativo deve contenere dati relativi a pazienti di una clinica medica e alle visite mediche da essi effettuate. Ogni visita è associata a un singolo paziente, ma ogni paziente può effettuare più visite. Il paziente Anna Rossi ha 45 anni, vive a Milano e ha il numero di telefono 02-1234567. Il paziente Marco Bianchi ha 32 anni, vive a Roma e ha un'email marco.bianchi@example.com. Anna Rossi ha effettuato una visita il 10-05-2024 con il Dr. Luca Verdi per una "Visita cardiologica". Marco Bianchi ha effettuato una visita il 15-07-2024 con il Dr. Giovanni Neri per un "Controllo annuale". Anna Rossi ha effettuato un'altra visita il 20-08-2024 con la Dr.ssa Elena Blu per un "Esame del sangue". Si stima che la base di dati dovrà contenere dati simili a quelli descritti, con circa 50.000 pazienti e 150.000 visite. Si prevede inoltre che saranno eseguite le seguenti operazioni con relative frequenze stimate:
Q1: Dato un identificativo di paziente p, trovare tutti i medici che hanno effettuato visite per il paziente identificato da p. (2.000 interrogazioni al giorno)
Q2: Dato l'identificativo v di una visita, trovare la descrizione della visita in questione. (3.000 interrogazioni al giorno)
Q3: Inserimento di una nuova visita con data, medico e paziente.

a) **Come memorizzeresti i dati di esempio, rappresentandoli in JSON, considerando le operazioni e la loro frequenza? Spiega brevemente le eventuali denormalizzazioni.**

```
{
  "pazienti": [
      {
        "pazId": 1,
```

```
        "nome": "Anna Rossi",
        "anni": 45,
        "localita": "Milano",
        "telefono": "02-1234567",
        "email": "",
        "visite": [
          {
            "visId": 1,
            "data": "10-05-2024",
            "dottore": "Luca Verdi",
            "tipologia": "Visita cardiologica",
          },
          {
            "visId": 2,
            "data": "20-08-2024",
            "dottore": "Elena Blu",
            "tipologia": "Esame del sangue",
          }
        ]
      },
      {
        "pazId": 2,
        "nome": "Marco Bianchi",
        "anni": 32,
        "localita": "Roma",
        "telefono": "",
        "email": "marco.bianchi@example.com",
        "visite": [
          {
            "visId": 3,
            "data": "15-07-2024",
            "dottore": "Giovanni Neri",
            "tipologia": "Controllo annuale",
          }
        ]
      }
    ]
}
```

n questa struttura, i dati dei pazienti e delle visite sono denormalizzati. Le informazioni sui medici sono ripetute all'interno di ogni visita, il che può portare a una ridondanza di dati. Tuttavia, questa scelta è giustificata dalle frequenti interrogazioni che richiedono di ottenere tutte le visite di un paziente o i dettagli di una visita specifica, migliorando le prestazioni di queste query.

b) **Si formulino le query in MongoDB Query Language (con valore a piacere)**

Query Q1

- **Soluzione 1**:

```
db.pazienti.find(
  {"pazId": 1},
```

```
  {"visite.dottore": 1, _id: 0},
)
```

- **Soluzione 2**:

```
db.pazienti.aggregate([
  // Filtra il paziente per quello di interesse
  { $match: { "pazId": 1 } },
  // Espande le visite in documenti singoli
  { $unwind: "$visite" },
  // Raggruppa per il campo dottore
  { $group: { _id: "$visite.dottore" } },
  // Proietta solo il campo dottore senza duplicati
  { $project: { _id: 0, dottore: "$_id" } }
])
```

Query Q2

- **Soluzione 1**:

```
db.pazienti.find(
  {"visite.visId": 1},
  {"visite.$": 1, _id: 0}
)
```

- **Soluzione 2**:

```
db.pazienti.aggregate([
  --Espande le visite in documenti singoli
  { $unwind: "$visite" },
  --Filtra le visite per quella di interesse
  { $match: { "visite.visId": 1 } },
  --Raggruppa per il campo tipologia
  { $group: { _id: "$visite.tipologia" } },
  --Proietta solo il campo tipologia senza duplicati
  { $project: { _id: 0, tipologia: "$_id" } }
])
```

Query Q3

```
db.pazienti.updateOne(
  { pazId: 2 },
  {
    $push: {
      visite: {
        visId: 4,
        data: "30-09-2024",
        dottore: "Mario␣Rossi",
        tipologia: "Visita␣oculistica"
      }
    }
  }
)
```

E' possibile testare l'esercizio con il seguente compilatore online: OneCompiler

Altri esempi

```
db.ordini.aggregate([
  {
    --Decompone l'array di prodotti in documenti separat
    $unwind: "$prodotti"
  },
  {
    --Raggruppa per cliente
    $group: {
      _id: "$cliente",
      totaleSpeso: { $sum: { $multiply: ["$prodotti.unita", "
          $prodotti.prezzo"] } } --moltiplica unita * prodotti
    }
  },
  {
    $match: {
      totaleSpeso: { $gt: 2000 }
    }
  },
  {
    $project: {
      _id: 0,
      cliente: "$_id",
      totaleSpeso: 1
    }
  }
]);
```

Listing 1: Query MongoDB per cercare tutti i clienti che hanno speso più di una certa somma totale (2000) su tutti gli ordini

```
db.ordini.aggregate([
  {
    $unwind: "$prodotti"
  },
  {
    $group: {
      _id: { cliente: "$cliente", prodId: "$prodotti.prodId" },
      ordiniCount: { $sum: 1 }
    }
  },
  {
    $match: {
      ordiniCount: { $gt: 1 }
    }
  },
  {
    $group: {
      _id: "$_id.cliente",
      prodottiRipetuti: { $push: "$_id.prodId" }
    }
  }
]);
```

Listing 2: Query MongoDB per individuare i clienti che hanno ordinato lo stesso

prodotto in ordini diversi.

```
school.students.updateMany(
   { "name": "John Doe" },
   { $set: { "age": 22 } }
)
```

Listing 3: Query MongoDB su db di nome school per aggiornare l'età di tutti gli studenti chiamati "John Doe" a 22.

```
school.students.updateOne(
   { _id: ObjectId("5f1a9cbd1c4ae23a2c5d9e34") },
   { $push: { "grades": 90 } } --grades e' una lista di voti
)
```

Listing 4: Query MongoDB su db di nome school aggiungere un nuovo voto (90) alla lista dei voti per uno studente con id specifico.

```
db.books.find(
  {
    "year": { $gt: 2000 },
    "genres": { $elemMatch: { $eq: "Fantasy" } },
    "copies_sold": { $gt: 1000000 }
  }
)
```

Listing 5: Query MongoDB per trovare libri pubblicati dopo il 2000 nel genere Fantasy con più di 1.000.000 di copie vendute.

```
db.studenti.findOne(
  {
    "matricola": { "$eq": "matricola1" }
  },
  {
    "_id": 0,
    "cdl": 1
  }
)
```

Listing 6: Query MongoDB che data una matricola m, trova il corso di laurea a cui lo studente con matricola m è iscritto.

```
db.studenti.findOne(
  {
    "nome": { "$eq": "n" }
  },
  {
    "_id": 0,
    "dipartimenti.sede": 1
  }
)
```

Listing 7: Query MongoDB che Ddato un dipendente con nome n, trova la sede del dipartimento in cui lavora.

5 Database Grafici

5.1 Cypher

Nota: Rispetto a SQL, in Cypher, non è necessario specificare la clausola GROUP BY, in quanto Cypher è progettato per gestire automaticamente l'aggregazione dei dati in base alle variabili specificate nella query.

E' possibile passare dei boundary da una sottoquery ad un'altra. Tuttavia, è importante notare che tutto ciò che non viene incluso nella clausola *WITH* nella sottoquery non può essere utilizzato successivamente. La clausola *WITH* viene utilizzata per manipolare e filtrare i risultati intermedi prima di passarli alla prossima operazione nella query.

Esercizio 1

Un sistema deve rappresentare una rete sociale di professionisti e le connessioni professionali tra di loro. Ogni professionista può avere diversi colleghi, e alcuni possono lavorare per la stessa azienda. Le aziende possono essere connesse attraverso progetti comuni. Consideriamo i seguenti dati:

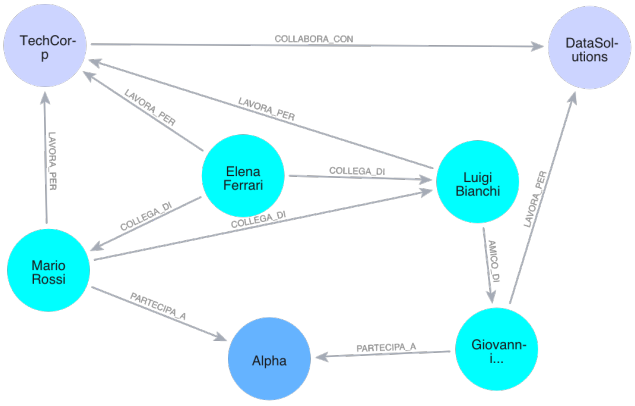
- Elena Ferrari lavora per TechCorp e ha come colleghi Mario Rossi e Luigi Bianchi.
- Mario Rossi lavora per TechCorp e ha partecipato al progetto Alpha con DataSolutions.
- Luigi Bianchi lavora per TechCorp ed è amico di Giovanni Verdi.
- Giovanni Verdi lavora per DataSolutions ed è coinvolto nel progetto Alpha.
- DataSolutions è collegata a TechCorp attraverso il progetto Alpha.

a) **Rappresenta i dati sopra descritti in un property graph. Indica chiaramente i nodi e le relazioni.**

```
// Nodi Persone
CREATE (elena:Persona {name: "Elena Ferrari"})
CREATE (mario:Persona {name: "Mario Rossi"})
CREATE (luigi:Persona {name: "Luigi Bianchi"})
CREATE (giovanni:Persona {name: "Giovanni Verdi"})

// Nodi Aziende
CREATE (techCorp:Azienda {name: "TechCorp"})
CREATE (dataSolutions:Azienda {name: "DataSolutions"})

// Nodo Progetto
CREATE (alpha:Progetto {name: "Alpha"})

// Relazioni Colleghi
CREATE (elena)-[:COLLEGA_DI]->(mario)
```

```
CREATE (elena)-[:COLLEGA_DI]->(luigi)
CREATE (mario)-[:COLLEGA_DI]->(luigi)

// Relazioni Amici
CREATE (luigi)-[:AMICO_DI]->(giovanni)

// Relazioni Lavora
CREATE (elena)-[:LAVORA_PER]->(techCorp)
CREATE (mario)-[:LAVORA_PER]->(techCorp)
CREATE (luigi)-[:LAVORA_PER]->(techCorp)
CREATE (giovanni)-[:LAVORA_PER]->(dataSolutions)

// Relazione Progetto
CREATE (mario)-[:PARTECIPA_A]->(alpha)
CREATE (giovanni)-[:PARTECIPA_A]->(alpha)
CREATE (techCorp)-[:COLLABORA_CON]->(dataSolutions)
```

