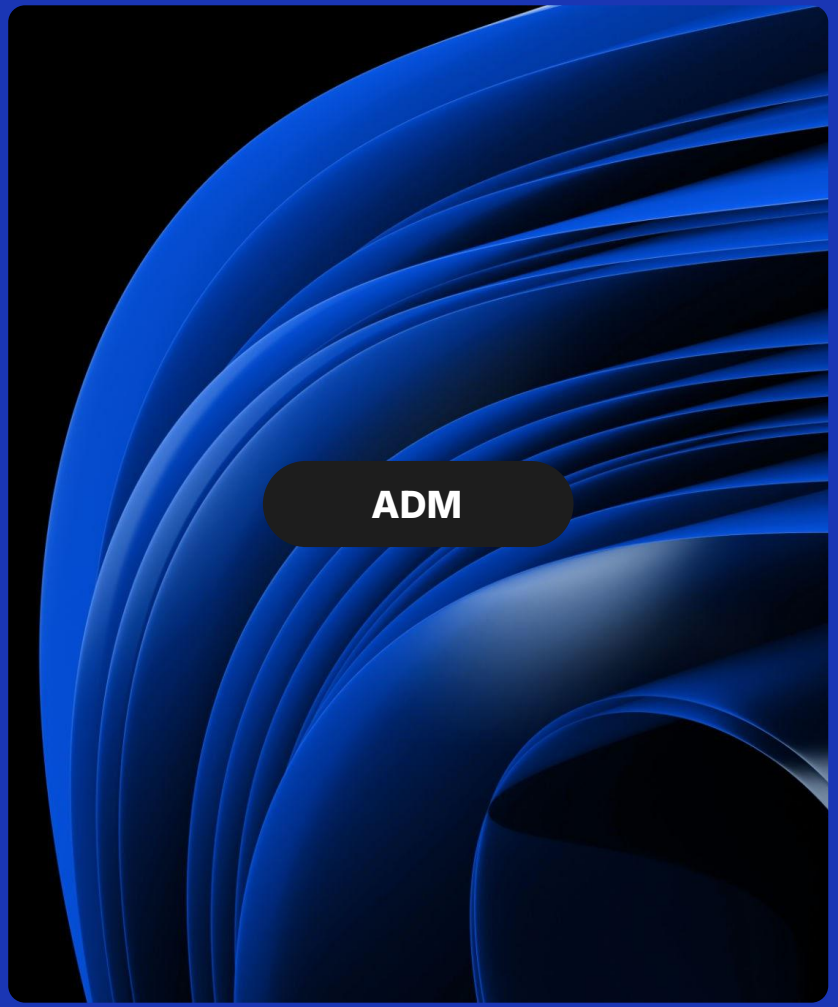
UniGe

Linked-ADM-In

Lorenzo Foschi, Enrico Pezzano
December 2024



The dataset 💾

LinkedIn Job Postings (2023 - 2024)

A Snapshot Into the Current Job Market

Data Card | Code (30) | Discussion (28) | Suggestions (0)

