Building Better Pipelines with Snowpark

D4S 42

Snowflake Summit - 2023

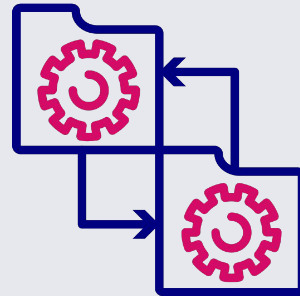
Bruce Oliver - Sr. Team Lead

Who we are

At DAS42, we use data to transform your organization for long-term success.

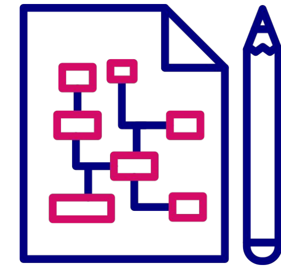
DAS42

We’re a data consultancy that brings clarity to complex data issues.

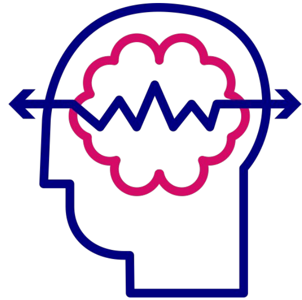


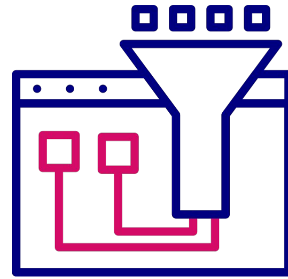
We engage end to end across data ecosystems.

We prescribe customized solutions for companies’ most urgent needs.



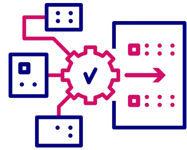
We help companies take control of their data.





Our modern point of view ensures we always prescribe the right solutions.

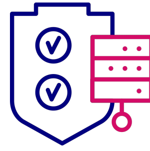
Common Challenges We Help You Tackle



Data Platform Modernization & Migration

Consolidate disparate data into a centralized data lake / data warehouse

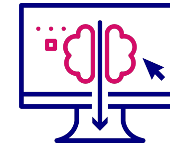
Transition to new tools without completely upending your current processes and ability to measure key business metrics



Data Governance

Build the bridge to bring your people and data together

Create a common language and shared understanding



Self-Service Business Intelligence

Design an ecosystem where users can self-serve to get the data they need through Data Democracy

Snowpark at a Glance

DAS42

What is Snowpark

- An API for querying and processing data using Scala, Java, or Python.
  - Snowpark converts operations into SQL
- Move your code to the data, and not your data to the code.
  - Distributed processing
  - Snowflake's end-to-end data security and governance
- UDFs, UDTFs, and Stored Procedures

Snowpark API Reference (Python)

Snowpark is a new developer experience that provides an intuitive API for queryi
data pipelines and allows you to interact with Snowflake directly without moving c
see the Snowpark Developer Guide for Python.

