Raul G. Martinez (PID: A12461871)

MAS DSE 201 Homework: 201Cats

Milestone IV [due March 18 midnight]

Your next goal will be to improve the performance of the "My Kind of cats – with preference" option by appropriate precomputations. It is understood that the maintenance of the precomputed tables will lead to some slowdown while viewers interact. Your precomputation choices must be such that the precomputations that you introduce collectively save more time to this option than they cost to maintain. Precomputed tables will also benefit from creating certain indices on them. Build any beneficial indices. Calibrate your solution against cold runs of the activity script provided.

Submit the following.

- *The precomputed tables you created: CREATE TABLE statement and query that initially loaded it.*
- *Your new "My Kind of cats – with preference" query, which makes the best use of the precomputed tables you choose.*
- *The indices you created (CREATE INDEX scripts)*

201 Cats - Summary of Database Size

Table Name	Number of Tuples
users	10,000
sessions (not used in queries)	5
videos	25,000
friends	150,000
suggestions (not used in queries)	5
likes	100,000
watched (not used in queries)	5

201 Cats - Pre-Computed Tables Created

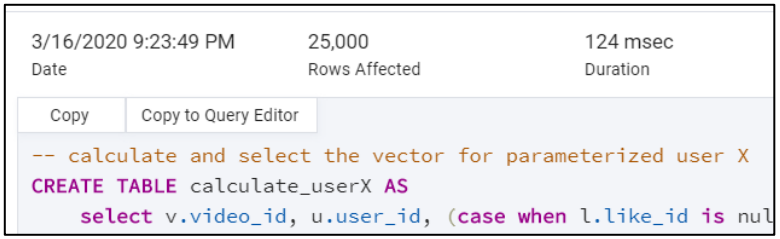
Approach	Table Name	Run Time (without Indices)	Run Time (with Indices)
#1	calculate_userX	124 msec	107 msec
	calculate_userY	10 min 40 secs	12 min 17 secs
#2	calculate_logcosine	4 min 38 secs	4 min 13 secs

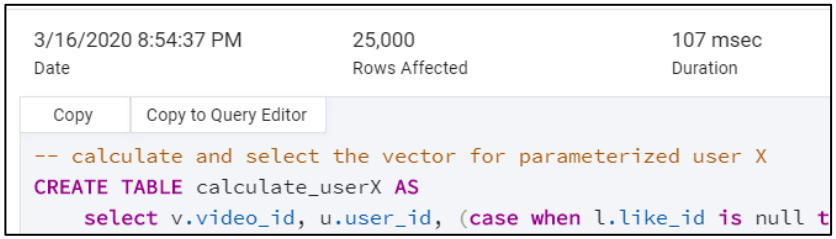
Screenshots for Approach #1

```
-- calculate and select the vector for parameterized user X
CREATE TABLE calculate_userX AS
    select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
    from videos v
    cross join users u
    left join likes l
    on u.user_id = l.user_id and  v.video_id = l.video_id
    where u.user_id = 3 -- specify user X
;
```

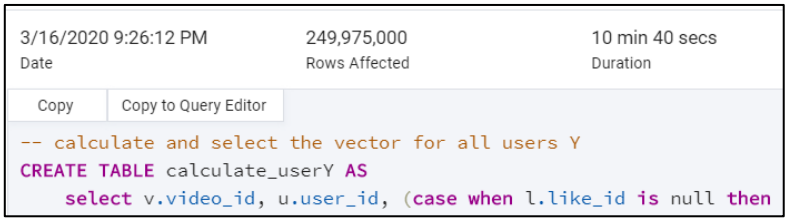
```
-- calculate and select the vector for all users Y
CREATE TABLE calculate_userY AS
    select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
    from videos v
    cross join users u
    left join likes l
    on u.user_id = l.user_id and  v.video_id = l.video_id
    where u.user_id != 3 -- exclude user X
;
```

