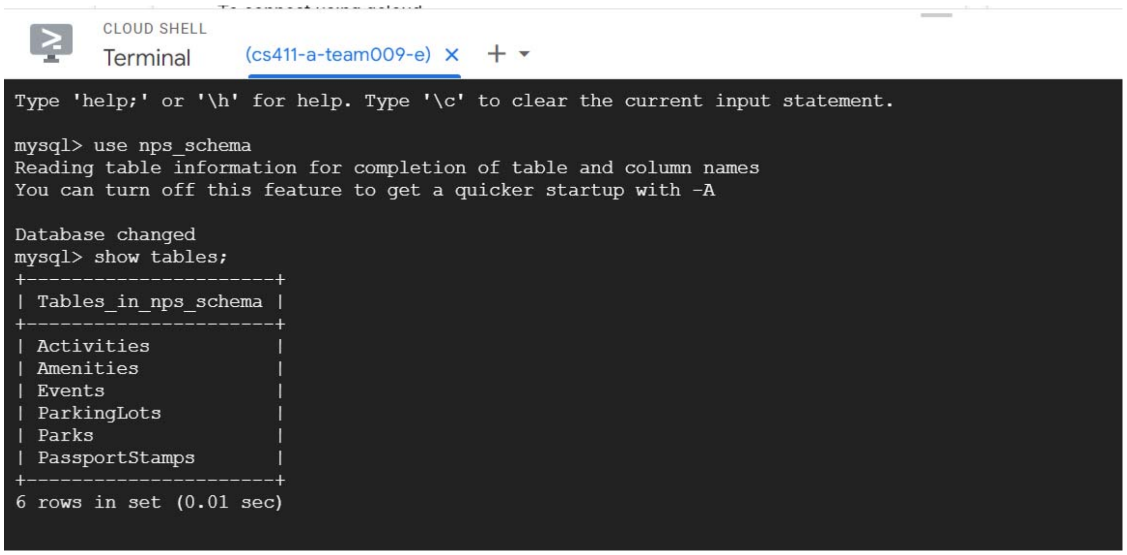
Database Design

Database Implementation



```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use nps_schema
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------------+
| Tables_in_nps_schema |
+----------------------+
| Activities           |
| Amenities            |
| Events               |
| ParkingLots          |
| Parks                |
| PassportStamps       |
+----------------------+
6 rows in set (0.01 sec)
```

<u>Parks Table Schema</u>

```sql
CREATE TABLE `Parks` (
`name` VARCHAR(1024),
`parkCode` VARCHAR(1024) PRIMARY KEY,
`description` VARCHAR(1024),
`stateAbbr` VARCHAR(1024),
`directionsUrl` VARCHAR(1024),
`imageTitle` VARCHAR(1024),
`imageCaption` VARCHAR(1024),
`imageUrl` VARCHAR(1024)
);
```

<u>Amenities Table Schema</u>

```sql
CREATE TABLE `Amenities` (
`amenID` VARCHAR(1024) PRIMARY KEY,
`name` VARCHAR(1024),
`parkCode` VARCHAR(1024) REFERENCES parks(parkCode)
);
```

<u>PassportStamps Table Schema</u>

```sql
CREATE TABLE `PassportStamps` (
`stamplocationid` VARCHAR(1024) PRIMARY KEY,
`label` VARCHAR(1024),
`type` VARCHAR(1024),
`parkCode` VARCHAR(1024) REFERENCES Parks(parkCode)
);
```

Activities Table Schema

```
CREATE TABLE `Activities` (
`activID` VARCHAR(1024) PRIMARY KEY,
`title` VARCHAR(1024),
`parkCode` VARCHAR(1024) REFERENCES parks(parkCode),
`description` VARCHAR(1024),
`image_title` VARCHAR(1024),
`image_url` VARCHAR(1024),
`image_caption` VARCHAR(1024),
`petsAllowed` BOOL,
`season` VARCHAR(1024),
`hasFee` BOOL
);
```

