



Snowpark for Snowflake

Complete Interview Study Guide

Python · Java · Scala · DataFrames · UDFs · ML Pipelines · Optimization

[All Snowpark APIs](#)[Interview Q&A](#)[Code Examples](#)[Best Practices](#)

Table of Contents

Chapter 1: Introduction to Snowpark

- What is Snowpark?
- Why Snowpark?
- Supported Languages
- Key Architecture

Chapter 2: Snowpark Setup & Session

- Installation
- Session Creation
- Connection Options
- Session Properties

Chapter 3: DataFrame API Deep Dive

- Creating DataFrames
- Transformations
- Actions
- Schema Inference
- Lazy Evaluation

Chapter 4: Column Expressions & Functions

- Built-in Functions
- Column Operations
- Window Functions
- Conditional Logic

Chapter 5: User-Defined Functions (UDFs)

- Scalar UDFs
- Vectorized UDFs
- UDTFs
- Permanent vs Temporary
- Dependencies

Chapter 6: Stored Procedures

- Snowpark Stored Procs
- Caller vs Owner Rights
- Exception Handling
- Logging

Chapter 7: Data Loading & Unloading

- Reading from Stages
- Writing to Tables
- File Formats
- COPY INTO

Chapter 8: Snowpark ML & ML Pipelines

- Feature Engineering
- ModelRegistry
- Preprocessing
- Training & Inference

Chapter 9: Performance & Optimization

- Query Pushdown
- Caching
- Partitioning
- Pruning
- Profiling

Chapter 10: Advanced Topics

- Dynamic Tables
- Snowpark Container Services
- Async Execution
- Streaming

Chapter 11: Interview Q&A; Bank

- Conceptual Questions
- Coding Scenarios
- Architecture Questions
- Tricky Questions

Chapter 12: Cheat Sheet & Quick Reference

- API Summary
- Common Patterns
- Error Reference

CHAPTER 1

Introduction to Snowpark

1.1 What is Snowpark?

Snowpark is Snowflake's developer framework that allows data engineers, data scientists, and developers to write code in their native programming language — Python, Java, or Scala — and execute it natively inside Snowflake's compute engine. Unlike traditional approaches where data must be pulled out of Snowflake for processing, Snowpark pushes computation directly into the Snowflake cloud data platform, eliminating data movement and leveraging Snowflake's massively parallel processing (MPP) engine.

Snowpark was introduced to bridge the gap between SQL-centric Snowflake operations and the rich ecosystems of Python (pandas, scikit-learn, PyTorch), Java, and Scala. It exposes a DataFrame API that should feel familiar to Apache Spark developers while running entirely on Snowflake's infrastructure — no Spark cluster required.

1.2 Why Snowpark? — The Core Value Proposition

Challenge (Without Snowpark)	Solution (With Snowpark)
Data must leave Snowflake for Python processing	Python runs inside Snowflake — zero data movement
ETL pipelines require separate Spark/Databricks cluster	Single platform: storage + compute + orchestration
SQL alone cannot express complex ML preprocessing	Full Python/Java/Scala expressiveness in-platform
Security & governance gaps between systems	Unified RBAC, data masking, lineage across all workloads
Operational overhead of managing clusters	Serverless execution on Snowflake virtual warehouses
Cost of egress & duplicate storage	Process data where it lives — no egress fees

1.3 Supported Languages & Client Libraries

Snowpark currently supports three languages, each with a distinct client library:

Language	Library / Package	Minimum Version	Execution Environment
Python	snowflake-snowpark-python	1.0.0+	Python 3.8, 3.9, 3.10, 3.11
Java	snowflake-snowpark-java	1.0.0+	JVM 11+
Scala	snowflake-snowpark-scala	1.0.0+	Scala 2.12, JVM 11+

1.4 Architecture Overview

Understanding Snowpark's architecture is critical for interviews. The key insight is that **Snowpark code is translated into SQL**, which then runs on Snowflake's query engine. Here is the execution flow:

- **1. Developer writes code:** Python/Java/Scala code using the DataFrame API or UDF registration on the client.
- **2. Plan construction:** Snowpark builds a logical plan (DAG of transformations) locally — no data fetched yet.
- **3. SQL generation:** When an action is triggered (`.collect()`, `.show()`, `.write()`), Snowpark translates the plan to SQL.
- **4. SQL submission:** The generated SQL is sent to Snowflake via the driver (JDBC/Arrow-based transport).
- **5. Query engine execution:** Snowflake's MPP engine executes the SQL across the virtual warehouse nodes.
- **6. Result return:** Results are streamed back to the client or written to a Snowflake table/stage.

★ KEY INTERVIEW POINT

Snowpark does NOT use Apache Spark. It is a completely independent framework. The DataFrame API surface is similar to Spark by design, but the execution engine is Snowflake's own SQL engine. Candidates who confuse the two will be marked down.

1.5 Snowpark vs. Traditional Approaches — Interview Comparison

Aspect	JDBC/ODBC + Python	Apache Spark	Snowpark
Data Movement	Pull entire dataset	Pull or sync data to HDFS/S3	Zero — compute goes to data
Language	Any + SQL strings	Python/Scala/Java/R	Python/Java/Scala
Infrastructure	Separate servers	Separate Spark cluster	Snowflake VW only
Governance	Fragmented	Fragmented	Unified Snowflake RBAC
SQL Generation	Manual	Catalyst optimizer → Spark SQL	Automatic (Snowpark → SQL)
Cost Model	Variable cluster costs	Per-cluster-hour	Per-credit Snowflake
Learning Curve	Low (familiar)	Medium-High	Low (Spark-like API)

CHAPTER 2

Snowpark Setup & Session

2.1 Installation

Install the Snowpark Python library via pip. Always pin to a specific version in production:

```
# Installation
# Standard install
pip install snowflake-snowpark-python

# With optional extras (for ML workloads)
pip install 'snowflake-snowpark-python[pandas]'

# Verify installation
python -c "import snowflake.snowpark; print(snowflake.snowpark.__version__)"
```

2.2 Creating a Session

The **Session** object is the entry point to all Snowpark operations. It encapsulates the connection to Snowflake and tracks all state. There is always exactly one active Session per program thread in typical usage.

```
# Session Creation — Three Methods

from snowflake.snowpark import Session

# Method 1: Dictionary of connection parameters
connection_params = {
    'account': 'your account identifier', # e.g. xy12345.us-east-1
    'user': 'your username',
    'password': 'your password', # prefer key-pair in prod
    'role': 'SYSADMIN',
    'warehouse': 'COMPUTE WH',
    'database': 'MY DB',
    'schema': 'PUBLIC',
}

session = Session.builder.configs(connection_params).create()

# Method 2: Using a named connection from ~/.snowflake/config.toml
session = Session.builder.config('connection name', 'myconn').create()

# Method 3: From environment variables
import os
session = Session.builder.configs({
    'account': os.environ['SNOWFLAKE_ACCOUNT'],
    'user': os.environ['SNOWFLAKE_USER'],
    'password': os.environ['SNOWFLAKE_PASSWORD'],
}).create()

# Always close the session when done
session.close()
```