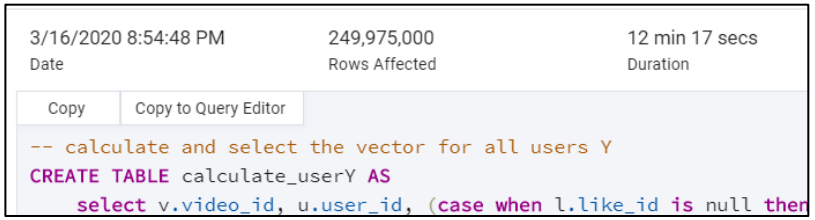
Run Time for *calculate_userX* - without Indices (left) and with indices (right)





Run Time for *calculate_userY* - without Indices (left) and with indices (right)

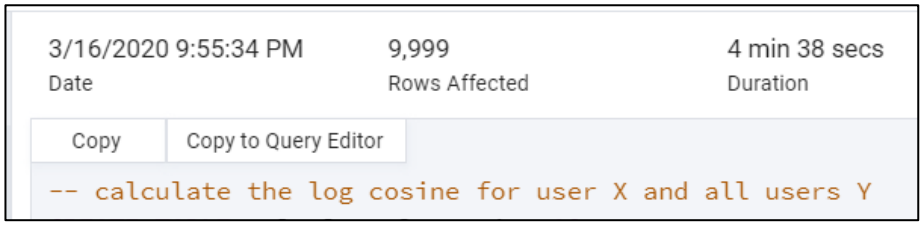


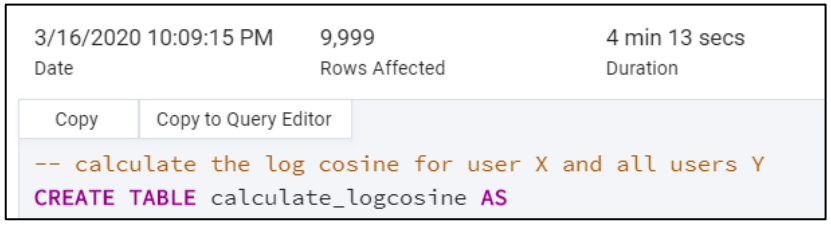


Screenshots for Approach #2

```sql
-- calculate the log cosine for user X and all users Y
CREATE TABLE calculate_logcosine AS
    -- calculate inner product and log cosine
    select user_y, log(sum(product)+1) as lc
    from(
        -- calculate and select multiplication from inner product
        select t1.video_id, t1.user_id as user_x, t2.user_id as user_y, t1.liked as liked_x, t2.liked
            as liked_y, (t1.liked * t2.liked) as product
        from (
            -- calculate and select the vector for parameterized user X
            select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
            from videos v
            cross join users u
            left join likes l
            on u.user_id = l.user_id and  v.video_id = l.video_id
            where u.user_id = 3 -- specify user X
        ) as t1
        left join (
            -- calculate and select the vector for all users Y
            select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
            from videos v
            cross join users u
            left join likes l
            on u.user_id = l.user_id and  v.video_id = l.video_id
            where u.user_id != 3 -- exclude user X
            ) as t2
        on t1.video_id = t2.video_id
        ) as t3
    group by user_y
;
```

Run Time for *calculate_logcosine* – without Indices (left) and with indices (right)





201 Cats - New "My Kind of cats – with preference" queries

Query Type	Run Time (with indices)	Run Time (without indices)
Original	4 min 19 secs	4 min 19 secs
With Pre-Computed Tables (Approach #1)	2 min 24 secs	2 min 23 secs
With Pre-Computed Tables (Approach #2)	361 msec	351 msec

