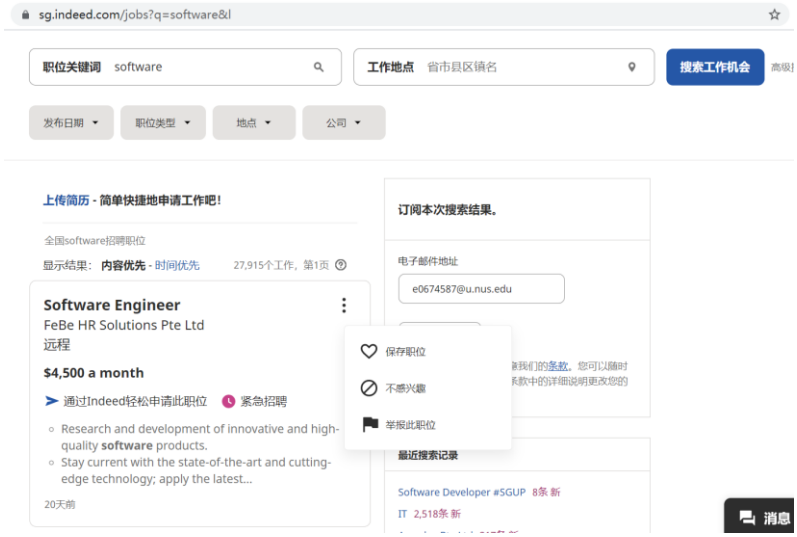
Job Searching & Management Platform

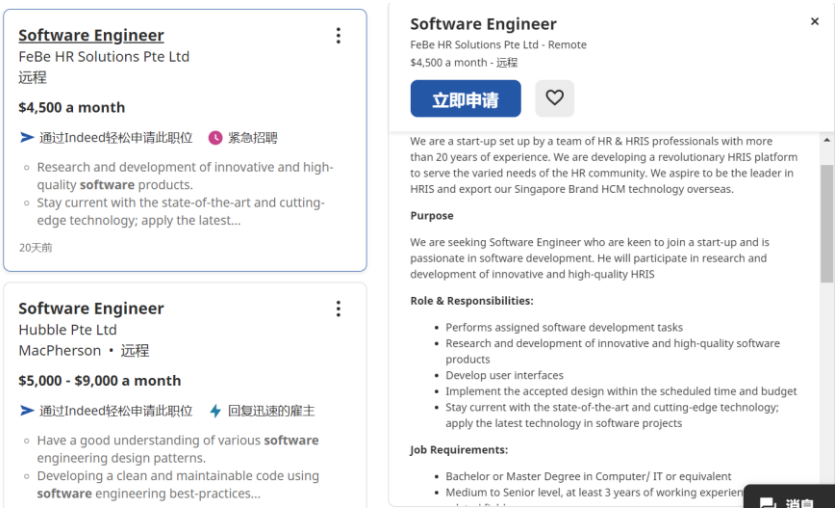
1. Objective

This project aims to design a simple job searching website. It provides users with up-to-date job post information and supports job searching and filtering. It also allows companies to post and manage new job opportunities decently. The resources are collected from Indeed. The developer hopes to realize similar functions as a normal job searching website.

Indeed is a popular job searching site that connects millions of job seekers with recruiting enterprices worldwide. It has a concise mainpage that provides users with a list of job cards, which can be searched, filtered and sorted. Each of the job cards provides the title, location of the company, salary, a descriptive short snippet, etc. More detailed information can be found by clicking the link the card has been attached to. Users can choose to favor, unfavor or report an issue on a job post.



A sample searching page on Indeed



Detailed display for a job card

2. Overview

This project is built up with 4 important components, which can be listed as follows:

1) ***web crawler***: used to collect job posts from Indeed routinely

2) ***front-end UI***: provide user interfaces for job searching and result displaying

3) ***back-end API***: realize the logic for web pages, retrieve and proceed with user inputs while interact with databases to collect and return the results

4) ***databases***: used to store all the useful information (roles, users, jobs, etc.) and provide quick searches for job posts.