Listing 8: Codice per la creazione del grafo in Neo4j



b) **Scrivi le seguenti query in Cypher:**

- **Trova tutte le aziende che hanno un dipendente che è amico di un dipendente di TechCorp.**
  - Soluzione 1

```
MATCH (a:Azienda)<-[:LAVORA_PER]-(amico:Persona)
    -[:AMICO_DI]-(dipendente:Persona)-[:LAVORA_PER
    ]->(techCorp:Azienda {name: "TechCorp"})
RETURN DISTINCT a.name
```

  - Soluzione 2

```
MATCH (azienda:Azienda {name: "TechCorp"})<-[:
    LAVORA_PER]-(persona:Persona)-[:AMICO_DI]-(
    persona2:Persona)-[:LAVORA_PER]->(azienda2:
    Azienda)
RETURN azienda2.name
```

- **Trova tutti i colleghi di Elena Ferrari**

```
MATCH (elena:Persona {name: "Elena Ferrari"})-[:COLLEGA_DI]-(collega:Persona)
RETURN collega.name
```

- **Trova tutte le persone che lavorano per DataSolutions.**

```
MATCH (persona:Persona)-[:LAVORA_PER]->(azienda:Azienda {name: "DataSolutions"})
RETURN persona.name
```

- **Trova tutti i progetti a cui ha partecipato un dipendente di TechCorp.**

```
MATCH (azienda: Azienda {name: "TechCorp"})<-[:LAVORA_PER]-(dipendente: Persona)-[:PARTECIPA_A]->(progetto: Progetto)
RETURN progetto.name
```

- **Trova tutte le persone che lavorano per la stessa azienda di un amico di Luigi Bianchi.**
  - Soluzione 1

```
MATCH (luigi:Persona {name: "Luigi Bianchi"})-[:AMICO_DI]-(amico:Persona)-[:LAVORA_PER]->(azienda:Azienda)<-[:LAVORA_PER]-(collega:Persona)
RETURN DISTINCT collega.name
```

  - Soluzione 2

```
MATCH (amici: Persona)-[:AMICO_DI]-(luigi: Persona {name: "Luigi Bianchi"})
WITH amici
MATCH (amici)-[:LAVORA_PER]->(azienda: Azienda)<-[:LAVORA_PER]-(persona: Persona)
RETURN DISTINCT persona.name
```

- **Trova tutte le aziende che sono collegate a TechCorp.**

```
MATCH (techCorp:Azienda {name: "TechCorp"})-[:COLLABORA_CON]->(altraAzienda:Azienda)
RETURN altraAzienda.name
```

- **Conta il numero di colleghi di Mario Rossi.**

```
MATCH (mario: Persona {name: "Mario Rossi"})-[:COLLEGA_DI]-(collega: Persona)
RETURN COUNT(collega) AS Num_Colleghi
```

- **Trova i nomi di tutte le persone coinvolte nel progetto Alpha.**

```
MATCH (persona: Persona)-[:PARTECIPA_A]->(progetto:
    Progetto {name: "Alpha"})
RETURN persona.name
```

- **Trova tutte le aziende per cui lavora un collega di Elena Ferrari.**

```
MATCH (elena: Persona {name: "Elena Ferrari"})-[:
    COLLEGA_DI]-(collega: Persona)-[:LAVORA_PER]->(azienda
    : Azienda)
RETURN DISTINCT azienda.name
```

- **Trova i nomi delle persone che lavorano per aziende diverse da quelle dei loro amici.**

```
MATCH (persona: Persona)-[:LAVORA_PER]->(azienda: Azienda)
MATCH (azienda2:Azienda)<-[:LAVORA_PER]-(amici: Persona)
    -[:AMICO_DI]-(persona)
WHERE azienda <> azienda2
RETURN persona.name, amici.name, azienda2.name, azienda.
    name
```

- **Trova tutte le persone che non hanno amici.**

```
MATCH (persona:Persona)
WHERE NOT (persona)-[:AMICO_DI]-()
RETURN persona.name
```

Altri esempi

```
MATCH (a:Azienda)<-[:LAVORA]-(p1:Dipendente)-[:AMICO]->(p2:
    Dipendente)-[:LAVORA]->(a)
RETURN p1,p2
```

Listing 9: Query Cypher per trovare coppie di amici che lavorano per la stessa azienda.

```
MATCH (p1:Person)-[:HAS_HOBBY]->(h:Hobby)<-[:HAS_HOBBY]-(p2:Person)
WHERE p1 <> p2
RETURN DISTINCT p1.name, p2.name
```

Listing 10: Query Cypher per trovare trovare tutte le persone che hanno un hobby in comune con almeno un'altra persona.

```
MATCH (dip1:Dipendente)-[:LAVORA]->(a:Azienda)<-[:LAVORA]-(dip2:
    Dipendente)
MATCH (dip1)<-[:AMICO_DI]->(dip2)
WHERE dip1 <> dip2 AND id(dip1) < id(dip2) // Per evitare di
    contare la stessa coppia due volte
WITH a, collect({dipendente1: dip1.name, dipendente2: dip2.name})
    AS coppie
WHERE size(coppie) >= 2
```

```
RETURN a.name AS NomeAzienda, size(coppie) AS NumeroCoppieAmici,
    coppie
```

Listing 11: Query Cypher per trovare tutte le aziende che hanno almeno due dipendenti che sono amici tra loro.

```
MATCH (p1:Persona {name: "Alice"})-[:FRIEND]-(p2:Persona)
WHERE p2.age > 30
RETURN p2 AS Amici
```

Listing 12: Query Cypher per trovare persone sopra i 30 anni amiche di "Alice".

```
MATCH (f:Film)-[:DIRECTED_BY]->(p1:Persona {name: "Christopher␣
    Nolan"})
WHERE f.year > 2010
RETURN f.title AS Titolo, f.year AS Anno_di_uscita
```

Listing 13: Query Cypher per trovare film diretti da "Christopher Nolan" usciti dopo il 2010.

```
MATCH (p1:Persona {position: "CEO"})-[:MANAGES]->(p2:Persona)
WHERE p2.salary > 100000
RETURN p2.name AS Nome, p2.position AS Posizione_Lavorativa, p2.
    salary AS Salario
```

Listing 14: Query Cypher per trovare dipendenti gestiti da un CEO con stipendio superiore a 100.000.

```
MATCH (u:University)-[:OFFERS]->(c:Course)
WHERE c.credits >= 6
WITH u.universityName AS universityName, COUNT(c) AS Corsi
ORDER BY Corsi DESC
LIMIT 1
RETURN universityName AS Nome_Uni, Corsi
```

Listing 15: Query Cypher per trovare l'università che offre il maggior numero di corsi con almeno 6 crediti.

6 Database RDF

6.1 SPARQL

Esercizio 1

Immagina di dover rappresentare un sistema di gestione delle ricette in RDF. Il sistema deve contenere informazioni su ricette, ingredienti, e chef. Dati di esempio:

- La ricetta "Carbonara" è stata creata dallo chef "Gordon Ramsay". Gli ingredienti necessari per questa ricetta sono "Pasta", "Uova", "Pancetta" e "Parmigiano".

- La ricetta “Tiramisu” è stata creata dallo chef “Massimo Bottura”. Gli ingredienti necessari per questa ricetta sono “Mascarpone”, “Caffè”, “Savoiardi” e “Cacao”.
- La ricetta “Pesto alla Genovese” è stata creata dallo chef “Carlo Cracco”. Gli ingredienti necessari per questa ricetta sono “Basilico”, “Aglio”, “Pinoli” e “Parmigiano”.

a) **Rappresenta i dati di esempio in RDF.**

```
@prefix ex: <http://example.org/recipes#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dct: <http://purl.org/dc/terms/> .

# Chef
ex:GordonRamsay a foaf:Person ;
    foaf:name "Gordon␣Ramsay" .
ex:MassimoBottura a foaf:Person ;
    foaf:name "Massimo␣Bottura" .
ex:CarloCracco a foaf:Person ;
    foaf:name "Carlo␣Cracco" .

# Ingredienti
ex:Pasta a ex:Ingrediente ;
    dct:title "Pasta" .
ex:Uova a ex:Ingrediente ;
    dct:title "Uova" .
ex:Pancetta a ex:Ingrediente ;
    dct:title "Pancetta" .
ex:Parmigiano a ex:Ingrediente ;
    dct:title "Parmigiano" .
ex:Mascarpone a ex:Ingrediente ;
    dct:title "Mascarpone" .
ex:Caffe a ex:Ingrediente ;
    dct:title "Caffe" .
ex:Savoiardi a ex:Ingrediente ;
    dct:title "Savoiardi" .
ex:Cacao a ex:Ingrediente ;
    dct:title "Cacao" .
ex:Basilico a ex:Ingrediente ;
    dct:title "Basilico" .
ex:Aglio a ex:Ingrediente ;
    dct:title "Aglio" .
ex:Pinoli a ex:Ingrediente ;
    dct:title "Pinoli" .

# Ricette
ex:Carbonara a ex:Ricetta ;
    dct:title "Carbonara" ;
    dct:creator ex:GordonRamsay ;
    ex:haIngrediente ex:Pasta, ex:Uova, ex:Pancetta, ex:
        Parmigiano .
ex:Tiramisu a ex:Ricetta ;
    dct:title "Tiramisu" ;
    dct:creator ex:MassimoBottura ;
    ex:haIngrediente ex:Mascarpone, ex:Caffe, ex:Savoiardi, ex
        :Cacao .
```

```
ex:PestoAllaGenovese a ex:Ricetta ;
    dct:title "Pesto alla Genovese" ;
    dct:creator ex:CarloCracco ;
    ex:haIngrediente ex:Basilico, ex:Aglio, ex:Pinoli, ex:
        Parmigiano .
```

b) **Formula le seguenti query SPARQL.**

- Q1: Data una ricetta, trovare tutti gli ingredienti necessari.

```
PREFIX ex: <http://example.org/recipes#>
PREFIX dct: <http://purl.org/dc/terms/>

SELECT ?ingredientName
WHERE {
  ex:Carbonara ex:haIngrediente ?ingredient .
  ?ingredient dct:title ?ingredientName .
}
```

- Q2: Dato uno chef, trovare tutte le ricette che ha creato.

```
PREFIX ex: <http://example.org/recipes#>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?recipeName
WHERE {
  ?recipe dct:creator ex:GordonRamsay ;
          dct:title ?recipeName .
}
```

- Q3: Dato un ingrediente, trovare tutte le ricette che lo utilizzano.

```
PREFIX ex: <http://example.org/recipes#>
PREFIX dct: <http://purl.org/dc/terms/>

SELECT ?recipeName
WHERE {
  ?recipe ex:haIngrediente ex:Parmigiano ;
          dct:title ?recipeName .
}
```

- Q4: Trovare tutte le ricette che non sono state create da un determinato chef.

```
PREFIX ex: <http://example.org/recipes#>
PREFIX dct: <http://purl.org/dc/terms/>

SELECT ?recipeName
WHERE {
  ?recipe dct:title ?recipeName .
  FILTER NOT EXISTS {
    ?recipe dct:creat ex:CarloCracco .
  }
}
```

- Q5: Trovare tutti gli ingredienti utilizzati in almeno due ricette diverse.

```
PREFIX ex: <http://example.org/recipes#>
PREFIX dct: <http://purl.org/dc/terms/>

SELECT ?ingredientName (COUNT(?recipe) AS ?recipeCount)
WHERE {
  ?recipe ex:haIngrediente ?ingredient .
  ?ingredient dct:title ?ingredientName .
}
GROUP BY ?ingredientName
HAVING (COUNT(?recipe) >= 2)
```

E' possibile testare l'esercizio con il seguente tool online: SPARQL-Playground

Esercizio 2

Un sistema di gestione bibliotecaria deve contenere informazioni sui libri, gli autori, e le categorie di ogni libro. Di ogni libro sono presenti il titolo, l'anno di pubblicazione, e l'ISBN. Ogni autore è descritto da un nome e una data di nascita, mentre ogni categoria è identificata da un nome.

