JOB PORTAL DATABASE SYSTEM

Team Number 17

Harshal Sanjiv Patil
UBID: hpatil2

Prathamesh Kishor Gadgil
UBID: pgadgil

MILESTONE 1

I. Introduction:

In today's world, job search and job recruitment processes have become complex and time consuming for both applicants and recruiters as there is no centralized system available.
Handling these processes manually will require significant time investment and come up with the risk of errors. To address this challenge, a centralized database system is necessary for streamlining job applications, achieving accuracy along with enhancing experience.
This project aims to create a database system that effectively manages recruitment process, applicants and status. This system will come up with functionality like job postings, locations, applications tracking, skilled based filtering.

II. Problem Statement:

Manually managing applications, applicants' profile and job postings is unproductive, prone to mistakes, and does not scale effectively. It is challenging to track applicants for each candidate, to match people to their skill matching jobs, challenging to maintain job details and maintain data accuracy for each candidate as well as firm that post applications.
The process can be streamlined with the help of a database system, which can handle large datasets effectively and efficiently, enable sophisticated searches, and ensure data integrity through organized relationships. A database offers scalability, precision, and automation in contrast to manual approaches, increasing the effectiveness of hiring for administrators, employers, and job searchers.

III. Synopsis (Milestone 1)

During Milestone 1 we accomplished all items on the assignment brief whilst establishing a solid base for our Job-Portal database. Task 1 selecting a large, but manageable use-case, landed us in the recruitment domain where we identified natural read-write workflows (posting jobs, applying for jobs, updating application status, and administrative clean-up) which would rely on sophisticated queries and repeated updates. We specified exactly how each User (jobseeker, employer, administrator) would interrogate and change the data e.g. "what Python roles in New York match my profile?" and "how many applications have there been to this posting?".

To make development as simple as possible, Task 2 requires our site to be a handcrafted sandbox. We built a small 'mini portal' with a dozen example rows per table and bundled it with SQL scripts, to create, load, reset, or destroy the example database in a single command, allowing us to debug constraints and logic before building a full production model.

For Task 3 we produced a straightforward E/R diagram that included eight core entities (Applicant, Company, Job, Application, Skill, Category, and three junction tables) and converted it to nine fully normalized relational tables, complete with primary/foreign-key constraints and ON DELETE rules that would define the expected cascades or protection

In Task 4, we used the faker library to create a production-grade dataset by generating thousands of people, jobs, skills, and category links in such a way that many-to-many relationships meaningfully populate the generated dataset. The generator produces output CSVs, along with helper scripts that transform them into bulk COPY loads for PostgreSQL. The dataset generated will fit precisely into the schema.

Lastly, in Task 5 we illustrated the expressiveness of the schema by generating four more advanced SQL examples: a multi-criteria job search, where multiple JOINs were used; a GROUP BY COUNT example to rank jobs by the number of applicants; a sub-query that allows a job-seeker/user to track the status of an application; and a self-joined query that identifies other postings by the same company so they could be flagged as a duplicate. We included screen shots of the results in the report and all four queries were executed correctly with the production data.

Overall, we have a good schema design that can be debugged, and an automated process for loading data, and validated example queries—this makes for

a solid entry point to indexing, performance tuning, and adding on the application-layer integration deliverables in Milestone 2.

MILESTONE 2

I. Normalization

We performed the steps mentioned below to make sure our Job Portal Database was normalized up to BCNF :

Step 1: Analyze Functional Dependencies:
To find out whether attributes depend on one another, analyze each relation in the schema.

Step 2: Explore Candidate Key:
Identify the minimum number of attributes required for identifying every tuple in the relation.

Step 3: Normalization:
Make sure that each relation has a super key outcome for each non-trivial functional dependency. Otherwise, split down the relationship properly.

BCNF Analysis of tables:

Table1: Application Table
Attributes: applicant_id (Primary Key), name, email, phone, resume, location
Candidate key: {application_id}, {email}
Functional Dependencies:
application_id → {name, email, phone, resume, location}
email → application_id

Table 2: Company Table:
Attributes: company_id (Primary Key), name, industry location, website
Candidate key: {company_id}
Functional Dependencies:
comapny_id → {name, industry, location, website}

Table 3: Skill table
Attributes: skill_id (Primary Key), skill_name are attributes.
Candidate key: {skill_id}, {skill_name}
Functional Dependencies:
skill_id → skill_name
skill_name → skill_id

Table 4: Category Table
Attributes: category_id (Primary Key), category_name
Candidate key: {category_id}, {category_name}
Functional Dependencies:
category_id → category_name
category_name → category_id