Meanwhile, the job searching platform generally provides the following functions:

1) **job posts searching platform** open to any users (registered or unregistered); the page also supports post-searching operations like filtering and sorting;

2) **registration and logging in** for users;

3) **a simple recommendation of jobs** for users based on searching histories when no search is conducted;

4) **creation and management of job posts** specified for companies;

5) **management of users** specified for administrators.

3. Infrastructure (Technology Stack)

The overall project is written in Python, and supported by Flask.

The web crawler uses Selenium to simulate user login and to retrieve the static / dynamic contents from the original website. Once the contents are collected, they are proceeded locally with Beautiful Soup (bs4).

The front-end pages are displayed as html files with CSS styles. Bootstrap is chosen for page rendering. The results of interaction between front-end and back-end APIs are reflected in web pages with the help of Jinja2 template rendering engine, which is provided by Flask.

The back-end logics are realized with Flask, a lightweight Python framework. The visiting activities are recorded by cookies and sessions, and the sessions are stored in Redis.

Apart from the use of Redis, two types of database systems are applied in this project. MySQL is selected for storing all the detailed information, including roles, accounts and job posts (forward indexed). Sqlalchemy is applied to interact with MySQL through Python script. Meanwhile, since the main function of a job searching site is the search for jobs, Elasticsearch is adopted to support inverted indexing of the job posts in order to accelerate the searching process.

4. Detailed Design

4.1. Web Scrawler

Selenium is used to automate the process of logging into the website and retrieving the latest information. Initially, the scrawler returns all the job posts with keywords “software” and “IT”. The following information are extracted from each job card (a job post record from Indeed):

Column Name	Description
Title	Name/Title of the job
Link	Link to the detailed description of a job post
Company	Name of the company who releases this job post
Salary	Claimed salary on the job post
Date	Date when the job post was released
Description	Job-snippet of an indeed job post

Table 1. key information for web scrawler

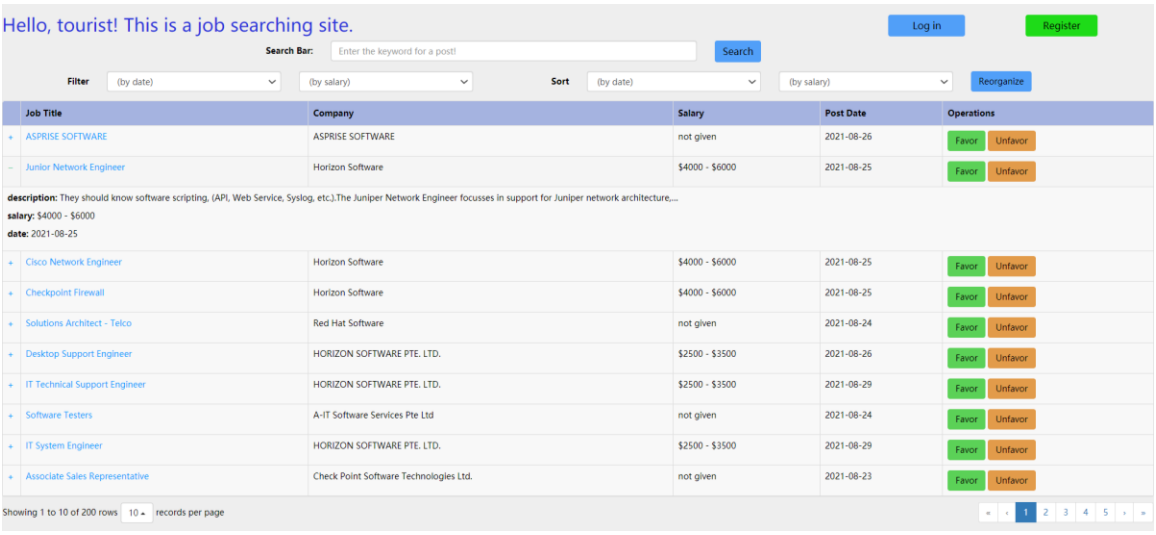
Specifically, a selenium.webdriver object is instantiated with web driver Chrome. After that, a search URL is generated with desired keywords with the help of urllib. The resulting webpages are scrapped one by one by the webdriver object, and returned to a BeatifulSoup object for finer extraction.

