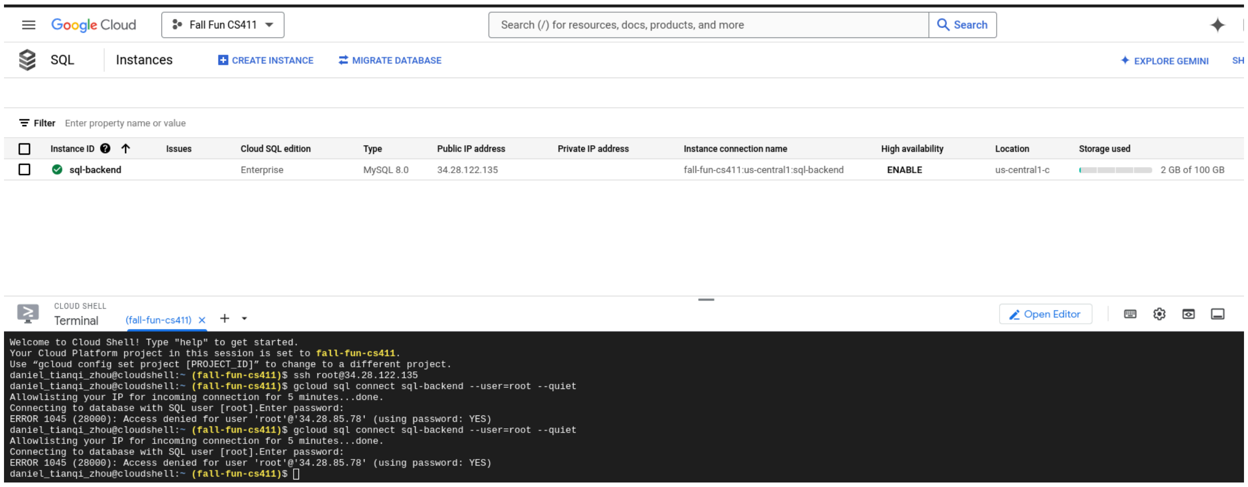
Stage 3 Database Implementation and Indexing

Part 1: Database implementation

is worth 10% and is graded as follows:

1. +4% for implementing the database tables locally or on GCP, you should provide a screenshot of the connection (i.e. showing your terminal/command-line information)
2. +4% for providing the DDL commands for your tables. (-0.5% for each mistake)
3. +2% for inserting at least 1000 rows in the tables. (You should do a count query to sÅhow this, -1% for each missing table)

GCP Connection Screenshot



Instance ID	Issues	Cloud SQL edition	Type	Public IP address	Private IP address	Instance connection name	High availability	Location	Storage used
sql-backend		Enterprise	MySQL 8.0	34.28.122.135		fall-fun-cs411:us-central1:sql-backend	ENABLE	us-central1-c	2 GB of 100 GB

```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to fall-fun-cs411.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
daniel_tianqi_zhou@cloudshell:~ (fall-fun-cs411)$ ssh root@34.28.122.135
daniel_tianqi_zhou@cloudshell:~ (fall-fun-cs411)$ gcloud sql connect sql-backend --user=root --quiet
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'34.28.85.78' (using password: YES)
daniel_tianqi_zhou@cloudshell:~ (fall-fun-cs411)$ gcloud sql connect sql-backend --user=root --quiet
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'34.28.85.78' (using password: YES)
daniel_tianqi_zhou@cloudshell:~ (fall-fun-cs411)$ 
```

DDL Create Table Command

~~~
### `applications` table
```sql
CREATE TABLE applications (
    posting_id INT,
    user_id INT,
    application_date DATETIME,
    PRIMARY KEY (posting_id, user_id),
    FOREIGN KEY (posting_id) REFERENCES posting(posting_id),
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```
~~~

`school` table
```sql
CREATE TABLE school (
    school_id INT PRIMARY KEY,
    school_name VARCHAR(256),
    school_size INT,
    school_address VARCHAR(256)
);
```

`leetcode_problem` table
```sql
CREATE TABLE leetcode_problem (
    problem_id INT PRIMARY KEY,
    title VARCHAR(64),
    description VARCHAR(256),
    company_id INT,
    url VARCHAR(256),
    frequency INT,
    rating INT,
    FOREIGN KEY (company_id) REFERENCES employer_companies(company_id)
);
```

`user` table
```sql
CREATE TABLE user (
    user_id INT PRIMARY KEY,
    school_id INT,
    company_id INT,
    year INT,
    user_name VARCHAR(256),
    skills VARCHAR(512),
    current_streak INT,
    points INT,
    FOREIGN KEY (school_id) REFERENCES school(school_id),
    FOREIGN KEY (company_id) REFERENCES employer_companies(company_id)
);
```

`posting` table
```sql
CREATE TABLE posting (
    posting_id INT PRIMARY KEY,
    job_name VARCHAR(256),
    job_description VARCHAR(256),
    med_salary INT,
    sponsor VARCHAR(32),
    remote_allowed VARCHAR(10),
    location VARCHAR(32),
    post_date DATETIME,
    ng_or_internship VARCHAR(32),
    company_id INT,
    FOREIGN KEY (company_id) REFERENCES employer_companies(company_id)
);
```

`employer_companies` table
```sql
CREATE TABLE employer_companies (
    company_id INT PRIMARY KEY,
    company_name VARCHAR(32),
    description VARCHAR(256),
    url VARCHAR(32),
    address VARCHAR(64)
);
```

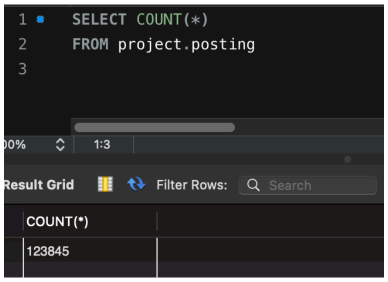
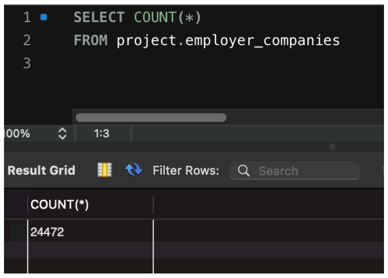
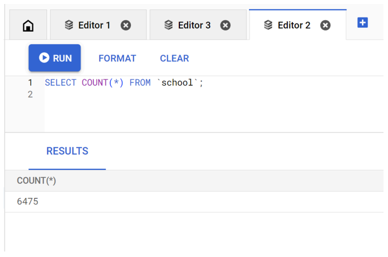
`interview_question` table
```sql
CREATE TABLE interview_question (
    problem_id INT,
    company_name VARCHAR(32),
    PRIMARY KEY (problem_id, company_name),
    FOREIGN KEY (problem_id) REFERENCES leetcode_problem(problem_id),
    FOREIGN KEY (company_name) REFERENCES employer_companies(company_name)
);
```

Inserting at least 1000 rows in 3 tables

Source
https://www.kaggle.com/datasets/joebeachcapital/us-colleges-and-universities
https://www.kaggle.com/datasets/arshkon/linkedin-job-postings/data

Summary
Table Name : # of rows inserted

posting : 123845	employer_companies : 24472	school : 6475

 `SELECT COUNT(*)`
`FROM project.posting`
COUNT(*)
123845
	 `SELECT COUNT(*)`
	`FROM project.employer_companies`
	COUNT(*)
	24472
		 `SELECT COUNT(*) FROM `school`;`
		COUNT(*)
		6475 Editor 1
		Editor 3
		Editor 2
		RUN
		FORMAT
		CLEAR
		RESULTS

Part 2: Advanced Queries

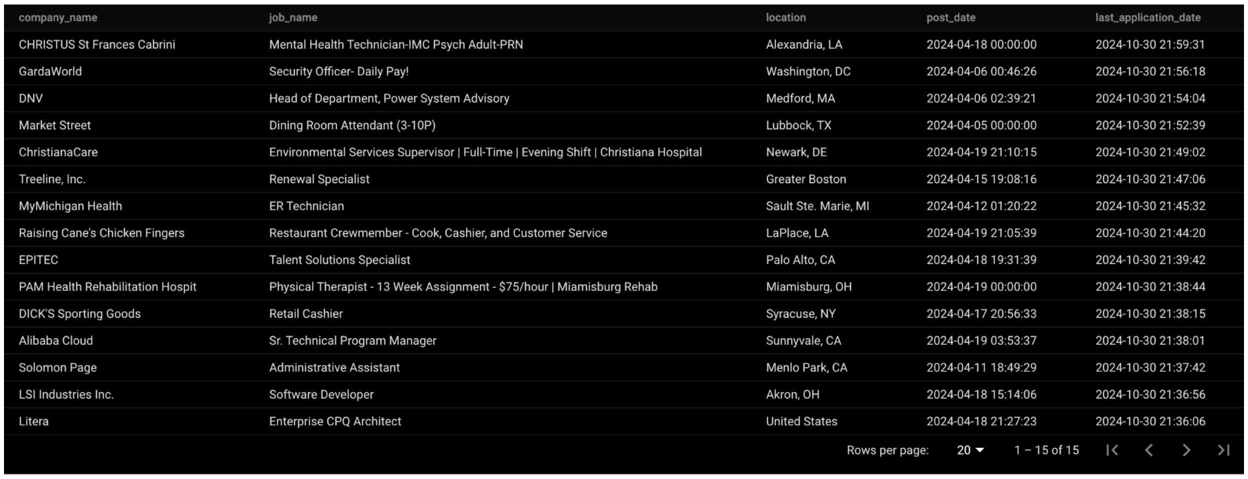
are worth 10% and are graded as follows:

1. +8% for developing four advanced queries (see point 4 for this stage, 2% each)
2. +2% for providing screenshots with the top 15 rows of the advanced query results (0.5% each)

1. **Query for Finding Active Job Posting (< 2 months)**

This query will return only postings with recent applications (within the last two months), including company name, job title, location, posting date, and the date of the last application received. This way, only currently active postings with recent interest are displayed.

