Raul G. Martinez (PID: A12461871)

MAS DSE 201 Homework: 201Cats

Milestone III [due March 4 midnight]

Your next goal is to deliver top performance for the queries developed in Milestone II. Expect a significant amount of experimentation until you tune the performance.

In particular, create any indices that will be beneficially used by the queries. Do not create useless indices. Measure the performance of the queries on cold runs and report the performance improvements.

Submit your index choices (in the form of the script with the CREATE INDEX statements) and the measured query performance improvements.

Summary of Database Sizes

Table Name	**Number of Tuples (Large Dataset)**	**Number of Tuples (Small Dataset)**
users	10,000,000	10,000
sessions (not used in queries)	5	5
videos	2,500,000	25,000
friends	150,000,000	150,000
suggestions (not used in queries)	5	5
likes	100,000,000	100,000
watched (not used in queries)	5	5

Indices Automatically Generated with Database 201 Cats Schema

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	public	users	users_pkey	[null]	CREATE UNIQUE INDEX users_pkey ON public.users USING btree (user_id)
2	public	sessions	sessions_pkey	[null]	CREATE UNIQUE INDEX sessions_pkey ON public.sessions USING btree (session_id)
3	public	videos	videos_pkey	[null]	CREATE UNIQUE INDEX videos_pkey ON public.videos USING btree (video_id)
4	public	suggestions	suggestions_pkey	[null]	CREATE UNIQUE INDEX suggestions_pkey ON public.suggestions USING btree (suggestion_id)
5	public	likes	likes_pkey	[null]	CREATE UNIQUE INDEX likes_pkey ON public.likes USING btree (like_id)
6	public	likes	like_once	[null]	CREATE UNIQUE INDEX like_once ON public.likes USING btree (user_id, video_id)
7	public	watched	watched_pkey	[null]	CREATE UNIQUE INDEX watched_pkey ON public.watched USING btree (watch_id)

Index Choices

- Postgres creates indices by default for all primary keys in the data base. The indices below were created and tested in addition to the ones generated automatically (screenshot above).

```
-- friends table
CREATE INDEX friends_userid_idx
ON friends(user_id);

-- likes table
CREATE INDEX likes_videoid_idx
ON likes(video_id);

CREATE INDEX likes_userid_idx
ON likes(user_id);
```

Summary of Performance

- Table summary (queries were re-ran by adding all three index choices to the database)

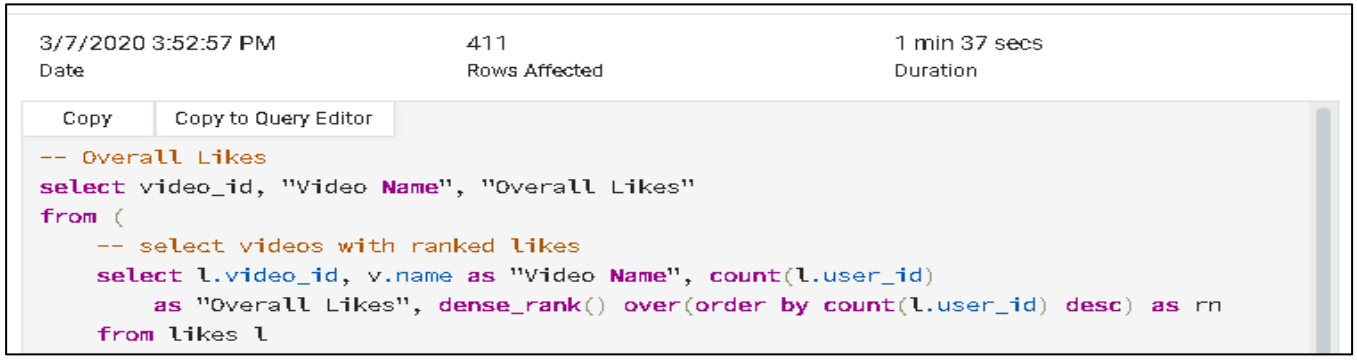
Query	Run Time without Index Choices	Run Time with Index Choices	Improved Performance?	Dataset Size
Overall Likes	1 min 37 secs	1 min 37 secs	No	Large
Friends Likes	6 secs 121 msec	45 msec	Yes	Large
Friends of Friends Likes	21 secs 463 msec	80 msec	Yes	Large
My Kind of Cats	5 secs 958 msec	106 msec	Yes	Large
My Kind of Cats with Preference	3 min 36 secs	3 min 52 secs	No	Small

- Comments:
  - Indexing seems to be useful on queries where a small fraction of all tuples needs to be accessed. For the queries below, the most benefit was observed when filtering (with WHERE clause) for a specific user_id and video_id tuple as a foreign key index. Using indices on other clauses such as DISTINCT, GROUP BY, JOIN, and ORDER BY was not as effective.

