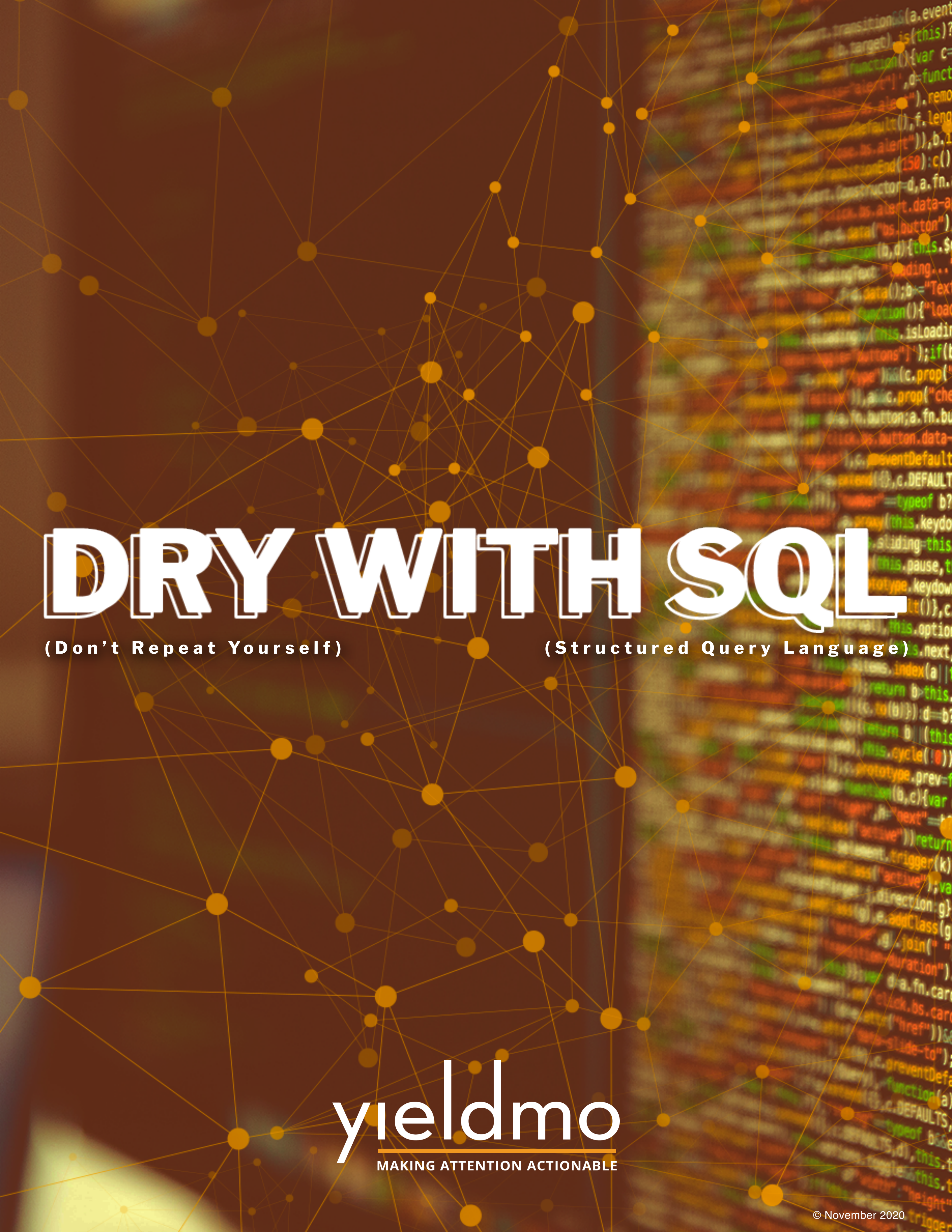


DRY WITH SQL

(Don't Repeat Yourself) (Structured Query Language)

yieldmo

MAKING ATTENTION ACTIONABLE

yieldmo
MAKING ATTENTION ACTIONABLE

TABLE OF CONTENTS

INTRODUCTION

As a data professional I have spent most of my career operating with information in relational databases and SQL(Structured Query Language) as the choice of language to analyze and gain intelligence. The thing I enjoy most about the language is its simplicity: clearly declare data points, define operations to be executed on them and state the relationships between the entities. This also means that each time I have to query for information or transform data based on business rules, I start with the declarations afresh. My colleagues who are proficient at programming languages and OOPs (Object Oriented Programing) principles find it exhausting, the lack of reusability in SQL and cringe at the possibility of repeated code.

