CS 411: Project 1 Stage 3

For this project we are using Google Cloud Platform (GCP) to host our database. Our database consists of 5 tables provided in Image 1.

Image 1: Tables that make up the Database

We were successfully able to connect & host the database tables on GCP as showcased in Image 2 below.

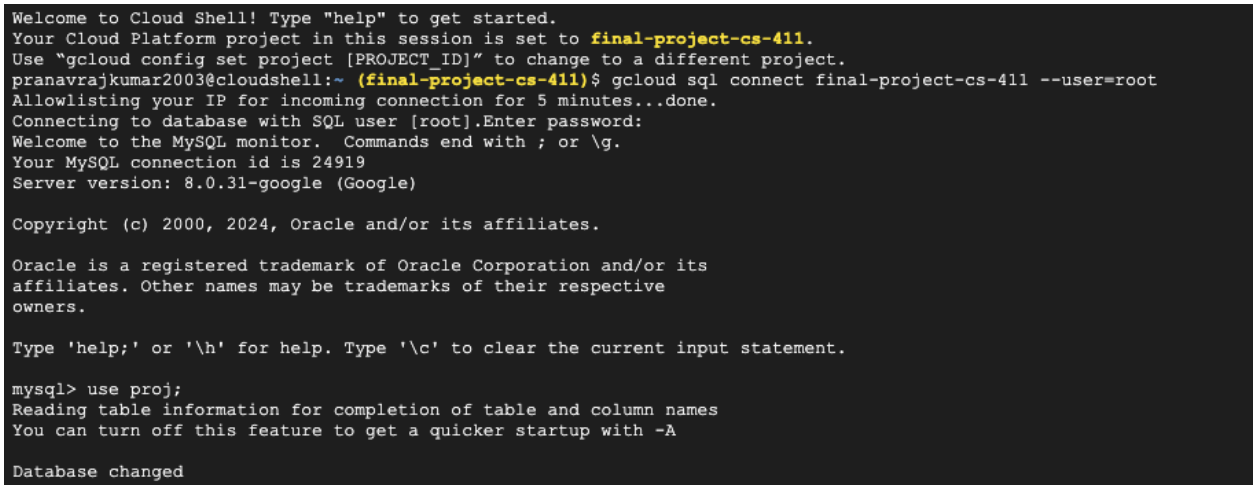


Image 2: A successful connection to the database on GCP

The DDL commands we used for our tables are shown below in Image 3.

```
-- This file contains the English Phrases, their translations, and their
pronunciations to the language of the user's choice.
CREATE TABLE combined_phrases (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   english_phrase TEXT NOT NULL,
   translation TEXT NOT NULL,
   pronunciation TEXT NOT NULL,
   language TEXT NOT NULL
);

--The following table is used to store the data for the dishes for each city in the
countries.
CREATE TABLE dishes (
   City VARCHAR(255),
   Country VARCHAR(255) NOT NULL,
   Dish VARCHAR(255) NOT NULL,
   DishID VARCHAR(255) PRIMARY KEY
);

-- The following table is used to store the data for the monuments for each city in
the countries.
CREATE TABLE monuments (
   City VARCHAR(255),
   Country VARCHAR(255) NOT NULL,
   Monument VARCHAR(255) NOT NULL,
   MonumentID VARCHAR(255) PRIMARY KEY
);

-- The following table is used to store the information about the users.
CREATE TABLE Users (
   username VARCHAR(50) NOT NULL PRIMARY KEY,
   points INT NOT NULL,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50),
   email VARCHAR(100) NOT NULL,
   Language VARCHAR(50) NOT NULL
);

-- The following table is used to store the information about the friends of the
users.
CREATE TABLE friends (
```

```
   friendname1 VARCHAR(50) NOT NULL,
   friendname2 VARCHAR(50) NOT NULL,
   PRIMARY KEY (friendname1, friendname2),
   FOREIGN KEY (friendname1) REFERENCES users(username),
   FOREIGN KEY (friendname2) REFERENCES users(username)
);

-- The following table is used to store the information about the countries and their
languages.
CREATE TABLE country_languages (
   Country VARCHAR(50) PRIMARY KEY,
   Language VARCHAR(50) not null
);
```