- Il libro "Il nome della rosa" è stato pubblicato nel 1980, ha l'ISBN "978-8830423863", ed è scritto da Umberto Eco, nato il 5 gennaio 1932. Questo libro appartiene alle categorie "Romanzo" e "Giallo".
- Il libro "1984" è stato pubblicato nel 1949, ha l'ISBN "978-0451524935", ed è scritto da George Orwell, nato il 25 giugno 1903. Questo libro appartiene alle categorie "Romanzo" e "Distopia".
- Il libro "Orgoglio e pregiudizio" è stato pubblicato nel 1813, ha l'ISBN "978-0141040349", ed è scritto da Jane Austen, nata il 16 dicembre 1775. Questo libro appartiene alla categoria "Romanzo".

a) **Rappresenta i dati di esempio in RDF.**

```
@prefix ex: <http://example.org/library#> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

# Autori
ex:UmbertoEco a foaf:Person ;
    foaf:name "Umberto Eco" ;
    foaf:birthDate "1932-01-05"^^xsd:date .
ex:GeorgeOrwell a foaf:Person ;
    foaf:name "George Orwell" ;
    foaf:birthDate "1903-06-25"^^xsd:date .
ex:JaneAusten a foaf:Person ;
    foaf:name "Jane Austen" ;
    foaf:birthDate "1775-12-16"^^xsd:date .

# Categorie
```

```
ex:Romanzo a dct:Subject ;
    dct:title "Romanzo" .
ex:Giallo a dct:Subject ;
    dct:title "Giallo" .
ex:Distopia a dct:Subject ;
    dct:title "Distopia" .

# Libri
ex:IlNomeDellaRosa a dct:BibliographicResource ;
    dct:title "Il nome della rosa" ;
    dct:creator ex:UmbertoEco ;
    dct:date "1980"^^xsd:gYear ;
    dct:identifier "978-8830423863" ;
    dct:subject ex:Romanzo, ex:Giallo .
ex:1984 a dct:BibliographicResource ;
    dct:title "1984" ;
    dct:creator ex:GeorgeOrwell ;
    dct:date "1949"^^xsd:gYear ;
    dct:identifier "978-0451524935" ;
    dct:subject ex:Romanzo, ex:Distopia .
ex:OrgoglioEPregiudizio a dct:BibliographicResource ;
    dct:title "Orgoglio e pregiudizio" ;
    dct:creator ex:JaneAusten ;
    dct:date "1813"^^xsd:gYear ;
    dct:identifier "978-0141040349" ;
    dct:subject ex:Romanzo .
```

b) **Formula le seguenti query SPARQL.**

- Q1: Dato un ISBN, trovare il titolo del libro e l'autore.

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/library#>

SELECT ?title ?authorName
WHERE {
  ?book dct:identifier "978-8830423863" ;
        dct:title ?title ;
        dct:creator ?author .
  ?author foaf:name ?authorName .
}
```

- Q2: Data una categoria, trovare tutti i libri che appartengono a quella categoria.

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX ex: <http://example.org/library#>

SELECT ?title
WHERE {
  ?book dct:subject ex:Romanzo ;
        dct:title ?title .
}
```

- Q3: Dato un autore, trovare tutti i libri che ha scritto.

```
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/library#>

SELECT ?title
WHERE {
  ?book dct:creator ex:UmbertoEco ;
        dct:title ?title .
}
```

E' possibile testare l'esercizio con il seguente tool online: SPARQL-Playground

Altri esempi

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?article ?title ?date
WHERE {
    ?article dc:creator "John␣Smith" .
    ?article dc:title ?title .
    ?article dc:date ?date .
    FILTER (?date > "2015-01-01"^^xsd:date)
}
```

```
PREFIX ex: <http://example.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?eventType ?category ?categoryTitle (COUNT(?event) AS ?
    eventCount)
WHERE {
  ?event ex:eventType ?eventType .
  ?event ex:category ?category .
  ?event ex:eventTitle ?eventTitle .
  ?event ex:eventDate ?eventDate .
  ?category ex:categoryTitle ?categoryTitle .
  FILTER (?eventDate > "2023-01-01"^^xsd:date)
}
GROUP BY ?eventType ?category ?categoryTitle
ORDER BY ?eventType DESC ?eventCount DESC
```

Listing 16: Query SPARQL per trovare la categoria con il maggior numero di eventi dopo il 1 gennaio 2023.

7 Database Logici

7.1 Datalog

Nota: In Datalog standard, possiamo definire solo condizioni locali e non possiamo tracciare uno stato globale dei nodi visitati. Ciò significa che non è possibile mantenere un registro completo dei nodi visitati lungo un cammino.

Datalog è un linguaggio dichiarativo e non offre meccanismi per mantenere uno stato globale o una memoria delle visite ai nodi. Non esistono costrutti in Datalog che permettano di verificare a livello globale se un nodo è stato visitato più di una volta durante la computazione di un percorso.

```
reachable_member(X) :- manager_of(john, X).
reachable_member(X) :- manager_of(Y, X), reachable_member(Y).
```

Listing 17: Query Datalog per calcolare tutti i membri della gerarchia raggiungibili partendo da John.

```
reachable(File) :- link(file1, File).
reachable(File) :- link(file1, Intermediate), reachable(
    Intermediate), link(Intermediate, File).
```

Listing 18: Query Datalog per calcolare tutti i file raggiungibili da file1 attraverso collegamenti.

```
reachable_within_3_flights(X) :- flight(new_york, X).
reachable_within_3_flights(X) :- flight(new_york, Y), flight(Y, X).
reachable_within_3_flights(X) :- flight(new_york, Y), flight(Y, Z),
     flight(Z, X).
reachable_within_3_flights(X) :- flight(new_york, Y), flight(Y, Z),
     flight(Z, W), flight(W, X).
```

Listing 19: Query Datalog per calcolare tutte le città raggiungibili da New York con al massimo 3 voli.

```
friend_of_friend(X) :- friend(alice, Y), friend(Y, X).
```

Listing 20: Query Datalog per calcolare tutti i nodi amici di amici di Alice.

```
reachable(a, 0).
reachable(Y, Cost) :- road(a, Y, Cost), Cost =< 10.
reachable(Y, TotalCost) :- road(X, Y, Cost), reachable(X, AccCost),
     TotalCost = AccCost + Cost, TotalCost =< 10.
```

Listing 21: Query Datalog per calcolare tutte le città raggiungibili dalla città A con un costo totale massimo di 10.

```
reachable(rome).
reachable(Y) :- flight(rome, Y).
reachable(Y) :- flight(X, Y), reachable(X).
```

Listing 22: Query Datalog per calcolare tutte le destinazioni raggiungibili da Roma.

```
supervised_by_alice(bob).
supervised_by_alice(X) :- supervises(alice, X).
supervised_by_alice(X) :- supervises(Y, X), supervised_by_alice(Y).
```

Listing 23: Query Datalog per calcolare tutti i dipendenti (diretti o indiretti) supervisionati da Alice.

```
descendant(john).
descendant(Y) :- parent(john, Y).
descendant(Y) :- parent(X, Y), descendant(X).
```

Listing 24: Query Datalog per calcolare tutti i discendenti di John.

```
p(a).
p(Y) :- e(X,Z), e(Z,Y), p(X).
```

Listing 25: Query Datalog per calcolare tutti i nodi che sono raggiungibili, attraverso cammini che includono un numero pari di archi, dal nodo a.

