Stage 3 Database Design

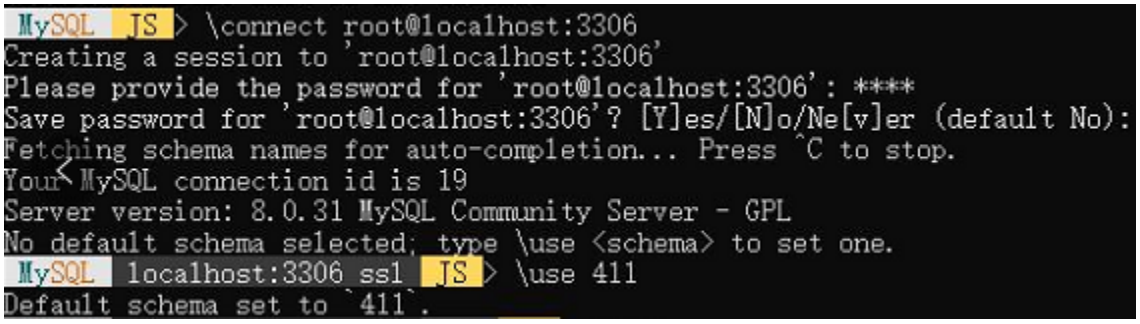
DDLs:

```
CREATE TABLE `comment` (
  `CommentId` int NOT NULL,
  `Text` varchar(1000) DEFAULT NULL,
  `Rate` int DEFAULT NULL,
  PRIMARY KEY (`CommentId`)
)

CREATE TABLE `dish` (
  `Name` varchar(255) NOT NULL,
  `Restaurant` varchar(255) NOT NULL,
  `Price` int DEFAULT NULL,
  PRIMARY KEY (`Restaurant`,`Name`),
  CONSTRAINT `dish_ibfk_1` FOREIGN KEY (`Restaurant`) REFERENCES `restaurant`
(`business_id`) ON DELETE CASCADE ON UPDATE CASCADE
)

CREATE TABLE `have` (
  `Name` varchar(255) DEFAULT NULL,
  `Restaurant` varchar(255) DEFAULT NULL,
  `CommentId` int DEFAULT NULL
)

CREATE TABLE `ingredients` (
  `name` varchar(255) NOT NULL,
  `manufacture` varchar(255) NOT NULL,
  `origin` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`name`,`manufacture`)
)

CREATE TABLE `madeby` (
  `name` varchar(255) DEFAULT NULL,
  `manufacture` varchar(255) DEFAULT NULL,
  `DishName` varchar(255) DEFAULT NULL
)
```

