2024년 09월 11일

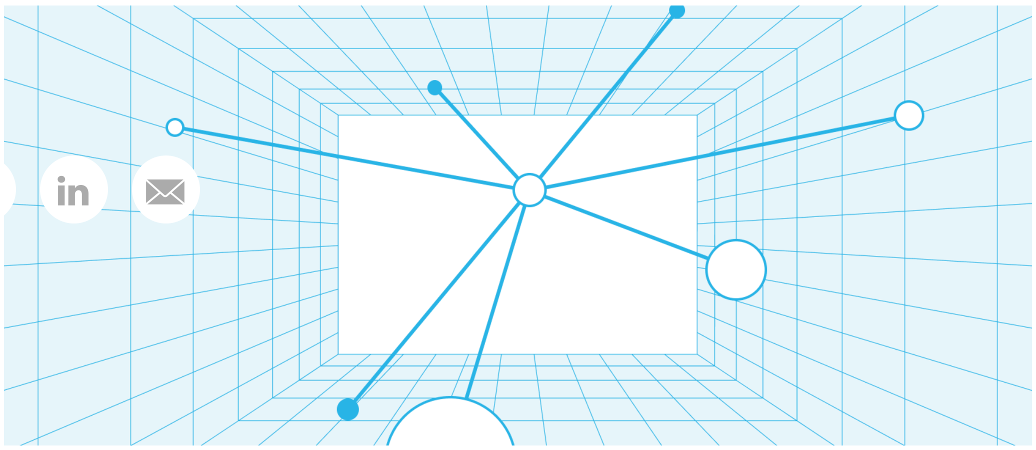
AUTHOR

Niranjan Sharma

Optimize Your Data Pipelines by Augmenting Network Concurrency with Snowpark External Access

Machine Learning

SHARE



Snowpark External Access enables easier connectivity to external network locations from Snowpark user-defined functions (UDFs) and stored procedures, offering a secure and efficient method for data ingestion and enrichment. This blog post explores how to leverage this feature for optimal concurrency in your data operations, focusing on the implementation of parallel network connections.

Understanding Snowpark External Access

With Snowpark External Access, Snowflake users can establish secure connections with specific external network locations.

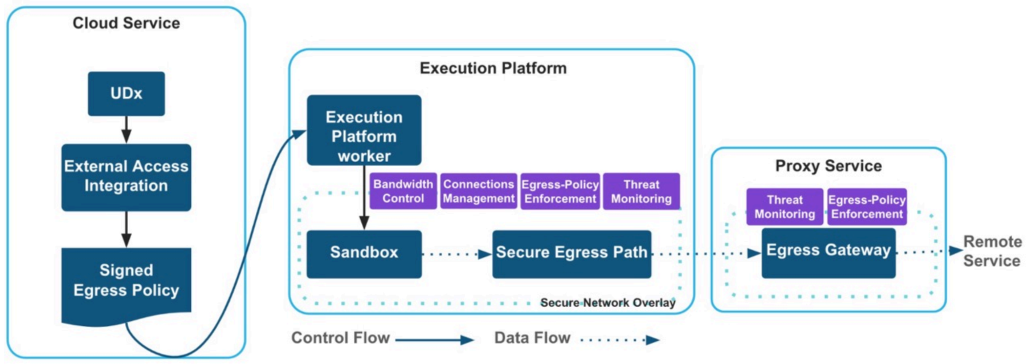


Figure 1. Diagram of Snowpark External Access, which offers secure network overlay with data and control flow.

As shown in Figure 1, Snowflake utilizes a secure network overlay at the Execution Platform (Snowpark Sandbox) to manage and isolate network resources for Sandbox, enforce egress policies and monitor for threats. The design includes:

AUTHOR

Niranjan Sharma

Secure network overlay: Provides strict control over Sandbox traffic, enabling effective segmentation and optimized performance.

Bandwidth control: Limits bandwidth for Sandbox traffic to prevent overuse by potentially malicious user workloads.

Connection management: Regulates TCP connections to safeguard the Execution Platform's network resources.

Egress-policy enforcement: Enforces packet-level policies to regulate oi affic. Snowflake applies egress policies at both the Execution Platform level and the Proxy Service to address advanced threat actors who might bypass Sandbox protections and evade policy enforcement at the Execution Platform.

Threat monitoring: Detects potential malicious activity by analyzing unusual network traffic patterns.

When a Snowpark External Access UDx (e.g., UDF, UDTF or UDAF) is executed within a query, the following process occurs:

1. **Authorization:** The UDx is granted permission to access hosts defined in the allowed network rules through the external access integration.
2. **Egress-policy generation:** The cloud service generates and signs the egress policy for the UDx, outlining the list of permitted destinations. This policy is then delivered to the Execution Platform.
3. **Pre invocation setup:** Before invoking the UDx handler, the Execution Platform Worker performs two essential tasks:

    Policy distribution: The egress policy is sent to the Proxy Service.

    Secure-path Establishment: A secure egress path is established for the Sandbox where the Snowpark External Access UDx will be executed.

