Linked-adm-In

Final project for the Advanced Data Mangaement course

Assignment Steps:

- [x] 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).
- [x] 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).
- [x] 3. Design a conceptual schema for the identified domain. The schema should include at least three associations.
- [x] 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.
- [x] 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.
- [x] 6. Design in MongoDB:
    - [x] a. Design a schema for MongoDB (including partition keys and indexes), starting from (step 5), using the approaches/methodologies proposed in the course.
    - [x] b. Specify each operation of the workload in the language supported by MongoDB
- [x] 7. Design in Cassandra:
    - [x] a. Design a schema for Cassandra (including partition keys and indexes), starting from (step 5), using the approaches/methodologies proposed in the course.
    - [x] b. Specify in CQL each operation of the workload.
- [x] 8. Design in Neo4J:
    - [x] a. Design a schema for Neo4j, using the approaches/methodologies proposed in the course.
    - [x] b. Specify in Neo4j each operation of the workload.
- [x] 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.
- [ ] 10. Provide details about the system configuration needed in system S for storing/processing your data according to the chosen application.
- [ ] 11. Create the logical schema in system S.
- [ ] 12. Create an instance of your schema in the selected system, according to the logical schema just created. To this aim:
    - [ ] a. You can use either an already available dataset or a synthetic one but we encourage the first option (it might be difficult to synthetically generate a relevant dataset for your reference application). The dataset should have a reasonable size (few Mb).
    - [ ] b. Notice that selected datasets might need to be transformed in order to be used by your application. For dataset transformation, you can rely on either data transformation tools, such as Tableaux Prep (www.tableau.com), Apache Superset (superset.apache.org) Trifacta (

www.trifacta.com), or other ETL tools such as Talend (www.talend.com), or scripts in your favorite language.
    - ☐ c. For importing datasets in the chosen system, you should refer to the available documentation for the system you have selected (e.g. https://www.datastax.com/dev/blog/simple-data-importing-and-exporting-with-cassandra for Cassandra and https://neo4j.com/developer/guide-importing-data-and-etl/perneo4J for Neo4J).
- ☐ 13. Implement the workload in system S.
- ☐ 14. Model in RDFS / OWL the main classes and the main properties corresponding to the entities and associations in the conceptual schema (step 3). In addition:
    - ☐ a. For each property, specify the corresponding domain and range.
    - ☐ b. Express which classes are equivalent and which ones are disjoint.
    - ☐ c. Specify (or add) at least an inverse property.
    - ☐ d. For all the modeled properties, specify whether they are functional (or inverse functional).
- ☐ 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.
- ☐ 16. 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.
- ☐ 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.

dataset link

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

project description link

https://2024.aulaweb.unige.it/mod/page/view.php?id=56196

Nature of the Proposed Application

(1-2) The proposed application is a job and industry relationship analysis tool. Its features and requirements include:

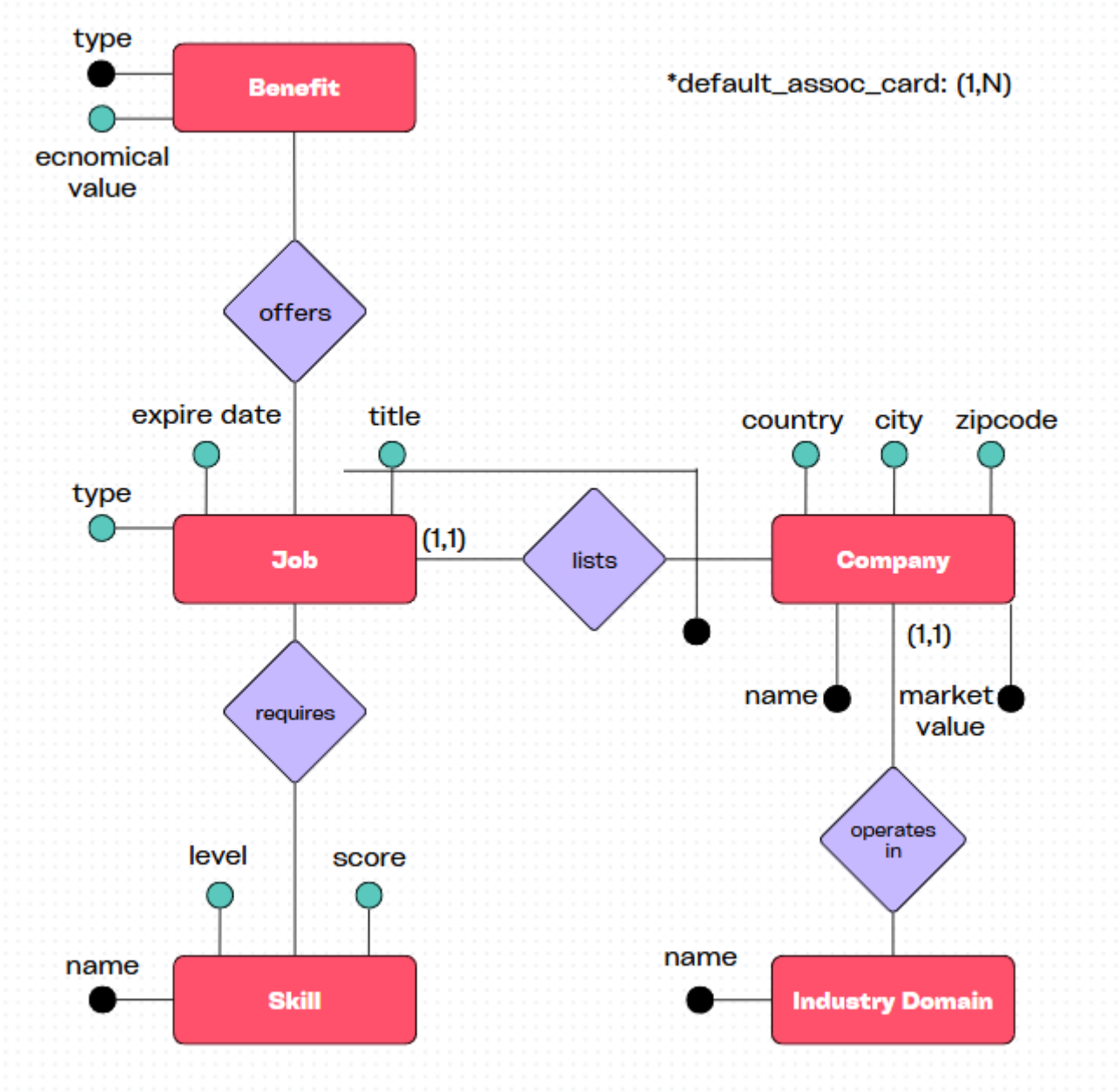
```
Read/Write Intensity
The application is predominantly read-intensive, as it emphasizes
retrieving data; such as querying the skills required for a job, the
benefits offered, or the industries associated with a company.
Write operations occur moderately, primarily when new jobs, companies,
skills, 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.
For core operations (e.g., 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.
However, having high availability clearly is the most important goal to
reach, particularly for supporting real-time data retrieval and queries.

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.
```

(3) Conceptual Schema

The following conceptual schema captures key entities and their relationships:

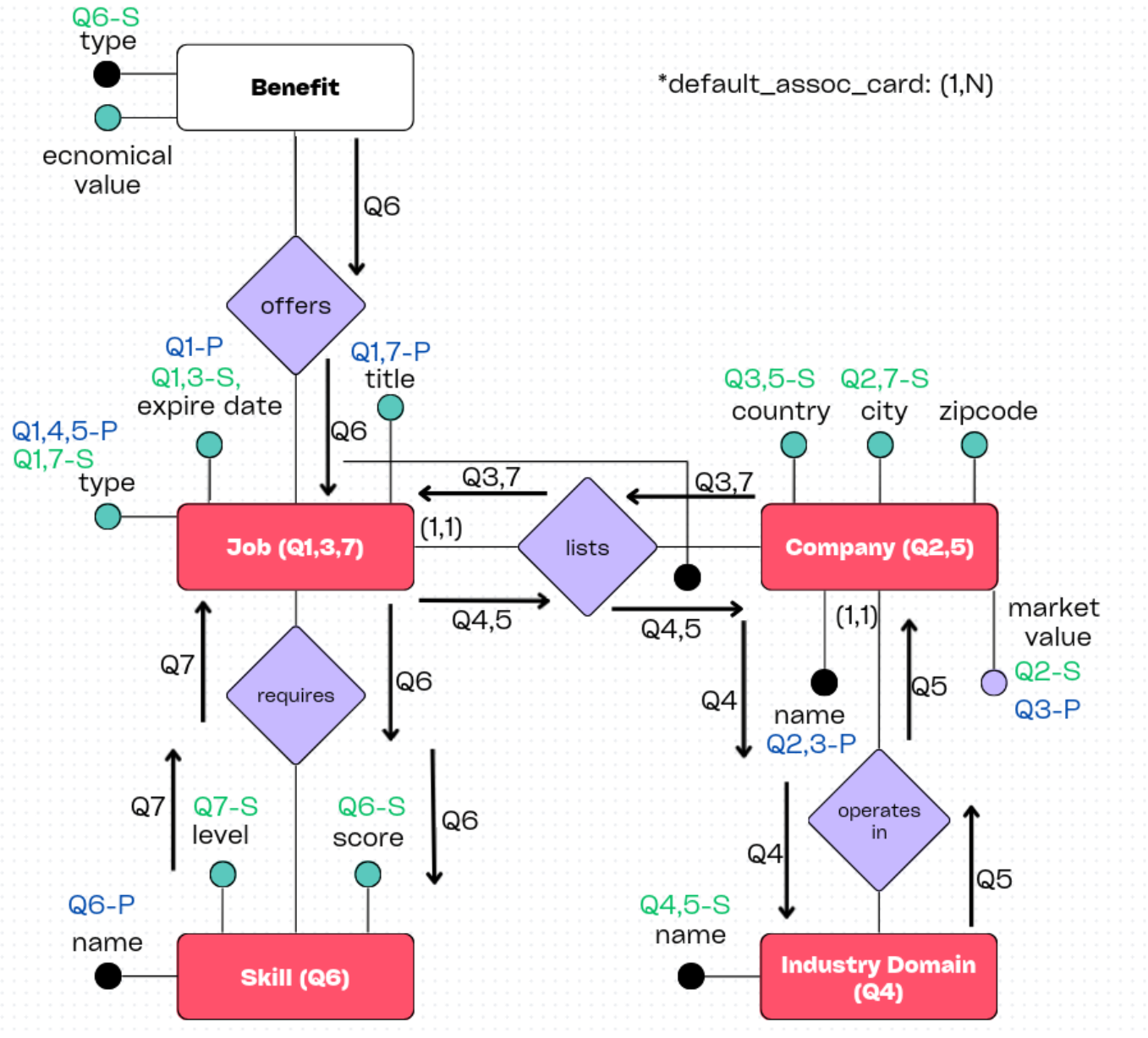
```
Nodes:
    Job(Title, Expire date, Type): Represents a job posting (for now has a
title attribute)
    Company(Name, Country, City, ZipCode, MarketValue): Represents a
company (for now has a name attribute)
    IndustryDomain(Name): Represents an industry domain
    Skill(Name, Level, Score): Represents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

Relationships:
    BELONGS_TO: Links a job to an company (1-N: with an attribute "salary"
of the association)
    --> the key of Job will be the composite of title and the foreign key
of company
    REQUIRES: Links a job to the skills it demands (N-N)
    OFFERS: Links a job to the benefits it provides (N-N)
    OPERATES_IN: Links a company to the industries it operates in (1-N)
```



(4) Queries:

```
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)_!])
```



(5) Aggregates

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} }] }] }

Skill:

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

Note: inspect .md to see comments

(6) Design in MongoDB

Queries associated with Skill: Q6

Selection attributes for Q6: {score, type}

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

- A non-unique index on the partition key with Partition key = {score, type}

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

- A compound-unique index which contains the full shard key as a prefix of the index = {score, type, name}