To alleviate some of these intrinsic limitations with SQL language, the data engineers at Yieldmo have designed a straightforward approach that leverages open source Python libraries such as Jinja to assemble and render SQL statements at time of application execution. I have the honor of documenting their work here with an aim to speak to SQL engineers such as myself, how modularizing code helps in ease of read and makes room for common code.

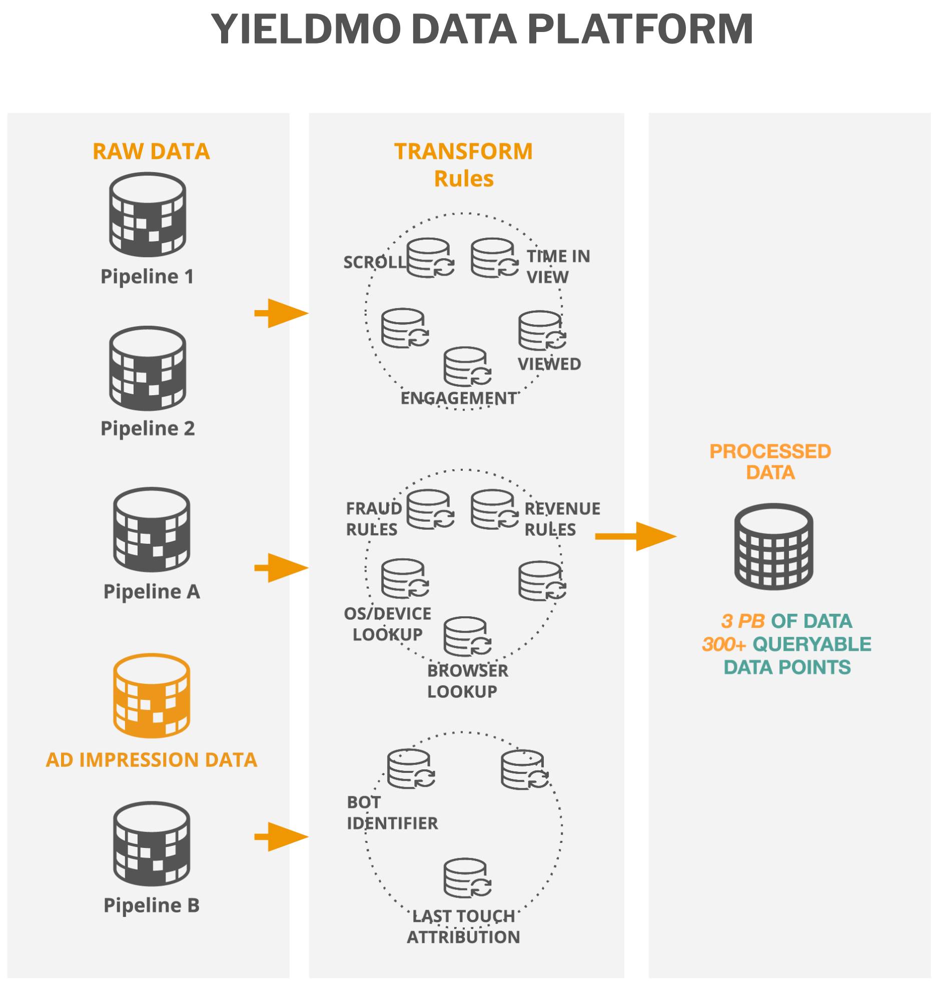
PROBLEM STATEMENT

SQL codes are long, repeated and complex.
How do I break down SQL code to increase legibility and code reuse?

DATA PLATFORM

Let me begin with an overview of our data platform. Today we have over 12+ ingest pipelines bringing in first party data into our data lake on Snowflake. These pipelines process close to 60TB of data per day. The data pipelines at Yieldmo follow the ELT (Extract Load Transform) model. The data from the source(AWS S3) is Extracted and Loaded in a raw form (json, parquest, csv etc) into our Snowflake data lake. We subsequently leverage Snowflake distributed compute to perform the transforms to flatten the semi structured data into structured columns for downstream analytics process. There are about 200 transformation rules executed every 5min in each pipeline. All transformation rules are written in SQL language. We use Python wrapper scripts to execute these SQL codes.