```
SELECT ec.company_name, p.job_name, p.location, p.post_date,
MAX(a.application_date) AS last_application_date
FROM posting AS p
JOIN applications AS a ON p.posting_id = a.posting_id
JOIN employer_companies AS ec ON p.company_id = ec.company_id
GROUP BY p.posting_id, ec.company_name, p.job_name, p.location, p.postÅ_date
HAVING MAX(a.application_date) >= DATE_SUB(CURDATE(), INTERVAL 2 MONTH)
ORDER BY last_application_date DESC;
```



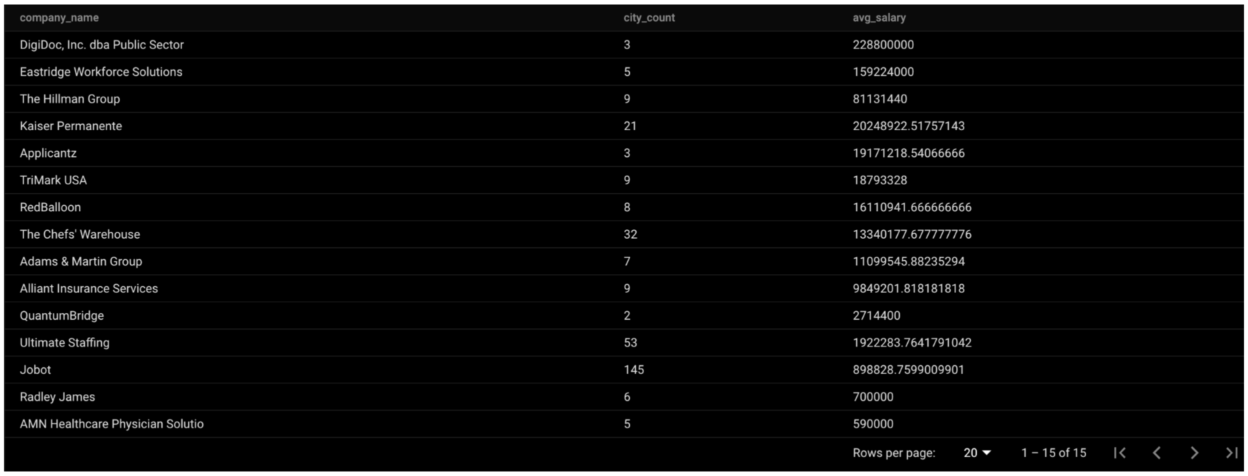
company_name	job_name	location	post_date	last_application_date
CHRISTUS St Frances Cabrini	Mental Health Technician-IMC Psych Adult-PRN	Alexandria, LA	2024-04-18 00:00:00	2024-10-30 21:59:31
GardaWorld	Security Officer- Daily Pay!	Washington, DC	2024-04-06 00:46:26	2024-10-30 21:56:18
DNV	Head of Department, Power System Advisory	Medford, MA	2024-04-06 02:39:21	2024-10-30 21:54:04
Market Street	Dining Room Attendant (3-10P)	Lubbock, TX	2024-04-05 00:00:00	2024-10-30 21:52:39
ChristianaCare	Environmental Services Supervisor | Full-Time | Evening Shift | Christiana Hospital	Newark, DE	2024-04-19 21:10:15	2024-10-30 21:49:02
Treeline, Inc.	Renewal Specialist	Greater Boston	2024-04-15 19:08:16	2024-10-30 21:47:06
MyMichigan Health	ER Technician	Sault Ste. Marie, MI	2024-04-12 01:20:22	2024-10-30 21:45:32
Raising Cane's Chicken Fingers	Restaurant Crewmember - Cook, Cashier, and Customer Service	LaPlace, LA	2024-04-19 21:05:39	2024-10-30 21:44:20
EPITEC	Talent Solutions Specialist	Palo Alto, CA	2024-04-18 19:31:39	2024-10-30 21:39:42
PAM Health Rehabilitation Hospit	Physical Therapist - 13 Week Assignment - $75/hour | Miamisburg Rehab	Miamisburg, OH	2024-04-19 00:00:00	2024-10-30 21:38:44
DICK'S Sporting Goods	Retail Cashier	Syracuse, NY	2024-04-17 20:56:33	2024-10-30 21:38:15
Alibaba Cloud	Sr. Technical Program Manager	Sunnyvale, CA	2024-04-19 03:53:37	2024-10-30 21:38:01
Solomon Page	Administrative Assistant	Menlo Park, CA	2024-04-11 18:49:29	2024-10-30 21:37:42
LSI Industries Inc.	Software Developer	Akron, OH	2024-04-18 15:14:06	2024-10-30 21:36:56
Litera	Enterprise CPQ Architect	United States	2024-04-18 21:27:23	2024-10-30 21:36:06

Rows per page: 20 1 – 15 of 15

2.Companies with Job Openings Above the Overall Average Salary and Hiring in Multiple Cities

This query finds companies that offer job postings with an average salary above the overall average across all postings and have job openings in more than one city. It uses aggregation with GROUP BY, a subquery to calculate the overall average salary, and a join.

