

Part 1: Database Implementation

1.1 Implement at least five main tables. These tables represent the relational schema you provided in Stage 2.

We created 7 database tables in Google Cloud Platform: users, zip_zones, locations, plots, crops, planting_plans, and weather_observations.

1.2 In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you used to create each of these tables in the database.

Below are the DDL commands for our database tables

```
CREATE TABLE users (  
  user_id          INT NOT NULL AUTO_INCREMENT,  
  email            VARCHAR(320) UNIQUE NOT NULL,  
  name             VARCHAR(255) NOT NULL,  
  password         VARCHAR(320) NOT NULL,  
  created_at       DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,  
  PRIMARY KEY (user_id)  
);
```

```
CREATE TABLE zip_zones(  
  zip              CHAR(5) NOT NULL,  
  usda_zone        VARCHAR(10) NOT NULL,  
  PRIMARY KEY (zip)  
);
```

```
CREATE TABLE locations (  
  location_id      INT NOT NULL AUTO_INCREMENT,  
  city             VARCHAR(100) NOT NULL,  
  state            VARCHAR(50) NOT NULL,  
  zip              CHAR(5),  
  PRIMARY KEY (location_id),  
  FOREIGN KEY (zip) REFERENCES zip_zones(zip) ON DELETE SET NULL  
);
```

```
CREATE TABLE plots (  
  plot_id          INT NOT NULL AUTO_INCREMENT,  
  user_id          INT NOT NULL,  
  location_id      INT NOT NULL,  
  plot_name        VARCHAR(100) NOT NULL,  
  area_sq_m        DECIMAL,
```

```

        soil_type                VARCHAR(50),
        sunlight_exposure        VARCHAR(100),
        is_active                BOOLEAN,
        PRIMARY KEY (plot_id),
        FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
        FOREIGN KEY (location_id) REFERENCES locations(location_id) ON DELETE
CASCADE
    );

```

```

CREATE TABLE crops (
    crop_id                    INT NOT NULL AUTO_INCREMENT,
    ideal_soil                VARCHAR(100),
    sun_req                  VARCHAR(20),
    water_req                VARCHAR(20),
    common_name              VARCHAR(100),
    scientific_name          VARCHAR(100),
    usda_zone_min            INT,
    usda_zone_max            INT,
    PRIMARY KEY (crop_id)
);

```

```

CREATE TABLE planting_plans (
    plot_id                  INT NOT NULL,
    crop_id                  INT NOT NULL,
    start_date               DATE NOT NULL,
    end_date                 DATE NOT NULL,
    amount                   INT,
    PRIMARY KEY (plot_id, crop_id),
    FOREIGN KEY (plot_id) REFERENCES plots(plot_id) ON DELETE CASCADE,
    FOREIGN KEY (crop_id) REFERENCES crops(crop_id) ON DELETE CASCADE
);

```

```

CREATE TABLE weather_observations (
    obs_id                  INT NOT NULL AUTO_INCREMENT,
    location_id              INT NOT NULL,
    observed_on              DATE NOT NULL,
    tmin_c                  DECIMAL NOT NULL,
    tmax_c                  DECIMAL NOT NULL,
    precip_mm               DECIMAL,
    humidity_pct             DECIMAL,
    frost_flag               BOOLEAN,
    source                   VARCHAR(40) NOT NULL,
    PRIMARY KEY (obs_id),

```

```
FOREIGN KEY (location_id) REFERENCES locations(location_id) ON DELETE  
CASCADE
```

1.3 Insert data into these tables. You should insert at least 1000 rows in three different tables each. Try to use real data, but if you cannot find a good dataset for a particular table, you may use auto-generated data.

Inserted data into all our database tables. Please look at a25-cs411-team016-scaw
/data for all the data that was inserted

```
MySQL [db]> SELECT COUNT(*) FROM crops;  
+-----+  
| COUNT(*) |  
+-----+  
|      1000 |  
+-----+  
1 row in set (0.012 sec)  
  
MySQL [db]> SELECT COUNT(*) FROM weather_observations;  
+-----+  
| COUNT(*) |  
+-----+  
|      1000 |  
+-----+  
1 row in set (0.003 sec)  
  
MySQL [db]> SELECT COUNT(*) FROM locations;  
+-----+  
| COUNT(*) |  
+-----+  
|     29754 |  
+-----+  
1 row in set (0.031 sec)
```

1.4/1.5 Implemented 4 advanced SQL queries. Below are our queries and screenshots:

Advanced Query 1:

```
SELECT c.common_name, c.scientific_name FROM crops c JOIN zip_zones z  
      ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max WHERE z.zip  
= "94539" AND c.sun_req = "Partial sun/shade"  
UNION
```

```
SELECT c.common_name, c.scientific_name FROM crops c JOIN zip_zones z ON  
z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max WHERE z.zip = "94539"  
AND c.sun_req = "Full sun";
```

This query would be used if a user wants to know plants that would be suitable in their zip code that also thrive in full sun and/or partial shade. In this example, the location is Fremont, California, and the zip code is 94539.

```
MySQL [db]> SELECT c.common_name, c.scientific_name FROM crops c JOIN zip_zones z ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max WHERE z.zip = "94539" AND c.sun_req = "
Partial sun/shade" UNION SELECT c.common_name, c.scientific_name FROM crops c JOIN zip_zones z ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max WHERE z.zip = "94539" A
ND c.sun_req = "Full sun" LIMIT 15;
+-----+
| common_name | scientific_name |
+-----+
| Mountain pepper | Tasmannia lanceolata |
| Celery | Apium graveolens |
| Allium tricoccum | Allium tricoccum |
| Vernicia fordii | Vernicia fordii |
| Cucubalus baccifer | Cucubalus baccifer |
| Phyllostachys nigra henonis | Phyllostachys nigra henonis |
| Phyllostachys nigra punctata | Phyllostachys nigra punctata |
| Pea | Pisum sativum |
| Common bean | Phaseolus vulgaris |
| Peach | Prunus persica |
| Sage | Salvia officinalis |
| Black locust | Robinia pseudoacacia |
| Black walnut | Juglans nigra |
| English walnut | Juglans regia |
| Honey locust | Gleditsia triacanthos |
+-----+
15 rows in set, 20 warnings (0.004 sec)
```

Advanced Query 2:

```
SELECT l.city, l.state, w.observed_on, 'heat_wave' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.tmax_c >= 35
```

UNION

```
SELECT l.city, l.state, w.observed_on, 'hard_freeze' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.tmin_c <= 30
```

UNION

```
SELECT l.city, l.state, w.observed_on, 'very_heavy_rain' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.precip_mm >= 50
ORDER BY state, city, observed_on, event;
```

```
MySQL [db]> SELECT l.city, l.state, w.observed_on, 'heat wave' AS event FROM weather_observations AS w JOIN locations AS l ON l.location_id = w.location_id WHERE w.tmax_c >= 35 UNION
SELECT l.city, l.state, w.observed_on, 'hard freeze' AS event FROM weather_observations AS w JOIN locations AS l ON l.location_id = w.location_id WHERE w.tmin_c <= 0 UNION
SELECT l.city, l.state, w.observed_on, 'very heavy rain' AS event FROM weather_observations AS w JOIN locations AS l ON l.location_id = w.location_id WHERE w.precip_mm >= 50 ORDER BY
state, city, observed_on, event LIMIT 15;
```

city	state	observed_on	event
PHOENIX	AZ	2025-05-03	heat wave
PHOENIX	AZ	2025-05-08	heat wave
PHOENIX	AZ	2025-05-09	heat wave
PHOENIX	AZ	2025-05-10	heat wave
PHOENIX	AZ	2025-05-11	heat wave
PHOENIX	AZ	2025-05-12	heat wave
PHOENIX	AZ	2025-05-20	heat wave
PHOENIX	AZ	2025-05-21	heat wave
PHOENIX	AZ	2025-05-22	heat wave
PHOENIX	AZ	2025-05-23	heat wave
PHOENIX	AZ	2025-05-24	heat wave
PHOENIX	AZ	2025-05-25	heat wave
PHOENIX	AZ	2025-05-26	heat wave
PHOENIX	AZ	2025-05-27	heat wave
PHOENIX	AZ	2025-05-28	heat wave

15 rows in set (0.004 sec)

This query returns the city, state, and date of extreme weather events, such as heat waves, hard freeze, and very heavy rain.