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

- 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});
```

Queries associated with IndustryDomain: Q4

Selection attributes for Q4: {name}

IndustryDomain:

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

- 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}

Company:

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

- 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 non-unique index for sharding and to then support the uniqueness of name with a compound index:

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

- A compound-unique index which contains the full shard key as a prefix of the index = {score, type, name}

```
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}

JobOffer:

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

- 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 index:

```
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
  }
);
```

(7) Design in Cassandra

Queries associated with Skill: Q6

Selection attributes for Q6: {score, type}

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

- Given that we have only one query we can safely select {score, type} 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, type}
  - Primary key = {score, key, name} with name as clustering column
- Here are the CREATE commands in Cassandra:

```
  CREATE TYPE benefit_t (
    type text
);

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

In summary:

TYPE benefit_t defines the structure for benefits for semantic reasons

((score, type)) as partition key ensures data is grouped by score and type

name as clustering column differentiates entries within each (score, type) partition

provides represents the benefits associated with a skill as a set of the previously created frozen benefit_t

- Q6:

```
SELECT name
FROM Skills
WHERE score > 70 AND type = '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>>
);
```

In summary:

TYPE job_t represents the type of jobs associated with a company

TYPE company_t represents a company, with its associated job types encapsulated as a list of job_t

IndustryDomain uses name as the Partition Key & the operated field stores the hierarchical relationship of companies and jobs.

- 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 }

- Unlikely 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 are able 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 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).

```
CREATE TYPE skill_t (
    level text
);

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

A further analysis can lead us to optimize, for example, Job7 table by removing unnecessary fields (we still have Job1_3 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 7.

```
CREATE TABLE Job7 (
    type text,
    city text,
    level text,
    title text,
    companyName text,
    PRIMARY KEY ((type, city, level), title, companyName)
);
```

- Q1:

```
SELECT *
FROM Job1_3
WHERE type = 'Full-time' AND expire_date <= toDate(now()) + 30;
```

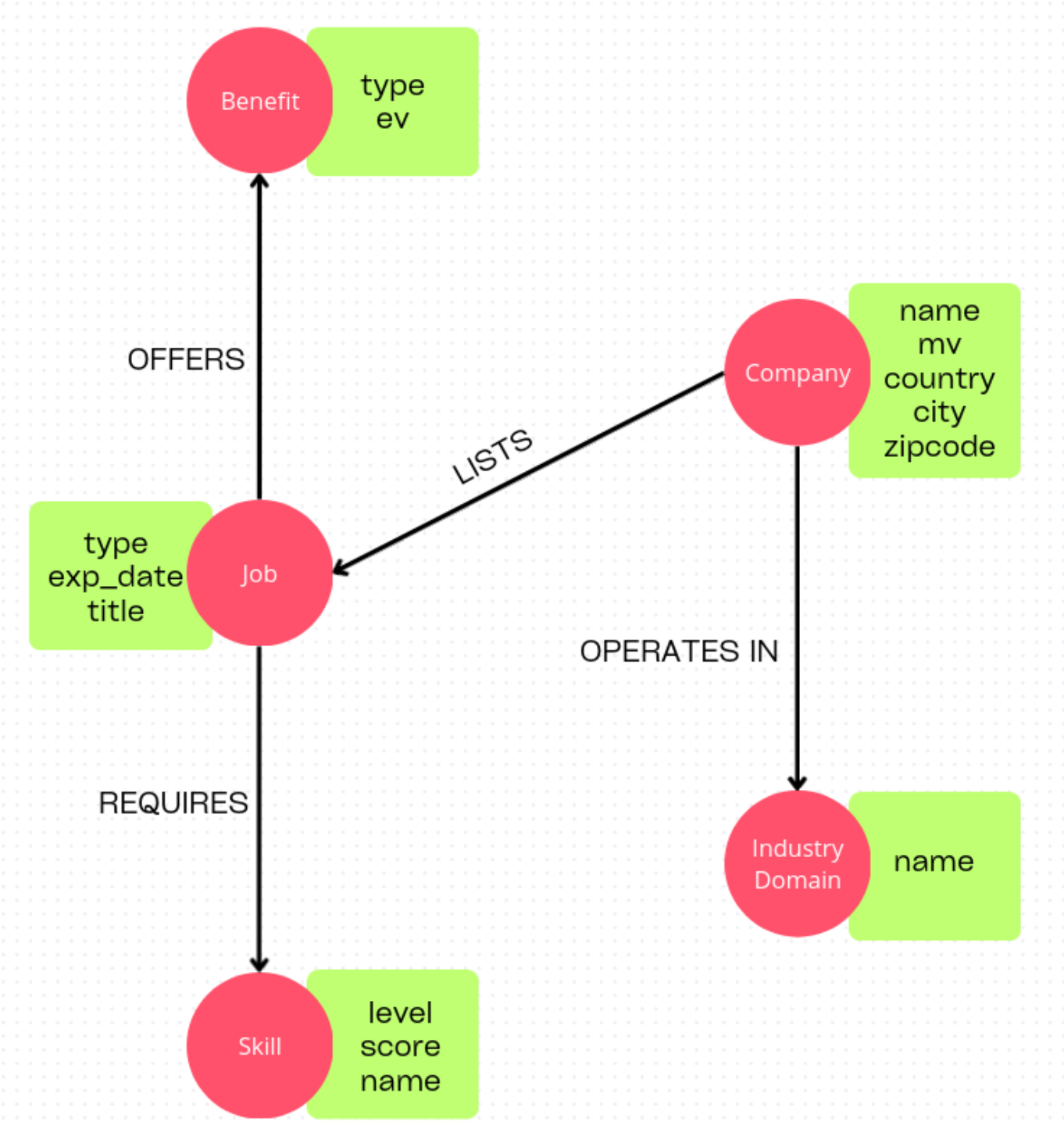
- Q3:

```
SELECT companyName, marketValue
FROM Job1_3
WHERE country = 'Russia' AND expire_date > toDate(now()) + 60;
```