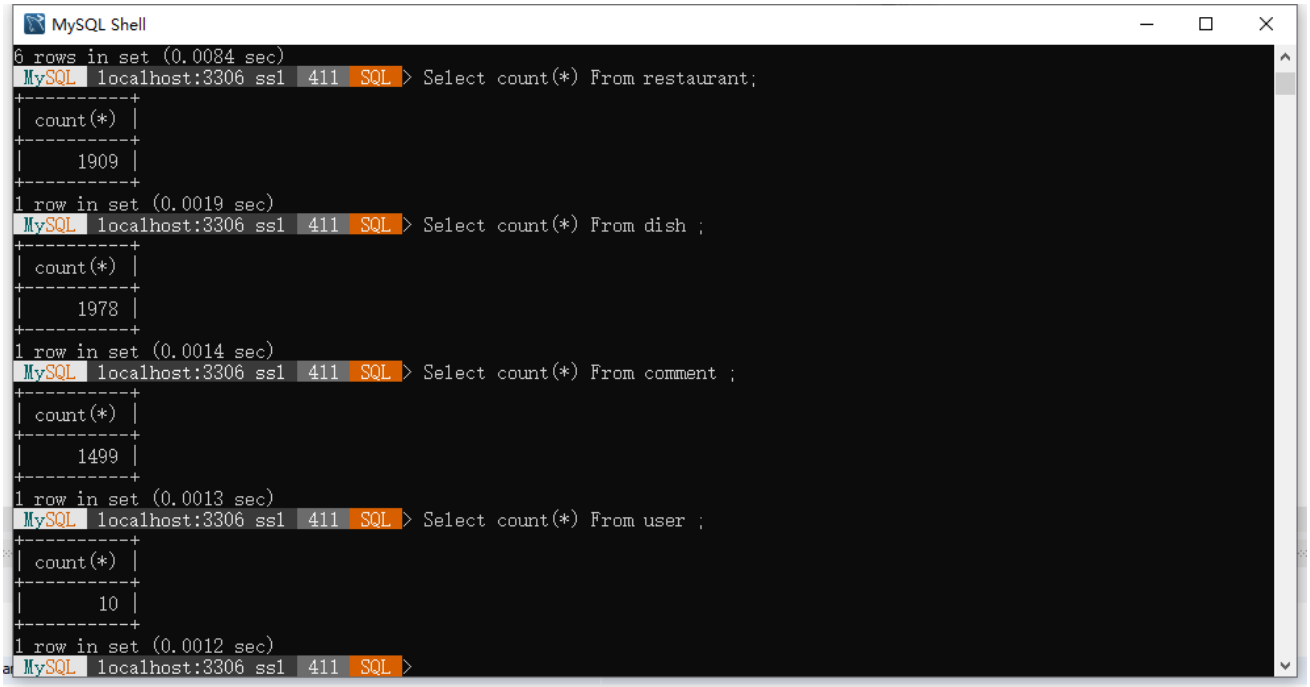
```
CREATE TABLE `restaurant` (
  `business_id` varchar(100) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `address` varchar(255) DEFAULT NULL,
  `city` varchar(255) DEFAULT NULL,
  `state` varchar(10) DEFAULT NULL,
  `postal_code` int DEFAULT NULL,
  `latitude` double DEFAULT NULL,
  `longitude` double DEFAULT NULL,
  `stars` double DEFAULT NULL,
  `categories` text,
  PRIMARY KEY (`business_id`)
)

CREATE TABLE `user` (
  `username` varchar(255) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL
)

CREATE TABLE `writtenby` (
  `username` varchar(255) DEFAULT NULL,
  `CommentID` int DEFAULT NULL
)
```

Note: For Restaurant, we have more attributes than we previously designed because the Yelp data contains these attributes. For users, we might add more attributes to make it consistent with the Yelp data.

DB connection:



Number of Rows in Tables that have contents:



Advance Queries:

Query 1:

```
Explain ANALYZE Select City, avg(Price)
From restaurant r Join dish d on d.restaurant = r.business_id
Where state = "FL" and stars > 3
Group by City
Order by avg(Price) DESC
LIMIT 15;
```

City	avg(Price)
Gulfport	55.0000
Holiday	55.0000
Palm Harbor	31.7500
Tampa Bay	30.7500
St. Pete Beach	30.2500
St Petersburg	28.3333
New Port Richey	22.5833
Tampa	22.0403
Riverview	21.3000
Saint Petersburg	21.0833
Wesley Chapel	19.5000
Spring Hill	19.5000
Greater Northdale	18.0000
Apollo Beach	18.0000
Temple Terrace	16.5000

Query 2:

```
Select City, count(*)
From restaurant Natural Join
        (Select business_id, avg(Price) as avgPrice
        From restaurant r Join dish d on d.restaurant = r.business_id
        Group by business_id) as temp
Where avgPrice > 20 or avgPrice < 10
Group by City
Order by count(*) DESC
LIMIT 15
```

City	count(*)
Philadelphia	55
Tampa	36
Indianapolis	23
New Orleans	21
Tucson	20
Nashville	19
Reno	9
Saint Louis	8
Saint Petersburg	7
Santa Barbara	7
St. Petersburg	6
Sparks	6
Metairie	6
St Petersburg	5
West Chester	5

Index Analysis:

Query1:

Default situation:

```
-> Limit: 15 row(s)  (actual time=2.691..2.693 rows=15 loops=1)
    -> Sort: avg(Price) DESC, limit input to 15 row(s) per chunk  (actual time=2.690..2.691 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.622..2.646 rows=38 loops=1)
            -> Aggregate using temporary table  (actual time=2.620..2.620 rows=38 loops=1)
                -> Nested loop inner join  (cost=262.53 rows=129) (actual time=0.052..2.353 rows=336 loops=1)
                    -> Filter: ((r.state = 'FL') and (r.stars > 3))  (cost=217.45 rows=64) (actual time=0.036..0.964
rows=297 loops=1)
                        -> Table scan on r  (cost=217.45 rows=1932) (actual time=0.029..0.727 rows=1909 loops=1)
                    -> Filter: (d.Restaurant = r.business_id)  (cost=0.50 rows=2) (actual time=0.004..0.004 rows=1
loops=297)
                        -> Index lookup on d using PRIMARY (Restaurant=r.business_id)  (cost=0.50 rows=2) (actual
time=0.003..0.004 rows=1 loops=297)
```