Advanced Query 3:

```
SELECT u.user_id, u.email
FROM users AS u
LEFT JOIN plots AS p
  ON p.user_id = u.user_id
  AND p.is_active = 1
GROUP BY u.user_id, u.email
HAVING COUNT(p.plot_id) = 0
LIMIT 15;
```

This query returns the users that have no active plots.

```
MySQL [db]> SELECT u.user_id, u.email
-> FROM users AS u
-> LEFT JOIN plots AS p
->   ON p.user_id = u.user_id
->   AND p.is_active = 1
-> GROUP BY u.user_id, u.email
-> HAVING COUNT(p.plot_id) = 0
-> LIMIT 15;
```

user_id	email
712	abigail.baker5356@demo.net
831	abigail.flores9764@mail.com
150	abigail.hernandez8285@example.com
487	abigail.howard6748@example.com
188	abigail.jimenez4041@mail.com
182	abigail.lewis2857@test.org
176	abigail.long2762@inbox.com
171	abigail.morris1835@test.org
756	abigail.patel@mail.com
231	abigail.phillips@demo.net
1000	abigail.sanchez8155@inbox.com
331	abigail.torres1066@test.org
966	abigail.turner4106@inbox.com
477	addison.davis@example.com
164	addison.gomez6619@test.org

15 rows in set (0.005 sec)

Advanced Query 4:

```
SELECT
    l.city,
    l.state,
    ROUND(AVG(w.tmax_c), 1) AS avg_high,
    ROUND(STDDEV_SAMP(w.tmax_c), 2) AS stdev_high,
    COUNT(*) AS days
FROM
    weather_observations w
JOIN
    locations l ON l.location_id = w.location_id
WHERE
    w.observed_on >= DATE_SUB('2025-08-05', INTERVAL 30 DAY)
GROUP BY
    l.city, l.state
HAVING days >= 30
ORDER BY stdev_high DESC
LIMIT 15;
```

```
MySQL [db]> SELECT
-> l.city,
-> l.state,
-> ROUND(AVG(w.tmax_c), 1) AS avg_high,
-> ROUND(STDDEV_SAMP(w.tmax_c), 2) AS stdev_high,
-> COUNT(*) AS days
-> FROM
-> weather_observations w
-> JOIN
-> locations l ON l.location_id = w.location_id
-> WHERE
-> w.observed_on >= DATE_SUB('2025-08-05', INTERVAL 30 DAY)
-> GROUP BY
-> l.city, l.state
-> HAVING days >= 30
-> ORDER BY stdev_high DESC
-> LIMIT 15;
```

city	state	avg_high	stdev_high	days
SEATTLE	WA	25.4	3.44	34
NEW YORK	NY	29.8	3.17	34
CHICAGO	IL	29.4	3.09	34
PHOENIX	AZ	42.4	3.04	34
PHILADELPHIA	PA	30.9	2.78	34
CHAMPAIGN	IL	29.6	2.29	34
DALLAS	TX	33.5	1.76	34
SAN FRANCISCO	CA	20.7	1.75	34
LOS ANGELES	CA	24.6	1.54	34
SAN DIEGO	CA	22.7	0.91	34

10 rows in set (0.026 sec)

This query will return the cities with the most unpredictable weather, meaning that their temperature swing a lot over the past 60 days. We look at the standard deviation of max temp from each city. Note that this query only has 10 rows because we only have 10 cities with weather_observation data.

Part 2: Indexing

QUERY 1

Query recap: find crops for ZIP=94539 whose USDA zone range includes that ZIP and whose sun is partial shade or full sun.

Baseline:

```
EXPLAIN ANALYZE
SELECT c.common_name, c.scientific_name
FROM crops AS c
JOIN zip_zones AS z
  ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max
WHERE z.zip = '94539'
  AND c.sun_req IN ('Partial sun/shade', 'Full sun');
```

Screenshot 1: baseline plan (no extra indexes):

```
MySQL [db]> EXPLAIN ANALYZE
-> SELECT c.common_name, c.scientific_name
-> FROM crops AS c
-> JOIN zip_zones AS z
->   ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max
-> WHERE z.zip = '94539'
->   AND c.sun_req IN ('Partial sun/shade', 'Full sun');
+-----+
| EXPLAIN |
+-----+
| -> Filter: ((c.sun_req in ('Partial sun/shade','Full sun')) and ('9b' between c.usda_zone_min and c.usda_zone_max)) (cost=84.7 rows=22.2) (actual time=0.0544..0.819 rows=230 loops=1)
| -> Table scan on c (cost=84.7 rows=1000) (actual time=0.0409..0.373 rows=1000 loops=1)
|
+-----+
1 row in set, 479 warnings (0.004 sec)
```

Index attempt A (crops by sun):

```
CREATE INDEX idx_crops_sun ON crops (sun_req);
```