- Q7:

```
SELECT title
FROM Job7
WHERE type = 'Internship'
  AND city = 'Hamburg'
  AND level = 'Beginner';
```

(8) Design in Neo4J



Query workload in Neo4J (from 1 to 7):

```
MATCH (j:Job)
WHERE j.type = 'Full-time' AND j.exp_date <= 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 j.exp_date > date() + duration({days: 60})
RETURN c.name, c.mv
```

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

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

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

(9) System S choice:

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.

(10) Neo4J: Configuration details

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 N, 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, 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!

(11) Neo4J: Schema details

Given Neo4J schema-less nature we don't have to provide any schema details (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. 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);
```

(12) Neo4J: Create an instance of your schema in the selected system ... 0xmYonyPlGmxOx1je29k25BMOMjEPUrkPXnBJEva7cU

```
- [ ] a. You can use an already available dataset. The dataset should have
a reasonable size (few Mb).
- [ ] b. Notice that selected datasets might need to be transformed in
order to be used by your application. For dataset transformation, you can
rely on either data transformation tools, such as Tableaux Prep
(www.tableau.com), Apache Superset (superset.apache.org) Trifacta (
www.trifacta.com ), or other ETL tools such as Talend ( www.talend.com ),
or scripts in your favorite language.
```

(13) Neo4J: Workload implementation

- Here follows the workload implementation on Neo4J

  << DA METTERE QUA I COMANDI E LE LORO ESECUZIONI ED EXPLAIN >>

- Here are some indexes that could enhance systems' capabilities:

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

// Company node 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);

// IndustryDomain node indexes
CREATE INDEX FOR (id:IndustryDomain) ON (id.name);

// Skill node indexes
CREATE INDEX FOR (s:Skill) ON (s.score);
CREATE INDEX FOR (s:Skill) ON (s.level);

// Benefit node indexes
CREATE INDEX FOR (b:Benefit) ON (b.type);

// Composite indexes
CREATE INDEX FOR (j:Job) ON (j.type, j.exp_date);
CREATE INDEX FOR (c:Company) ON (c.city, c.mv);
CREATE INDEX FOR (c:Company) ON (c.country, c.mv);
```

<< CONTROLLARE LE VARIE QUERY DOPO AVER MESSO GLI INDICI >>