About Dataset

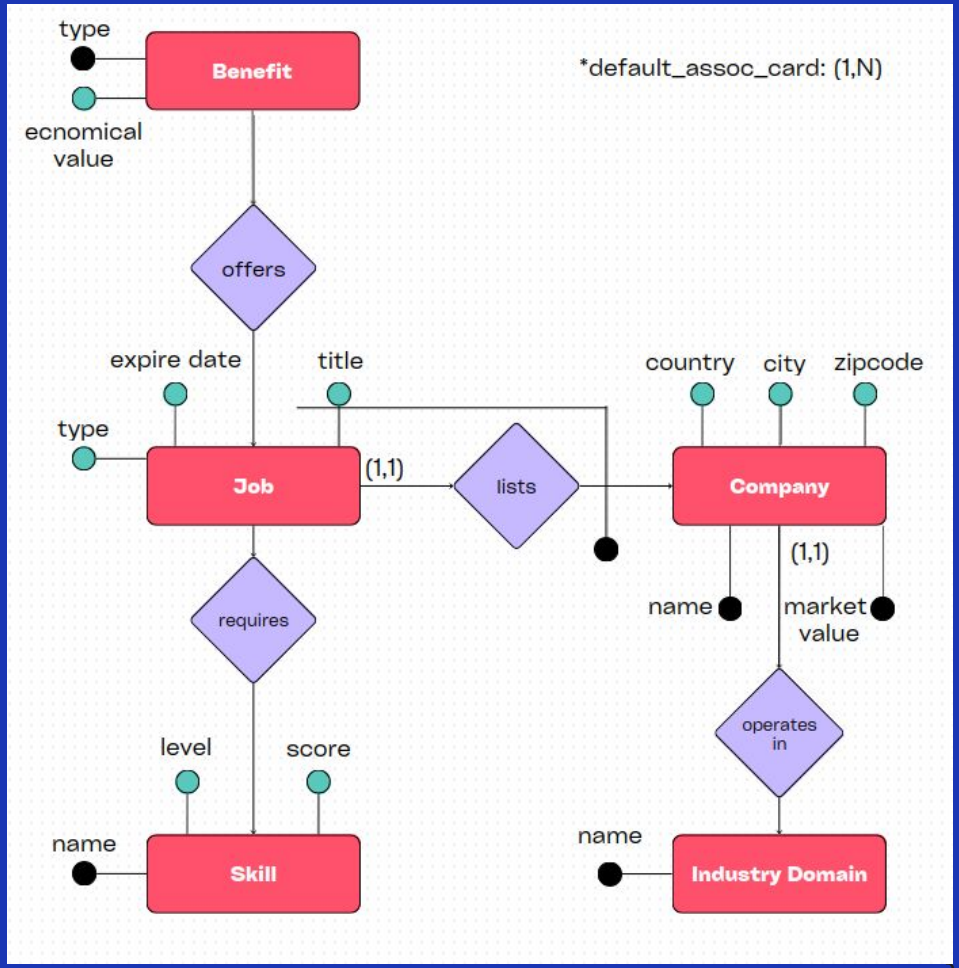
Description

Scraper Code - `https://github.com/ArshKA/LinkedIn-Job-Scraper`

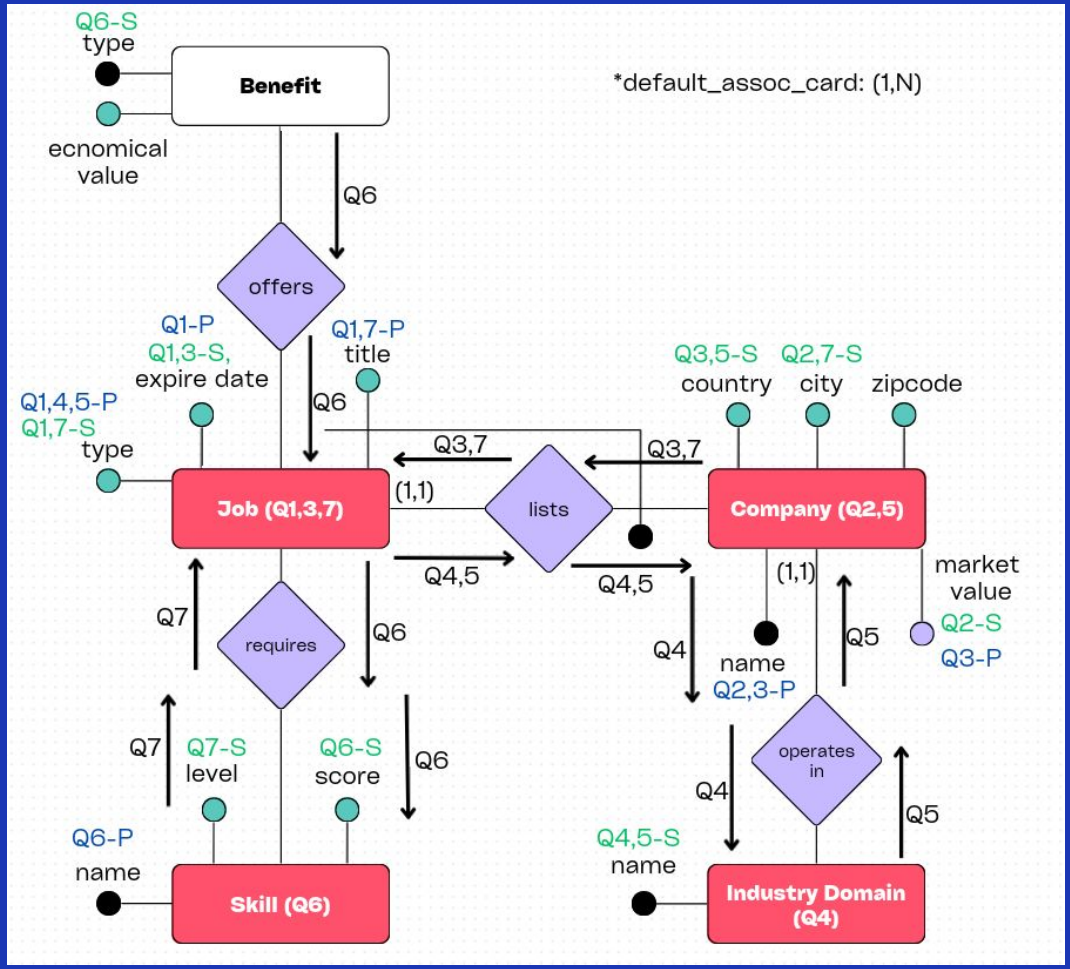
Every day, thousands of companies and individuals turn to LinkedIn in search of talent. This dataset contains a nearly comprehensive record of **124,000+ job postings** listed in 2023 and 2024. Each individual posting contains **dozens of valuable attributes for both postings and companies**, including the title, job description, salary, location, application URL, and work-types (remote, contract, etc), in addition to separate files containing the benefits, skills, and industries associated with each posting. The majority of jobs are also linked to a company, which are all listed in another csv file containing attributes such as the company description, headquarters location, and number of employees, and follower count.