```
p(a).
p(Y) :- e(X,Z), e(Z,Y), p(X).
```

Listing 26: Query Datalog per calcolare tutti i nodi che sono raggiungibili, attraverso cammini che includono un numero pari di archi, dal nodo a.

8 Esame del 26/06/24

8.1 Esercizio 1: DB Distribuiti e Consistenza (6 punti)

a) **Nel contesto di DB distribuiti, spiegare brevemente vantaggi e svantaggi della replicazione Master-Slave rispetto alla replicazione Peer-to-Peer.**

La replicazione *Master-Slave* è un tipo di replica dove un nodo principale chiamato Master, riceve tutte le scritture e le propaga ai nodi secondari chiamati Slave. I nodi Slave sono letti solo per aumentare la disponibilità. I vantaggi della replicazione Master-Slave sono:

- *Semplicità*: La replicazione Master-Slave è più semplice da implementare in quanto esiste un singolo punto di scrittura (il Master), che facilita la gestione delle scritture.
- *Coerenza dei Dati*: Essendo le scritture eseguite solo sul Master, si evita la problematica di conflitti di scrittura che possono verificarsi in ambienti Peer-to-Peer.
- *Performance*: Le letture possono essere distribuite sui nodi Slave, migliorando le performance di lettura grazie alla riduzione del carico sul Master.

Gli svantaggi della replicazione Master-Slave sono:

- *Single Point of Failure*: Il Master rappresenta un singolo punto di fallimento. Se il Master va offline, le operazioni di scrittura non possono essere eseguite fino a quando non viene ristabilito o non viene promosso un nuovo Master.

- *Latency*: Le scritture devono essere propagate agli Slave, introducendo ritardi. Inoltre gli Slave potrebbero non essere aggiornati in tempo reale con i dati del Master.
- *Scalabilità limitata*: Poiché solo il Master gestisce le scritture, la scalabilità delle operazioni di scrittura è limitata dalla capacità del Master.

La replicazione *Peer-to-Peer* è un tipo di replica dove ogni nodo del cluster può ricevere sia letture che scritture, propagando gli aggiornamenti agli altri nodi.
I vantaggi della replicazione Peer-to-Peer sono:

- *No Single Point of Failure*: Non esiste un singolo punto di fallimento. Ogni nodo può gestire sia letture che scritture, aumentando la disponibilità del sistema.
- *Scalabilità delle Scritture*: Poiché ogni nodo può gestire le scritture, il sistema può scalare orizzontalmente molto più facilmente rispetto a un ambiente Master-Slave.
- *Bilanciamento del Carico*: Il carico può essere distribuito uniformemente tra tutti i nodi, migliorando l'efficienza complessiva del sistema.

Gli svantaggi della replicazione Peer-to-Peer sono:

- *Gestione dei Conflitti*: Con più nodi che possono eseguire scritture, esiste una maggiore probabilità di conflitti di scrittura, il che richiede meccanismi complessi per la risoluzione dei conflitti.
- *Complessità di Implementazione*: La gestione della coerenza dei dati e la risoluzione dei conflitti rendono la replicazione Peer-to-Peer più complessa da implementare e mantenere.
- *Performance di Sincronizzazione*: Sincronizzare continuamente i dati tra più nodi può introdurre latenza e influenzare negativamente le performance del sistema

b) **Si consideri un client che esegue una scrittura sul nodo Master, replicata (propagata) in modo asincrono verso le repliche. Spiegare cosa è la read-your-writes consistency, e perché la replicazione asincrona non la garantisce. Illustrare come la replicazione sincrona (synchronous replication) costituisce una tecnica che garantisce detta consistenza read-your-writes; spiegare brevemente i costi e i problemi di questa tecnica.**

La *Read-Your-Writes Consistency* garantisce che un client, dopo aver eseguito una scrittura su un database, sia in grado di leggere immediatamente i dati appena scritti. In altre parole, un client vedrà sempre le sue ultime scritture durante le operazioni di lettura successive. Garantisce dunque

che una volta che una scrittura è stata completata, ogni lettura successiva vedrà almeno quella scrittura.

Nella *replicazione asincrona*, il Master esegue la scrittura e poi propaga (replica) l'aggiornamento ai nodi Slave in modo asincrono, cioè senza attendere che tutti i nodi Slave abbiano confermato la ricezione dei dati. Questo può comportare un ritardo nella visibilità delle scritture sui nodi Slave. Di conseguenza, un client potrebbe non vedere immediatamente le sue scritture se legge da uno Slave. La scrittura è considerata completata una volta che è stata effettuata su un nodo primario, e le repliche vengono aggiornate in seguito. Questo aumenta la disponibilità e riduce la latenza, poichè l'operazione può restituire una risposta immediatamente senza attendere l'aggiornamento delle repliche. Tuttavia, può portare a inconsistenze temporanee, poichè le repliche possono contenere dati obsoleti fino a quando non vengono sincronizzate.

La *replicazione sincrona*, al contrario, implica che il Master attenda fino a quando tutte le repliche (Slave) non abbiano confermato la ricezione e l'applicazione dell'aggiornamento prima di considerare la scrittura completata. Questo garantisce che tutti i nodi siano aggiornati simultaneamente, fornendo la consistenza Read-Your-Writes, ma può aumentare la latenza, poichè l'operazione deve attendere che tutte le repliche siano aggiornate prima di restituire una risposta.

8.2 Esercizio 2: DB Documentali (6 punti)

Un sistema informativo deve contenere dati relativi a giocatori di calcio e, per ciascun giocatore, alla squadra in cui egli gioca attualmente. Di ogni giocatore si rappresentano nome, anno di nascita e altezza; di ogni squadra si rappresentano nome, anno di fondazione e città (alcuni di questi dati possono essere mancanti, e si possono specificare dati aggiuntivi, vista l'assenza di schema). Si vuole rappresentare che il giocatore Mathías Olivera (nato nel 1997, altezza 1,84, di ruolo terzino) gioca nella SSC Napoli; il giocatore Lucas Ocampos (nato nel 1994, altezza 1,88) gioca nel Sevilla FC. La SSC Napoli, fondata nel 1926, è una società sportiva di Napoli; Il Sevilla FC, fondato nel 1890, è di Siviglia. Si stima che la base di dati dovrà contenere dati come quelli di cui sopra, con circa 300.000 giocatori e 10.000 squadre. Si prevede inoltre che saranno eseguite le sequenti interrogazioni con relative frequenze stimate:
Q1: Dato una squadra s e un anno a, trovare tutti i giocatori nati in un anno uguale o precedente ad a che giocano in s. (10.000 interrogazioni al giorno)
Q2: Data una città c, trovare tutti i giocatori di squadre di c. (15.000 interrogazioni al giorno)
Q3: Dato un giocatore con nome g, trovare la città della squadra in cui egli gioca. (5 interrogazioni al giorno)

a) **Si specifichi se la relazione tra Giocatore e Squadra (che sareb-**

bero entità se le rappresentassimo con uno schema entità-relazione) è uno-a-uno, uno-a-molti o molti-a-molti.

La relazione tra Giocatore e Squadra è di tipo "molti-a-uno" (many-to-one), poiché molti giocatori possono appartenere a una sola squadra

b) **Si illustri come memorizzare i dati di esempio, rappresentandoli in JSON2, considerando le loro frequenza. Si spieghi brevemente l'uso di eventuali denormalizzazioni.**

```
{
  "squadre": [
        {
        "squadraId": 1,
        "nome": "SSC Napoli",
        "annoFondazione": 1926 ,
        "citta": "Napoli",
        "giocatori": [
          {
            "giocatoreId": 1,
            "nome": "Mathias Olivera",
            "annoNascita": 1997,
            "altezza": 1.84
          }
        ]
      },
      {
        "squadraId": 2,
        "nome": "Sevilla FC",
        "annoFondazione": 1890 ,
        "citta": "Siviglia",
        "giocatori": [
          {
            "giocatoreId": 2,
            "nome": "Lucas Ocampos",
            "annoNascita": 1994,
            "altezza": 1.88
          }
        ]
      }
  ]
}
```

La denormalizzazione può essere utile per evitare join complessi, migliorando le performance di lettura, soprattutto per query frequenti come *Q1* e *Q2*, che richiedono dati correlati. La cosa importante è avere la lista dei giocatori in ogni documento di squadra, con i dati necessarî per rispondere alle query Q1, Q2. Ciò consente di eseguire le query piú frequenti (Q1, Q2) senza andare a consultare la collezione dei giocatori.

c) **Si formulino le query in MongoDB Query Language (con valore a piacere)**

Query Q1

- **Soluzione 1**:

```
db.squadre.aggregate([
  // Filtra per squadraId
  { $match: { "squadraId": 1, "annoFondazione": 1926 } },
  // Espande array giocatori
  { $unwind: "$giocatori" },
  // Filtra i giocatori per anno di nascita
  { $match: { "giocatori.annoNascita": { $lte: 1926 } } },
  // Proietta solo i campi desiderati
  { $project: { _id: 0, "giocatori": 1} }
])
```

- **Soluzione 2**:

```
db.squadre.aggregate([
    {
        $project: {
            giocatori: {
                $filter: {
                    input: "$giocatori",
                    as: "giocatori",
                    cond: { $lte: ["$giocatori.annoNascita
                        ", 1926] }
                }
            }
        }
    }
])
```

L'utilizzo di $elemMatch in questa query non è consigliato siccome se ci fossero più giocatori che soddisfano la condizione, $elemMatch ne restituirebbe solo uno. Per ottenere tutti i giocatori che soddisfano la condizione, si utilizza l'aggregazione.

Query Q2

```
db.squadre.find(
  {"citta": "Napoli"},
  {"giocatori": 1, _id: 0}
)
```

Query Q3

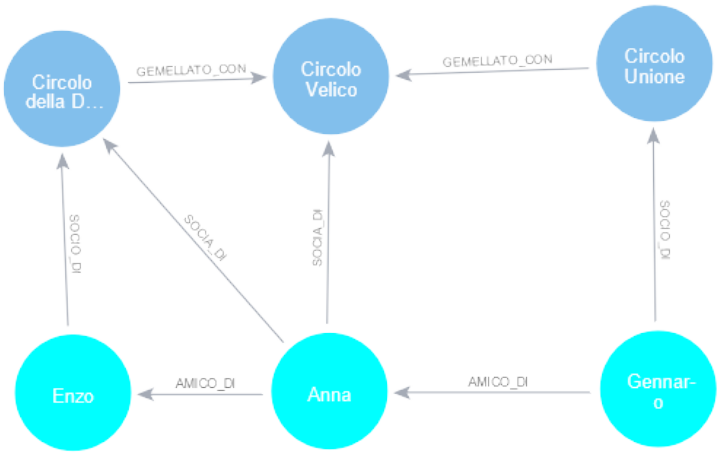
```
db.squadre.find(
  {"giocatori": {$elemMatch: {"nome": "Mathias␣Olivera"}}},
  {"citta": 1, _id: 0}
)
```

E' possibile testare l'esercizio con il seguente compilatore online: OneCompiler

8.3 Esercizio 3: Graph DBs (7 punti)

Si vogliono rappresentare circoli sociali e loro soci. Gennaro è socio del Circolo Unione dal 1995; Anna è socia del Circolo Velico dal 1988 e del Circolo della Dama dal 2001; Enzo è socio del Circolo della Dama dal 1999. Gennaro e Anna sono amici, e sono anche amici Anna ed Enzo. Il Circolo Unione è gemellato col Circolo Velico; il Circolo della Dama è anche gemellato col Circolo Velico (la relazione di gemellaggio è evidentemente simmetrica ma non transitiva).

a) **Rappresentare i dati sopra descritti in un property graph.**




```
// Creazione dei nodi
CREATE (gennaro:Persona {nome: 'Gennaro'})
CREATE (anna:Persona {nome: 'Anna'})
CREATE (enzo:Persona {nome: 'Enzo'})
CREATE (circolo_unione:Circolo {nome: 'Circolo Unione'})
CREATE (circolo_velico:Circolo {nome: 'Circolo Velico'})
CREATE (circolo_dama:Circolo {nome: 'Circolo della Dama'})

// Creazione delle relazioni di appartenenza
CREATE (gennaro)-[:SOCIO_DI {dal: 1995}]->(circolo_unione)
CREATE (anna)-[:SOCIA_DI {dal: 1988}]->(circolo_velico)
CREATE (anna)-[:SOCIA_DI {dal: 2001}]->(circolo_dama)
CREATE (enzo)-[:SOCIO_DI {dal: 1999}]->(circolo_dama)

// Creazione delle relazioni di amicizia
CREATE (gennaro)-[:AMICO_DI]->(anna)
CREATE (anna)-[:AMICO_DI]->(enzo)

// Creazione delle relazioni di gemellaggio
CREATE (circolo_unione)-[:GEMELLATO_CON]->(circolo_velico)
CREATE (circolo_dama)-[:GEMELLATO_CON]->(circolo_velico)
```

Listing 27: Codice per la creazione del grafo in Neo4j

b) **Scrivere in Cypher, assumendo opportune etichette (label, relationship type e property) una query che trova coppie di amici che sono soci dello stesso circolo**

```
MATCH (p1:Persona)-[:AMICO_DI]-(p2:Persona)
MATCH (p1)-[:SOCIO_DI]->(c:Circolo)<-[:SOCIO_DI]-(p2)
RETURN p1.nome AS Amico1, p2.nome AS Amico2, c.nome AS Circolo
```

c) **Scrivere in Cypher una query che conta quanti soci ha ciascun circolo che ha come socio, da prima del 1990, un amico (o un'amica) di Gennaro**