```
SELECT ec.company_name, COUNT(DISTINCT p.location) AS city_count, AVG(p.med_salary)
AS avg_salary
FROM employer_companies AS ec
JOIN posting AS p ON ec.company_id = p.company_id
GROUP BY ec.company_name
HAVING AVG(p.med_salary) > (
   SELECT AVG(med_salary) FROM posting
) AND COUNT(DISTINCT p.location) > 1
ORDER BY avg_salary DESC;
```



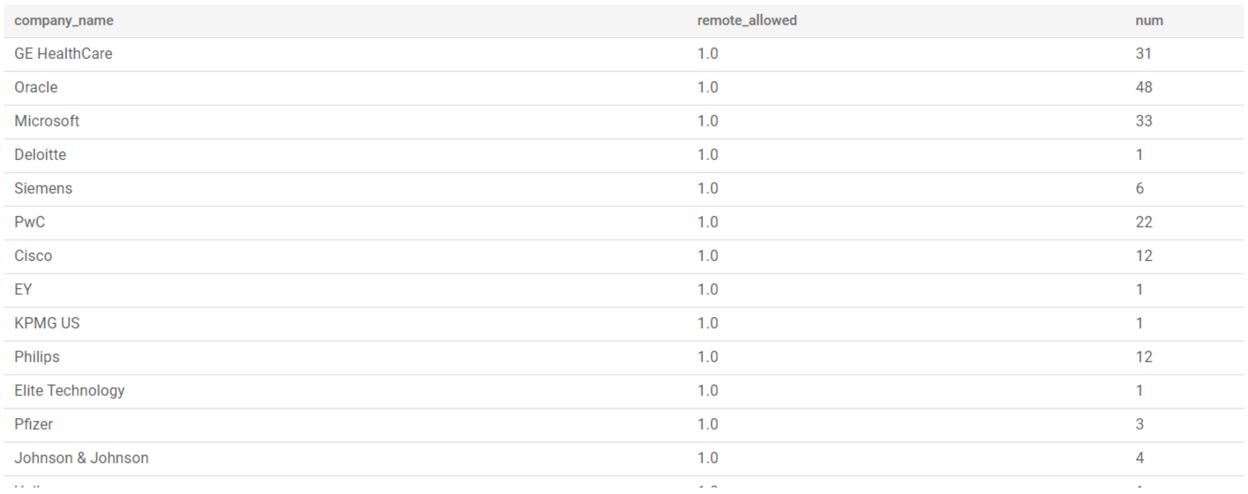
company_name	city_count	avg_salary
DigiDoc, Inc. dba Public Sector	3	228800000
Eastridge Workforce Solutions	5	159224000
The Hillman Group	9	81131440
Kaiser Permanente	21	20248922.51757143
Applicantz	3	19171218.54066666
TriMark USA	9	18793328
RedBalloon	8	16110941.666666666
The Chefs' Warehouse	32	13340177.677777776
Adams & Martin Group	7	11099545.88235294
Alliant Insurance Services	9	9849201.818181818
QuantumBridge	2	2714400
Ultimate Staffing	53	1922283.7641791042
Jobot	145	898828.7599009901
Radley James	6	700000
AMN Healthcare Physician Solutio	5	590000

---

3. **Query for Remote Job count offered by various companies**

This query retrieves the company names and remote work availability from job postings, along with a count of job postings (`num`) for each company. It leverages a natural join between the `posting` and `employer_companies` tables, grouping the results by `company_name` and `remote_allowed`. This aggregation provides insight into the number of job postings each company offers and whether remote work is an option for each listing.

```
SELECT company_name, remote_allowed, COUNT(*) AS num
FROM posting
NATURAL JOIN employer_companies
GROUP BY company_name, remote_allowed
HAVING remote_allowed = 1
ORDER BY num DESC;
```

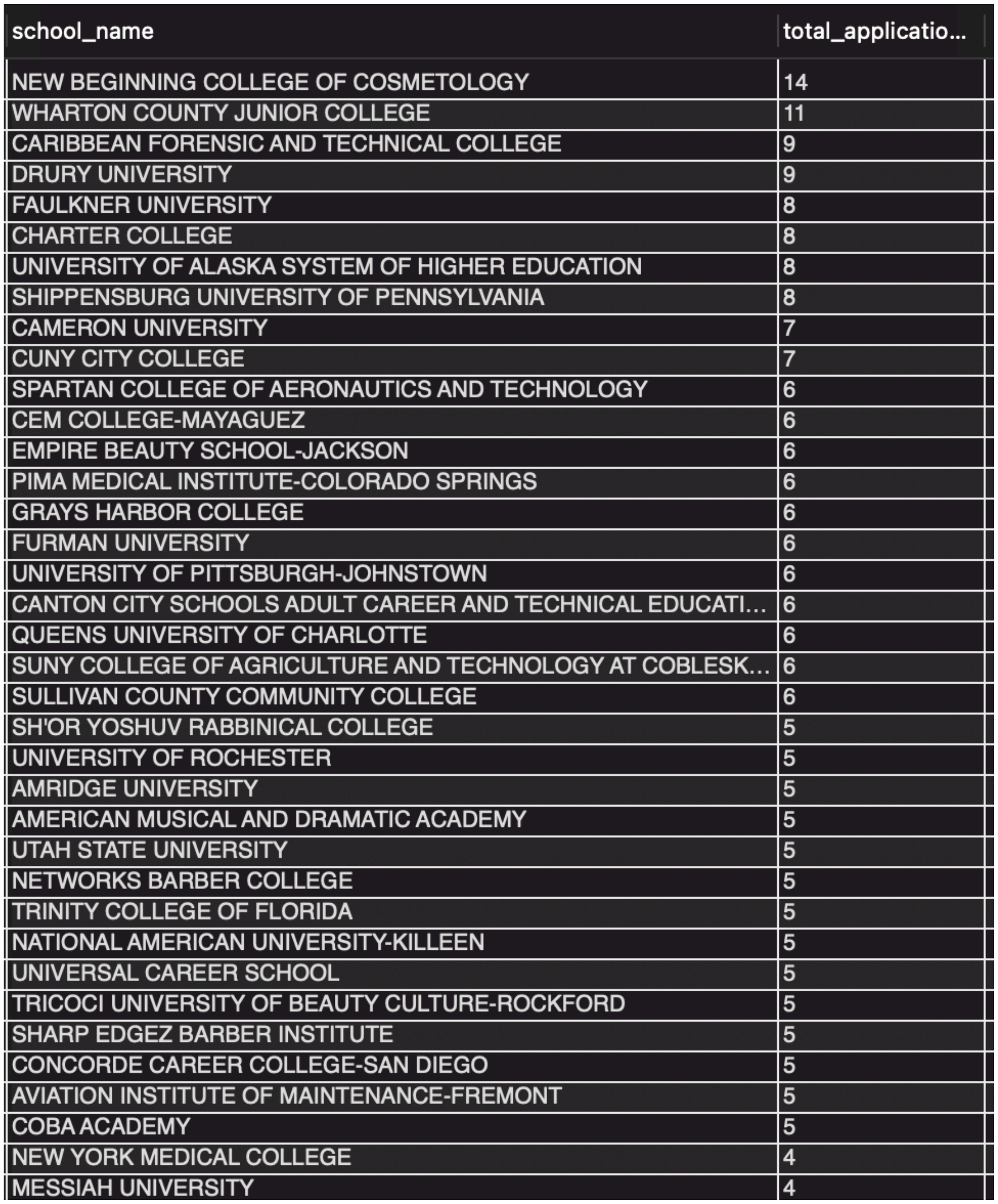


company_name	remote_allowed	num
GE HealthCare	1.0	31
Oracle	1.0	48
Microsoft	1.0	33
Deloitte	1.0	1
Siemens	1.0	6
PwC	1.0	22
Cisco	1.0	12
EY	1.0	1
KPMG US	1.0	1
Philips	1.0	12
Elite Technology	1.0	1
Pfizer	1.0	3
Johnson & Johnson	1.0	4

4. **Query for Top Schools with Most Applications Submitted**

This query lists schools by the number of job applications their students have submitted, highlighting institutions with high job-seeking activity.