4. **Execution and Traffic Routing:** Once the UDx starts executing, all egress traffic is securely routed through the established egress path to the Proxy Service, allowing it to reach the designated remote service.

Snowpark External Access is particularly useful for:

1. **AI/ML**: Integrate with external machine learning systems.
2. **Applications**: Connect to external services beyond Snowflake and deploy as a Native App.
3. **Data engineering**: Data ingestion and enrichment from a variety of external sources.

AUTHOR

Niranjan Sharma

Building Data Pipelines With Snowpark External Access

SHARE

Efficient network usage is vital for optimizing UDx performance. This relies on two key factors: bandwidth and concurrency.

Bandwidth: Snowflake manages bandwidth allocation to prevent excessive resource consumption by Sandbox traffic, enabling a balanced distribution between core and Sandbox traffic. This effective management maintains responsive and high-performing data workflows.

Concurrency: Concurrency impacts UDx throughput by allowing multiple processing tasks to run simultaneously. Fine-tuning concurrency can improve both the efficiency and performance of user workloads.

With Snowpark External Access, Snowflake handles bandwidth management, while the users retain the ability to tune the concurrency of UDxs or stored procedures to achieve optimal workload parallelization. This combined approach enables your data pipelines to operate efficiently, delivering robust performance and reliability. Let's explore how to best tune our UDx concurrency to achieve optimal parallelization.

Understanding Concurrency in Snowpark External Access

The concurrency of external network access in Snowflake is influenced by multiple factors and controlled in a layered manner:

1. Warehouse size: The compute capacity directly affects parallelism. Larger warehouses offer more compute power, enabling higher

concurrency.

2. **Snowpark concurrency**: This refers to the parallel execution of the UDx handler function, determining how many rows (or batches, or partitions in the case of vectorized and table functions) can be processed concurrently.

AUTHOR

Niranjan Sharma

3. **User implementation**: This is where we'll focus in this post, as it is entirely within the user's control.

We aim to achieve optimal concurrency at the user-implementation level. This optimized performance will then scale with warehouse size, provided that:

- The remote service can handle the increased load
- The underlying data source has enough rows to process across all ·h nodes

SHARE

It is important to note that the remote service must be able to handle the scale of concurrent requests; this is outside the scope of Snowflake/Snowpark.

Let's run an experiment with a million-row data set, using:

- An XS warehouse as our baseline
- A data source table with one million rows
- A remote API capable of handling the load

The number of concurrent connections created directly impacts the requests per second (RPS) rate. However, the actual RPS also depends on how long the remote service takes to process each request.

By optimizing at the user implementation level, we can establish a strong foundation for performance. As you scale up to larger warehouses, you'll see multiplicative improvements in concurrency and throughput, assuming your data volume and remote service capacity can keep pace. If the remote service that is accepting the request gets throttled due to the large number of requests per second, then you will receive a 429 error.

```
CREATE OR REPLACE TABLE one_million_rows (concurrer
INSERT INTO one_million_rows
SELECT SEQ4() AS concurrency_test
FROM TABLE(GENERATOR(ROWCOUNT => 1000000));
```

We'll use a simple Python UDF to interact with a remote API:

```
import requests
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

session = requests.Session()
retry_strategy = Retry(
    total=5,
    backoff_factor=1,
    status_forcelist=[500, 502, 503, 504],
    allowed_methods=["POST"],
    raise_on_status=False
)
session.mount("https://", HTTPAdapter(max_retries=r

def trigger_remote_api(concurrency_test):
    response = session.post(url, json={"data": conc
    response.raise_for_status()
    return response.text
```

AUTHOR

Niranjan Sharma

SHARE

This configuration establishes a TCP connection as a global state shared across UDF handler invocations, making it essential to manage scenarios like connection resets or drops. We'll be processing a table with one million rows using an XS warehouse.

Query-execution overview

In this process, we invoke a UDF handler for each row. The execution is parallelized, leveraging both the parallel execution of the UDF handler function factor (denoted as *S*) and the warehouse's parallelism (denoted as *W*).

> **Concurrency calculation:**The total query execution concurrency can be represented as

> **Execution-time estimation:** To estimate the execution time, consider that each remote API call takes *N* seconds. The total time required to process 1 million rows would be

Enhancing concurrency

To improve overall concurrency, we can employ two powerful approaches:

1. **Vectorized Python UDFs:** These UDFs process batches of input rows as pandas DataFrames, returning results as pandas arrays or series. This approach significantly reduces the overhead of function calls.

2. **Connection pooling:** Instead of a single connection, we create a pool of TCP connections for parallel processing.