The below workflow diagram demonstrates the transformation workflow that each of the data points go through before making their way to structured tables. In this post, we will be focusing on the ad_impression pipeline and Lookup Transform code.



SCOPE OF DATA

The ad_impression pipeline ingests all the data points of an ad impression request made through a web/app content page. An ad request page would be such as https://cnn.com/home. Payload of Ad Impression data, includes a combination of distinct data points such as impression id, time of impression. Payload also contains highly cardinal text attributes such as Device info, Location, Browser info etc. A typical raw ad_impression payload would look like this:

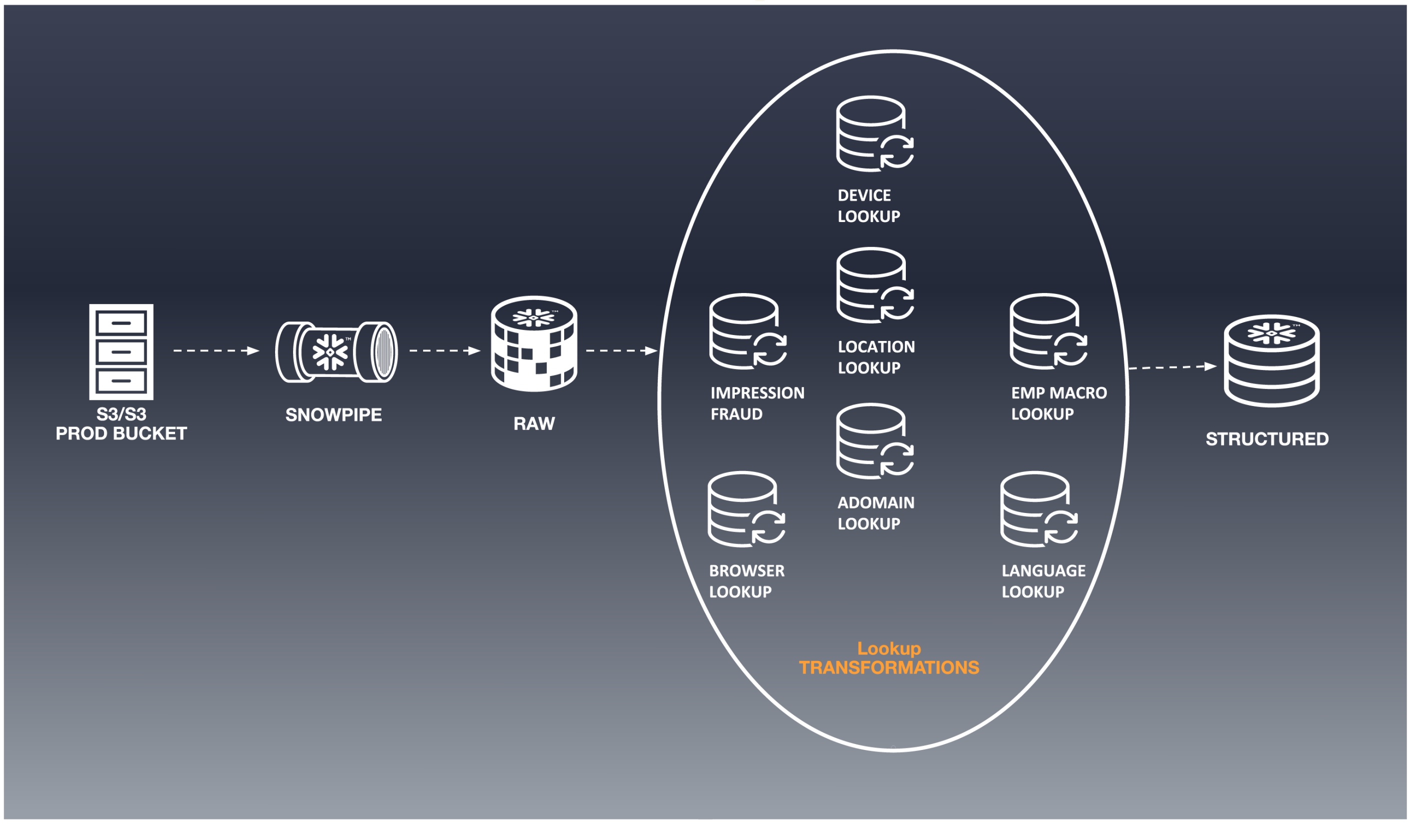
```
{
  "impression_id": 257105167335504,
  "impression_time": "2020-09-17 10:00:43.393 -0400",
  "browser": [
      {
            "browser_name": "Chrome Mobile",
                  "browser_ver": "85.0.4183.101"
      }
   ] ,
```

```
   "location": [
      {
            "city": "Winter Garden",
            "country_code": "US",
            "dma": "534",
            "postal_code": "34761",
            "region_code": "FL"
      }
   ] ,
   "device":[
      {
            "model": "LM-Q720",
            "model_code": "Stylo 5",
            "manufacturer": "LG",
            "device_type": "Phone"
      }
   ] ,
...
...
}
```

