PT 1 STAGE 3 CS 411 Document

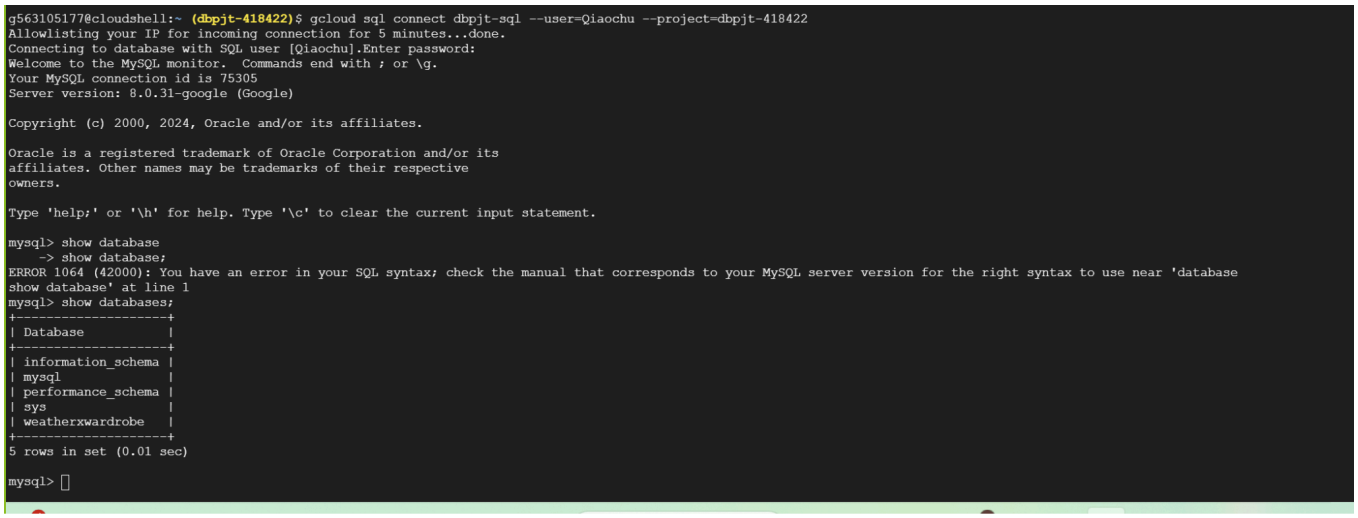


Table rows count:

```
mysql> SELECT COUNT(*) FROM location;
+----------+
| COUNT(*) |
+----------+
|     1009 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM user;
+----------+
| COUNT(*) |
+----------+
|     1466 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM weather;
+----------+
| COUNT(*) |
+----------+
|     3027 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM apparel;
+----------+
| COUNT(*) |
+----------+
|      306 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM usersubscribeslist;
+----------+
| COUNT(*) |
+----------+
|     3484 |
+----------+
1 row in set (0.01 sec)
```

1)Create a markdown or pdf file called “Database Design” in the doc folder of your GitHub repo.

In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you used to create each of these tables in the database. Here’s the syntax of the CREATE TABLE DDL command:

```
create table user(
userid INT PRIMARY KEY,
username VARCHAR(50),
gender VARCHAR(10));

create table location (
CityID INT PRIMARY KEY,
CityName VARCHAR(50),
State VARCHAR(20),
Latitude REAL,
Longitude REAL);

create table weather (
date DATETIME, cityID INT PRIMARY KEY,
Min_temp INT, Max_temp INT, Avg_temp INT,
Wind_speed real,
Category VARCHAR(20),
FOREIGN KEY(cityID) REFERENCES location (cityID));

Create table apparel (
ApparelID INT PRIMARY KEY,
product_name VARCHAR(50),
category VARCHAR(50),
Temp_from INT,
Temp_to INT
);

create table outfit(
outfit_id INTEGER PRIMARY KEY AUTO_INCREMENT,
Season Varchar(10),
Datecreated DATETIME,
user_id INTEGER,
apparelID INTEGER,
FOREIGN KEY(user_id) REFERENCES user(userid),
FOREIGN KEY(apparelID) REFERENCES apparel(ApparelId)
);


CREATE TABLE usersubscribeslist (
user_id INTEGER,
 date DATETIME,
 city_id INTEGER,
  FOREIGN KEY(city_id)REFERENCES weather(cityId),
FOREIGN KEY (user_id) REFERENCES user(userid),
```

```
  FOREIGN KEY (date) REFERENCES weather (date),
  PRIMARY KEY(user_id,date, city_id) )
```