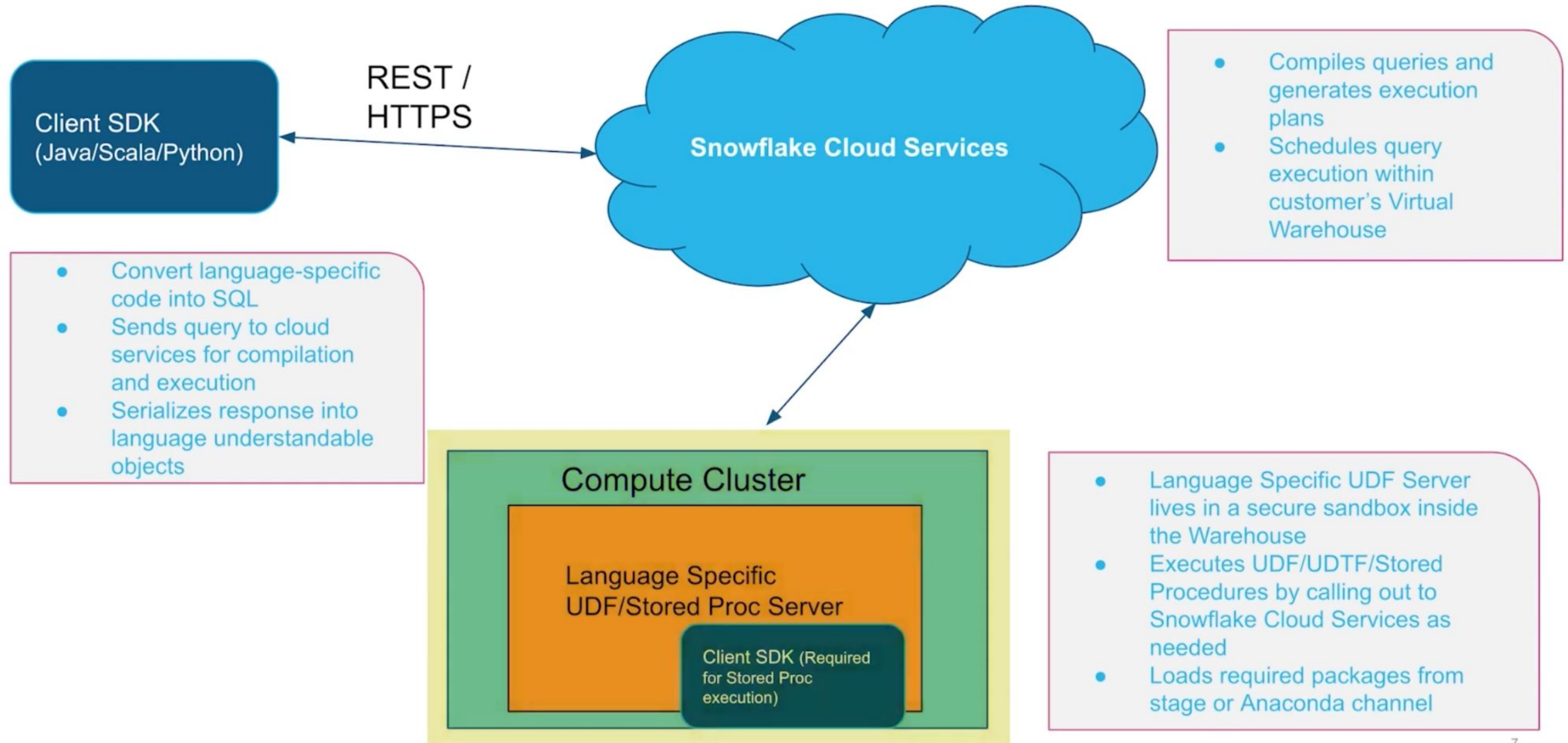
- Snowpark Session
  - snowflake.snowpark.Session
  - snowflake.snowpark.Session.add_import
  - snowflake.snowpark.Session.add_packages
  - snowflake.snowpark.Session.add_requirements
  - snowflake.snowpark.Session.call
  - snowflake.snowpark.Session.cancel_all
  - snowflake.snowpark.Session.clear_imports
  - snowflake.snowpark.Session.clear_packages
  - snowflake.snowpark.Session.close
  - snowflake.snowpark.Session.createDataFrame
  - snowflake.snowpark.Session.create_async_job
  - snowflake.snowpark.Session.create_dataframe
  - snowflake.snowpark.Session.flatten
  - snowflake.snowpark.Session.generator
  - snowflake.snowpark.Session.get_current_account
  - snowflake.snowpark.Session.get_current_database
  - snowflake.snowpark.Session.get_current_role
  - snowflake.snowpark.Session.get_current_schema
  - snowflake.snowpark.Session.get_current_warehouse
  - snowflake.snowpark.Session.get_fully_qualified_current_schema

Snowpark in Action

- ML/AI
- Complex Tasks – specifically transformations that are more of a python/java/scala problem than a SQL problem
- Data-intensive applications
- Spark Jobs

