Database Design

DDL Commands

```sql
CREATE TABLE User (
    userID INT PRIMARY KEY,
    userName VARCHAR(255),
    email VARCHAR(255),
    universityID INT,
    FOREIGN KEY (universityID) REFERENCES University(universityID)
);

CREATE TABLE Family (
    familyID INT PRIMARY KEY,
    leaderID INT,
    accessType VARCHAR(255),
    serviceName VARCHAR(255),
    FOREIGN KEY (leaderID) REFERENCES User(userID),
    FOREIGN KEY (serviceName) REFERENCES SubscriptionService(serviceName)
);

CREATE TABLE SubscriptionService (
    serviceName VARCHAR(255) PRIMARY KEY,
    price DECIMAL(10,2),
    maxMembers INT
);

CREATE TABLE University (
    universityID INT PRIMARY KEY,
    universityName VARCHAR(255),
    city VARCHAR(255)
);

CREATE TABLE Membership (
    memberID INT,
    familyID INT,
    memberStatus VARCHAR(255),
    PRIMARY KEY (memberID, familyID),
    FOREIGN KEY (memberID) REFERENCES User(userID),
    FOREIGN KEY (familyID) REFERENCES Family(familyID)
);

CREATE TABLE BankAccount (
    accountName VARCHAR(255),
    platform VARCHAR(255),
    userID INT,
    PRIMARY KEY (accountName, platform),
    FOREIGN KEY (userID) REFERENCES User(userID)
);
```

```
CREATE TABLE Payment (
    paymentID INT PRIMARY KEY,
    payerID INT,
    recipientID INT,
    amount DECIMAL(10,2),
    paid BIT,
    deadline DATE,
    FOREIGN KEY (payerID) REFERENCES User(userID),
    FOREIGN KEY (recipientID) REFERENCES User(userID)
);
```

Advanced Queries

Query 1

Our first query returns the number of pending user invitations per university. We first join the `University` table with the `User` table, then group by the `universityID`. We order the outputby descending number of users.

Code

```
select un.universityName, count(*) as numUsers
from University un
natural join User us
group by un.universityID
order by numUsers desc;
```

Result

```python
    with pool.connect() as con:
        statement = '''
        select un.universityName, count(*) as numPending
        from University un
        natural join User us
        join Membership m
        on m.memberID = us.userID
        where m.memberStatus = "Pending"
        group by un.universityName
        order by numPending desc;
        '''
        result = con.execute(statement).fetchall()
        for row in result:
            print(row)
[123]   ✓  0.9s                                                                                    Python
...   ('Harper College', 73)
      ('Northwestern University', 61)
      ('Yale University', 61)
      ('University of Illinois at Urbana-Champaign', 60)
      ('Harvard University', 57)
      ('Massachusetts Institute of Technology', 57)
```

Initial Performance

Below is the performance of our query without any added indexes. Between each index, we made sure to drop the added index from the previous part, to ensure standardization between comparisons.

```
| -> Sort: numPending DESC  (actual time=3.004..3.004 rows=6 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.001 rows=6 loops=1)
        -> Aggregate using temporary table  (actual time=2.958..2.959 rows=6 loops=1)
            -> Nested loop inner join  (cost=683.50 rows=400) (actual time=0.388..2.652 rows=369 loops=1)
                -> Nested loop inner join  (cost=543.50 rows=400) (actual time=0.380..2.345 rows=369 loops=1)
                    -> Filter: (m.memberStatus = 'Pending')  (cost=403.50 rows=400) (actual time=0.365..1.670 rows=369 loops=1)
                        -> Table scan on m  (cost=403.50 rows=4000) (actual time=0.358..1.329 rows=4000 loops=1)
                    -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=369)
                        -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
                -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
|
+-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
---+
1 row in set (0.01 sec)
```

Index 1

For our first index design, we added an index on `Membership.memberStatus` because we filter on that column in the query.

```sql
alter table Membership add index memId(memberStatus);
```