BeautifulSoup is imported to process with html files locally. It will extract all the key information listed in Table 1. All the extracted information will be collected in a Python dictionary which will later be transferred into a record in jobposts table in the database.

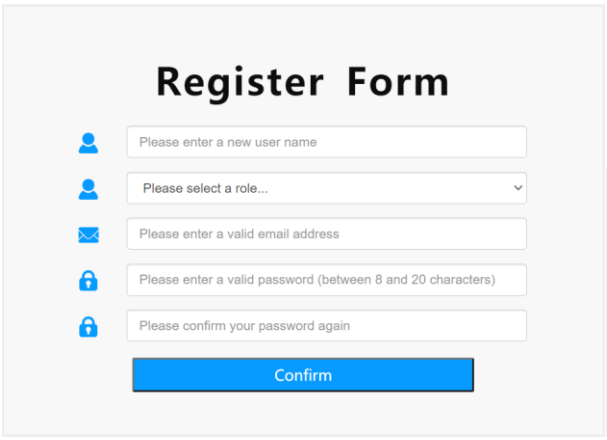
The range of time can be specified for the collected posts, and the number of posts to be crawled can also be configured. The web scrawler is implemented in a multi-threaded way to collect enough information fast and safely.

4.2. Front-end UI

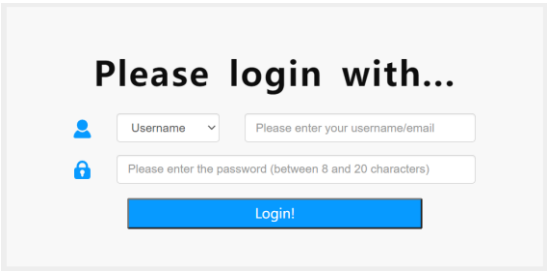
The front-end UI would mainly be supported by Bootstrap, including the CSS configurations and Javascripts. The rought layouts of the above files are shown in Graph 1.