DAS42

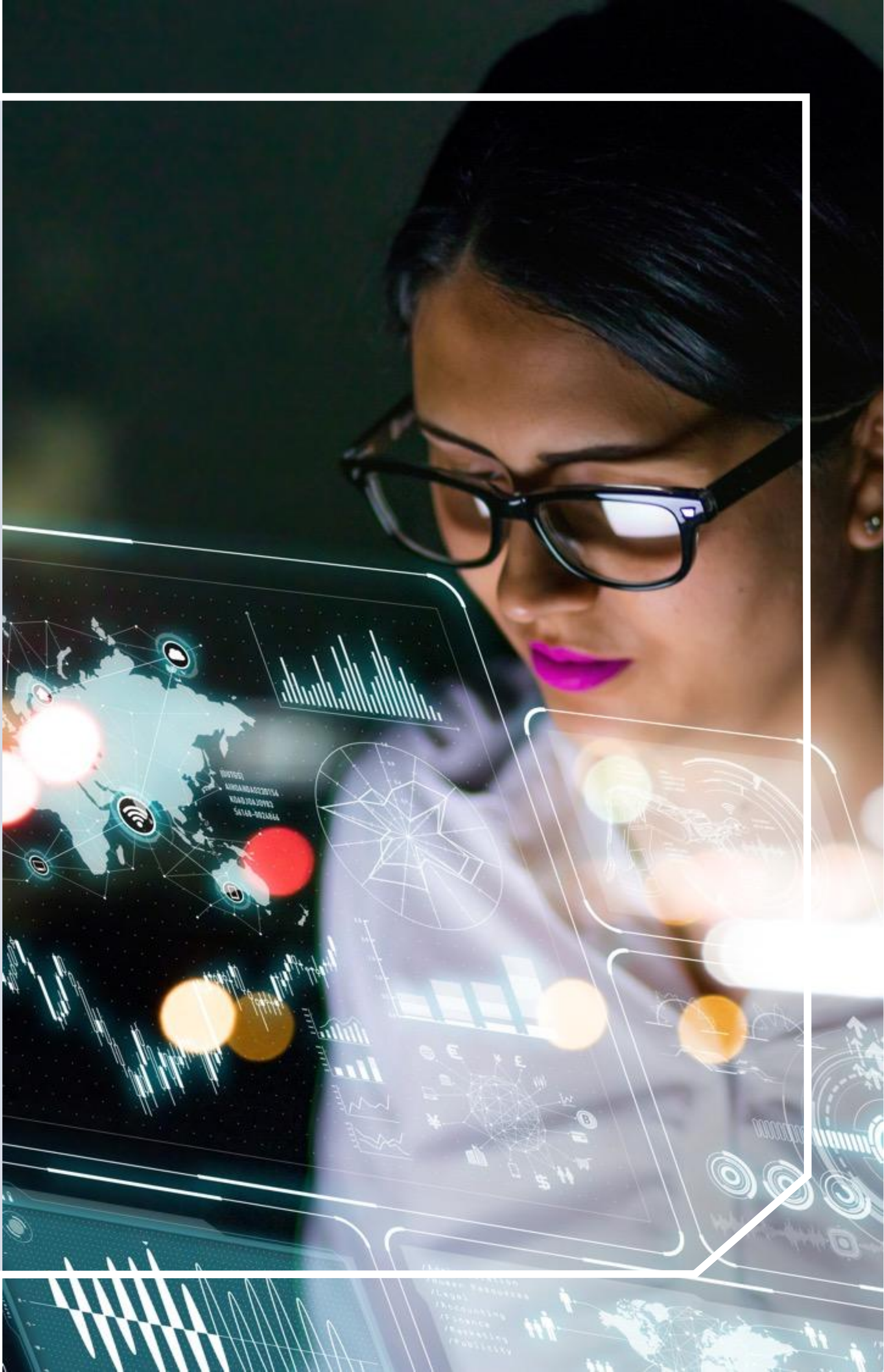
Under the Hood



DAS42

Snowpark Demo

DAS42



Problem

We need to turn this:

```
2023-04-17T16:33:25+0000 flowIdLog, applianceName=VCG-EAST-01-PHE,
tenantName=Westeros, flowId=34762083, flowCookie=1681749523,
sourceIPv4Address=172.16.102.102, destinationIPv4Address=8.8.8.8,
sourcePort=64447, destinationPort=53, tenantId=9, vsnId=0,
applianceId=1, ingressInterfaceName=dtvi-0/3766,
egressInterfaceName=vni-0/0.0, fromCountry=, toCountry=,
protocolIdentifier=17, fromZone=ptvi, fromUser=Unknown,
toZone=RTI-INET-Zone, icmpTypeIPv4=0
```

DAS42

Problem

Into this:

LOG_TYPE	APPLIANCEID	APPLIANCENAME	DESTINATIONIPV4ADDRESS	DESTINATIONPORT	EGRESSINTERFACENAME	FLOWCOOKIE ···	FLOWCOOKIETIME	FLOWID	FROMCOUN	FROMUSER	FROMZONE	ICMPTYPEIPV4	INGRESSINTERFACI
flowIdLog	1	Dallas	10.40.255.255	138	vni-0/0.0	1,681,754,897	2023-04-17 18:08:17	33569124		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,754,981	2023-04-17 18:09:41	33569416		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,755,054	2023-04-17 18:10:54	33569283		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	Kansas-City-Site	10.40.255.255	137	vni-0/0.0	1,681,754,973	2023-04-17 18:09:33	2181050535		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,754,978	2023-04-17 18:09:38	33569350		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,755,090	2023-04-17 18:11:30	33569383		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Kansas-City-Site	10.40.255.255	137	vni-0/0.0	1,681,755,088	2023-04-17 18:11:28	2181050634		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,760,337	2023-04-17 19:38:57	33579644		Unknown	INET	0	vni-0/0.0
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,760,339	2023-04-17 19:38:59	1108050915		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,760,360	2023-04-17 19:39:20	1108050929		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,766,255	2023-04-17 21:17:35	33592097		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,766,206	2023-04-17 21:16:46	1108060417		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	Dallas	10.40.255.255	138	vni-0/0.0	1,681,751,555	2023-04-17 17:12:35	33563320		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,751,676	2023-04-17 17:14:36	33563239		Unknown	LAN1-VNI-0-2	0	vni-0/2.0

