databricks8.2 Managing Records

Managing Records

In the previous reading we demonstrated how to create a Delta table. We used basic data exploration strategies to identify two problems within the data. In this notebook, we will demonstrate how to correct those problems and write to a new, clean gold-level table that you can use for queries. Also, we will demonstrate how to repair and correct records

In this notebook, you will:

- Use a window function to interpolate missing values
- Update a Delta table
- Check the version history in a Delta table

Getting started

Run the cell below to set up your classroom environment

```
%run ../Includes/8-2-Setup
```

Mounting course-specific datasets to **/mnt/training**...
Datasets are already mounted to **/mnt/training** from **s3a://databricks-corp-training/common**

```
res1: Boolean = true
```

```
res2: Boolean = false
```

```
res3: Boolean = false
```

OK

Query returned no results

OK

OK

	num_affected_rows	num_inserted_rows
1	3408	3408

Showing all 1 rows.

OK

Repairing Records

In the previous reading, we found two problems in the data:

1. We were missing records on a single device for a period of three days
2. There was at least one broken reading (heartrate less than zero) per day in our set

We will start by demonstrating how to merge a set of updates and insertions to repair these problems.

First, we will work on the broken sensor readings. Previously, you used a window function, paired with the `AVG` function, to calculate an average value over a group of rows. Here, we will use a window function, paired with the built-in functions `LAG` and `LEAD`, to interpolate values to replace the broken readings.

`LAG`: fetches data from the previous row. Learn more (https://spark.apache.org/docs/latest/api/sql/#lag).
`LEAD`: fetches data from a subsequent row. Learn more (https://spark.apache.org/docs/latest/api/sql/#lead).

Examine the code in the next cell.

`line 1`: We create or replace a temporary view names `updates`
`line 2`: We are using a CTAS pattern to create this new view
`line 3`: We select a subgroup of columns to include from the window function defined in lines 5 - 8. Note the expression `(prev_amt+next_amt)/2`. For any missing entry, we calculate a new data point that is the mean of the previous entry and the subsequent entry. These values are defined in the window function below
`line 4`: Indicates the window function as the source. The parenthesis marks the start of the window function
`line 5`: Select all columns from `health_tracker_silver`

```
CREATE OR REPLACE TEMPORARY VIEW updates
AS (
  SELECT name, (prev_amt+next_amt)/2 AS heartrate, time, dte, p_device_id
  FROM (
    SELECT *,
    LAG(heartrate) OVER (PARTITION BY p_device_id, dte ORDER BY p_device_id,
dte) AS prev_amt,
    LEAD(heartrate) OVER (PARTITION BY p_device_id, dte ORDER BY p_device_id,
dte) AS next_amt
    FROM health_tracker_silver
  )
  WHERE heartrate < 0
)
```

OK

Check schema

We will want to use the values in `updates` to update our `health_tracker_silver` table. Let's check both schemas to see if they match.

```
DESCRIBE updates
```

	col_name	data_type	comment
1	name	string	null
2	heartrate	double	null
3	time	timestamp	null
4	dte	date	null
5	p_device_id	bigint	null

```
DESCRIBE health_tracker_silver
```

	col_name	data_type	comment
1	name	string	
2	heartrate	double	

3	time	timestamp	
4	dte	date	
5	p_device_id	bigint	
6			
7	# Partitioning		

Showing all 8 rows.

Late-arriving data

We're ready to update our silver table with our interpolated values, but before we do, we find out that those missing readings have finally come through! We can get that data ready to merge with our other updates.

Run the cell below to read in the raw data.

```
DROP TABLE IF EXISTS health_tracker_data_2020_02_late;

CREATE TABLE health_tracker_data_2020_02_late
USING json
OPTIONS (
  path "dbfs:/mnt/training/healthcare/tracker/raw-late.json",
  inferSchema "true"
  );
```

OK

Prepare inserts

We can apply the same transformations we used to create our `health_tracker_silver` table on this raw data. This we'll give us a view with the same schema as our other tables and views.

```
CREATE OR REPLACE TEMPORARY VIEW inserts AS (
  SELECT
    value.name,
    value.heartrate,
    CAST(FROM_UNIXTIME(value.time) AS timestamp) AS time,
    CAST(FROM_UNIXTIME(value.time) AS DATE) AS dte,
    value.device_id p_device_id
  FROM
    health_tracker_data_2020_02_late
)
```

OK

Prepare upserts

The word "upsert" is a portmanteau of the words "update" and "insert," and this is what it does. An upsert will update records where some criteria are met and otherwise will insert the record. We create a view that is the union of our `updates` and `inserts` and holds all records we would like to add and modify.