THE USE CASE

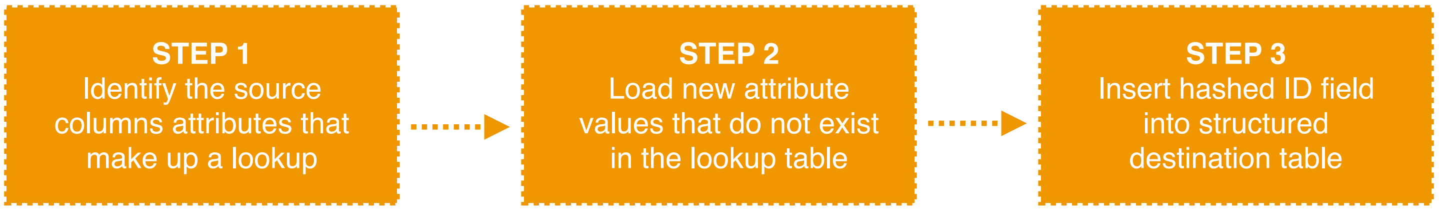
In this post, we will follow the transformation process of converting the cardinal data points in raw ad_impression into unique identifiers and recording the distinct values in a lookup table. I am going to name this transformation: **the Lookup Transform process**. The ad_impression pipeline has close to 20 such lookup processes transforming and recording corresponding unique Ids in the destination table. To explain the concepts clearly, I'll follow the transform cycle for the Browser lookup process. As shown in the sample record earlier, data points BROWSER_NAME, BROWSER_VER makeup Browser lookup.

AD IMPRESSION: LOOKUP ELT



TRANSFORM RULE FOR LOOKUP PROCESS

ELT rule for lookup transform would look like:


STEP 1
Identify the source columns attributes that make up a lookup
STEP 2
Load new attribute values that do not exist in the lookup table
STEP 3
Insert hashed ID field into structured destination table

Step#1 of the transforms differs per source - lookup combination. While **Step #2** and **Step #3** are standard rules that all lookup transforms follow. Since all the data transformation happens in Snowflake, the ELT code is written in SQL language. Below is snippet of Impression_transform.sql file:

```
-- Calculate lookup hash in temporary table
create temp table temp_src
as
(select src.impression_id
, src.browser_name
, src.browser_ver
, hash(src.browser_col_1,src.browser_col_2) as browser_hash
, hash(...) as location_hash
, ...
from raw_table);


-- Load new lookup values that do not exist in lookup table
insert into brower_lookup_table
(
 browser_name
, browser_ver
, browser_hash
)
select distinct
src.browser_name
, src.browser_ver
, browser_hash
from temp_src  src
where not exists (
                select 'x'
                from brower_lookup_table  lkp
                where src.browser_hash = lkp.browser_hash
                );

insert into location_lookup_table
(
, ...
, ...
)
select distinct
...
, ...
from temp_src  src
where not exists (
                select 'x'
                from location_lookup_table  lkp
                where src.location_hash = lkp.location_hash
                );
.. .
.. .
```

```
-- Insert browser_hash into destination table
insert into dest_table
select
src.impression_id
, browser_hash
, location_hash
...
from temp_src  src;
```

THE ISSUE

I want to highlight two main issues with the above described setup.