Overall Likes – Query (Large Dataset)

```sql
-- Overall Likes
select video_id, "Video Name", "Overall Likes"
from (
    -- select videos with ranked likes
    select l.video_id, v.name as "Video Name", count(l.user_id)
        as "Overall Likes", dense_rank() over(order by count(l.user_id) desc) as rn
    from likes l
    join videos v
    on l.video_id = v.video_id
    group by l.video_id, v.name
    ) as t
-- filter for top videos
where rn <= 10
;
```

Overall Likes - Total Query Runtime without Index Choices



Overall Likes – Observations

- No improvement in performance observed, run time without index choices is 1 min 37 secs.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);	1 min 42 secs	Inner join generates new tuples on the fly, is likely that the index is not useful for this reason.
likes	video_id (Foreign Key)	JOIN GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);	1 min 36 secs	There are 2.5 million videos on my videos table and all of them have at least one like, therefore, it seems that indexing the foreign key video_id is not benefiting the optimizer as it still must scan every entry.
likes	user_id (Foreign Key)	ORDER BY	CREATE INDEX likes_userid_idx ON likes(user_id);	1 min 37 secs	Is possible that the index is not valuable because new data is generated with the COUNT clause and then the WHERE clause is used on a new variable ‘rn’.

Friend Likes – Query (Large Dataset)

```
-- Friend Likes
select video_id, "Video Name", "Overall Likes"
from (
    -- select videos with ranked likes
    select l.video_id, v.name as "Video Name", count(l.user_id)
        as "Overall Likes", dense_rank() over(order by count(l.user_id) desc) as rn
    from likes l
    join videos v
    on l.video_id = v.video_id
    where user_id in (
        -- select list for friends
        select friend_id
        from friends
        where user_id = 3 -- specify user X
        )
    group by l.video_id, v.name
) as t
-- filter for top videos
where rn <= 10
;
```

Friend Likes - Total Query Runtime without Index Choices

Friend Likes – Observations

- Improvement in performance observed, run time without index choices is 6 secs 121 msec. Using an index on 'friends' table for 'user_id' column the query run time was reduced to 57 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
friends	user_id	WHERE	CREATE INDEX friends_userid_idx ON friends(user_id);	57 msec	This index yields a significant improvement in run time as the query specifies a filter for a single user X.
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);	6 secs 94 msec	Inner join generates new tuples on the fly, is likely that the index is not useful for this reason.
likes	video_id (Foreign Key)	JOIN GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);	6 secs 123 msec	There are 2.5 million videos on my videos table and all of them have at least one like, therefore, it seems that indexing the foreign key video_id is not benefiting the optimizer as it still must scan every entry.
likes	user_id (Foreign Key)	ORDER BY	CREATE INDEX likes_userid_idx ON likes(user_id);	6 secs 92 msec	Is possible that the index is not valuable because new data is generated with the COUNT clause and then the WHERE clause is used on a new variable 'rn'.

- Screenshot for improved query, friends - user_id

Friends of Friends Likes – Query (Large Dataset)

```
-- Friends of Friends Likes
select video_id, "Video Name", "Overall Likes"
from (
    -- select videos with ranked likes
    select l.video_id, v.name as "Video Name", count(l.user_id) as "Overall Likes",
        dense_rank() over(order by count(l.user_id) desc) as rn
    from likes l
    join videos v
    on l.video_id = v.video_id
    where user_id in (
        -- select list for friends
        select friend_id
        from friends
        where user_id = 3 -- specify user X
        union -- union both lists with user ids
        -- select list for friends of friends
        select friend_id
        from friends
        where user_id in (
            select friend_id
            from friends
            where user_id = 3 -- specify user X
        )
    )
    group by l.video_id, v.name
) as t
-- filter for top videos
where rn <= 10
;
```