Run again:

```
MySQL [db]> CREATE INDEX idx_crops_sun ON crops (sun_req);
Query OK, 0 rows affected (0.090 sec)
Records: 0 Duplicates: 0 Warnings: 0

MySQL [db]> EXPLAIN ANALYZE
-> SELECT c.common_name, c.scientific_name
-> FROM crops AS c
-> JOIN zip_zones AS z
->   ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max
-> WHERE z.zip = '94539'
->   AND c.sun_req IN ('Partial sun/shade', 'Full sun');
+-----+
| EXPLAIN |
+-----+
| -> Filter: ((c.sun_req in ('Partial sun/shade','Full sun')) and ('9b' between c.usda_zone_min and c.usda_zone_max)) (cost=59.9 rows=53.2) (actual time=0.0607..0.887 rows=230 loops=1)
| -> Table scan on c (cost=59.9 rows=1000) (actual time=0.0478..0.437 rows=1000 loops=1)
|
+-----+
1 row in set, 479 warnings (0.003 sec)
```

Analysis:

As shown in the screenshots above, the baseline query without any indexes cost approximately 84.7 as the query performed a full table scan of all 1000 rows of the crops table, meaning that the database had to analyze every row of the table before filtering based on the provided conditions. After adding an index on the sun_req column (`CREATE INDEX idx_crops_sun ON crops (sun_req);`), the cost was reduced from 84.7 to 59.9, suggesting that the new index helped the database filter out approximately 53.2 rows by the sunlight requirements of partial sun/shade and full sun. However, in the next line, it shows that the full crops table still had to be scanned, also suggesting that the index on sun_req was not fully utilized based on the second constraint that values had to be between usda_zone_min and usda_zone_max.

Index attempt B (crops by zone range):

```
CREATE INDEX idx_crops_zone ON crops (usda_zone_min, usda_zone_max);
```

Run again.

```
+-----+
|
+-----+
| EXPLAIN
|
+-----+
|
+-----+
| -> Filter: ((c.sun_req in ('Partial sun/shade','Full sun')) and ('9b' between c.usda_zone_min and c.usda_zone_max)) (cost=90.2 rows=15.4) (actual time=0.0467..0.808 rows=230 loops=1)
|       -> Table scan on c (cost=90.2 rows=1000) (actual time=0.0415..0.398 rows=1000 loops=1)
|
+-----+
|
+-----+
|
+-----+
1 row in set, 481 warnings (0.003 sec)
```

Analysis:

As shown in the screenshot above, the cost increased from the baseline cost of 84.7 to 90.2 after indexing the crops table on the min and max usda zones. The output of the explain analyze, shows that the query was able to filter out approximately 15.4 rows. However, because this index focused on a range of constraints – usda_zone_min and usda_zone_max – it is likely that the cost was not optimized because the range-based indexing still required the table to analyze two separate columns in the table to determine whether a value was appropriate. Overall, while it seemed like a good idea to index based on the usda zones, the query still needed to scan through the majority of the table and evaluate the range of values, ultimately increasing the cost of the query.

Index attempt C (zip_zones by usda_zone):

CREATE INDEX idx_usda_zone ON zip_zones (usda_zone);

```
MySQL [db]> CREATE INDEX idx_usda_zone ON zip_zones (usda_zone);
Query OK, 0 rows affected (0.262 sec)
Records: 0 Duplicates: 0 Warnings: 0

MySQL [db]> EXPLAIN ANALYZE
-> SELECT c.common_name, c.scientific_name
-> FROM crops AS c
-> JOIN zip_zones AS z
->   ON z.usda_zone BETWEEN c.usda_zone_min AND c.usda_zone_max
-> WHERE z.zip = '94539'
->   AND c.sun_req IN ('Partial sun/shade', 'Full sun');
+-----+
| EXPLAIN
|
+-----+
| -> Filter: ((c.sun_req in ('Partial sun/shade','Full sun')) and ('9b' between c.usda_zone_min and c.usda_zone_max)) (cost=84.7 rows=22.2) (actual time=0.0605..0.895 rows=230 loops=1)
|   -> Table scan on c (cost=84.7 rows=1000) (actual time=0.0491..0.382 rows=1000 loops=1)
|
+-----+
1 row in set, 479 warnings (0.003 sec)
```

Analysis:

Looking at the output of the explain analyze, this third index, relatively, did not have an effect on the baseline query. As shown in the screenshot, the cost of running the query was still approximately 84.7 and the crops table was still scanned in full, suggesting that the index on the zip_zones did not have an impact on optimizing the query. Similar to the prior two indexes, it is likely that the chosen index did not have an impact on the performance because the join condition uses a range comparison of the usda_zone to the c.usda_zone_min and c.usda_zone_max, causing the query to still comb through the table and compare all the values of interest. As a result, the cost of the query before and after the indexing remained the same.

