Linked-adm-In:

Foschi Lorenzo, Pezzano Enrico

Final project for the Advanced Data Management course

Dataset link

https://www.kaggle.com/datasets/arshkon/linkedin-job-postings

Nature of the Proposed Application

1. Propose a domain you are interested in and a relevant application for it (e.g.,e-commerce domain, shopping cart and session data management application). Take inspirations from datasets available online (e.g., on https://www.kaggle.com).
2. Provide details about the nature of the proposed application (e.g., the application is read/write intensive, requires batch processing, ...), according to what discussed in the course and corresponding system requirements (e.g., eventual/strong consistency needed, high availability needed).

The proposed application is a job and industry relationship analysis tool. Its features and requirements include:

Read/Write Intensity

The application is predominantly read-intensive, as it emphasizes retrieving data; such as querying the skills required for a job, the benefits offered, or the industries associated with a company. Write operations occur moderately, primarily when new jobs, companies, skills, or benefits are added to the system. These updates may include inserting related attributes like market values, skill scores, or job expiration dates.

Batch Processing

No Batch processing, beacause the only big import can be required during initialization: no large updates to the dataset will occur. The only updates might be occasionaly integrating new company records or updated job postings.

Consistency and Availability

Eventual consistency is sufficient for updates to non-critical attributes, such as adding new benefits or updating a company's market value. These changes are not immediately critical for most queries. However, for other operations like adding new jobs or companies ensuring consistency is more critical, as incomplete or incorrect relationships (e.g. a job missing its associated company) could affect query accuracy. Also, having high availability clearly is the most important goal to reach, particularly for supporting fast real-time data retrieval and queries (we want to enable people to find jobs!)

Performance

The system should be optimized for retrieving and filtering data (also across entities), such as: Searching for jobs based on attributes like type, location, or expiration date. Filtering companies by criteria like

market value, location, or associated industries.

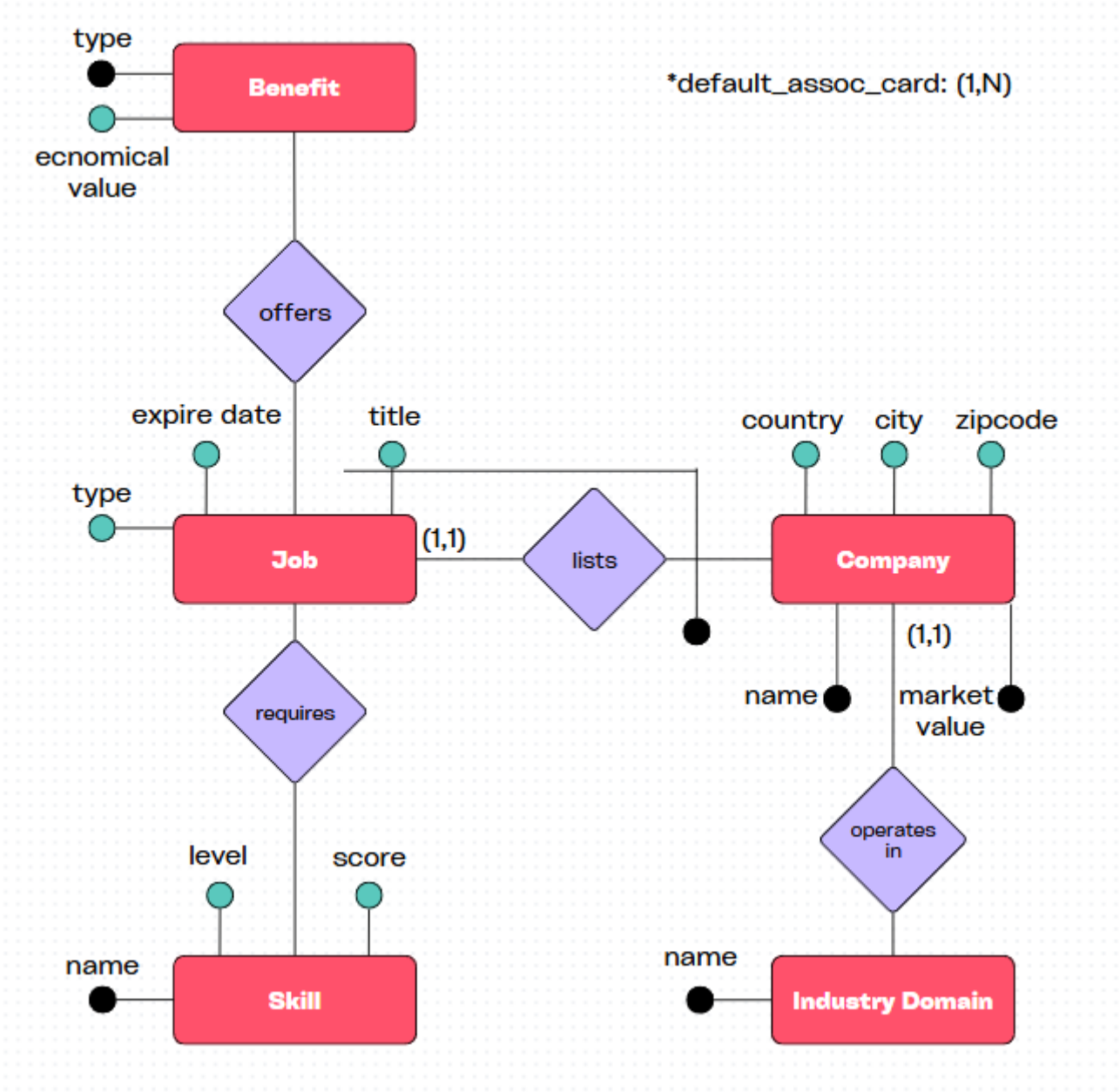
Conceptual Schema

3. Design a conceptual schema for the identified domain. The schema should include at least three associations.

The following conceptual schema captures key entities and their relationships:

```
Nodes:
    Job(Title, Expire date, Type): Represents a specifc job posting
    Company(Name, Country, City, ZipCode, MarketValue): Represents a
company
    IndustryDomain(Name): Represents an industry domain
    Skill(Name, Level, Score): Represents skills associated with a job
    Benefit(Type, Economical Value): Represents benefits offered for a job

Relationships:
    BELONGS_TO: Links a job to an company
    REQUIRES: Links a job to the skills it demands
    OFFERS: Links a job to the benefits it provides
    OPERATES_IN: Links a company to the industries it operates in
```



Queries:

4. Identify a workload, i.e., a set of relevant and frequent operations, related to the chosen application. The workload should contain at least 5 structurally different operations. Describe each workload operation in natural language:

   Query 1: Find all jobs that are Full-time and expiring within the next 30 days. **Q1**(Job, [Job(type)*!, Job(expire_date)*!], [Job_!])

   Query 2: List the name of companies in New York with a market value greater than $1,000,000. **Q2**(Company, [Company(city)*!, Company(mv)*!], [Company(name)_!])

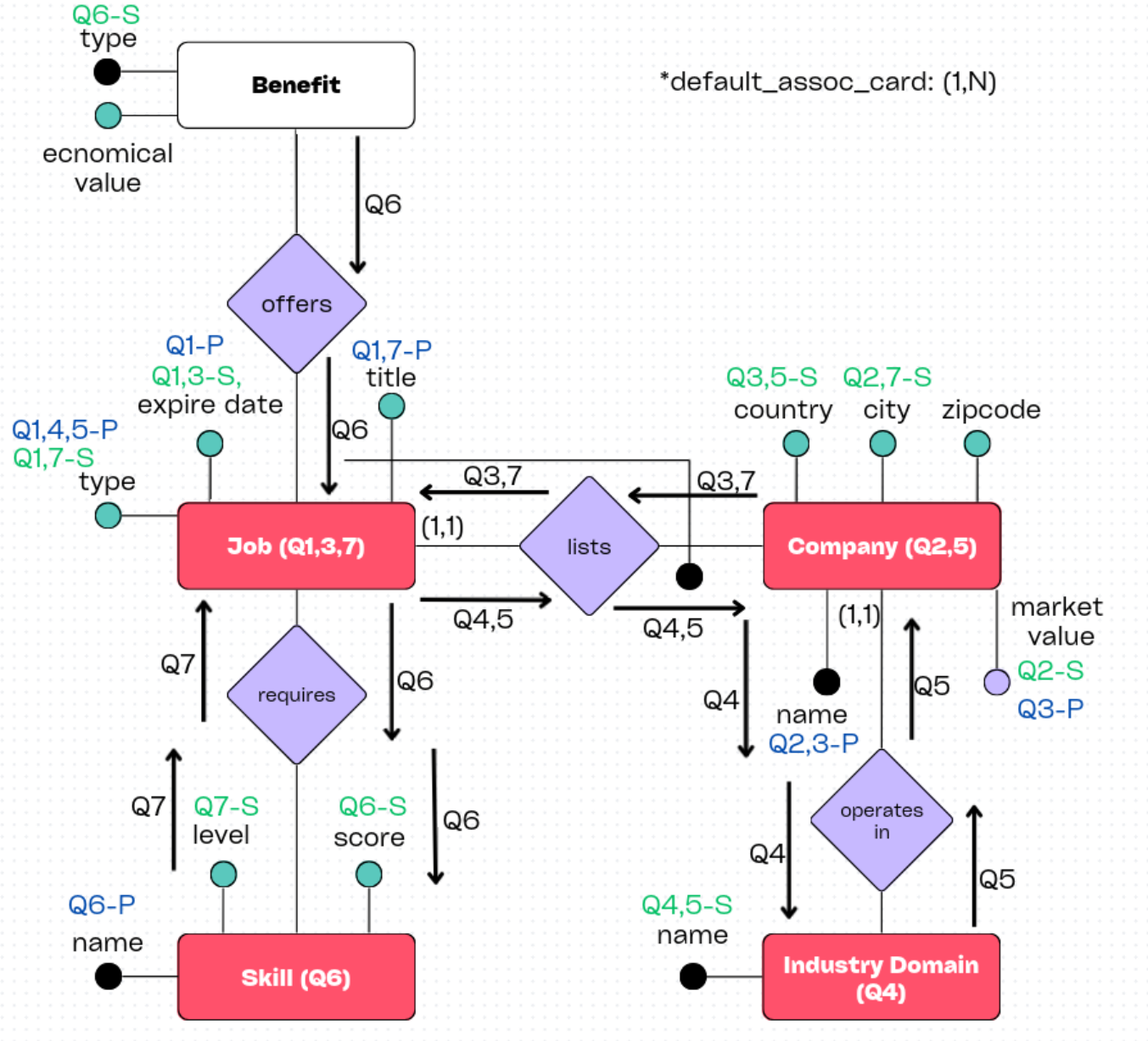
   Query 3: List name and market value of companies operating in Russia and associated with jobs that expire in more than 60 days. **Q3**(Job, [Company(country)*L, Job(expire_date)*!], [Company(name, mv)_L])