DAS42

Problem

- We almost certainly *could* write SQL to perform this transformation.
- But it would be:
  - Very gross
  - Very difficult to write
  - Very difficult to debug
  - Likely slow and inefficient
- Or we could leverage the Snowpark python SDK to transform this unstructured data into semi structured data, and let Snowflake features take care of the rest.

Solution

- You can now define a Python UDF directly in Snowflake's new Python worksheets.
- Allows you to leverage Python and, in this case, the regex library.
- This ends up being a relatively simple problem to solve with Python – just three functions.
- The best part: you can simply call the UDF in a SQL statement.

```
create or replace function transform_to_json(raw_data varchar)
returns variant
language python
runtime_version = '3.8'
handler = 'transform'
as
$$
import re

def handle_commas_in_quotes(input_string):
    pattern = '\".*\"'
    match = re.search(pattern, input_string)
    if match:
        matched_string = match.group()
        new_string = matched_string.replace(',', '|')
        output_string = input_string.replace(matched_string, new_string)
        return output_string
    else:
        return input_string

def handle_commas_in_parenthesis(input_string):
    pattern = '\(.*\)'
    match = re.search(pattern, input_string)
    if match:
        matched_string = match.group()
        new_string = matched_string.replace(',', '|')
        output_string = input_string.replace(matched_string, new_string)
        return output_string
    else:
        return input_string

def transform(raw_data):
    string_data = raw_data
    string_data = handle_commas_in_quotes(string_data)
    string_data = handle_commas_in_parenthesis(string_data)
    split_data = string_data.split(', ')
    log_dict = {}
    try:
        log_dict['timestamp'] = split_data[0].split(' ',)[0]
    except:
        print('error getting timestamp.')
    try:
        log_dict['log_type'] = split_data[0].split(' ',)[1]
    except:
        print('error getting log_type')
    remaining_data = split_data[1:]

    for record in remaining_data:
        key = record.split('=')[0]
        value = record.split('=')[1].replace(',','')
        log_dict[key] = value

    return log_dict
$$;
```