```
SELECT s.school_name, s.school_size, COUNT(a.user_id) AS total_applications
FROM school s
JOIN user u ON s.school_id = u.school_id
JOIN applications a ON u.user_id = a.user_id
GROUP BY s.school_name, s.school_size
ORDER BY total_applications DESC;
```



school_name	total_applicatio…
NEW BEGINNING COLLEGE OF COSMETOLOGY	14
WHARTON COUNTY JUNIOR COLLEGE	11
CARIBBEAN FORENSIC AND TECHNICAL COLLEGE	9
DRURY UNIVERSITY	9
FAULKNER UNIVERSITY	8
CHARTER COLLEGE	8
UNIVERSITY OF ALASKA SYSTEM OF HIGHER EDUCATION	8
SHIPPENSBURG UNIVERSITY OF PENNSYLVANIA	8
CAMERON UNIVERSITY	7
CUNY CITY COLLEGE	7
SPARTAN COLLEGE OF AERONAUTICS AND TECHNOLOGY	6
CEM COLLEGE-MAYAGUEZ	6
EMPIRE BEAUTY SCHOOL-JACKSON	6
PIMA MEDICAL INSTITUTE-COLORADO SPRINGS	6
GRAYS HARBOR COLLEGE	6
FURMAN UNIVERSITY	6
UNIVERSITY OF PITTSBURGH-JOHNSTOWN	6
CANTON CITY SCHOOLS ADULT CAREER AND TECHNICAL EDUCATI…	6
QUEENS UNIVERSITY OF CHARLOTTE	6
SUNY COLLEGE OF AGRICULTURE AND TECHNOLOGY AT COBLESK…	6
SULLIVAN COUNTY COMMUNITY COLLEGE	6
SH'OR YOSHUV RABBINICAL COLLEGE	5
UNIVERSITY OF ROCHESTER	5
AMRIDGE UNIVERSITY	5
AMERICAN MUSICAL AND DRAMATIC ACADEMY	5
UTAH STATE UNIVERSITY	5
NETWORKS BARBER COLLEGE	5
TRINITY COLLEGE OF FLORIDA	5
NATIONAL AMERICAN UNIVERSITY-KILLEEN	5
UNIVERSAL CAREER SCHOOL	5
TRICOCI UNIVERSITY OF BEAUTY CULTURE-ROCKFORD	5
SHARP EDGEZ BARBER INSTITUTE	5
CONCORDE CAREER COLLEGE-SAN DIEGO	5
AVIATION INSTITUTE OF MAINTENANCE-FREMONT	5
COBA ACADEMY	5
NEW YORK MEDICAL COLLEGE	4
MESSIAH UNIVERSITY	4

Part 3: Indexing Analysis

is worth 10% and is graded as follows:

1. +3% on trying at least three different indexing designs (excluding the default index) for each advanced query.

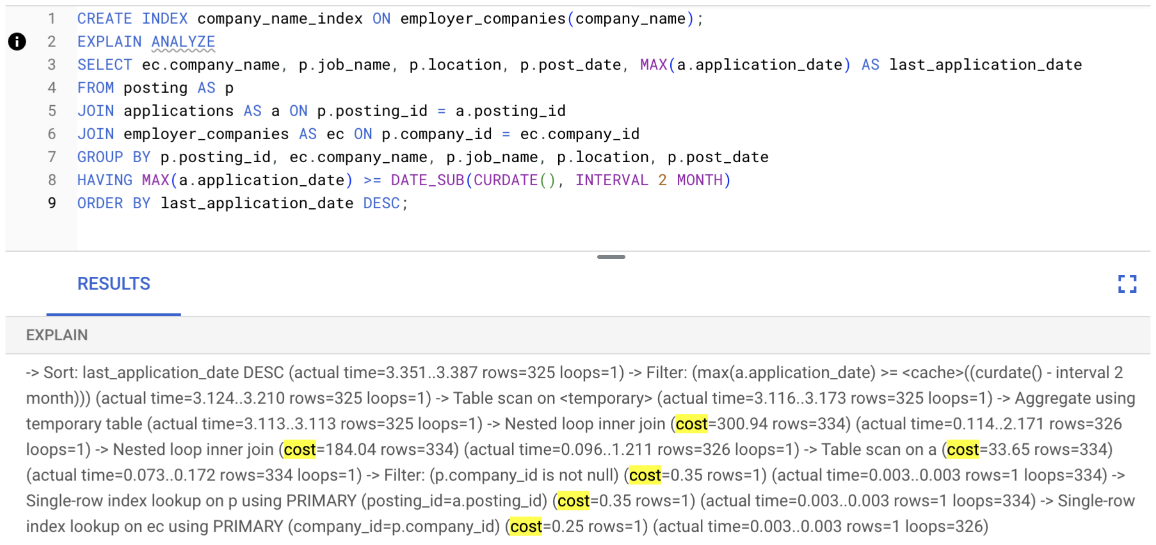
2. +5% on the indexing analysis reports which includes screenshots of the EXPLAIN ANALYZE commands.
3. +2% on the accuracy and thoroughness of the analyses.

1. **Query for Finding Active Job Posting (< 2 months)**

EXPLAIN ANALYZE without Indexing

```
-> Sort: last_application_date DESC (actual time=5.113..5.149 rows=325 loops=1) -> Filter: (max(a.application_date) >= <cache>((curdate() - interval 2 month))) (actual time=4.890..4.977 rows=325 loops=1) -> Table scan on <temporary> (actual time=4.880..4.938 rows=325 loops=1) -> Aggregate using temporary table (actual time=4.877..4.877 rows=325 loops=1) -> Nested loop inner join (cost=300.94 rows=334) (actual time=0.125..3.759 rows=326 loops=1) -> Nested loop inner join (cost=184.04 rows=334) (actual time=0.090..1.512 rows=326 loops=1) -> Table scan on a (cost=33.65 rows=334) (actual time=0.060..0.213 rows=334 loops=1) -> Filter: (p.company_id is not null) (cost=0.35 rows=1) (actual time=0.004..0.004 rows=1 loops=334) -> Single-row index lookup on p using PRIMARY (posting_id=a.posting_id) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on ec using PRIMARY (company_id=p.company_id) (cost=0.25 rows=1) (actual time=0.007..0.007 rows=1 loops=326)
```

1. Indexing on *company_name* from *employer_companies*

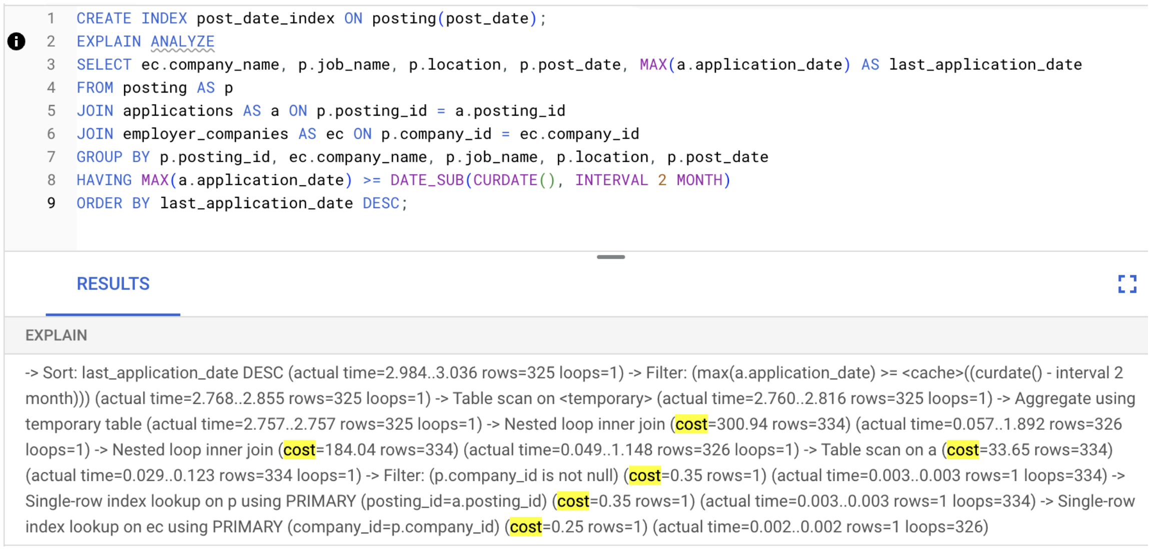