For outfit table the records are entered using this stored procedure whenever a user requests for a new outfit for that particular weather:
Stored procedure for outfit:

```
CREATE procedure OutfitCreate(IN season VARCHAR(10), IN datecreated DATETIME, IN
user_id2 INTEGER)

   BEGIN
      DECLARE int outfit_id 0;
      DECLARE apparelcur CURSOR FOR select a.apparelID From apparel a
JOIN( select w.Avg_temp FROM weather w JOIN location l ON w.cityID =l.cityID JOIN
usersubscribeslist u ON l.cityID = u.city_ID)
AS current_weather
WHere a.Temp_from <=current_weather.Avg_temp AND a.Temp_to >=
current_weather.Avg_temp where u.user_id=user_id2;
   SELECT count(*)+1 into outfit_id from outfit;
      OPEN apparelcur;

      REPEAT
         FETCH apparelcur INTO apparelIDval;
         INSERT IGNORE INTO outfit VALUES(outfit_id,season,datecreated,user_id2,
apparelIDval);
      UNTIL done
      END REPEAT;


   END;
```

So it is empty at this point. We will use this procedure along with the front end to generate it whenever a user requests for it.

We had a separate table for outfit and the relationship outfit_has_apparel in stage 2.
We combined that relationship into the outfit table in the db in sql since it makes it easier to do queries on that table. Otherwise we would always do a join with outfit and outfit_has, only increasing complexity unnecessarily.

2) and 3) 4 complex queries and screenshots of top 15 results, and Indexing on the query:

a) **Printing all apparel appropriate for a user's list's location's weather**

```
SELECT a.category, a.product_name
FROM apparel a
JOIN (
    SELECT w.Avg_temp
    FROM weather w
    JOIN location l ON w.cityID = l.cityID
    JOIN usersubscribeslist u ON l.cityID = u.city_ID
)
AS current_weather
WHERE a.Temp_from <= current_weather.Avg_temp AND a.Temp_to >= current_weather.Avg_temp
```

category	product_name
Jacket	HOODED TECHNICAL JACKET
Jacket	HOODED TECHNICAL JACKET
Jacket	HOODED TECHNICAL JACKET
Jacket	LINEN - COTTON OVERSHIRT
Jacket	LINEN - COTTON OVERSHIRT
Jacket	LINEN - COTTON OVERSHIRT
Jacket	TECHNICAL WINDBREAKER JACKET
Jacket	TECHNICAL WINDBREAKER JACKET
Jacket	TECHNICAL WINDBREAKER JACKET
Jacket	TECHNICAL ANORAK WINDBREAKER
Jacket	TECHNICAL ANORAK WINDBREAKER
Jacket	TECHNICAL ANORAK WINDBREAKER
Jacket	REFLECTIVE EFFECT JACKET
Jacket	REFLECTIVE EFFECT JACKET
Jacket	REFLECTIVE EFFECT JACKET

Rows per page: 20 1 – 15 of 15

INDEXING:

Before:

```
| EXPLAIN
+-------------------------------------------------------------------------------
| -> Nested loop inner join  (cost=160662.52 rows=347169) (actual time=0.414..631.628 rows=502004 loops=1)
    -> Nested loop inner join  (cost=100681.79 rows=100544) (actual time=0.395..241.743 rows=144985 loops=1)
        -> Filter: ((a.Temp_from <= w.Avg_temp) and (a.Temp_to >= w.Avg_temp))  (cost=90543.37 rows=100544) (actual time=0.363..178.595 rows=144985 loops=1)
            -> Inner hash join (no condition)  (cost=90543.37 rows=100544) (actual time=0.347..89.189 rows=926262 loops=1)
                -> Table scan on w  (cost=0.13 rows=3027) (actual time=0.125..2.056 rows=3027 loops=1)
                -> Hash
                    -> Table scan on a  (cost=30.65 rows=299) (actual time=0.068..0.174 rows=306 loops=1)
        -> Single-row covering index lookup on l using PRIMARY (cityID=w.cityID)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=144985)
    -> Covering index lookup on u using city_id (city_id=w.cityID)  (cost=0.25 rows=3) (actual time=0.002..0.002 rows=3 loops=144985)
 |
```

This query has many records since it displays all apparel appropriate for a weather for all users and their subscribed locations. There are thousands of rows in each of these and there is apparel(in the order of 100) info for multiple locations(in the order of 1000) for all users(in the order of 1000s) This is used to show the impact of indices. In our real application we would do it user by user, but it still is a useful scenario to make the best index for our usecase.