First: The transform sql file similar to the snippet above is usually over 3000 lines of sql code. The business logic is embedded somewhere in these seemingly large monolithic pieces of code which is hard to read and implement future changes.

Second: Adding some more complexity to the ELT process, sometimes the same group of attributes such as Browser info comes in multiple ingest pipelines and has to go through the exact same lookup transformation in each occasion. As it happens, to avoid any error, an engineer will typically copy/ paste the same sql in all other pipelines. There goes the **DRY (Don't Repeat Yourself)** principles out of the door.

THE SOLUTION

Now that we have a fairly good understanding of the lay of the land, in the following sections I'll walk through how we have evolved the SQL code for the Lookup process. Here I'll demonstrate how the same SQL code can be modularized to allow for ease of read and kept DRY for reusability. We have instrumented these concepts through Jinja libraries in Python. Think of Jinja as a programming language that can be used to build out statements which then get executed within Python's main program. Jinja templates offer various functionalities such as variable substitution, filters, loops, function calls etc most of which we have leveraged in this implementation.

MODULARIZING SQL

The idea of modularity is to break down long procedural codes into smaller sub routines which then gets called within a much shorter main program.

Modularity serves two purposes:

- **One** the decomposition of long main code aids to ease of readability and faster troubleshooting.
- **Second** smaller sub routines allow design for reusable/ common code objects which can be called in multiple workflows.

HELPER UTILITY

Referring to the transform rules for the Lookup process described earlier, transform Step#1 differs per source - lookup combination, while transform Step #2&3 is standard rules that all lookup transforms follow. So there is opportunity for building common code. From the code snippet provided earlier, it is clear that the INSERT statements for Step #2 can be generated once we have the **source table** name, **lookup table** name and the map of corresponding **column attributes**. The map of return column per lookup table will get us to Step #3. We have achieved generating the required statements for Step #2 and #3 through a helper utility. The helper utility takes the necessary parameters to generate an INSERT template in Jinja. The same helper utility has a class object to provide the return column fields.

```
class LookupTableInfo(NamedTuple):
   table_name: str
   column_list: List[str]
   join_column_list: List[str]
   return_column: str

class SourceTableInfo(NamedTuple):
    table_name: str
    column_list: List[str] = []
    join_column_list: List[str] = []


class LookupHelper(object):
    def __init__(self, source_table: SourceTableInfo, lookup_table: LookupTableInfo):
):
    ...
    ...

  # Define attributes
    ...
    ...
    @staticmethod
    def convert_list_to_str(column_list: List[str]) -> str:
        return "\n, ".join(column_list)

    def generate_lookup_insert_statement(self) --> str:
    return f"""
insert into {self.lookup_table.table_name}
(
{self.convert_list_to_str(self.lookup_table.column_list)}
)
select distinct {self.convert_list_to_str(self.source_table.column_list)}
from {self.source_table.table_name} src
where not exists (
            select 1
            from {self.lookup_table.table_name} lkp
            where lkp.{self.convert_list_to_str(self.lookup_table.join_column_list)} =
src.{self.convert_list_to_str(self.source_table.join_column_list)}
            )
;
"""
```

This helper utility sits outside of any of the pipeline load processes. In the main pipeline ingest sql template where we earlier had code for INSERT into the lookup table, I have now replaced them with calls to the utility. When jinja renders the main ingest sql template it also appends the INSERT statements by executing calls to the utility functions. So when the final sql renders it looks identical to the original Impression_transform.sql. In the ad_impression pipeline alone there are over 19 such lookup transforms, for this Jinja allows us to iterate through a lookup list to create all the 19 insert statements and corresponding return columns. With the helper library in place, I am able to break down the main program into subroutines and introduce looping functionality.

Now let me give a walk through of the implementation of the three transform steps and show we have achieved modularity for SQL code using Python and Jinja.