```sql
CREATE INDEX company_name_index ON employer_companies(company_name);
EXPLAIN ANALYZE
SELECT ec.company_name, p.job_name, p.location, p.post_date, MAX(a.application_date) AS last_application_date
FROM posting AS p
JOIN applications AS a ON p.posting_id = a.posting_id
JOIN employer_companies AS ec ON p.company_id = ec.company_id
GROUP BY p.posting_id, ec.company_name, p.job_name, p.location, p.post_date
HAVING MAX(a.application_date) >= DATE_SUB(CURDATE(), INTERVAL 2 MONTH)
ORDER BY last_application_date DESC;
```

```
-> Sort: last_application_date DESC (actual time=3.351..3.387 rows=325 loops=1) -> Filter: (max(a.application_date) >= <cache>((curdate() - interval 2 month))) (actual time=3.124..3.210 rows=325 loops=1) -> Table scan on <temporary> (actual time=3.116..3.173 rows=325 loops=1) -> Aggregate using temporary table (actual time=3.113..3.113 rows=325 loops=1) -> Nested loop inner join (cost=300.94 rows=334) (actual time=0.114..2.171 rows=326 loops=1) -> Nested loop inner join (cost=184.04 rows=334) (actual time=0.096..1.211 rows=326 loops=1) -> Table scan on a (cost=33.65 rows=334) (actual time=0.073..0.172 rows=334 loops=1) -> Filter: (p.company_id is not null) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on p using PRIMARY (posting_id=a.posting_id) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on ec using PRIMARY (company_id=p.company_id) (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=326)
```

- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

2. Indexing on *post_date* from *posting*



```sql
CREATE INDEX post_date_index ON posting(post_date);
EXPLAIN ANALYZE
SELECT ec.company_name, p.job_name, p.location, p.post_date, MAX(a.application_date) AS last_application_date
FROM posting AS p
JOIN applications AS a ON p.posting_id = a.posting_id
JOIN employer_companies AS ec ON p.company_id = ec.company_id
GROUP BY p.posting_id, ec.company_name, p.job_name, p.location, p.post_date
HAVING MAX(a.application_date) >= DATE_SUB(CURDATE(), INTERVAL 2 MONTH)
ORDER BY last_application_date DESC;
```

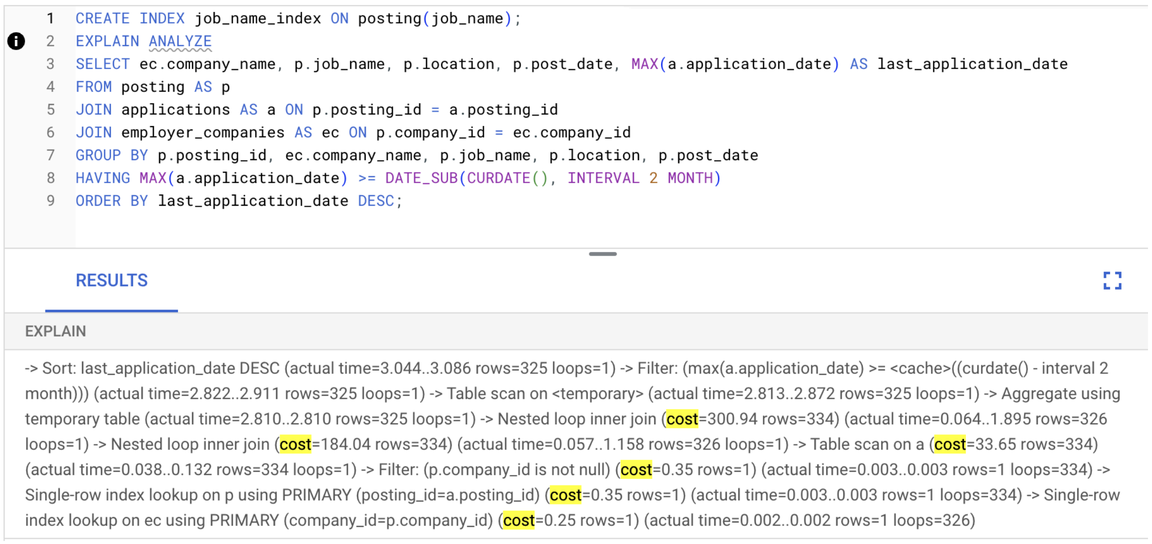
RESULTS

EXPLAIN

```
-> Sort: last_application_date DESC (actual time=2.984..3.036 rows=325 loops=1) -> Filter: (max(a.application_date) >= <cache>((curdate() - interval 2 month))) (actual time=2.768..2.855 rows=325 loops=1) -> Table scan on <temporary> (actual time=2.760..2.816 rows=325 loops=1) -> Aggregate using temporary table (actual time=2.757..2.757 rows=325 loops=1) -> Nested loop inner join (cost=300.94 rows=334) (actual time=0.057..1.892 rows=326 loops=1) -> Nested loop inner join (cost=184.04 rows=334) (actual time=0.049..1.148 rows=326 loops=1) -> Table scan on a (cost=33.65 rows=334) (actual time=0.029..0.123 rows=334 loops=1) -> Filter: (p.company_id is not null) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on p using PRIMARY (posting_id=a.posting_id) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on ec using PRIMARY (company_id=p.company_id) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=326)
```

- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

3. Indexing on *job_name* from *posting*



```sql
CREATE INDEX job_name_index ON posting(job_name);
EXPLAIN ANALYZE
SELECT ec.company_name, p.job_name, p.location, p.post_date, MAX(a.application_date) AS last_application_date
FROM posting AS p
JOIN applications AS a ON p.posting_id = a.posting_id
JOIN employer_companies AS ec ON p.company_id = ec.company_id
GROUP BY p.posting_id, ec.company_name, p.job_name, p.location, p.post_date
HAVING MAX(a.application_date) >= DATE_SUB(CURDATE(), INTERVAL 2 MONTH)
ORDER BY last_application_date DESC;
```

RESULTS

EXPLAIN

```
-> Sort: last_application_date DESC (actual time=3.044..3.086 rows=325 loops=1) -> Filter: (max(a.application_date) >= <cache>((curdate() - interval 2 month))) (actual time=2.822..2.911 rows=325 loops=1) -> Table scan on <temporary> (actual time=2.813..2.872 rows=325 loops=1) -> Aggregate using temporary table (actual time=2.810..2.810 rows=325 loops=1) -> Nested loop inner join (cost=300.94 rows=334) (actual time=0.064..1.895 rows=326 loops=1) -> Nested loop inner join (cost=184.04 rows=334) (actual time=0.057..1.158 rows=326 loops=1) -> Table scan on a (cost=33.65 rows=334) (actual time=0.038..0.132 rows=334 loops=1) -> Filter: (p.company_id is not null) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on p using PRIMARY (posting_id=a.posting_id) (cost=0.35 rows=1) (actual time=0.003..0.003 rows=1 loops=334) -> Single-row index lookup on ec using PRIMARY (company_id=p.company_id) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=326)
```

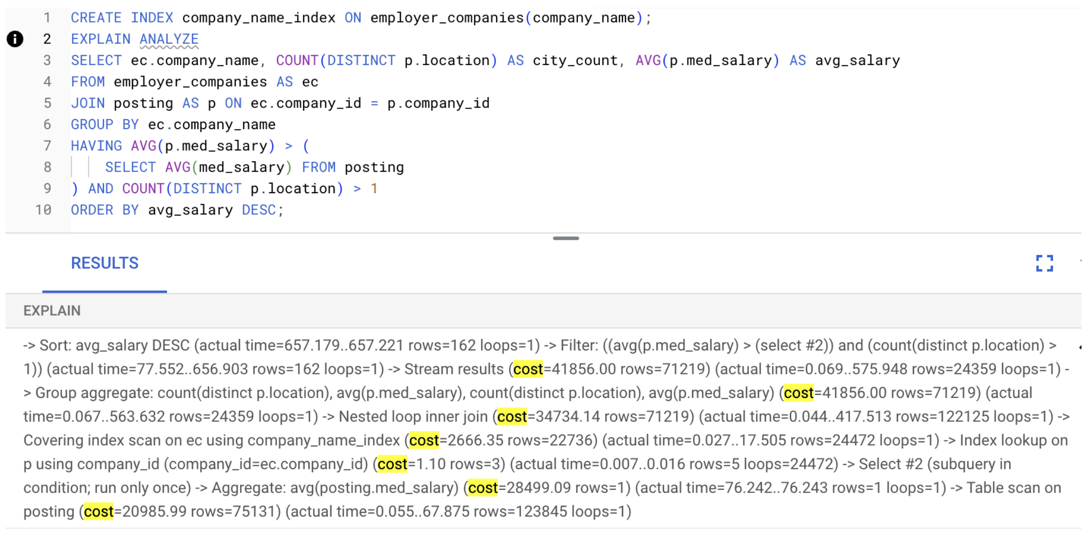
- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

2. Companies with Job Openings Above the Overall Average Salary and Hiring in Multiple Cities

EXPLAIN ANALYZE without Indexing