Final indexing decision:

Comparing all three indexes, the final index design selected was indexing attempt A:

CREATE INDEX idx_crops_sun ON crops (sun_req);

This first index was able to filter out more of the rows and reduce the cost by a larger margin (baseline: 84.7 vs. A: 59.9, B: 90.2, C: 84.7) compared to the other indexes.

QUERY 2:

Query recap: union of 3 weather conditions (hot, freeze, heavy rain) joined to locations.

Baseline:

EXPLAIN ANALYZE

```
SELECT l.city, l.state, w.observed_on, 'heat_wave' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.tmax_c >= 35
```

UNION

```
SELECT l.city, l.state, w.observed_on, 'hard_freeze' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.tmin_c <= 0
```

UNION

```
SELECT l.city, l.state, w.observed_on, 'very_heavy_rain' AS event
FROM weather_observations AS w
JOIN locations AS l ON l.location_id = w.location_id
WHERE w.precip_mm >= 50
ORDER BY state, city, observed_on, event;
```

Screenshot 2: baseline

```
MySQL [db]> EXPLAIN ANALYZE
-> SELECT l.city, l.state, w.observed_on, 'heat_wave' AS event
-> FROM weather_observations AS w
-> JOIN locations AS l ON l.location_id = w.location_id
-> WHERE w.tmax_c >= 35
->
-> UNION
->
-> SELECT l.city, l.state, w.observed_on, 'hard_freeze' AS event
-> FROM weather_observations AS w
-> JOIN locations AS l ON l.location_id = w.location_id
-> WHERE w.tmin_c <= 0
->
-> UNION
->
-> SELECT l.city, l.state, w.observed_on, 'very_heavy_rain' AS event
-> FROM weather_observations AS w
-> JOIN locations AS l ON l.location_id = w.location_id
-> WHERE w.precip_mm >= 50
-> ORDER BY state, city, observed_on, event;
```

```
1 row in set (0.005 sec)
```

Index attempt A (location join):

```
CREATE INDEX idx_weather_loc ON weather_observations (location_id);
```

```
1 row in set (0.004 sec)
```

Analysis:

As shown in the screenshots above, the baseline query without any indexes had a cost of about 2146. After adding `CREATE INDEX idx_weather_loc ON weather_observations (location_id);`, the explain analyze results showed that the overall cost remained nearly the same, indicating that the index might have improved join efficiency but did not significantly reduce total runtime. This outcome suggests that the performance cost is primarily due to another condition instead of the join itself. As a result, indexing `location_id` alone provided minimal benefit since the query still needed to scan most of the weather data before applying those filters.

Index attempt B (hot filter):

```
CREATE INDEX idx_weather_tmax ON weather_observations (tmax_c);
```

```
| EXPLAIN
|
+-----+
| -> Sort: state, city, observed_on, `event` (cost=1674..1674 rows=785) (actual time=1.12..1.13 rows=120 loops=1)
|   -> Table scan on <union temporary> (cost=829..842 rows=785) (actual time=1.05..1.07 rows=120 loops=1)
|     -> Union materialize with deduplication (cost=829..829 rows=785) (actual time=1.05..1.05 rows=120 loops=1)
|       -> Nested loop inner join (cost=128 rows=118) (actual time=0.185..0.26 rows=118 loops=1)
|         -> Index range scan on w using idx_weather_tmax over (35 <= tmax_c), with index condition: (w.tmax_c >= 35) (cost=53.4 rows=118) (actual
time=0.174..0.188 rows=118 loops=1)
|           -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.531 rows=1) (actual time=412e-6..442e-6 rows=1 loops=118)
|             -> Nested loop inner join (cost=312 rows=333) (actual time=0.346..0.346 rows=0 loops=1)
|               -> Filter: (w.tmin_c <= 0) (cost=102 rows=333) (actual time=0.345..0.345 rows=0 loops=1)
|                 -> Table scan on w (cost=102 rows=1000) (actual time=0.0481..0.277 rows=1000 loops=1)
|                   -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.53 rows=1) (never executed)
|                     -> Nested loop inner join (cost=312 rows=333) (actual time=0.0921..0.369 rows=2 loops=1)
|                       -> Filter: (w.precip_mm >= 50) (cost=102 rows=333) (actual time=0.0888..0.364 rows=2 loops=1)
|                         -> Table scan on w (cost=102 rows=1000) (actual time=0.0303..0.293 rows=1000 loops=1)
|                           -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.53 rows=1) (actual time=0.00236..0.00239 rows=1 loops=2)
```

Analysis:

As shown in the screenshots above, the baseline query without any indexes had a cost of about 2146. After adding `CREATE INDEX idx_weather_tmax ON weather_observations (tmax_c);`, the explain analyze results showed that the overall cost was reduced to 1674. This drop and the report indicates that MySQL used an index range scan instead of a table scan (“Index range scan on w using idx_weather_tmax...”), meaning that the database was able to