   Query 4: List type of jobs associated with companies operating in the Technology domain **Q4**(IndustryDomain, [IndustryDomain(Name)_!], [Job(type)_LO])

Query 5: List type jobs associated with italian companies operating in the Technology domain **Q5**(Company, [IndustryDomain(Name)*O, Company(country)*!], [Job(type)_L])

Query 6: Retrieve name of skills required for jobs offering benefits of type 401(k) and having a score above 70. **Q6**(Skill, [Skill(score)_!, Benefit(type)*OR], [Skill(name)*!])

Query 7: Find the title of all jobs of type Internship that require skills with a level of "Beginner" and are associated with companies in Hamburg **Q7**(Job, [Job(type)_!, Skill(level)_R, Company(city)*L], [Job(title)*!])



Aggregates

5. Use the aggregate-oriented design methodology (STEP 1-2-3) to design a set of aggregates for the domain and the workload at hand.

JobOffer: { title, companyName, expire_date, type, country, city, marketValue, requires: [{skill: {level}}] }

Company: { name, marketValue, country, city, job_offers: [{job: {type}}], industryName }

IndustryDomain: { name, operated: [{ company: [{ job: {type} }] }] }

Note: dobule n-n relationship kept as list[lists] to semanthically keep the companies for further needs

Skill: { name,score, provides: [{benefit: {type}}] }

Note: double n-n relationship unpacked into a single list: we're only interested in benefits that skills provides, regardless of the jobs

Design in MongoDB

6. Design in MongoDB: a. Design a schema for MongoDB (including partition keys and indexes), starting from step 5, using the approaches/methodologies proposed in the course. b. Specify each operation of the workload in the language supported by MongoDB

Queries associated with Skill: Q6

Selection attributes for Q6: {score, type}

The aggregate will remain the same...

Skill: { name,score, provides: [{benefit: {type}}]

Then MongoDB will add the _id:

{ _id, name,score, provides: [{benefit: {type}}] }

About indexes: We have to support the shard key via an index, either with full or compound index with the shard key as a prefix. Then, for enforcing uniqueness we would also need a unique index on the shard key itself (otherwise the uniqueness constraint across shards can't be enforced). Nevertheless, we opt to put a index which contains the full shard key as a prefix of the index = {score, type} and avoid enforcing uniqueness across partitions (we don't consider the skill name to be so crucial to be actually unique across shards).

```
db.skills.createIndex({ score: 1, "provides.benefit.type": 1 });
```

- Shard collection

```
db.adminCommand({
  shardCollection: "db.skill",
  key: { score: 1, "provides.benefit.type": 1 }
});
```

- Q6:

```
  db.skills.find({score: {$gt: 70}, "provides.benefit.type": "401(k)"},
{name: 1, _id: 0});
```

Note: Remembering that there's no such a thing as "CREATE TABLE" in MongoDB we avoid reporting here insert commands of example data for the collection.

Queries associated with IndustryDomain: Q4 Selection attributes for Q4: {name}

The aggregate will remain the same...

IndustryDomain: { name, operated: [{ company: [{ job: {type} }] }] }

Then MongoDB will add the _id:

{ _id, name, operated: [{ company: [{ job: {type} }] }] }

About indexes:

- A unique index on the partition key, which is also the aggregate key = {name}

```
  db.industryDomains.createIndex({ name: 1 }, { unique: true });
```

- Shard collection

```
db.adminCommand({
  shardCollection: "db.industryDomain",
  key: { name: 1 }
});
```

- Q4:

```
  db.industryDomains.aggregate([
  {
    $match: {
      name: "Technology"
    }
  },
  {
    $unwind: "$operated"
  },
  {
    $unwind: "$operated.company"
  },
  {
    $unwind: "$operated.company.job"
  },
  {
    $project: {
      _id: 0,
      jobType: "$operated.company.job.type"
    }
  },
```

```
    {
      $group: {
        _id: null,
        jobTypes: { $addToSet: "$jobType" }
      }
    },
    {
      $project: {
        _id: 0,
        jobTypes: 1
      }
    }
  ]);
```

Queries associated with Company: Q2, Q5 Selection attributes for Q2: {city, mv}

Selection attributes for Q5: {country, industryName}

The aggregate will remain the same...

Company: { name, marketValue, country, city, job_offers: [{job: {type}}], industryName }

Then MongoDB will add the _id:

{ _id, name, marketValue, country, city, job_offers: [{job: {type}}], industryName }

About indexes:

- Given that the intersection between selection attributes in Q2 and Q5 is empty: instead of having a single non-unique index on {city, mv, country, industryName} we choose to separetely support both queries by creating two separate non-unique indexes onto them:

```
    db.companies.createIndex({ city: 1, mv: 1 });
    db.companies.createIndex({ country: 1, industryName: 1 });
```

Then, after having thought about both queries, we create a mixed unique index for sharding and to then support the uniqueness of name with a compound index. This index has to be unique to support uniqueness across all shards.

```
    db.companies.createIndex({ city: 1, country: 1 },{ unique: true });
```

- A compound-unique index which contains the full shard key as a prefix of the index:

```
    db.companies.createIndex(
      { city: 1, country: 1, name: 1 },
      { unique: true }
    );
```

- Shard collection

```
    db.adminCommand({
      shardCollection: "db.Company",
      key: { city: 1, country: 1 }
    });
```

- Q2:

```
  db.companies.find({ city: "New York", mv: { $gt: 1000000 } }, { name: 1,
_id: 0 });
```

- Q5:

```
  db.companies.aggregate([
  {
    $match: {
      country: "Italy",
      industryName: "Technology"
    }
  },
  {
    $unwind: "$job_offers"
  },
  {
    $project: {
      _id: 0,
      jobType: "$job_offers.job.type"
    }
  },
  {
    $group: {
      _id: null,
      jobTypes: { $addToSet: "$jobType" }
    }
  },
  {
    $project: {
      _id: 0, // the grouping creates again an _id :(
      jobTypes: 1
    }
  }
]);
```

Queries associated with JobOffer: Q1, Q3, Q7 Selection attributes for Q1: {type, expire_date}

Selection attributes for Q3: {country, expire_date}

Selection attributes for Q7: {type, city, level}

The aggregate will remain the same...

JobOffer: { title, companyName, expire_date, type, country, city, marketValue, requires: [{skill: {level}}] }

Then MongoDB will add the _id:

{ _id, title, companyName, expire_date, type, country, city, marketValue, requires: [{skill: {level}}] }

About indexes:

- We observe that type appears in both Q1 and Q7 while expire_date in both Q1 and Q3: so we have an empty intersection but a partial overlap that can be leveraged to support multiple queries efficiently. Consequently, we can think about:
    - A composite index on {type, expire_date, city, level}: mongo uses a prefix for filtering so this will support both Q1 and Q7

```
db.jobOffers.createIndex({ type: 1, expire_date: 1, city: 1, level: 1
});
```

    - We can follow a similar approach to combine Q1 and Q3 obtaining an index on {expire_date, type, country}

```
db.jobOffers.createIndex({ expire_date: 1, type: 1, country: 1 });
```

    - Then, we think about having a compound unique index to enforce key constraint on the aggregate key {title, companyName}: given that type and expire_date are in partial overlap between queries we select them as a shard key. Here the unique indexes:

```
db.jobOffers.createIndex(
  { type: 1, expire_date: 1 },
  { unique: true }
);

db.jobOffers.createIndex(
  { type: 1, expire_date: 1, title: 1, companyName: 1 },
  { unique: true }
);
```

- Shard collection

```
db.adminCommand({
  shardCollection: "db.JobOffer",
```

```
  key: { type: 1, expire_date: 1 }
});
```

- Q1:

```
  db.jobOffers.find(
  {
    type: "Full-time",
    expire_date: { $lte: new Date(new Date().setDate(new Date().getDate() +
30)) }
  },
  {
    _id: 0
  }
);
```

- Q3:

```
  db.jobOffers.aggregate([
  {
    $match: {
      country: "Russia",
      expire_date: { $gt: new Date(new Date().setDate(new Date().getDate()
+ 60)) }
    }
  },
  {
    $project: {
      _id: 0,
      companyName: 1,
      marketValue: 1
    }
  },
  {
    $group: {
      _id: "$companyName", // Group by company name
      marketValue: { $first: "$marketValue" } // Collect the first market
value (one value per company!)
    }
  }
]);
```

Side note on this Q3: $first is used in MongoDB grouping to select the first encountered value of a field (e.g., marketValue) that is not part of the grouping key, providing a simple and efficient way to handle such fields without requiring computation.