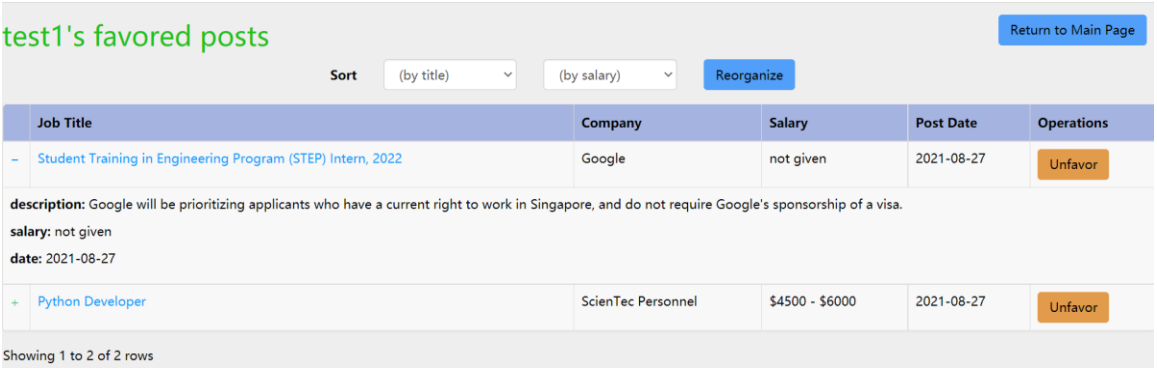
Job_search.html



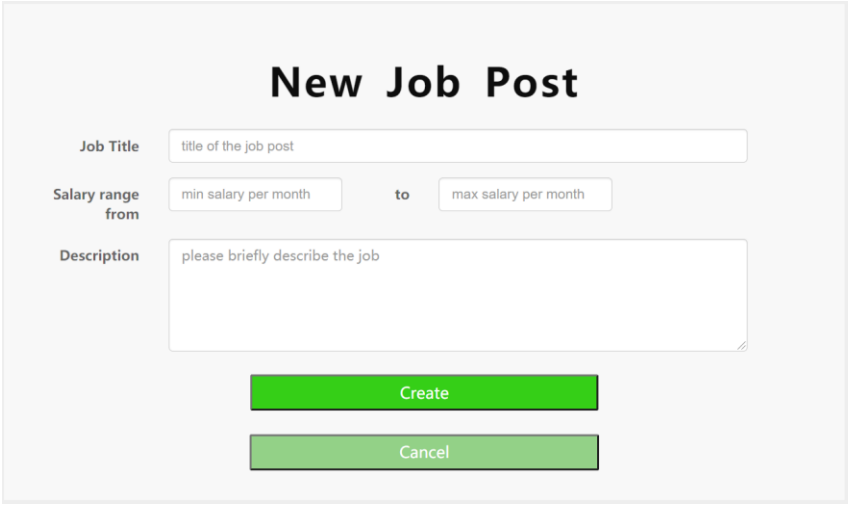
Register.html



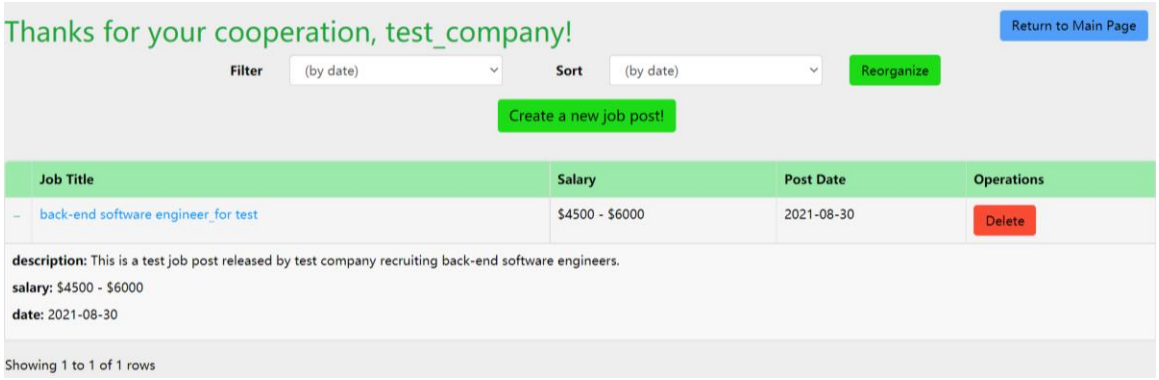
login.html



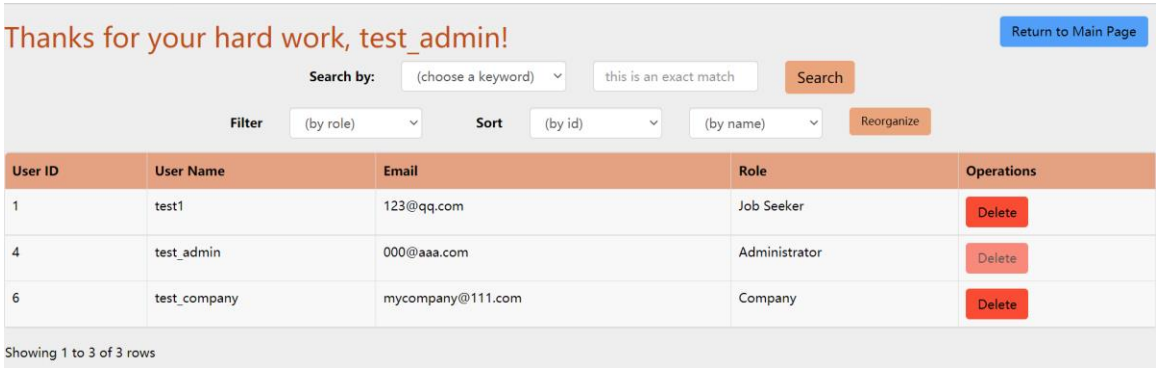
Job_favors.html



Job_create.html



Job_manage.html



User_manage.html

Graph 1. Rough layouts of front-end pages

As a detailed description, job_search.html is a public webpage for all kinds of users (tourists, job seekers, companies, admins). A search bar is used to collect the searching keyword, and a set of filters are used to select/reorganize some of the records from the search results. The search results are organized in pages (ten records for a page by default). Besides, registered users can favor some of the job posts for recording and comparing. The favored posts and past searching records help build up the user portraits, and would be used be recommend users with related job posts. The recommendations will be listed if no search is conducted by a registered user or a tourist.

Register.html and login.html serve as a universal platform for all kinds of users to register and login.

Job_favors.html is provided for all registered users. It offers users with a convenient access to retrieve and manage their concerned job posts.

Job_manage.html and job_create.html are restricted to registered companies. Companies can choose to add a new job post, view the posted jobs and delete some of the posts. A filter is provided to select some of the posted jobs from the entire list.

User_manage.html is restricted to administrators. All the registered users would be listed in the user list. Administrators can see the name and type of the users and choose to delete some of them (which is dangerous!). An administrator cannot delete other administrators.

4.3. Back-end API

4.3.1. job searching, filtering, sorting and favoring

This API is mainly designed for job searching. First of all, check if any user has logged in. If yes, retrieve the user's name from current session and generate a greeting at the top of the webpage. After that, refresh the webpage and show 10 recommended job posts. The simple scheme to generate the recommendations is introduced in the next section.

The searching process begins upon pressing the confirm button beside the search bar. The searching keyword would be analyzed by ElasticSearch and help locate the targets. At the same time, the keywords' analyzing results would be recorded in the field of "search_history" in table "users" as well as in the public search history stored in Redis.

```
QUERY = {
    "_source": ['post_id'],
    "size": 200,
    "query": {
        "multi_match": {
            "query": "software",
