**Milestone IV: Precomputed Tables**

Work flow

Professor’s solutions were used for performance testing. The ipython code for creating random data was used from Kyle (Cats) and Tony/Sanjay(Sales). For performance testing, queries were run 3 times with the average time taken as the performance metric, the % change was calculated using

[1 – (with index time/without index time)]\*100%

**Sales**

set search\_path to sales;

/\*\*\*

\* 6: For each one of the top 20 product categories and top 20 customers,

\* return a tuple (top product category, top customer, quantity sold, dollar value)

\*\*\*/

CREATE VIEW q1 AS

SELECT c.customer\_id,

coalesce (sum (s.quantity), 0) AS quantity\_sold,

coalesce (sum (s.quantity\*s.price), 0.0) AS dollar\_value

FROM sales.customer c LEFT JOIN sales.sale s ON c.customer\_id = s.customer\_id

GROUP BY c.customer\_id;

CREATE VIEW q4 AS

SELECT c.customer\_id, c.customer\_name, p.product\_id,

coalesce (SUM (s.quantity), 0) AS quantity\_sold,

coalesce (SUM (s.quantity\*s.price), 0.0) AS dollar\_value,

c.state\_id, p.category\_id

FROM (sales.customer c CROSS JOIN sales.product p) LEFT JOIN sales.sale s

ON c.customer\_id = s.customer\_id AND p.product\_id = s.product\_id

GROUP BY c.customer\_id, p.product\_id

ORDER BY c.customer\_id, dollar\_value DESC;

CREATE VIEW q5 AS

SELECT s.state\_id, c.category\_id,

coalesce (SUM (q.quantity\_sold), 0) AS quantity\_sold,

coalesce (SUM (q.dollar\_value), 0.0) AS dollar\_value

FROM (sales.state s CROSS JOIN sales.category c)

LEFT JOIN q4 q ON s.state\_id = q.state\_id AND c.category\_id = q.category\_id

GROUP BY s.state\_id, c.category\_id;

CREATE VIEW top\_customer\_values AS

SELECT DISTINCT dollar\_value

FROM q1

ORDER BY dollar\_value DESC

LIMIT 20;

CREATE VIEW all\_top\_customers AS

SELECT customer\_id

FROM q1

WHERE dollar\_value IN (SELECT dollar\_value FROM top\_customer\_values);

CREATE VIEW top\_category\_values AS

SELECT DISTINCT SUM (dollar\_value) AS dollar\_value

FROM q5

GROUP BY category\_id

ORDER BY dollar\_value DESC

LIMIT 20;

CREATE VIEW all\_top\_categories AS

SELECT category\_id

FROM q5

GROUP BY category\_id

HAVING SUM (dollar\_value) IN (SELECT dollar\_value FROM top\_category\_values);

CREATE MATERIALIZED VIEW q6\_all\_mat AS

SELECT ca.category\_id, cu.customer\_id,

coalesce (SUM (q.quantity\_sold), 0) AS quantity\_sold,

coalesce (SUM (q.dollar\_value), 0.0) AS dollar\_value

FROM (all\_top\_customers cu CROSS JOIN all\_top\_categories ca) LEFT JOIN q4 q

ON q.customer\_id = cu.customer\_id AND q.category\_id = ca.category\_id

GROUP BY ca.category\_id, cu.customer\_id;

SELECT \*

FROM q6\_all\_mat;

Reasoning

By creating and querying on the precomputed tables, the performance of the query increased by 25% relative to the cold run queries. A materialized view contains the results of the query and draws directly from all\_top\_customers, all\_top\_categories, and q4 tables. Materialized views are typically expensive to maintain, and if the user and interested in attaining only the top 20 products and top 20 customers, precomputing the above query would provide for the fastest relay of data. No indexes were added to the materialized view, since the materialized view creates the answer for the SELECT \* statement and the addition of any indexes would not increase performance time or decrease the performance cost of the query.

**Cats**

SET search\_path TO cats;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\* MY kind of cats

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE VIEW init (uid, vid, verdict) AS

select u.user\_id as uid, v.video\_id as vid, 0 as verdict

from cats.user u, cats.video v;

CREATE VIEW cats.mykindOfUser (user\_id, other\_id) AS

select distinct ul.user\_id, ol.user\_id as other\_id

from cats.likes ul, cats.likes ol

where ul.user\_id != ol.user\_id and

ul.video\_id = ol.video\_id;

CREATE VIEW cats.mykindLikes (uid, vid, verdict) AS

select u.user\_id as uid, l.video\_id as vid, 1 as verdict

from cats.user u, cats.mykindOfUser m, cats.likes l

where m.user\_id = u.user\_id and

l.user\_id = m.other\_id

union all

select \* from cats.init;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*weighted

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE VIEW cats.commonLikes (x,y,likeSame) AS

select l1.user\_id as x, l2.user\_id as y, 1 as likeSame

from cats.likes l1, cats.likes l2

where l1.video\_id = l2.video\_id and

l1.user\_id != l2.user\_id

union all

select u1.user\_id as x, u2.user\_id as y, 0 as likeSame

from cats.user u1, cats.user u2

where u1.user\_id != u2.user\_id;

CREATE VIEW cats.inner\_product (x,y,prod) AS

select x, y, sum (likeSame) as prod

from cats.commonLikes

group by x, y;

CREATE VIEW cats.weightedMykindLikes (uid, vid, verdict) AS

select u.user\_id as uid, l.video\_id as vid, log(1+i.prod) as verdict

from cats.user u, cats.inner\_product i, cats.likes l

where u.user\_id = i.x and l.user\_id = i.y;

SELECT \*

FROM cats.weightedMykindLikes;

-- same query, replacing overallLikes with weightedMykindLikes

CREATE MATERIALIZED VIEW cats.weightedMykindLikes\_mat AS

select vid, sum (verdict) as rank

from (

select u.user\_id as uid, l.video\_id as vid, log(1+i.prod) as verdict

from cats.user u, cats.inner\_product i, cats.likes l

where u.user\_id = i.x and l.user\_id = i.y) o

where o.uid = 13 and

not exists (select 1 from cats.watch w where w.user\_id = o.uid and w.video\_id = o.vid) and

not exists (select 1 from cats.likes l where l.user\_id = o.uid and l.video\_id = o.vid)

group by vid

order by rank desc

limit 10;

SELECT \*

FROM cats.weightedMykindLikes\_mat;

Reasoning

By creating and querying on the precomputed table, the performance of the query increased by 10% relative to the cold run queries. Note, if the data set size was larger the performance increase would be more notable, since data processing would rely on disk. The above query would provide the fastest relay of data because it would precompute the exact solution for the query after references from user, inner\_product, likes, and watch tables. No indexes were added to the materialized view, since the materialized view creates the answer for the SELECT \* statement and the addition of any indexes would not increase performance time or decrease the performance cost of the query.