The output of our analyze query follows:

```
--------------------------------------------------------------------------------------------------------------------------+
--------------------------------------------------------------------------------------------------------------------------+
| -> Sort: numPending DESC  (actual time=1.440..1.440 rows=6 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=6 loops=1)
        -> Aggregate using temporary table  (actual time=1.424..1.425 rows=6 loops=1)
            -> Nested loop inner join  (cost=325.35 rows=369) (actual time=0.097..1.100 rows=369 loops=1)
                -> Nested loop inner join  (cost=196.20 rows=369) (actual time=0.090..0.799 rows=369 loops=1)
                    -> Index lookup on m using memId (memberStatus='Pending')  (cost=67.05 rows=369) (actual time=0.046..0.157 rows=369 loops=1)
                    -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=369)
                        -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
                -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
 |
+-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

We see a speedup in the "Filter" command that addresses `m.memberStatus = "Pending"`, and an overall speedup in the whole query, so we can conclude that this index is a good optimization to add for this specific query.

Index 2

For our second index design, we added an index on `University.universityName` because we use it in the grouping step of our query.

```
alter table University add index uniName(universityName);
```

The output of our analyze query follows:

```
| -> Sort: numPending DESC  (actual time=2.886..2.886 rows=6 loops=1)
    -> Table scan on <temporary>  (actual time=0.001..0.002 rows=6 loops=1)
        -> Aggregate using temporary table  (actual time=2.867..2.868 rows=6 loops=1)
            -> Nested loop inner join  (cost=683.50 rows=400) (actual time=0.061..2.541 rows=369 loops=1)
                -> Nested loop inner join  (cost=543.50 rows=400) (actual time=0.055..2.211 rows=369 loops=1)
                    -> Filter: (m.memberStatus = 'Pending')  (cost=403.50 rows=369) (actual time=0.047..1.515 rows=369 loops=1)
                        -> Table scan on m  (cost=403.50 rows=4000) (actual time=0.041..1.113 rows=4000 loops=1)
                    -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=369)
                        -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
                -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=369)
 |
+-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
---+
1 row in set (0.01 sec)
```

We see some marginal performance improvement on the aggregation step of the query, which means that we could still use this index design as well.

Index 3

For our third index design, we added an index on `User.userID`, which is used to join the two tables.

```
alter table User add index useId(userID);
```

The output of our analyze query follows:

```
| EXPLAIN
+-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
| -> Sort: numUsers DESC   (actual time=1.899..1.899 rows=6 loops=1)
    -> Stream results  (cost=962.26 rows=4793) (actual time=0.435..1.885 rows=6 loops=1)
        -> Group aggregate: count(0)  (cost=962.26 rows=4793) (actual time=0.430..1.878 rows=6 loops=1)
            -> Nested loop inner join  (cost=482.96 rows=4793) (actual time=0.105..1.589 rows=4999 loops=1)
                -> Index scan on un using PRIMARY  (cost=0.85 rows=6) (actual time=0.067..0.069 rows=6 loops=1)
                -> Index lookup on us using universityID (universityID=un.universityID)  (cost=13.78 rows=799) (actual time=0.017..0.199 rows=833 loops=6)
 |
+-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------
1 row in set (0.01 sec)
```

We again see some improvement in the actual time of both the single iteration and total iteration times of the nested loop inner join. We actually also see speedups in the sort and aggregation processes, so this is also a good index to add for our query.

Query 2

Our second query returns the number of accepted users for every university and subscription service. We begin by joining the User, Membership, Family, SubscriptionService, and University tables. We then filter by accepted members, and group by both universityID and serviceName. We order the output by the universityName in ascending order and aggregated numUsers in descending order. Finally, we limit the length of the output to 15 rows for visualization.

Code

```sql
select un.universityName, ss.serviceName, count(*) as numUsers
from User us
join Membership m
    on m.memberID = us.userID
natural join Family f
natural join SubscriptionService ss
natural join University un
where m.memberStatus = "Accepted"
group by un.universityID, ss.serviceName
order by un.universityName asc, numUsers desc
limit 15;
```

Result

```python
    with pool.connect() as con:
        statement = '''
        select un.universityName, ss.serviceName, count(*) as numUsers
        from User us
        join Membership m
            on m.memberID = us.userID
        natural join Family f
        natural join SubscriptionService ss
        natural join University un
        where m.memberStatus = "Accepted"
        group by un.universityID, ss.serviceName
        order by un.universityName asc, numUsers desc
        limit 15;
        '''
        result = con.execute(statement).fetchall()
        for row in result:
            print(row)
```
[118] ✓ 0.1s   Python

