DENORMALIZATION:

One of the advantages of using MongoDB over a relational database like MySQL is that unstructured data can be stored in MongoDB. This along with the scalability facilitated by the embedded document structure of data and scale-out architecture results in low effort to set up, unlike MySQL which requires thorough schema design with primary keys, foreign keys, constraints and so on. This advantage of MongoDB allows us to design denormalized databases. Denormalization is based on the simple principle of "*Data that is accessed together should be stored together*". Denormalized databases can improve read performance and query performance in a variety of cases, such as:

- A recurring query requires a few fields from a large document in another collection. We can choose to maintain a copy of those fields in an embedded document in the collection that the recurring query targets to avoid merging two distinct collections or performing frequent $lookup operations
- An average value of some field in a collection is frequently requested. We can choose to create a derived field in a separate collection that is updated as part of your writes and maintains a running average for that field

While embedding documents or arrays without data duplication is preferred for grouping related data, denormalization can improve read performance when separate collections must be maintained. To summarize, denormalization makes sense when we have a high read-to-write ratio [Link to official guide].

In the case of our database, unlike our previous proposed architecture in Lab 1 wherein we had designed an elaborate ER diagram for a relational database with detailed relationships between the entities (i.e. tables), in the MongoDB database case, we can use the principle of denormalization to consolidate multiple tables into 3 collections:

- Companies collection –
    - Combination of the companies, company_industries and company_specialties datasets, all joined together by company_id
    - Industries and specialties are embedded into arrays to reduce redundancy
    - Snapshots of Python code used to create the underlying dataframe for companies collection below:

```
# Grouping industries df by company_id and converting industry into a list

company_industries_grouped_df = company_industries_df.groupby('company_id').agg({'industry': lambda x: list(x)}).reset_index()


# Grouping specialities df by company_id and converting specialities into a list

company_specialities_grouped_df = company_specialities_df.groupby('company_id').agg({'speciality': lambda x: list(x)}).reset_index()

# Merging all datasets with companies_df being the left table
```

merged_companies_df = companies_df.merge(company_industries_grouped_df, on='company_id', how='left').merge(company_specialities_grouped_df,on='company_id',how='left')

Renaming the column name to company_name for better understanding

merged_companies_df.rename(columns = {'name':'company_name'}, inplace = True)

- Job postings –
    - This collection combines the job_postings, job skills, job_benefits and job_indsutries datasets all joined together by job_id
    - Skills, benefits and industries are embedded into arrays to reduce redundancy
    - Snapshots of Python code used to create the underlying dataframe for job_postings collection below:

Grouping benefits df by job_id and converting benefits (type column) into a list

benefits_grouped_df = benefits_df.groupby('job_id').agg({'type': lambda x: list(x)}).reset_index()

Grouping benefits df by job_id and converting inferred into a list

inferred_grouped_df = benefits_df.groupby('job_id').agg({'inferred': lambda x: list(x)}).reset_index()

Combining the above-created dataframes

benefits_group_combined_df = benefits_grouped_df.merge(inferred_grouped_df, on='job_id', how='left')

Renaming the column 'type' to 'benefits' for better understanding

benefits_group_combined_df.rename(columns = {'type':'benefits'}, inplace = True)

Grouping industries df by job_id and converting industry_id column into a list

job_industries_grouped_df = job_industries_df.groupby('job_id').agg({'industry_id': lambda x: list(x)}).reset_index()

Grouping skills df by job_id and converting skill_abr column into a list

job_skills_grouped_df= job_skills_df.groupby('job_id').agg({'skill_abr': lambda x: list(x)}).reset_index()

Merging all datasets with job_postings being the left table

merged_job_postings_df = job_postings_df.merge(benefits_group_combined_df,on='job_id',how='left').merge(job_industries_grouped_df,on='job_id',how='left').merge(job_skills_grouped_df,on='job_id',how='left')

- Employee counts –
    - This dataset will remain as is since it contains date recorded
    - The only change is that we're creating 2 additional date & time columns, extracted from time_recorded column to make it easier to query

Snapshot below:

```
# Creating date column to make it easier to query in mongodb

employee_counts_df['time_recorded_ts'] = pd.to_datetime(employee_counts_df.time_recorded * 1e9)

employee_counts_df['date_recorded']  =  employee_counts_df['time_recorded_ts'].dt.strftime('%Y-%m-%d')
```

These dataframes once created can be exported into MongoDB collections as per the conceptual database design.