            "type": "best_fields",
            "fields": ["title", "company", "description"],
            "tie_breaker": 0.3
        }
    }
}
```

A template query for Elasticsearch

```
# case 4: receiving request from the search bar
elif "keyword" in keys:
    # step 1) record the resulting post ids from es
    filtered_kw = info["keyword"]
    query = Config.QUERY.copy()
    query["query"]["multi_match"]["query"] = filtered_kw
    response = es.search(index="index_jobposts", body=query)["hits"]["hits"]
    # step 2) retrieve the records from MySQL and rearrange them according to the order in id_dict
    session["search_results"] = get_post_MySQL(response)
    resp = make_response(render_template("job_search.html", \
        name=session.get("user_name"), posts=session["search_results"], role=role))
```

A general process of job searching

After each search by a user/tourist, the results would be shown in a table. Only the first 300 records would be displayed in order to control the space cost. These records would be stored as a list of python dictionaries. This enables the fast switching between the original and filtered results, and reduces the queries on the database.

The filters of the search results work on certain keys of the dictionary, and filtering would take O(N) time. The sorting process is supported by mergesort, which takes O(NlogN) time on average. Mergesort is selected to support multi-keyword sorting (due to its stability) while ensuring the efficiency. As the search results are temporarily stored in Python, the use of mergesort does not degrade the space complexity. The filtering and sorting functions are provided in app/utils.py.

```
# case 6: receiving response from the filters or sorters
else:
    operated_results = posts
    if operated_results != []:
        for key, val in request.form.items():    # different filters and sorters can add up
            operation, kw = key.split("_")
            if operation == "filter":
                operated_results = filter_results(operated_results, kw, val)    # non-destructive
            else:
                operated_results = sort_results(operated_results, kw, val)    # !! destructive
    return render_template("job_search.html", \
        name=session.get("user_name"), posts=operated_results, role=role)
```

A general process of filtering and sorting

4.3.2. simple recommendation scheme

In order not to display the main webpage as a nearly blank page for any tourists (unregistered) or registered users, 10 recommendation posts would be generated and placed in the results table. This process happens when the users first accesses the mainpage and before any search is conducted.