Solution

- What would have been some very nasty SQL transformations is now just:

```
-- Ingestion of data from Azure into event_hub_logs_temp table
copy into raw_ingest_{{ params.env }}.versa_ingest_{{ params.env }}.event_hub_logs_temp from (
select
    $1 as raw_data,
    to_varchar(raw_data:Body::BINARY, 'utf-8') as record_content,
    transform_to_json(record_content) as converted_json,
    converted_json:log_type::string as log_type,
    metadata$filename as load_file,
    current_timestamp as load_timestamp,
    {{ execution_date.strftime("%Y%m%d%H") }} as dag_execution_date_key
from
    @event_hub_stage_{{ params.env }}/{{ params.azure_blob_path }}/{{ execution_date.strftime("%Y/%m/%d/%H" )}} t)
file_format = (format_name = 'event_hub_avro_format')
pattern = '.*[.]avro'
on_error = continue
;
```

- Record_content is our syslog-formatted string. We just call our UDF and hand it record_content as an input arg, and it spits out the same string, but in JSON format.

DAS42

Solution

- The previous COPY INTO statement leaves us with a table that looks like this:

Raw Binary Data | **Syslog format (binary converted to utf-8)** | **JSON variant type of SysLog format (UDF output)**

RAW_DATA	RECORD_CONTENT	CONVERTED_JSON	LOG_TYPE
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:05+0000 flowMonLog, applianceName=[	{ "appFamily": "general-internet", "appIdStr": "mozilla", "appProdu	flowMonLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:27+0000 monStatsLog, applianceName=	{ "applianceName": "VCG-WEST-01-PHE", "destIp": "34.93.91.7",	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:32+0000 monStatsLog, applianceName=	{ "accCkt": "INET", "accCktId": "1", "appId": "smb", "applianceNar	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:32+0000 monStatsLog, applianceName=	{ "applianceName": "Dallas", "log_type": "monStatsLog", "mstatsAt	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:32+0000 monStatsLog, applianceName=	{ "accCkt": "INET", "accCktId": "1", "appId": "ssdp", "applianceNa	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:32+0000 monStatsLog, applianceName=	{ "accCkt": "INET-2", "accCktId": "2", "appId": "smb", "applianceN	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:32+0000 monStatsLog, applianceName=	{ "accCkt": "INET-2", "accCktId": "2", "appId": "nbns", "applianceI	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:34+0000 monStatsLog, applianceName=	{ "applianceName": "LosAngeles", "log_type": "monStatsLog", "ms	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:34+0000 monStatsLog, applianceName=	{ "applianceName": "LosAngeles", "log_type": "monStatsLog", "ms	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:34+0000 monStatsLog, applianceName=	{ "applianceName": "LosAngeles", "log_type": "monStatsLog", "ms	monStatsLog
{ "Body": "323032332D30342D31375431363A32393A3	2023-04-17T16:29:34+0000 monStatsLog, applianceName=	{ "accCkt": "INET", "accCktId": "1", "appId": "nbns", "applianceNa	monStatsLog