TRANSFORM #1

Let’s create a lookup table info file called **lookup_map**, which keeps track of lookup tables, mapping columns, join columns and return columns. Note this file will have an entry for all lookups irrespective of pipeline.

```
browser = LookupTableInfo(table_name='browser',
                          column_list=['browser_name', 'browser_ver', 'browser_hash'],
                          join_column_list=['browser_hash'],
                          return_column='browser_hash')

Location = LookupTableInfo(table_name='location',
                          column_list=[...],
                          join_column_list=['location_hash'],
                          return_column='location_hash')

Location = LookupTableInfo(table_name='device'
                          column_list=[...],
                          join_column_list=['device_hash'],
                          return_column='device_hash')
...
...
```

Lets instantiate a dictionary object that maps destination table with LookupHelper object list.

```
import common.lookup_map as lkp
from common.lookup_helper import LookupHelper

lookup_helper_object_list = {'ad_impression': [LookupHelper(src, lkp.browser),
                                                LookupHelper(src, lkp.location),
                                                LookupHelper(...),
                                                ...]
```

TRANSFORM #2

In the main sql template, instead of explicit insert statements, I have added a **for** loop that iterates over the **LookupHelper** object in dictionary **lookup_helper_object_list** to generate insert statements per lookup.

TRANSFORM #3

In the main sql template, at the final insert into the destination table sql statement, I appended a **for** loop that iterates over **LookupHelper** object in dictionary **lookup_helper_object_list** to return column.

When jinja renders the main sql template, it executes the helper call, variable substitutes the source, lookup table names and column list to generate insert sql and return columns. Jinja will iterate over the object list and generate statements per lookup table. Jinja appends these statements to the finally executed sql. Below is modular Impression_transform.sql with calls for insert statements and return columns. Notice how the sql code is now rather small and broken down into subroutines. Future enhancements for new lookup requests or adding new attributes to existing lookups, we just have to change the lookup_map and dictionary files.

```
CREATE or REPLACE TEMPORARY TABLE {{tgt_schema}}.temp_src
AS
SELECT impression_id
, browser_name
, browser_ver
, browser_hash
...
FROM {{src_schema}}.{{src_table}}
;

{% for lookup_helper in lookup_helper_object_list %}
{{lookup_helper.generate_insert_statement()}}
{% endfor %}

--LOAD DESTINATION TABLE
INSERT INTO {{TGT_SCHEMA}}.{{TGT_TABLE_NAME}}
(
IMPRESSION_ID
...
{% for lookup_helper in lookup_helper_object_list %}
{{', ' + lookup_helper.lookup_table.return_column}}
{% endfor %}
)
(
SELECT
IMPRESSION_ID
...

{% for lookup_helper in lookup_helper_object_list %}
{{', ' + lookup_helper.lookup_table.return_column}}
```

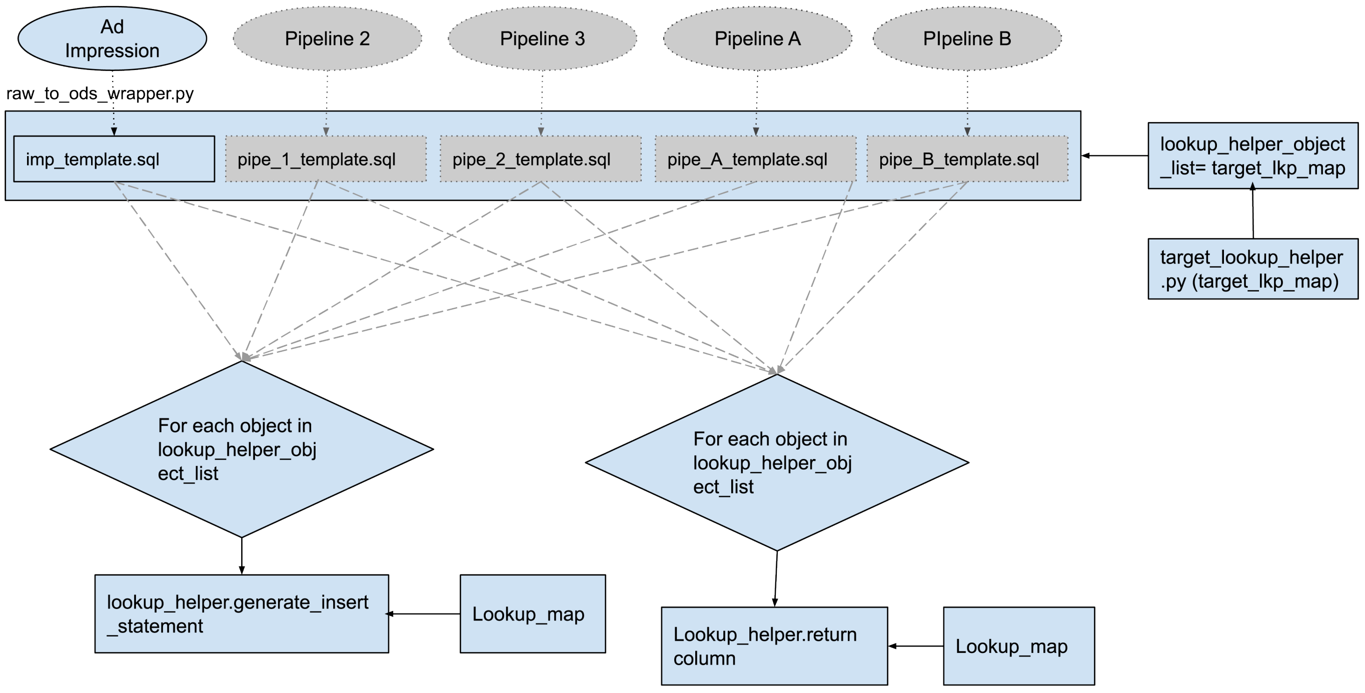
```
{% endfor %}

FROM {{TGT_SCHEMA}}.TEMP_TGT
```

