```
create database apartmenthunt;

create table Providers (
      id INT PRIMARY KEY,
      firstName VARCHAR(100) NOT NULL,
      lastName VARCHAR(100) NOT NULL,
      email VARCHAR(100) NOT NULL,
      phone VARCHAR(50) NOT NULL,
      password VARCHAR(100) NOT NULL,
      company VARCHAR(100) NOT NULL
);

create table Buildings (
      id INT PRIMARY KEY,
      address VARCHAR(200) NOT NULL,
      city VARCHAR(100) NOT NULL,
      state VARCHAR(2) NOT NULL,
      zip INT NOT NULL,
      bikeRacks INT,
      parkingSpaces INT
);

create table Units (
      id INT PRIMARY KEY,
      postedBy INT NOT NULL,
      unitOf INT NOT NULL,
      moveIn DATE NOT NULL,
      moveOut DATE NOT NULL,
      rentCost DECIMAL(10,2),
      canSublease BOOL,
      gasUtility DECIMAL(10,2),
      electricityUtility DECIMAL(10,2),
      waterUtility DECIMAL(10,2),
      internetUtility DECIMAL(10,2),
      unitNumber INT,
      FOREIGN KEY (postedBy) REFERENCES Providers(id) ON DELETE
CASCADE ON UPDATE CASCADE,
      FOREIGN KEY (unitOf) REFERENCES Buildings(id) ON DELETE CASCADE
ON UPDATE CASCADE
);

#I think the function to add the units should somehow give the
possible IDs for a specific building address.
```

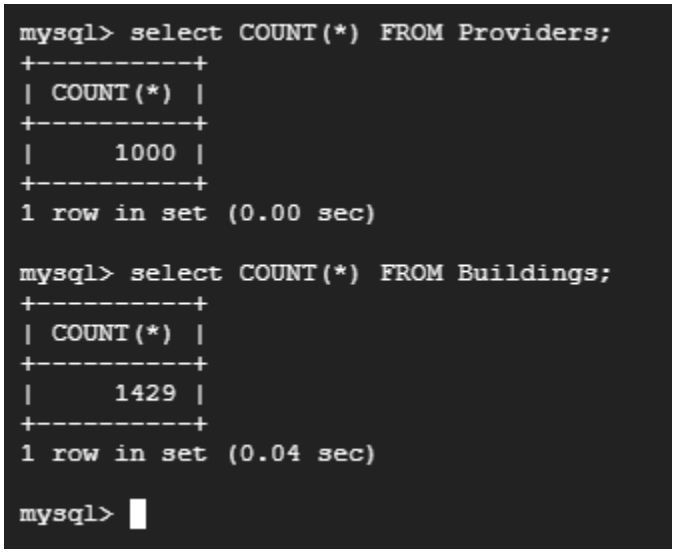
```
create table Ratings (
	id INT PRIMARY KEY,
	ratedBuilding INT NOT NULL,
	ratedBy INT NOT NULL,
	travelTimeMinutes INT,
	travelRating INT,
	noiseRating INT,
	airRating INT,
	furnishingRating INT,
	spaceRating INT,
	creationDate DATE,
	comments VARCHAR(1000),
	FOREIGN KEY (ratedBuilding) REFERENCES Buildings(id) ON DELETE
CASCADE ON UPDATE CASCADE,
	FOREIGN KEY (ratedBy) REFERENCES Users(id) ON DELETE CASCADE ON
UPDATE CASCADE
);

create table Tracking (
	id INT PRIMARY KEY,
	trackedBy INT NOT NULL,
	trackedUnit INT NOT NULL,
	startDate DATE NOT NULL,
	endDate DATE,
	accepted BOOL NOT NULL DEFAULT 0,
	FOREIGN KEY (trackedBy) REFERENCES Users(id) ON DELETE CASCADE
ON UPDATE CASCADE,
	FOREIGN KEY (trackedUnit) REFERENCES Units(id) ON DELETE
CASCADE ON UPDATE CASCADE
);

Create table Users (
	id INT PRIMARY KEY,
	firstName VARCHAR(100) NOT NULL,
	lastName VARCHAR(100) NOT NULL,
	email VARCHAR(100) NOT NULL,
	phone VARCHAR(50) NOT NULL,
	password VARCHAR(100) NOT NULL

);
```