Image 3: DDL Commands to generate tables

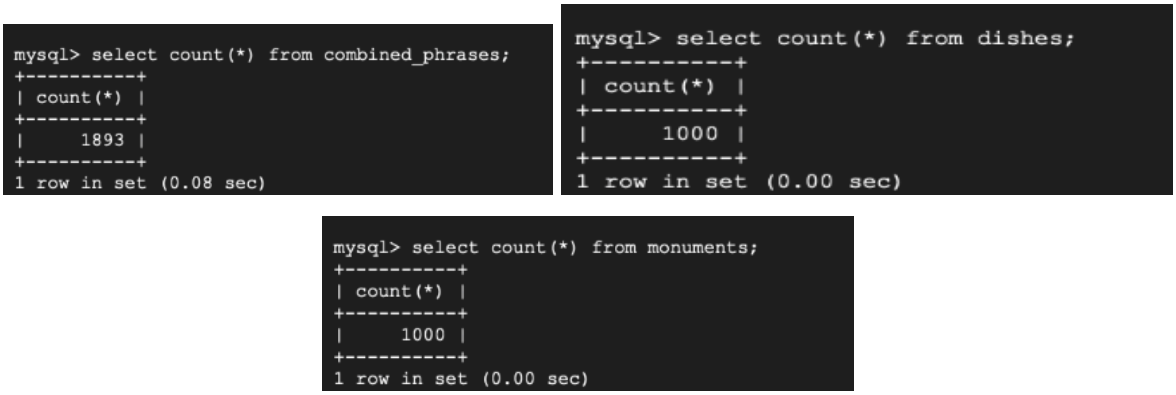


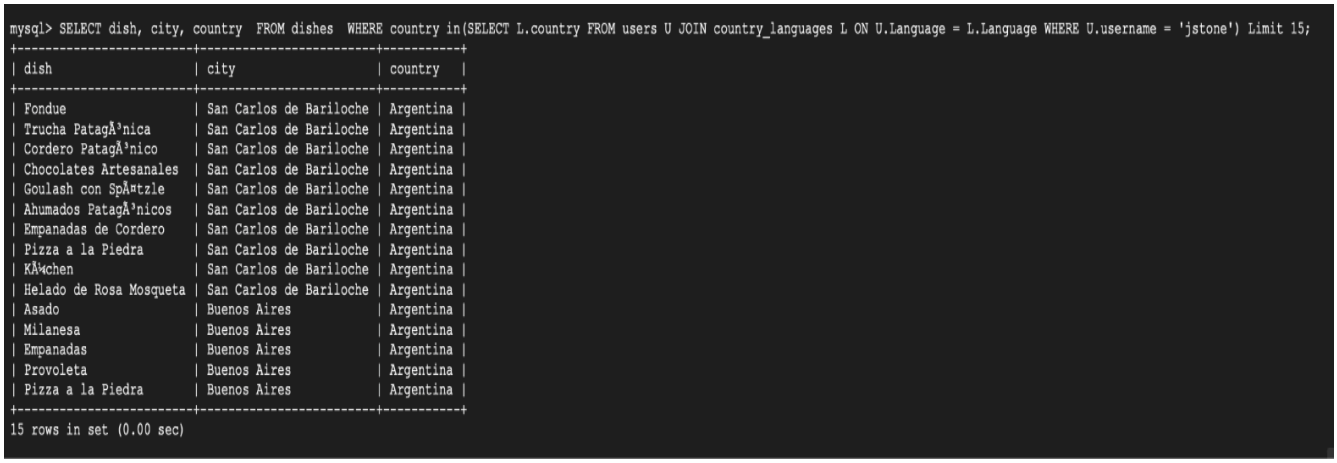
Image 4: 3 tables of length 1000

The Advanced Queries that were used along with their outputs are provided below in Images 5 - 11.