For any registered users, the recommendation system first look at the “search_history” field of the client’s record in the database. It would pick up at most three mostly searched keywords and conduct a search in Elasticsearch database. Among the returned results, only the top-10-scored posts are selected. These 10 posts are checked and would not repeat the posts already favored by the user.

```
# case 1: local search history found: use the 3 most frequently used keywords for recommendation
if history_str:
    history_str_new, hotspots = get_hotspots(history_str)    # this function would change the order of history
    recommend_posts = gen_recommendations(es, hotspots, favored_list)
    resp = make_response(render_template("job_search.html", \
        name=session.get("user_name"), posts=recommend_posts, role=role))
    session["search_history"] = history_str_new
    db.session.query(Users).filter_by(user_id=session["user_id"]).update({"search_history" : history_str_new})
    db.session.commit()
    db.session.close()
    session["search_results"] = recommend_posts
    return resp
```

general process to recommend using user’s search history

Specifically speaking, the search history stored in the browser is not sorted as it would be read and written frequently during each search of the user. The sorting process only happens when the recommendation process is triggered. In order to get the 3 most frequently searched keywords, quick selection is used to locate the top 3 keywords. This process would reorder the search history to reduce any future repetitive work. This sorting process takes an average of $O(N)$ times.

For unregistered tourists, the server first tries to generate the recommendations with 3 latest used keywords by any of the users. As a tourist does not have his own search history, it would be better to recommend them with the latest hotspots. Therefore, a general search history is stored in Redis in a simple format. Each time a search is conducted by a user, any new keywords would be appended to this general search history, and the existing keywords (unique) are reordered so that the latest searched keywords are always at the rear.

In case the general history cannot be accessed or is somehow empty, the server would use the lasted posts added to the database as the recommendations.

```
else:
    redis_instance = redis.Redis(connection_pool=redis_pool)
    history_all = redis_instance.get("search_history_all")
    # case 2: search history for all users found: use the latest three keywords to generate recommendation
    if history_all:
        history_lst = history_all.decode("utf-8").split("&")
        latest_words = history_lst[-1:-4:-1]
        recommend_posts = gen_recommendations(es, latest_words, [])
    # case 3: use the latest 10 posts for recommendation
    else:
        posts = JobPost.query.order_by(JobPost.post_id.desc())[0:10]
        for post in posts:
            recommend_posts.append(create_post(post))
    session["search_results"] = recommend_posts
    return render_template("job_search.html", \
        name=session.get("user_name"), posts=recommend_posts, role=role)
```

general process of recommendation for unregistered users

4.3.3. register

This API uses the request class from flask to retrieve the payload of a post request from register.html. The payload should include a form with the following data: name, role, email, password, repassword. JQuery.validate is used to ensure the correct format of these information.

```
{
    "name" : "John",
    "role" : "jobseeker",
    "email" : "123123@qq.com",
    "password" : "11111111",
    "repassword" : "11111111"
}
```

Sample of a post request payload (as a form)

The respective back-end API would interact with MySQL to ensure the username and email are unique for each account. At the same time, the password would be compared with "repassword" to make sure there's no difference. All the errors are collected as a list and result in an alert in register.html. If no error happens, a new account would be created and automatically logged in.