Solution

- All that is left to do is convert the output of the UDF (JSON format in Variant data type) into the typical columnar storage we all know and love:

```
select
    log_type,
    converted_json:applianceId::string as applianceId,
    converted_json:applianceName::string as applianceName,
    converted_json:destinationIPv4Address::string as destinationIPv4Address,
    converted_json:destinationPort::string as destinationPort,
    converted_json:egressInterfaceName::string as egressInterfaceName,
    converted_json:flowCookie::number as flowCookie,
    TO_TIMESTAMP_NTZ(converted_json:flowCookie::number) as flowCookieTime,
    converted_json:flowId::string as flowId,
    converted_json:fromCountry::string as fromCountry,
    converted_json:fromUser::string as fromUser,
    converted_json:fromZone::string as fromZone,
    converted_json:icmpTypeIPv4::string as icmpTypeIPv4,
    converted_json:ingressInterfaceName::string as ingressInterfaceName,
    converted_json:protocolIdentifier::string as protocolIdentifier,
    converted_json:sourceIPv4Address::string as sourceIPv4Address,
    converted_json:sourcePort::string as sourcePort,
    converted_json:tenantId::string as tenantId,
    converted_json:tenantName::string as tenantName,
    TO_TIMESTAMP_NTZ(converted_json:timestamp::string, 'YYYY-MM-DDTHH24:MI:SSTZHTZM') as timestamp,
    converted_json:toCountry::string as toCountry,
    converted_json:toZone::string as toZone,
    converted_json:vsnId::string as vsnId,
    sysdate() as last_updated
from
    raw_ingest_{{ params.env }}.versa_ingest_{{ params.env }}.event_hub_logs
```

DAS42

Solution

- Which leaves us with our desired final output:

LOG_TYPE	APPLIANCEID	APPLIANCENAME	DESTINATIONIPV4ADDRESS	DESTINATIONPORT	EGRESSINTERFACENAME	FLOWCOOKIE	···	FLOWCOOKIETIME	FLOWID	FROMCOUN	FROMUSER	FROMZONE	ICMPTYPEIPV4	INGRESSINTERFACI
flowIdLog	1	Dallas	10.40.255.255	138	vni-0/0.0	1,681,754,897		2023-04-17 18:08:17	33569124		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,754,981		2023-04-17 18:09:41	33569416		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,755,054		2023-04-17 18:10:54	33569283		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	Kansas-City-Site	10.40.255.255	137	vni-0/0.0	1,681,754,973		2023-04-17 18:09:33	2181050535		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,754,978		2023-04-17 18:09:38	33569350		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,755,090		2023-04-17 18:11:30	33569383		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Kansas-City-Site	10.40.255.255	137	vni-0/0.0	1,681,755,088		2023-04-17 18:11:28	2181050634		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	10.40.255.255	137	vni-0/0.0	1,681,760,337		2023-04-17 19:38:57	33579644		Unknown	INET	0	vni-0/0.0
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,760,339		2023-04-17 19:38:59	1108050915		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,760,360		2023-04-17 19:39:20	1108050929		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,766,255		2023-04-17 21:17:35	33592097		Unknown	LAN1-VNI-0-2	0	vni-0/2.0
flowIdLog	1	VCG-WEST-01-PHE	34.107.221.82	80	vni-0/0.0	1,681,766,206		2023-04-17 21:16:46	1108060417		Unknown	ptvi	0	dtvi-0/4312
flowIdLog	1	Dallas	10.40.255.255	138	vni-0/0.0	1,681,751,555		2023-04-17 17:12:35	33563320		Unknown	INET	0	vni-0/0.0
flowIdLog	1	Dallas	34.107.221.82	80	dtvi-0/553	1,681,751,676		2023-04-17 17:14:36	33563239		Unknown	LAN1-VNI-0-2	0	vni-0/2.0

Solution Benefits

- We're able to use the right tool for the job. This particular transformation is more of a Python problem than a SQL problem.
  - Saves development time
  - Easier code to read, write, and debug
- Our data never left Snowflake – even though we interacted with the data using a different programming language.
- We get simplicity and readability in our pipeline. An otherwise unique data source format looks and feels like the rest of our sources once the UDF transform is applied.

Questions?

DAS42

©2023 DAS42, LLC. All rights reserved.

das42.com