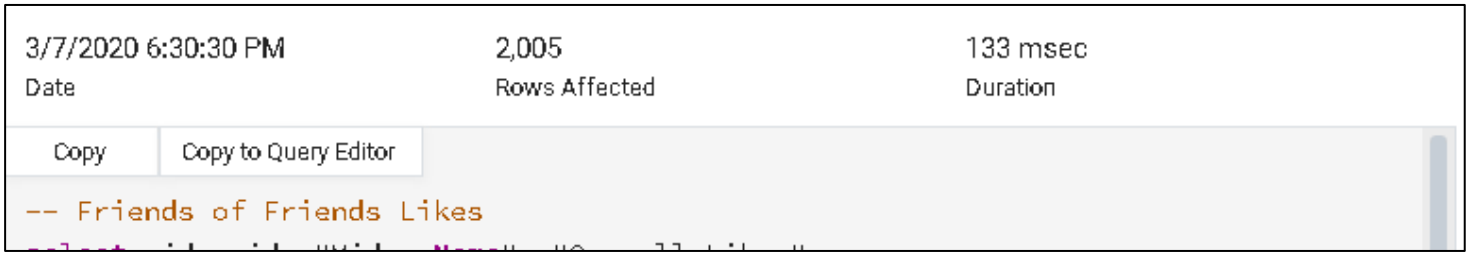
Friends of Friends Likes - Total Query Runtime without Index Choices

Friends of Friends Likes - Observations

- Improvement in performance observed, run time without index choices is 21 secs 463 msec. Using an index on 'friends' table for 'user_id' column the query run time was reduced to 133 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
friends	user_id	WHERE	CREATE INDEX friends_userid_idx ON friends(user_id);	133 msec	This index yields a significant improvement in run time as the query specifies a filter for a single user X.
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);	21 secs 595 msec	Inner join generates new tuples on the fly, is likely that the index is not useful for this reason.
likes	video_id (Foreign Key)	JOIN GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);	22 secs 147 msec	There are 2.5 million videos on my videos table and all of them have at least one like, therefore, it seems that indexing the foreign key video_id is not benefiting the optimizer as it still must scan every entry.
likes	user_id (Foreign Key)	ORDER BY	CREATE INDEX likes_userid_idx ON likes(user_id);	21 secs 603 msec	Is possible that the index is not valuable because new data is generated with the COUNT clause and then the WHERE clause is used on a new variable 'rn'.

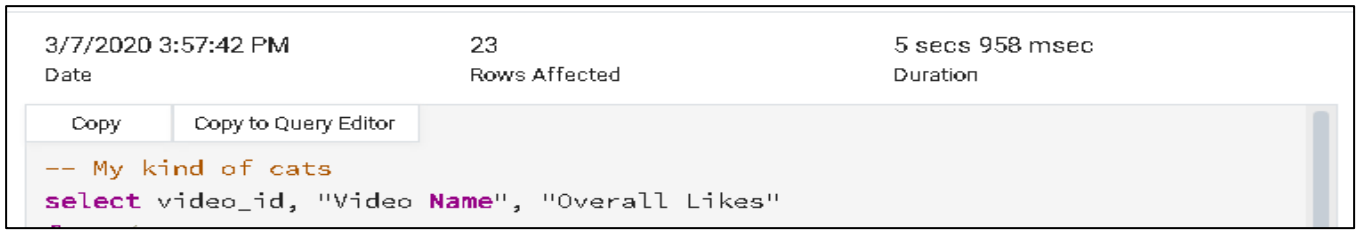
- Screenshot for improved query, friends - user_id



My Kind of Cats – Query (Large Dataset)

```
-- My kind of cats
select video_id, "Video Name", "Overall Likes"
from (
    -- select videos with ranked likes
    select l.video_id, v.name as "Video Name", count(l.user_id) as "Overall Likes",
        dense_rank() over(order by count(l.user_id) desc) as rn
    from likes l
    join videos v
    on l.video_id = v.video_id
    where user_id in (
        -- select users who have liked at least one cat video liked by X
        select  distinct user_id
        from likes
        where video_id in (
            -- select videos liked by user X
            select video_id
            from likes
            where user_id = 3 -- specify user X
            )
        )
        -- exclude videos liked by user X
        and user_id != 3 -- specify user X
    group by l.video_id, v.name
) as t
-- filter for top videos
where rn <= 10
;
```