4.3.4. log in

This API works for login.html. Similar to 4.3.2., this API gets a post request with name/email and password, find out the user with the given email and check the correctness of password. Occurance of empty account or wrong password would be recorded as a list and displayed as alerts in login.html.

```python
@users.route("/login", methods=["POST", "GET"])
def login():
    if request.method == "POST":
        mode = request.form["mode"]
        identity = request.form["identity"]
        password = request.form["password"]

        errors = []
        if mode == "name":
            errors.extend(checkByName(identity, password))
        else:
            errors.extend(checkByEmail(identity, password))
    if len(errors) > 0:
        return render_template("login.html", errors=errors)
    user = Users.query.filter_by(name=identity).first() if mode == "name" \
            else Users.query.filter_by(email=identity).first()
    session["user_id"] = user.user_id
    session["user_name"] = user.name
    # session["favors"] = user.favors
    session["search_history"] = user.search_history
    return redirect("/")
else:
    return render_template("login.html")
```

General process for login

4.3.5. job post creation

This API is called when a company click on the “create a new job post” button, and the user (whose role should be “company”) must pass the permission check (wrapped as a decorator of the API). All the information provided in job_create.html would be sent as a post request, while the company name woud be automatically acquired from the current user session. The creation date would be set as the current date.

```
{
  “title” : “Python Software Developer”,
  “salary_min” : “4500”,
  “salary_max” : “6000”,
  “description” : “This is a sample post for a new job created by test_company.”
}
```

Sample of a job creation request payload

4.3.6. job management & user management

Job managing API and user managing API follow a similar logic. First, the user permission should be checked. After that, the respective user information would be displayed in the managing page, and the job posts/users list would be displayed in a table. Different from that of a job search, the job post list / users list are directly accessed from MySQL, considering that many / all of the records in a table should be returned. The filtering and sorting processes are the same as that of the

search results for job searching (4.3.1.). For job management, the user (company) can choose to create or delete a job post, whereas for user management, the administrators are not granted the permission to create a user, and cannot delete any other administrators.

4.4. Database Systems

Three database systems are used for this project. Elasticsearch is adopted specifically for job searching, and includes one main index: index_jobposts. MySQL is used for storing all the detailed information, and include roles, users and jobposts. Redis is also adopted and is responsible for storing sessions and public search histories.

4.4.1. Elasticsearch

The index in Elasticsearch stores some of the information of a job post that needs to be analyzed. The metadata of table/type jobposts in ElasticSearch is described in Table 2. Each of the files in ES represents a job post.

Column Name	Data Type	Description
Post_id	Integer	A unique ID for the job post in the database
Title	Text	Name/Title of the job
Company	Text	Name of the company who releases this job post
Description	Text	Job-snippet of an indeed job post or self-added descriptions by the company.

Table 2. metadata of type “jobposts”

Specifically, “job_title”, “company” and “description” are used for matching and scoring. Post_id is used to link the Elasticsearch file to a detailed job record in MySQL.

```
mappings = {
    "mappings": {
        "properties": {
            "post_id": {
                "type": "long",
                "index": True
            },
            "title": {
                "type": "text",
                "index": True,
                "analyzer": "default"
            },
            "company": {
                "type": "text",
                "index": True,
                "analyzer": "default"
            },
            "description": {
                "type": "text",
                "index": True,
                "analyzer": "default"
            }
        }
    },
    "settings": {
        "number_of_replicas": 2,
        "number_of_shards": 5
    }
}
```

Structure of document “jobpost”

4.4.2. MySQL

MySQL stores all the detailed information of the users and the job posts.

The Role table in MySQL stores the basic information for different roles in the project. Only registered roles would be recorded in this table, which include jobseekers, companies and admins. The metadata of table Role is shown in Table 3.

Column Name	Data Type	Description
Role_id	Integer	(primary key) Unique id for a role
Name	Varchar(80)	Name of the role
Permissions	Integer	A union set of permissions for the role

Table 3. metadata of table “roles”

The permissions are listed as follows:

Notation	Representation	Description
a	NO NEED	job searching, filtering, sorting
b	0x01	job post favoring
c	0x02	creating a new job post
d	0x04	managing the job posts of his own company
e	0x08	managing the users

Table 4. permissions and their representations

Table Users in MySQL stores the information of each registered users, their metadata described in Table 5.