Original

```
-- My kind of cats - with preference
select video_id, "Video Name", "Sum Weighted Likes"
from (
    -- selectect videos with ranked log cosine
    select l.video_id, v.name as "Video Name", cast(sum(t4.lc) as decimal(36,4)) as "Sum Weighted Likes",
        dense_rank() over(order by sum(t4.lc) desc) as rn
    from likes l
    join(
        -- calculate inner product and log cosine
        select user_y, log(sum(product)+1) as lc
        from(
            -- calculate and select multiplication from inner product
            select t1.video_id, t1.user_id as user_x, t2.user_id as user_y, t1.liked as liked_x, t2.liked
                as liked_y, (t1.liked * t2.liked) as product
            from (
                -- calculate and select the vector for parameterized user X
                select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
                from videos v
                cross join users u
                left join likes l
                on u.user_id = l.user_id and  v.video_id = l.video_id
                where u.user_id = 3 -- specify user X
                ) as t1
            left join (
                -- calculate and select the vector for all users Y
                select v.video_id, u.user_id, (case when l.like_id is null then 0 else 1 end) as liked
                from videos v
                cross join users u
                left join likes l
                on u.user_id = l.user_id and  v.video_id = l.video_id
                where u.user_id != 3 -- exclude user X
                ) as t2
            on t1.video_id = t2.video_id
            ) as t3
        group by user_y
        ) as t4
    on l.user_id = t4.user_y
    join videos v
    on l.video_id = v.video_id
    group by  l.video_id, v.name
) as t5
-- filter for top videos
where rn <= 10
;
```

3/16/2020 10:35:31 PM Date | 24,548 Rows Affected | 4 min 19 secs Duration

Copy | Copy to Query Editor

```
-- My kind of cats - with preference
```

Approach #1

```
-- Cats: New Query with Precomputed Tables
-- My kind of cats - with preference
select video_id, "Video Name", "Sum Weighted Likes"
from (
    -- selectect videos with ranked log cosine
    select l.video_id, v.name as "Video Name", cast(sum(t4.lc) as decimal(36,4)) as "Sum Weighted Likes",
        dense_rank() over(order by sum(t4.lc) desc) as rn
    from likes l
    join(
        -- calculate inner product and log cosine
        select user_y, log(sum(product)+1) as lc
        from(
            -- calculate and select multiplication from inner product
            select t1.video_id, t1.user_id as user_x, t2.user_id as user_y, t1.liked as liked_x, t2.liked
                as liked_y, (t1.liked * t2.liked) as product
            from calculate_userX t1 -- use precomputed table
            left join calculate_userY t2 -- use precomputed table
            on t1.video_id = t2.video_id
            ) as t3
        group by user_y
        ) as t4
    on l.user_id = t4.user_y
    join videos v
    on l.video_id = v.video_id
    group by  l.video_id, v.name
) as t5
-- filter for top videos
where rn <= 10
;
```

3/16/2020 10:40:57 PM Date | 24,548 Rows Affected | 2 min 24 secs Duration

Copy | Copy to Query Editor

```
-- Cats: New Query with Precomputed Tables
-- My kind of cats - with preference
```

Approach #2

```
-- Cats: New Query with Precomputed Tables
-- My kind of cats - with preference
select video_id, "Video Name", "Sum Weighted Likes"
from (
    -- selectect videos with ranked log cosine
    select l.video_id, v.name as "Video Name", cast(sum(t4.lc) as decimal(36,4)) as "Sum Weighted Likes",
        dense_rank() over(order by sum(t4.lc) desc) as rn
    from likes l
    join calculate_logcosine t4
    on l.user_id = t4.user_y
    join videos v
    on l.video_id = v.video_id
    group by  l.video_id, v.name
) as t5
-- filter for top videos
where rn <= 10
;
```

3/16/2020 10:48:00 PM Date | 24,548 Rows Affected | 361 msec Duration

Copy | Copy to Query Editor

```
-- Cats: New Query with Precomputed Tables
-- My kind of cats - with preference
```

201 Cats - Indices Created

Table	Column	Used In Clause	SQL Statement
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);
likes	video_id (Foreign Key)	LEFT JOIN JOIN GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);
likes	user_id (Foreign Key)	LEFT JOIN JOIN	CREATE INDEX likes_userid_idx ON likes(user_id);