Query 1:

```
SELECT dish, city, country
FROM dishes
WHERE country in(SELECT L.country
                 FROM users U JOIN country_languages L ON U.Language = L.Language
                 WHERE U.username = 'jstone');
```

Image 5: Advanced Query used to get the Dishes, City & Country based on a language the user is trying to learn



```
mysql> SELECT dish, city, country  FROM dishes  WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'jstone') Limit 15;
+------------------------+------------------------+-----------+
| dish                   | city                   | country   |
+------------------------+------------------------+-----------+
| Fondue                 | San Carlos de Bariloche | Argentina |
| Trucha PatagÃ³nica      | San Carlos de Bariloche | Argentina |
| Cordero PatagÃ³nico     | San Carlos de Bariloche | Argentina |
| Chocolates Artesanales | San Carlos de Bariloche | Argentina |
| Goulash con SpÃ¤tzle    | San Carlos de Bariloche | Argentina |
| Ahumados PatagÃ³nicos   | San Carlos de Bariloche | Argentina |
| Empanadas de Cordero   | San Carlos de Bariloche | Argentina |
| Pizza a la Piedra      | San Carlos de Bariloche | Argentina |
| KÃ¼chen                 | San Carlos de Bariloche | Argentina |
| Helado de Rosa Mosqueta | San Carlos de Bariloche | Argentina |
| Asado                  | Buenos Aires           | Argentina |
| Milanesa               | Buenos Aires           | Argentina |
| Empanadas              | Buenos Aires           | Argentina |
| Provoleta              | Buenos Aires           | Argentina |
| Pizza a la Piedra      | Buenos Aires           | Argentina |
+------------------------+------------------------+-----------+
15 rows in set (0.00 sec)
```

Image 6: Output of Advanced Query 1

Query 2:

```
SELECT Monument, city, country
FROM monuments
WHERE country in(SELECT L.country
                 FROM users U JOIN country_languages L ON U.Language = L.Language WHERE
U.username = 'tlong');
```

Image 7: Advanced Query used to get the Monument, City & Country based on a language the user is trying to learn

```
mysql> SELECT Monument, city, country
    -> FROM monuments
    -> WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'tlong') limit 15;
+------------------------------+-------+-------------+
| Monument                     | city  | country     |
+------------------------------+-------+-------------+
| Zytglogge (Clock Tower)      | Bern  | Switzerland |
| Federal Palace of Switzerland | Bern  | Switzerland |
| Bear Pit                     | Bern  | Switzerland |
| Bern Cathedral (MÃ¼nster)     | Bern  | Switzerland |
| Rosengarten (Rose Garden)    | Bern  | Switzerland |
| Bundesplatz (Federal Square) | Bern  | Switzerland |
| Einstein House               | Bern  | Switzerland |
| Historical Museum of Bern    | Bern  | Switzerland |
| Nydegg Bridge                | Bern  | Switzerland |
| Kornhaus Bridge              | Bern  | Switzerland |
| Basel Minster                | Basel | Switzerland |
| Tinguely Fountain            | Basel | Switzerland |
| Rathaus (Town Hall)          | Basel | Switzerland |
| Spalentor (Spalen Gate)      | Basel | Switzerland |
| Kunstmuseum Basel            | Basel | Switzerland |
+------------------------------+-------+-------------+
15 rows in set (0.01 sec)
```

Image 8: Output for Advanced Query 2

Query 3:

```
SELECT potential_friend.friendname2 AS suggested_friend,
COUNT(mutual_friends.friendname2) AS mutual_friend_count
FROM friends AS direct_friends JOIN friends AS mutual_friends ON
direct_friends.friendname2 = mutual_friends.friendname1
JOIN friends AS potential_friend ON mutual_friends.friendname2 =
potential_friend.friendname2
WHERE direct_friends.friendname1 = 'aJoshua' AND potential_friend.friendname2 !=
'aJoshua' AND potential_friend.friendname2
NOT IN ( SELECT friendname2 FROM friends WHERE friendname1 = 'aJoshua' )
GROUP BY potential_friend.friendname2 HAVING COUNT(mutual_friends.friendname2) >= 1
ORDER BY mutual_friend_count DESC;
```