<u>Index design 1</u>: apparel(category,product_name)

After:

```
create index cat_name_index on apparel (category,product_name);
```

```
----------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                                |
+----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=160662.52 rows=347169) (actual time=0.313..632.213 rows=502004 loops=1)
    -> Nested loop inner join  (cost=100681.79 rows=100544) (actual time=0.293..236.975 rows=144985 loops=1)
        -> Filter: ((a.Temp_from <= w.Avg_temp) and (a.Temp_to >= w.Avg_temp))  (cost=90543.37 rows=100544) (actual time=0.281..172.896 rows=144985 loops=1)
            -> Inner hash join (no condition)  (cost=90543.37 rows=100544) (actual time=0.274..85.229 rows=926262 loops=1)
                -> Table scan on w  (cost=0.13 rows=3027) (actual time=0.029..1.933 rows=3027 loops=1)
                -> Hash
                    -> Table scan on a  (cost=30.65 rows=299) (actual time=0.084..0.194 rows=306 loops=1)
        -> Single-row covering index lookup on l using PRIMARY (cityID=w.cityID)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=144985)
    -> Covering index lookup on u using city_id (city_id=w.cityID)  (cost=0.25 rows=3) (actual time=0.002..0.002 rows=3 loops=144985)
 |
```

Interpretation: doing an Index on an attribute in the SELECT part of the query doesn't have an impact on the cost of the query. The reason for this is that the select step is done at the end once all the records in the resultset have been fetched. This doesn't decrease the cost of the actual implementation of the query.

Index design 2: apparel(Temp_from):

```
Database changed
mysql> explain analyze Select a.category, a.product_name
    -> From apparel a
    -> JOIN( select w.Avg_temp FROM weather w JOIN location l ON w.cityID =l.cityID JOIN usersubscribeslist u ON l.cityID = u.city_ID)
    -> AS current_weather
    -> WHere a.Temp_from <=current_weather.Avg_temp AND a.Temp_to >= current_weather.Avg_temp;
+----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                          |
+----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=151650.74 rows=347169) (actual time=0.186..734.045 rows=502004 loops=1)
    -> Nested loop inner join  (cost=91670.01 rows=100544) (actual time=0.176..353.872 rows=144985 loops=1)
        -> Nested loop inner join  (cost=1161.85 rows=3027) (actual time=0.097..9.551 rows=3027 loops=1)
            -> Covering index scan on l using PRIMARY  (cost=102.40 rows=1009) (actual time=0.057..0.660 rows=1009 loops=1)
            -> Index lookup on w using cityID (cityID=l.cityID)  (cost=0.75 rows=3) (actual time=0.007..0.008 rows=3 loops=1009)
        -> Filter: ((a.Temp_from <= w.Avg_temp) and (a.Temp_to >= w.Avg_temp))  (cost=0.00 rows=33) (actual time=0.043..0.110 rows=48 loops=3027)
            -> Index range scan on a (re-planned for each iteration)  (cost=0.00 rows=299) (actual time=0.011..0.097 rows=85 loops=3027)
    -> Covering index lookup on u using city_id (city_id=l.cityID)  (cost=0.25 rows=3) (actual time=0.002..0.002 rows=3 loops=144985)
 |
```

Interpretation: Cost decreases massively since the range query used in the Where clause now has an index to efficiently perform the action in. A B-tree hash would allow us to get to the appropriate Temp_from in logn time which is a logarithmic decrease on the time to perform the 'Filter' action as opposed to no index. Then it does a linear search among the leaves that takes as long as the number of leaf nodes matching the condition + 1 (the first one higher than the condition).

Index design 3: apparel(Temp_from,Temp_to):

```
----------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                          |
+----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=151650.74 rows=347169) (actual time=0.125..750.134 rows=502004 loops=1)
    -> Nested loop inner join  (cost=91670.01 rows=100544) (actual time=0.118..374.170 rows=144985 loops=1)
        -> Nested loop inner join  (cost=1161.85 rows=3027) (actual time=0.059..10.548 rows=3027 loops=1)
            -> Covering index scan on l using PRIMARY  (cost=102.40 rows=1009) (actual time=0.029..0.748 rows=1009 loops=1)
            -> Index lookup on w using cityID (cityID=l.cityID)  (cost=0.75 rows=3) (actual time=0.008..0.009 rows=3 loops=1009)
        -> Filter: ((a.Temp_from <= w.Avg_temp) and (a.Temp_to >= w.Avg_temp))  (cost=0.00 rows=33) (actual time=0.048..0.116 rows=48 loops=3027)
            -> Index range scan on a (re-planned for each iteration)  (cost=0.00 rows=299) (actual time=0.014..0.102 rows=85 loops=3027)
    -> Covering index lookup on u using city_id (city_id=l.cityID)  (cost=0.25 rows=3) (actual time=0.001..0.002 rows=3 loops=144985)
```