2.3 Key-Pair Authentication (Production Best Practice)

```
# Key-Pair Auth (Recommended for Production)
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization

with open('/path/to/rsa_key.p8', 'rb') as key_file:
    private_key = serialization.load_pem_private_key(
        key_file.read(),
        password=b'your passphrase',
        backend=default_backend()
    )

pkb = private_key.private_bytes(
    encoding=serialization.Encoding.DER,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption()
)

session = Session.builder.configs({
    'account': 'xv12345.us-east-1',
    'user': 'svc_account',
    'private_key': pkb,
    'role': 'DATA_ENGINEER',
    'warehouse': 'ETL_WH',
    'database': 'PROD_DB',
    'schema': 'ETL',
}).create()
```

2.4 Session Properties & Configuration

After creating a session, you can inspect and modify its properties at runtime:

```
# Session Management

# Inspect current settings
print(session.get_current_database())
print(session.get_current_schema())
print(session.get_current_role())
print(session.get_current_warehouse())

# Change context mid-session
session.use_database('ANALYTICS_DB')
session.use_schema('REPORTING')
session.use_warehouse('REPORTING_WH')
session.use_role('ANALYST')

# Set query tags for monitoring
session.query_tag = 'snowpark_etl_pipeline_v2'

# Enable auto-commit (default True in Snowpark)
session.sql('ALTER SESSION SET AUTOCOMMIT = TRUE').collect()
```

```
# Get Snowflake version  
print(session.sql('SELECT CURRENT_VERSION()').collect())
```

■ INTERVIEW GOTCHA

Q: How many Sessions can you have simultaneously?

A: Snowpark Python allows multiple Sessions in the same program (they use separate connections).

However, inside a Stored Procedure running in Snowflake, only ONE session exists — it is automatically provided as the first argument and you cannot create additional sessions.

CHAPTER 3

DataFrame API Deep Dive

3.1 What is a Snowpark DataFrame?

A Snowpark DataFrame is a **lazily evaluated, distributed representation of tabular data**. It is conceptually identical to a Spark DataFrame or a pandas DataFrame, but the computation always happens inside Snowflake. A DataFrame does not hold any data until an *action* is called — it simply builds a logical execution plan.

3.2 Creating DataFrames — All Methods

```
# Creating DataFrames
from snowflake.snowpark.functions import col, lit
from snowflake.snowpark.types import IntegerType, StringType, StructField, StructType

# --- From a Snowflake table ---
df_table = session.table('SALES.PUBLIC.ORDERS')

# --- From SQL string ---
df_sql = session.sql('SELECT order id, amount FROM orders WHERE status = \"CLOSED\"')

# --- From Python list (for small datasets / testing) ---
data = [(1, 'Alice', 5000), (2, 'Bob', 3200)]
df_list = session.create_dataframe(data, schema=['id', 'name', 'salary'])

# --- With explicit schema ---
schema = StructType([
    StructField('id', IntegerType(), nullable=False),
    StructField('name', StringType(), nullable=True),
    StructField('salary', IntegerType(), nullable=True),
])
df_typed = session.create_dataframe(data, schema=schema)

# --- From a stage file (CSV) ---
df_stage = session.read.option('header', True) \
    .option('infer schema', True) \
    .csv('@MY_STAGE/data/')

# --- From a Parquet file ---
df_parquet = session.read.parquet('@MY_STAGE/parquet data/')

# --- Range (like Spark) ---
df_range = session.range(0, 100, 2) # 0,2,4,...,98
```

3.3 Transformations (Lazy)

Transformations return a new DataFrame and do NOT trigger execution. Snowpark builds a logical plan internally. Common transformations:

```
# Common Transformations
```

```

from snowflake.snowpark.functions import col, upper, sum as sum , avg, count, when
df = session.table('ORDERS')
# SELECT – column selection & renaming
df2 = df.select('order id', 'customer id', col('amount').alias('order amount'))
# FILTER / WHERE
df3 = df.filter(col('status') == 'CLOSED')
df4 = df.where((col('amount') > 1000) & (col('region') == 'EMEA'))
# WITHCOLUMN – add or replace a column
df5 = df.with column('tax', col('amount') * lit(0.18))
# WITHCOLUMNRENAMED
df6 = df.with column renamed('order ts', 'order timestamp')
# DROP columns
df7 = df.drop('internal flag', 'temp col')
# GROUPBY + AGGREGATION
df agg = df.group by('region').agg(
    sum ('amount').alias('total sales'),
    avg('amount').alias('avg order'),
    count('order id').alias('num orders')
)
# ORDER BY
df sorted = df agg.sort(col('total sales').desc())
# LIMIT
df top10 = df sorted.limit(10)
# JOIN
customers = session.table('CUSTOMERS')
df joined = df.join(customers, df['customer id'] == customers['cust id'], 'left')
# UNION / UNION ALL
df union = df.union(session.table('ORDERS ARCHIVE'))
df union all = df.union all(session.table('ORDERS ARCHIVE'))
# DISTINCT
df distinct = df.select('customer id').distinct()
# FLATTEN (for semi-structured data)
df_flat = df.join_table_function('flatten', col('json_column')['items'])

```

3.4 Actions (Eager — Trigger Execution)

Actions send the query to Snowflake and return results. Each action generates and executes a SQL statement.

```

# Actions – Triggering Execution
# collect() – returns list of Row objects (pulls all data to driver!)
rows = df.collect()
for row in rows:
    print(row['order id'], row['amount'])
# show() – prints tabular output to console (default 10 rows)

```

```

df.show()           # 10 rows
df.show(50)         # 50 rows

# to_pandas() – convert to pandas DataFrame (careful with large data!)
pandas df = df.to_pandas()

# to_pandas_batches() – iterator for large result sets
for batch in df.to_pandas_batches():
    process(batch)

# count() – row count without returning data
n = df.count()

# first() – first N rows
first row = df.first()      # 1 row as Row object
first five = df.first(5)    # list of 5 Row objects

# write – persist to table (see Chapter 7 for full details)
df.write.mode('overwrite').save_as_table('MY_DB.PUBLIC.RESULT_TABLE')

# copy into table – use COPY INTO for high-performance bulk writes
df.write.copy_into_location('@MY_STAGE/output/', file_format_type='csv')

```

3.5 Lazy Evaluation — Deep Dive

Snowpark's lazy evaluation is identical in concept to Apache Spark and SQL query optimizers. Understanding this thoroughly is expected at a senior level:

- **Plan accumulation:** Each transformation appends to an internal logical plan tree (DAG).
- **No data movement:** The driver machine processes zero data during transformations.
- **Optimizer hints:** Snowflake's query optimizer can reorder, push down filters, and prune columns from the generated SQL.
- **Debugging:** Use `df.queries` to inspect the generated SQL before execution.
- **Caching:** Unlike Spark, Snowpark DataFrames do NOT have an explicit `.cache()` method. Snowflake's result cache serves this role.

```

# Debugging Lazy Evaluation

# Inspect generated SQL without executing
print(df.queries)

# Returns: {'queries': ['SELECT ...'], 'post actions': []}

# Explain plan (logical + physical)
df.explain()

```

3.6 Schema Handling

```

# Schema Inspection

# Print schema
df.schema.print_tree()

# Access as StructType
schema: StructType = df.schema

for field in schema.fields:
    print(field.name, field.datatype, field.nullable)

```

```
# Snowpark type mapping
from snowflake.snowpark.types import (
    LongType, DoubleType, DecimalType, BooleanType,
    DateType, TimestampType, StringType, BinaryType,
    ArrayType, MapType, VariantType, GeographyType
)
```

CHAPTER 4

Column Expressions & Functions

4.1 Column Object

Every expression in a DataFrame transformation produces a Column object:

```
# Column Expressions
from snowflake.snowpark.functions import col, lit

# Three ways to reference a column
c1 = col('amount')          # function form
c2 = df['amount']           # subscript on DataFrame
c3 = df.amount              # attribute access

# Literal value
c4 = lit(42)                # integer literal
c5 = lit('ACTIVE')          # string literal
c6 = lit(None)              # NULL

# Column arithmetic
tax_col = col('amount') * lit(0.18)
total_col = col('amount') + col('tax')
diff_col = col('budget') - col('actual')
ratio_col = col('actual') / col('budget')

# Column comparison operators
mask1 = col('status') == 'ACTIVE'
mask2 = col('amount') > 1000
mask3 = col('amount').between(500, 5000)
mask4 = col('name').like('%Corp%')
mask5 = col('region').isin(['EMEA', 'APAC'])
mask6 = col('note').is null()
mask7 = col('note').is not null()

# Boolean combinations
combined = mask1 & mask2      # AND
either = mask1 | mask2       # OR
negated = ~mask1             # NOT

# Alias
renamed = col('amount').alias('order value')

# Cast
from snowflake.snowpark.types import DoubleType
casted = col('str_amount').cast(DoubleType())
```

4.2 Built-in Functions

Snowpark exposes all Snowflake SQL functions through `snowflake.snowpark.functions`:

```
# Built-in Function Examples
```

```

from snowflake.snowpark import functions as F

# --- String ---
F.upper(col('name')), F.lower(col('name')), F.trim(col('name'))
F.substr(col('code'), 1, 3)          # 1-based like SQL
F.concat(col('first'), lit(' '), col('last'))
F.regexp_replace(col('phone'), r'^[0-9]', '')
F.split(col('csv field'), lit(','))

# --- Numeric ---
F.round(col('amount'), 2)
F.abs(col('diff'))
F.floor(col('score')), F.ceil(col('score'))
F.pow(col('base'), lit(2))
F.sqrt(col('variance'))
F.log(lit(10), col('value'))        # log10

# --- Date / Time ---
F.current_date(), F.current_timestamp()
F.dateadd('day', lit(30), col('order_date'))
F.datediff('day', col('start'), col('end_date'))
F.date_trunc('month', col('ts'))
F.year(col('ts')), F.month(col('ts')), F.dayofweek(col('ts'))
F.to_date(col('str date'), lit('YYYY-MM-DD'))

# --- Aggregate ---
F.sum(col('amount')), F.avg(col('amount'))
F.min(col('amount')), F.max(col('amount'))
F.count(col('id')), F.count_distinct(col('customer id'))
F.stddev(col('score')), F.variance(col('score'))
F.listagg(col('tag'), lit(',')).within_group(col('tag').asc())

# --- Conditional ---
F.iff(col('amount') > 1000, lit('HIGH'), lit('LOW'))
F.coalesce(col('phone'), col('mobile'), lit('N/A'))
F.nullif(col('code'), lit(''))
F.nvl(col('discount'), lit(0))

# --- Array / Object (semi-structured) ---
F.array_construct(lit(1), lit(2), lit(3))
F.array_size(col('tags'))
F.array_contains(lit('VIP'), col('tags'))
F.object_construct(lit('k'), col('v'))
F.get_path(col('variant col'), lit('address.city'))
F.parse_ison(col('raw_ison'))
F.to_variant(col('amount'))

```

4.3 Window Functions

Window functions apply aggregate or ranking operations over a sliding window of rows:

```
# Window Functions
```

```
from snowflake.snowpark import Window
from snowflake.snowpark.functions import row_number, rank, dense_rank, lag, lead, sum as
    sum
# Define a window spec
w = Window.partition by('region').order by(col('order date')).desc()
# Row number within each region (ordered by date)
df2 = df.with_column('rn', row_number().over(w))
df2 = df2.with_column('rnk', rank().over(w))
df2 = df2.with_column('drk', dense_rank().over(w))
# Lag / Lead
df2 = df2.with_column('prev amount', lag(col('amount'), 1).over(w))
df2 = df2.with_column('next amount', lead(col('amount'), 1).over(w))
# Running total (rows between unbounded preceding and current row)
w running = Window.partition by('region').order by('order date') \
    .rows between(Window.UNBOUNDED PRECEDING, Window.CURRENT ROW)
df2 = df2.with_column('running total', sum ('amount').over(w running))
# Moving average (last 3 rows)
w moving = Window.partition by('region').order by('order date') \
    .rows between(-2, Window.CURRENT ROW)
from snowflake.snowpark.functions import avg
df2 = df2.with_column('moving avg3', avg('amount').over(w moving))
# Get latest record per group (common deduplication pattern)
df_deduped = df2.filter(col('rn') == 1)
```

CHAPTER 5

User-Defined Functions (UDFs)

5.1 UDF Types Overview

Type	Cardinality	Execution	Best For
Scalar UDF	1 row → 1 value	Row-by-row in Snowflake	Simple transformations, lookups
Vectorized UDF	N rows → N values	Pandas batches (Arrow)	NumPy/Pandas operations, ML inference
UDTF (Table Function)	1 row → N rows	Row-by-row in Snowflake	Exploding arrays, parsing, expansion
Aggregate UDF	N rows → 1 value	Custom accumulator	Custom aggregations

5.2 Scalar UDFs

```
# Scalar UDF Registration
from snowflake.snowpark.functions import udf
from snowflake.snowpark.types import StringType, IntegerType

# --- Inline (anonymous) UDF ---
double_it = udf(lambda x: x * 2, return_type=IntegerType(),
                input_types=[IntegerType()])
df.with_column('doubled', double_it(col('amount')))

# --- Decorator style (named, registered in Snowflake) ---
@udf(name='classify_customer',
     return_type=StringType(),
     input_types=[IntegerType()],
     is_permanent=True,
     stage_location='@ML_STAGE',
     replace=True,
     session=session)
def classify_customer(revenue: int) -> str:
    if revenue >= 100_000:
        return 'PLATINUM'
    elif revenue >= 50_000:
        return 'GOLD'
    elif revenue >= 10_000:
        return 'SILVER'
    return 'BRONZE'

# Use it like a column function
df.with_column('tier', classify_customer(col('annual_revenue')))

# Register existing function without decorator
def parse_icd_code(code: str) -> str:
    return code.split('.')[0] if code else None

icd_udf = session.udf.register(
```

```

    parse icd code,
    name='PARSE ICD',
    return type=StringType(),
    input types=[StringType()],
    replace=True
)

```

5.3 Vectorized UDFs (Pandas UDFs)

Vectorized UDFs receive a pandas Series or DataFrame as input instead of individual rows. This is dramatically faster because Snowflake passes data in Arrow format batches, eliminating per-row Python overhead. Ideal for ML inference, NumPy operations, or regex.

```

# Vectorized (Pandas) UDFs
import pandas as pd
from snowflake.snowpark.functions import pandas_udf
from snowflake.snowpark.types import PandasSeries, PandasDataFrame

# Single-column input → single-column output
@pandas_udf(name='normalize score',
            return type=PandasSeries[float],
            input types=[PandasSeries[float]],
            session=session,
            replace=True)

def normalize_score(scores: pd.Series) -> pd.Series:
    # Entire batch processed as a numpy array – very fast
    return (scores - scores.min()) / (scores.max() - scores.min())

# Multi-column input using PandasDataFrame
@pandas_udf(name='predict churn',
            return type=PandasSeries[float],
            input types=[PandasDataFrame[float, float, int]],
            session=session,
            replace=True,
            packages=['scikit-learn'],
            imports=['@ML_STAGE/model.pkl'])

def predict_churn(df: pd.DataFrame) -> pd.Series:
    import joblib, sys
    model = joblib.load(sys.options.get('snowflake import directory') + 'model.pkl')
    return pd.Series(model.predict_proba(df.values)[: , 1])

```

5.4 User-Defined Table Functions (UDTFs)

```

# UDTF – Table Function
from snowflake.snowpark.functions import udtf
from snowflake.snowpark.types import StructType, StructField, StringType, IntegerType

@udtf(name='split_tags',
      output_schema=StructType([
          StructField('tag', StringType()),

```

```

        StructField('position', IntegerType()),
    ),
    input_types=[StringType()],
    session=session,
    replace=True)

class SplitTags:
    def process(self, tag_string: str):
        # yield one tuple per output row
        for i, tag in enumerate(tag_string.split(',')):
            yield (tag.strip(), i)

# UDTFs are called as table functions (join table function)
df_tags = session.table('PRODUCTS').join table function(
    split_tags(col('tags'))    # expands each row to multiple rows
)

```

5.5 UDF Dependencies — Packages & Imports

UDFs run inside Snowflake's sandboxed Python environment. You must explicitly declare any third-party packages and custom file imports:

```

# UDF Packages & Imports

# Packages from Snowflake's Anaconda channel (zero upload required)
@udf(packages=['pandas', 'scikit-learn==1.3.0', 'numpy', 'xgboost'],
    ...)
def train_model(...):
    from sklearn.ensemble import GradientBoostingClassifier
    ...

# Upload a custom local module to a stage, then import it
session.add_import('/local/path/mymodule.py')
session.add_import('/local/path/model.pkl')

# Or upload directly to a named stage
session.file.put('/local/path/utls.py', '@CODE_STAGE', auto_compress=False)
@udf(imports=['@CODE_STAGE/utls.py', '@CODE_STAGE/model.pkl'],
    packages=['xgboost'],
    ...)
def score(features):
    import sys
    import_dir = sys._xoptions['snowflake_import_directory']
    import utls # available because of imports=
    ...

```

5.6 Permanent vs Temporary UDFs

Aspect	Temporary UDF	Permanent UDF
Scope	Current session only	Registered in Snowflake catalog

Aspect	Temporary UDF	Permanent UDF
Persistence	Dropped at session end	Survives indefinitely
Stage req.	Not required	stage_location required to store code
Use case	Ad-hoc, development	Production pipelines, shared teams
Call syntax	DataFrame API only	SQL + DataFrame API

CHAPTER 6

Snowpark Stored Procedures

6.1 What Makes Snowpark Stored Procs Different?

Traditional Snowflake stored procedures use JavaScript. Snowpark stored procedures extend this to Python, Java, and Scala, allowing full use of the DataFrame API, UDFs, and Snowpark ML inside a server-side procedure that runs entirely in Snowflake — no client needed.

Aspect	JavaScript Stored Proc	Snowpark Python Stored Proc
Language	JavaScript	Python (+ Java, Scala)
DataFrame API	Not available	Full Snowpark DataFrame API
ML Libraries	None	scikit-learn, XGBoost, PyTorch (via packages)
SQL Execution	snowflake.execute()	session.sql() or DataFrame actions
Ecosystem	Limited	Full Python ecosystem

6.2 Writing & Registering a Stored Procedure

```
# Stored Procedure — Registration & Call
```

```
from snowflake.snowpark import Session
from snowflake.snowpark.functions import sproc

# The first argument is ALWAYS the Session object (injected by Snowflake)
def refresh_daily_summary(session: Session, run_date: str) -> str:
    """Rebuild the DAILY SUMMARY table for a given date."""
    # Use DataFrame API
    orders = session.table('ORDERS').filter(
        session.sql(f"SELECT * FROM ORDERS WHERE DATE(order ts) = '{run_date}'").collect()
    )
    # Or use session.sql directly
    session.sql(f"""
        INSERT INTO DAILY SUMMARY
        SELECT '{run_date}', region, SUM(amount)
        FROM ORDERS WHERE DATE(order ts) = '{run_date}'
        GROUP BY region
    """).collect()
    return f'Summary built for {run_date}'

# Register as a permanent stored procedure
session.sproc.register(
    refresh_daily_summary,
    name='REFRESH DAILY SUMMARY',
    is_permanent=True,
    stage_location='@PROC_STAGE',
```

```
packages=['snowflake-snowpark-python'],
replace=True,
execute_as='caller'    # or 'owner'
)

# Call it from SQL
# CALL REFRESH DAILY SUMMARY('2024-11-15');

# Call it from Python
result = session.call('REFRESH DAILY SUMMARY', '2024-11-15')
print(result) # 'Summary built for 2024-11-15'
```