```
-> Sort: avg_salary DESC (actual time=690.916..690.932 rows=162 loops=1) -> Filter: ((avg(p.med_salary) > (select #2)) and (count(distinct p.location) >
1)) (actual time=120.189..690.652 rows=162 loops=1) -> Stream results (cost=41856.00 rows=71219) (actual time=35.618..601.821 rows=24359
loops=1) -> Group aggregate: count(distinct p.location), avg(p.med_salary), count(distinct p.location), avg(p.med_salary) (cost=41856.00 rows=71219)
(actual time=35.614..588.800 rows=24359 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=35.561..440.036 rows=122125
loops=1) -> Sort: ec.company_name (cost=2666.35 rows=22736) (actual time=35.505..39.597 rows=24472 loops=1) -> Table scan on ec (cost=2666.35
rows=22736) (actual time=0.051..12.801 rows=24472 loops=1) -> Index lookup on p using company_id (company_id=ec.company_id) (cost=1.10 rows=3)
(actual time=0.008..0.016 rows=5 loops=24472) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(posting.med_salary)
(cost=28499.09 rows=1) (actual time=83.297..83.298 rows=1 loops=1) -> Table scan on posting (cost=20985.99 rows=75131) (actual
time=0.084..74.140 rows=123845 loops=1)
```

1. Indexing on *company_name* from *employer_companies*

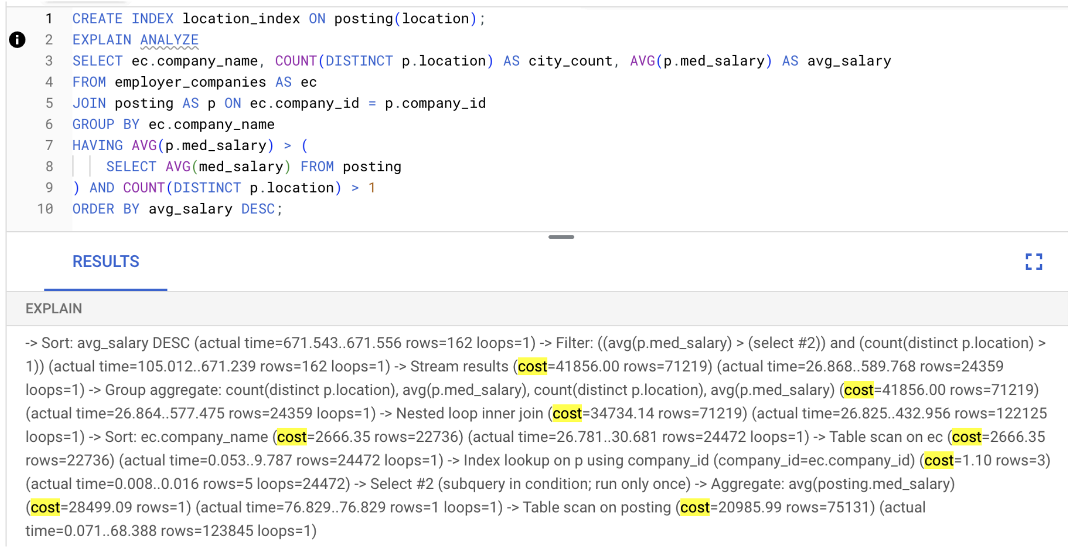


```sql
CREATE INDEX company_name_index ON employer_companies(company_name);
EXPLAIN ANALYZE
SELECT ec.company_name, COUNT(DISTINCT p.location) AS city_count, AVG(p.med_salary) AS avg_salary
FROM employer_companies AS ec
JOIN posting AS p ON ec.company_id = p.company_id
GROUP BY ec.company_name
HAVING AVG(p.med_salary) > (
    SELECT AVG(med_salary) FROM posting
) AND COUNT(DISTINCT p.location) > 1
ORDER BY avg_salary DESC;
```

```
-> Sort: avg_salary DESC (actual time=657.179..657.221 rows=162 loops=1) -> Filter: ((avg(p.med_salary) > (select #2)) and (count(distinct p.location) >
1)) (actual time=77.552..656.903 rows=162 loops=1) -> Stream results (cost=41856.00 rows=71219) (actual time=0.069..575.948 rows=24359 loops=1) -
> Group aggregate: count(distinct p.location), avg(p.med_salary), count(distinct p.location), avg(p.med_salary) (cost=41856.00 rows=71219) (actual
time=0.067..563.632 rows=24359 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=0.044..417.513 rows=122125 loops=1) ->
Covering index scan on ec using company_name_index (cost=2666.35 rows=22736) (actual time=0.027..17.505 rows=24472 loops=1) -> Index lookup on
p using company_id (company_id=ec.company_id) (cost=1.10 rows=3) (actual time=0.007..0.016 rows=5 loops=24472) -> Select #2 (subquery in
condition; run only once) -> Aggregate: avg(posting.med_salary) (cost=28499.09 rows=1) (actual time=76.242..76.243 rows=1 loops=1) -> Table scan on
posting (cost=20985.99 rows=75131) (actual time=0.055..67.875 rows=123845 loops=1)
```

- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

2. Indexing on *location* from *posting*

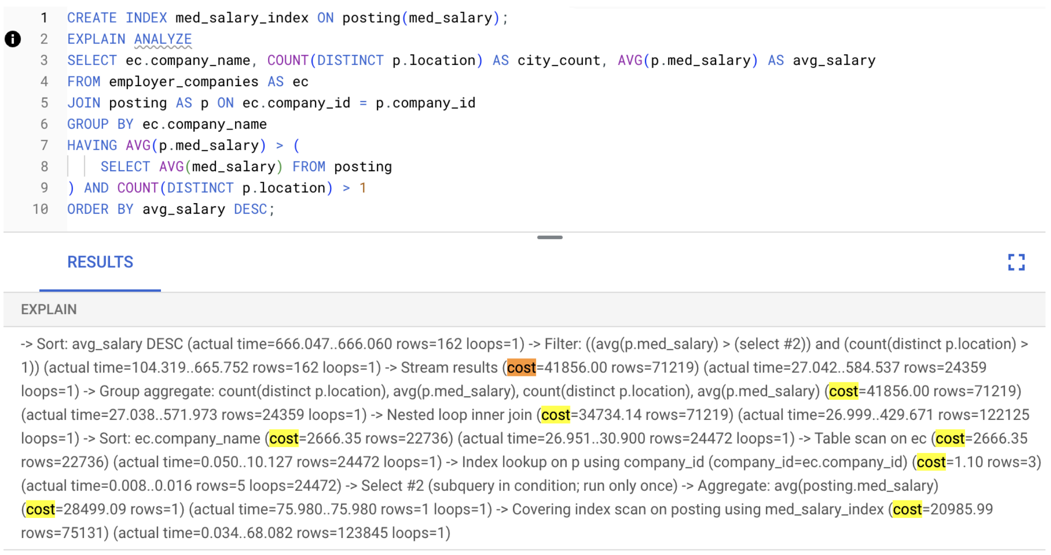