Column Name	Data Type	Description
User_id	Integer	(primary key) Unique id for a user
Name	Varchar(100)	Name of the user
Email	Varchar(100)	Email of the user
Role_id	Integer	(foreign key) role id of the user
Password	Varchar(128)	Password for the user’s account
Search_history	Text	Search history recorded as anayzed key words with frequency

Table 5. metadata of table “users”

Different kinds of users are assigned with different permissions. Table 6 shows the relationship between roles and permissions.

Role Name	Permissions (sum)	Permitted operations(in notation)
Tourist(unregistered)	0x01	a
Jobseeker	0x03	a, b
Company	0x0f	a, b, c, d
Admin	0x1f	a, b, e

Table 6. relationship between roles and permissions

Table jobpost in MySQL is very much similar to table/type jobpost in ES. The metadata of table “jobpost” is described in table 7.

Column Name	Data Type	Description
Post_id	Integer	(primary key) Unique id for the file
Title	Varchar(100)	Name/Title of the job
Link	Text	Link to the detailed description of a job post; nullable
Company	Varchar(100)	Name of the company who releases this job post
Salary_min	Integer	Loweset claimed salary on the job post
Salary_max	Integer	Highest claimed salary on the job post
Date	Date	Date when the job post was released

Description	Text	Job-snippet of an indeed job post or self-added descriptions by the company.

Table 7. metadata for table "jobpost"

In order to support persistent storage of user's favoring, an association table "favors" is created which supports the "one-to-many" relationship between the user and his favored posts.

Column Name	Data Type	Description
User_id	Integer	(references users(user_id)) id for the user
Post_id	Integer	(references jobpost(post_id)) id for the favored file

4.4.3. Redis

Redis is adopted mainly to cache the session information. It stores some important user information, as well as the search results obtained in one job search. The user id is stored for accessing detailed information of the user, which are required in situations like permission checks. The search results in one searching process are also stored in order to realize operations like filtering and sorting easily and avoid redundant visits on MySQL. These data are cleared when a session has finished.

Field	Description
Session_id	Id of the current session
User_id	Id of the user in the current session
User_name	Name of the current user
Search_results	Temporary storage of the results in a job search

Complete structure of a session

5. Performance Analysis

5.1. Keyword searching with/without ES

5.2. Sorting and Filtering with/without Redis Caching

Without the use of caching medium like Redis, the detailed information of posts have to be retrieved repeatedly from MySQL upon each sorting / filtering request on the search results.

Comparatively, with the help of Redis, sorting and filtering on the current search result can be operated conveniently without redundant visit to the database, and the search results can be bound with specific user's login sessions. Below are the performance comparison on sorting and filtering of the search results. The "Query on MySQL for filtered/sorted results" column does not include the time to retrieve post ids from keyword searching based on ES.

Filter	Filter on cached results	Query on MySQL for filtered results
Within 24 hours	0.158309936	30.555009841
Within 3 days	0.128507614	30.091524124
Within 7 days	0.181913375	34.332275390

Min salary >= $2000	0.141859054	21.971464157
Min salary >= $5000	0.090360641	10.779619216
Max salary >= $5000	0.096321105	21.590948104
Max salary >= $8000	0.079393386	10.316610336
Within 3 days & Min salary >= $5000	0.211000442	37.926435469
all	0.073909759	31.763792037

Performance Comparison on Filtering in ms (based on the same set of post data)

Sorting Order	Sorting on cached results	Query on MySQL for sorted results
Date: ascending	3.631591796	30.889511108
Date: descending	5.634784698	30.148506164
Min salary: ascending	5.316495895	30.669450759
Min salary: descending	4.635810852	31.548500061
Max salary: ascending	3.282308578	29.303550720
Max salary: descending	4.630327224	29.700517654
Date: descending & Min salary: descending	7.877601852	59.790372848

Performance Comparison on Sorting in ms (based on the same set of post data)

Taking the average of the results, we can come to a conclusion that filtering with cache reduces the processing time by 99.4%, while sorting with cache reduces the processing time by 85.5%.

Actually, the caches can be further compressed as the post ids of the search results, but this would not avoid extra queries on MySQL. As the search results are limited to reasonable sizes (200 - 300), caching the complete job post card would not require much space.