6.3 Caller Rights vs. Owner Rights

This is a frequently tested interview concept. The `execute_as` parameter controls whose privileges are used when the procedure runs:

Aspect	<code>execute_as='caller'</code>	<code>execute_as='owner'</code>
Who's privilege used?	The calling user's role	The procedure owner's role
Data access	Caller must have access	Owner's grants apply
Security model	Transparent — caller sees everything proc calls do	Encapsulated — caller needn't have direct table access
Use case	Ad-hoc, user-specific access	Shared ETL, controlled elevation
Default	No — must be explicit	Yes (default if omitted)

6.4 Exception Handling & Logging in Stored Procs

```
# Exception Handling
import logging
from snowflake.snowpark import Session

logger = logging.getLogger('snowpark.proc')

def robust_etl(session: Session, table name: str) -> str:
    try:
        logger.info(f'Starting ETL for {table name}')
        df = session.table(table name)
        count_before = df.count()
        # ... transformations ...
        logger.info(f'Processed {count_before} rows')
        return f'SUCCESS: {count_before} rows processed'
    except Exception as e:
        logger.error(f'ETL failed: {str(e)}')
        raise # Re-raise so Snowflake marks the call as failed

# Enable event table logging in Snowflake to capture logger output
# ALTER PROCEDURE ROBUST_ETL(VARCHAR) SET LOG_LEVEL = INFO;
```

CHAPTER 7

Data Loading & Unloading

7.1 Reading from Stages

```
# Reading Files from Stages
# CSV with header inference
df_csv = session.read \
    .option('header', True) \
    .option('infer schema', True) \
    .option('skip blank lines', True) \
    .csv('@MY_STAGE/input/2024/')

# JSON
df_json = session.read.json('@MY_STAGE/events/')

# Parquet
df_parquet = session.read.parquet('@MY_STAGE/parquet/')

# Avro
df_avro = session.read.avro('@MY_STAGE/avro/')

# ORC
df_orc = session.read.orc('@MY_STAGE/orc/')

# Specify file format object
df_custom = session.read \
    .option('format_name', 'MY_CSV_FORMAT') \
    .csv('@MY_STAGE/data/')
```

7.2 Writing to Tables

```
# Writing DataFrames to Tables & Stages
# Overwrite existing table
df.write.mode('overwrite').save as table('TARGET TABLE')

# Append to existing table
df.write.mode('append').save as table('TARGET TABLE')

# Error if table exists (default)
df.write.mode('errorifexists').save as table('TARGET TABLE')

# Ignore if table exists
df.write.mode('ignore').save as table('TARGET TABLE')

# Qualify with full path
df.write.mode('overwrite').save as table('MY_DB.PROD.DAILY_SUMMARY')

# Write with table clustering
df.write.mode('overwrite') \
    .option('cluster by', ['region', 'order date']) \
    .save as table('CLUSTERED_ORDERS')

# COPY INTO – high-performance unload to stage
```

```
df.write \
    .copy into location('@EXPORT STAGE/output/',
                       file format type='parquet',
                       header=True,
                       overwrite=True)
```

7.3 File Upload / Download

```
# PUT / GET Files
# Upload local file to Snowflake stage
put result = session.file.put(
    '/local/data/customers.csv', # local path
    '@MY STAGE/input/',         # stage path
    auto compress=True,         # GZIP compress
    overwrite=True
)
print(put result) # List of PutResult objects
# Download from stage to local
get result = session.file.get(
    '@MY STAGE/output/report.csv', # stage path
    '/local/downloads/'           # local directory
)
```

CHAPTER 8

Snowpark ML & ML Pipelines

8.1 Snowpark ML Overview

Snowpark ML is a Python library (snowflake-ml-python) built on top of Snowpark that provides scikit-learn-compatible preprocessing, model training, and model registry capabilities — all executing within Snowflake. It eliminates the need to move training data out of Snowflake.

Component	Description	Analogue
Preprocessing	StandardScaler, OrdinalEncoder, OneHotEncoder, etc.	sklearn.preprocessing
Model Training	XGBClassifier, LinearRegression, etc. (optimized for Snowflake)	sklearn estimators
Model Registry	Versioned model storage inside Snowflake	MLflow / SageMaker Registry
Feature Store	Centralized feature definitions with lineage (preview)	Feast / Tecton
ML Pipelines	Pipeline class for chaining transformers + estimator	sklearn.pipeline.Pipeline

8.2 Preprocessing with Snowpark ML

```
# Feature Preprocessing
from snowflake.ml.modeling.preprocessing import StandardScaler, OrdinalEncoder, OneHotEncoder
from snowflake.ml.modeling.pipeline import Pipeline

# Training data as Snowpark DataFrame
train_df = session.table('ML TRAINING DATA')

# StandardScaler
scaler = StandardScaler(input_cols=['AGE', 'SALARY', 'SCORE'],
                        output_cols=['AGE SCALED', 'SAL SCALED', 'SCORE SCALED'])
scaled_df = scaler.fit(train_df).transform(train_df)

# OrdinalEncoder for ordinal categoricals
enc = OrdinalEncoder(input_cols=['EDUCATION_LEVEL'],
                    output_cols=['EDU ENCODED'],
                    categories={'EDUCATION_LEVEL': ['HIGH_SCHOOL', 'BACHELORS', 'MASTERS', 'PHD']})
enc_df = enc.fit(scaled_df).transform(scaled_df)

# OneHotEncoder for nominal categoricals
ohe = OneHotEncoder(input_cols=['REGION'], output_cols=['REGION_OHE'],
                    drop_input_cols=True)
ohe_df = ohe.fit(enc_df).transform(enc_df)
```

8.3 Full ML Pipeline

```
# End-to-End ML Pipeline in Snowflake
```

```

from snowflake.ml.modeling.pipeline import Pipeline
from snowflake.ml.modeling.xgboost import XGBClassifier
from snowflake.ml.modeling.model_selection import train_test_split
from snowflake.ml.modeling.metrics import accuracy_score

# Split data
train, test = train_test_split(session.table('CHURN_FEATURES'),
                               test_size=0.2, random_state=42)

FEATURE_COLS = ['AGE', 'TENURE', 'BALANCE', 'NUM PRODUCTS', 'IS ACTIVE']
LABEL_COL = 'CHURNED'

pipeline = Pipeline(steps=[
    ('scaler', StandardScaler(input_cols=FEATURE_COLS,
                              output_cols=[f'{c} SC' for c in FEATURE_COLS])),
    ('clf', XGBClassifier(
        input_cols=[f'{c} SC' for c in FEATURE_COLS],
        label_cols=[LABEL_COL],
        output_cols=['PREDICTED CHURN'],
        n_estimators=200,
        learning_rate=0.05,
        max_depth=6
    ))
])

# Train – runs inside Snowflake, no data leaves!
pipeline.fit(train)

# Predict on test set
predictions = pipeline.predict(test)
predictions.select(LABEL_COL, 'PREDICTED CHURN').show()

# Evaluate
acc = accuracy_score(df=predictions, y_true_col_names=[LABEL_COL],
                     y_pred_col_names=['PREDICTED CHURN'])
print(f'Accuracy: {acc:.4f}')