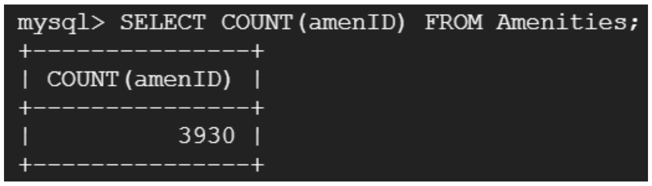
ParkingLots Table Schema

```
CREATE TABLE `ParkingLots` (
`lotID` VARCHAR(1024) PRIMARY KEY,
`name` VARCHAR(1024),
`description` VARCHAR(1024),
`latitude` FLOAT,
`longitude` FLOAT,
`hasFee` BOOL,
`parkCode` VARCHAR(1024) REFERENCES parks(parkCode),
`numSpaces` INT
);
```

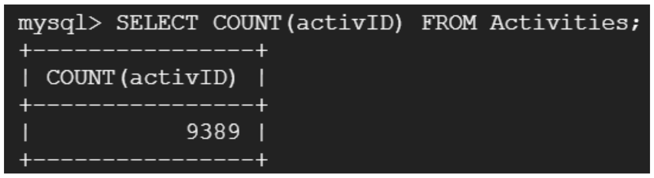
Events Table Schema

```
CREATE TABLE `Events` (
`dateStart` VARCHAR(1024),
`description` VARCHAR(1024),
`category` VARCHAR(1024),
`eventID` VARCHAR(1024) PRIMARY KEY,
`dateEnd` VARCHAR(1024),
`parkFullName` VARCHAR(1024) REFERENCES parks(name),
`title` VARCHAR(1024),
`image` VARCHAR(1024),
`hasFee` BOOL
);
```

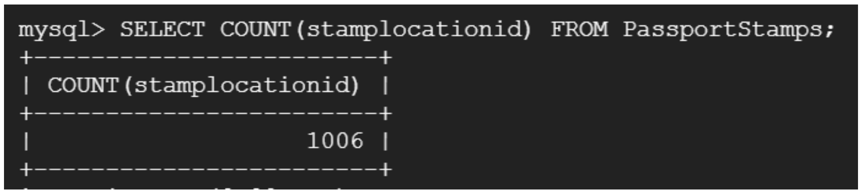
Row Counts



```
mysql> SELECT COUNT(amenID) FROM Amenities;
+---------------+
| COUNT(amenID) |
+---------------+
|          3930 |
+---------------+
```



```
mysql> SELECT COUNT(activID) FROM Activities;
+----------------+
| COUNT(activID) |
+----------------+
|           9389 |
+----------------+
```

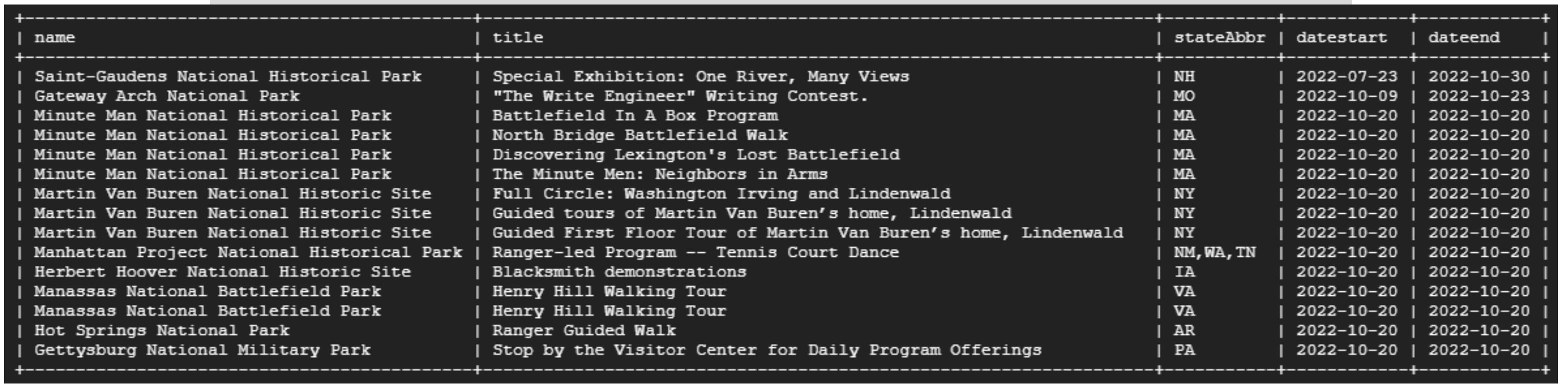


```
mysql> SELECT COUNT(stamplocationid) FROM PassportStamps;
+------------------------+
| COUNT(stamplocationid) |
+------------------------+
|                   1006 |
+------------------------+
```