```
('Harper College', 'Netflix', 135)
('Harper College', 'Spotify', 131)
('Harper College', 'Amazon Prime Video', 126)
('Harper College', 'Hulu', 99)
('Harper College', 'Disney+', 90)
('Harvard University', 'Netflix', 107)
('Harvard University', 'Disney+', 106)
('Harvard University', 'Amazon Prime Video', 100)
('Harvard University', 'Spotify', 100)
('Harvard University', 'Hulu', 93)
('Massachusetts Institute of Technology', 'Spotify', 112)
('Massachusetts Institute of Technology', 'Amazon Prime Video', 104)
('Massachusetts Institute of Technology', 'Netflix', 103)
('Massachusetts Institute of Technology', 'Disney+', 93)
('Massachusetts Institute of Technology', 'Hulu', 93)
```

Initial Performance

Below is the performance of our query without any added indexes. Between each index, we made sure to drop the added index from the previous part, to ensure standardization between comparisons.

```
| -> Limit: 15 row(s)  (actual time=44.573..44.575 rows=15 loops=1)
    -> Sort: un.universityName, numUsers DESC, limit input to 15 row(s) per chunk  (actual time=44.572..44.574 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.005 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=44.503..44.508 rows=30 loops=1)
                -> Nested loop inner join  (cost=963.50 rows=400) (actual time=0.252..40.955 rows=3239 loops=1)
                    -> Nested loop inner join  (cost=823.50 rows=400) (actual time=0.246..38.364 rows=3239 loops=1)
                        -> Nested loop inner join  (cost=683.50 rows=400) (actual time=0.223..19.839 rows=3239 loops=1)
                            -> Nested loop inner join  (cost=543.50 rows=400) (actual time=0.171..16.376 rows=3239 loops=1)
                                -> Filter: (m.memberStatus = 'Accepted')  (cost=403.50 rows=400) (actual time=0.059..2.058 rows=3239 loops=1)
                                    -> Table scan on m  (cost=403.50 rows=4000) (actual time=0.056..1.349 rows=4000 loops=1)
                                -> Filter: (f.serviceName is not null)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3239)
                                    -> Single-row index lookup on f using PRIMARY (familyID=m.familyID)  (cost=0.25 rows=1) (actual time=0.004..0.004 rows=1 loops=3239)
                            -> Single-row index lookup on ss using PRIMARY (serviceName=f.serviceName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                        -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.005..0.005 rows=1 loops=3239)
                            -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.005..0.005 rows=1 loops=3239)
                    -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
|
1 row in set (0.05 sec)
```

Index 1

For our first index design, we added an index on `Membership.memberStatus` because we filter on that column in the query.

```
alter table Membership add index memId(memberStatus);
```

The output of our analyze query follows:

```
----------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=58.875..58.878 rows=15 loops=1)
    -> Sort: un.universityName, numUsers DESC, limit input to 15 row(s) per chunk  (actual time=58.875..58.876 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.008 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=58.825..58.833 rows=30 loops=1)
                -> Nested loop inner join  (cost=3545.56 rows=3028) (actual time=0.583..55.395 rows=3239 loops=1)
                    -> Nested loop inner join  (cost=2485.60 rows=3028) (actual time=0.573..52.645 rows=3239 loops=1)
                        -> Nested loop inner join  (cost=1425.64 rows=3028) (actual time=0.558..46.791 rows=3239 loops=1)
                            -> Nested loop inner join  (cost=116.64 rows=935) (actual time=0.316..1.059 rows=1000 loops=1)
                                -> Index scan on ss using PRIMARY  (cost=0.75 rows=5) (actual time=0.137..0.149 rows=5 loops=1)
                                -> Index lookup on f using subscriptionService (serviceName=ss.serviceName)  (cost=8.22 rows=187) (actual time=0.062..0.168 rows=200 loops=5)
                            -> Filter: (m.memberStatus = 'Accepted')  (cost=1.00 rows=3) (actual time=0.041..0.045 rows=3 loops=1000)
                                -> Index lookup on m using familyID (familyID=f.familyID)  (cost=1.00 rows=4) (actual time=0.041..0.045 rows=4 loops=1000)
                        -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.001..0.002 rows=1 loops=3239)
                            -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                    -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
|
+----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------+
1 row in set (0.06 sec)
```

This time around, we find that the new index actually slows down both the individual runtime of our filtering step and the runtime of the overall query, so this would not be a good index to add to optimize our second query.

Index 2

For our second index design, we added an index on `Family.familyID` because it is used to join the `Family` table with the `Membership` table.

