Submitter: Eric Wolfson

Exercise: The technical assessment for the Sr. Data Engineer MDM position at HubSpot

Files included with this exercise:

1) The original csv files, assessment PDF and readme questions provided with the exercise
2) **entityresolution.py**: The code for the entire pipeline in Dagster and Python
3) **query.py**: The code for the SQL queries as strings used while executing the SQL code by making calls to the database
4) **config.py**: The variable that must be configured in order to connect to the users database
5) **HowToRun.pdf**: The steps I took to set up and run the pipeline
6) **current_state_final.csv**: the final result csv file run with the original csv files provided with the exercise
7) **ApproachAndAssumptionsPPSlides.pptx and .pdf**: A technical explanation of the pipeline and answer to the first readme question
8) **ReadmeAnswers**: The answers to questions 2 and 3 of the Readme (this document)

Approach and Assumptions:

Please see the included PowerPoint in the zip file "**ApproachAndAssumptionsPPSlides.pptx**" and "**ApproachAndAssumptionsPPSlides.pdf**" for a full explanation of the approach and assumptions.

When viewing the slides in the pdf viewer for GitHub, Click "more pages" at the bottom continuously to expand the entire pdf.

In addition, it might be easier to view in the pptx file, since the slides are not condensed.

Challenges and Considerations:

During the implementation process there were several challenges I came across:

There are inconsistent phone number formats. I left this as is to follow the principle that data should be left as is that was entered by the user. The downsides are that the same number can be written in two separate formats, and the pipeline would consider them two separate numbers. I made the decision to tradeoff accuracy in this regard for data that is original. This is following a principle listed as a common falsehood about phone numbers at [libphonenumber/FALSEHOODS.md at master · google/libphonenumber · GitHub]:

24. **Users will only store phone numbers in your product's phone number fields**

Some users use their contact lists to store things like birthdays or other information. Unless a piece of user-supplied data has actually been verified to be a phone number, it should be stored as-is as entered by the user.

Some of the requirements that need to be taken into account for this pipeline to address the needs of stakeholders are:

1) Availability of new data: How often should the result data be available according to the stakeholders' needs? The solution I have provided contains a basic mechanism to address this, by running the pipeline automatically at set periods of time depending on which input dataset was modified.
2) Speed and Resource constraints: What tradeoffs should be used in terms of compute power and complexity of tools needed to run the solution? While adding an RDBMS (SQL Server) increases the tooling complexity, it adds to the speed of the solution. Going further and doing data cleaning and extrapolation in SQL would make the code more complicated (performing string concatenations in a declarative language), but reduce runtime even further.
3) Data Quality: Questions can arise about whether it would be preferrable to have multiple entries for the same contact in the result to preserve data, or make the data simpler with having a risk of lowering data quality by a small amount in case some records are not merged in a way to preserve data. One way this solution attempts to preserve data quality by preventing the reduction of time data is by taking records that match for all three data sets, and combining time intervals for the created_at and updated_at times into one long interval. This takes into account all the dates and times for a common record.

Next Steps and Productionization:

The next steps would involve optimizing the solution. Unfortunately, while the run time for the join is nearly instant for large datasets, that is the combine_exact_contents asset runtime (>100000 records), the rest of the pipeline doing computations with Python and Pandas scales perhaps exponentially as can be seen in the run time chart below.

The chart below shows run times for datasets of different sizes:

Seconds	Acme dataset size in records	CRM dataset size in records	RapidData dataset size in records	Combine_exact_contents (SQL joining of tables asset runtime)
14	~300	~1000	~1000	nearly instant
53	~3000	~10000	~10000	nearly instant
> 40 minutes	~30000	~100000	~100000	0.006 seconds

There are a few reasons for this. One is the query_db_into_dataframe function uses the following code:

```
get_all_query = "SELECT * FROM total_combined ORDER BY name"
data = pds.read_sql(get_all_query, conn)
```

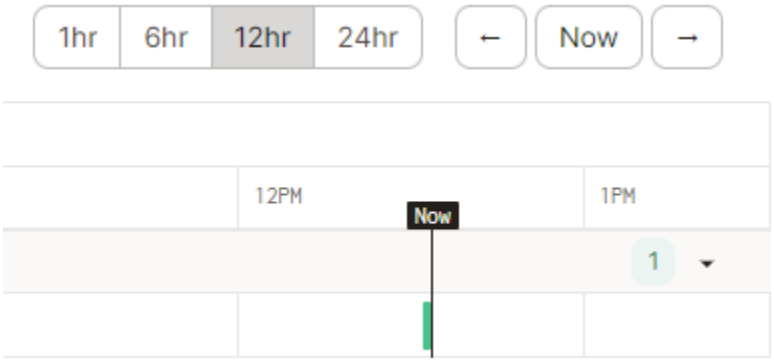
This uses the Pandas real_sql function. However, with a solution like:

```
cursor.execute("SELECT * FROM total_combined ORDER BY name")
result = cursor.getAll()
for row in result
    #do computation with result dataframe
```

We are running a query directly with SQL Server, which might be significantly faster.

One way to optimize the solution do would be to use the above code instead of read_sql. Perhaps, the best way that it could be optimized would be to allow the solution to take further advantage of parallelism, although this would increase the computing resources needed.

Notes:
The way the time was calculated was using the Dagster overview graph shown here:



And the SQL merge runtime was computed using the time() function in Python:

```
start_time = time.time()
# Join the rd_duplicates_removed table with the crm_contacts table
execute_sql_query("rdc_combined", create_rapid_data_crm_combined_view)
# Join the combined table above with the acme_contacts table
execute_sql_query("total_combined_dates", create_total_combined_view_dates)
# Create a table with the merged dates from the last table without the ip_address column
execute_sql_query("total_combined", create_total_combined_view)
sql_time = (time.time() - start_time)
print("seconds %s" % sql_time)
```

And the result was printed to the command prompt: