Orysya Stus
3.8.2017

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
********************/
```

Orysya Stus
3.8.2017

```
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
```

Orysya Stus
3.8.2017

```
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.