Table 5: Job Table
Attributes: job_id (Primary Key), title, description, location, salary, posted_date, company_id
Candidate key: {job_id}
Functional Dependencies:
job_id → {title, description, location, salary, posted_date, company_id}

Table 6: Application Table
Attributes: application_id (Primary Key), application_id, job_id, application_date, status
Candidate key: {application_id} and {applicant_id, job_id}
Functional Dependencies:
application_id → {application_id, job_id, application_date, status}
{application_id, job_id} → application_id

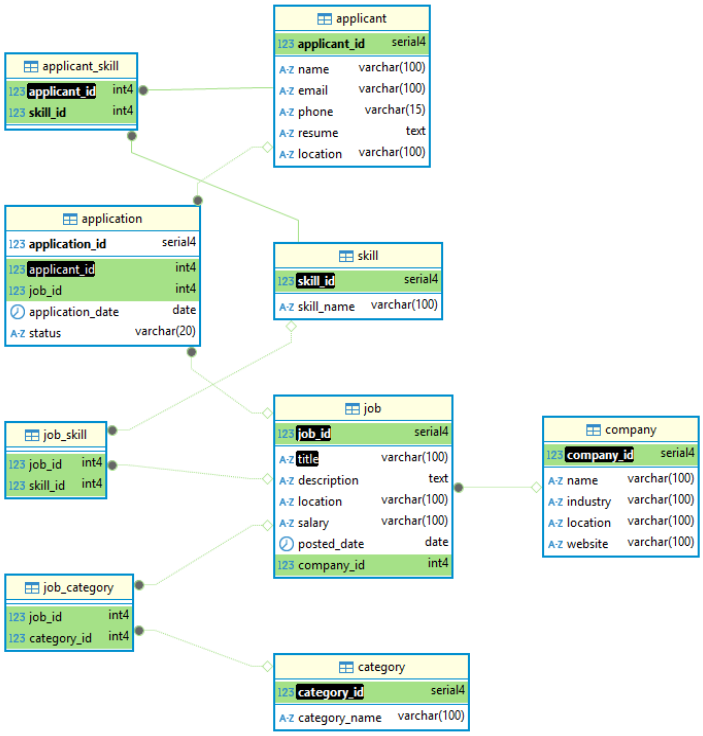
Table 7: Job_Skill Table:
Attributes: job_id, skill_id (Composite Primary Key)
Functional Dependencies:
{job_id, skill_id} → {}

Table 8: Job_Category Table :
Attributes: job_id and category_id (Composite Primary key)
Functional Dependencies:
{job_id, category_id} → {}

Table 9: Applicant_Skill Table
Attributes: application_id, skill_id
Candidate key: {application_id, skill_id}
Functional Dependencies:
{application_id, skill_id} → {}

II. Final ER Diagram

An ER diagram provides a clearer perspective of the whole overall database structure. It highlights the mapping of the tables and relates table keys to the individual keys. The ER diagram will show: Table structure, including column names and data types. Primary and foreign key constraints.



III. Indexing

As we know, job portal database keeps growing with the increase in number of positions, applications, and applicants. This got us to face performance problems when we are scaling. Below are some challenges that we have:

1. Job searches became slow when we use multiple filters.
2. Applicants and jobs got skill – matching.
3. Applications status tracking delays.

So, to address these problems, we got indexes on columns based on EXPLAIN ANALYZE diagnostics.

Indexing queries:

CREATE INDEX idx_job_composite ON job (title, location, salary).

CREATE INDEX idx_applicant_location ON applicant(location).

CREATE INDEX idx_applicant_skills ON applicant_skill (applicant_id, skill_id).

CREATE UNIQUE INDEX idx_mv_company_skills ON mv_company_skills(company_id).

CREATE INDEX idx_pending_apps ON application(job_id)

WHERE status = 'Pending'.

IV. SQL Queries

The following are the SQL queries we implemented on the BCNF normalized dataset:

1.Query to count total applications submitted in job category.

```
Query  Query History
1  SELECT c.category_name, COUNT(a.application_id) AS application_count
2  FROM category c
3  JOIN job_category jc ON c.category_id = jc.category_id
4  JOIN job j ON jc.job_id = j.job_id
5  LEFT JOIN application a ON j.job_id = a.job_id
6  GROUP BY c.category_name
7  ORDER BY application_count DESC;
```

Data Output | Messages | Notifications

	category_name character varying (100)	application_count bigint
1	Leg	2146
2	Area	2122
3	Data Science	2108
4	Be	2100
5	Probably	2090
6	Number	2085
7	That	2080