Index attempt C (cold/rain filters):

[illegible]

As shown in the screenshots above, the baseline query without any indexes had a cost of about 2146. After adding `CREATE INDEX idx_weather_precip ON weather_observations (precip_mm);`, the explain analyze results showed that the overall cost was reduced to 1398. This drop and the report indicates that MySQL used an index range scan instead of a table scan ("Index range scan on w using idx_weather_precip..."), meaning that the database was able to target a subset of rows rather than looking through all the rows. This ultimately reduced the cost for the query as a whole.

```
CREATE INDEX idx_weather_precip ON weather_observations (precip_mm);
CREATE INDEX idx_weather_tmax ON weather_observations (tmax_c);
CREATE INDEX idx_weather_tmin ON weather_observations (tmin_c);
```

From the previous analysis, we saw that indexing on the JOIN didn't improve any cost, but indexing on the WHERE sections in our query caused a decrease in cost. Therefore, we decided to add all three WHERE sections to our index. While we didn't try the third WHERE index, since the design is very similar to the other two we can conclude that adding that index

will also improve the cost. We believe that these three indexes would improve the overall cost and efficiency of this query.

Query 3:

Query recap: last 30 days, group per city/state, compute avg and stddev, keep those with >= 30 days.

Baseline:

```
EXPLAIN ANALYZE
SELECT u.user_id, u.email
FROM users AS u
LEFT JOIN plots AS p
  ON p.user_id = u.user_id
  AND p.is_active = 1
GROUP BY u.user_id, u.email
HAVING COUNT(p.plot_id) = 0
LIMIT 15;
```

Screenshot 3: baseline

```
+-----+
| EXPLAIN |
+-----+
| -> Limit: 15 row(s) (actual time=3.03..3.04 rows=15 loops=1)
|   -> Filter: ('count(p.plot_id)' = 0) (actual time=3.03..3.03 rows=15 loops=1)
|     -> Table scan on <temporary> (actual time=3.03..3.03 rows=15 loops=1)
|       -> Aggregate using temporary table (actual time=3.03..3.03 rows=1000 loops=1)
|         -> Nested loop left join (cost=8853 rows=50000) (actual time=0.0622..2.12 rows=1257 loops=1)
|           -> Covering index scan on u using email (cost=103 rows=1000) (actual time=0.042..0.243 rows=1000 loops=1)
|             -> Filter: (p.is_active = 1) (cost=3.75 rows=50) (actual time=0.00121..0.00171 rows=0.267 loops=1000)
|               -> Index lookup on p using user_id (user_id=u.user_id) (cost=3.75 rows=50) (actual time=0.00109..0.00156 rows=0.5 loops=1000)
|
+-----+
1 row in set (0.006 sec)

Query OK, 0 rows affected (0.098 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Index attempt A (active plot filter):

```
CREATE INDEX idx_plots_user_active ON plots(user_id, is_active);
```

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=2.33..2.34 rows=15 loops=1)
    -> Filter: ('count(p.plot_id)' = 0) (actual time=2.33..2.34 rows=15 loops=1)
        -> Table scan on <temporary> (actual time=2.33..2.33 rows=15 loops=1)
            -> Aggregate using temporary table (actual time=2.33..2.33 rows=1000 loops=1)
                -> Nested loop left join (cost=2860 rows=25000) (actual time=0.0386..1.51 rows=1257 loops=1)
                    -> Covering index scan on u using email (cost=103 rows=1000) (actual time=0.0295..0.23 rows=1000 loops=1)
                    -> Covering index lookup on p using idx_plots_user_active (user_id=u.user_id, is_active=1) (cost=0.26 rows=25) (actual time=0.00105..0.00112 rows=0.267 loops=1000)
                |
            +-----+
1 row in set (0.003 sec)

```

Index attempt B (indexes based on email):

CREATE INDEX idx_users_email ON users(email);