final count queries for tables with insertions from real data.



```
mysql> select COUNT(*) FROM Providers;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql> select COUNT(*) FROM Buildings;
+----------+
| COUNT(*) |
+----------+
|     1429 |
+----------+
1 row in set (0.04 sec)

mysql>
```

```
#example queries

#example query 1, selects the unit IDs, building IDs, and building
addresses of unoccupied units.

SELECT u.id, b.id, b.address
FROM Units u JOIN Buildings b ON u.UnitOf = b.id
WHERE NOT EXISTS (SELECT * FROM Tracking t WHERE t.trackedUnit = u.id
AND t.accepted > 0);
```

id	id	address
608	7	608 Kingsley St Ste A
19	12	607 Gundersen Dr
17	17	14335 Jamestown Rd
133	18	2901 Butterfield Rd
179	18	2901 Butterfield Rd
518	21	540 Clifford Ave Unit Ofce
697	23	906 W Belmont Ave
461	24	330 W Diversey Pkwy
498	27	1350 N Wells St Ofc
55	28	3633 Breakers Dr
300	28	3633 Breakers Dr
933	31	901 S 1st St Ste 8
954	32	5413 S Woodlawn Ave
604	35	2015 S Finley Rd
880	38	500 N Dearborn St Ste 1016
333	44	77 W Huron St
104	52	1124 University Dr Apt 3
942	52	1124 University Dr Apt 3
525	53	505 N State St
917	53	505 N State St

20 rows in set (0.00 sec)

#example query 2, selects the building ids and average ratings of buildings with addresses that start with the letter “A”.

```
SELECT b.id, AVG(travelTimeMinutes), AVG(travelRating),
AVG(noiseRating), AVG(airRating), AVG(furnishingRating),
AVG(spaceRating)
```

id	AVG(travelTimeMinutes)	AVG(travelRating)	AVG(noiseRating)	AVG(airRating)	AVG(furnishingRating)	AVG(spaceRating)
891	15.0000	9.0000	7.0000	5.0000	6.0000	1.0000
545	6.0000	6.0000	0.0000	0.0000	10.0000	3.0000
456	12.0000	4.0000	0.0000	3.0000	2.0000	10.0000
1262	9.5000	7.5000	3.5000	6.5000	7.0000	6.0000
87	6.0000	7.0000	3.0000	2.0000	3.0000	4.0000
1077	7.0000	4.0000	9.0000	1.0000	4.0000	5.0000
890	12.5000	4.5000	7.5000	3.0000	6.0000	7.5000
1357	9.0000	6.0000	5.0000	10.0000	5.0000	5.0000
898	13.0000	5.0000	10.0000	10.0000	9.0000	9.0000
734	6.0000	0.0000	2.0000	7.0000	5.0000	1.0000
457	14.0000	2.0000	8.0000	8.0000	4.0000	1.0000
1102	19.0000	10.0000	3.0000	0.0000	7.0000	3.0000
64	17.5000	3.5000	3.5000	7.5000	4.5000	3.5000
397	13.0000	3.0000	1.0000	8.0000	10.0000	7.0000
310	14.0000	8.0000	3.5000	6.5000	5.5000	5.5000
467	9.0000	6.0000	1.0000	5.0000	2.0000	8.0000
1286	10.0000	4.0000	8.5000	8.5000	3.5000	5.5000
1097	14.7500	4.7500	2.2500	7.5000	4.7500	7.2500
1208	14.0000	8.0000	5.0000	8.0000	8.0000	0.5000
260	4.0000	4.0000	10.0000	3.0000	4.0000	1.0000
1332	15.0000	2.0000	1.0000	0.0000	4.0000	2.0000
606	13.3333	4.0000	4.3333	4.3333	6.6667	4.0000

```
FROM  Ratings r JOIN Buildings b ON r.ratedBuilding = b.id
WHERE r.creationDate >= ALL (SELECT creationDate FROM Ratings r1
WHERE r.ratedBy = r1.ratedBy)
GROUP BY b.id;
```