Advanced Queries

1. Returns the park’s name and states it’s located in, event title, event start date, and event end date for all events located in national parks with free parking

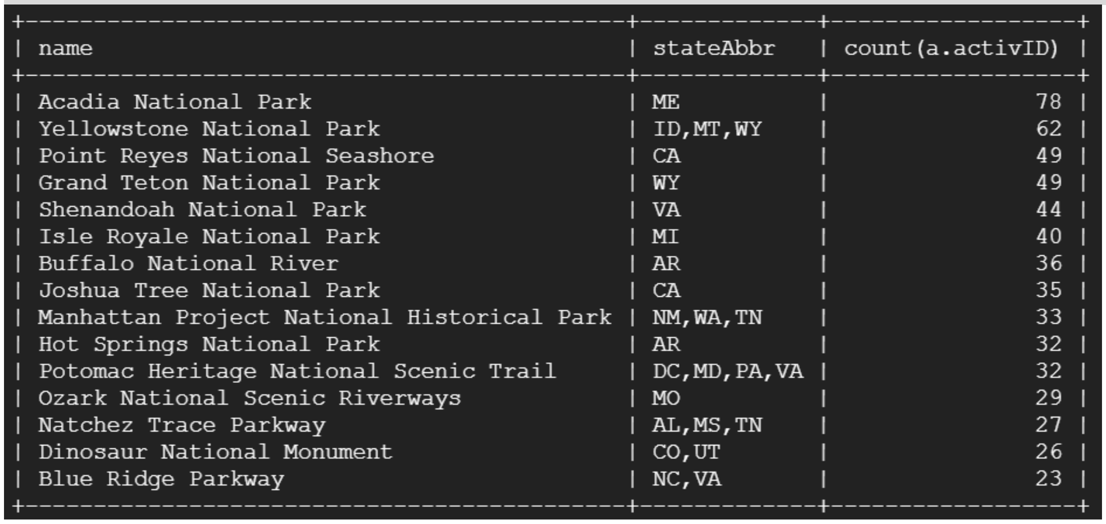
```
SELECT name, title, stateAbbr, datestart, dateend
FROM  Parks p1 JOIN Events e1 ON (p1.name = e1.parkfullname)
WHERE name NOT IN (SELECT p.name
                                        FROM Parks p JOIN ParkingLots l ON (p.parkCode = l.parkCode)
                                        WHERE hasFee=True)
ORDER BY datestart
LIMIT 15
```



```
+--------------------------------------------+------------------------------------------------------------------+-----------+------------+------------+
| name                                       | title                                                            | stateAbbr | datestart  | dateend    |
+--------------------------------------------+------------------------------------------------------------------+-----------+------------+------------+
| Saint-Gaudens National Historical Park     | Special Exhibition: One River, Many Views                        | NH        | 2022-07-23 | 2022-10-30 |
| Gateway Arch National Park                 | "The Write Engineer" Writing Contest.                            | MO        | 2022-10-09 | 2022-10-23 |
| Minute Man National Historical Park        | Battlefield In A Box Program                                     | MA        | 2022-10-20 | 2022-10-20 |
| Minute Man National Historical Park        | North Bridge Battlefield Walk                                    | MA        | 2022-10-20 | 2022-10-20 |
| Minute Man National Historical Park        | Discovering Lexington's Lost Battlefield                         | MA        | 2022-10-20 | 2022-10-20 |
| Minute Man National Historical Park        | The Minute Men: Neighbors in Arms                                | MA        | 2022-10-20 | 2022-10-20 |
| Martin Van Buren National Historic Site    | Full Circle: Washington Irving and Lindenwald                    | NY        | 2022-10-20 | 2022-10-20 |
| Martin Van Buren National Historic Site    | Guided tours of Martin Van Buren’s home, Lindenwald            | NY        | 2022-10-20 | 2022-10-20 |
| Martin Van Buren National Historic Site    | Guided First Floor Tour of Martin Van Buren’s home, Lindenwald | NY        | 2022-10-20 | 2022-10-20 |
| Manhattan Project National Historical Park | Ranger-led Program -- Tennis Court Dance                         | NM,WA,TN  | 2022-10-20 | 2022-10-20 |
| Herbert Hoover National Historic Site      | Blacksmith demonstrations                                        | IA        | 2022-10-20 | 2022-10-20 |
| Manassas National Battlefield Park         | Henry Hill Walking Tour                                          | VA        | 2022-10-20 | 2022-10-20 |
| Manassas National Battlefield Park         | Henry Hill Walking Tour                                          | VA        | 2022-10-20 | 2022-10-20 |
| Hot Springs National Park                  | Ranger Guided Walk                                               | AR        | 2022-10-20 | 2022-10-20 |
| Gettysburg National Military Park          | Stop by the Visitor Center for Daily Program Offerings           | PA        | 2022-10-20 | 2022-10-20 |
+--------------------------------------------+------------------------------------------------------------------+-----------+------------+------------+
```

