12 Querying Data with Time Travel

12.1 Lab Introduction

12.1.1 Purpose

The purpose of this lab is to familiarize you with how Time Travel can be used to analyze data. Specifically, you'll take a data set and compare two different points in time to satisfy a what-if question. In order to do this, you'll clone a table and use syntax designed to query two distinct Time Travel data snapshots.

If you're a data analyst, you'll learn a creative way to use a feature designed for continuous data protection for data analysis. If you're a database admin or a data engineer, you'll learn the fundamentals of how Time Travel functions for data protection.

12.1.2 What you'll learn

- How to clone a table
- How to write query clauses that support Time Travel actions
- How to fetch and utilize the ID of the last query you ran

12.1.3 How to complete this lab

In order to complete this lab, you can key the SQL commands below directly into a worksheet. It is not recommended that you cut and paste from the workbook pdf as that sometimes results in errors.

You can also use the code file for this lab that was provided at the start of the class. You would simply need to open it in TextEdit (Mac) or Notepad (Windows), and then copy and paste the code directly into a worksheet.

12.1.4 Scenario

Snowbear Air charges a different tax rate depending on the country in which the customer lives. It just so happens that countries across the world are thinking about setting a tax rate of around 5% for the kind of products that Snowbear Air sells via its promotional catalog.

Although Snowbear Air doesn't know what the new tax rate will be, leadership has decided it would like to see some kind of what-if analysis in order to determine how much more they will potentially be collecting. The concern is that a higher tax rate could result in consumers paying a higher cost based on current product prices. If that price is too high, Snowbear Air may make the decision to lower its prices in order to keep sales high.

Let's get started!

12.2 Conducting a What-If Scenario

The goal will be to determine what has been paid in taxes over the past seven years, what the amount would have been if the tax rate had been 5%, what the overall difference is, and what the difference per year might be.

12.2.1 Using the skills you learned in the first lab, create a new folder called Time Travel

12.2.2 Using the skills you learned in the first lab, create a worksheet inside the folder you just created and name it Time Travel

12.2.3 Create a new schema

For the purposes of this lab, we're going to create a new schema. Let's create the schema by running the following statements in your new worksheet:

```
USE ROLE TRAINING_ROLE;
USE WAREHOUSE LEARNER_WH;

CREATE DATABASE IF NOT EXISTS LEARNER_DB;
CREATE OR REPLACE SCHEMA LEARNER_DB.LEARNER_LAB;
```

12.2.4 The Plan

We are going to clone the REVENUE_FACT table into our schema, which will make a copy of the table that still points to all the data in the original table. Next, we are going to fetch a reference point in time. Then, we will update all the tax-related fields in the clone. Finally, using that point of reference, we will run some queries to compare the data at the time of cloning to the data after the update.

12.2.5 Key Concepts

A Time Travel reference point can consist of a TIMESTAMP, an OFFSET or a STATEMENT. You then use your reference point with an AT or BEFORE clause to fetch the data from a specific time period.

Now let's take a more detailed look at TIMESTAMP, OFFSET and STATEMENT.

TIMESTAMP, OFFSET, STATEMENT with an AT or BEFORE clause

DEFINITIONS:

A TIMESTAMP is a point in time expressed in timestamp format: 'Mon, 01 May 2020 16:20:00 -0700'.

An OFFSET is a negative integer that expresses the difference in time from the current time in seconds: i.e., -60 means 60 seconds prior to the instant the query is run.

A STATEMENT specifies the query ID of a SQL statement.

Now let's look at some examples:

EXAMPLES:

Here is a timestamp example:

```
-- EXAMPLE ONLY
-- DO NOT RUN
SELECT *
FROM SENSOR_READINGS AT(TIMESTAMP => '2018-07-27 12:00:00'::timestamp);
```

This statement selects all rows from the SENSOR_READINGS table with a timestamp equal to the one in the AT clause. Note that the text value must be converted to a timestamp in order to work.