Image 9: Get the users who are friends with the friends of the current user and are trying to learn the same language

```
mysql> SELECT potential_friend.friendname2 AS suggested_friend, COUNT(mutual_friends.friendname2) AS mutual_friend_count
    -> FROM friends AS direct_friends JOIN friends AS mutual_friends ON direct_friends.friendname2 = mutual_friends.friendname1
    -> JOIN friends AS potential_friend ON mutual_friends.friendname2 = potential_friend.friendname2
    -> WHERE direct_friends.friendname1 = 'aJoshua' AND potential_friend.friendname2 != 'aJoshua' AND potential_friend.friendname2
    -> NOT IN ( SELECT friendname2 FROM friends WHERE friendname1 = 'aJoshua' )
    -> GROUP BY potential_friend.friendname2 HAVING COUNT(mutual_friends.friendname2) >= 1
    -> ORDER BY mutual_friend_count DESC;
+-----------------+---------------------+
| suggested_friend | mutual_friend_count |
+-----------------+---------------------+
| pJon            |                   1 |
| tlong           |                   1 |
+-----------------+---------------------+
2 rows in set (0.05 sec)
```

Image 10: Output for Advanced Query 3

Query 4:

```
SELECT COUNT(DISTINCT m.MonumentID) AS num_monuments, COUNT(DISTINCTd.DishID) AS
num_dishes, m.City AS city
FROM monuments AS m JOIN dishes AS d ON m.City = d.City
GROUP BY m.City;
```

Image 11: Advanced Query to get the count of all the monuments and dishes that have common cities as a fun fact to display to the users.

```
mysql> SELECT COUNT(DISTINCT m.MonumentID) AS num_monuments,
    ->        COUNT(DISTINCT d.DishID) AS num_dishes,
    ->        m.City AS city
    -> FROM monuments AS m
    -> JOIN dishes AS d ON m.City = d.City
    -> GROUP BY m.City limit 15;
+---------------+------------+----------------+
| num_monuments | num_dishes | city           |
+---------------+------------+----------------+
|            10 |         10 | Ãvora          |
|            10 |         10 | Antwerp        |
|            10 |         10 | Aveiro         |
|            10 |         10 | Barcelona      |
|            10 |         10 | Basel          |
|            10 |         10 | Belo Horizonte |
|            10 |         10 | Berlin         |
|            10 |         10 | Bern           |
|            10 |         10 | Bilbao         |
|            10 |         10 | Bordeaux       |
|            10 |         10 | Braga          |
|            10 |         10 | BrasÃlia      |
|            10 |         10 | Bremen         |
|            10 |         10 | Bruges         |
|            10 |         10 | Brussels       |
+---------------+------------+----------------+
15 rows in set (0.01 sec)
```

Image 12: Output to Advanced Query 4

Indexing Outputs:

Monuments:

Output with no indexing:

```
mysql> explain analyze SELECT dish, city, country
    -> FROM dishes
    -> WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'jstone');
+------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                    |
+------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
| -> Filter: (dishes.Country = L.Country)  (cost=114.10 rows=110) (actual time=0.273..0.864 rows=300 loops=1)
    -> Inner hash join (<hash>(dishes.Country)=<hash>(L.Country))  (cost=114.10 rows=110) (actual time=0.272..0.804 rows=300 loops=1)
        -> Table scan on dishes  (cost=11.60 rows=1000) (actual time=0.087..0.422 rows=1000 loops=1)
        -> Hash
            -> Filter: (L.`Language` = 'Spanish')  (cost=1.35 rows=1) (actual time=0.122..0.129 rows=3 loops=1)
                -> Table scan on L  (cost=1.35 rows=11) (actual time=0.063..0.069 rows=11 loops=1)
 |
+------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------+
1 row in set (0.02 sec)
```

Index on country_languages.Language:

```
mysql> CREATE
    -> INDEX id_country_languages_language ON country_languages (Language) ;
Query OK, 0 rows affected (0.23 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT dish, city, country  FROM dishes  WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'jstone');
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------+
| EXPLAIN

                                         |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------+
| -> Filter: (dishes.Country = L.Country)  (cost=303.34 rows=300) (actual time=0.104..0.635 rows=300 loops=1)
    -> Inner hash join (<hash>(dishes.Country)=<hash>(L.Country))  (cost=303.34 rows=300) (actual time=0.102..0.572 rows=300 loops=1)
        -> Table scan on dishes  (cost=4.25 rows=1000) (actual time=0.031..0.368 rows=1000 loops=1)
        -> Hash
            -> Covering index lookup on L using id_country_languages_language (Language='Spanish')  (cost=0.57 rows=3) (actual time=0.019..0.022 rows=3 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------+
1 row in set (0.01 sec)
```

We wanted to add an index on Language in country_languages to help the join process but as you can see this increased the cost even though it reduced the time for the filtering. This was not the best move and we removed this indexing.