With so many datapoints, the potential for exploration of this dataset is vast and includes exploring the highest compensated titles, companies, and locations; predicting salaries/benefits through NLP; and examining how industries and companies vary through their internship offerings and benefits. Future updates will permit further exploration into time-based trends, including company growth, prevalence of remote jobs, and demand of individual job titles over time.

Conceptual model



Annotated ER



Design in MongoDB

mongoDB

- **Indexing**: use non-unique and compound indexes to optimize queries and ensure uniqueness where needed
- **Sharding**: distribute data for scalability: remembering that the unique index must contain the full shard key as prefix of the index!
- **Query Optimization**: efficient aggregation with *$unwind*,
- **Many choices:** MongoDB is flexible... but this leads to the freedom of making many choices. We did them!

Design in Cassandra

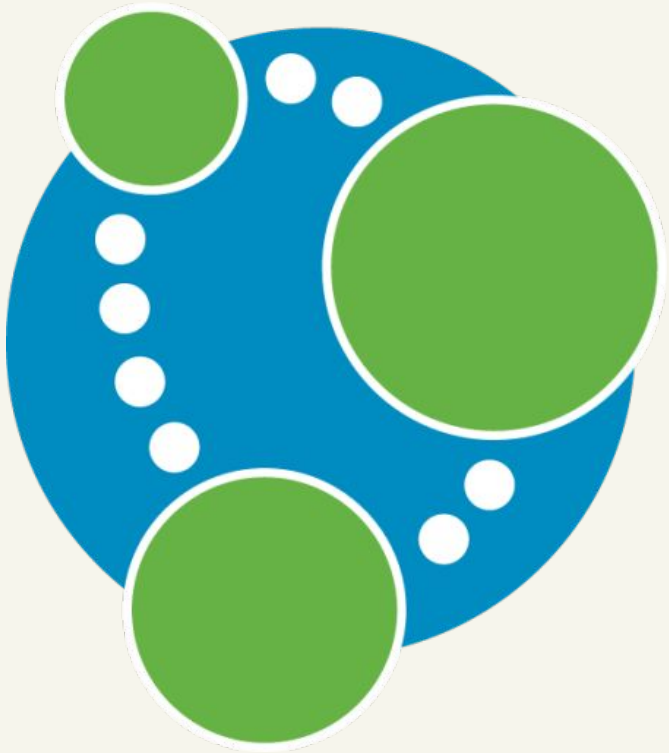
- **Partitioning**: Queries drive partition key design
- **Schema Design**: use clustering columns for unique row identification; split data into multi-tables when needed.
- **Query Optimization**: Try to combine shared query attributes (e.g Job1_3 for Q1 and Q3) and remove unnecessary fields for query-specific tables (e.g., Job7).
- **Customization tailored schemas** for efficient filtering and aggregation to align with query requirements.