```
MATCH (gennaro:Persona {nome: "Gennaro"})-[:AMICO_DI]-(amico:
    Persona)
MATCH (amico)-[socio:SOCIO_DI]->(circolo:Circolo)
WHERE socio.dal < 1990
WITH circolo
MATCH (persona)-[socio:SOCIO_DI]->(circolo)
RETURN circolo.nome AS Circolo, COUNT(DISTINCT persona) AS
    NumeroDiSoci
```

8.4 Esercizio 4: RDF (5 punti)

Si vogliono rappresentare dei dati relativi ad un'azienda di giocattoli e ai suoi progetti interni. Francesco lavora al progetto Bambole come disegnatore; Pasquale lavora al progetto Trenini come assemblatore, e Vincenzo lavora al progetto Trenini come collaudatore.

a) **Si illustri (in modo conciso) l'utilità degli IRIS in RDF.**

Gli IRIs (Internationalized Resource Identifiers) sono degli identificatori internazionalizzati delle risorse, usati per identificare univocamente risorse in RDF (Resource Description Framework). Gli IRIs vengono utilizzati per identificare in modo univoco le risorse nel web semantico. Gli IRIs estendono gli URI (Uniform Resource Identifier) per consentire l'uso di caratteri internazionali. Le loro utilità includono:

- Unicità Globale: Garantiscono che ogni risorsa abbia un identificatore unico su scala globale.
- Interoperabilità: Facilitano l'integrazione e l'interoperabilità tra diversi dataset e sistemi.
- Risoluzione: Possono essere utilizzati per recuperare rappresentazioni delle risorse che identificano, rendendo i dati più accessibili e collegati

b) **Si spieghi brevemente perché i grafi (database) RDF sono in realtà ipergrafi con archi ternari**

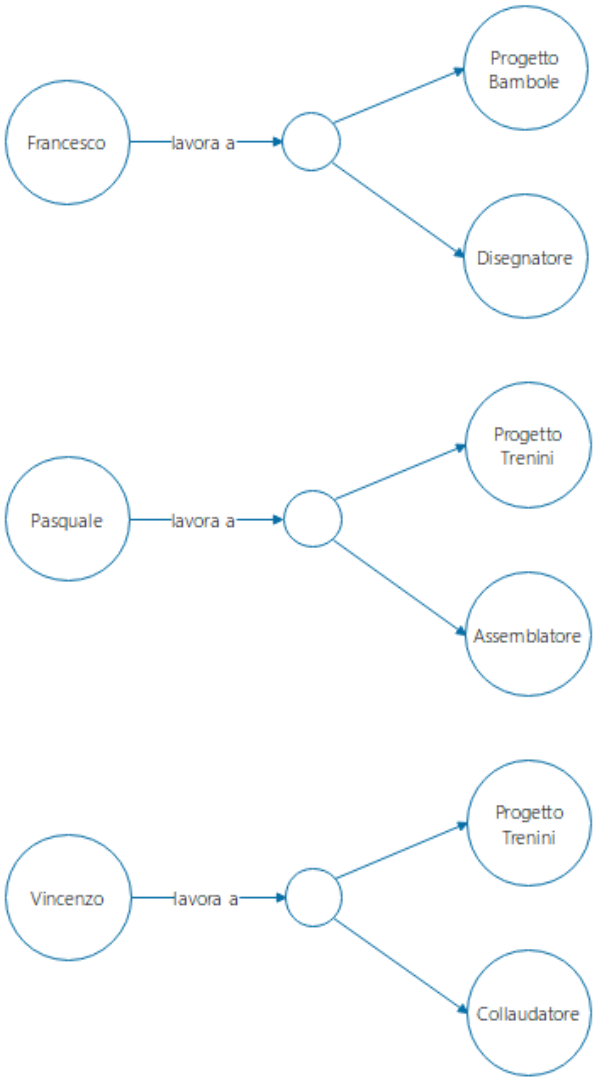
I grafi RDF sono rappresentati come ipergrafi con archi ternari perché

quest'ultimi rappresentano una struttura dati composta da triple (soggetto, predicato, oggetto) che rappresentano informazioni un'asserzione RDF. Ogni tripla rappresenta una relazione diretta tra tre componenti:

- Soggetto: La risorsa di cui si parla.
- Predicato: Il tipo di relazione.
- Oggetto: La risorsa o il valore che è collegato al soggetto attraverso il predicato.

Dunque, gli RDF possono essere visti come ipergrafi dove ogni tripla (soggetto, predicato, oggetto) rappresenta un arco ternario, collegando tre nodi

c) **Si rappresentino i dati di cui sopra in RDF (fornire una rappresentazione grafica)**



8.5 Esercizio 5: Datalog (6 punti)

Sia dato un grafo rappresentato con un predicato binario e; si veda per esempio il database D come segue:

```
1 e(a,b).
2 e(b,c).
```

```
3 e(c,b).
4 e(c,a).
```

Listing 28: Fatti

a) **Formulare una query Datalog (programma P più predicato per la risposta) che calcoli tutti i nodi che sono raggiungibili, attraverso cammini che includono un numero pari di archi, dal nodo a (ovviamente la query deve essere corretta per tutti i database). I cammini in questione possono passare più di una volta sullo stesso nodo.**

```
1 p(a).
2 p(Y) :- e(X,Z), e(Z,Y), p(X).
```

b) **Fornire la risposta alla query (quindi un insieme di valori compresi in {a, b, c} che rappresentano nodi) in questione ove essa fosse eseguita su D.**.

I nodi raggiungibili con un numero pari di archi sono:

```
1 {a, c}
2 {b, c}
3 {c, c}
```

c) **Si assuma ora di voler formulare una query simile a quella al punto (a), ma richiedendo che i cammini passino al massimo una volta per ogni nodo. È possibile formulare una tale query? In caso di risposta affermativa, formulare la query in Datalog: altrimenti, spiegare brevemente perché la query non è formulabile in Datalog.**

Per formulare una query che richieda che i cammini passino al massimo una volta per ogni nodo, dovremmo introdurre un meccanismo per tracciare i nodi già visitati. Tuttavia, Datalog classico non supporta questa funzionalità direttamente. Quindi, non è possibile formulare tale query in Datalog standard perché Datalog non supporta controlli di unicità nei cammini. Per una gestione efficiente di tali query, sarebbe più appropriato utilizzare un database grafico con supporto per queste operazioni.

9 Esame del 26/07/24

9.1 Esercizio 1: DB Distribuiti e Consistenza (6 punti)

a) **Nel contesto del CAP Theorem, si definisca brevemente il concetto di linearizzabilità (linearizability) e di CAP-availability**.

La *linearizzabilità* una proprietà di consistenza nei sistemi distribuiti che garantisce che tutte le operazioni appaiono come se fossero eseguite in un ordine sequenziale e che ogni operazione prende effetto in un punto tra il suo inizio e la sua fine. In altre parole, le operazioni appaiono atomiche. In altre parole, se un'operazione B inizia dopo che un'operazione A è stata completata, l'operazione B deve vedere il sistema nello stesso stato in cui si trovava alla fine dell'operazione A, o in uno stato più recente.
La *disponibilità* è una delle proprietà del teorema CAP e afferma che in un sistema distribuito ogni richiesta fatta a un nodo funzionante riceve una risposta, anche se non può garantire che la risposta contenga l'ultimo dato scritto. La disponibilità implica che il sistema deve essere progettato per gestire i guasti in modo da garantire che le operazioni possano continuare. Ad esempio, nei database distribuiti, la disponibilità può essere garantita replicando i dati su più nodi. In questo modo, se uno o più nodi diventano inaccessibili, il sistema può ancora rispondere alle richieste utilizzando le repliche disponibili.

b) **Si consideri una situazione come quella in Figura 1 (pag. 2), dove c'è una interruzione della comunicazione (network partition) tra i due nodi illustrati. Si supponga che in presenza di tale interruzione uno dei DBMS sceglie di evitare entrambi i nodi che continuano a funzionare, con le scritture che ovviamente non si propagano. Stabilire se come conseguenza di questa scelta valgono o meno linearizzabilità e CAP-availability. Giustificare brevemente la risposta fornita**

In presenza di un'interruzione di rete, se entrambi i nodi continuano a funzionare ma le scritture non si propagano, allora la *linearizzabilità* non è garantita perché le operazioni potrebbero non riflettere un ordine globale coerente a causa della mancanza di propagazione delle scritture a differenza della *disponibilità* che potrebbe essere mantenuta se i nodi rispondono comunque alle richieste di lettura e scrittura, ma le risposte potrebbero non essere coerenti con la linearizzabilità.

c) **Mostrare almeno un caso di DBMS in cui in pratica non c'è né linearizzabilità né CAP-availability.**

Un esempio di DBMS in cui non c'è né linearizzabilità né CAP-availability è un sistema distribuito che utilizza la replica asincrona senza coordinazione tra le repliche ossia *Cassandra*. In questo caso, durante una partizione di rete, le scritture possono essere accettate solo da alcune repliche e non propagate immediatamente alle altre, violando così la linearizzabilità. Se durante la partizione alcune repliche non rispondono, anche la disponibilità può essere compromessa.
In Cassandra, se si richiede che ogni scrittura e lettura debbano essere confermate da tutti i nodi, si introduce una dipendenza critica dalla disponibilità di tutti i nodi del cluster. Se uno o più nodi diventano inaccessibili,

né le scritture né le letture possono essere completate, violando la disponibilità (CAP-Availability). Inoltre, se una partizione di rete isola alcuni nodi, il sistema non può garantire la linearizzabilità poiché non tutte le repliche sono in grado di sincronizzarsi immediatamente, causando potenziali letture inconsistenti fino al ripristino della comunicazione.

9.2 Esercizio 2: DB Documentali (6 punti)

Un sistema informativo deve contenere dati relativi a clienti di un'azienda che vende prodotti e reclami da essi presentati"; ogni reclamo è presentato da un unico cliente; ogni cliente ovviamente può presentare piú reclami. II cliente Francesco Scannapieco abita a Napoli e ha numero di telefono 081-2292929; il cliente Gaetano Scandurra ha 33 anni, abita a San Gennaro Vesuviano e ha indirizzo di posta elettronica gsca@scandurra.com. Il cliente Francesco Scannapieco ha presentato un reclamo il 27-08-2023, per il prodotto con codice identificativo P123 con motivo "Pacco mai arrivato", e un altro il 21-09-2023 con motivo "Numero clienti risulta occupato". Il cliente Gaetano Scandurra ha presentato un reclamo il 12-12-2022, per il prodotto P321, con motivo "Prodotto difettoso". Si stima che la base di dati dovrà contenere dati come quelli di cui sopra, con circa 100.000 clienti e 20.000 reclami. Si prevede inoltre che saranno eseguite le sequenti operazioni con relative frequenze stimate:
Q1: Dato un identificativo di cliente c, trovare tutti gli identificativi dei prodotti per i quali il cliente identificato da c ha presentato un reclamo. (3.000 interrogazioni al giorno)
Q2: Dato l'identificativo r di un reclamo, trovare il motivo del reclamo in questione. (5.000 interrogazioni al giorno)
I3: Inserimento di un reclamo con ingresso data, cliente

a) **Si illustri come memorizzare i dati di esempio, rappresentandoli in JSON, considerando le operazioni e la loro frequenza. Si spieghi brevemente le eventuali denormalizzazioni.**