```

8.4 Model Registry

```

# Model Registry – Log Version, Deploy
from snowflake.ml.registry import Registry

# Create/connect to registry
registry = Registry(session=session, database_name='ML_DB', schema_name='MODELS')

# Log a trained model
model_ver = registry.log_model(
    pipeline, # trained Pipeline / estimator
    model_name='churn_xgb_v2',
    version_name='v1.0',
    metrics={'accuracy': acc, 'auc': 0.87},
    sample_input_data=train.limit(100), # for schema inference
    comment='XGBoost churn model trained on 03 2024 data'
)

```

```
# List models
for m in registry.get_model('churn_xgb_v2').versions():
    print(m.version name, m.metrics)

# Load and run inference
loaded_model = registry.get_model('churn_xgb_v2').version('v1.0')
inference_df = session.table('NEW_CUSTOMERS')
result = loaded_model.run(inference_df, function_name='predict')
result.show()

# Deploy as a Snowflake function (batch inference)
loaded_model.deploy(
    deployment name='churn predict fn',
    options={'compute pool': 'MY GPU POOL'} # for Snowpark Container Services
)
```

CHAPTER 9

Performance & Optimization

9.1 Query Pushdown — The Most Important Concept

Query pushdown means Snowpark translates the entire DataFrame transformation chain into a single, optimized SQL statement and sends it to Snowflake. The data never leaves Snowflake until an action is called. This is fundamentally different from pulling data to pandas and processing locally.

✓ CORRECT PATTERN — Full pushdown (one SQL statement)

```
df = session.table('ORDERS')
    .filter(col('year') == 2024) # pushed to WHERE clause
    .group_by('region')         # pushed to GROUP BY
    .agg(sum_('amount'))        # pushed to SUM()
    .sort(col('sum(amount)').desc())
result = df.collect() # ONE SQL sent to Snowflake
```

✗ ANTI-PATTERN — Breaks pushdown (data leaves Snowflake!)

```
df = session.table('ORDERS').to_pandas() # pulls ALL data
df_filtered = df[df['year'] == 2024]     # local CPU
result = df_filtered.groupby('region')['amount'].sum() # local CPU
```

9.2 Caching Strategies

Snowflake has three layers of caching. Understanding them is critical for performance tuning:

Cache Layer	What Is Cached	Duration	Scope
Result Cache	Full query results	24 hours (resets on data change)	Account-wide
Local Disk Cache	Micro-partition data	LRU per warehouse node	Per-warehouse
Remote Disk Cache	Micro-partition files	Indefinite (S3/Azure/GCS)	Account-wide

For Snowpark: if the same DataFrame plan generates the same SQL query and the underlying data has not changed, Snowflake will return the cached result immediately without consuming any compute credits. This is why setting `session.query_tag` consistently helps identify and leverage cached queries.

9.3 Partitioning, Clustering & Pruning

Snowflake automatically partitions data into micro-partitions (~50-500MB compressed). Micro-partition pruning (eliminating partitions that cannot satisfy a filter predicate) is Snowflake's primary performance mechanism. Snowpark filters are pushed into SQL WHERE clauses, enabling pruning automatically.

- **Cluster Keys:** Define custom clustering on frequently filtered columns using ALTER TABLE ... CLUSTER BY. Snowpark DataFrames benefit automatically when filters match cluster key columns.
- **Automatic Clustering (AC):** Snowflake continuously re-clusters large tables. Works transparently with Snowpark.
- **Filter order matters:** Apply the most selective filters first in your DataFrame chain for maximum pruning.
- **Avoid SELECT *:** Use explicit column selection to enable column-level pruning (Snowflake stores data columnar).

9.4 Performance Best Practices

Anti-Pattern	Better Approach	Why
df.to_pandas() for large data	Keep transformations in Snowpark	Eliminates data movement
Multiple .collect() calls	Combine into one DataFrame chain	Reduces round-trips
UDF for simple string ops	Use built-in F.* functions	Built-ins are SQL — zero serialization
Scalar UDF on large data	Vectorized (pandas) UDF	Batch processing via Arrow
Cartesian join (no condition)	Always specify join condition	Explodes row count
Non-selective first filter	Apply most selective filter first	Maximizes micro-partition pruning
show() in production	write.save_as_table() or collect()	show() has row limit, misleading
Ignore warehouse size	Right-size VW for workload	S → XL → 2XL for heavier transforms

CHAPTER 10

Advanced Topics

10.1 Dynamic Tables

Dynamic Tables are a Snowflake feature that declaratively defines a transformation as a query and automatically refreshes the results on a schedule or triggered by upstream changes. They can be created and queried using Snowpark:

```
# Dynamic Tables with Snowpark
# Create a Dynamic Table via session.sql()
session.sql("""
    CREATE OR REPLACE DYNAMIC TABLE CUSTOMER SUMMARY
        TARGET LAG = '1 hour'
        WAREHOUSE = COMPUTE WH
    AS
    SELECT customer id,
           SUM(amount) AS total spent,
           COUNT(*)    AS num orders,
           MAX(order ts) AS last order
    FROM ORDERS
    GROUP BY customer id
""").collect()

# Query the dynamic table like any other table
df summary = session.table('CUSTOMER SUMMARY')
df summary.filter(col('total spent') > 10000).show()

# Manually refresh
session.sql('ALTER DYNAMIC TABLE CUSTOMER_SUMMARY REFRESH').collect()
```

10.2 Snowpark Container Services (SPCS)

SPCS allows running containerized workloads (Docker containers) inside Snowflake's network boundary. This enables GPU-accelerated ML inference, custom web services, and LLM deployments without data leaving Snowflake.

- **Compute Pools:** Named pools of container nodes (CPU or GPU) managed by Snowflake.
- **Services:** Long-running containers that expose HTTP endpoints.
- **Job Services:** Ephemeral containers that run a task and exit.
- **Integration with Snowpark ML:** Deploy models as SPCS services for real-time inference.
- **Network policy:** All egress/ingress is controlled by Snowflake's network policy.