- Q7:

```
  db.jobOffers.find(
  {
    type: "Internship",
    city: "Hamburg",
    "requires.skill.level": "Beginner"
  },
  {
    _id: 0,
    title: 1
  }
);
```

Design in Cassandra

7. Design in Cassandra: a. Design schema for Cassandra (including partition keys and indexes), starting from step 5, using the approaches/methodologies proposed in the course. b. Specify in CQL each operation of the workload.

Queries associated with Skill: Q6 Selection attributes for Q6: {score, type}

Skill: { name,score, provides: [{benefit: {type}}] }

- Given that we only have one query, we can safely select {score} as partition key, while for the primary key we have to add the aggregate key "name" in order to have the aggregate identifier. We obtain:
  - Partition key = {score}
  - Primary key = {score, name} with name as clustering column

However, we have still purposely ignored the selection on benefit type!

As an **important note**, we can't create a custom type because this would enforce the set to be frozen and therefore disallowing us to create an INDEX on provides.

- Here is the CREATE command in Cassandra:

```
CREATE TABLE Skills (
    name text,
    score int,
    provides set<text>,
    PRIMARY KEY (score, name)
);

CREATE INDEX ON Skills (provides);
```

- Q6:

```
SELECT name
FROM Skills
WHERE score = 71 AND provides CONTAINS '401(k)';
```

Queries associated with IndustryDomain: Q4 Selection attributes for Q4: {name}

IndustryDomain: { name, operated: [{ company: [{ job: {type} }] }] }

- Given that name is both the only selection attribute and the aggregate key we only have to select it as Partition Key!
  - Partition key = {name}
- Here are the CREATE commands in Cassandra:

```
CREATE TYPE job_t (
    type text
);

CREATE TYPE company_t (
    job list<frozen<job_t>>
);

CREATE TABLE IndustryDomain (
    name text PRIMARY KEY,
    operated list<frozen<company_t>>
);
```

- Q4:

```
SELECT operated
FROM IndustryDomain
WHERE name = 'Technology';
```

Queries associated with Company: Q2, Q5 Selection attributes for Q2: {city, mv}

Selection attributes for Q5: {country, industryName}

Company: { name, marketValue, country, city, job_offers: [{job: {type}}], industryName }

- Unluckily, the intersection between Q2 and Q5 selection attributes is empty: the only solution is to split the aggregate into two column-families!

```
CREATE TYPE job_t (
    type text
);

CREATE TABLE Company2 (
    city text,
    marketValue double,
    name text,
    country text,
    job_offers list<frozen<job_t>>,
```

```
    industryName text,
    PRIMARY KEY ((city, marketValue), name)
);

CREATE TABLE Company5 (
    city text,
    marketValue double,
    name text,
    country text,
    job_offers list<frozen<job_t>>,
    industryName text,
    PRIMARY KEY ((country, industryName), name)
);
```

Company2 will be used for executing Q2 and Company5 for executing Q5.

A further analysis can lead us to optimize, for example, C5 table by removing unnecessary fields (we still have C2 for other fields if required, so no needs to include extra field on a table that should be mainly tailored to allow the execution of query 5.

```
CREATE TABLE Company5 (
    country text,
    industryName text,
    name text,
    job_offers list<frozen<job_t>>,
    PRIMARY KEY ((country, industryName), name)
);
```

- Q2:

```
SELECT name
FROM Company2
WHERE city = 'New York' AND marketValue > 1000000.0;
```

- Q5:

```
SELECT job_offers
FROM Company5
WHERE country = 'Italy' AND industryName = 'Technology';
```

Queries associated with JobOffer: Q1, Q3, Q7 Selection attributes for Q1: {type, expire_date}

Selection attributes for Q3: {country, expire_date}

Selection attributes for Q7: {type, city, level}

JobOffer: { title, companyName, expire_date, type, country, city, marketValue, requires: [{skill: {level}}] }

- Given our partial overlap we can think to carefully design this solution. Here are two candidate pairings:
    - Option 1: Combine Q1 and Q3
        - Shared attribute: expire_date.
        - Partitioning by expire_date allows efficient filtering for both queries.
    - Option 2: Combine Q1 and Q7
        - Shared attribute: type.
        - Partitioning by type allows efficient filtering for both queries.

We could decide to opt for mixing Q1 and Q3 because expire_date is time-based and therefore with high selection factor (we have less types then expire_dates so makes sense to group the latter) --> We'll see later on that this approach brings some problems!

Also, as seen before, we opt for a list to allow the INDEX creation for Q7's level selection.

```
CREATE TABLE Job1_3 (
    expire_date DATE,
    type text,
    country text,
    title text,
    companyName text,
    city text,
    marketValue double,
    requires list<text>,
    PRIMARY KEY (expire_date, type, country, title, companyName)
);

CREATE TABLE Job7 (
    expire_date DATE,
    type text,
    country text,
    title text,
    companyName text,
    city text,
    marketValue double,
    requires list<text>,
    PRIMARY KEY ((type, city), title, companyName)
);

CREATE INDEX ON Job7 (requires);
```

A further analysis can lead us to optimize tables by removing unnecessary fields. Also, unfortunately, we see that based on Job1_3 we can't execute Q1 and Q3 without either creating secondary indexes or by ALLOW FILTERING (e.g expire_date is a range selection and therefore has to appear as the last). So we opt to further divide the two tables:

```
CREATE TABLE Job1 (
    expire_date DATE,
    type text,
    title text,
    companyName text,
    PRIMARY KEY (type, expire_date, title, companyName)
);

CREATE TABLE Job3 (
    expire_date DATE,
    country text,
    title text,
    marketValue double,
    companyName text,
    PRIMARY KEY (country, expire_date, title, companyName)
);

CREATE TABLE Job7 (
    type text,
    city text,
    title text,
    companyName text,
    requires list<text>,
    PRIMARY KEY ((type, city), title, companyName)
);

CREATE INDEX ON Job7 (requires);
```

- Q1:

```
SELECT *
FROM Job1
WHERE type = 'Full-time' AND expire_date <= toDate(now()) + 30;
```

- Q3:

```
SELECT companyName, marketValue
FROM Job3
WHERE country = 'Russia' AND expire_date > toDate(now()) + 60;
```

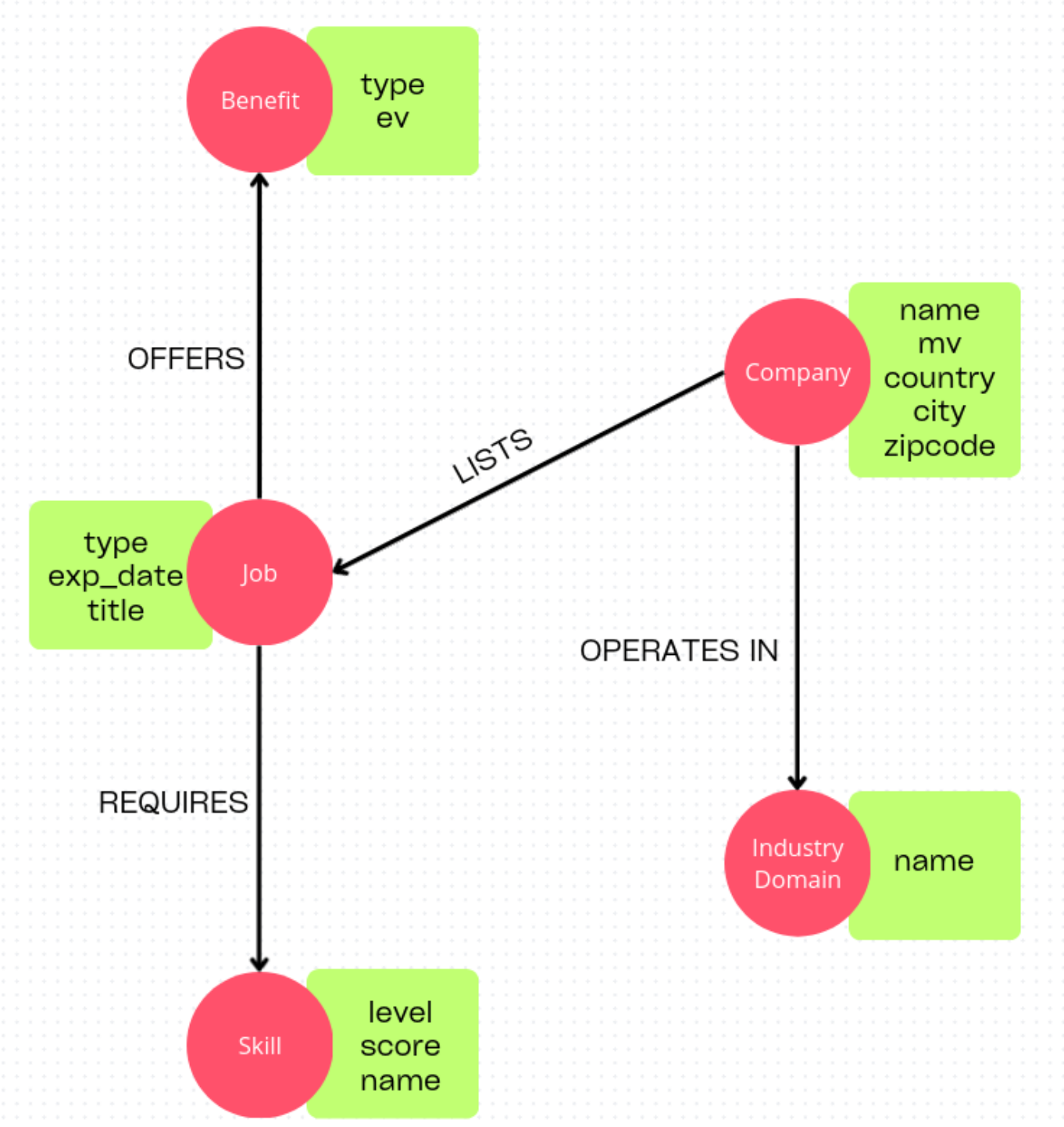
- Q7:

```
SELECT title
FROM Job7
WHERE type = 'Internship'
```

```
    AND city = 'Hamburg'
    AND requires CONTAINS 'Beginner';
```