Here, we use `UNION ALL` to capture all records in both views, even duplicates.

```
CREATE OR REPLACE TEMPORARY VIEW upserts
AS (
    SELECT * FROM updates
    UNION ALL
    SELECT * FROM inserts
    )
```

OK

Perform upsert

When upserting into an existing Delta table, use Spark SQL to perform the merge from another registered table or view. The Transaction Log records the transaction, and the Metastore immediately reflects the changes.

The merge appends both the new/inserted files and the files containing the updates to the Delta file directory. The transaction log tells the Delta reader which file to use for each record.

This operation is similar to the SQL `MERGE` command but has added support for deletes and other conditions in updates, inserts, and deletes. In other words, using the Spark SQL command `MERGE` provides full support for an upsert operation.

Use the comments to better understand how this command integrates records from our existing tables and views.

Read more about `MERGE INTO` here (https://docs.databricks.com/spark/latest/spark-sql/language-manual/merge-into.html#merge-into--delta-lake-on-databricks).

```
MERGE INTO health_tracker_silver                                    -- the MERGE
instruction is used to perform the upsert
USING upserts

ON health_tracker_silver.time = upserts.time AND
   health_tracker_silver.p_device_id = upserts.p_device_id  -- ON is used to
describe the MERGE condition

WHEN MATCHED THEN                                                   -- WHEN MATCHED
describes the update behavior
  UPDATE SET
  health_tracker_silver.heartrate = upserts.heartrate
WHEN NOT MATCHED THEN                                               -- WHEN NOT MATCHED
describes the insert behavior
  INSERT (name, heartrate, time, dte, p_device_id)
  VALUES (name, heartrate, time, dte, p_device_id)
```

	num_affected_rows	num_updated_rows	num_deleted_rows	num_inserte
1	146	74	0	72

Showing all 1 rows.

Time travel

Let's check the number of records in the different versions of our tables.

Version 1 shows the data after we added records from February. Recall that this is where we first discovered the missing records.

The current version shows everything including the records we upcerted.

```
-- VERSION 1
SELECT COUNT(*) FROM health_tracker_silver VERSION AS OF 1
```

	count(1)
1	7128

Showing all 1 rows.

```
-- CURRENT VERSION
SELECT COUNT(*) FROM health_tracker_silver
```

	count(1)
1	7128

Showing all 1 rows.

Describe history

You can check the full history of a Delta table including the operation, user, and so on for each new write to a table.

```
DESCRIBE HISTORY health_tracker_silver
```

	version	timestamp	userId	userName
1	1	2021-10-08T02:05:31.000+0000	2348247610201072	ricardo.ninoder
2	0	2021-10-08T02:05:18.000+0000	2348247610201072	ricardo.ninoder

Showing all 2 rows.

Write to gold

So far, we have ingested raw (bronze-level) data and applied transformations to create a silver table. We have used Spark SQL to explore and transform that data further, adding new values when we found collection errors and updating the table to reflect

late-arriving data. Now that our data is clean and polished, we can write to a gold table. Gold tables are used to hold business level aggregates. When we create this table, we also apply aggregate functions to several columns.

```
DROP TABLE IF EXISTS health_tracker_gold;

CREATE TABLE health_tracker_gold
USING DELTA
LOCATION "/health_tracker/gold"
AS
SELECT
  AVG(heartrate) AS meanHeartrate,
  STD(heartrate) AS stdHeartrate,
  MAX(heartrate) AS maxHeartrate
FROM health_tracker_silver
GROUP BY p_device_id
```

```
  Error in SQL statement: AnalysisException: Cannot create table ('`default`.`
  health_tracker_gold`'). The associated location ('/health_tracker/gold') is
   not empty but it's not a Delta table
```

```
SELECT
  *
FROM
  health_tracker_gold
```

```
  Error in SQL statement: AnalysisException: Table or view not found: health_t
  racker_gold; line 4 pos 2;
  'Project [*]
  +- 'UnresolvedRelation [health_tracker_gold], [], false
```

Cleanup

Run the next cell to clean up your classroom environment

```
%run ../Includes/Classroom-Cleanup
```

Great work! Now that you've got a basic understanding of how data moves through the Delta architecture, we're ready to get back to analytics. In the next reading, we'll

see how to write high-performance Spark queries with Databricks Delta.