201 Cats – Summary of Observations

- Two approaches were taken to implement pre-computed tables:
  - For approach #1: a table is created for each vector X and Y, where X is the user specified and Y are all the other users. This approach is useful because it takes care of all the expensive computations happening during the cross join between tables "users" and "likes", however, generating the pre-computed table "calculate_userY" requires a significant amount of memory and run time with 249,975,000 tuples and more than 10 minutes to execute (in this database tested).
  - For approach #2: a table is created to take care of similar computations mentioned on approach #1 but to also to handle joining the tables for vectors X and Y, plus performing the vector inner product and the log cosine calculation.
- Indices
  - On pre-computed tables: Indices seem to be benefiting tables "calculate_userX" and "calculate_logcosine", but affecting table "calculate_userY" from 10min 40sec to 12min 17sec in run time. As mentioned before, table "calculate_userY" is quite large and is possible that memory access and writing makes the run time slower.
  - On queries: Using indices to run the queries, for both pre-computed tables and without, does not reflect an impact in run time; on milestone 3 I previously showed the same result for the query without pre-computed tables. However, what I believe is interesting on this homework is that indices are probably not helping the query as a whole but when you create a pre-computed table on a smaller segment this one can reflect an improvement in run time.
- The Best Approach/Conclusion
  - Approach #2 is definitely performing better than #1. Creating the table "calculate_logcosine" will take less than half of the time needed to create "calculate_userY" alone (for the reasons mentioned previously). Another step that is very expensive on approach #1 is the left join between "calculate_userX" and "calculate_userY", considering that the latter table will always be quite large in reference to the former.
  - Approach #2 is also a better option than the original query, when comparing the best scenarios running it collectively (pre-computed table + query) takes less than 4 min 14 sec while the original query takes 4 min 19 sec.

MAS DSE 201 Homework: Sales Cube

Milestone IV [due March 18 midnight]

Your next goal will be to improve the performance of Query 6 by appropriate precomputations. It is understood that the maintenance of the precomputed table(s) will lead to some slowdown while viewers interact. Your precomputation choices must be such that the precomputations that you introduce collectively save more time to this option than they cost to maintain. Precomputed tables will also benefit from creating certain indices on them. Build any beneficial indices. Calibrate your solution against cold runs of the activity script.

Submit the following.

- *The precomputed tables you created: CREATE TABLE statement and query that initially loaded it.*
- *Your new Query 6, which makes the best use of the precomputed tables you chose.*
- *The indices you created (CREATE INDEX scripts)*

Sales Cube - Summary of Database Size

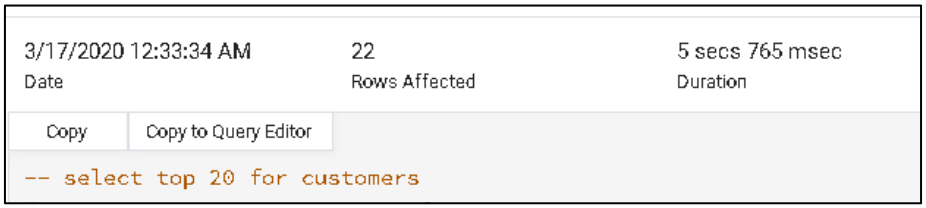
Table Name	Number of Tuples
states	5,000
customers	700,000
categories	900,000
products	800,000
sales	1,000,000

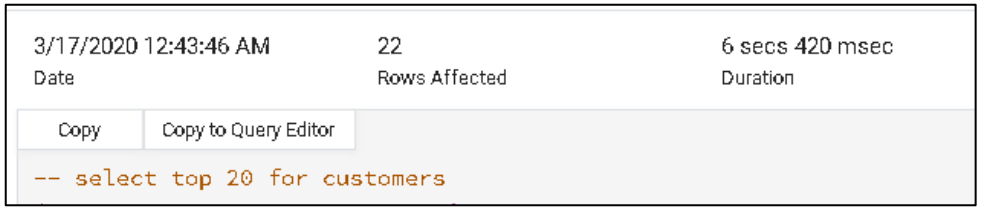
Sales Cube - Pre-Computed Tables Created