My Kind of Cats - Total Query Runtime without Index Choices

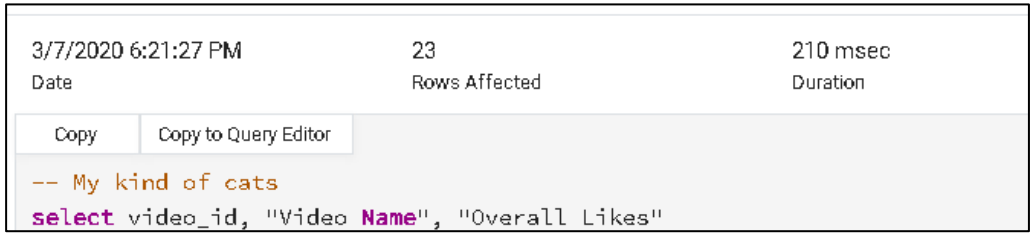


My Kind of Cats – Observations

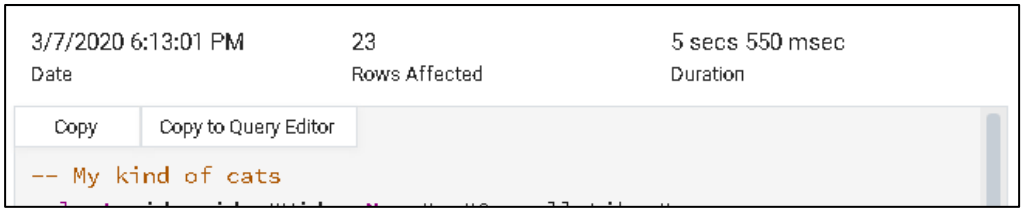
- Improvement in performance observed, run time without index choices is 5 secs 958 msec. Using an index on 'likes' table for 'video_id' column the query run time was reduced to 210 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);	5 secs 862 msec	Inner join generates new tuples on the fly, is likely that the index is not useful for this reason.
likes	video_id (Foreign Key)	JOIN WHERE GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);	210 msec	Significant performance improvement, the foreign key video_id is used as a filter in a sub-query to create a list.
likes	user_id (Foreign Key)	DISTINCT WHERE	CREATE INDEX likes_userid_idx ON likes(user_id);	5 secs 550 msec	There is a small improvement of about 300 msec, it likely happens in both sub-queries where the distinct user_id tuples are selected from the likes table, and on the filter where user X is specified.

- Screenshot for improved query, likes - video_id



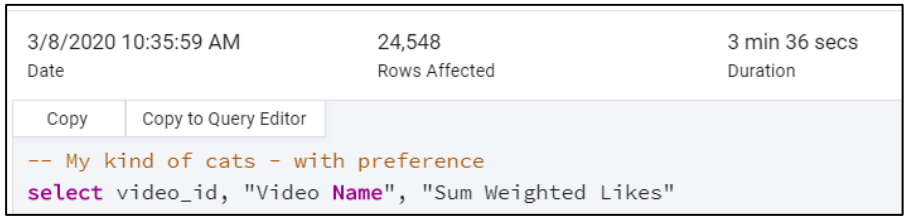
- Screenshot for improved query, likes - user_id



My Kind of Cats with Preference – Query (Small Dataset)

```
-- 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
;
```

My Kind of Cats with Preference - Total Query Runtime without Index Choices



My Kind of Cats with Preference - Observations

- No improvement in performance observed, run time without index choices is 3 min 36 secs.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
videos	name	GROUP BY	CREATE INDEX videos_name_idx ON videos(name);	3 min 42 secs	Inner join generates new tuples on the fly, is likely that the index is not useful for this reason.
likes	video_id (Foreign Key)	LEFT JOIN JOIN GROUP BY	CREATE INDEX likes_videoid_idx ON likes(video_id);	3 min 45 secs	Indexing is not helping joins because the optimizer still must visit a large percentage of entries to merge both tables.
likes	user_id (Foreign Key)	LEFT JOIN JOIN	CREATE INDEX likes_userid_idx ON likes(user_id);	3 min 51 secs	Indexing is not helping joins because the optimizer still must visit a large percentage of entries to merge both tables.

MAS DSE 201 Homework: Sales Cube

Milestone III [due March 4 midnight]

Your next goal is to deliver top performance for the queries. Expect a significant amount of experimentation until you tune the performance.