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=2.41..2.41 rows=15 loops=1)
    -> Filter: ('count(p.plot_id)' = 0) (actual time=2.41..2.41 rows=15 loops=1)
        -> Table scan on <temporary> (actual time=2.4..2.41 rows=15 loops=1)
            -> Aggregate using temporary table (actual time=2.4..2.4 rows=1000 loops=1)
                -> Nested loop left join (cost=2860 rows=25000) (actual time=0.0584..1.59 rows=1257 loops=1)
                    -> Covering index scan on u using email (cost=103 rows=1000) (actual time=0.0274..0.254 rows=1000 loops=1)
                    -> Covering index lookup on p using idx_plots_user_active (user_id=u.user_id, is_active=1) (cost=0.26 rows=25) (actual time=0.00111..0.00117 rows=0.267 loops=1000)
                |
            +-----+
1 row in set (0.004 sec)

```

Index attempt C (should decrease count cost):

CREATE INDEX idx_plots_active_plotid ON plots(is_active, plot_id);

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=2.44..2.44 rows=15 loops=1)
    -> Filter: ('count(p.plot_id)' = 0) (actual time=2.44..2.44 rows=15 loops=1)
        -> Table scan on <temporary> (actual time=2.44..2.44 rows=15 loops=1)
            -> Aggregate using temporary table (actual time=2.43..2.43 rows=1000 loops=1)
                -> Nested loop left join (cost=2860 rows=25000) (actual time=0.062..1.58 rows=1257 loops=1)
                    -> Covering index scan on u using email (cost=103 rows=1000) (actual time=0.045..0.247 rows=1000 loops=1)
                    -> Covering index lookup on p using idx_plots_user_active (user_id=u.user_id, is_active=1) (cost=0.26 rows=25) (actual time=0.00107..0.00116 rows=0.267 loops=1000)
                |
            +-----+
1 row in set (0.004 sec)

```

Analysis:

Based on the screenshots above, it can be seen that the cost for the baseline query without any indexes costs about 8853. Interestingly, each of the three indexes brought the cost down to 2860. The **CREATE INDEX idx_plots_user_active ON plots(user_id, is_active);** index ran the fastest at 2.33 ms, beating out the other two indexes by mere milliseconds (2.41 ms and 2.44 ms respectively). All three indexes seem viable since they all reduced the cost significantly.

Final indexing decision:

CREATE INDEX idx_plots_user_active ON plots(user_id, is_active); seems to be the best index since it significantly reduces the cost from 8853 down to 2860, just like the other two indexes, but is faster than the other two indexes (2.33 ms vs 2.41 ms and 2.44 ms). Thus, the final indexing decision is to choose **INDEX idx_plots_user_active**.

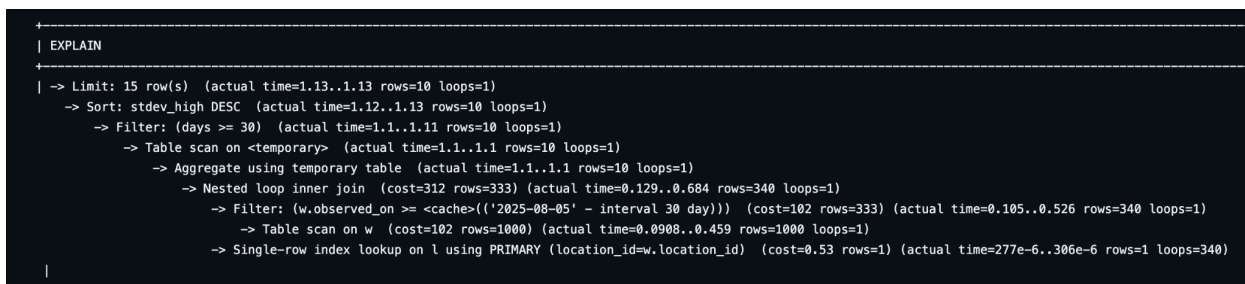
QUERY 4:

Query recap: match plots to locations to zip_zones to crops, then filter by soil/sun and zone, then group by crop.

Baseline:

```
EXPLAIN ANALYZE
SELECT
    l.city,
    l.state,
    ROUND(AVG(w.tmax_c), 1) AS avg_high,
    ROUND(STDDEV_SAMP(w.tmax_c), 2) AS stdev_high,
    COUNT(*) AS days
FROM
    weather_observations w
JOIN
    locations l ON l.location_id = w.location_id
WHERE
    w.observed_on >= DATE_SUB('2025-08-05', INTERVAL 30 DAY)
GROUP BY
    l.city, l.state
HAVING days >= 30
ORDER BY stdev_high DESC
LIMIT 15;
```