```
{
    "clienti": [
        {
            "clienteId": 1,
            "nome": 'Francesco␣Scannapietro',
            "indirizzo": 'Napoli',
            "eta": 30,
            "telefono": '081-2292929',
            "email": '',
            "reclami": [
              {
                "recid": 1,
                "data": '27-08-2023',
                "prodotto": 'P123',
                "motivo": 'Pacco␣mai␣arrivato'
              },
```

```
            {
              "recid": 2,
              "data": '21-09-2023',
              "prodotto": 'P123',
              "motivo": 'Numero clienti risulta occupato'
            }
          ]
      },
      {
          "clienteId": 2,
          "nome": 'Gaetano Scandurra',
          "indirizzo": 'San Gennaro Vesuviano',
          "eta": 33,
          "telefono": '',
          "email": 'gsca@scandurra.com',
          "reclami": [
            {
              "recid": 3,
              "data": '12-12-2022,',
              "prodotto": 'P321',
              "motivo": 'Prodotto difettoso'
            }
          ]
      }
  ]
}
```

Questo approccio denormalizza i dati dei reclami all'interno del documento cliente, facilitando le operazioni di lettura frequente che coinvolgono sia il cliente che i suoi reclami.

b) **Si formuli la query Q1 e Q2 in MongoDB Query Language (uso del valore a piacere per r) in base ai dati forniti al punto precedente.**

Query Q1

- **Soluzione 1**

```
db.clienti.find(
  { "clienteId":  1 },
  { "reclami.prodotto": 1, _id: 0 }
)
```

- **Soluzione 2**

```
db.clienti.aggregate([
  // Filtra per il cliente di interesse
  { $match: { clienteId: 1 } },
  // Espande i reclami in documenti singoli
  { $unwind: "$reclami" },
  // Raggruppa per il campo prodotto
  { $group: { _id: "$reclami.prodotto" } },
  // Proietta solo il campo prodotto senza duplicati
  { $project: { _id: 0, prodotto: "$_id" } }
])
```

Query Q2

- **Soluzione 1**

```
db.clienti.find(
  { "reclami.recid": 1 },
   { "reclami.motivo": 1, _id: 0 }
)
```

- **Soluzione 2**

```
db.clienti.find(
  {
    "reclami": { $elemMatch: { "recid": 1 } }
  },
  { "reclami.motivo": 1, _id: 0 }
)
```

E' possibile testare l'esercizio con il seguente compilatore online: OneCompiler

9.3 Esercizio 3: Graph DBs (6 punti)

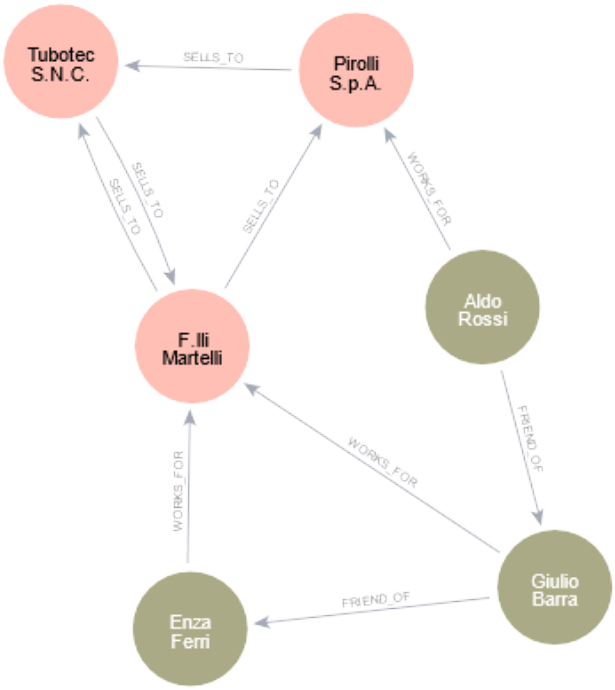
Si vogliono rappresentare aziende e relazioni tra di esse; in particolare quali aziende vendono servizi e/o beni ad altre; inoltre si rappresentano dipendendi di dette aziende e le amicizie tra di loro. L'azienda Pirolli S.p.A. vende a Tubotec S.N.C. dal 1998, la quale vende a F.lli Martelli dal 2003. L'azienda F.lli Martelli vende a Pirolli S.p.A. dal 1989 e a Tubotec S.N.C. dal 1994. Aldo Rossi lavora per Pirolli S.p.A. ed è amico di Giulio Barra che lavora per F.lli Martelli. Enza Ferri è amica di Giulio Barra e lavora per F.lli Martelli.

a) **Rappresentare i dati sopra descritti in un property graph.**

```
// Creazione dei nodi per le aziende
CREATE (pirolli:Company {name: 'Pirolli␣S.p.A.'})
CREATE (tubotec:Company {name: 'Tubotec␣S.N.C.'})
CREATE (flliMartelli:Company {name: 'F.lli␣Martelli'})

// Creazione dei nodi per le persone
CREATE (aldo:Person {name: 'Aldo␣Rossi'})
CREATE (giulio:Person {name: 'Giulio␣Barra'})
CREATE (enza:Person {name: 'Enza␣Ferri'})

// Creazione delle relazioni commerciali
CREATE (pirolli)-[:SELLS_TO {since: 1998}]->(tubotec)
CREATE (tubotec)-[:SELLS_TO {since: 2003}]->(flliMartelli)
CREATE (flliMartelli)-[:SELLS_TO {since: 1989}]->(pirolli)
CREATE (flliMartelli)-[:SELLS_TO {since: 1994}]->(tubotec)

// Creazione delle relazioni di lavoro
CREATE (aldo)-[:WORKS_FOR]->(pirolli)
CREATE (giulio)-[:WORKS_FOR]->(flliMartelli)
CREATE (enza)-[:WORKS_FOR]->(flliMartelli)

// Creazione delle amicizie
```

```
23 CREATE (aldo)-[:FRIEND_OF]->(giulio)
24 CREATE (giulio)-[:FRIEND_OF]->(enza)
```

Listing 29: Codice per la creazione del grafo in Neo4j



b) **Scrivere in Cypher una query che conta quanti dipendenti ha ciascuna azienda che ha come dipendente un amico di Aldo Rossi, e vende a Pirolli S.p.A**

- **Soluzione 1**

```
1 MATCH (aldo:Person {name: 'Aldo Rossi'})-[:FRIEND_OF]-(
      friend:Person)-[:WORKS_FOR]-(company:Company),
2       (company)-[:SELLS_TO]->(pirolli:Company {name: '
            Pirolli S.p.A.'})
3 WITH company
4 MATCH (company)<-[:WORKS_FOR]-(employee:Person)
5 RETURN company.name AS Company, COUNT(employee) AS
      NumberOfEmployees
```

- **Soluzione 2**

```
1 MATCH (pirolli:Company {name: "Pirolli S.p.A."})<-[:
      SELLS_TO]-(a2:Company)<-[:WORKS_FOR]-(e:Person)-[:
      FRIEND_OF]-(aldo:Person {name: "Aldo Rossi"})
```

```
WITH a2, COUNT(e) AS numDipendenti
RETURN a2.name AS Company, numDipendenti
```

9.4 Esercizio 4: RDF (5 punti)

a) **Si illustri brevemente l'uso dei nomi di dominio (domain names) negli IRIs in RDF.**

Gli Internationalized Resource Identifiers (IRIs) sono utilizzati in RDF per identificare in modo univoco le risorse. L'uso dei nomi di dominio negli IRIs è importante per garantire l'unicità globale delle risorse, in quanto i nomi di dominio sono controllati centralmente e possono essere utilizzati per creare spazi dei nomi unici. Questo evita conflitti tra i nomi delle risorse definite da diverse organizzazioni o individui.

b) **Si consideri il grafo RDF di SPARQL Playground visto a lezione e rappresentato in Figura 2 a pag. 4. Formulare in SPARQL una query che conta, per ciascuna persona, di quanti gatti detta persona è padrone (tto:pet denota il fatto che una persona è padrone di un animale)**

- **Soluzione 1**

```
PREFIX tto: <http://example.org/tto#>

SELECT ?person (COUNT(?cat) AS ?numeroDiGatti)
WHERE {
  ?person tto:pet ?pet .
  ?pet tto "Cat"
}
GROUP BY ?person
```

- **Soluzione 2**

```
PREFIX tto: <http://example.org/tto#>

SELECT ?person (COUNT(?pet) AS ?numberOfPets)
WHERE {
  ?person tto:pet ?pet .
  ?pet a <http://example.org/cat#Cat> .
}
GROUP BY ?person
```

9.5 Esercizio 5: Datalog (7 punti)

Sia dato un database D espresso con i seguenti fatti Datalog.

```
coniuge(enzo, concetta).
genitore(enzo, pasquale).
genitore(francesca, gennaro).
fs(enzo, francesca).
```

Listing 30: Fatti

a) **Scrivere un programma Datalog P, definendo opportuni predicati IDB, con le seguenti caratteristiche**

- I predicati *coniuge* e *fs* sono simmetrici (ciò va espresso con una regola Datalog per ciascun predicato coniuge e fs).

```
coniuge(X, Y) :- coniuge(Y, X).
fs(X, Y) :- fs(Y, X).
```

Listing 31: Predicati Simmetrici

- Se una persona p è genitore di una persona f, allora anche il coniuge di p è genitore di f (ciò va espresso con una regola Datalog).

```
genitore(Y, F) :- genitore(X, F), coniuge(X, Y).
```

Listing 32: Regola per il coniuge genitore

- Il predicato binario cug esprime che due persone sono cugini l'uno dell'altro (col significato ovvio che due persone sono cugini se figli di due persone che sono fratelli/sorelle tra loro).

```
cug(X, Y) :- genitore(P1, X), genitore(P2, Y), fs(P1, P2).
```

Listing 33: Predicato cugini

- Il predicato binario cug-gen ("cugino generalizzato") esprime che due persone (di qualunque sesso) sono cugini, oppure uno cugino di un cugino generalizzato dell'altro.