In particular, create any indices that will be beneficially used by the queries. Do not create useless indices. Deploy your solution and measure the performance of the Milestone II queries on cold runs.

Submit your index choices (in the form of the script with the CREATE INDEX statements) and the measured query performance improvement.

Summary of Database Sizes

Table Name	**Number of Tuples (Large Dataset)**	**Number of Tuples (Small Dataset)**
states	400	25
customers	320,000	2,000
categories	160,000	1,000
products	240,000	1,500
sales	80,000	5,000

Indices Automatically Generated with Database Sales Cube Schema

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	public	states	states_pkey	[null]	CREATE UNIQUE INDEX states_pkey ON public.states USING btree (state_id)
2	public	customers	customers_pkey	[null]	CREATE UNIQUE INDEX customers_pkey ON public.customers USING btree (customer_id)
3	public	categories	categories_pkey	[null]	CREATE UNIQUE INDEX categories_pkey ON public.categories USING btree (category_id)
4	public	products	products_pkey	[null]	CREATE UNIQUE INDEX products_pkey ON public.products USING btree (product_id)
5	public	sales	sales_pkey	[null]	CREATE UNIQUE INDEX sales_pkey ON public.sales USING btree (sale_id)

Index Choices

- Postgres creates indices by default for all primary keys in the data base (screenshot above).
- No additional indices were selected for the queries 1-6 below as there was not an observable improvement in performance. Although several different indices were tested, and their implementation is summarized in detail.
- Is still unclear to me whether performance differences will be evident with larger amounts of data (i.e. 10^6 tuples, something my computer was not able to generate), in this exercise I tried my best to pinpoint the areas where an index might be useful when running my queries.

Summary of Performance

- Table summary

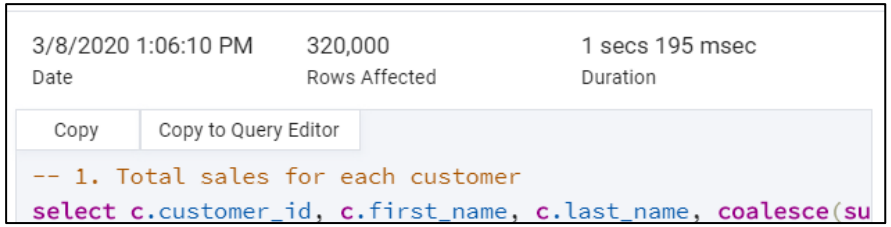
Query Number	Run Time without Index Choices	Run Time with Index Choices	Improved Performance?	Dataset Size
1	1 secs 195 msec	No Indices Chosen	No	Large
2	297 msec	No Indices Chosen	No	Large
3	82 msec	No Indices Chosen	No	Large
4	15 secs 774 msec	No Indices Chosen	No	Small
5	6 secs 117 msec	No Indices Chosen	No	Small
6	1 secs 106 msec	No Indices Chosen	No	Large

- Comments:
  - No significant improvement in performance was observed in any of the queries across the indices tested.

1 – Query (Large Dataset)

```
-- 1. Total sales for each customer
select c.customer_id, c.first_name, c.last_name, coalesce(sum(s.quantity),0)
    as "quantity sold", coalesce(sum(s.price_paid), cast(0 as decimal(36,2))) as "dollar value"
from customers c
left join sales s
on s.customer_id = c.customer_id
group by c.customer_id
order by "dollar value" desc
;
```

1 - Total Query Runtime without Index Choices



1 - Observations

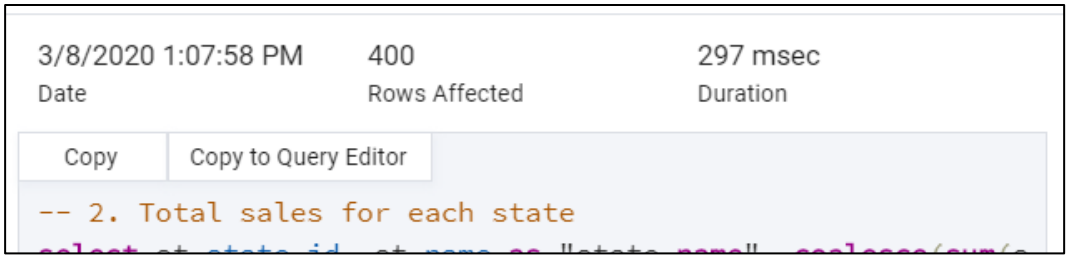
- No improvement in performance observed, run time without index choices is 1 secs 195 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_customer_id_idx ON sales(customer_id);	1 secs 177 msec	Is likely that no significant difference in performance is observed during left join because the optimizer still must visit a high percentage of tuples, therefore the index is not very useful.