Table Name	Run Time (without Indices)	Run Time (with Indices)
top20_customers	5 secs 765 msec	6 secs 420 msec
top20_categories	4 secs 666 msec	4 secs 676 msec
sales_customerandcategories	8 secs 803 msec	6 secs 875 msec

```
-- select top 20 for customers
CREATE TABLE top20_customers AS
    select *
    from (
        select c.customer_id, c.first_name, c.last_name, dense_rank() over(order by sum(s.price_paid) desc) as rn
        from sales s
        join customers c
        on s.customer_id = c.customer_id
        group by c.customer_id
    ) as temp_t
    where rn <= 20
;
```

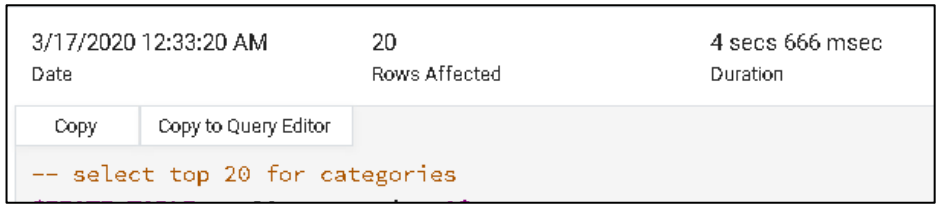
Run Time for *top20_customers* – without Indices (left) and with indices (right)

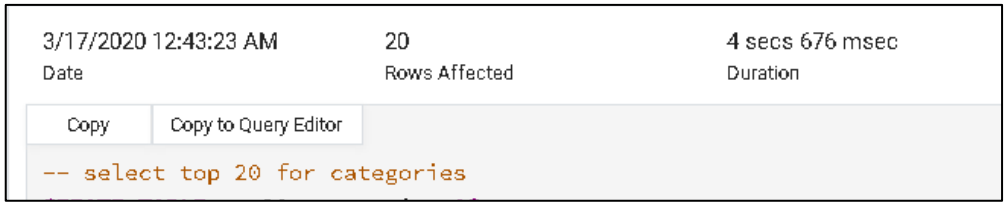




```
-- select top 20 for categories
CREATE TABLE top20_categories AS
    select *
    from (
        select cat.category_id, cat.name, dense_rank() over(order by sum(s.price_paid) desc) as rn
        from sales s
        join products p
        on s.product_id = p.product_id
        join categories cat
        on cat.category_id = p.category_id
        group by cat.category_id, cat.name
    ) as temp_t
    where rn <= 20
;
```

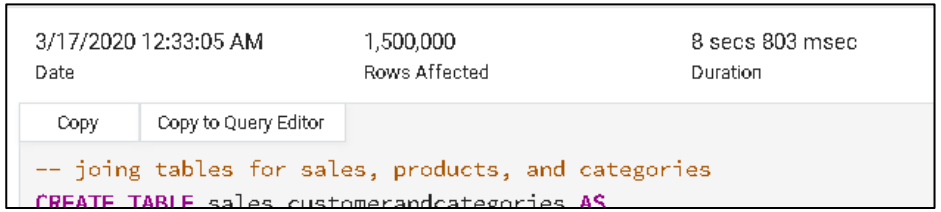
Run Time for *top20_categories* – without Indices (left) and with indices (right)