Design in Neo4J

- **Query Examples:**

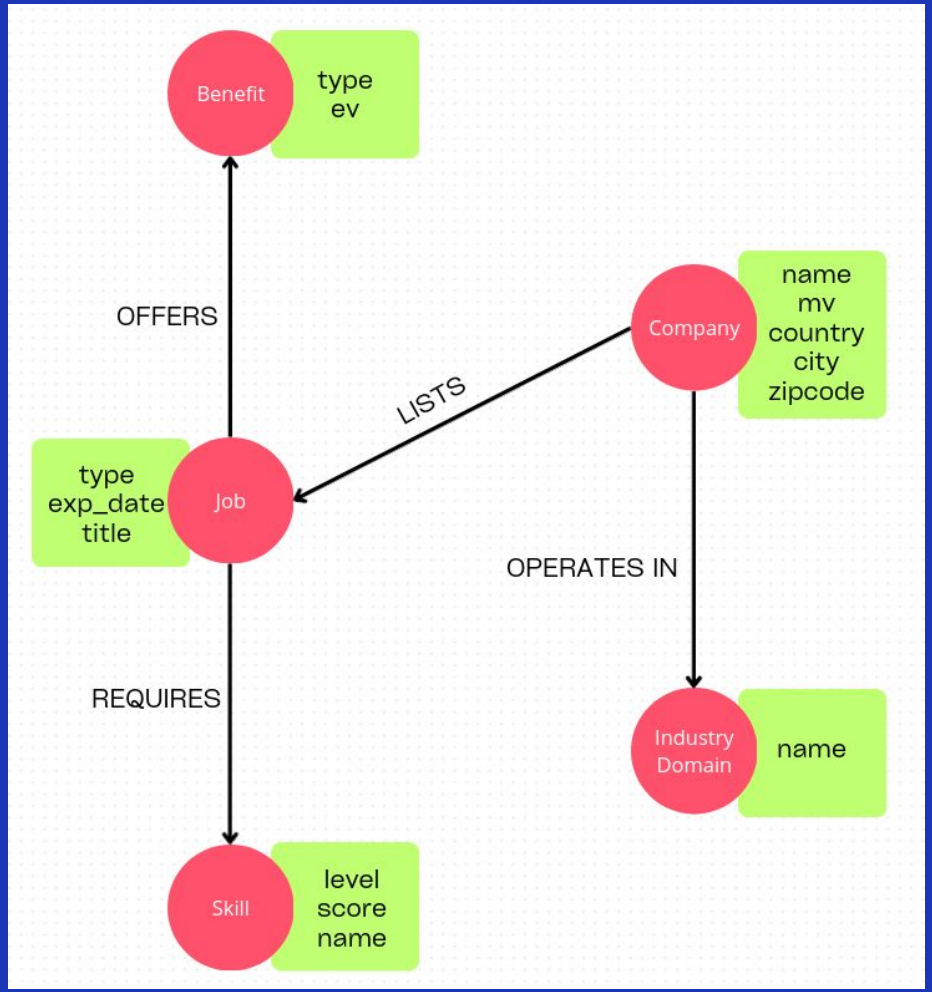
```
MATCH (j:Job {type: 'Internship'})-[:REQUIRES]→(s:Skill
{level: 'Beginner'}),
    (j)←[:LISTS]-(c:Company {city: 'Hamburg'})
RETURN j.title

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

- **Key Benefit:** Simplicity in definition of queries + Seamless traversal and querying of interconnected data.



Our choice!

neo4j



Neo4J reasoning 🧠

- Perfect for **relationship-centric** applications.
- **Cypher** simplifies complex relational and traversal tasks
- **Schema Alignment**: directly reflects the conceptual schema, avoiding denormalization required in Cassandra.
- High availability and consistency (**CA**).
- **Tradeoff**: write scalability is limited, but this is acceptable due to infrequent write operations.
- **R+W>N** rationales!

Neo4J reasoning 🧠

Why not MongoDB?

- Poor handling of complex relationships.
- Aggregation pipelines not suited for highly relational queries.

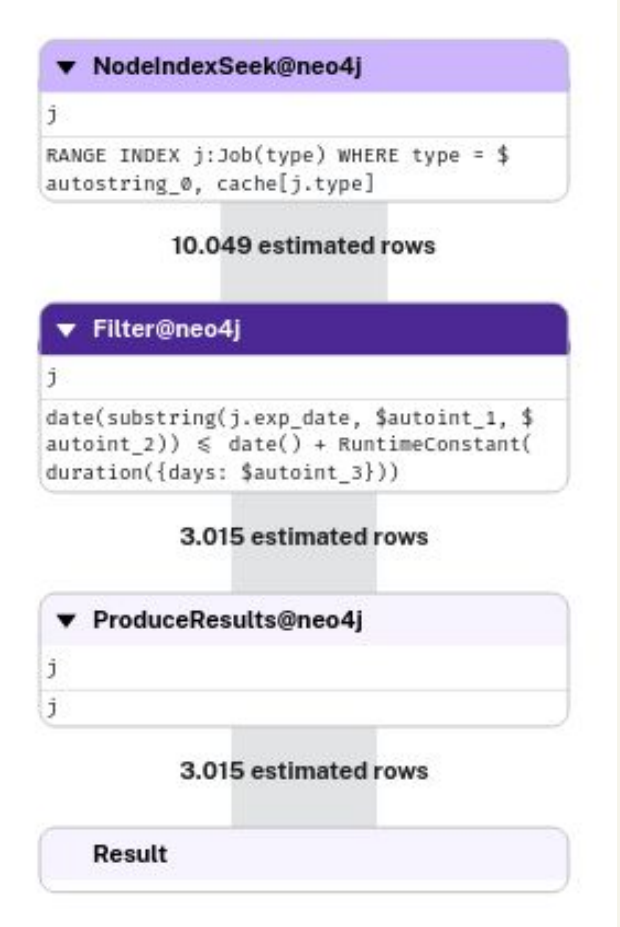
Why Not Cassandra?

- Requires multiple denormalized tables tailored to specific queries.
- Limited flexibility for adapting to changing query patterns.

Neo4J results ⛑️

- Use of indexes if primary keys
  - Due to previous UNIQUE CONSTRAINTs

- **N4J used indexes 2 times out of 7**:
  - 2 times has preferred unique indexes;