Here is an offset example:

```
-- EXAMPLE ONLY
-- DO NOT RUN
SELECT * FROM SENSOR_READINGS AT(OFFSET => -60);
```

Selects all rows in the SENSOR_READINGS table that are 60 seconds prior to the current time. As you can imagine, offsets work well when you want to pull data regularly and at a consistent offset.

Now let's look at a statement:

```
-- EXAMPLE ONLY
-- DO NOT RUN
SELECT *
FROM SENSOR_READINGS
BEFORE(STATEMENT => '8e5d0ca9-005e-44e6-b858-a8f5b37c5726');
```

In this example, we're fetching the state of the SENSOR_READINGS table prior to the execution of the statement to which the ID shown corresponds.

Now that we have more familiarity with AT and BEFORE clauses, let's continue.

12.2.6 Clone the REVENUE_FACT table

Now let's create a zero-copy clone of REVENUE_FACT. This is essentially a snapshot of the table which shares the underlying data of the original table at the point of creation. However, once we make changes as part of this lab, those changes will be unique to our new table. And any changes to the original table of course do not impact the clone.

Clone the table using the code below and run the query to ensure the table looks as you expected:

```
CREATE OR REPLACE TABLE REVENUE_FACT_CLONE CLONE
    SNOWBEARAIR_DB.PROMO_CATALOG_SALES_STAR.REVENUE_FACT;
```

```
SELECT * FROM REVENUE_FACT_CLONE;
```

You should now see the contents of the table below the query.

12.2.7 Fetch the last query id

If you didn't run the SELECT statement from the previous step, go ahead and do it now. Running that query is critical as we are going to fetch the query id from that query to use as our reference point. If you've already run the SELECT statement from the previous step, run the statement below to fetch the query id:

```
SET UPDATE_ID = LAST_QUERY_ID();
```

Note that what we just did was to set the value of a SQL variable called UPDATE_ID to the unique id of the last query we ran by calling the LAST_QUERY_ID() function.

12.2.8 Updating the REVENUE_FACT_CLONE table

Now that we have our reference point, we'll be able to access the data that existed in the table prior to executing the UPDATE below.

The following UPDATE statement will update three columns to apply a 5% tax rate. Note that the WHERE excludes the updating of any rows where the tax rate is already 5%. Go ahead and run the UPDATE statement below now:

```
UPDATE REVENUE_FACT_CLONE
SET
      TAX_CHARGED = ROUND((GROSS_REVENUE - DISCOUNT) * (0.05),2)
    , NET_REVENUE_PLUS_TAX = ROUND((GROSS_REVENUE - DISCOUNT) * (1+0.05),2)
    , TAX_PER_PART = ROUND((GROSS_REVENUE/PART_SOLD) * 0.05,2)
WHERE
    ROUND(TAX_CHARGED/GROSS_REVENUE,2)<>0.05;
```

12.2.9 Checking the results

Now let's do our first comparison. We're going to compare the original tax rate to the new one, and the original NET_REVENUE_PLUS_TAX value to the new one.

The query below fetches the original values and the new values in two separate result sets and then joins them together.

Note that the sub-query that fetches the original values uses the following BEFORE clause:

- BEFORE (STATEMENT => $update_id)

In essence it is fetching the state of the cloned table prior to our update. Also notice that we're using a LIMIT 50 clause as well. The goal with this query is simply to check a sampel of our results and determine if the original values are indeed different from the new ones.