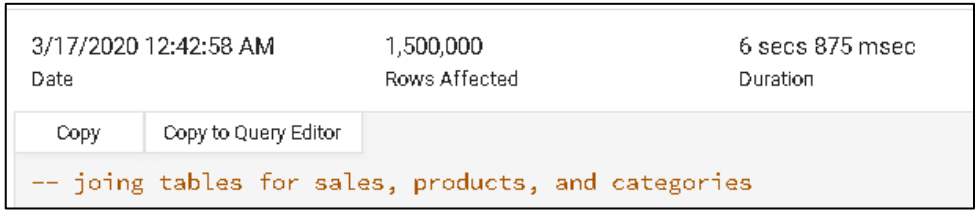





```
-- joing tables for sales, products, and categories
CREATE TABLE sales_customerandcategories AS
    select s.price_paid, s.customer_id, cat.category_id, s.quantity
    from sales s
    join products p
    on s.product_id = p.product_id
    join categories cat
    on cat.category_id = p.category_id
;
```

Run Time for *sales_customerandcategories* – without Indices (left) and with indices (right)



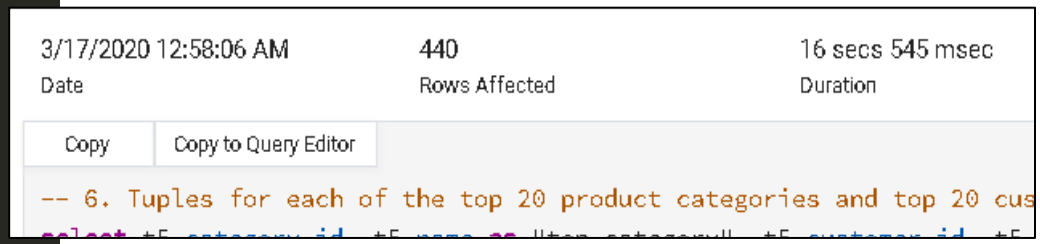


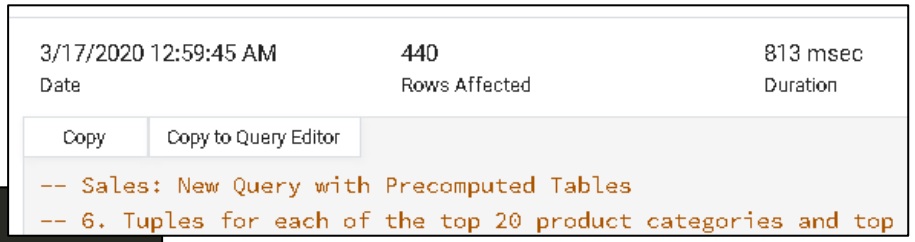
Sales Cube – Queries for #6

```
-- 6. Tuples for each of the top 20 product categories and top 20 customers
select t5.category_id, t5.name as "top category", t5.customer_id, t5.first_name
    as "top customer first name", t5.last_name as "top customer last name",
    coalesce(sum(s.quantity),0) as "quantity sold", coalesce(sum(s.price_paid),0) as "dollar value"
from sales s
join products p
on s.product_id = p.product_id
join categories cat
on cat.category_id = p.category_id
right join (
    select * from (
        -- select top 20 for categories
        select *
        from (
            select cat.category_id, cat.name, dense_rank() over(order by sum(s.price_paid) desc) as rn
            from sales s
            join products p
            on s.product_id = p.product_id
            join categories cat
            on cat.category_id = p.category_id
            group by cat.category_id, cat.name
            ) as t1
            where rn <= 20
            ) as t2
            cross join ( -- find all possible combinations ~400 with cross join
            -- select top 20 for customers
            select *
            from (
            select c.customer_id, c.first_name, c.last_name, dense_rank() over(order by sum(s.price_paid) desc) as rn
            from sales s
            join customers c
            on s.customer_id = c.customer_id
            group by c.customer_id
            ) as t3
            where rn <= 20
            ) as t4
        ) as t5
-- join on categories and customers only
on cat.category_id = t5.category_id and s.customer_id = t5.customer_id
group by t5.category_id, t5.name, t5.customer_id, t5.first_name, t5.last_name
order by "dollar value" desc
;
```