Fig 4.1

2.Using a subquery to get any jobs with high salaries.

```
Query  Query History
1  SELECT j.title, c.name AS company, j.salary
2  FROM job j
3  JOIN company c ON j.company_id = c.company_id
4  WHERE j.salary::numeric > (
5      SELECT AVG(salary::numeric) FROM job
6  )
7  ORDER BY j.salary::numeric DESC;
```

Data Output | Messages | Notifications

	title character varying (100)	company character varying (100)	salary character varying (100)
1	Contracting civil engineer	Sexton, Cunningham and...	157431.75
2	Health and safety inspector	Sexton, Cunningham and...	157266.90
3	Radiographer, diagnostic	Sexton, Cunningham and...	156900.45
4	Cartographer	Sexton, Cunningham and...	156753.45
5	Merchandiser, retail	Sexton, Cunningham and...	156648.45
6	Neurosurgeon	Sexton, Cunningham and...	156390.15
7	Barrister's clerk	Sexton, Cunningham and...	155998.50

Fig 4.2

3.Query to rank applicants by number of skills.

```
Query  Query History
SELECT a.name, a.email,
       COUNT(ak.skill_id) AS skill_count,
       RANK() OVER (ORDER BY COUNT(ak.skill_id) DESC) AS applicant_rank
FROM applicant a
LEFT JOIN applicant_skill ak ON a.applicant_id = ak.applicant_id
GROUP BY a.applicant_id
LIMIT 100;
```

Data Output | Messages | Notifications

	name character varying (100)	email character varying (100)	skill_count bigint	applicant_rank bigint
1	Michael Hernandez	rossmarilyn@example.com	10	1
2	Jon Richardson	mcgeeryan@example.net	9	2
3	Elizabeth Anthony	tiffanyjimenez@example.com	8	3
4	Robert Clark Jr.	clifford24@example.org	8	3
5	Alejandro Goodwin	richard13@example.com	8	3
6	Ana Patrick	raymondmallory@example.net	8	3
7	William Mckinney	johnsonerica@example.net	8	3

Fig 4.3

4. Insert query to add a new job with required skills.

```
Query  Query History
WITH new_job AS (
  INSERT INTO job(title, description,
  location, salary, posted_date, company_id)
  VALUES
  ('Data Scientist', 'Analyze large datasets...',
  'Remote', '120000', CURRENT_DATE, 16)
  RETURNING job_id
)
INSERT INTO job_skill(job_id, skill_id)
SELECT nj.job_id, s.skill_id
FROM new_job nj, skill s
WHERE s.skill_name IN ('Python', 'Machine Learning', 'SQL');
```

Data Output | Messages | Notifications

```
INSERT 0 2

Query returned successfully in 35 msec.
```

Fig 4.4

5. Insert query to add a new company to the database.

```
Query  Query History
INSERT INTO company(name, industry, location, website)
VALUES ('Tech Innovations', 'IT', 'San Francisco', 'techinnov.com')
RETURNING company_id;
```

Data Output | Messages | Notifications

	company_id [PK] integer
1	101

Fig 4.5

6. Query to update applications status.

```
Query  Query History
-- Batch update application statuses
UPDATE application
SET status = 'Rejected'
WHERE job_id IN (
  SELECT job_id FROM job
  WHERE posted_date < CURRENT_DATE - INTERVAL '90 days'
)
AND status = 'Pending';
```

Data Output | Messages | Notifications

```
UPDATE 3199

Query returned successfully in 121 msec.
```

Fig 4.6

7. Update query to increase salaries.

No limit

```
Query  Query History
UPDATE job
SET salary = salary::numeric * 1.05  -- 5% raise
WHERE company_id = 15
AND posted_date > CURRENT_DATE - INTERVAL '1 year';
```

Data Output | Messages | Notifications

```
UPDATE 460

Query returned successfully in 183 msec.
```

Fig 4.7

8.Delete query to remove inactive applicants.

```
Query  Query History
-- First delete related skills for those applicants
DELETE FROM applicant_skill
WHERE applicant_id IN (
  SELECT a.applicant_id
  FROM applicant a
  LEFT JOIN application ap ON a.applicant_id = ap.applicant_id
  GROUP BY a.applicant_id
  HAVING MAX(ap.application_date) < CURRENT_DATE - INTERVAL '1 year'
     OR MAX(ap.application_date) IS NULL
);

-- Then delete those applicants
DELETE FROM applicant
WHERE applicant_id IN (
  SELECT a.applicant_id
  FROM applicant a
  LEFT JOIN application ap ON a.applicant_id = ap.applicant_id
  GROUP BY a.applicant_id
  HAVING MAX(ap.application_date) < CURRENT_DATE - INTERVAL '1 year'
     OR MAX(ap.application_date) IS NULL
);
```