10.3 Asynchronous Execution

```
# Async Execution
# Fire-and-forget – get an AsyncJob object
```

```
async_job = df.collect(block=False)
# Check if done
if async_job.is_done():
    rows = async_job.result()
else:
    print('Still running...')
# Block until complete (with timeout)
rows = async_job.result(timeout=300) # seconds
# Cancel a running job
async_job.cancel()
# Use case: fire multiple independent queries in parallel
jobs = [
    session.table(t).count(block=False)
    for t in ['ORDERS', 'CUSTOMERS', 'PRODUCTS']
]
counts = [j.result() for j in jobs]
```

10.4 Transactions in Snowpark

```
# Transactions
# Begin transaction
session.sql('BEGIN').collect()
try:
    session.sql("UPDATE ACCOUNTS SET BALANCE = BALANCE - 100 WHERE ID = 1").collect()
    session.sql("UPDATE ACCOUNTS SET BALANCE = BALANCE + 100 WHERE ID = 2").collect()
    session.sql('COMMIT').collect()
except Exception as e:
    session.sql('ROLLBACK').collect()
    raise
```

CHAPTER 11

Interview Q&A Bank

CONCEPTUAL QUESTIONS**Q: What is Snowpark and how does it differ from Apache Spark?**

A: Snowpark is Snowflake's native developer framework for writing Python, Java, or Scala code that executes inside Snowflake's query engine. Unlike Apache Spark, which is an independent distributed compute engine requiring its own cluster, Snowpark translates code to SQL and runs it on Snowflake virtual warehouses. There is no Spark engine involved — the similarity is only in the DataFrame API surface.

Q: Explain lazy evaluation in Snowpark. Why is it important?

A: Lazy evaluation means that transformations on a DataFrame (filter, select, join, etc.) do not execute immediately. Instead, Snowpark builds a logical execution plan (DAG). Execution only happens when an action is called (collect, show, write, count). This is important because: (1) Snowflake's optimizer can see the full query before execution and optimize it; (2) multiple transformations collapse into a single SQL statement, reducing round-trips; (3) no unnecessary data is pulled to the client.

Q: What is the difference between a scalar UDF and a vectorized UDF in Snowpark?

A: A scalar UDF processes one row at a time using individual Python values. A vectorized UDF (pandas UDF) receives a pandas Series or DataFrame representing a batch of rows. Vectorized UDFs are much faster for large datasets because data is transferred in Apache Arrow columnar batches, eliminating per-row Python overhead. Use scalar UDFs for simple logic; use vectorized UDFs for NumPy/Pandas operations or ML inference.

Q: How does Snowpark handle dependencies for UDFs?

A: Snowpark UDFs declare dependencies through two mechanisms: (1) `packages=` parameter for libraries available in Snowflake's Anaconda channel (e.g., pandas, scikit-learn, xgboost); these are resolved by Snowflake without any file upload. (2) `imports=` parameter for custom Python files, serialized models, or any files uploaded to a stage — Snowflake makes them available in the UDF's import directory, accessible via `sys._xoptions['snowflake_import_directory']`.

Q: Explain the `execute_as` parameter in Snowpark stored procedures.

A: `execute_as` controls which user's privileges are used when the stored procedure runs. 'caller' uses the calling user's role — they must have direct access to all objects the proc touches. 'owner' (default) uses the procedure owner's role — the caller can access objects they normally couldn't, as long as the owner has access. Owner rights enable privilege encapsulation: a low-privileged user can call a proc that accesses sensitive tables they can't directly query.

Q: What happens when you call `df.to_pandas()` on a large Snowpark DataFrame?

A: The entire result set is transferred from Snowflake to the Python driver in the client machine's RAM. This is a full data movement operation — exactly what Snowpark is designed to avoid. For large tables, this causes: (1) high memory usage on the client; (2) high network egress costs; (3) slow performance. Use `to_pandas_batches()` for iteration, or keep all processing inside Snowpark and only call `to_pandas()` on small, aggregated results.

Q: What is query pushdown and why does it matter?

A: Query pushdown means Snowpark translates the full chain of DataFrame transformations into a single optimized SQL query that runs entirely inside Snowflake. No intermediate data is materialized on the client. This matters because: (1) Snowflake's MPP engine processes the data in parallel across warehouse nodes; (2) micro-partition pruning, columnar compression, and result caching all apply; (3) the client machine needs minimal CPU and memory.

Q: How does the Snowpark Session relate to a Snowflake connection?

A: A Session is a higher-level abstraction that wraps a Snowflake JDBC connection. It maintains the connection context (database, schema, warehouse, role) and provides the Snowpark API (`session.table()`, `session.sql()`, `session.udf`, etc.). Inside a Stored Procedure running in Snowflake, the Session is automatically injected as the first argument — you cannot create a new Session inside a stored proc.

CODING SCENARIO QUESTIONS

Q: Write a Snowpark pipeline that reads an ORDERS table, calculates a 7-day rolling average of revenue per region, and writes the result to a ROLLING_AVG table.

```
# Answer
from snowflake.snowpark import Session, Window
from snowflake.snowpark.functions import avg, col, to date

def build_rolling_avg(session: Session) -> None:
    df = session.table('ORDERS').select(
        to date(col('order ts')).alias('order date'),
        col('region'),
        col('amount')
    )

    w = (Window
        .partition by('region')
        .order by('order date')
        .rows between(-6, Window.CURRENT ROW)) # 7-day window

    result = df.with column('rolling avg 7d', avg('amount').over(w))
    result.write.mode('overwrite').save as table('ROLLING_AVG')
    print(f'Wrote {result.count()} rows to ROLLING_AVG')
```

Q: Register a permanent vectorized UDF that applies a pre-trained scikit-learn model stored in a stage to score customer records.

```
# Answer
import pandas as pd
from snowflake.snowpark.functions import pandas_udf
from snowflake.snowpark.types import PandasSeries, PandasDataFrame

# Upload model once
session.file.put('model.pkl', '@ML_STAGE', overwrite=True, auto compress=False)

@pandas_udf(
    name='SCORE_CUSTOMERS',
    return_type=PandasSeries[float],
    input_types=[PandasDataFrame[float, float, float]],
    is_permanent=True,
```

```

    stage location='@ML STAGE',
    imports=['@ML STAGE/model.pkl'],
    packages=['scikit-learn', 'joblib'],
    replace=True,
    session=session
)

def score customers(features: pd.DataFrame) -> pd.Series:
    import joblib, sys
    import dir = sys. xoptions['snowflake import directory']
    model = joblib.load(import dir + 'model.pkl')
    return pd.Series(model.predict_proba(features.values)[:, 1])

# Invoke
session.table('CUSTOMERS') \
    .with column('churn score',
        score customers(col('AGE'), col('TENURE'), col('BALANCE')))) \
    .show()

```

RICKY / ADVANCED QUESTION

Q: Can you use a Snowpark UDF inside a SQL query executed via session.sql()?