2. For each park, returns its name, all the states it’s located in, and the total number of Fall activities that are free

```
SELECT p.name, p.stateAbbr, count(a.activID)
FROM nps_schema.Activities a JOIN nps_schema.Parks p ON (a.parkCode = p.parkCode)
WHERE season = 'Fall' AND a.hasFee = false
GROUP BY a.parkCode
ORDER BY count(a.activID) DESC
LIMIT 15
```



```
+--------------------------------------------+-------------+------------------+
| name                                       | stateAbbr   | count(a.activID) |
+--------------------------------------------+-------------+------------------+
| Acadia National Park                       | ME          |               78 |
| Yellowstone National Park                  | ID,MT,WY    |               62 |
| Point Reyes National Seashore              | CA          |               49 |
| Grand Teton National Park                  | WY          |               49 |
| Shenandoah National Park                   | VA          |               44 |
| Isle Royale National Park                  | MI          |               40 |
| Buffalo National River                     | AR          |               36 |
| Joshua Tree National Park                  | CA          |               35 |
| Manhattan Project National Historical Park | NM,WA,TN    |               33 |
| Hot Springs National Park                  | AR          |               32 |
| Potomac Heritage National Scenic Trail     | DC,MD,PA,VA |               32 |
| Ozark National Scenic Riverways            | MO          |               29 |
| Natchez Trace Parkway                      | AL,MS,TN    |               27 |
| Dinosaur National Monument                 | CO,UT       |               26 |
| Blue Ridge Parkway                         | NC,VA       |               23 |
+--------------------------------------------+-------------+------------------+
```

Indexing:

1. Before creating any index, the output of EXPLAIN ANALYZE was:

```
-> Sort: e1.datestart  (actual time=124.798..124.905 rows=731 loops=1)
    -> Stream results  (cost=36553.31 rows=36218) (actual time=120.945..122.640 rows=731
loops=1)
        -> Filter: (e1.parkfullname = p1.`name`)  (cost=36553.31 rows=36218) (actual
time=120.941..122.176 rows=731 loops=1)
            -> Inner hash join (<hash>(e1.parkfullname)=<hash>(p1.`name`))  (cost=36553.31
rows=36218) (actual time=120.940..121.931 rows=731 loops=1)
                -> Table scan on e1  (cost=0.64 rows=796) (actual time=0.033..0.610 rows=820
loops=1)
                -> Hash
                    -> Filter: <in_optimizer>(p1.`name`,<exists>(select #2) is false)  (cost=52.25
rows=455) (actual time=0.423..120.186 rows=456 loops=1)
                        -> Table scan on p1  (cost=52.25 rows=455) (actual time=0.057..0.364 rows=467
loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Limit: 1 row(s)  (cost=63.87 rows=1) (actual time=0.256..0.256 rows=0
loops=467)
                                -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(p.`name`), true)
(cost=63.87 rows=45) (actual time=0.256..0.256 rows=0 loops=467)
                                    -> Nested loop inner join  (cost=63.87 rows=45) (actual
time=0.256..0.256 rows=0 loops=467)
                                        -> Filter: ((l.hasFee = true) and (l.parkCode is not null))  (cost=48.15
rows=45) (actual time=0.005..0.170 rows=56 loops=467)
                                            -> Table scan on l  (cost=48.15 rows=449) (actual time=0.004..0.136
rows=441 loops=467)
                                        -> Filter: <if>(outer_field_is_not_null, ((<cache>(p1.`name`) =
p.`name`) or (p.`name` is null)), true)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0
loops=26165)
                                            -> Single-row index lookup on p using PRIMARY
(parkCode=l.parkCode)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=26165)
```

We first tried creating an index based on the parkCode column in the ParkingLots table. Since we reference it so much, we figured that indexing the parkCodes would improve the ability of the subquery to find the parks that it was looking for. After creating this index CREATE INDEX code_idx ON ParkingLots(parkCode), the output of the EXPLAIN ANALYZE was:

```
-> Sort: e1.datestart  (actual time=497.439..497.545 rows=731 loops=1)
    -> Stream results  (cost=36553.31 rows=36218) (actual time=493.210..494.898 rows=731
loops=1)
        -> Filter: (e1.parkfullname = p1.`name`)  (cost=36553.31 rows=36218) (actual
time=493.204..494.435 rows=731 loops=1)
```

```
      -> Inner hash join (<hash>(e1.parkfullname)=<hash>(p1.`name`))  (cost=36553.31
rows=36218) (actual time=493.203..494.193 rows=731 loops=1)
        -> Table scan on e1  (cost=0.64 rows=796) (actual time=0.047..0.645 rows=820
loops=1)
        -> Hash
          -> Filter: <in_optimizer>(p1.`name`,<exists>(select #2) is false)  (cost=52.25
rows=455) (actual time=2.141..492.210 rows=456 loops=1)
            -> Table scan on p1  (cost=52.25 rows=455) (actual time=0.053..0.741 rows=467
loops=1)
            -> Select #2 (subquery in condition; dependent)
              -> Limit: 1 row(s)  (cost=63.87 rows=1) (actual time=1.051..1.051 rows=0
loops=467)
                -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(p.`name`), true)
(cost=63.87 rows=45) (actual time=1.050..1.050 rows=0 loops=467)
                  -> Nested loop inner join  (cost=63.87 rows=45) (actual
time=1.050..1.050 rows=0 loops=467)
                    -> Filter: (l.hasFee = true)  (cost=48.15 rows=45) (actual
time=0.025..0.987 rows=56 loops=467)
                      -> Index range scan on l using code_idx, with index condition:
(l.parkCode is not null)  (cost=48.15 rows=449) (actual time=0.024..0.958 rows=443 loops=467)
                    -> Filter: <if>(outer_field_is_not_null, ((<cache>(p1.`name`) =
p.`name`) or (p.`name` is null)), true)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0
loops=26348)
                      -> Single-row index lookup on p using PRIMARY
(parkCode=l.parkCode)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=26348)
```

However, after checking the output, we noticed that the runtime is abysmal and increased by almost four times after our indexing. We suppose that since we use parkCodes in the FROM clause, there is no reason to index anything since the FROM clause simply gathers everything from the table. Rather than saving time with indices, we instead ended up adding overhead that SQL must go through in order to reach each data point once.