Screenshot 4: baseline:



```
| EXPLAIN
|-----|
| -> Limit: 15 row(s) (actual time=1.13..1.13 rows=10 loops=1)
|       -> Sort: stdev_high DESC (actual time=1.12..1.13 rows=10 loops=1)
|             -> Filter: (days >= 30) (actual time=1.1..1.11 rows=10 loops=1)
|                   -> Table scan on <temporary> (actual time=1.1..1.1 rows=10 loops=1)
|                         -> Aggregate using temporary table (actual time=1.1..1.1 rows=10 loops=1)
|                               -> Nested loop inner join (cost=312 rows=333) (actual time=0.129..0.684 rows=340 loops=1)
|                                     -> Filter: (w.observed_on >= <cache>(('2025-08-05' - interval 30 day))) (cost=102 rows=333) (actual time=0.105..0.526 rows=340 loops=1)
|                                           -> Table scan on w (cost=102 rows=1000) (actual time=0.0908..0.459 rows=1000 loops=1)
|                                                 -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.53 rows=1) (actual time=277e-6..3.06e-6 rows=1 loops=340)
|
```

Index attempt A (supports join and group by):

```
CREATE INDEX idx_weather_location_date ON weather_observations(location_id,
observed_on);
```

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=1.15..1.16 rows=10 loops=1)
   -> Sort: stdev_high DESC (actual time=1.15..1.15 rows=10 loops=1)
       -> Filter: (days >= 30) (actual time=1.13..1.13 rows=10 loops=1)
           -> Table scan on <temporary> (actual time=1.13..1.13 rows=10 loops=1)
               -> Aggregate using temporary table (actual time=1.13..1.13 rows=10 loops=1)
                   -> Nested loop inner join (cost=312 rows=333) (actual time=0.133..0.704 rows=340 loops=1)
                       -> Filter: (w.observed_on >= <cache>(('2025-08-05' - interval 30 day))) (cost=102 rows=333) (actual time=0.111..0.545 rows=340 loops=1)
                           -> Table scan on w (cost=102 rows=1000) (actual time=0.0965..0.477 rows=1000 loops=1)
                               -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.53 rows=1) (actual time=277e-6..307e-6 rows=1 loops=340)

```

Index attempt B (support group by city):

CREATE INDEX idx_locations_city_state ON locations(city, state);

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=0.769..0.77 rows=10 loops=1)
   -> Sort: stdev_high DESC (actual time=0.768..0.769 rows=10 loops=1)
       -> Filter: (days >= 30) (actual time=0.741..0.744 rows=10 loops=1)
           -> Table scan on <temporary> (actual time=0.74..0.742 rows=10 loops=1)
               -> Aggregate using temporary table (actual time=0.739..0.739 rows=10 loops=1)
                   -> Nested loop inner join (cost=218 rows=333) (actual time=0.0636..0.45 rows=340 loops=1)
                       -> Filter: (w.observed_on >= <cache>(('2025-08-05' - interval 30 day))) (cost=102 rows=333) (actual time=0.0526..0.318 rows=340 loops=1)
                           -> Table scan on w (cost=102 rows=1000) (actual time=0.042..0.26 rows=1000 loops=1)
                               -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.25 rows=1) (actual time=206e-6..234e-6 rows=1 loops=340)

```

Index attempt C (supports temperature reading):

CREATE INDEX idx_weather_tmax ON weather_observations(tmax_c);

```

+-----+
| EXPLAIN
+-----+
| -> Limit: 15 row(s) (actual time=0.839..0.841 rows=10 loops=1)
   -> Sort: stdev_high DESC (actual time=0.838..0.839 rows=10 loops=1)
       -> Filter: (days >= 30) (actual time=0.824..0.827 rows=10 loops=1)
           -> Table scan on <temporary> (actual time=0.822..0.824 rows=10 loops=1)
               -> Aggregate using temporary table (actual time=0.821..0.821 rows=10 loops=1)
                   -> Nested loop inner join (cost=218 rows=333) (actual time=0.0718..0.507 rows=340 loops=1)
                       -> Filter: (w.observed_on >= <cache>(('2025-08-05' - interval 30 day))) (cost=102 rows=333) (actual time=0.0595..0.374 rows=340 loops=1)
                           -> Table scan on w (cost=102 rows=1000) (actual time=0.0491..0.295 rows=1000 loops=1)
                               -> Single-row index lookup on l using PRIMARY (location_id=w.location_id) (cost=0.25 rows=1) (actual time=205e-6..234e-6 rows=1 loops=340)

```

What happened:

Before adding indexes, MySQL performed a full scan of weather_observations and grouped results by city and state, giving a high baseline cost. Adding idx_weather_location_date (location_id, observed_on) let MySQL filter recent rows faster and reduced the total cost the most. The idx_locations_city_state index had little effect since grouping and ordering happened after aggregation. The idx_weather_tmax index did not help because the query aggregates all temperature values rather than filtering by them. Overall, the composite date+location index was the most effective and was kept in the final design.