KEEPING PIPELINES DRY

We start from a place of anti-DRY. As it happens in most platform development lifecycles, our ELT pipelines were built iteratively one at a time over the course of 3-4 years. The engineering team was sensitive to time to market, hence the initial versions of the transform code looked more like a long form running sql. The transformation logic is copy pasted in multiple locations to avoid errors. Future enhancements to the pipelines such as adding a new lookup attribute meant changing the same code in multiple locations, which in turn increased testing and deployment time. All this code duplication resulted in inefficiency in release cycles and incident triage.

Transformation rules for Lookups across all pipelines follow the same steps. I.E. Identify and insert unique values from the source into a lookup table, replace cardinal text fields with corresponding IDs in the flattened destination table. They differ in source to lookup mapping, attribute fields that make up the lookup and destination table.

Below is a visual of how we have achieved DRY with lookup transformations by creating two helper libraries: **Lookup_helper**, **lookup_map**. Both these libraries help us encapsulate the repeated parts of the lookup transform rules. While the dictionary defined in the beginning, **lookup_helper_obj_list** : helps us track a map of lookups to field list to destination tables


Ad Impression
Pipeline 2
Pipeline 3
Pipeline A
PIpeline B
raw_to_ods_wrapper.py
imp_template.sql
pipe_1_template.sql
pipe_2_template.sql
pipe_A_template.sql
pipe_B_template.sql
lookup_helper_object_list= target_lkp_map
target_lookup_helper.py (target_lkp_map)
For each object in lookup_helper_object_list
For each object in lookup_helper_object_list
lookup_helper.generate_insert_statement
Lookup_map
Lookup_helper.return column
Lookup_map

FINAL THOUGHTS

SQL is strongest as a medium for data analysis and manipulating information in a relational system. SQL's declarative nature is also limiting when we adapt it as our core language to build out an enterprise wide ELT platform. SQL rendered using Python Jinja templates is a simple way to introduce concepts such as inheritance, variable substitution, loops within the SQL code that we already love and are used to. In a data platform our size, modular and DRY code patterns bring in tremendous efficiencies. Just in the sample versions of the Impression_transform.sql shared earlier, we can see that modular code has reduced the code lines by 40%. Standardizing Lookup transform processes across pipelines with the use of helper libraries has reduced the development cycles from 5 days to 1 day.

SQL on steroids, that should have been the title of this post. My SQL friends, brave this whole new world of modular and reusable SQL code. It is a crazy journey, but believe me as it comes from a long time practitioner, all this new flexibility in an already powerful data manipulation language makes SQL a killer language for platform development. Hope discussion of our use case here will jog new code design ideas for your data systems.