We decided to next try creating an index for Parks(parkName) because the FROM clause uses the park name to join the Events and Parks together. If the query has indices for the park name, we guessed that the lookup would be faster. After dropping the index `DROP INDEX code_idx ON ParkingLots(parkCode)` and creating this index `CREATE INDEX name_idx ON Parks(name)` the output of EXPLAIN ANALYZE was:

```
-> Sort: e1.datestart  (actual time=129.211..129.315 rows=731 loops=1)
  -> Stream results  (cost=36553.31 rows=36218) (actual time=125.121..126.870 rows=731
loops=1)
    -> Filter: (e1.parkfullname = p1.`name`)  (cost=36553.31 rows=36218) (actual
time=125.117..126.362 rows=731 loops=1)
```

```
            -> Inner hash join (<hash>(e1.parkfullname)=<hash>(p1.`name`))  (cost=36553.31
rows=36218) (actual time=125.116..126.119 rows=731 loops=1)
                -> Table scan on e1  (cost=0.64 rows=796) (actual time=0.037..0.632 rows=820
loops=1)
                -> Hash
                    -> Filter: <in_optimizer>(p1.`name`,<exists>(select #2) is false)  (cost=52.25
rows=455) (actual time=0.862..124.048 rows=456 loops=1)
                        -> Table scan on p1  (cost=52.25 rows=455) (actual time=0.110..0.492 rows=467
loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Limit: 1 row(s)  (cost=63.87 rows=1) (actual time=0.264..0.264 rows=0
loops=467)
                                -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(p.`name`), true)
(cost=63.87 rows=45) (actual time=0.264..0.264 rows=0 loops=467)
                                    -> Nested loop inner join  (cost=63.87 rows=45) (actual
time=0.263..0.263 rows=0 loops=467)
                                        -> Filter: ((l.hasFee = true) and (l.parkCode is not null))  (cost=48.15
rows=45) (actual time=0.006..0.174 rows=56 loops=467)
                                            -> Table scan on l  (cost=48.15 rows=449) (actual time=0.005..0.138
rows=441 loops=467)
                                        -> Filter: <if>(outer_field_is_not_null, ((<cache>(p1.`name`) =
p.`name`) or (p.`name` is null)), true)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0
loops=26165)
                                            -> Single-row index lookup on p using PRIMARY
(parkCode=l.parkCode)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=26165)
```

After comparing the runtimes of the two queries, we observed that they are almost identical. We hypothesized that the runtime was unaffected because the (sub)query must also search through hasFee and parkCode, meaning that parkNames is not the highest priority column to index. In addition to that, there are only about 500 parks to search, so there are a higher number of items in other possible index designs.

We instantiated an index on the column hasFee in ParkingLots. The main reasoning behind this was, again, to facilitate the work performed by the WHERE clause in the subquery; each parking lot in the ParkingLots table has a boolean hasFee value. With this index design, we try to index the WHERE clause that the (sub)query must make. After dropping the index: DROP INDEX name_idx ON Parks and creating this index: CREATE INDEX fee_idx ON ParkingLots(hasFee), the output of EXPLAIN ANALYZE was:

```
-> Sort: e1.datestart  (actual time=93.824..93.929 rows=731 loops=1)
    -> Stream results  (cost=36553.31 rows=36218) (actual time=88.669..90.412 rows=731
loops=1)
        -> Filter: (e1.parkfullname = p1.`name`)  (cost=36553.31 rows=36218) (actual
time=88.660..89.946 rows=731 loops=1)
```

```
            -> Inner hash join (<hash>(e1.parkfullname)=<hash>(p1.`name`))  (cost=36553.31 rows=36218) (actual time=88.658..89.693 rows=731 loops=1)
                -> Table scan on e1  (cost=0.64 rows=796) (actual time=0.375..1.018 rows=820 loops=1)
                -> Hash
                    -> Filter: <in_optimizer>(p1.`name`,<exists>(select #2) is false)  (cost=52.25 rows=455) (actual time=0.370..87.828 rows=456 loops=1)
                        -> Table scan on p1  (cost=52.25 rows=455) (actual time=0.048..0.436 rows=467 loops=1)
                        -> Select #2 (subquery in condition; dependent)
                            -> Limit: 1 row(s)  (cost=35.40 rows=1) (actual time=0.186..0.186 rows=0 loops=467)
                                -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(p.`name`), true)  (cost=35.40 rows=57) (actual time=0.186..0.186 rows=0 loops=467)
                                    -> Nested loop inner join  (cost=35.40 rows=57) (actual time=0.186..0.186 rows=0 loops=467)
                                        -> Filter: (l.parkCode is not null)  (cost=15.45 rows=57) (actual time=0.017..0.095 rows=56 loops=467)
                                            -> Index lookup on l using fee_idx (hasFee=true)  (cost=15.45 rows=57) (actual time=0.015..0.088 rows=56 loops=467)
                                        -> Filter: <if>(outer_field_is_not_null, ((<cache>(p1.`name`) = p.`name`) or (p.`name` is null)), true)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=0 loops=26165)
                                            -> Single-row index lookup on p using PRIMARY (parkCode=l.parkCode)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=26165)
```

