SQL

Assignment

We have a **single table** below. This table sits behind a **web application** that allows our existing customers **subscribe** to use Autodesk software. That application also allows our existing customers to **update** customer info, like their name or address.

When a customer updates their contact phone number, **what query** should we run in order to save that update to the database?

```
create table
tblSubscriptionInfo
(
  subscription_id int
  product_id int
  product_name varchar
  subscription_start_date
datetime
  subscription_end_date
datetime
  customer_id int
  customer_contact_phone
varchar
  customer_name varchar
  customer_address varchar
)
```

We've noted that the phone number update feature in the web application is **too slow**, and have identified that the **update query is the primary bottleneck**. What could we do to speed up this query?

Come up with the queries to find:

- number of subscribers whose subscriptions will be ending in 2023;
- number of subscribers who have subscribed for more than 3 months in 2022;
- subscribers who have subscribed for more than two products;
- product with the most/2ndmost/3rdmost number of subscribers in 2022;
- number of subscribers who have re-subscribed more than once for each product;
- subscribers who have re-subscribed a higher version of the product in 2023 - for example Autocad 2022 to Autocad 2023.

Answers

1. When a customer **updates their contact phone number**, **what query** should we run in order to save that update to the database?

```
UPDATE tblSubscriptionInfo
SET customer_contact_phone = 'NEW_CONTACT_NUMBER'
WHERE customer_id = 'GIVEN_CUSTOMER_ID';
```

2. We've noted that the **phone number update feature** in the web application is **too slow**, and have identified that the **update query is the primary bottleneck**. **What could we do to speed up this query**?

- Assumptions
    - customer_id is the unique identifier of a customer.

- Firstly, based on the CREATE TABLE statement, it seems that the table does not have a primary key. This suggests that every time an expensive **whole table scan** is required whenever an UPDATE query is executed.
    - An index can be built on the 'customer_id' column to speed up the searching of the row to be updated in the UPDATE query.
    - An example of building an index on the 'customer_id' column is presented below:

```
CREATE INDEX customer_index ON tblSubscriptionInfo (customer_id);
```

- Secondly, the table is **not normalised**. A phone number update may potentially require updating many rows since a customer can potentially subscribe to many products.
    - The table should be normalised. This can eliminate the need of updating multiple rows when updating the phone number.
    - Based on the 'tblSubscriptionInfo' table, the functional dependencies below are assumed:
        - customer_id -> (customer_name, customer_address, customer_contact_phone)
        - product_id -> (product_name)
        - subscription_id -> (customer_id, product_id, subscription_start_date, subscription_end_date)

- Using the set of functional dependencies above, an example normalised "tblSubscriptionInfo" table in BCNF is presented below:
  - Using the normalised tables below, only one row in the CustomerInfo needs to be updated whenever a customer updates his/her phone number.

```
create table CustomerInfo
(
  customer_id int,
  customer_contact_phone varchar,
  customer_name varchar,
  customer_address varchar,
  PRIMARY KEY (customer_id)
);

create table ProductInfo (
  product_id int,
  product_name varchar,
  PRIMARY KEY (product_id)
);

– This assumes that a resubscription will create a new row instead of updating the existing
– subscription_end_date.
create table SubscriptionInfo (
  subscription_id int,
  customer_id int,
  product_id int,
  subscription_start_date datetime,
  subscription_end_date datetime,
  PRIMARY KEY (subscription_id),
  FOREIGN KEY (customer_id) REFERENCES CustomerInfo(customer_id) ON UPDATE
CASCADE,
  FOREIGN KEY (product_id) REFERENCES ProductInfo(product_id) ON UPDATE
CASCADE
);
```

- Thirdly, based on the answer in the second point, if the CustomerInfo table is too big, **sharding** can be applied to break the table into multiple pieces. This can help to distribute the incoming update workload to multiple RDBMS nodes to parallelise the workload, which in turn reduces the latency of an UPDATE query

3. Come up with the queries to find:
    - The queries below assume the RDBMS is MySQL.
    - Inference: the term ‘subscribers’ is inferred as ‘customers’.

- number of subscribers whose subscriptions will be ending in 2023;

```
SELECT COUNT(DISTINCT customer_id)
FROM tblSubscriptionInfo
WHERE subscription_end_date BETWEEN ‘2023-01-01 00:00:00’ AND ‘2023-12-31 23:59:59’;
```

- number of subscribers who have subscribed for more than 3 months in 2022;

```
SELECT COUNT(DISTINCT customer_id)
FROM tblSubscriptionInfo
WHERE
(
  subscription_end_date <= ‘2022-12-31 23:59:59’
  AND subscription_start_date >= ‘2022-01-01 00:00:00’
  AND DATEDIFF(subscription_start_date, subscription_end_date) > (31 * 3)
)
OR
(
  subscription_end_date > ‘2022-01-01 00:00:00’
  AND subscription_start_date < ‘2022-01-01 00:00:00’
  AND DATEDIFF(‘2022-01-01 00:00:00’, subscription_end_date) > (31 * 3)
)
OR
(
  subscription_end_date > ‘2022-12-31 23:59:59’
  AND subscription_start_date <= ‘2022-12-31 23:59:59’
  AND DATEDIFF(subscription_start_date, ‘2022-12-31 23:59:59’) > (31 * 3)
)
```

- subscribers who have subscribed for more than two products;

```
SELECT customer_id
FROM tblSubscriptionInfo
GROUP BY customer_id
HAVING COUNT(DISTINCT product_id) > 2;
```

- product with the most/2ndmost/3rdmost number of subscribers in 2022;

```
SELECT product_id
FROM tblSubscriptionInfo
WHERE
subscription_end_date >= '2022-01-01 00:00:00'
AND subscription_start_date <= '2022-12-31 23:59:59'
GROUP BY product
ORDER BY COUNT(DISTINCT customer_id) DESC
LIMIT 3;
```

- number of subscribers who have re-subscribed more than once for each product;

```
SELECT product_id, COUNT(customer_id)
FROM (
  SELECT product_id, customer_id
  FROM tblSubscriptionInfo
  GROUP BY product_id, customer_id
  HAVING COUNT(*) > 2
)
GROUP BY product_id;
```

- subscribers who have re-subscribed a higher version of the product in 2023 - for example Autocad 2022 to Autocad 2023.
    - Inference:
        - The statement: "subscribers who have re-subscribed a higher version of the product in 2023" is inferred as finding the customers that did resubscription(s) in 2023 and resubscribed to a more recent version of a product but not necessarily a product released in 2023.
    - Assumptions:
        - Each product_name is in the following format: '[name]:[year]', e.g. 'Autocad:2022' and ':' cannot appear in the '[name]' section of a product name.

```
WITH
T1 AS (
  SELECT customer_id,
    SUBSTRING_INDEX(product_name, ':', 1) AS product_name_without_year,
    SUBSTRING_INDEX(product_name, ':', -1) AS product_year,
    subscription_start_date
```

```
   FROM tblSubscriptionInfo
),
T2 AS (
   SELECT *,
      ROW_NUMBER() OVER W AS row_number,
      LEAD(product_year) OVER W AS prev_product_year_used,
    FROM T1
    WINDOW W AS (
       PARTITION BY customer_id, product_name_without_year
       ORDER BY subscription_start_date DESC
     )
),
SELECT DISTINCT customer_id
FROM T2
WHERE row_number = 1
AND product_year > prev_product_year_used
AND subscription_start_date >= '2023-01-01 00:00:00'
```