#explain analyze for query 1:

```
| EXPLAIN
|
+--------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=129150.15 rows=643536) (actual time=0.454..2.343 rows=704 loops=1)
    -> Nested loop antijoin  (cost=64551.30 rows=643536) (actual time=0.440..1.326 rows=704 loops=1)
        -> Index scan on u using unitOf  (cost=99.60 rows=981) (actual time=0.066..0.263 rows=981 loops=1)
        -> Single-row index lookup on <subquery2> using <auto_distinct_key> (trackedUnit=u.id)  (actual time=0.000..0.000 rows=0 loops=981)
            -> Materialize with deduplication  (cost=131.95..131.95 rows=656) (actual time=0.860..0.860 rows=277 loops=1)
                -> Filter: (t.trackedUnit is not null)  (cost=66.35 rows=656) (actual time=0.049..0.303 rows=320 loops=1)
                    -> Filter: (t.accepted > 0)  (cost=66.35 rows=656) (actual time=0.048..0.264 rows=320 loops=1)
                        -> Table scan on t  (cost=66.35 rows=656) (actual time=0.044..0.187 rows=656 loops=1)
    -> Single-row index lookup on b using PRIMARY (id=u.unitOf)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=704)
 |
+--------------------------------------------------------------------------------+
1 row in set, 1 warning (0.01 sec)
```

#explain analyze for query 2:

```
|
+--------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=0.001..0.032 rows=504 loops=1)
    -> Aggregate using temporary table  (actual time=7.919..7.993 rows=504 loops=1)
        -> Nested loop inner join  (cost=442.95 rows=981) (actual time=0.112..7.130 rows=617 loops=1)
            -> Filter: <not>(<in_optimizer>(r.creationDate,<exists>(select #2)))  (cost=99.60 rows=981) (actual time=0.104..6.172 rows=617 loops=1)
                -> Table scan on r  (cost=99.60 rows=981) (actual time=0.062..0.486 rows=981 loops=1)
                -> Select #2 (subquery in condition; dependent)
                    -> Limit: 1 row(s)  (cost=0.56 rows=1) (actual time=0.005..0.005 rows=0 loops=981)
                        -> Filter: <if>(outer_field_is_not_null, <is_not_null_test>(r1.creationDate), true)  (cost=0.56 rows=2) (actual time=0.005..0.005 rows=0 loops=981)
                            -> Filter: <if>(outer_field_is_not_null, ((<cache>(r.creationDate) < r1.creationDate) or (r1.creationDate is null)), true)  (cost=0.56 rows=2) (actual time=0.005..0.005 rows=0 loops=981)
                                -> Index lookup on r1 using ratedBy (ratedBy=r.ratedBy)  (cost=0.56 rows=2) (actual time=0.004..0.004 rows=2 loops=981)
            -> Single-row index lookup on b using PRIMARY (id=r.ratedBuilding)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=617)
 |
+--------------------------------------------------------------------------------+
1 row in set, 1 warning (0.02 sec)
```

#test index 1

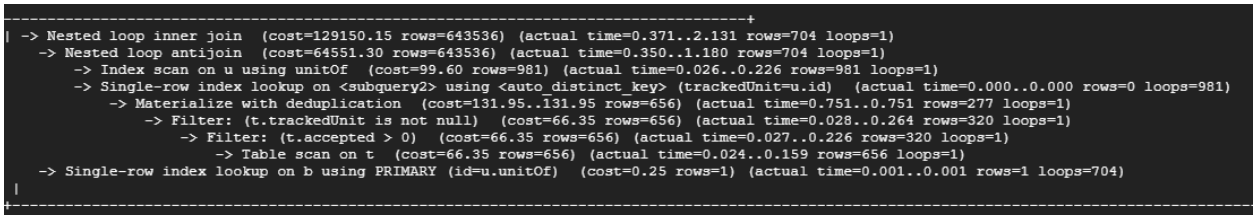
CREATE INDEX trackedUnitIdIndex ON Tracking (trackedUnit);

```
---------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=129150.15 rows=643536) (actual time=0.434..2.212 rows=704 loops=1)
    -> Nested loop antijoin  (cost=64551.30 rows=643536) (actual time=0.423..1.253 rows=704 loops=1)
        -> Index scan on u using unitOf  (cost=99.60 rows=981) (actual time=0.044..0.233 rows=981 loops=1)
        -> Single-row index lookup on <subquery2> using <auto_distinct_key> (trackedUnit=u.id)  (actual time=0.000..0.000 rows=0 loops=981)
            -> Materialize with deduplication  (cost=131.95..131.95 rows=656) (actual time=0.819..0.819 rows=277 loops=1)
                -> Filter: (t.trackedUnit is not null)  (cost=66.35 rows=656) (actual time=0.024..0.317 rows=320 loops=1)
                    -> Filter: (t.accepted > 0)  (cost=66.35 rows=656) (actual time=0.023..0.277 rows=320 loops=1)
                        -> Table scan on t  (cost=66.35 rows=656) (actual time=0.020..0.203 rows=656 loops=1)
    -> Single-row index lookup on b using PRIMARY (id=u.unitOf)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=704)
|
---------------------------------------------------------------------------------+
1 row in set, 1 warning (0.01 sec)
```