AUTHOR

Niranjan Sharma

The following code example demonstrates:

Use of a session pool for efficient connection management

Implementation of retry logic for improved reliability

Vectorized processing to handle batches of data

SHARE

Concurrent execution using ThreadPoolExecutor

```
import pandas as pd
import requests
import math
from concurrent.futures import ThreadPoolExecutor
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry
from _snowflake import vectorized

# Configuration
SESSION_POOL_SIZE = 50
MAX_BATCH_SIZE = 1000
URL = "remote_service_endpoints"

# Network requests should have retries
retry_strategy = Retry(
    total=5,
    backoff_factor=1,
    status_forcelist=[500, 502, 503, 504],
    allowed_methods=["POST"],
    raise_on_status=False
)
adapter = HTTPAdapter(max_retries=retry_strategy)
session_pool = [requests.Session() for _ in range(S
for session in session_pool:
    session.mount("https://", adapter)
```

AUTHOR

Niranjan Sharma

SHARE

```
# Makes the network call
def make_request(session, data):
    payload = {"concurrency_test": int(data)}
    try:
        response = session.post(URL, json=payload)
        response.raise_for_status()
        return response.text
    except requests.RequestException as e:
        return f"Error: {e}"
# Process each chunk
def process_chunk(chunk, session):
    return [make_request(session, data) for data ir

# Vectorized python UDF
@vectorized(input=pd.DataFrame, max_batch_size=MAX_
def trigger_remote_api(df):
    chunk_size = math.ceil(len(data) / SESSION_POOL
    chunks = [(df[0][i:i + chunk_size], session_poo

    results = []
    with ThreadPoolExecutor(max_workers=SESSION_POO
        futures = [executor.submit(process_chunk, c
        for future in futures:
            results.extend(future.result())

    return pd.Series(results)
```

Query-execution overview

This approach implements a vectorized UDF with a batch size of 1,000 and utilizes a pool of 50 TCP connections. We employ a global connection pool to prevent the creation of new connections for each handler invocation. This strategy is necessary because Snowflake imposes limits on the number of TCP connections to safeguard the underlying network resources.

Each UDF invocation processes 1,000 rows, distributed across 50 connections (approximately 20 rows per connection). This results in 1,000 handler invocations per UDF call. The execution is parallelized by utilizing the UDF handler's parallel execution factor (denoted as *S*), the warehouse's parallelism (denoted as *W*), and a pool of 50 TCP connections per UDF handler.

Concurrency calculation

The total query execution concurrency is calculated as: *S × W × 50*

Execution-time estimation

With the increased concurrency factor of 50, execution time improves significantly, though this comes with added complexity and slight overhead.

AUTHOR

Niranjan Sharma

When architecting parallel network connections to optimal throughput, you should consider:

1. **Batch size and TCP connections:** The batch size and TCP connections can be adjusted based on the remote service you're accessing. It's important to note that if the remote service takes significantly longer to respond, you may encounter UDF timeouts.

2. **Use case dependency:** There's no one-size-fits-all approach to m performance, as it largely depends on your specific use case. However, as a general guideline, we recommend keeping the execution of the UDF handler under one minute. If it takes longer, consider processing fewer rows per invocation.

SHARE

Conclusion

While warehouse size and load affect performance, the implementation of the user handler code plays a crucial role in achieving optimal concurrency. Performance optimization is a shared responsibility between Snowflake and the user's code implementation.

To achieve the best performance, we recommend that you:

1. Select the features that best suit your use as Snowflake provides a comprehensive, unified platform.

2. Utilize the programming language's capabilities and libraries to optimize code further.

Libraries like `asyncio` and `httpx` can be used to implement asynchronous tasks, potentially improving performance. Furthermore, if the remote service supports it, HTTP/2 multiplexing can be utilized for even better efficiency.

Looking Forward

Improving performance and concurrency is an ongoing process. Snowflake continually enhances its data platform, while users are responsible for optimizing their code to best utilize Snowflake's platform and language capabilities. By focusing on both aspects, user can achieve better overall performance in Snowpark External Access scenarios.

AUTHOR

Niranjan Sharma

SHARE

RELATED CONTENT

2024년 09월 05일

Model Hotswapping: Optimizing AI Infrastructure and Enhancing LLM Efficiency

At Snowflake, we support customer AI workloads by offering a diverse range of open source and proprietary large language models (LLMs). As we expand our first-party product offerings in Snowflake...

Delve into the details

2024년 08월 20일

Secure Connections with New Outbound Private Link with Snowflake Support in Preview

For various data engineering, AI and ML workloads, customers need to connect to external systems...

Full Details

2024년 08월 06일

Accelerating Hyperparameter Tuning in Snowflake ML

Hyperparameter optimization (HPO) is a frequently used machine learning technique to select the best parameter...

Full Details

START YOUR
30-DAY FREE TRIAL

START NOW

AUTHOR

Inc.

Niranjan Sharma

플랫폼 개요

아키텍처

데이터 애플리케이션

데이터 마켓플레이스

SNOWFLAKE 파트너 네트워크

지원 및 서비스

회사

문의하기

SHARE

Sign up for Snowflake Communications

diana.shaw@snow

United States

By submitting this form, I understand Snowflake will process my personal information in accordance with their **Privacy Notice**. Additionally, I consent to my information being shared with Event Partners in accordance with Snowflake's **Event Privacy Notice**. I understand I may withdraw my consent or update my preferences **here** at any time.

SUBSCRIBE NOW

Privacy Notice | Site Terms | Cookie Settings