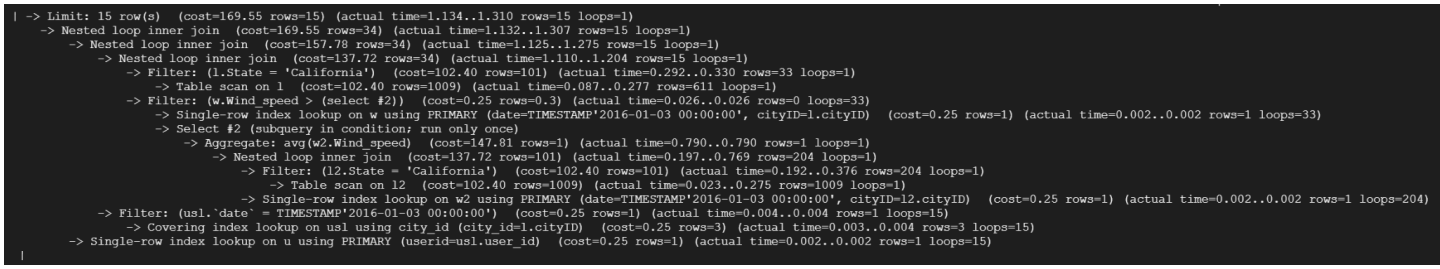
Interpretation:
Here the values are very similar to the previous one, showing that hashing the Temp_to doesnt really have a massive impact on the increase in performance as having just one of Temp_from and Temp_to in index . A composite index does take up more space and can't be reused as much since a suffix of the index cant be used in other queries to perform them efficiently. So here design 2 seems most efficient and usable.

b) print userid , username , gender,cityName of users who subscribe cities that has daily wind speed higher than the avg wind speed of California state in date of 2016-01-03

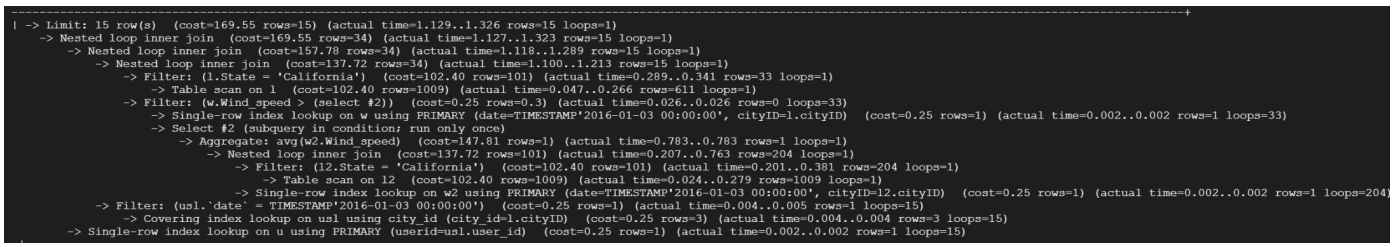
```
SELECT u.userid, u.username, u.gender, l.cityName
FROM user u
JOIN usersubscribeslist usl ON u.userid = usl.user_id
JOIN weather w ON usl.city_id = w.cityID AND usl.date = w.date
JOIN location l ON w.cityID = l.cityID
WHERE w.date = '2016-01-03'
AND w.Wind_speed > (
    SELECT AVG(w2.Wind_speed)
    FROM weather w2
    JOIN location l2 ON w2.cityID = l2.cityID
    WHERE l2.State = 'California'
    AND w2.date = '2016-01-03'
)
AND l.State = 'California';
```

userid	username	gender	cityName
965	cmutty	male	El Dorado Hills
611	lupusslugger	female	Davis
981	ThatBlondeWon	female	North Highlands
535	Steve_Mcteta	male	Citrus Heights
765	albertoarchulet	male	Brentwood
83	Beens03	female	Concord
962	TheRealJacoby	male	Ceres
370	Doctor_Math	male	Clovis
78	howugethegirl	female	Bakersfield
850	kolenda23	male	South Whittier
1000	bgdeuce13	male	Rowland Heights
966	GabsterSmitty	female	Cerritos
568	morgane28v	female	Alhambra
865	ShesUntamable	female	Arcadia
953	Haley_1D	female	Azusa

Initial result of default index:

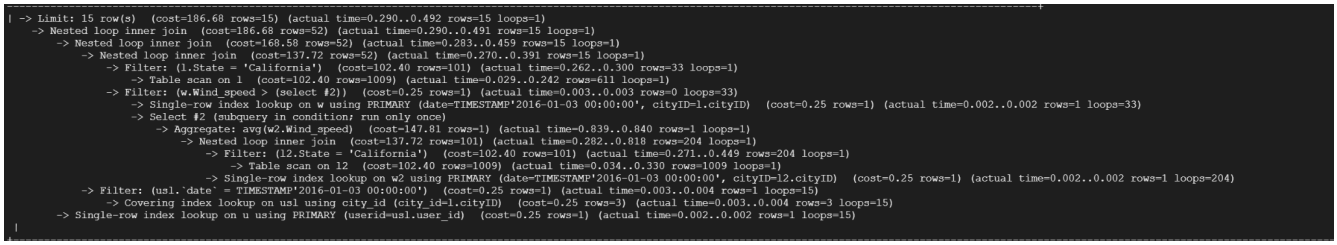


Index 1 : if create the index on user.username:



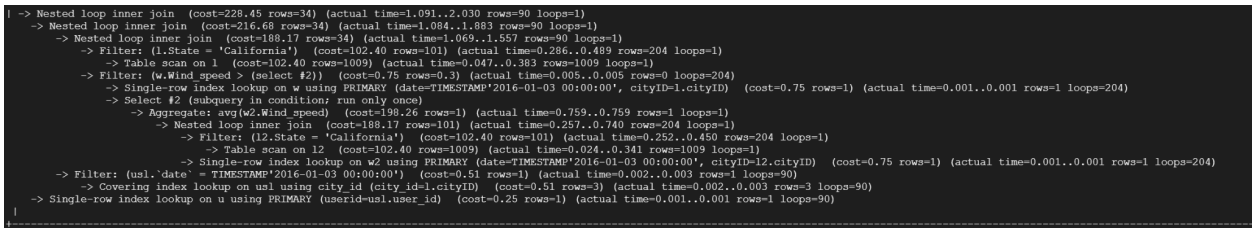
After adding the index, the result remains the same.This is probably caused by our query doesn't use username in order by or where clause and query does not leverage the indexed column in a way that benefits from the index.

Index 2: if create the index on weather.Wind_speed



After adding an index on wind_speed, the total cost increases. One possible reason for this is that the wind_speed index is not very selective (the threshold (SELECT avg(w2.Wind_speed)) does not filter out too many rows), then scanning the index is probably more costly than doing a full table scan, especially when our result has a lot of rows with wind speeds above the average.

Index design 3: if create the index on user.gender



After creating an index on gender, the total cost increases.This is probably because the gender values are not very selective (since many rows have the same values), the database may end up scanning a large portion of the index, which is more expensive than scanning the table directly.

c) Latitude, longitude and cities that have the most extreme weather (highest and lowest temperatures)

```
SELECT l.CityName, l.Latitude, l.Longitude, w.date, w.Max_temp, w.Min_temp
FROM weatherxwardrobe.weather w JOIN weatherxwardrobe.location l ON w.cityID = l.CityID JOIN (
  SELECT cityID, MAX(Max_temp) AS Max_temp, MIN(Min_temp) AS Min_temp
  FROM weatherxwardrobe.weather
  GROUP BY cityID)
extremes ON w.cityID = extremes.cityID AND (w.Max_temp = extremes.Max_temp OR w.Min_temp =
extremes.Min_temp)
ORDER BY l.CityName LIMIT 15
```

CityName	Latitude	Longitude	date	Max_temp	Min_temp
Abilene	32.4543	-99.7384	2016-01-04 00:00:00	55	44
Abilene	32.4543	-99.7384	2016-01-05 00:00:00	46	37
Aguadilla	18.4382	-67.1537	2016-01-05 00:00:00	58	51
Aguadilla	18.4382	-67.1537	2016-01-03 00:00:00	25	12
Akron	41.0798	-81.5219	2016-01-03 00:00:00	71	66
Akron	41.0798	-81.5219	2016-01-04 00:00:00	22	17
Alafaya	28.528	-81.1868	2016-01-04 00:00:00	56	43
Alafaya	28.528	-81.1868	2016-01-03 00:00:00	46	35
Alameda	37.7668	-122.267	2016-01-04 00:00:00	60	50
Alameda	37.7668	-122.267	2016-01-03 00:00:00	41	36
Albany	31.5776	-84.1762	2016-01-05 00:00:00	70	57
Albany	42.6664	-73.7987	2016-01-03 00:00:00	82	76
Albany	42.6664	-73.7987	2016-01-05 00:00:00	23	15
Albany	31.5776	-84.1762	2016-01-04 00:00:00	34	28
Albany	31.5776	-84.1762	2016-01-03 00:00:00	38	28

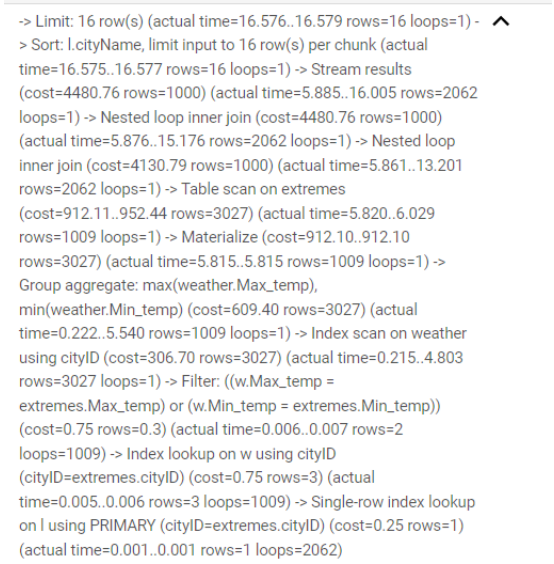
Rows per page: 20 1 – 15 of 15

Before Indexing:

EXPLAIN

-> Limit: 16 row(s) (actual time=15.129..15.132 rows=16 loops=1) -> Sort: l.cityName, limit input to 16 row(s) per chunk (actual time=15.128..15.130 rows=16 loops=1) -> Stream results (cost=4734.67 rows=1725) (actual time=4.764..14.545 rows=2062 loops=1) -> Nested loop inner join (cost=4734.67 rows=1725) (actual time=4.751..13.674 rows=2062 loops=1) -> Nested loop inner join (cost=4130.79 rows=1725) (actual time=4.735..11.818 rows=2062 loops=1) -> Table scan on extremes (cost=912.11..952.44 rows=3027) (actual time=4.683..4.890 rows=1009 loops=1) -> Materialize (cost=912.10..912.10 rows=3027) (actual time=4.678..4.678 rows=1009 loops=1) -> Group aggregate: max(weather.Max_temp), min(weather.Min_temp) (cost=609.40 rows=3027) (actual time=0.202..4.507 rows=1009 loops=1) -> Index scan on weather using cityID (cost=306.70 rows=3027) (actual time=0.195..3.918 rows=3027 loops=1) -> Filter: ((w.Max_temp = extremes.Max_temp) or (w.Min_temp = extremes.Min_temp)) (cost=0.75 rows=1) (actual time=0.006..0.007 rows=2 loops=1009) -> Index lookup on w using cityID (cityID=extremes.cityID) (cost=0.75 rows=3) (actual time=0.005..0.006 rows=3 loops=1009) -> Single-row index lookup on l using PRIMARY (cityID=extremes.cityID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2062)

Index 1: CREATE INDEX weather_index ON weather(Max_temp,Min_temp);

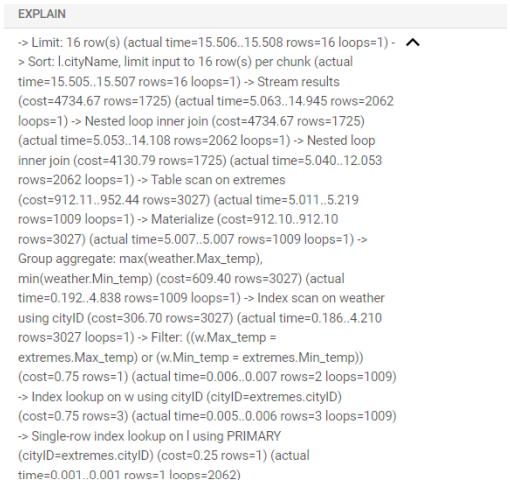


Indexing the values max_temp and min_temp from the weather table increased cost of query

Index 2:
CREATE INDEX date_index ON weather(date);

After indexing the date values from the weather table, the cost remains the same. This tells us that it takes the same amount of effort to perform the query.

Index 3:
CREATE INDEX latlong_index ON location(latitude, longitude);



After indexing the latitude and longitude values from the location table, the cost remains the same. This tells us that it takes the same amount of effort to perform the query.

Index 4:

CREATE INDEX city_index ON location(cityName);

After indexing the cityName values from location, the cost to perform the query remains the same.

d) Cities with temperature higher than the Average temperatures of all over a date range.

EXPLAIN ANALYZE QUERY FOR INDEXING:

```
EXPLAIN ANALYZE SELECT l.CityName, w.date, w.Max_temp, w.Wind_speed, w.Category
FROM weatherxwardrobe.weather w
JOIN weatherxwardrobe.location l ON w.cityID = l.CityID
WHERE w.Max_temp > (
    SELECT AVG(Max_temp)
    FROM weatherxwardrobe.weather
    WHERE date BETWEEN '2016-01-03 00:00:00' AND '2016-01-05 00:00:00'
    )
AND w.date IN (
    SELECT DISTINCT date
    FROM weatherxwardrobe.weather
    WHERE Max_temp > 30
    GROUP BY date
    HAVING COUNT(*) > 2
    )
ORDER BY w.date, l.CityName
LIMIT 15;
```

DEFAULT INDEXED RESULT:

-> Limit: 15 row(s) (actual time=13.062..13.065 rows=15 loops=1) -> Sort: w.`date`, l.cityName, limit input to 15 row(s) per chunk (actual time=13.061..13.063 rows=15 loops=1) -> Stream results (cost=458.00 rows=1009) (actual time=7.494..12.553 rows=1493 loops=1) -> Nested loop inner join (cost=458.00 rows=1009) (actual time=7.487..11.971 rows=1493 loops=1) -> Filter: ((w.Max_temp > (select #2)) and <in_optimizer>(w.`date`,w.`date` in (select #3))) (cost=104.89 rows=1009) (actual time=7.459..9.713 rows=1493 loops=1) -> Table scan on w (cost=104.89 rows=3027) (actual time=0.071..1.476 rows=3027 loops=1) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(weather.Max_temp) (cost=454.76 rows=1) (actual time=5.887..5.888 rows=1 loops=1) -> Filter: (weather.`date` between '2016-01-03 00:00:00' and '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.049..5.288 rows=3027 loops=1) -> Index range scan on weather using PRIMARY over ('2016-01-03 00:00:00' <= date <= '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.038..2.114 rows=3027 loops=1) -> Select #3 (subquery in condition; run only once) -> Filter: ((w.`date` = `<materialized_subquery>`.`date`)) (cost=508.58..508.58 rows=1) (actual time=0.362..0.362 rows=1 loops=4) -> Limit: 1 row(s) (cost=508.48..508.48 rows=1) (actual time=0.357..0.357 rows=1 loops=4) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (date=w.`date`) (actual time=0.356..0.356 rows=1 loops=4) -> Materialize with deduplication (cost=508.48..508.48 rows=1009) (actual time=1.405..1.405 rows=3 loops=1) -> Filter: (count(0) > 2) (cost=407.59 rows=1009) (actual time=0.485..1.384 rows=3 loops=1) -> Group aggregate: count(0) (cost=407.59 rows=1009) (actual time=0.484..1.381 rows=3 loops=1) -> Filter: (weather.Max_temp > 30) (cost=306.70 rows=1009) (actual time=0.043..1.210 rows=2564 loops=1) -> Index scan on weather using PRIMARY (cost=306.70 rows=3027) (actual time=0.040..0.938 rows=3027 loops=1) -> Single-row index lookup on l using PRIMARY (cityID=w.cityID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1493)

Toggle f

FIRST INDEXING METHOD:

CREATE INDEX weather_index ON weather(Max_temp,Min_temp);

RESULT:

Increases the cost for the query. Meaning, by indexing the Max_temp and Min_temp values from the weather table, it takes more effort to perform the query.\

DELETED INDEX

ALTER TABLE weather DROP INDEX weather_index;

Command used to delete the indexing method.

SECOND INDEXING METHOD:

CREATE INDEX location_index ON location(cityID);

Result:

-> Limit: 15 row(s) (actual time=8.812..8.815 rows=15 loops=1) -> Sort: w.`date`, l.cityName, limit input to 15 row(s) per chunk (actual time=8.811..8.813 rows=15 loops=1) -> Stream results (cost=458.00 rows=1009) (actual time=4.072..8.390 rows=1493 loops=1) -> Nested loop inner join (cost=458.00 rows=1009) (actual time=4.067..7.865 rows=1493 loops=1) -> Filter: ((w.Max_temp > (select #2)) and <in_optimizer>(w.`date`,w.`date` in (select #3))) (cost=104.89 rows=1009) (actual time=4.051..5.822 rows=1493 loops=1) -> Table scan on w (cost=104.89 rows=3027) (actual time=0.032..0.973 rows=3027 loops=1) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(weather.Max_temp) (cost=454.76 rows=1) (actual time=2.756..2.756 rows=1 loops=1) -> Filter: (weather.`date` between '2016-01-03 00:00:00' and '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.009..2.470 rows=3027 loops=1) -> Index range scan on weather using PRIMARY over ('2016-01-03 00:00:00' <= date <= '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.007..1.085 rows=3027 loops=1) -> Select #3 (subquery in condition; run only once) -> Filter: ((w.`date` = `<materialized_subquery>`.`date`)) (cost=508.58..508.58 rows=1) (actual time=0.312..0.312 rows=1 loops=4) -> Limit: 1 row(s) (cost=508.48..508.48 rows=1) (actual time=0.311..0.311 rows=1 loops=4) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (date=w.`date`) (actual time=0.311..0.311 rows=1 loops=4) -> Materialize with deduplication (cost=508.48..508.48 rows=1009) (actual time=1.235..1.235 rows=3 loops=1) -> Filter: (count(0) > 2) (cost=407.59 rows=1009) (actual time=0.419..1.226 rows=3 loops=1) -> Group aggregate: count(0) (cost=407.59 rows=1009) (actual time=0.417..1.224 rows=3 loops=1) -> Filter: (weather.Max_temp > 30) (cost=306.70 rows=1009) (actual time=0.028..1.072 rows=2564 loops=1) -> Index scan on weather using PRIMARY (cost=306.70 rows=3027) (actual time=0.026..0.833 rows=3027 loops=1) -> Single-row index lookup on l using PRIMARY (cityID=w.cityID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1493)

The cost is not changed by indexing the cityID value from the location table. This could mean that it does not change the performance to index the values from the location table for this query.