```
cug_gen(X, Y) :- cug(X, Y).
cug_gen(X, Y) :- cug_gen(Y, X).
cug_gen(X, Y) :- cug_gen(X, Z), cug_gen(Z, Y).
```

Listing 34: Predicato cugino generalizzato

b) **Si determini il minimo punto fisso $TP, \infty(D)$ contenente D, dell'operatore di conseguenza immediata T_p.**

L'*operatore di conseguenza immediata* (TP) è un operatore che, partendo da un insieme di fatti iniziali, applica le regole per generare nuovi fatti. Ogni applicazione di TP si chiama "iterazione".
Il *punto fisso* è una condizione in cui ulteriori applicazioni dell'operatore TP non producono nuovi fatti. In altre parole, si raggiunge un punto in cui l'insieme dei fatti non cambia più.
Il *minimo punto fisso* TP, ∞ è il punto fisso raggiunto applicando ripetutamente TP a partire dai fatti iniziali fino a quando non si possono più generare nuovi fatti.
Partendo dunque dai fatti iniziali si ha:

- **Alla prima iterazione di T_p**

```
coniuge(concetta, enzo).
fs(francesca, enzo).
```

Listing 35: Applicato alla simmetria

```
genitore(concetta, pasquale).
```

Listing 36: Applicato alla regola del coniuge genitore

- **Alla seconda iterazione di T_p**

```
cug(pasquale, gennaro).
```

Listing 37: Applicato alla regola dei cugini

- **Alla terza iterazione di T_p**

```
cug_gen(pasquale, gennaro).
cug_gen(gennaro, pasquale)..
```

Listing 38: Applicato alla regola dei cugini generalizzati

- **Alla quarta iterazione di T_p** non possiamo più generare nuovi fatti applicando le regole, quindi abbiamo raggiunto il punto fisso.

Si ha il *minimo punto fisso* $TP, \infty(D)$ (che consiste nell'insieme dei fatti ottenuti dalle iterazioni precedenti unito ai fatti dati):

```
coniuge(enzo, concetta).
coniuge(concetta, enzo).
genitore(enzo, pasquale).
genitore(francesca, gennaro).
genitore(concetta, pasquale).
fs(enzo, francesca).
fs(francesca, enzo).
cug(pasquale, gennaro).
cug_gen(pasquale, gennaro).
cug_gen(gennaro, pasquale).
```

Listing 39: Minimo punto fisso $TP, \infty(D)$

10 Esame del 12/09/24

10.1 Esercizio 1: DB Distribuiti e Consistenza (6 punti)

a) **Nel contesto del CAP Theorem, si definisca brevemente il concetto di linearizzabilità (linearizability), CAP-availability, e partition tolerance**.

La *Linearizzabilità* (linearizability) una proprietà di consistenza nei sistemi distribuiti che garantisce che tutte le operazioni appaiono come se fossero eseguite in un ordine sequenziale e che ogni operazione prende effetto in un punto tra il suo inizio e la sua fine. In altre parole, le operazioni appaiono atomiche. In altre parole, se un'operazione B inizia dopo che un'operazione A è stata completata, l'operazione B deve vedere il sistema nello stesso stato in cui si trovava alla fine dell'operazione A, o in uno stato più recente.
La *Disponibilità* (availability) è una delle proprietà del teorema CAP e afferma che in un sistema distribuito ogni richiesta fatta a un nodo funzionante riceve una risposta, anche se non può garantire che la risposta contenga l'ultimo dato scritto. La disponibilità implica che il sistema deve essere progettato per gestire i guasti in modo da garantire che le operazioni possano continuare. Ad esempio, nei database distribuiti, la disponibilità può essere garantita replicando i dati su più nodi. In questo modo, se uno o più nodi diventano inaccessibili, il sistema può ancora rispondere alle richieste utilizzando le repliche disponibili.
La *Tolleranza alle partizioni* (partition tolerance) garantisce che un sistema distribuito continua a funzionare correttamente anche quando ci sono partizioni o guasti di rete che impediscono la comunicazione tra alcuni nodi. In un sistema tollerante alle partizioni, i nodi continueranno a rispondere alle richieste anche se sono isolati da altre parti del sistema. Il teorema CAP indica che, in presenza di partizioni, un sistema deve scegliere tra disponibilità e consistenza.

b) **Fornire un breve esempio in cui la linearizzabilità non sussiste, aiutandosi eventualmente con una figura.**

Immagina un sistema distribuito con due nodi A e B. Supponiamo che un client C1 scriva il valore X = 10 su A, e successivamente un altro client C2 legge il valore di X da B. Nel frattempo, prima che B riceva l'aggiornamento da A, C2 legge il vecchio valore di X = 5.

- Scrittura (C1): Scrive X = 10 su A.
- Lettura (C2): Legge X = 5 da B (che non è stato ancora aggiornato).

In questo scenario, la linearizzabilità non è soddisfatta, perché nonostante la scrittura su A sia stata completata, la lettura su B non riflette l'aggiornamento più recente. In un sistema lineare, C2 avrebbe dovuto leggere X = 10, non X = 5.

```
Tempo
 |
 |
 |  C1 scrive X = 10 su nodo A
 |  ------------------------>  Nodo A ha X = 10
 |
 |  C2 legge X = 5 da nodo B
```

```
8 |  ------------------------>  Nodo B ha ancora X = 5 (non
      aggiornato)
9 |
```

In questo esempio, la mancata sincronizzazione immediata tra A e B viola la linearizzabilità, perché i client vedono versioni diverse dello stesso dato.

Un esempio di sistema *non linearizzabile* può verificarsi in un database NoSQL distribuito, come *Cassandra*, quando si utilizza la consistenza eventuale.
Per esempio: Immaginiamo un'applicazione di social media con più server distribuiti in diverse aree geografiche. Se un utente modifica il proprio stato da offline a online, questa modifica viene inviata a più server per essere replicata. Tuttavia, a causa di latenze di rete, alcuni server potrebbero non ricevere immediatamente l'aggiornamento. A questo punto, il server A potrebbe vedere l'utente come online mentre il server B, che non ha ancora ricevuto l'aggiornamento, potrebbe vederlo come offline.
Se due utenti interrogano questi server allo stesso tempo, vedranno informazioni diverse sull'utente, il che indica che il sistema non è linearizzabile. Questo compromesso è accettato in molti sistemi distribuiti NoSQL per migliorare la disponibilità e la tolleranza ai guasti.

Approfondimento: **Linearizzabilità vs Consistenza**
La linearizzabilità (linearisability) e la consistenza nei sistemi distribuiti non sono considerate esattamente la stessa cosa, anche se sono concetti correlati.
La *consistenza* in un sistema distribuito si riferisce alla garanzia che tutte le repliche di un dato abbiano lo stesso valore in ogni momento, esistendo cosi anche diversi livelli di consistenza, e la consistenza forte è uno di questi. Nella consistenza forte, ogni operazione di lettura riflette sempre l'ultima scrittura, ovvero le scritture sono visibili a tutte le repliche prima di qualsiasi lettura successiva.
La *linearizzabilità* è una proprietà di consistenza forte che va oltre la semplice coerenza dei dati. È un modello più rigoroso di consistenza. Un sistema è lineare se ogni operazione sembra accadere in un momento preciso, come se fosse eseguita in modo "istantaneo".
In conclusione la linearizzabilità è un tipo di consistenza forte, ma più rigoroso, perché impone che tutte le operazioni rispettino un ordine globale preciso. Non tutte le forme di consistenza forte garantiscono però la linearizzabilità, ma la linearizzabilità garantisce sempre consistenza forte.

10.2 Esercizio 2: DB Documentali (6 punti)

Un sistema informativo deve contenere dati relativi a clienti di un'azienda che vende prodotti e reclami da essi presentati"; ogni reclamo è presentato da un unico cliente; ogni cliente ovviamente può presentare piú reclami. II cliente

Francesco Scannapieco abita a Napoli e ha numero di telefono 081-2292929; il cliente Gaetano Scandurra ha 33 anni, abita a San Gennaro Vesuviano e ha indirizzo di posta elettronica gsca@scandurra.com. Il cliente Francesco Scannapieco ha presentato un reclamo il 27-08-2023, per il prodotto con codice identificativo P123 con motivo "Pacco mai arrivato", e un altro il 21-09-2023 con motivo "Numero clienti risulta occupato". Il cliente Gaetano Scandurra ha presentato un reclamo il 12-12-2022, per il prodotto P321, con motivo "Prodotto difettoso". Si stima che la base di dati dovrà contenere dati come quelli di cui sopra, con circa 100.000 clienti e 20.000 reclami. Si prevede inoltre che saranno eseguite le sequenti operazioni con relative frequenze stimate:
Q1: Dato l'identificativo r di un reclamo, trovare l'identificativo e il nome del cliente che ha presentato detto reclamo. (8.000 interrogazioni al giorno)
Q2: Dato l'identificativo c di un cliente, trovare il recapito telefonico del cliente in questione. (7.000 interrogazioni al giorno)

a) **Si illustri come memorizzare i dati di esempio, rappresentandoli in JSON, considerando le operazioni e la loro frequenza. Si spieghi brevemente le eventuali denormalizzazioni.**

```
{
  "clienti": [
        {
             "clienteId": 1,
             "nome": "Francesco␣Scannapieco",
             "citta": "Napoli",
             "telefono": "081-2292929",
             "reclami": [
               {
                 "codice": 1,
                 "codice_prodotto": "P123",
                 "data": "2023-08-27",
                 "motivo": "Pacco␣mai␣arrivato"
               },
               {
                 "codice": 2,
                 "data": "2023-09-21",
                 "motivo": "Numero␣clienti␣occupato"
               }
             ]
      },
      {
             "clienteId": 2,
             "nome": "Gaetano␣Scandurra",
             "citta": "San␣Gennaro␣Vesuviano",
             "eta": 33,
             "email": "gsca@scandurra.com",
             "reclami": [
               {
                 "codice": 3,
                 "codice_prodotto": "P321",
                 "data": "2022-12-12",
                 "motivo": "Prodotto␣difettoso"
               }
```

```
            ]
        }
    ]
}
```

Questo approccio denormalizza i dati dei reclami all'interno del documento cliente, facilitando le operazioni di lettura frequente che coinvolgono sia il cliente che i suoi reclami.

b) **Si formulino le query Q1 e Q2 in MongoDB Query Language (uso del valore a piacere) in base ai dati forniti al punto precedente.**

- **Query Q1**

```
db.clienti.findOne(
  {"reclami.codice": 1},
  {"nome":1, "clienteId": 1 ,_id: 0}
)
```

- **Query Q2**