Design in Neo4J

8. Design in Neo4J: a. Design a schema for Neo4j, using the approaches/methodologies proposed in the course. b. Specify in Neo4j each operation of the workload.



Query workload in Neo4J (from 1 to 7):

```
MATCH (j:Job)
WHERE j.type = 'Full-time'
```

```
  AND date(substring(j.exp_date, 0, 10)) <= date() + duration({days: 30})
RETURN j

MATCH (c:Company)
WHERE c.city = 'New York' AND c.mv > 1000000
RETURN c.name

MATCH (c:Company {country: 'Russia'})-[:LISTS]->(j:Job)
WHERE date(substring(j.exp_date, 0, 10)) <= date() + duration({days: 60})
RETURN DISTINCT c.name, c.mv

MATCH (id:Industry)<-[:OPERATES_IN]-(c:Company)-[:LISTS]->(j:Job)
WHERE id.name = 'Technology'
RETURN DISTINCT j.type

MATCH (id:Industry)<-[:OPERATES_IN]-(c:Company {country: 'Italy'})-
[:LISTS]->(j:Job)
WHERE id.name = 'Technology'
RETURN DISTINCT j.type

MATCH (s:Skill)<-[:REQUIRES]-(j:Job)-[:OFFERS]->(b:Benefit)
WHERE s.score > 70 AND b.type = '401(k)'
RETURN DISTINCT s.name

MATCH (j:Job {type: 'Internship'})-[:REQUIRES]->(s:Skill {level:
'Beginner'}),
      (j)<-[:LISTS]-(c:Company {city: 'Hamburg'})
RETURN j.title
```

Note: when it comes on choosing whether to put "WHERE" or leveraging "{...}" the decision relies on both flexibility and clarity. For complex conditions we chose WHERE, while for simple conditions of the "starting" node in the path we opted for a direct {...} inside the MATCH parenthesis.

System S choice

9. Discuss which, among the three systems, is the most suitable to be used as back-end for your application. Motivate your choice, taking into account the features of your application (step 2) and the identified workload (step 4). Let S be the selected system.

Given that our application is:

- predominanty read-intensive
- with high avaibility constraints
- some consistency constraints
- no need of batch-processing
- highly relational queries

We select **Neo4J** as system S for suitable backend for our application. This is driven by the following reasonings:

- The application is centered on relationships and traversals (e.g linking jobs to companies, industries, skills, and benefits), as graph databases benefits provide.
- Cypher queries easily handles our workload, as opposed to MongoDB.
- Our modelled graph is able to direclty reflect the conceptual schema, as opposed to Cassandra.
- Perfectly suitable for high avaibility and read-intensive data
- CA tailored

The only weakness is with respect to scalability of writes, but as we said the percentage of write operations is hugely lower than the write one (also, write operations are limited to little additions of new jobs when occasionaly they pop-up).

- We discarded MongoDB due to:
    - unintuitive relationships handling
    - not so optimized aggregation pipelines on most of the queries (due to them being highly relational)
- We discarded Cassandra due to:
    - Multiple denormalized tables, originated to a schema design that has to be tailored to specific queries (that present multiple disjoint scenarios with respect to selection attributes)
    - No query flexibility for future workload changes, highly probable in this dynamic world of jobs.

Neo4J: Configuration details

10. Provide details about the system configuration needed in system S for storing/processing your data according to the chosen application.

We need:

- Neo4J Aura instance to implement our system
- Query performance optimization by indexing when feasible (see (13))
- High RAM size for data traversal (Neo4J instance will be enough for our dataset's size)

If really implementing a scalable business solution, we would also need a clustering mechanisms to ensure high avaibility and causal consistency (let's remember that Neo4J is a **CA** system and it relies on causal consistency). Let's make an example configuration:

```
causal_clustering.enabled=true
causal_clustering.minimum_core_cluster_size_at_runtime=3
```

Then, when deploying Neo4j for a read-intensive workload with high availability and potential fault tolerance, the "R + W > N" rule becomes crucial: R (# of replicas that respond to read requests) and W (# of core nodes required for a successful write) must exceed the total number of N (core nodes in the cluster).

Keeping in mind that in Neo4J each write operation is replicated across core nodes to keep consistency we have to carefully choose values for R, W and N.

- As said before N = 3 (so with a fault tolerance of 1 node failure and therefore a minimum quorum of 2 nodes)
- We can set W = 2 (at minimum 2 nodes have to commit a write for it being successfull)

```
dbms.cluster.minimum_core_write_quorum=2
```

- We can set R = 2 (at least 2 replicas have to respond to that request)

```
causal_clustering.read_replica_count=2
```

- R + W in this scenario = 4, that indeed is higher then N = 3!

Then, to scale-up in our read-intensive application, we would need to increase the read replicas.

For example, the following might be a feasible *job board* scenario, in which the course concepts could be highly applied:

To scale up in our read-intensive application we would primarily focus on increasing the number of read replicas, which in Neo4j are dedicated nodes that serve only read queries, offloading the weight from the core nodes and allowing the system to handle significantly higher query volumes.

For **example**, in our job-board scenario mentioned above, the system must handle a high volume of user queries. Since the majority of these operations are read-only, the read replicas can efficiently handle these requests without impacting the performance of the core nodes.

So, in our previous example, as the number of users grows, we can step-by-step increase the number of read replicas depending on the query load; and since read replicas only need to synchronize with the core nodes periodically they can scale horizontally (and also being geographically replicated) without requiring changes to the core cluster configuration!

Neo4J: Schema details

11. Create the logical schema in system S

Given Neo4J schema-less nature **we don't have to provide any schema detail** (like in Cassandra for example, where we would have needed some table specifications). Here we just need to dynamically create nodes and their relationships, as we'll do in the next step. For simplicity, Jobs will be considered as merely identified by their title. Nevertheless, here follows a basic **example** on how the nodes should appear in our database insertion:

```
CREATE (:Industry {name: 'Data management'});
CREATE (:Company {name: 'Unige', mv: 'Data management', country: 'Italy',
city: 'Genova', zipcode: '16100'});
CREATE (:Job {type: 'Full-time', exp_date: '2025-01-01', title: 'Pitch
creator'});
CREATE (:Skill {name: 'Video editor', level: 'Advanced', score: 85});
CREATE (:Benefit {type: 'mark', ev: '110L'});
```

```
MATCH (j:Job {title: 'Pitch creator'}),
      (c:Company {name: 'Unige'}),
      (i:Industry {name: 'Data management'}),
      (s:Skill {name: 'Video editor'}),
      (b:Benefit {type: 'mark'})
CREATE (c)-[:LISTS]->(j);
CREATE (j)-[:OPERATES_IN]->(i);
CREATE (j)-[:REQUIRES]->(s);
CREATE (j)-[:OFFERS]->(b);
CREATE (c)-[:OPERATES_IN]->(i);
```

Neo4J: Graph details

12. Create an instance of your schema in the selected system, according to the logical schema just created.

We started from the linkedin-job-postings database and we continued by transforming it to meet our needs (removing unused fields and adding market value & economical value for queries enrichment).

These are our dataset links:

```
https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/company.csv

https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/job.csv

https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/benefit.csv

https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/industry.csv

https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/skill.csv

https://raw.githubusercontent.com/enriicola/Linked-adm-
In/refs/heads/main/data/relationships.csv
```

We noted that, when loading the last CSV file (for the relationships), Neo4J was very slow due to its unfeasability on handling batch processing.