```
alter table Family add index famId(familyID);
```

The output of our analyze query follows:

```
| -> Limit: 15 row(s)  (actual time=23.210..23.212 rows=15 loops=1)
    -> Sort: un.universityName, numUsers DESC, limit input to 15 row(s) per chunk  (actual time=23.209..23.210 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.005 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=23.164..23.169 rows=30 loops=1)
                -> Nested loop inner join  (cost=963.50 rows=400) (actual time=0.071..19.132 rows=3239 loops=1)
                    -> Nested loop inner join  (cost=823.50 rows=400) (actual time=0.066..16.305 rows=3239 loops=1)
                        -> Nested loop inner join  (cost=683.50 rows=400) (actual time=0.061..11.567 rows=3239 loops=1)
                            -> Nested loop inner join  (cost=543.50 rows=400) (actual time=0.055..7.796 rows=3239 loops=1)
                                -> Filter: (m.memberStatus = 'Accepted')  (cost=403.50 rows=400) (actual time=0.044..2.411 rows=3239 loops=1)
                                    -> Table scan on m  (cost=403.50 rows=4000) (actual time=0.041..1.614 rows=4000 loops=1)
                                -> Filter: (f.serviceName is not null)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                                    -> Single-row index lookup on f using PRIMARY (familyID=m.familyID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                            -> Single-row index lookup on ss using PRIMARY (serviceName=f.serviceName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                        -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                            -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                    -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
|
+----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------
----------+
1 row in set (0.03 sec)
```

We find a significant improvement from using the new index. There is a cascaded speedup in the runtime of all of the nested inner joins, and a resulting speedup on the overall query as well, shown at the bottom.

Index 3

For our third index design, we tried a similar approach as the previous indexing step, adding an indexing to `Membership.memberID`, which is used in one of the inner joins.

```
alter table Membership add index memId(memberID);
```

The output of our analyze query follows:

```
| -> Limit: 15 row(s)  (actual time=19.492..19.494 rows=15 loops=1)
    -> Sort: un.universityName, numUsers DESC, limit input to 15 row(s) per chunk  (actual time=19.491..19.493 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=0.001..0.004 rows=30 loops=1)
            -> Aggregate using temporary table  (actual time=19.448..19.453 rows=30 loops=1)
                -> Nested loop inner join  (cost=963.50 rows=400) (actual time=0.093..16.087 rows=3239 loops=1)
                    -> Nested loop inner join  (cost=823.50 rows=400) (actual time=0.088..13.675 rows=3239 loops=1)
                        -> Nested loop inner join  (cost=683.50 rows=400) (actual time=0.083..9.567 rows=3239 loops=1)
                            -> Nested loop inner join  (cost=543.50 rows=400) (actual time=0.074..6.457 rows=3239 loops=1)
                                -> Filter: (m.memberStatus = 'Accepted')  (cost=403.50 rows=400) (actual time=0.060..1.910 rows=3239 loops=1)
                                    -> Table scan on m  (cost=403.50 rows=4000) (actual time=0.056..1.261 rows=4000 loops=1)
                                -> Filter: (f.serviceName is not null)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                                    -> Single-row index lookup on f using PRIMARY (familyID=m.familyID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                            -> Single-row index lookup on ss using PRIMARY (serviceName=f.serviceName)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                        -> Filter: (us.universityID is not null)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                            -> Single-row index lookup on us using PRIMARY (userID=m.memberID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
                    -> Single-row index lookup on un using PRIMARY (universityID=us.universityID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=3239)
|
+-------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------
---------+
1 row in set (0.02 sec)
```

We find an even bigger improvement by using the `memberID` index, again both in the several inner joins, and in the query as a whole, more than halving the runtime. As a result, we might next try to implement both indexes at once, to increase the speedup of the query even further, but this would need to be analyzed again to be sure.