Index on dishes.Country:

```
mysql> CREATE INDEX idx_dishes_country ON dishes(Country);
Query OK, 0 rows affected (0.16 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT dish, city, country  FROM dishes  WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'jstone');
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
----+
| EXPLAIN

    |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
----+
| -> Nested loop inner join  (cost=21.43 rows=110) (actual time=0.887..12.555 rows=300 loops=1)
    -> Filter: (L.`Language` = 'Spanish')  (cost=1.35 rows=1) (actual time=0.015..0.031 rows=3 loops=1)
        -> Table scan on L  (cost=1.35 rows=11) (actual time=0.013..0.023 rows=11 loops=1)
    -> Index lookup on dishes using idx_dishes_country (Country=L.Country), with index condition: (dishes.Country = L.Country)  (cost=17.34 rows=100) (actual time=4.012..4.166 rows=100 loops=3)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------------
----+
1 row in set (0.02 sec)
```

Instead we decided to add indexing on Country in dishes to help the WHERE clause in the main query filter rows from dishes by Country. As you can see this has helped in reducing the cost and we decided to keep this indexing.

Monuments:

Output with no indexing:

```
mysql> explain analyze SELECT Monument, city, country
    -> FROM monuments
    -> WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'tlong')
    -> ;
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------+
| -> Filter: (monuments.Country = L.Country)  (cost=114.10 rows=110) (actual time=0.311..0.682 rows=200 loops=1)
    -> Inner hash join (<hash>(monuments.Country)=<hash>(L.Country))  (cost=114.10 rows=110) (actual time=0.311..0.636 rows=200 loops=1)
        -> Table scan on monuments  (cost=11.60 rows=1000) (actual time=0.041..0.403 rows=1000 loops=1)
        -> Hash
            -> Filter: (L.`Language` = 'German')  (cost=1.35 rows=1) (actual time=0.055..0.058 rows=2 loops=1)
                -> Table scan on L  (cost=1.35 rows=11) (actual time=0.027..0.030 rows=11 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Index on country_languages.Language:

```
mysql> CREATE INDEX id_country_languages_language ON country_languages (Language);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT Monument, city, country  FROM monuments  WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'tlong');
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------+
| EXPLAIN
                                                |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------+
| -> Filter: (monuments.Country = L.Country)  (cost=203.22 rows=200) (actual time=0.369..0.732 rows=200 loops=1)
    -> Inner hash join (<hash>(monuments.Country)=<hash>(L.Country))  (cost=203.22 rows=200) (actual time=0.367..0.675 rows=200 loops=1)
        -> Table scan on monuments  (cost=6.38 rows=1000) (actual time=0.035..0.400 rows=1000 loops=1)
        -> Hash
            -> Covering index lookup on L using id_country_languages_language (Language='German')  (cost=0.46 rows=2) (actual time=0.013..0.017 rows=2 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------+
1 row in set (0.00 sec)
```

We wanted to add an index on Language in country_languages to help the join process but as you can see this increased the cost even though it reduced the time for the filtering. This was not the best move and we removed this indexing.

Index on monuments.Country:

```
mysql> CREATE INDEX idx_monuments_country ON monuments(Country);
Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT Monument, city, country  FROM monuments  WHERE country in(SELECT L.country FROM users U JOIN country_languages L ON U.Language = L.Language WHERE U.username = 'tlong');
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------+
| EXPLAIN
            |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------+
| -> Nested loop inner join  (cost=21.43 rows=110) (actual time=0.164..10.083 rows=200 loops=1)
    -> Filter: (L.`Language` = 'German')  (cost=1.35 rows=1) (actual time=0.015..0.024 rows=2 loops=1)
        -> Table scan on L  (cost=1.35 rows=11) (actual time=0.011..0.020 rows=11 loops=1)
    -> Index lookup on monuments using idx_monuments_country (Country=L.Country), with index condition: (monuments.Country = L.Country)  (cost=17.34 rows=100) (actual time=4.808..5.021 rows=100 loops=2)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------
------------+
1 row in set (0.02 sec)
```

Instead we decided to add indexing on Country in monuments to help the WHERE clause in the main query filter rows from dishes by Country. As you can see this has helped in reducing the cost and we decided to keep this indexing.

Friends:

Output with no indexing:

```
mysql> explain analyze SELECT potential_friend.friendname2 AS suggested_friend, COUNT(mutual_friends.friendname2) AS mutual_friend_count FROM friends AS direct_friends JOIN friends AS mutual_friends ON direct_friends.friendname2 = mutual_fri
ends.friendname1 JOIN friends AS potential_friend ON mutual_friends.friendname2 = potential_friend.friendname2 WHERE direct_friends.friendname1 = 'aJoshua' AND potential_friend.friendname2 != 'aJoshua' AND potential_friend.friendname2 NOT IN
 ( SELECT friendname2 FROM friends WHERE friendname1 = 'aJoshua' ) GROUP BY potential_friend.friendname2 HAVING COUNT(mutual_friends.friendname2) >= 1 ORDER BY mutual_friend_count DESC;
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                            |
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: mutual_friend_count DESC  (actual time=0.327..0.327 rows=2 loops=1)
    -> Filter: (count(mutual_friends.friendname2) >= 1)  (actual time=0.234..0.235 rows=2 loops=1)
        -> Table scan on <temporary>  (actual time=0.170..0.171 rows=2 loops=1)
            -> Aggregate using temporary table  (actual time=0.167..0.167 rows=2 loops=1)
                -> Nested loop antijoin  (cost=1.97 rows=2) (actual time=0.065..0.100 rows=2 loops=1)
                    -> Nested loop inner join  (cost=1.37 rows=2) (actual time=0.061..0.093 rows=2 loops=1)
                        -> Nested loop inner join  (cost=0.76 rows=2) (actual time=0.055..0.058 rows=2 loops=1)
                            -> Covering index lookup on direct_friends using PRIMARY (friendname1='aJoshua')  (cost=0.35 rows=1) (actual time=0.021..0.022 rows=1 loops=1)
                            -> Filter: (mutual_friends.friendname2 <> 'aJoshua')  (cost=0.41 rows=2) (actual time=0.009..0.011 rows=2 loops=1)
                                -> Covering index lookup on mutual_friends using PRIMARY (friendname1=direct_friends.friendname2)  (cost=0.41 rows=2) (actual time=0.007..0.009 rows=3 loops=1)
                        -> Covering index lookup on potential_friend using friendname2 (friendname2=mutual_friends.friendname2)  (cost=0.36 rows=2) (actual time=0.004..0.017 rows=1 loops=2)
                    -> Single-row covering index lookup on friends using PRIMARY (friendname1='aJoshua', friendname2=mutual_friends.friendname2)  (cost=0.21 rows=1) (actual time=0.003..0.003 rows=0 loops=2)
 |
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