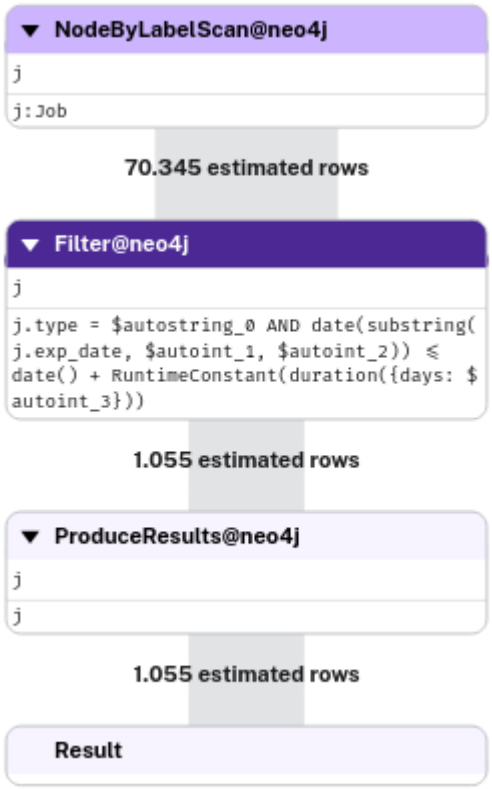
Neo4J: Workload implementation

13. Implement the workload in system S.

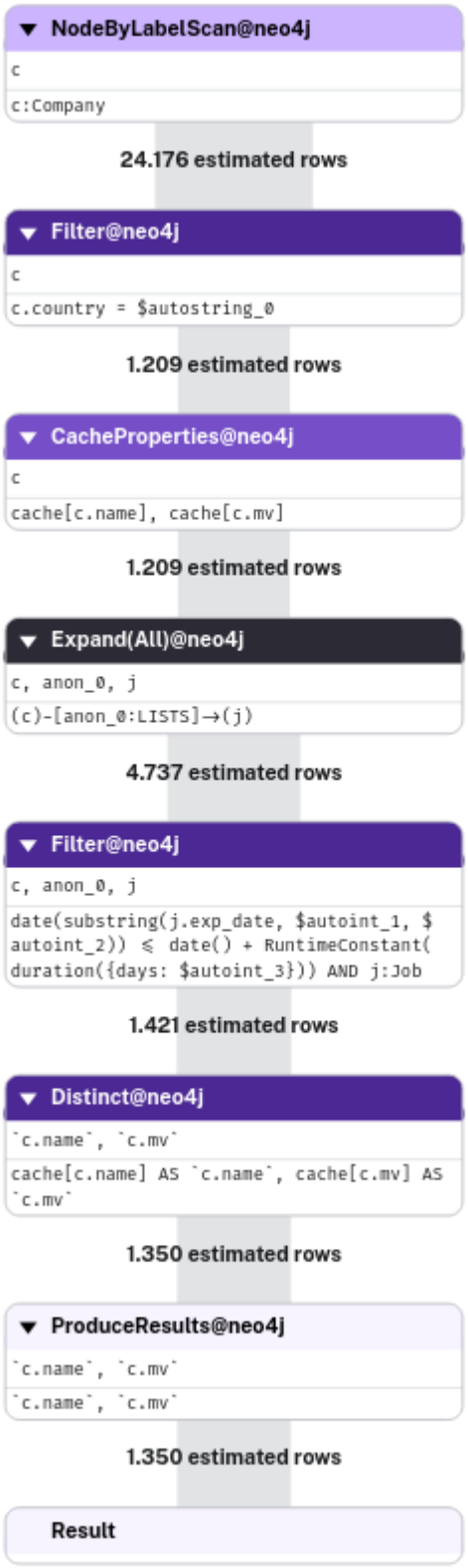
Here follows the workload implementation on Neo4J together with their **execution explanation**. We also decided to show some meaninfgul results with some indexes that could enhance systems' capabilities.

Once Neo4j has located the starting node, traversals through relationships are very efficient (O(1) basically). However... finding the starting node in a query is where indexes come into play!

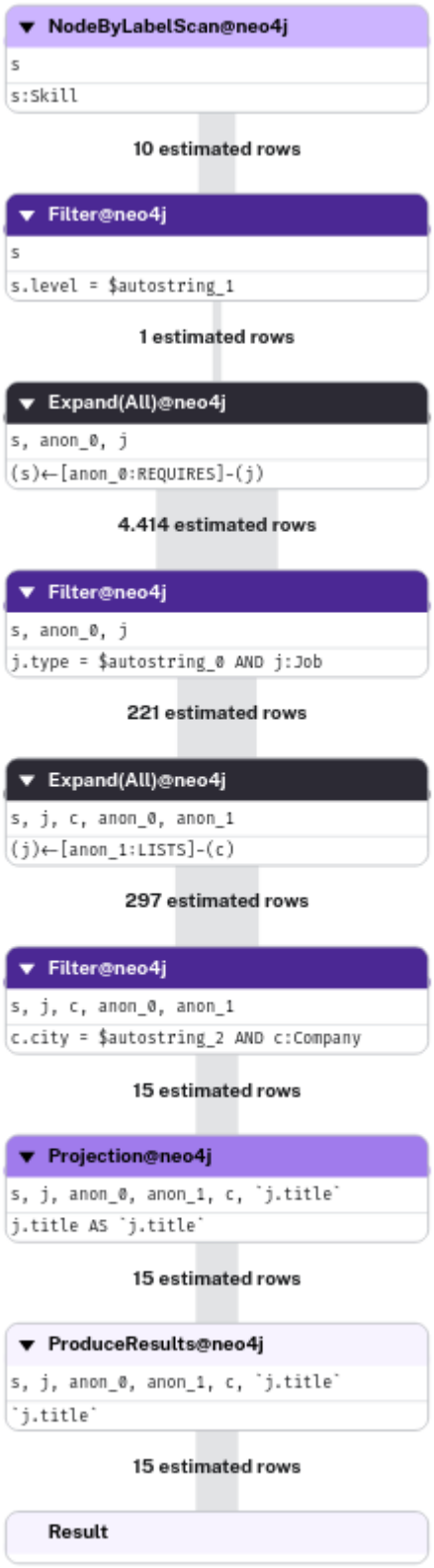
```
MATCH (j:Job)
WHERE j.type = 'Full-time'
  AND date(substring(j.exp_date, 0, 10)) <= date() + duration({days: 30})
RETURN j
```



```
MATCH (c:Company {country: 'Russia'})-[:LISTS]->(j:Job)
WHERE date(substring(j.exp_date, 0, 10)) <= date() + duration({days: 60})
RETURN DISTINCT c.name, c.mv
```



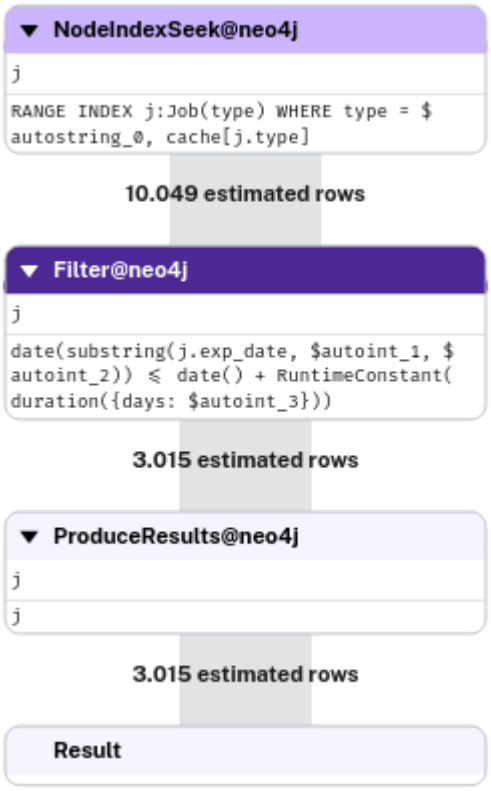
```
MATCH (j:Job {type: 'Internship'})-[:REQUIRES]->(s:Skill {level:
'Beginner'}),
      (j)<-[:LISTS]-(c:Company {city: 'Hamburg'})
RETURN j.title
```