```sql
CREATE INDEX location_index ON posting(location);
EXPLAIN ANALYZE
SELECT ec.company_name, COUNT(DISTINCT p.location) AS city_count, AVG(p.med_salary) AS avg_salary
FROM employer_companies AS ec
JOIN posting AS p ON ec.company_id = p.company_id
GROUP BY ec.company_name
HAVING AVG(p.med_salary) > (
    SELECT AVG(med_salary) FROM posting
) AND COUNT(DISTINCT p.location) > 1
ORDER BY avg_salary DESC;
```

```
-> Sort: avg_salary DESC (actual time=671.543..671.556 rows=162 loops=1) -> Filter: ((avg(p.med_salary) > (select #2)) and (count(distinct p.location) >
1)) (actual time=105.012..671.239 rows=162 loops=1) -> Stream results (cost=41856.00 rows=71219) (actual time=26.868..589.768 rows=24359
loops=1) -> Group aggregate: count(distinct p.location), avg(p.med_salary), count(distinct p.location), avg(p.med_salary) (cost=41856.00 rows=71219)
(actual time=26.864..577.475 rows=24359 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=26.825..432.956 rows=122125
loops=1) -> Sort: ec.company_name (cost=2666.35 rows=22736) (actual time=26.781..30.681 rows=24472 loops=1) -> Table scan on ec (cost=2666.35
rows=22736) (actual time=0.053..9.787 rows=24472 loops=1) -> Index lookup on p using company_id (company_id=ec.company_id) (cost=1.10 rows=3)
(actual time=0.008..0.016 rows=5 loops=24472) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(posting.med_salary)
(cost=28499.09 rows=1) (actual time=76.829..76.829 rows=1 loops=1) -> Table scan on posting (cost=20985.99 rows=75131) (actual
time=0.071..68.388 rows=123845 loops=1)
```

- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

3. Indexing on *med_salary* from *posting*



```sql
CREATE INDEX med_salary_index ON posting(med_salary);
EXPLAIN ANALYZE
SELECT ec.company_name, COUNT(DISTINCT p.location) AS city_count, AVG(p.med_salary) AS avg_salary
FROM employer_companies AS ec
JOIN posting AS p ON ec.company_id = p.company_id
GROUP BY ec.company_name
HAVING AVG(p.med_salary) > (
    SELECT AVG(med_salary) FROM posting
) AND COUNT(DISTINCT p.location) > 1
ORDER BY avg_salary DESC;
```

RESULTS

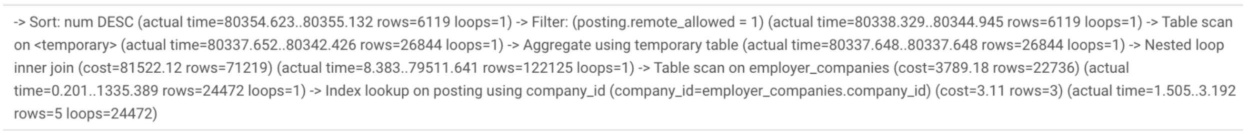
EXPLAIN

```
-> Sort: avg_salary DESC (actual time=666.047..666.060 rows=162 loops=1) -> Filter: ((avg(p.med_salary) > (select #2)) and (count(distinct p.location) > 1)) (actual time=104.319..665.752 rows=162 loops=1) -> Stream results (cost=41856.00 rows=71219) (actual time=27.042..584.537 rows=24359 loops=1) -> Group aggregate: count(distinct p.location), avg(p.med_salary), count(distinct p.location), avg(p.med_salary) (cost=41856.00 rows=71219) (actual time=27.038..571.973 rows=24359 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=26.999..429.671 rows=122125 loops=1) -> Sort: ec.company_name (cost=2666.35 rows=22736) (actual time=26.951..30.900 rows=24472 loops=1) -> Table scan on ec (cost=2666.35 rows=22736) (actual time=0.050..10.127 rows=24472 loops=1) -> Index lookup on p using company_id (company_id=ec.company_id) (cost=1.10 rows=3) (actual time=0.008..0.016 rows=5 loops=24472) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(posting.med_salary) (cost=28499.09 rows=1) (actual time=75.980..75.980 rows=1 loops=1) -> Covering index scan on posting using med_salary_index (cost=20985.99 rows=75131) (actual time=0.034..68.082 rows=123845 loops=1)
```

- From the results, we found that this index configuration doesn't change the performance as there is no difference in the costs so we drop this index.

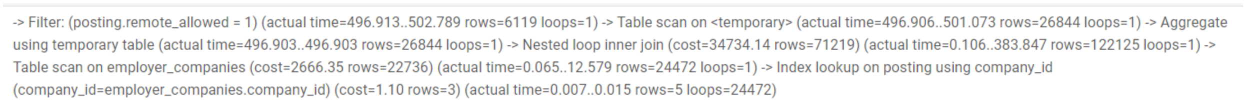
3. **Query for Remote Job count offered by various companies**

First we run EXPLAIN ANALYZE on the current query as is.



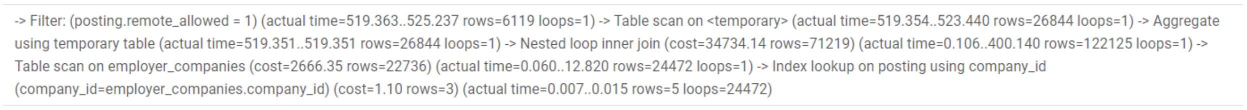
```
-> Sort: num DESC (actual time=80354.623..80355.132 rows=6119 loops=1) -> Filter: (posting.remote_allowed = 1) (actual time=80338.329..80344.945 rows=6119 loops=1) -> Table scan on <temporary> (actual time=80337.652..80342.426 rows=26844 loops=1) -> Aggregate using temporary table (actual time=80337.648..80337.648 rows=26844 loops=1) -> Nested loop inner join (cost=81522.12 rows=71219) (actual time=8.383..79511.641 rows=122125 loops=1) -> Table scan on employer_companies (cost=3789.18 rows=22736) (actual time=0.201..1335.389 rows=24472 loops=1) -> Index lookup on posting using company_id (company_id=employer_companies.company_id) (cost=3.11 rows=3) (actual time=1.505..3.192 rows=5 loops=24472)
```

1. Given the query that we have, all the JOIN attributes are already primary keys which we are trying to avoid adding an index for. That leaves *remote_allowed* on the posting table. We add an index for *remote_allowed* and run the query again.



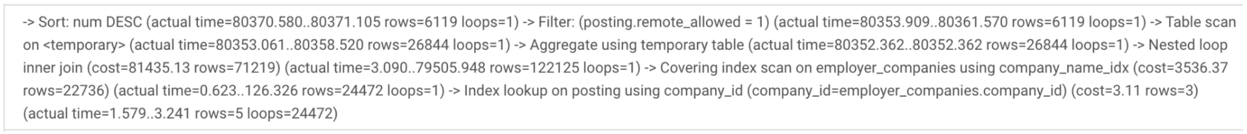
```
-> Filter: (posting.remote_allowed = 1) (actual time=496.913..502.789 rows=6119 loops=1) -> Table scan on <temporary> (actual time=496.906..501.073 rows=26844 loops=1) -> Aggregate using temporary table (actual time=496.903..496.903 rows=26844 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=0.106..383.847 rows=122125 loops=1) -> Table scan on employer_companies (cost=2666.35 rows=22736) (actual time=0.065..12.579 rows=24472 loops=1) -> Index lookup on posting using company_id (company_id=employer_companies.company_id) (cost=1.10 rows=3) (actual time=0.007..0.015 rows=5 loops=24472)
```

2. Then we try a composite index of the posting_id and remote_allowed:



```
-> Filter: (posting.remote_allowed = 1) (actual time=519.363..525.237 rows=6119 loops=1) -> Table scan on <temporary> (actual time=519.354..523.440 rows=26844 loops=1) -> Aggregate using temporary table (actual time=519.351..519.351 rows=26844 loops=1) -> Nested loop inner join (cost=34734.14 rows=71219) (actual time=0.106..400.140 rows=122125 loops=1) -> Table scan on employer_companies (cost=2666.35 rows=22736) (actual time=0.060..12.820 rows=24472 loops=1) -> Index lookup on posting using company_id (company_id=employer_companies.company_id) (cost=1.10 rows=3) (actual time=0.007..0.015 rows=5 loops=24472)
```

3. We also try to add company_name as an index:



```
-> Sort: num DESC (actual time=80370.580..80371.105 rows=6119 loops=1) -> Filter: (posting.remote_allowed = 1) (actual time=80353.909..80361.570 rows=6119 loops=1) -> Table scan on <temporary> (actual time=80353.061..80358.520 rows=26844 loops=1) -> Aggregate using temporary table (actual time=80352.362..80352.362 rows=26844 loops=1) -> Nested loop inner join (cost=81435.13 rows=71219) (actual time=3.090..79505.948 rows=122125 loops=1) -> Covering index scan on employer_companies using company_name_idx (cost=3536.37 rows=22736) (actual time=0.623..126.326 rows=24472 loops=1) -> Index lookup on posting using company_id (company_id=employer_companies.company_id) (cost=3.11 rows=3) (actual time=1.579..3.241 rows=5 loops=24472)
```

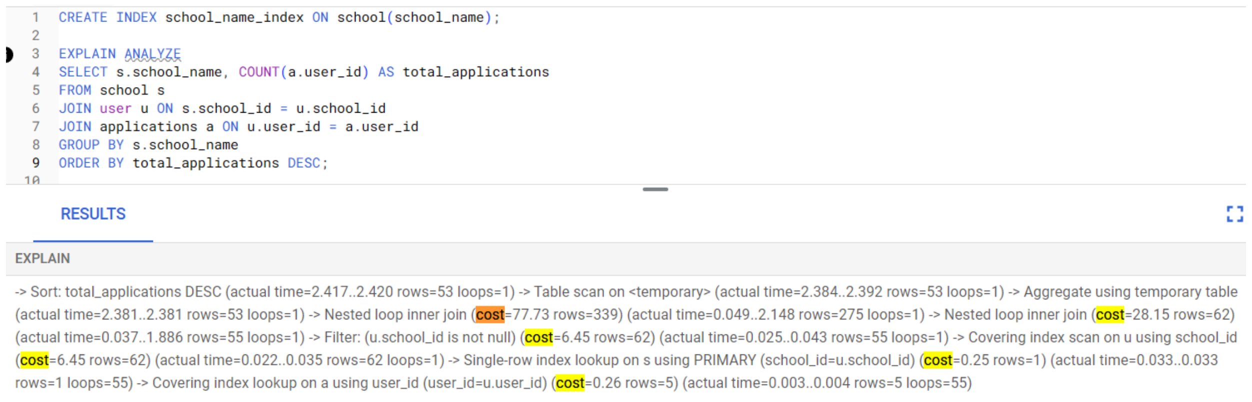
Since company_name yields a slight improvement, we keep company_name as an index.

4. **Query for Top Schools with Most Applications Submitted**

First we run EXPLAIN ANALYZE on the current query as is.

```
-> Sort: total_applications DESC (actual time=0.862..0.866 rows=53 loops=1) -> Table scan on <temporary> (actual time=0.826..0.834 rows=53 loops=1) -> Aggregate using temporary table (actual time=0.825..0.825 rows=53 loops=1) -> Nested loop inner join (cost=127.59 rows=449) (actual time=0.090..0.547 rows=275 loops=1) -> Nested loop inner join (cost=62.01 rows=82) (actual time=0.075..0.343 rows=55 loops=1) -> Filter: (u.school_id is not null) (cost=8.45 rows=82) (actual time=0.054..0.068 rows=55 loops=1) -> Covering index scan on u using school_id (cost=8.45 rows=82) (actual time=0.047..0.057 rows=82 loops=1) -> Single-row index lookup on s using PRIMARY (school_id=u.school_id) (cost=0.55 rows=1) (actual time=0.005..0.005 rows=1 loops=55) -> Covering index lookup on a using user_id (user_id=u.user_id) (cost=0.26 rows=5) (actual time=0.002..0.003 rows=5 loops=55)
```

1. Given the query that we have, all the JOIN attributes are already primary keys which we are trying to avoid adding an index for. That leaves *school_name* on the school table. We add an index for *school_name* and run the query again.



```sql
CREATE INDEX school_name_index ON school(school_name);

EXPLAIN ANALYZE
SELECT s.school_name, COUNT(a.user_id) AS total_applications
FROM school s
JOIN user u ON s.school_id = u.school_id
JOIN applications a ON u.user_id = a.user_id
GROUP BY s.school_name
ORDER BY total_applications DESC;
```

```
-> Sort: total_applications DESC (actual time=2.417..2.420 rows=53 loops=1) -> Table scan on <temporary> (actual time=2.384..2.392 rows=53 loops=1) -> Aggregate using temporary table (actual time=2.381..2.381 rows=53 loops=1) -> Nested loop inner join (cost=77.73 rows=339) (actual time=0.049..2.148 rows=275 loops=1) -> Nested loop inner join (cost=28.15 rows=62) (actual time=0.037..1.886 rows=55 loops=1) -> Filter: (u.school_id is not null) (cost=6.45 rows=62) (actual time=0.025..0.043 rows=55 loops=1) -> Covering index scan on u using school_id (cost=6.45 rows=62) (actual time=0.022..0.035 rows=62 loops=1) -> Single-row index lookup on s using PRIMARY (school_id=u.school_id) (cost=0.25 rows=1) (actual time=0.033..0.033 rows=1 loops=55) -> Covering index lookup on a using user_id (user_id=u.user_id) (cost=0.26 rows=5) (actual time=0.003..0.004 rows=5 loops=55)
```

We see no difference here. The explanation is simple, each school has a unique name and since we are querying through every school, the query must go through every row anyways. Therefore, we don't keep this index.

2. We then try a composite index of the school id and school name. There is no difference either.

```
-> Sort: total_applications DESC (actual time=1.060..1.065 rows=53 loops=1) -> Table scan on <temporary> (actual time=1.017..1.028 rows=53 loops=1) -> Aggregate using temporary table (actual time=1.015..1.015 rows=53 loops=1) -> Nested loop inner join (cost=103.48 rows=449) (actual time=0.096..0.691 rows=275 loops=1) -> Nested loop inner join (cost=37.90 rows=82) (actual time=0.083..0.350 rows=55 loops=1) -> Filter: (u.school_id is not null) (cost=9.20 rows=82) (actual time=0.062..0.084 rows=55 loops=1) -> Covering index scan on u using school_id (cost=9.20 rows=82) (actual time=0.053..0.074 rows=82 loops=1) -> Single-row index lookup on s using PRIMARY (school_id=u.school_id) (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=55) -> Covering index lookup on a using user_id (user_id=u.user_id) (cost=0.26 rows=5) (actual time=0.004..0.005 rows=5 loops=55)
```

3. We add an index on school_size.

```
-> Sort: total_applications DESC (actual time=25.046..25.050 rows=53 loops=1) -> Table scan on <temporary> (actual time=24.992..25.006 rows=53 loops=1) -> Aggregate using temporary table (actual time=24.988..24.988 rows=53 loops=1) -> Nested loop inner join (cost=227.01 rows=449) (actual time=4.458..24.321 rows=275 loops=1) -> Nested loop inner join (cost=99.40 rows=82) (actual time=3.138..22.447 rows=55 loops=1) -> Filter: (u.school_id is not null) (cost=9.20 rows=82) (actual time=1.593..2.603 rows=55 loops=1) -> Covering index scan on u using school_id (cost=9.20 rows=82) (actual time=1.583..2.536 rows=82 loops=1) -> Single-row index lookup on s using PRIMARY (school_id=u.school_id) (cost=1.00 rows=1) (actual time=0.360..0.360 rows=1 loops=55) -> Covering index lookup on a using user_id (user_id=u.user_id) (cost=1.02 rows=5) (actual time=0.031..0.033 rows=5 loops=55)
```

Since none of them improves the query's costs, we retain the default index design.