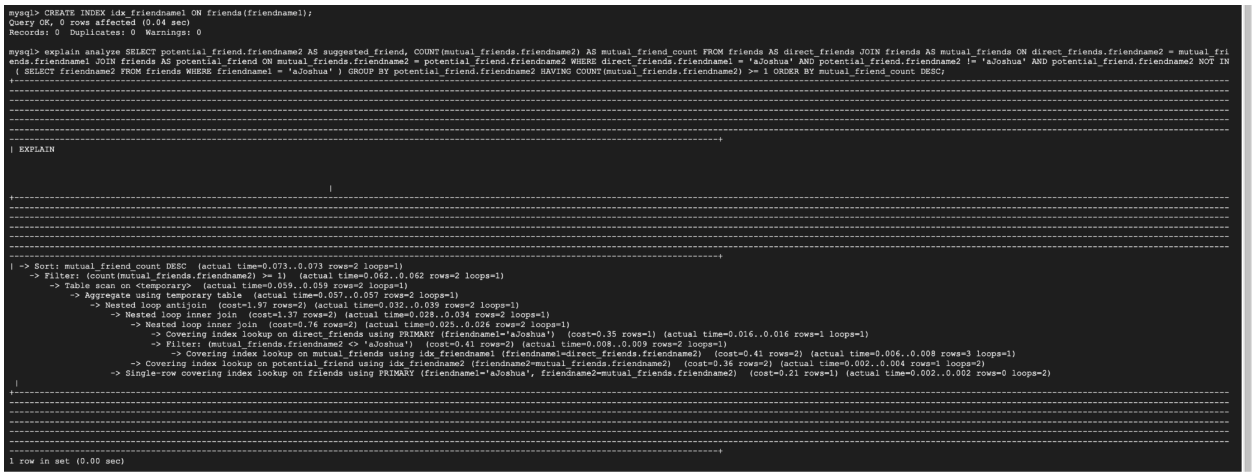
Output with Index on friends (friendname1, friendname2):

```
mysql> CREATE INDEX idx_friendname1_friendname2 ON friends(friendname1, friendname2);
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain analyze SELECT potential_friend.friendname2 AS suggested_friend, COUNT(mutual_friends.friendname2) AS mutual_friend_count FROM friends AS direct_friends JOIN friends AS mutual_friends ON direct_friends.friendname2 = mutual_fri
ends.friendname1 JOIN friends AS potential_friend ON mutual_friends.friendname2 = potential_friend.friendname2 WHERE direct_friends.friendname1 = 'aJoshua' AND potential_friend.friendname2 != 'aJoshua' AND potential_friend.friendname2 NOT IN
 ( SELECT friendname2 FROM friends WHERE friendname1 = 'aJoshua' ) GROUP BY potential_friend.friendname2 HAVING COUNT(mutual_friends.friendname2) >= 1 ORDER BY mutual_friend_count DESC;
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                            |
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
| -> Sort: mutual_friend_count DESC  (actual time=0.139..0.139 rows=2 loops=1)
    -> Filter: (count(mutual_friends.friendname2) >= 1)  (actual time=0.127..0.128 rows=2 loops=1)
        -> Table scan on <temporary>  (actual time=0.124..0.124 rows=2 loops=1)
            -> Aggregate using temporary table  (actual time=0.122..0.122 rows=2 loops=1)
                -> Nested loop antijoin  (cost=1.97 rows=2) (actual time=0.093..0.102 rows=2 loops=1)
                    -> Nested loop inner join  (cost=1.37 rows=2) (actual time=0.089..0.096 rows=2 loops=1)
                        -> Nested loop inner join  (cost=0.76 rows=2) (actual time=0.084..0.086 rows=2 loops=1)
                            -> Covering index lookup on direct_friends using PRIMARY (friendname1='aJoshua')  (cost=0.35 rows=1) (actual time=0.021..0.022 rows=1 loops=1)
                            -> Filter: (mutual_friends.friendname2 <> 'aJoshua')  (cost=0.41 rows=2) (actual time=0.061..0.063 rows=2 loops=1)
                                -> Covering index lookup on mutual_friends using PRIMARY (friendname1=direct_friends.friendname2)  (cost=0.41 rows=2) (actual time=0.059..0.060 rows=3 loops=1)
                        -> Covering index lookup on potential_friend using friendname2 (friendname2=mutual_friends.friendname2)  (cost=0.36 rows=2) (actual time=0.004..0.005 rows=1 loops=2)
                    -> Single-row covering index lookup on friends using PRIMARY (friendname1='aJoshua', friendname2=mutual_friends.friendname2)  (cost=0.21 rows=1) (actual time=0.002..0.002 rows=0 loops=2)
 |
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

We added this indexing as we thought this would help the joins perform faster by covering both friendname1 and friendname2 columns and as you can see it has reduced the time but not the cost.

Output with Index on friends (friendname2) and friends (friendname1):



We added an index on friends(friendname2) to speed up the anti-join subquery, especially for potential_friend retrievals and we can see this is happening from the images as the time is reducing. Similarly index on friends(friendname1) enables efficient scans on friendname1 for quick filtering of relevant users.

Count Information:

Output with no indexing:

```
mysql> EXPLAIN ANALYZE
    -> SELECT COUNT(DISTINCT m.MonumentID) AS num_monuments,
    ->        COUNT(DISTINCT d.DishID) AS num_dishes,
    ->        m.City AS city
    -> FROM monuments AS m
    -> JOIN dishes AS d ON m.City = d.City
    -> GROUP BY m.City;
+------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                           |
+------------------------------------------------------------------------------------------+
| -> Group aggregate: count(distinct monuments.MonumentID), count(distinct dishes.DishID)  (actual time=9.724..19.263 rows=99 loops=1)
    -> Sort: m.City  (actual time=9.619..10.238 rows=10200 loops=1)
        -> Stream results  (cost=100126.95 rows=100000) (actual time=0.648..5.510 rows=10200 loops=1)
            -> Filter: (d.City = m.City)  (cost=100126.95 rows=100000) (actual time=0.643..3.183 rows=10200 loops=1)
                -> Inner hash join (<hash>(d.City)=<hash>(m.City))  (cost=100126.95 rows=100000) (actual time=0.640..1.522 rows=10200 loops=1)
                    -> Table scan on d  (cost=0.03 rows=1000) (actual time=0.023..0.270 rows=1000 loops=1)
                    -> Hash
                        -> Table scan on m  (cost=102.75 rows=1000) (actual time=0.039..0.405 rows=1000 loops=1)
 |
+------------------------------------------------------------------------------------------+
1 row in set (0.03 sec)
```

Output with index on monument.City and dish.City:

```
mysql> CREATE INDEX idx_monuments_city ON monuments(City);
Query OK, 0 rows affected (0.20 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_dishes_city ON dishes(City);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Group aggregate: count(distinct m.MonumentID), count(distinct d.DishID)  (cost=3885.58 rows=10101) (actual time=2.155..15.944 rows=99 loops=1)
    -> Nested loop inner join  (cost=2875.48 rows=10101) (actual time=1.942..7.305 rows=10200 loops=1)
        -> Filter: (m.City is not null)  (cost=102.75 rows=1000) (actual time=1.826..2.084 rows=1000 loops=1)
            -> Covering index scan on m using idx_monuments_city  (cost=102.75 rows=1000) (actual time=0.825..1.016 rows=1000 loops=1)
        -> Covering index lookup on d using idx_dishes_city (City=m.City)  (cost=1.76 rows=10) (actual time=0.003..0.004 rows=10 loops=1000)
```

We added indexes on monument.City and dish.City as the query joins monuments and dishes on City and indexing City in both tables should optimize the join process. As we can see the cost has significantly reduced and we decided to keep this indexing.

Output with index on monument.MonumentsID and dish.DishID:

```
mysql> CREATE INDEX idx_monuments_monumentid ON monuments(MonumentID);
Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_dishes_dishid ON dishes(DishID);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> EXPLAIN ANALYZE  SELECT COUNT(DISTINCT m.MonumentID) AS num_monuments,        COUNT(DISTINCT d.DishID) AS num_dishes,        m.City AS city  FROM monuments AS m  JOIN dishes AS d ON m.City = d.City  GROUP BY m.City;
+--------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+
| EXPLAIN

                                                                                                                       |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+
| -> Group aggregate: count(distinct m.MonumentID), count(distinct d.DishID)  (cost=3885.58 rows=10101) (actual time=0.379..17.632 rows=99 loops=1)
    -> Nested loop inner join  (cost=2875.48 rows=10101) (actual time=0.080..6.082 rows=10200 loops=1)
        -> Filter: (m.City is not null)  (cost=102.75 rows=1000) (actual time=0.049..0.471 rows=1000 loops=1)
            -> Covering index scan on m using idx_monuments_city  (cost=102.75 rows=1000) (actual time=0.047..0.389 rows=1000 loops=1)
        -> Covering index lookup on d using idx_dishes_city (City=m.City)  (cost=1.76 rows=10) (actual time=0.003..0.005 rows=10 loops=1000)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------+
1 row in set (0.02 sec)
```

The count operations can be sped up by indexing MonumentID and DishID in the monuments and dishes table respectively and looking at the output we can see the time has reduced and the cost has not changed.

Updates to Database Design Based on Stage 2 Feedback

Based on the feedback, we combined the tables for all languages instead of having a separate table for each language. We also accounted for the self-loop for the friends entity but have kept the table as we have a many to many relationship. We added new entities to meet the 5 entity requirements by adding the new dishes, monuments & countries entities that provide information about the most popular languages, food dishes & monuments in countries. With this addition, we are not only proposing a language-learning application but an overall tourist experience where we will provide users with the language of the city, popular food items and the famous monuments of the city.