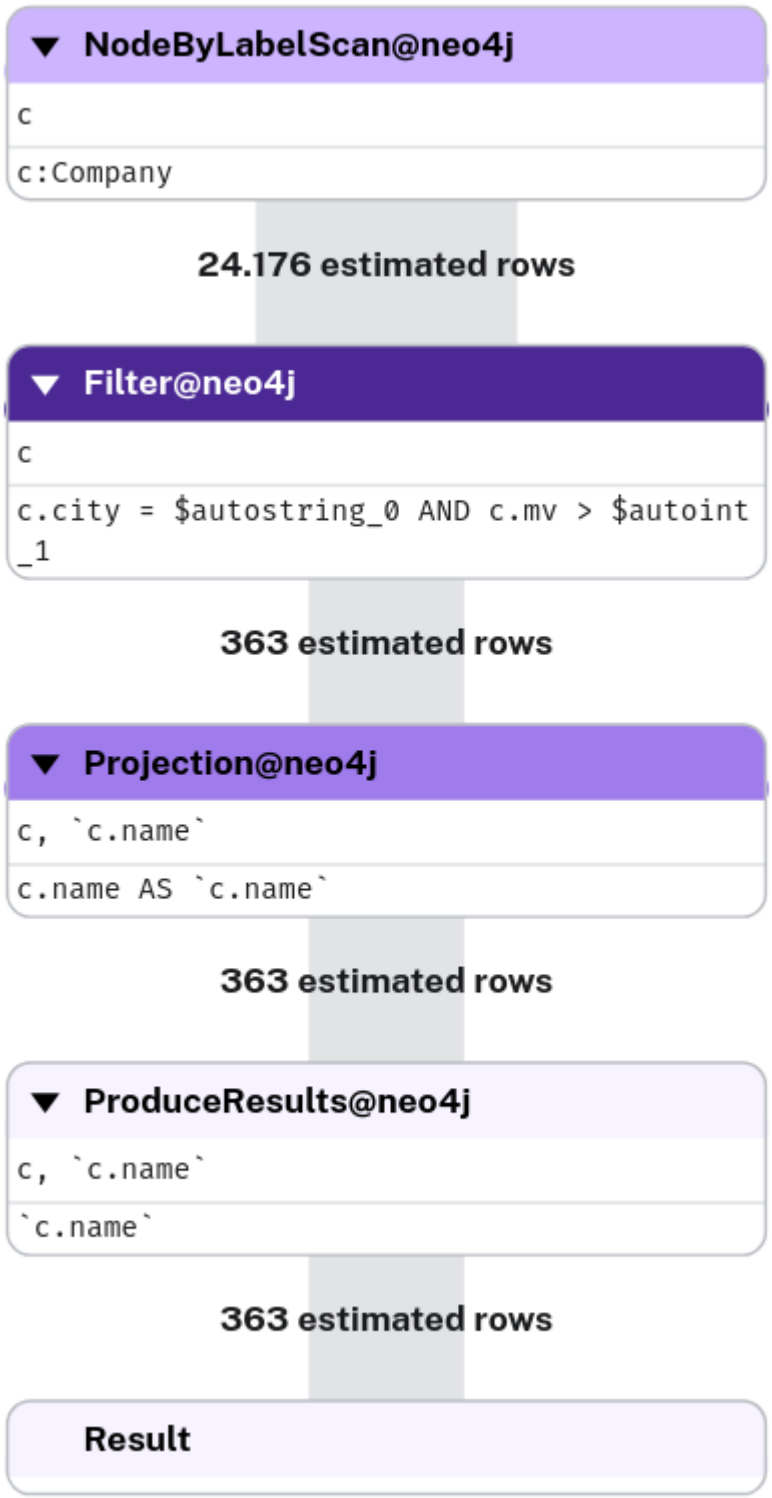
Let's put some indexes:

```
CREATE INDEX FOR (j:Job) ON (j.type);
CREATE INDEX FOR (j:Job) ON (j.exp_date);
```

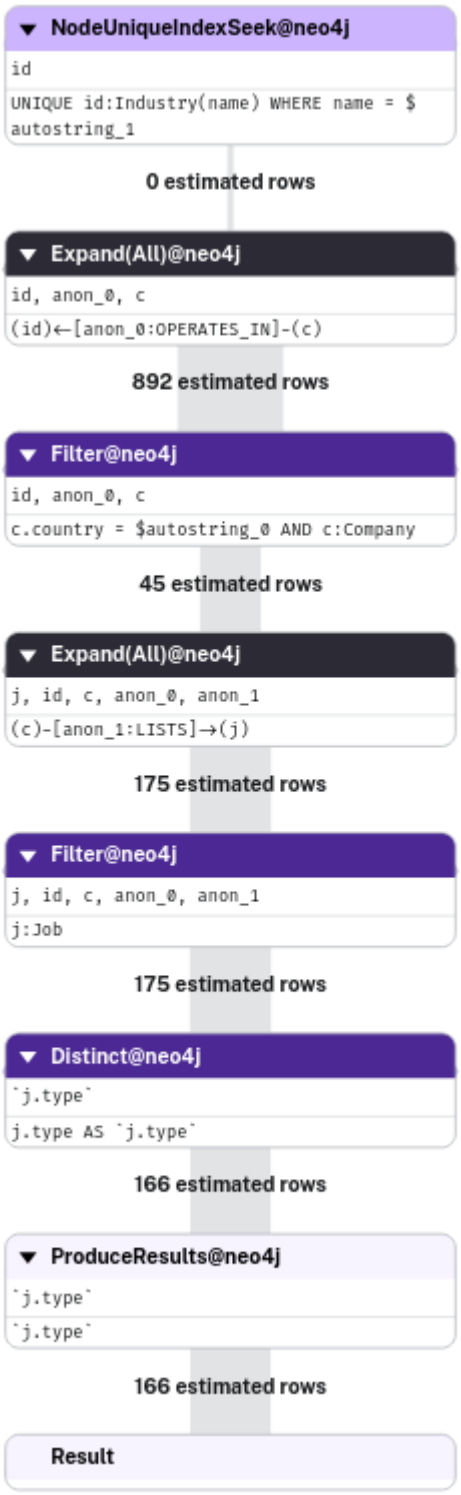
The first query changed its execution plan in the following way, while the other two remained unchanged. In our opinion this can be related to the fact that in the first query the index matched with all the selection attributes.



```
MATCH (c:Company)
WHERE c.city = 'New York' AND c.mv > 1000000
RETURN c.name
```



```
MATCH (id:Industry)<-[:OPERATES_IN]-(c:Company {country: 'Italy'})-
[:LISTS]->(j:Job)
WHERE id.name = 'Technology'
RETURN DISTINCT j.type
```

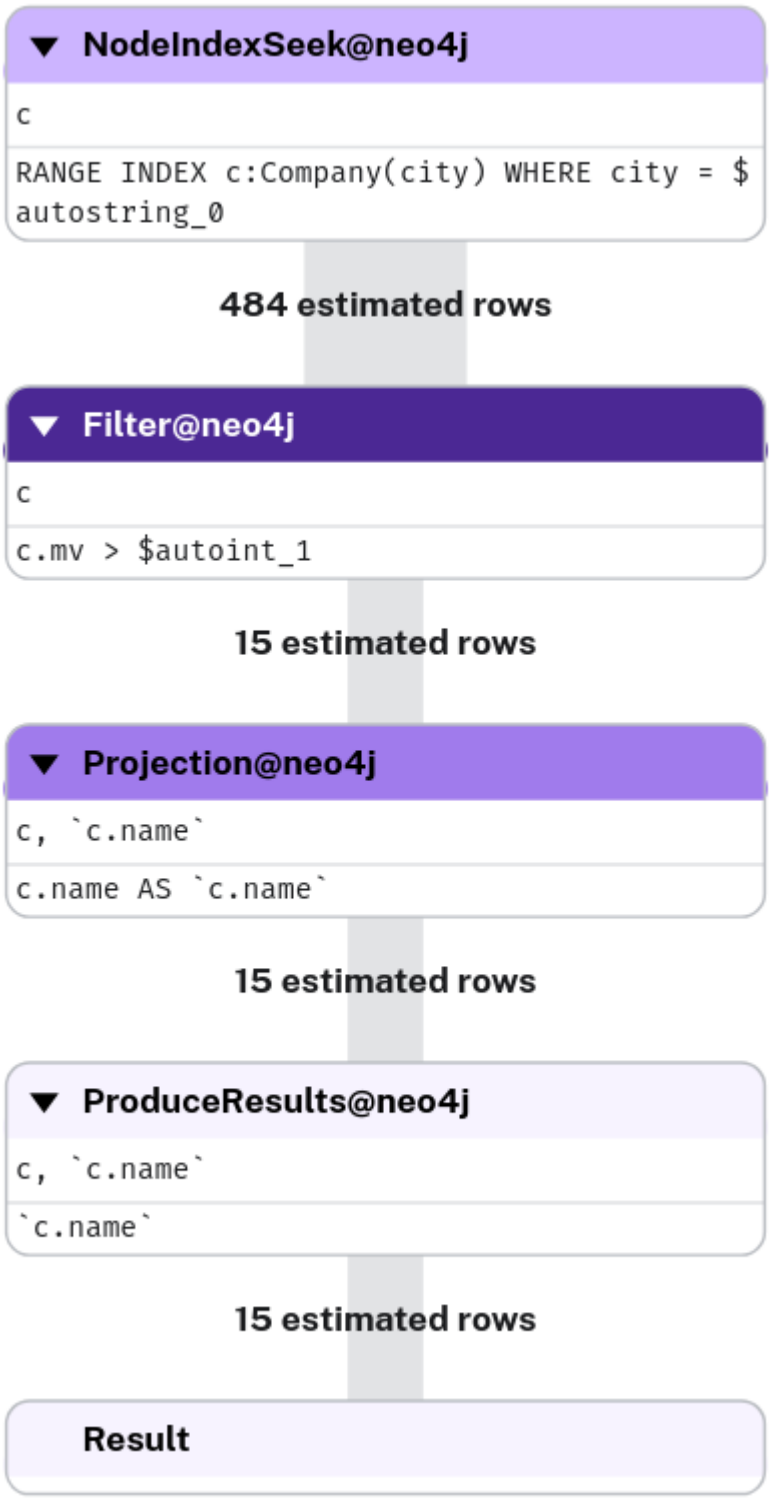


Let's put some indexes:

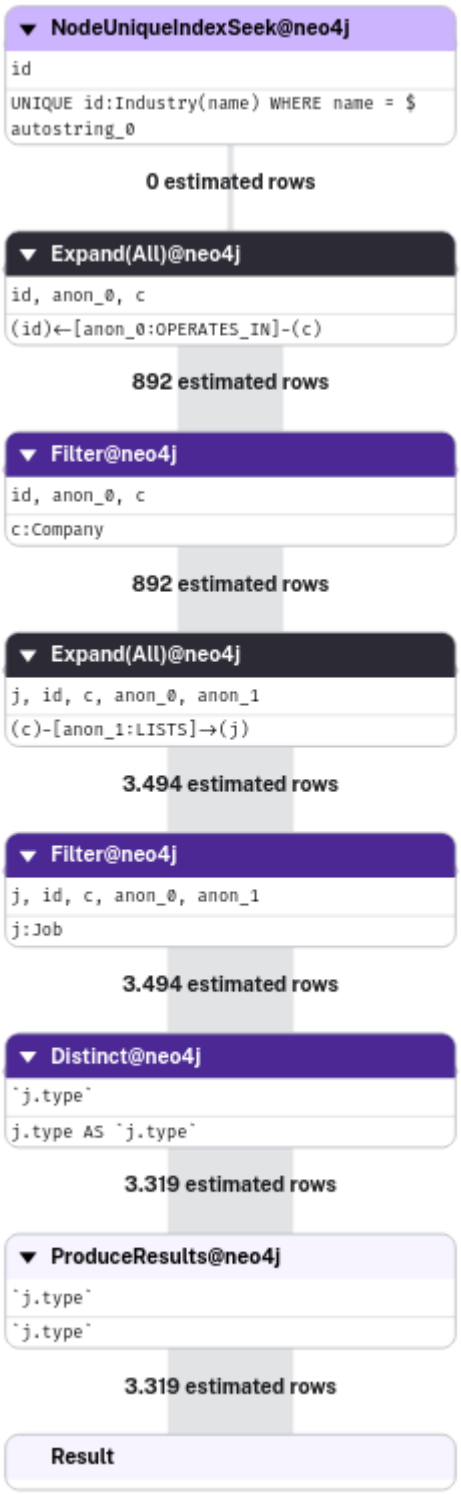
```
CREATE INDEX FOR (c:Company) ON (c.city);
CREATE INDEX FOR (c:Company) ON (c.mv);
CREATE INDEX FOR (c:Company) ON (c.country);
```

The first query changed its execution plan in the following way, while the other one remained unchanged. In our opinion this can be related to the fact that in the second query the automatically created unique

index on the primary key field "name" was indeed used (so no need to use the new ones).



```
MATCH (id:Industry)<-[:OPERATES_IN]-(c:Company)-[:LISTS]->(j:Job)
WHERE id.name = 'Technology'
RETURN DISTINCT j.type
```

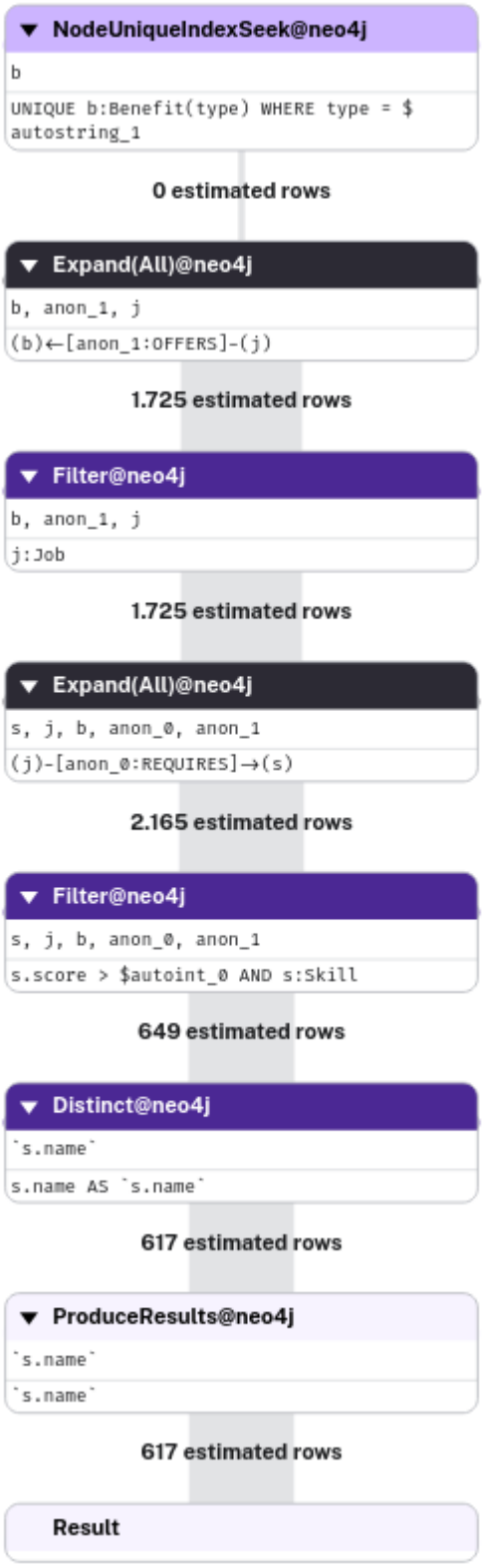


In this case the following index already exists by Neo4J, for the reasons explained before.

```
CREATE INDEX FOR (id:IndustryDomain) ON (id.name);
```

```
MATCH (s:Skill)<-[:REQUIRES]-(j:Job)-[:OFFERS]->(b:Benefit)
WHERE s.score > 70 AND b.type = '401(k)'
```

```
RETURN DISTINCT s.name
```



Let's put some indexes:

```
CREATE INDEX FOR (s:Skill) ON (s.score);
CREATE INDEX FOR (s:Skill) ON (s.level);
```

Again, these indexes are not used due to the benefit.type unique index being already present.

Model in RDFS / OWL the main classes and properties

14. Model in RDFS / OWL the main classes and the main properties corresponding to the entities and associations in the conceptual schema (step 3)

a. (Specify classes and properties) + for each property, specify the corresponding domain and range.

For simplicity, Jobs will be considered as merely identified by their title.

```
@prefix ex: <http://example.org/schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

# Classes:

ex:Job a rdfs:Class .
ex:Company a rdfs:Class .
ex:IndustryDomain a rdfs:Class .
ex:Skill a rdfs:Class .
ex:Benefit a rdfs:Class .

# (Subject to Object) Properties:

ex:belongsTo a rdf:Property ; rdfs:domain ex:Job ; rdfs:range ex:Company .
ex:requires a rdf:Property ; rdfs:domain ex:Job ; rdfs:range ex:Skill .
ex:offers a rdf:Property ; rdfs:domain ex:Job ; rdfs:range ex:Benefit .
ex:operatesIn a rdf:Property ; rdfs:domain ex:Company ; rdfs:range
ex:IndustryDomain .

# (Subject to Literal) Properties

# Job
ex:jobTitle a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:string .

ex:jobType a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:string .

ex:expireDate a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:date .

# Company
ex:companyName a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string .
```

```
ex:country a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string .

ex:city a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string .

ex:zipCode a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string .

ex:marketValue a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:decimal .

# Attributes for IndustryDomain
ex:industryName a rdf:Property ;
    rdfs:domain ex:IndustryDomain ;
    rdfs:range xsd:string .

# Skill
ex:skillName a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:string .

ex:skillLevel a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:string .

ex:skillScore a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:decimal .

# Benefit
ex:benefitType a rdf:Property ;
    rdfs:domain ex:Benefit ;
    rdfs:range xsd:string .

ex:economicalValue a rdf:Property ;
    rdfs:domain ex:Benefit ;
    rdfs:range xsd:decimal .
```

b. Express which classes are equivalent and which ones are disjoint.

In RDFS and OWL the semantics expresses that:

- **Open World Assumption**: The absence of a triple in a graph does not imply that the corresponding statement does not hold
- **No Unique Name Assumption**: differently named individuals can denote the same thing

Given that, we can for example express that "Job" and "JobOffer" are indeed the same concept in our domain:

```
ex:JobOffer a owl:Class ;
owl:equivalentClass ex:Job .
```

These are instead all disjoint specifications:

```
ex:Job owl:disjointWith ex:Company, ex:IndustryDomain, ex:Skill, ex:Benefit
.
ex:Company owl:disjointWith ex:IndustryDomain, ex:Skill, ex:Benefit .
ex:IndustryDomain owl:disjointWith ex:Skill, ex:Benefit .
ex:Skill owl:disjointWith ex:Benefit .
```

c. Specify (or add) at least an inverse property.

Let's repeat once again all the (Subject to Object) properties in order to define their possible inverse relationships:

```
ex:belongsTo a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range ex:Company ;
    owl:inverseOf ex:offersJob .

ex:requires a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range ex:Skill ;
    owl:inverseOf ex:isRequired .

ex:offers a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range ex:Benefit ;
    owl:inverseOf ex:isOffered .

ex:operatesIn a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range ex:IndustryDomain ;
    owl:inverseOf ex:hasCompany .
```

For example we can have:

```
ex:offersJob a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range ex:Job ;
    owl:inverseOf ex:belongsTo .
```

d. For all the modeled properties, specify whether they are functional (or inverse functional).