A: Yes — once a UDF is registered as permanent (is_permanent=True), it becomes a first-class Snowflake function callable from any SQL query, not just Snowpark DataFrames. Example: session.sql('SELECT MY_UDF(col1) FROM mytable').collect(). Temporary UDFs are only available for the current session and can also be called from session.sql() within that session.

Q: What is the difference between df.collect() and df.to_pandas()?

A: collect() returns a list of Snowpark Row objects. to_pandas() returns a pandas DataFrame. Internally both transfer all data to the client. The practical difference is: Row objects use Snowpark's type system; pandas uses numpy dtypes. For large result sets, to_pandas_batches() is preferable as it streams chunks rather than loading all rows into memory at once.

Q: How does Snowpark handle NULL values?

A: NULL handling in Snowpark mirrors SQL: any arithmetic with NULL returns NULL; equality checks with NULL (col == None) return NULL, not True/False. Use col.is_null() for NULL checks. Functions like F.coalesce(), F.nvl(), F.ifnull() handle NULL substitution. When writing schema-aware DataFrames, StructField has a nullable parameter that maps to NOT NULL constraints.

Q: What are the limitations of Snowpark UDFs versus built-in SQL functions?

A: UDFs have higher overhead than built-in functions because: (1) data must be serialized/deserialized between Snowflake's engine and the Python sandbox (scalar UDFs); (2) UDFs run in a restricted sandbox with no network access and limited CPU; (3) UDFs cannot return cursors or multiple result sets. Always prefer built-in F.* functions for standard string, date, and math operations.

Q: How would you debug a slow Snowpark pipeline?

A: Step 1: Print df.queries to see the generated SQL. Step 2: Run EXPLAIN on that SQL in Snowflake to see the query plan. Step 3: Check Query Profile in Snowflake UI for bottlenecks (e.g., large TableScan with no pruning, Hash Join spill to disk). Step 4: Look for anti-patterns — missing cluster key alignment, SELECT *, unnecessary UDFs. Step 5: Consider increasing warehouse size or enabling multi-cluster for concurrency.

Q: Can Snowpark DataFrames be persisted/cached like Spark DataFrames?

A: No — Snowpark does not have an explicit `.cache()` method like Spark. However, you can achieve similar results by: (1) Writing the DataFrame to a temporary table with `.save_as_table()` and reading it back, which leverages Snowflake's columnar storage. (2) Relying on Snowflake's result cache — identical queries with unchanged data return cached results without compute. (3) Using `CREATE TEMPORARY TABLE` for within-session caching.

CHAPTER 12

Cheat Sheet & Quick Reference

12.1 API Quick Reference

Operation	Snowpark API	Notes
Create from table	<code>session.table('NAME')</code>	Lazy
Create from SQL	<code>session.sql('SELECT ...')</code>	Lazy
Create from list	<code>session.create_dataframe(data, schema)</code>	Small data only
Filter rows	<code>df.filter(condition)</code> / <code>df.where(condition)</code>	Lazy
Select columns	<code>df.select(...)</code> / <code>df.drop(...)</code>	Lazy
Add column	<code>df.with_column('name', expr)</code>	Lazy
Rename column	<code>df.with_column_renamed('old', 'new')</code>	Lazy
Join	<code>df.join(other, condition, join_type)</code>	Lazy
Group & aggregate	<code>df.group_by(...).agg(...)</code>	Lazy
Sort	<code>df.sort(col.asc())</code> / <code>df.sort(col.desc())</code>	Lazy
Limit	<code>df.limit(n)</code>	Lazy
Union	<code>df.union(other)</code> / <code>df.union_all(other)</code>	Lazy
Distinct	<code>df.distinct()</code>	Lazy
Show SQL	<code>df.queries</code>	No execution
Show plan	<code>df.explain()</code>	No execution
Execute → list	<code>df.collect()</code>	EAGER
Execute → pandas	<code>df.to_pandas()</code>	EAGER — full data
Count rows	<code>df.count()</code>	EAGER
First N rows	<code>df.first(n)</code>	EAGER
Print to console	<code>df.show(n)</code>	EAGER
Write to table	<code>df.write.mode('...').save_as_table('T')</code>	EAGER
Write to stage	<code>df.write.copy_into_location('@STAGE',...)</code>	EAGER
Register UDF	<code>session.udf.register(fn, name=...)</code>	—
Register UDTF	<code>session.udtf.register(cls, name=...)</code>	—
Register Sproc	<code>session.sproc.register(fn, name=...)</code>	—
Call Sproc	<code>session.call('NAME', arg1, arg2)</code>	—

Operation	Snowpark API	Notes
Upload file	<code>session.file.put(local, stage)</code>	—
Download file	<code>session.file.get(stage, local)</code>	—

12.2 Common Import Patterns

```
# Standard Imports
# Core
from snowflake.snowpark import Session, Window
from snowflake.snowpark import functions as F
from snowflake.snowpark.functions import col, lit, udf, pandas udf, udtf, sproc

# Types
from snowflake.snowpark.types import (
    StructType, StructField,
    StringType, IntegerType, LongType, DoubleType, DecimalType,
    BooleanType, DateType, TimestampType,
    ArrayType, MapType, VariantType,
    PandasSeries, PandasDataFrame
)

# Snowpark ML
from snowflake.ml.modeling.preprocessing import StandardScaler, OneHotEncoder
from snowflake.ml.modeling.pipeline import Pipeline
from snowflake.ml.modeling.xgboost import XGBClassifier
from snowflake.ml.registry import Registry
```

12.3 Key Interview One-Liners to Remember

- **Snowpark is NOT Spark** — it generates SQL that runs on Snowflake's engine.
- **Transformations are lazy; actions trigger execution (one SQL per action call).**
- **`df.queries` reveals the generated SQL before you execute anything.**
- **Vectorized UDFs use Apache Arrow batches** — orders of magnitude faster than scalar UDFs.
- **`to_pandas()` pulls all data to the client** — avoid on large DataFrames.
- **`execute_as='owner'` encapsulates privileges; `execute_as='caller'` is transparent.**
- **Session inside a stored procedure is auto-injected** — never call `Session.builder` inside a `sproc`.
- **Snowflake's result cache serves as Snowpark's implicit caching layer.**
- **Cluster keys + selective filters = micro-partition pruning = fast queries.**
- **Permanent UDFs require a `stage_location` to store the serialized Python code.**
- **UDTFs expand rows (1 in → N out); use `join_table_function()` to call them.**
- **Snowpark ML trains models inside Snowflake** — data never leaves for training.
- **Model Registry versions and deploys models as Snowflake functions or SPCS services.**

Compiled for Wreetojyoti Ray • Senior Data Engineer / Snowflake Consultant • 2025