2 – Query (Large Dataset)

```
-- 2. Total sales for each state
select st.state_id, st.name as "state name", coalesce(sum(s.quantity), 0)
    as "quantity sold", coalesce(sum(s.price_paid), cast(0 as decimal(36,2))) as "dollar value"
from sales s
join customers c
on s.customer_id = c.customer_id
right join states st
on c.state_id = st.state_id
group by st.name, st.state_id
order by "dollar value" desc
;
```

2 - Total Query Runtime without Index Choices



2 - Observations

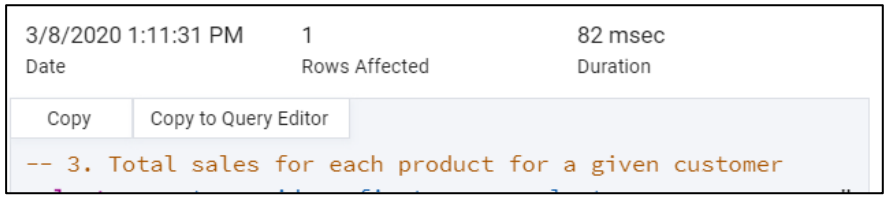
- No improvement in performance observed, run time without index choices is 297 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);	312 msec	Is likely that no improvement in performance is observed during left join because the optimizer still must visit a high percentage of tuples, therefore the index is not very useful.
customers	state_id (Foreign Key)	RIGHT JOIN	CREATE INDEX customers_stateid_idx ON customers(state_id);	305 msec	The right join is not benefited from indexing. Again, is probable that the query involves comparing most of the tuples in both tables.
states	name	GROUP BY	CREATE INDEX states_name_idx ON states(name);	284 msec	There is a very small, not sure if significant, improvement when indexing the string column ‘name’. We are also grouping by state_id which can possibly be another factor to consider during performance.

3 – Query (Large Dataset)

```
-- 3. Total sales for each product for a given customer
select c.customer_id, c.first_name, c.last_name, p.name as "product name",
    sum(s.quantity) as "quantity sold", sum(s.price_paid) as "dollar value"
from sales s
join customers c
on s.customer_id = c.customer_id
join products p
on s.product_id = p.product_id
-- Define customer and exclude zero entries for qty and price
where (c.customer_id = 25) and (s.quantity != 0) and (s.price_paid != 0)
group by p.name, c.customer_id
order by "dollar value" desc
;
```

3 - Total Query Runtime without Index Choices



3 - Observations

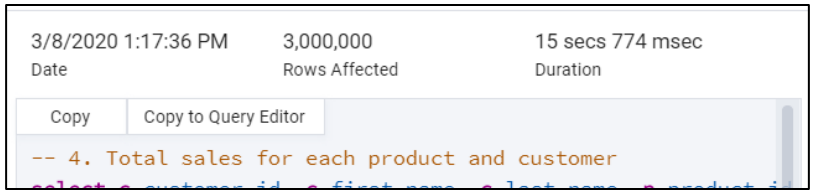
- No improvement in performance observed, run time without index choices is 82 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);	70 msec	There seems to be a very small improvement during the inner join with customer_id, although not convincing that is significant.
sales	product_id (Foreign Key)	JOIN	CREATE INDEX sales_productid_idx ON sales(product_id);	89 msec	Indexing product_id appears to not be optimizing the inner join. Interesting observation – there are more tuples in customers (320k) than products (240k) table, the inner join for customer_id (above) seems to be benefited more than product_id with indexing, is possible that this observation may scale up with more tuples.
products	name	GROUP BY	CREATE INDEX products_name_idx ON products(name);	72 msec	Slight reduction in run time by using an index on the string column 'name' from products table.