  - 1 time there was already that specific index;
  - 2 times has preferred the vanilla search due to the query having lots of filters on different attributes



Model in RDFS/OWL

- **Classes like:** ex:Job a rdfs:Class .
- **S-O properties like:**
  - ex:belongsTo a rdf:Property ;
    rdfs:domain ex:Job ; rdfs:range ex:Company .
- **S-L properties like:**
  - ex:jobTitle a rdf:Property ;
    rdfs:domain ex:Job ; rdfs:range xsd:string .

Only the belongsTo class property is **functional**, there are no **inverse functional** class properties, all literal properties are functional, and primary key literals are also inverse functional (e.g., two entities sharing the same primary key are identical).

Model in RDFS/OWL

Instances modeled:

- Job: *Software Engineer*
- Company: *GuerriniCorp*
- IndustryDomain: *Creativity*
- Skill: *SQL*
- Benefit: *Health Insurance*

sameAs & differentFrom:

- 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:Pitch owl:differentFrom ex:SQL .

SPARQL

- **We tried to use many constructors!**
  - SELECT: Retrieve specific attributes (e.g., job titles)
  - CONSTRUCT: Generate new graphs (e.g., jobs requiring beginner skills).
  - ASK: Verify existence of specific data patterns.
  - Use of *FILTER* for date-based conditions.
  - *DISTINCT* ensures non-redundant results.
  - We also added a query with UNION and OPTIONAL!

SEMANTIC DATASET