There was no noticeable change compared to normal. I think this is because the unitOf was in the Units table, while the index I tried creating was in the trackedUnit table. I was hoping it would let the query do the existence subquery faster, but I realize that it’s meant to scan the whole table anyways.

#test index 2

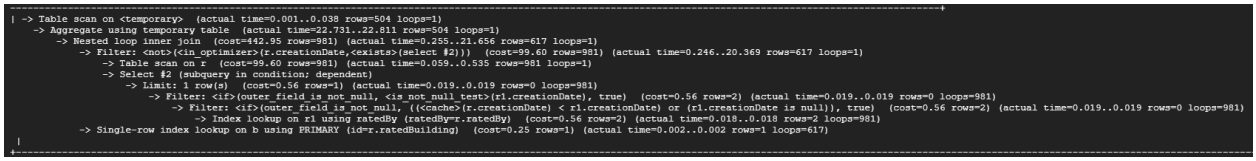
CREATE INDEX buildingIdIndex ON Buildings (id);



There was no noticeable change compared to normal. I tried creating an index for building id because I thought selecting an attribute that was near an already automatically indexed attribute would have more effect. However, Buildings is the one that is being joined, so it is added incrementally while the machine has full access to UnitOf because it is already called previously. Being able to search for UnitOf quickly does more than being able to search through an already ordered join of buildings.

#test index 3

CREATE INDEX ordercreationdate ON Ratings(creationDate);



This didn’t give much change either. I thought giving creationDate an index would make it faster to find a latest creationDate. However, it might only save time on finding the valid creationDate, while the automatic indexing by ratedBy saved time when searching for ratings rated by the same user.

```
# stored procedure, user-side search address. It will take a string
of characters and, using the `LIKE` operator, return the IDs of rooms
that are in buildings with the address. Operations are to JOIN Units
with Buildings on Buildings.id, and then select the rooms with
condition `WHERE Address LIKE @inputAddress`. Maybe we can use a
cursor for this when the amount of rooms to look for is too big.

CREATE PROCEDURE UnitSearchAddress() @inputAddress
BEGIN
SELECT u.id
FROM Units u JOIN Buildings b ON u.UnitOf = b.id
WHERE b.address LIKE (!!!!!!!!!!!!!!!!!how do I make it look for
elements?)@inputAddress AND NOT EXISTS (SELECT * FROM Tracking t
WHERE t.trackedUnit = u.id AND trackedUnit.accepted > 0)
END


# stored procedure, create rating.


CREATE PROCEDURE createRating(@inputBuildingId,
@inputTravelTimeMinutes, @inputTravelRating, @inputNoiseRating,
@inputAirRating, @inputFurnishingRating, @inputSpaceRating,
@currentDate)
BEGIN

INSERT INTO Ratings(

END



# stored procedure, average ratings. Averages the non-null ratings
for each ratings item. The conditions to be processed is that it
should be the most recent rating that the user did for the building,
and the rating is less than 3 years old. Maybe I should have it as a
cursor as well?

CREATE PROCEDURE BuildingAverageRatings() @inputAddress
BEGIN
SELECT AVG(travelTimeMinutes), AVG(travelRating), AVG(noiseRating),
AVG(airRating), AVG(furnishingRating), AVG(spaceRating),
AVG(creationDate)
FROM  Ratings r JOIN Buildings b ON r.ratedBuilding = b.id
```

```
WHERE b.address LIKE ********** AND r.creationDate >= ALL (SELECT
creationDate FROM Ratings r1 WHERE r.ratedBy = r1.ratedBy)
GROUP BY b.id
END
```