This index was fairly successful, bringing the actual time down to about 93 from 127, a 27% decrease in runtime. We surmise that the subquery is being executed a greater amount of times than the outer query, thus affecting the total runtime more than indexing the outer query even though they both should decrease the overall time. Therefore, we decided to use this final index design for optimizing the SQL query.

2. Before creating any index, the output of EXPLAIN ANALYZE was:

```
'-> Sort: p.`name`  (actual time=16.947..16.982 rows=304 loops=1)
    -> Table scan on <temporary>  (actual time=0.004..0.076 rows=304 loops=1)
        -> Aggregate using temporary table  (actual time=15.238..15.330 rows=304 loops=1)
            -> Nested loop inner join  (cost=1157.63 rows=96) (actual time=0.144..11.577 rows=2088 loops=1)
                -> Filter: ((a.hasFee = false) and (a.season = "Fall") and (a.parkCode is not null))  (cost=1124.20 rows=96) (actual time=0.092..7.366 rows=2092 loops=1)
                    -> Table scan on a  (cost=1124.20 rows=9552) (actual time=0.062..5.899 rows=9389 loops=1)
            -> Single-row index lookup on p using PRIMARY (parkCode=a.parkCode)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2092)’
```

After creating this index: CREATE INDEX seasons_idx ON Activities(season);
The output of EXPLAIN ANALYZE was:

```
'-> Sort: p.`name`  (actual time=16.401..16.435 rows=304 loops=1)
   -> Table scan on <temporary>  (actual time=0.014..0.087 rows=304 loops=1)
      -> Aggregate using temporary table  (actual time=15.160..15.251 rows=304
loops=1)
         -> Nested loop inner join  (cost=349.95 rows=247) (actual time=0.076..11.496
rows=2088 loops=1)
            -> Filter: ((a.hasFee = false) and (a.parkCode is not null))  (cost=263.50
rows=247) (actual time=0.059..7.356 rows=2092 loops=1)
               -> Index lookup on a using seasons_idx (season="Fall")  (cost=263.50
rows=2470) (actual time=0.055..6.963 rows=2470 loops=1)
            -> Single-row index lookup on p using PRIMARY (parkCode=a.parkCode)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2092)’
```

Notice that the cost of the nested loop inner join was significantly reduced from 1157.63 to 349.95, a reduction of 70%. The number of rows analyzed went up from 96 to 247. The actual time to complete the inner join went from 11.577 seconds to 11.496 seconds but these values could vary between individual runs, so the true difference in actual runtime is unknown without multiple runs.