With an index on Restaurant.State:

```
-> Limit: 15 row(s)  (actual time=2.638..2.640 rows=15 loops=1)
    -> Sort: avg(Price) DESC, limit input to 15 row(s) per chunk  (actual time=2.637..2.638 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.600..2.607 rows=38 loops=1)
            -> Aggregate using temporary table  (actual time=2.598..2.598 rows=38 loops=1)
                -> Nested loop inner join  (cost=138.42 rows=225) (actual time=0.125..2.318 rows=336 loops=1)
                    -> Filter: (r.stars > 3)  (cost=59.57 rows=113) (actual time=0.105..0.869 rows=297 loops=1)
                        -> Index lookup on r using idx_state (state='FL')  (cost=59.57 rows=338) (actual time=0.102..0.826
rows=338 loops=1)
                    -> Filter: (d.Restaurant = r.business_id)  (cost=0.50 rows=2) (actual time=0.004..0.005 rows=1
loops=297)
                        -> Index lookup on d using PRIMARY (Restaurant=r.business_id)  (cost=0.50 rows=2) (actual
time=0.003..0.004 rows=1 loops=297)
```

For the Nested Inner Join, in the default setting, it costs 262.53. We thought a large portion of this cost was based on the first filter inside the Inner Join. This filter is affected by two attributes, Restaurants.Stars and Restaurants.State. So Here we tried to index Restaurants.State and the cost of that inner join were reduced to 138.42.

With an index on stars:

```
-> Limit: 15 row(s)  (actual time=2.960..2.962 rows=15 loops=1)
    -> Sort: avg(Price) DESC, limit input to 15 row(s) per chunk  (actual time=2.959..2.961 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.918..2.926 rows=38 loops=1)
            -> Aggregate using temporary table  (actual time=2.917..2.917 rows=38 loops=1)
                -> Nested loop inner join  (cost=327.56 rows=315) (actual time=0.044..2.610 rows=336 loops=1)
                    -> Filter: ((r.state = 'FL') and (r.stars > 3))  (cost=217.45 rows=157) (actual time=0.031..1.109
rows=297 loops=1)
                        -> Table scan on r  (cost=217.45 rows=1932) (actual time=0.025..0.831 rows=1909 loops=1)
                    -> Filter: (d.Restaurant = r.business_id)  (cost=0.50 rows=2) (actual time=0.004..0.005 rows=1
loops=297)
                        -> Index lookup on d using PRIMARY (Restaurant=r.business_id)  (cost=0.50 rows=2) (actual
time=0.004..0.004 rows=1 loops=297)
```

As explained above, here we tried to index on Restaurants.Stars. But there is no effect on the inner join. We assume that it is because the stars only range from 1-5, so the indexing did help in this case

With an index on restaurant.city:

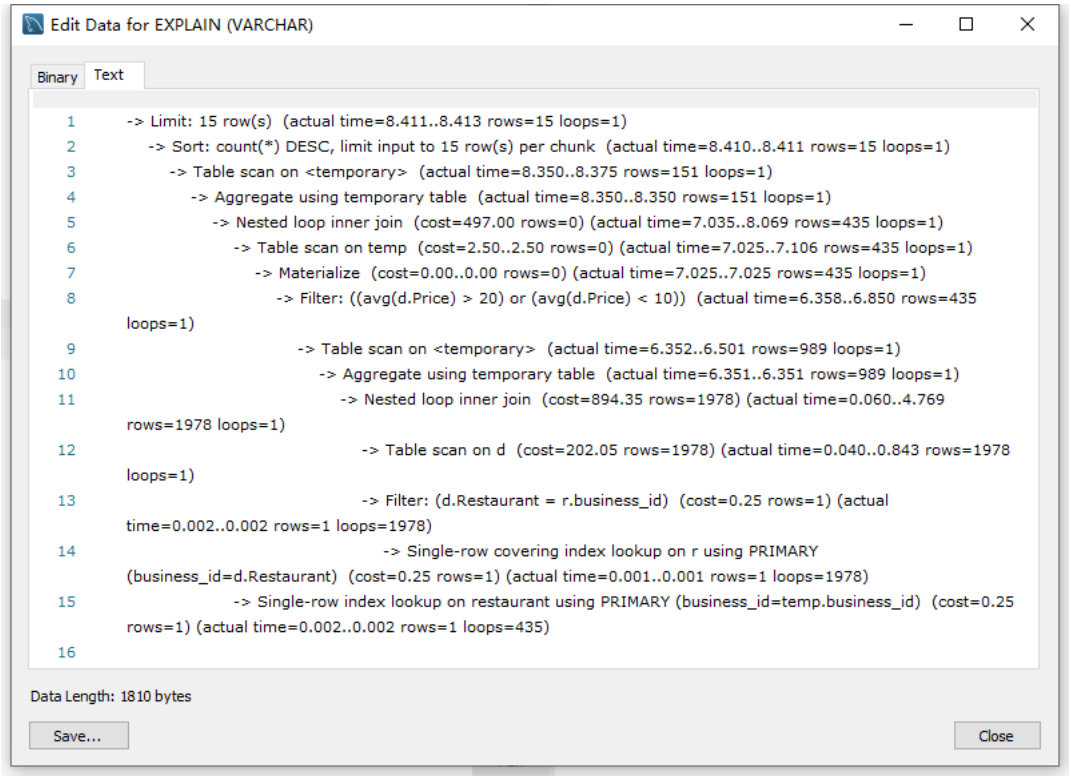
```
-> Limit: 15 row(s)  (actual time=2.921..2.923 rows=15 loops=1)
    -> Sort: avg(Price) DESC, limit input to 15 row(s) per chunk  (actual time=2.920..2.922 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.883..2.891 rows=38 loops=1)
            -> Aggregate using temporary table  (actual time=2.881..2.881 rows=38 loops=1)
                -> Nested loop inner join  (cost=262.53 rows=129) (actual time=0.046..2.585 rows=336 loops=1)
                    -> Filter: ((r.state = 'FL') and (r.stars > 3))  (cost=217.45 rows=64) (actual time=0.033..1.104
rows=297 loops=1)
                        -> Table scan on r  (cost=217.45 rows=1932) (actual time=0.025..0.819 rows=1909 loops=1)
                    -> Filter: (d.Restaurant = r.business_id)  (cost=0.50 rows=2) (actual time=0.004..0.005 rows=1
loops=297)
                        -> Index lookup on d using PRIMARY (Restaurant=r.business_id)  (cost=0.50 rows=2) (actual
time=0.004..0.004 rows=1 loops=297)
```

Based on the explain analyze result, there is no attribute we need to try since all the primary keys were indexed by default, so we decided to index on an attribute in the query. But there is not much effect on the results.

So, for this query, we conclude that adding an index to the city attribute performs the best,

Query2:

Default situation:



With index on restaurant.City:

-> Limit: 15 row(s) (actual time=8.386..8.388 rows=15 loops=1)
-> Sort: count(*) DESC, limit input to 15 row(s) per chunk (actual time=8.385..8.386 rows=15 loops=1)
-> Table scan on <temporary> (actual time=8.325..8.349 rows=151 loops=1)
-> Aggregate using temporary table (actual time=8.324..8.324 rows=151 loops=1)
-> Nested loop inner join (cost=497.00 rows=0) (actual time=6.999..8.037 rows=435 loops=1)
-> Table scan on temp (cost=2.50..2.50 rows=0) (actual time=6.990..7.066 rows=435 loops=1)
-> Materialize (cost=0.00..0.00 rows=0) (actual time=6.989..6.989 rows=435 loops=1)
-> Filter: ((avg(d.Price) > 20) or (avg(d.Price) < 10)) (actual time=6.321..6.812 rows=435 loops=1)
-> Table scan on <temporary> (actual time=6.315..6.464 rows=989 loops=1)
-> Aggregate using temporary table (actual time=6.313..6.313 rows=989 loops=1)
-> Nested loop inner join (cost=894.35 rows=1978) (actual time=0.065..4.756 rows=1978 loops=1)
-> Table scan on d (cost=202.05 rows=1978) (actual time=0.047..0.836 rows=1978 loops=1)
-> Filter: (d.Restaurant = r.business_id) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1978)
-> Single-row covering index lookup on r using PRIMARY (business_id=d.Restaurant) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1978)
-> Single-row index lookup on restaurant using PRIMARY (business_id=temp.business_id) (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=435)

Since we have a group by city in the query, we try on that attribute and it does not appear in the explain analyze results. But the results turned out to be not quite good. From here, we learned in our case indexing only the attribute on a group by won't work.