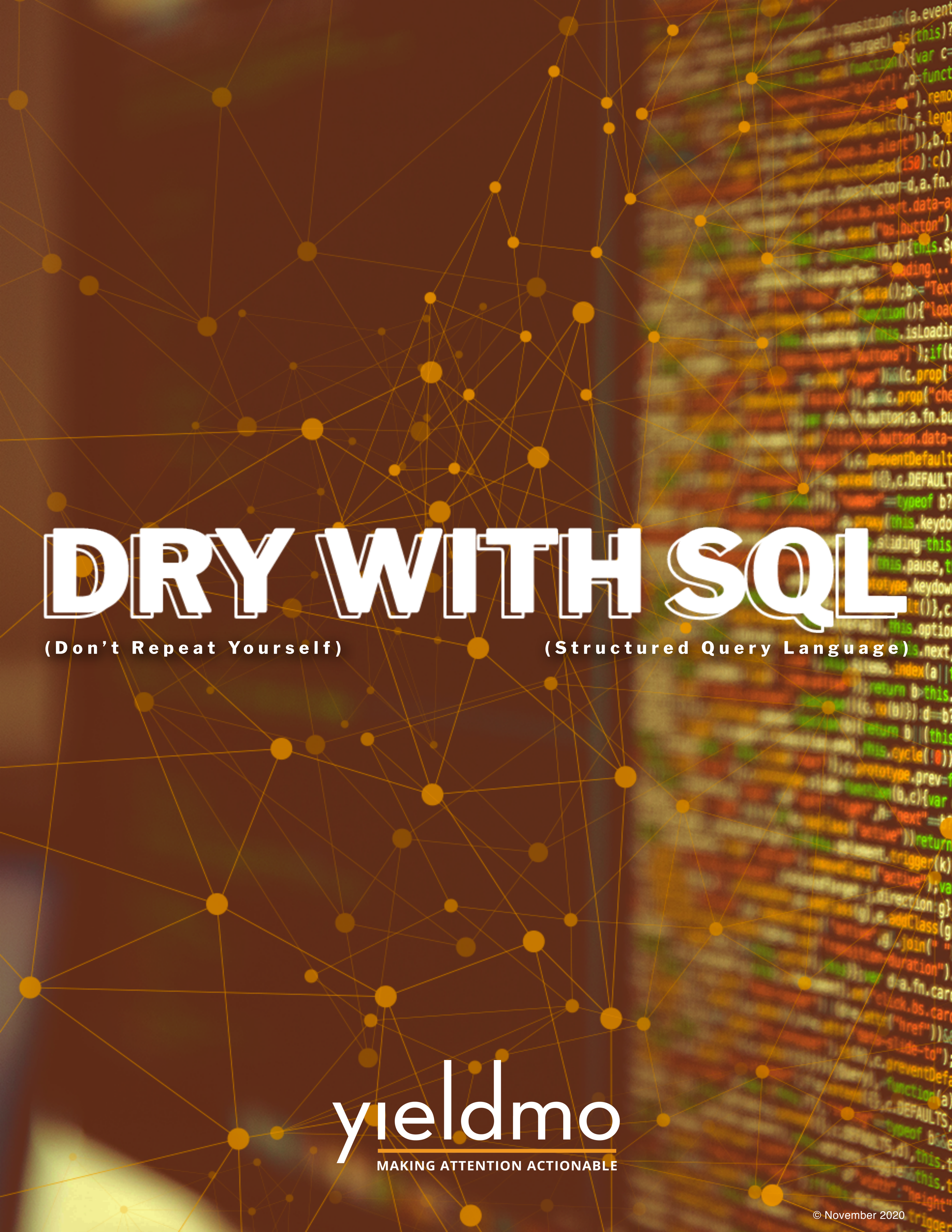
BY INDU NARAYAN

VP OF DATA ARCHITECTURE AT YIELDMO

Indu is a seasoned leader with a successful track record of developing high performance teams. As VP of Data Architecture, she is responsible for all aspects of Data at Yieldmo. She and her team are constantly involved in innovating scalable cloud based data platform that serve as the foundation for analytics products for Programmatic, Header, Measurement businesses.

Indu's passion for information and engineering has been core to how she has shaped the data practice at Yieldmo. Given the success of Yieldmo Data Platform, she and her team are often invited for speaking engagements at tech conferences in the country. At these architecture forums the team has evangelized Yieldmo's design of efficient data-lake platform and story of their transformation journey.

LEARN MORE ABOUT YIELDMO VISIT US AT WWW.YIELDMO.COM



DRY WITH SQL

(Don't Repeat Yourself) (Structured Query Language)

yieldmo

MAKING ATTENTION ACTIONABLE