After dropping the index DROP INDEX seasons_idx ON Activities and creating this index: CREATE INDEX fee_idx ON Activities(hasFee), the output of EXPLAIN ANALYZE was:

```
'-> Sort: p.`name`  (actual time=27.232..27.267 rows=304 loops=1)
   -> Table scan on <temporary>  (actual time=0.005..0.060 rows=304 loops=1)
      -> Aggregate using temporary table  (actual time=26.219..26.292 rows=304
loops=1)
         -> Nested loop inner join  (cost=453.72 rows=478) (actual time=0.119..22.515
rows=2088 loops=1)
            -> Filter: ((a.season = "Fall") and (a.parkCode is not null))  (cost=286.56
rows=478) (actual time=0.099..18.724 rows=2092 loops=1)
               -> Index lookup on a using seasons_idx (hasFee=false)  (cost=286.56
rows=4776) (actual time=0.095..17.651 rows=7908 loops=1)
            -> Single-row index lookup on p using PRIMARY (parkCode=a.parkCode)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2092)’
```

The cost of the nested loop inner join decreased from 1157.63 to 453.72, a relatively significant reduction. However, the number of rows analyzed went up from 96 to 478. The actual time to complete the inner join is from 11.577 seconds to 22.515 seconds, however, like before these times can be affected by external factors like the internet connection.

After recreating the index CREATE INDEX seasons_idx ON Activities(season) alongside the existing index fee_idx, the output of EXPLAIN ANALYZE was:

```
'-> Sort: p.`name`  (actual time=16.569..16.628 rows=304 loops=1)
    -> Table scan on <temporary>  (actual time=0.003..0.077 rows=304 loops=1)
        -> Aggregate using temporary table  (actual time=14.989..15.084 rows=304
loops=1)
            -> Nested loop inner join  (cost=794.55 rows=1235) (actual time=0.242..11.547
rows=2088 loops=1)
                -> Filter: ((a.hasFee = false) and (a.parkCode is not null))  (cost=362.30
rows=1235) (actual time=0.225..7.773 rows=2092 loops=1)
                    -> Index lookup on a using seasons_idx (season="Fall")  (cost=362.30
rows=2470) (actual time=0.221..7.373 rows=2470 loops=1)
                -> Single-row index lookup on p using PRIMARY (parkCode=a.parkCode)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2092)’
```

The cost of the nested loop inner join with both indices was actually higher than either the index seasons_idx alone OR the fee_idx alone. The number of rows analyzed also increased significantly. This suggests that the efficiency of using both indices is actually lesser than using either index alone. Interestingly enough, the actual runtime of the nested loop inner join, 11.547 seconds, is roughly the same as the original unindexed run of EXPLAIN ANALYZE, which had an actual runtime of 11.577 seconds.

After dropping the indices DROP INDEX seasons_idx ON Activities; DROP INDEX fee_idx ON ACTIVITIES; creating this index CREATE INDEX pCode_idx on Activities(parkCode) ,the output of EXPLAIN ANALYZE was:

```
'-> Sort: p.`name`  (actual time=17.166..17.200 rows=304 loops=1)
    -> Table scan on <temporary>  (actual time=0.004..0.081 rows=304 loops=1)
        -> Aggregate using temporary table  (actual time=15.540..15.636 rows=304
loops=1)
            -> Nested loop inner join  (cost=1157.63 rows=96) (actual time=0.108..11.719
rows=2088 loops=1)
                -> Filter: ((a.hasFee = false) and (a.season = "Fall") and (a.parkCode is not
null))  (cost=1124.20 rows=96) (actual time=0.076..7.406 rows=2092 loops=1)
                    -> Table scan on a  (cost=1124.20 rows=9552) (actual time=0.069..5.919
rows=9389 loops=1)
                -> Single-row index lookup on p using PRIMARY (parkCode=a.parkCode)
(cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=2092)’
```

The cost of the nested loop inner join with the index on parkCode alone led to no significant changes in runtime or cost. The actual runtime of the nested loop inner join was slightly higher than the original un-indexed runtime, but this is likely due to external factors rather than internal efficiency. This seems to be because matching on foreign and primary keys is already an index lookup by default.

We found that our experiments with indexing for this particular SQL command were inconclusive, especially with regard to actual runtime. Significant reductions in both cost and the number of rows iterated through simultaneously were not achieved by any of the indices on the Nested loop inner join. The best choice is likely `CREATE INDEX seasons_idx ON Activities(season);` alone due to having the lowest cost value and the quickest first-row actual runtime at 0.076 seconds. However, it is unclear if this indexing would lead to a tangible performance increase on this database compared to no indexing at all.