DELETED INDEX:

ALTER TABLE location DROP INDEX location_index;

Deleted second indexing method.

THIRD INDEXING METHOD:

CREATE INDEX final_index ON weather(Category,Wind_speed);

Result:

-> Limit: 15 row(s) (actual time=8.812..8.815 rows=15 loops=1) -> Sort: w.`date`, l.cityName, limit input to 15 row(s) per chunk (actual time=8.811..8.813 rows=15 loops=1) -> Stream results (cost=458.00 rows=1009) (actual time=4.072..8.390 rows=1493 loops=1) -> Nested loop inner join (cost=458.00 rows=1009) (actual time=4.067..7.865 rows=1493 loops=1) -> Filter: ((w.Max_temp > (select #2)) and <in_optimizer>(w.`date`,w.`date` in (select #3))) (cost=104.89 rows=1009) (actual time=4.051..5.822 rows=1493 loops=1) -> Table scan on w (cost=104.89 rows=3027) (actual time=0.032..0.973 rows=3027 loops=1) -> Select #2 (subquery in condition; run only once) -> Aggregate: avg(weather.Max_temp) (cost=454.76 rows=1) (actual time=2.756..2.756 rows=1 loops=1) -> Filter: (weather.`date` between '2016-01-03 00:00:00' and '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.009..2.470 rows=3027 loops=1) -> Index range scan on weather using PRIMARY over ('2016-01-03 00:00:00' <= date <= '2016-01-05 00:00:00') (cost=303.46 rows=1513) (actual time=0.007..1.085 rows=3027 loops=1) -> Select #3 (subquery in condition; run only once) -> Filter: ((w.`date` = `<materialized_subquery>`.`date`)) (cost=508.58..508.58 rows=1) (actual time=0.312..0.312 rows=1 loops=4) -> Limit: 1 row(s) (cost=508.48..508.48 rows=1) (actual time=0.311..0.311 rows=1 loops=4) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (date=w.`date`) (actual time=0.311..0.311 rows=1 loops=4) -> Materialize with deduplication (cost=508.48..508.48 rows=1009) (actual time=1.235..1.235 rows=3 loops=1) -> Filter: (count(0) > 2) (cost=407.59 rows=1009) (actual time=0.419..1.226 rows=3 loops=1) -> Group aggregate: count(0) (cost=407.59 rows=1009) (actual time=0.417..1.224 rows=3 loops=1) -> Filter: (weather.Max_temp > 30) (cost=306.70 rows=1009) (actual time=0.028..1.072 rows=2564 loops=1) -> Index scan on weather using PRIMARY (cost=306.70 rows=3027) (actual time=0.026..0.833 rows=3027 loops=1) -> Single-row index lookup on l using PRIMARY (cityID=w.cityID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1493)

Didn't change anything. This means that indexing the category and wind_speed data values from the weather table did not increase or decrease the efficiency of the query.
DELETED INDEXING:
ALTER TABLE weather DROP INDEX final_index;
Deleted the third indexing method.

D QUERY:

```
SELECT l.CityName, w.date, w.Max_temp
FROM weatherxwardrobe.weather w
JOIN weatherxwardrobe.location l ON w.cityID = l.CityID
WHERE w.Max_temp > (
    SELECT AVG(Max_temp)
    FROM weatherxwardrobe.weather
    WHERE date BETWEEN '2016-01-03 00:00:00' AND '2016-01-05 00:00:00'
    )
```

CityName	date	Max_temp
Akron	2016-01-03 00:00:00	71
Albany	2016-01-03 00:00:00	82
Albuquerque	2016-01-03 00:00:00	51
Alexandria	2016-01-03 00:00:00	52
Allen	2016-01-03 00:00:00	48
Aloha	2016-01-03 00:00:00	54
Alton	2016-01-03 00:00:00	63
Altoona	2016-01-03 00:00:00	55
Ankeny	2016-01-03 00:00:00	52
Anniston	2016-01-03 00:00:00	51
Antelope	2016-01-03 00:00:00	61
Apex	2016-01-03 00:00:00	62
Apopka	2016-01-03 00:00:00	52
Arvada	2016-01-03 00:00:00	54
Auburn	2016-01-03 00:00:00	49
Austin	2016-01-03 00:00:00	60

D QUERY RESULTS (TOP 15)

Changes for stage 2:

1.ER diagram remade completely:

Weather is a weak entity with a supporting relationship with Location since each weather is uniquely identified by the location of the weather. For eg. “Snowy” isn't unique since multiple locations can have the same snowy weather. So “Snowy”,”Urbana” is a unique combination.

Made user-list into a many to many relationship between user and weather, thus the schema remains the same but the relationship is more clearly defined.

Added more information to Location entity set.Now State is uniquely defined by City_Id and weather is uniquely defined by City_id and date.

2. Fixed inconsistencies between schema and ER diagram.