Original

Query Type	Run Time (with indices)	Run Time (without indices)
Original	16 secs 545 msec	14 secs 986 msec
With Pre-Computed Tables	813 msec	802 msec






```
-- Sales: New Query with Precomputed Tables
-- 6. Tuples for each of the top 20 product categories and top 20 customers
select t5.category_id, t5.name as "top category", t5.customer_id, t5.first_name
    as "top customer first name", t5.last_name as "top customer last name",
    coalesce(sum(s_cc.quantity),0) as "quantity sold", coalesce(sum(s_cc.price_paid),0) as "dollar value"
from sales_customerandcategories s_cc -- use precomputed table
right join (
    select *
    from top20_categories -- use precomputed table
    cross join top20_customers -- use precomputed table
) as t5
-- join on categories and customers only
on s_cc.category_id = t5.category_id and s_cc.customer_id = t5.customer_id
group by t5.category_id, t5.name, t5.customer_id, t5.first_name, t5.last_name
order by "dollar value" desc
;
```

With Pre-Computed Tables

Sales Cube - Indices Created

Table	Column	Used In Clause	SQL Statement
sales	customer_id (Foreign Key)	JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);
sales	product_id (Foreign Key)	JOIN	CREATE INDEX sales_productid_idx ON sales(product_id);
products	category_id (Foreign Key)	JOIN	CREATE INDEX products_categoryid_idx ON products(category_id);
categories	name	GROUP BY	CREATE INDEX categories_name_idx ON categories(name);

Sales Cube – Summary of Observations

- Pre-Computed Tables
  - “top20_customers” table: Finds the top 20 ranked customers based on the price paid. It computes the join between sales and customers, in addition to the rank. This is a nice summary for anyone that might be interested in who are the customers spending the most money in the organization's products or services.
  - “top20_categories” table: Finds the top 20 ranked categories based on the price paid. It computes the three joins among tables sales, products, and categories; in addition to the rank. This is also a nice summary for anyone that might be interested in maybe understanding what categories are bringing the most revenue or perhaps are the most popular among customers.
  - “sales_customerandcategories” table: Aggregates all the "sales" with the "product" and "categoties" table information. This pre-computed table seems very useful for users that query the "sales" table frequently but also want information coming from the "products" or "categories" tables, therefore, instead of joining all three tables in every single query (for each user) we can have an aggregation that becomes computationally cheaper in the long run.
- Indices
  - On Pre-Computed Tables: Indices seem to only be beneficial for table "sales_customerandcategories" by lowering the run time from 8secs 803msec to 6secs 875msec, while "top20_customers" decreased in performance by almost 1 second and "top20_categories" observed no change. Is likely that "sales_customerandcategories" table is benefiting the most from indices because the number of tuples (1,500,000) visited in memory is many orders of magnitude higher as compared to the other top 20 tables (with ~20 tuples).
  - On queries: There is no run time improvement when testing the queries with indices. The original query (no pre-computed tables) increased from 14secs 986msecs to 16secs 545msec when indices were applied, and the query with pre-computed tables also increased from 802msec to 813msec; the latter one is probably within variation (~10msec change).

- Conclusion
    - The cost to generate the query with pre-computed tables (including indices) is about 20 seconds, this is almost 4 seconds slower than the original query with a run time of 16secs 545msec. Given that pre-computed tables are giving worst performance, this is one of those cases where possible applications have to be considered for the tables generated, and how they bring utility to the end users. As mentioned above, someone may be particularly interested in just getting the top 20 customers or categories, or maybe query a single table to get all the sales information. All and all, if the needs of the data ecosystem consider more flexibility for data availability and cheaper performance over the long run it would make sense to implement pre-computed tables, but if optimizing the whole query is the main goal a better option would be to simply implement indices or perhaps to think about other possible optimization alternatives (i.e. re-writing the query or re-designing the schema).