Go ahead and run the query now.

```
SELECT
            ROUND(ORG.TAX_CHARGED/CHG.GROSS_REVENUE,2) TAX_RATE_ORIGINAL
          , ROUND(CHG.TAX_CHARGED/CHG.GROSS_REVENUE,2) TAX_RATE_NEW
          , ORG.NET_REVENUE_PLUS_TAX AS NET_REVENUE_PLUS_TAX_ORIGINAL
          , CHG.NET_REVENUE_PLUS_TAX AS NET_REVENUE_PLUS_TAX_NEW

FROM
        REVENUE_FACT_CLONE CHG
        INNER JOIN
        (
            SELECT
                  CUSTOMER_ID
                , PART_ID
                , SUPPLIER_ID
                , ORDER_DATE_ID
                , TAX_CHARGED
                , NET_REVENUE_PLUS_TAX
                , TAX_PER_PART
            FROM
                REVENUE_FACT_CLONE

            BEFORE (STATEMENT => $update_id)
        ) ORG ON CHG.CUSTOMER_ID = ORG.CUSTOMER_ID
             AND CHG.PART_ID = ORG.PART_ID
             AND CHG.SUPPLIER_ID = CHG.SUPPLIER_ID
             AND CHG.ORDER_DATE_ID = ORG.ORDER_DATE_ID

ORDER BY
       CHG.CUSTOMER_ID
     , CHG.PART_ID
     , CHG.SUPPLIER_ID
     , CHG.ORDER_DATE_ID

LIMIT 50;
```

We won't explain the fields as the field names are probably self-explanatory. As you reviewed the results, you probably noticed that in some instances the tax rate was the same, in some it was lower and in some it was higher. So, the impact of a tax change could be that customers will pay more tax or less tax than before. Let's proceed to the next step to see what the net impact could be.

12.2.10 Aggregating the differences

The query below is essentially the same as the one above except that it tells us the net difference overall and potentially per year. Go ahead and run the query.

```
SELECT
            to_varchar(SUM(ROUND(ORG.TAX_CHARGED_ORIGINAL,2)),'999,999,999.90') AS
              TAX_CHARGED_ORIGINAL
          , to_varchar(SUM(ROUND(CHG.TAX_CHARGED,2)),'999,999,999.90') AS
             TAX_CHARGED_NEW
          , to_varchar(SUM(ROUND(CHG.TAX_CHARGED-ORG.TAX_CHARGED_ORIGINAL,2)),
             '999,999,999.90') AS DIFFERENCE
          , to_varchar(SUM(ROUND(((CHG.TAX_CHARGED-ORG.TAX_CHARGED_ORIGINAL)/7),
             2)),'999,999,999.90') AS DIFFERENCE_PER_YEAR
FROM
        REVENUE_FACT_CLONE CHG
        INNER JOIN
        (
            SELECT
                  CUSTOMER_ID
                , PART_ID
                , SUPPLIER_ID
                , ORDER_DATE_ID
                , TAX_CHARGED AS TAX_CHARGED_ORIGINAL

            FROM
                REVENUE_FACT_CLONE

            BEFORE (STATEMENT => $update_id)

        ) ORG ON CHG.CUSTOMER_ID = ORG.CUSTOMER_ID
            AND CHG.PART_ID = ORG.PART_ID
            AND CHG.SUPPLIER_ID = CHG.SUPPLIER_ID
            AND CHG.ORDER_DATE_ID = ORG.ORDER_DATE_ID

ORDER BY
       CHG.CUSTOMER_ID
     , CHG.PART_ID
     , CHG.SUPPLIER_ID
     , CHG.ORDER_DATE_ID;
```

We can see that the net difference over the 7 years is $22,200,101.63. Dividing that value by 7 gives us $3,171,432.71. So we can see that the difference is higher, and not by a trivial amount.

Naturally, if this were a real scenario, this simplistic example would not be enough to make any determinations, but it demonstrates the power of Time Travel to compare two points in time.

12.2.11 Cleaning Up

Run the statements below to drop the schema and suspend your warehouse:

```
DROP SCHEMA LEARNER_LAB;
ALTER WAREHOUSE LEARNER_WH SUSPEND;
```

12.3 Key Takeaways

- In order to leverage Time Travel data for data analysis purposes, you need a reference point. This can be a TIMESTAMP, an OFFSET or a STATEMENT.
- You will need to put an AT|BEFORE clause in your query in order to pull the results of a particular data set as of a particular point in time.

- Zero-copy cloning means that when you clone a table, at the point in time the cloned table is created, it shares the same data as the original table. Once you start making changes to either table's data, the new data resides only within the table in which the change was made.