4 – Query (Small Dataset)

```
-- 4. Total sales for each product and customer
select c.customer_id, c.first_name, c.last_name, p.product_id, p.name
    as "product name", coalesce(sum(s.quantity),0) as "quantity sold",
    coalesce(sum(s.price_paid), cast(0 as decimal(36,2))) as "dollar value"
from products p
cross join customers c
left join sales s
on (s.product_id = p.product_id) and (c.customer_id = s.customer_id)
group by p.product_id, c.customer_id
order by "dollar value" desc
;
```

4 - Total Query Runtime without Index Choices



4 - Observations

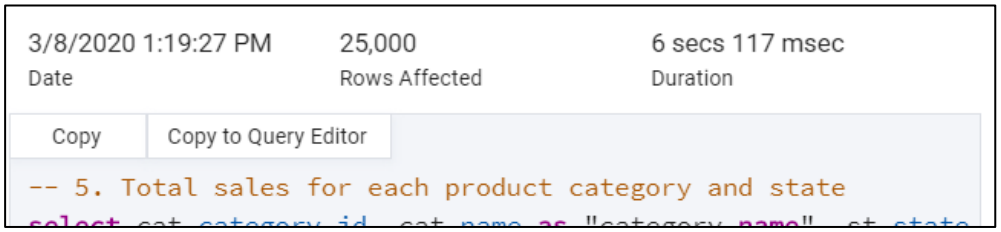
- No improvement in performance observed, run time without index choices is 15 secs 774 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);	15 secs 683 msec	There seems to be a very small improvement during the left join with customer_id, still not convinced this is significant.
sales	product_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_productid_idx ON sales(product_id);	15 secs 663 msec	There seems to be a very small improvement during the left join with product_id, still not convinced this is significant.

5 – Query (Small Dataset)

```
-- 5. Total sales for each product category and state
select cat.category_id, cat.name as "category name", st.state_id, st.name
    as "state name", coalesce(sum(s.quantity),0) as "quantity sold",
    coalesce(sum(s.price_paid), cast(0 as decimal(36,2))) as "dollar value"
from categories cat
cross join states st
left join products p
on p.category_id = cat.category_id
left join customers c
on st.state_id = c.state_id
left join sales s
on (s.product_id = p.product_id) and (c.customer_id = s.customer_id)
group by cat.category_id, st.state_id
order by "dollar value" desc
;
```

5 - Total Query Runtime without Index Choices



5 - Observations

- No improvement in performance observed, run time without index choices is 6 secs 117 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);	6 secs 115 msec	No performance improvement observed during left join. There is a possibility that more data is needed to make a conclusion, something in the order of 10^6 tuples.
sales	product_id (Foreign Key)	LEFT JOIN	CREATE INDEX sales_productid_idx ON sales(product_id);	6 secs 480 msec	No performance improvement observed during left join. There is a possibility that more data is needed to make a conclusion, something in the order of 10^6 tuples.
customers	state_id (Foreign Key)	LEFT JOIN	CREATE INDEX customers_stateid_idx ON customers(state_id);	6 secs 182 msec	No performance improvement observed during left join. There is a possibility that more data is needed to make a conclusion, something in the order of 10^6 tuples.
products	category_id (Foreign Key)	LEFT JOIN	CREATE INDEX products_categoryid_idx ON products(category_id);	6 secs 358 msec	No performance improvement observed during left join. There is a possibility that more data is needed to make a conclusion, something in the order of 10^6 tuples.

6 – Query (Large Dataset)

```sql
-- 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
;
```

6 - Total Query Runtime without Index Choices

6 - Observations

- No improvement in performance observed, run time without index choices is 1 secs 106 msec.
- Summary of indices attempted:

Table	Column	Used In Clause	SQL Statement	Run Time	Comments
sales	customer_id (Foreign Key)	JOIN	CREATE INDEX sales_customerid_idx ON sales(customer_id);	1 secs 132 msec	No performance improvement observed during join. Optimizer must be accessing a high percentage of tuples.
sales	product_id (Foreign Key)	JOIN	CREATE INDEX sales_productid_idx ON sales(product_id);	1 secs 188 msec	No performance improvement observed during join. Optimizer must be accessing a high percentage of tuples.
products	category_id (Foreign Key)	JOIN	CREATE INDEX products_categoryid_idx ON products(category_id);	1 secs 270 msec	No performance improvement observed during join. Optimizer must be accessing a high percentage of tuples.
categories	name	GROUP BY	CREATE INDEX categories_name_idx ON categories(name);	1 secs 193 msec	No performance improvement observed during group by. The other column ‘category_id’ in the same ‘group by’ clause has a primary key index, is probable that this one plays a factor during the aggregation.