Data Output | Messages | Notifications

```
DELETE 18382

Query returned successfully in 1 min 20 secs.
```

Fig 4.8

9. Query to find applicants with specific skills.

```
-- Find applicants with Python AND SQL skills
SELECT a.name, a.email
FROM applicant a
WHERE EXISTS (
    SELECT 1 FROM applicant_skill aps
    JOIN skill s ON aps.skill_id = s.skill_id
    WHERE aps.applicant_id = a.applicant_id
    AND s.skill_name = 'Python'
```

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	1
Hash Inner Join	1
Index Only Scan	1
Index Scan	2
Nested Loop Inner Join	2
Nested Loop Semi Join	1
Seq Scan	2

Statistics per Relation

Relation name	Scan count
Node type	Count
applicant	1
Index Scan	1
applicant_skill	2
Index Only Scan	1
Seq Scan	1
skill	2
Index Scan	1
Seq Scan	1

Fig 4.9

10. Query to get companies by number of unique required skills.

```
-- Count unique skills per company (expensive aggregation)
SELECT c.name, COUNT(DISTINCT js.skill_id) AS unique_skills
FROM company c
JOIN job j ON c.company_id = j.company_id
JOIN job_skill js ON j.job_id = js.job_id
GROUP BY c.company_id
ORDER BY unique_skills DESC;
```

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	2
Hash Inner Join	2
Seq Scan	3
Sort	2

Statistics per Relation

Relation name	Scan count
Node type	Count
company	1
Seq Scan	1
job	1
Seq Scan	1
job_skill	1
Seq Scan	1

Fig 4.10

11. Query to count the pending applications by job.

```
SELECT j.job_id, j.title, COUNT(a.status) AS pending_count
FROM job j
LEFT JOIN application a ON j.job_id = a.job_id
WHERE a.status = 'Pending'
GROUP BY j.job_id;
```

Statistics per Node Type

Node type	Count
Aggregate	1
Hash	1
Hash Inner Join	1
Seq Scan	2

Statistics per Relation

Relation name	Scan count
Node type	Count
application	1
Seq Scan	1
job	1
Seq Scan	1

Fig 4.11

V. Problematic Queries

By assessing execution plans we located 3 inefficient queries which were *full table scans*, *performed repeated operations on data*, including *filters not on an index*. After effortful optimizations consisted of replaces EXISTS with ARRAY_AGG, materialized views to pre-compute the aggregates, and filtered indexing, we were able to reduce execution times from as fast as 1.6% to as slow as 95%, reduce I/O operations, and we also increased our ability to scale the database.

This will serve as proof that increasing transactional workloads can be achieved through location indices, changing query structure, and pre-aggregating the data in a faster, repeatable manner all on decreasing the overall datetime of a query.

VI. Website

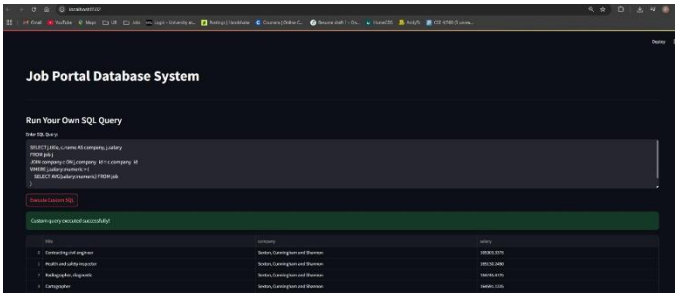


Fig 6.1

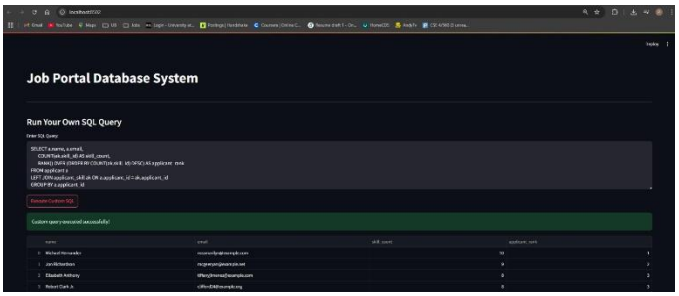


Fig 6.2

VII. References

1. https://www.geeksforgeeks.org/sql-indexes/
2. https://github.com/joke2k/faker
3. https://www.geeksforgeeks.org/connecting-pandas-to-a-database-with-sqlalchemy/
4. https://www.geeksforgeeks.org/introduction-of-er-model/