With index on dish.Price:

```
-> Limit: 15 row(s)  (actual time=10.354..10.356 rows=15 loops=1)
   -> Sort: count(*) DESC, limit input to 15 row(s) per chunk  (actual time=10.354..10.354 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=10.299..10.322 rows=151 loops=1)
         -> Aggregate using temporary table  (actual time=10.298..10.298 rows=151 loops=1)
            -> Nested loop inner join  (cost=497.00 rows=0) (actual time=8.999..10.015 rows=435 loops=1)
               -> Table scan on temp  (cost=2.50..2.50 rows=0) (actual time=8.988..9.071 rows=435 loops=1)
                  -> Materialize  (cost=0.00..0.00 rows=0) (actual time=8.988..8.988 rows=435 loops=1)
                     -> Filter: ((avg(d.Price) > 20) or (avg(d.Price) < 10))  (actual time=8.313..8.813 rows=435
loops=1)
                        -> Table scan on <temporary>  (actual time=8.308..8.464 rows=989 loops=1)
                           -> Aggregate using temporary table  (actual time=8.306..8.306 rows=989 loops=1)
                              -> Nested loop inner join  (cost=894.35 rows=1978) (actual time=0.057..6.507
rows=1978 loops=1)
                                 -> Covering index scan on d using idx_price  (cost=202.05 rows=1978) (actual
time=0.037..0.832 rows=1978 loops=1)
                                 -> Filter: (d.Restaurant = r.business_id)  (cost=0.25 rows=1) (actual
time=0.002..0.003 rows=1 loops=1978)
                                    -> Single-row covering index lookup on r using PRIMARY
(business_id=d.Restaurant)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1978)
               -> Single-row index lookup on restaurant using PRIMARY (business_id=temp.business_id)  (cost=0.25
rows=1) (actual time=0.002..0.002 rows=1 loops=435)
```

Since there is a filter affected by the dish.Price, we decided to index on that. However, it actually decreases the performance(actual time). We believe it is because the filter is based on average price instead of price, so indexing on this is redundant. And in this case, redundant indexing increases the computational time.

With index on dish.Price, restaurant.city:

```
-> Limit: 15 row(s)  (actual time=8.620..8.621 rows=15 loops=1)
   -> Sort: count(*) DESC, limit input to 15 row(s) per chunk  (actual time=8.619..8.620 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=8.574..8.593 rows=151 loops=1)
         -> Aggregate using temporary table  (actual time=8.574..8.574 rows=151 loops=1)
            -> Nested loop inner join  (cost=497.00 rows=0) (actual time=7.439..8.350 rows=435 loops=1)
               -> Table scan on temp  (cost=2.50..2.50 rows=0) (actual time=7.429..7.496 rows=435 loops=1)
                  -> Materialize  (cost=0.00..0.00 rows=0) (actual time=7.428..7.428 rows=435 loops=1)
                     -> Filter: ((avg(d.Price) > 20) or (avg(d.Price) < 10))  (actual time=6.898..7.294 rows=435
loops=1)
                        -> Table scan on <temporary>  (actual time=6.894..7.027 rows=989 loops=1)
                           -> Aggregate using temporary table  (actual time=6.892..6.892 rows=989 loops=1)
                              -> Nested loop inner join  (cost=894.35 rows=1978) (actual time=0.038..5.444
rows=1978 loops=1)
                                 -> Covering index scan on d using idx_price  (cost=202.05 rows=1978) (actual
time=0.024..0.604 rows=1978 loops=1)
                                 -> Filter: (d.Restaurant = r.business_id)  (cost=0.25 rows=1) (actual
time=0.002..0.002 rows=1 loops=1978)
                                    -> Single-row covering index lookup on r using PRIMARY
(business_id=d.Restaurant)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1978)
               -> Single-row index lookup on restaurant using PRIMARY (business_id=temp.business_id)  (cost=0.25
rows=1) (actual time=0.002..0.002 rows=1 loops=435)
```

Since only indexing the attribute in the group by won't work, we would like to test out if indexing it with dish.Price would help. But the results show that in this case, the indexing won't help as well. So for here, we believe that only attributes appear in Explain Analyze would affect the query cost.

So, for this query, we think the default query performs the best.