```
db.clienti.findOne(
  {"clienteId": 1},
  {"nome": 1, "telefono": 1, _id: 0}
)
```

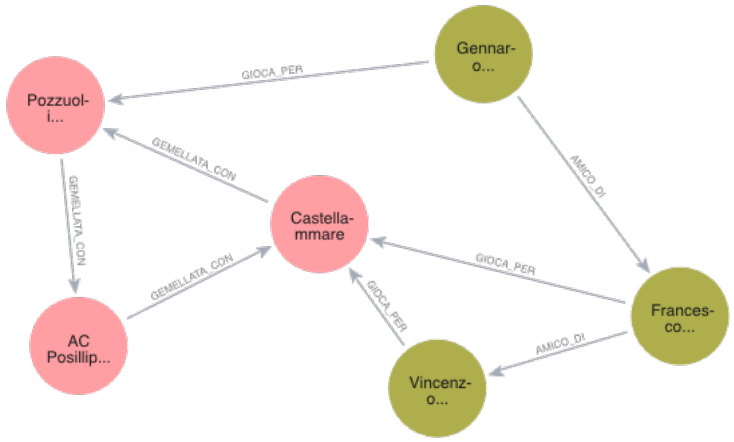
E' possibile testare l'esercizio con il seguente compilatore online: OneCompiler

10.3 Esercizio 3: Graph DBs (6 punti)

Si vogliono rappresentare squadre di calcio e relazioni tra di esse; in particolare quali squadre sono gemellate con altre; inoltre si rappresentano giocatori di dette squadre e le amicizie tra di loro. Pozzuoli FC è gemellata con AC Posillipo dal 2010; AC Posillipo è gemellata con Castellammare dal 2002; Castellammare è gemellata con Pozzuoli FC dal 1999. Gennaro Scandurra gioca con Pozzuoli FC ed è amico di Francesco Scannagatta il quale gioca con Castellammare; Vincenzo Fusco è amico di Francesco Scannagatta e gioca anch'egli con Castellammare.

a) **Rappresentare i dati sopra descritti in un property graph.**

```
// Creazione delle squadre
CREATE (s1:Squadra {nome: "Pozzuoli FC"})
CREATE (s2:Squadra {nome: "AC Posillipo"})
CREATE (s3:Squadra {nome: "Castellammare"})
CREATE (g1:Giocatore {nome: "Gennaro Scandurra"})
CREATE (g2:Giocatore {nome: "Francesco Scannagatta"})
CREATE (g3:Giocatore {nome: "Vincenzo Fusco"})

// Gemellaggi tra le squadre
CREATE (s1)-[:GEMELLATA_CON {dal: 2010}]->(s2)
CREATE (s2)-[:GEMELLATA_CON {dal: 2002}]->(s3)
CREATE (s3)-[:GEMELLATA_CON {dal: 1999}]->(s1)

```




```
// Relazioni di "Gioca_per" tra giocatori e squadre
CREATE (g1)-[:GIOCA_PER]->(s1)
CREATE (g2)-[:GIOCA_PER]->(s3)
CREATE (g3)-[:GIOCA_PER]->(s3)

// Relazioni di amicizia tra i giocatori
CREATE (g1)-[:AMICO_DI]->(g2)
CREATE (g2)-[:AMICO_DI]->(g3)
```

Listing 40: Codice per la creazione del grafo in Neo4j

b) **(Scrivere in Cypher una query che conta quanti giocatori ha ciascuna squadra che ha come giocatore un amico di Gennaro Scandurra, e è gemellata con Pozzuoli FC.**

```
MATCH (g: Giocatore {nome: "Gennaro Scandurra"})-[:AMICO_DI]-(
    amico: Giocatore)-[:GIOCA_PER]->(s:Squadra)-[:
    GEMELLATA_CON]->(:Squadra {nome: "Pozzuoli FC"})
WITH s
MATCH (giocatore: Giocatore)-[:GIOCA_PER]-(s)
RETURN s.nome AS Squadra,  COUNT(giocatore) AS numGiocatori
```

10.4 Esercizio 4: RDF (5 punti)

a) **Si illustri brevemente l'uso dei nomi di dominio (domain names) negli IRIs in RDF.**

Gli Internationalized Resource Identifiers (IRIs) sono utilizzati in RDF per identificare in modo univoco le risorse. L'uso dei nomi di dominio negli IRIs è importante per garantire l'unicità globale delle risorse, in quanto i nomi di dominio sono controllati centralmente e possono essere utilizzati per creare spazi dei nomi unici. Questo evita conflitti tra i nomi delle risorse definite da diverse organizzazioni o individui.

b) **Si consideri il grafo RDF di SPARQL Playground visto a lezione. Formulare in SPARQL3 una query che conta, per ciascun gatto, quanti padroni ha detto gatto (tto: pet denota il fatto che una persona è padrone di un animale).**

```
SELECT ?gatto, (COUNT(?padrone) AS ?numPadroni)
WHERE {
    ?gatto tto:Animal tto:Cat .
    ?padrone tto:pet ?gatto .
}
GROUP BY ?gatto
```

10.5 Esercizio 5: Datalog (7 punti)

Sia dato un database D espresso con i seguenti fatti Datalog.

```
coniuge(enzo, concetta).
fs(enzo, pasquale).
fs(concetta, carmela).
coniuge(carmela, ciro).
```

Listing 41: Fatti

a) **Scrivere un programma Datalog P, definendo ove opportuno opportuni predicati IDB, che abbia le seguenti proprietà:**

- I predicati *coniuge* e *fs* sono simmetrici (ciò va espresso con una regola Datalog per ciascun predicato coniuge e fs).

```
coniuge(X, Y) :- coniuge(Y, X).
fs(X, Y) :- fs(Y, X).
```

Listing 42: Predicati Simmetrici

- Il predicato binario *cogn* esprime che due persone sono cognati l'uno dell'altro, col significato ovvio che due persone sono cognati se uno è fratello/sorella del coniuge dell'altro.

```
cogn(X, Y) :- coniuge(X, Z), fs(Z, Y).
cogn(X, Y) :- fs(X, Z), coniuge(Z, Y).
```

Listing 43: Regola per il coniuge

- Il predicato binario *cogn_gen* ("cugino generalizzato") esprime che due persone (di qualunque sesso) sono cognati, oppure uno cognato di un cognato generalizzato dell'altro.

```
cogn_gen(X, Y) :- cogn(X, Y).
cogn_gen(X, Y) :- cogn_gen(X, Z), cogn_gen(Z, Y).
```

Listing 44: Predicato cognati generalizzati

Il predicato *cogn_gen* estende il concetto di cognato, includendo i casi in cui due persone sono cognati tramite una catena di cognati generalizzati.
La prima regola afferma che X e Y sono cognati generalizzati se sono cognati diretti. La seconda regola permette la catena di cognati generalizzati: X è cognato di Z e Z è cognato generalizzato di Y, quindi X è cognato generalizzato di Y.
La regola

```
cogn_gen(X, Y) :- cogn_gen(Y, X).
```

non è necessaria in quanto rende la relazione simmetrica in modo non controllato. Questa regola può portare a loop infiniti, perché permette una continua inversione dei ruoli di X e Y

b) **Si determini il minimo punto fisso $TP, \infty(D)$ contenente D, dell'operatore di conseguenza immediata T_p.**

L'*operatore di conseguenza immediata* (TP) è un operatore che, partendo da un insieme di fatti iniziali, applica le regole per generare nuovi fatti. Ogni applicazione di TP si chiama "iterazione".
Il *punto fisso* è una condizione in cui ulteriori applicazioni dell'operatore TP non producono nuovi fatti. In altre parole, si raggiunge un punto in cui l'insieme dei fatti non cambia più.
Il *minimo punto fisso* TP, ∞ è il punto fisso raggiunto applicando ripetutamente TP a partire dai fatti iniziali fino a quando non si possono più generare nuovi fatti.

Si ha il *minimo punto fisso* $TP, \infty(D)$:

```
coniuge(enzo, concetta).
coniuge(concetta, enzo).
coniuge(carmela, ciro).
coniuge(ciro, carmela).

fs(enzo, pasquale).
fs(pasquale, enzo).
fs(concetta, carmela).
fs(carmela, concetta).

cogn(enzo, carmela).
cogn(carmela, enzo).
cogn(pasquale, concetta).
cogn(concetta, pasquale).
cogn(ciro, concetta).
cogn(concetta, ciro).

cogn_gen(enzo, carmela).
cogn_gen(carmela, enzo).
cogn_gen(ciro, concetta).
cogn_gen(concetta, ciro).
```

```
22 cogn_gen(enzo, ciro).
```

Listing 45: Minimo punto fisso $TP, \infty(D)$

c) Approfondimento: **Upper bound del minimo punto fisso**

È un concetto teorico che si riferisce al più grande insieme di fatti che potrebbe essere derivato dalle regole del programma, considerando tutte le combinazioni possibili tra gli elementi (anche se alcune di queste combinazioni non sono effettivamente raggiungibili).
Quando si esegue un programma Datalog, l'obiettivo è trovare un insieme di fatti che soddisfi tutte le regole del programma. Quindi si parte da un insieme di fatti iniziali, si applicano le regole del programma per derivare nuovi fatti e si raggiunge un punto in cui non è più possibile aggiugerne degli altri (minimo punto fisso).
L'upper bound sarebbe l'insieme massimo di tutte le possibili combinazioni di persone che potrebbero essere coniugi, fratelli, cognati o cognati generalizzati. Anche se alcune di queste combinazioni non sono effettivamente derivabili dalle regole, teoricamente rappresentano il limite massimo che il sistema potrebbe raggiungere.

Come calcolare l'upper bound:

- *Possibili combinazioni di fatti*: L'upper bound si basa su tutte le combinazioni possibili di persone per ciascun predicato. Ad esempio, per il predicato coniuge, consideriamo ogni coppia di persone nel dominio (anche se non sono realmente coniugi).
- *Tutte le coppie possibili*: Se hai 4 persone nel database, puoi avere $4 \times 4 = 16$ combinazioni per ogni predicato binario. Quindi, l'upper bound sarebbe l'insieme più ampio di relazioni che potrebbero teoricamente essere derivate, anche se non è possibile derivarle tutte.

Nel nostro esercizio abbiamo 5 persone:

- Enzo
- Concetta
- Pasquale
- Carmela
- Ciro

E abbiamo 4 predicati binari: *coniuge*, *fs*, *cogn*, e *cogn_gen*. Per ogni predicato, possiamo considerare tutte le combinazioni possibili di queste persone.

Calcolo dell'upper bound:

- *Coniuge (coniuge)*: Il predicato coniuge può, teoricamente, essere applicato a tutte le coppie di persone, quindi avremmo $5 \times 5 = 25$ combinazioni di fatti. Tuttavia, in realtà solo alcune di queste combinazioni sono vere.
- *Fratello/Sorella (fs)*: Analogamente, il predicato fs può teoricamente applicarsi a tutte le coppie di persone, generando altre 25 combinazioni.
- *Cognati (cogn)*: Se applichiamo le regole per cogn(X, Y) a tutte le combinazioni di persone, anche qui teoricamente potremmo ottenere 25 combinazioni. Tuttavia, non tutte le persone nel database hanno relazioni coniugali o di fratellanza che permettono di derivare la relazione cogn.
- *Cognati generalizzati (cogn_gen)*: Questo predicato può applicarsi transitivamente a tutte le persone, e quindi anche qui potremmo teoricamente ottenere altre 25 combinazioni.

L'upper bound programma Datalog sarà il numero massimo di fatti che possono essere derivati considerando tutte le possibili combinazioni di persone per ciascun predicato. Nel caso di 5 persone, l'upper bound sarà basato su 25 combinazioni per ogni predicato.
Il numero massimo di fatti sarà:

$$25 + 25 + 25 + 25 = 100$$