- The only Class Property to be functional is belongsTo, because if a Job belongs to CompanyA and CompanyB; these two Companies are indeed the same Company.
- There are no Inverse Functional Class Properties
- All the Literal Properties are functional (there aren't any entities that can have multiple literal values, e.g a single Job Offer can't have two different expire dates, therefore if a Job has two expire_dates they're indeed the same expire_date).
- Only the PRIMARY KEY literals are also inverse functional (e.g if two Companies have the same name, the two Companies are indeed the same Company).

```
# Subject to Object Properties
ex:belongsTo a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range ex:Company ;
    a owl:FunctionalProperty .

# Subject to Literal Properties - Job
ex:jobTitle a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty, owl:InverseFunctionalProperty .

ex:jobType a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty .

ex:expireDate a rdf:Property ;
    rdfs:domain ex:Job ;
    rdfs:range xsd:date ;
    a owl:FunctionalProperty .

# Subject to Literal Properties - Company
ex:companyName a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty, owl:InverseFunctionalProperty .

ex:country a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty .

ex:city a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty .

ex:zipCode a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:string ;
```

```
    a owl:FunctionalProperty .

ex:marketValue a rdf:Property ;
    rdfs:domain ex:Company ;
    rdfs:range xsd:decimal ;
    a owl:FunctionalProperty .

# Attributes for IndustryDomain
ex:industryName a rdf:Property ;
    rdfs:domain ex:IndustryDomain ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty, owl:InverseFunctionalProperty .

# Subject to Literal Properties - Skill
ex:skillName a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty, owl:InverseFunctionalProperty .

ex:skillLevel a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty .

ex:skillScore a rdf:Property ;
    rdfs:domain ex:Skill ;
    rdfs:range xsd:decimal ;
    a owl:FunctionalProperty .

# Subject to Literal Properties - Benefit
ex:benefitType a rdf:Property ;
    rdfs:domain ex:Benefit ;
    rdfs:range xsd:string ;
    a owl:FunctionalProperty, owl:InverseFunctionalProperty .

ex:economicalValue a rdf:Property ;
    rdfs:domain ex:Benefit ;
    rdfs:range xsd:decimal ;
    a owl:FunctionalProperty .
```

Model RDF instances

15. Model in RDF some instances to populate your schema. In addition: a. Relate instances to the corresponding class or property. b. Clarify which individuals are identical and which ones are different.

```
ex:SwEng a ex:Job ;
    ex:jobTitle "Software Engineer" ;
    ex:jobType "Full-time" ;
    ex:expireDate "2025-02-15"^^xsd:date ;
    ex:belongsTo ex:GueCorp ;
```

```
    ex:requires ex:Pitch, ex:SQL ;
    ex:offers ex:HI, ex:TaxBenefit .

ex:GueCorp a ex:Company ;
    ex:companyName "GuerriniCorp" ;
    ex:country "Italy" ;
    ex:city "Genoa" ;
    ex:zipCode "16165" ;
    ex:marketValue "80000000.00"^^xsd:decimal ;
    ex:operatesIn ex:Creativity .

ex:Creativity a ex:IndustryDomain ;
    ex:industryName "Creativity" .

ex:Pitch a ex:Skill ;
    ex:skillName "Elevator Pitch" ;
    ex:skillLevel "Intermediate" ;
    ex:skillScore "85.0"^^xsd:decimal .

ex:SQL a ex:Skill ;
    ex:skillName "SQL" ;
    ex:skillLevel "Advanced" ;
    ex:skillScore "90.0"^^xsd:decimal .

ex:HI a ex:Benefit ;
    ex:benefitType "Health Insurance" ;
    ex:economicalValue "5000.00"^^xsd:decimal .

ex:TaxBenefit a ex:Benefit ;
    ex:benefitType "401(k)" ;
    ex:economicalValue "3000.00"^^xsd:decimal .
```

Let's now identify which instances are the same and which are different among each other.

For example, if I were to add a "SoftwareEng" node, for the same reason of the "Paris-Parigi" example discussed in class, it has to be referred as being the same as SwEng!

```
ex:SoftwareEng a ex:Job ;
    ex:jobTitle "Software Engineer" ;
    ex:jobType "Full-time" ;
    ex:expireDate "2025-02-15"^^xsd:date ;
    ex:belongsTo ex:GueCorp ;
    ex:requires ex:Pitch, ex:SQL ;
    ex:offers ex:HI, ex:TaxBenefit .

ex:SwEng owl:sameAs ex:SoftwareEng .
```

Regarding the "differentFrom" constraint, since we have already defined that our classes are distinct from each other using the "disjointWith" constraint, this information is implicitly inferred. Therefore, we only need to specify that nodes within the same class are distinct from one another:

```
ex:Pitch owl:differentFrom ex:SQL .
ex:HI owl:differentFrom ex:TaxBenefit .
```

SPARQL queries

6. Specify in SPARQL at least 3 queries to be executed over the defined RDF dataset. The requests should: a. be structurally different (i.e., each of them should contain different constructs) b. include at least one CONSTRUCT query c. refer as much as possible to the requests included in the workload specified in PART II.

For the **first** query we refer to our Query #1 and we decide to translate it in SPARQL paying attention on how to filter the date attribute:

```
SELECT ?jobTitle
WHERE {
  ?job a ex:Job ;
       ex:jobType "Full-time" ;
       ex:expireDate ?expireDate ;
       ex:jobTitle ?jobTitle .
  FILTER (xsd:date(?expireDate) <= xsd:date(NOW()) + "P30D"^^xsd:duration)
}
```

For the **second** query we refer to our Query #4 so that we can introduce the DISTINCT construct (differently from the previous query we're returning non-unique values):

```
SELECT DISTINCT ?jobType
WHERE {
  ?company ex:operatesIn ?industry ;
           ex:belongsTo ?job .
  ?industry a ex:IndustryDomain ;
            ex:industryName "Technology" .
  ?job ex:jobType ?jobType .
}
```

For the **third** query we want to use (as the assignment requires) the CONSTRUCT keyword:

Construct a graph with all jobs that require skills with a level of Beginner.

```
CONSTRUCT {
  ?job a ex:Job ;
       ex:requires ?skill .
}
WHERE {
  ?job a ex:Job ;
       ex:requires ?skill .
```

```
  ?skill ex:skillLevel "Beginner" .
}
```

We note that this redundancy inside both CONSTRUCT and WHERE constructs is actually a feature of SPARQL, which allows the user to specify different conditions with respect to the final graph that will be indeed produced!

For a **fourth** query we try to specify a query that involves both OPTIONAL and UNION clauses:

Retrieve jobs with their associated skills, if available, where jobs either belong to companies in Italy OR operate in the Technology industry.

```
SELECT ?jobTitle ?skillName
WHERE {
  {
    {
      ?job a ex:Job ;
           ex:belongsTo ?company ;
           ex:jobTitle ?jobTitle .
      ?company ex:country "Italy" .
    }
    UNION
    {
      ?job a ex:Job ;
           ex:belongsTo ?company ;
           ex:jobTitle ?jobTitle .
      ?company ex:operatesIn ?industry .
      ?industry ex:industryName "Technology" .
    }
  }

  OPTIONAL {
    ?job ex:requires ?skill .
    ?skill ex:skillName ?skillName .
  }
}
```

We end with a **fifth** query to also cover the "ASK" construct:

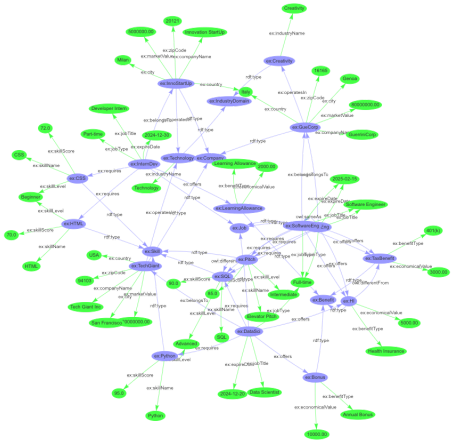
Check if any job requires a skill with a level of Beginner.

```
ASK {
  ?job a ex:Job ;
       ex:requires ?skill .
  ?skill ex:skillLevel "Beginner" .
}
```

Let's try 14-16 content in RDF playground!

17. Check the correctness of the proposed RDF dataset, extended with RDFS /OWL constraints, and of the proposed SPARQL queries using RDF playground (http://rdfplayground.dcc.uchile.cl/) or any other RDF data store at your choice.

We "played" into the "play"ground and verified the correctness of all classes, properties, constraints and SPARQL queries. For obtaining a more meaningful and enriched graph we also added some other instances. You can find the whole dataset at this link: (https://raw.githubusercontent.com/enriicola/Linked-adm-In/refs/heads/main/SPARQL-data-and-queries.ttl)



Below, for reference, there are results of the queries for the above dataset:

```
# --- 1 ---
--------------------
| jobTitle         |
====================
| "Data Scientist" |
--------------------

# --- 2 ---
-----------
| jobType |
===========
-----------

# --- 3 ---
(graph
  (triple ex:InternDev ex:requires ex:HTML)
  (triple ex:InternDev ex:requires ex:CSS)
  (triple ex:InternDev rdf:type ex:Job)
)

# --- 4 ---
-----------------------------------------
| jobTitle            | skillName       |
=========================================
| "Developer Intern"  | "CSS"           |
| "Developer Intern"  | "HTML"          |
| "Software Engineer" | "SQL"           |
```

```
| "Software Engineer" | "Elevator Pitch" |
| "Software Engineer" | "Elevator Pitch" |
| "Software Engineer" | "SQL"            |
| "Developer Intern"  | "CSS"            |
| "Developer Intern"  | "HTML"           |
| "Data Scientist"    | "SQL"            |
| "Data Scientist"    | "Python"         |
----------------------------------------

# --- 5 ---
yes
```

Presentation Link 😀

https://docs.google.com/presentation/d/10JpM2nPsat2lPP40ubgr755MYcMWq2DNsRzwk-B55IA/edit?usp=sharing