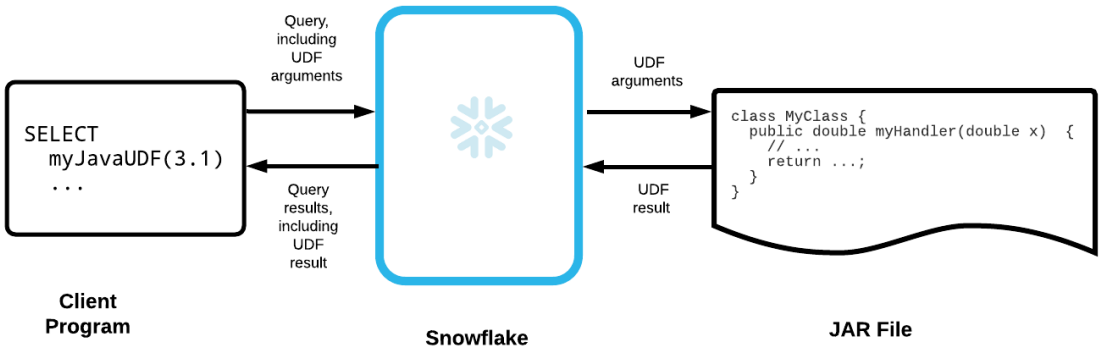
[Snowflake] 1-5. User Defined Functions (UDF)

💡 노션 웹 공유 링크 (댓글 & 상세설명 참고)

> **References**
>
> - **Snowflake Learn (SnowPro PREP-CORE Course) 1장 5강**
> - **Snowflake 설명서 (UDFs)**
> - **Snowflake 설명서 (반정형 데이터 타입)**

- **사용자 정의 함수, User-Defined Functions (UDFs)**

UDF Data Flow



JAVA UDF 데이터 흐름 예시

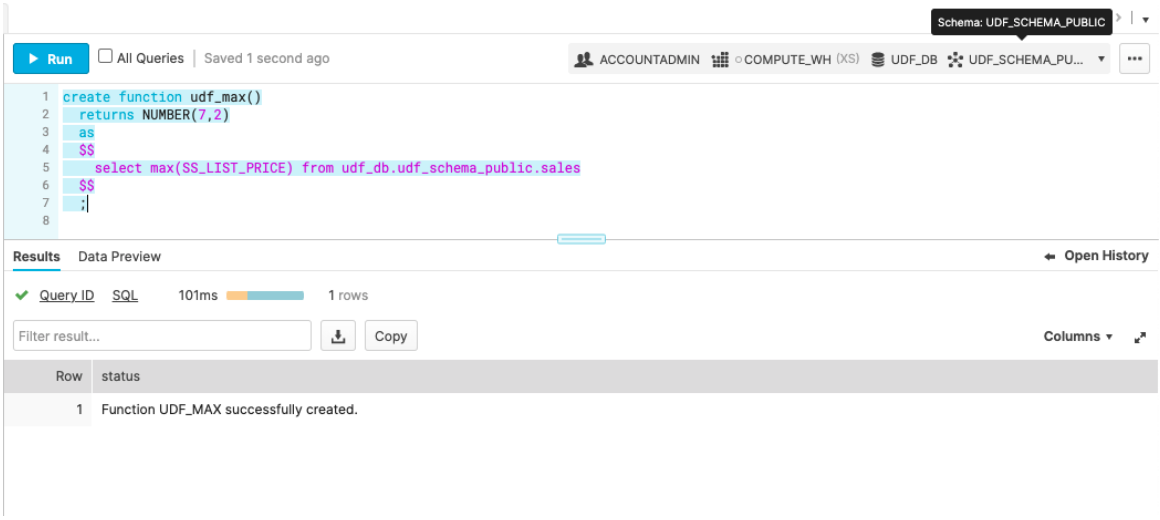
- Snowflake에서는 사용자 정의 함수 (UDFs)를 작성하여 데이터베이스에서 사용자 정의 함수를 정의하고 호출할 수 있음
- 동일 로직을 반복하지 않고, UDF를 통해 코드의 재사용성을 높임
- SQL UDF와 javaScript UDF 를 모두 지원
- Secure, Unsecure 기능
- No DDL/DML 은 지원되지 않음.

- **UDFs 와 저장 프로시저(Stored Procedures)**
  - UDF(User Defined Functions)와 저장 프로시저(Stored Procedure)는 데이터베이스에서 코드를 재사용할 수 있는 방법

UDFs	Stored Procedures
함수는 항상 식을 지정하여 값을 명시적으로 반환하는 호출을 사용 (ex. JavaScript 의 return 문)	일반적으로 관리작업을 수행하며, 저장 프로시저 본문 값을 명시적 반환 가능하지만 필수는 아님
DML 실행할 수 없음	DML 실행할 수 있음 (DDL을 포함한 데이터베이스 관리작업의 경우 프로시저를 생성)
Java, JavaScript, Pyshon SQL	Java, JavaScript, Python, Scala, Snowflake Scripting
UDF 에서 반환된 값은 SQL 문에 직접 사용 가능	저장프로시저에서 반환된 값은 SQL 문에 직접 사용 불가능 (간접 사용)
다른문의 컨텍스트에서 호출 가능 (ex. SELECT {function 명} …~)	독립적 사용 (CALL {Procedure 명})
데이터베이스 작업을 수행할 수 있는 API 에 대한 엑세스 권한이 없음	데이터베이스에 엑세스 가능

- **Secure UDF와 저장프로시저를 통한 보안**
  - UDF 또는 프로시저 정의에서 가시성 제한
    - Secure UDF, 또는 저장 프로시저를 사용하여, 권한이 있는 사용자, 또는 해당 함수를 소유하는 역할이 부여된 사용자에게만 세부 정보들이 보임
    - 제한 되는 명령 : SHOW, DESCRIBE, Information Scheam 뷰
  - UDF의 중요 데이터 가시성 제한
    - UDF의 기본 데이터에 대한 가시성을 제한하려면 데이터 생성 또는 변경 시에 SECURE 키워드를 사용
  - Secure UDF 설정
    - 쿼리 성능과 데이터 보안 간 균형 고려 (쿼리 성능 감소)
    - 민감 데이터의 경우 다른오브젝트 (예: 뷰) 또한 보안 유지방안을 고려하는 것이 좋음
    - secure UDF를 설정하면 최적화 프로그램이 특정 필터를 푸시다운 하지 않아 성능에 영향

- **UDFs 생성**



Snowsight > Worksheets > UDF 생성 예시

  - UDF 생성 명명규칙
    - UDF는 `db . schema . function_name` 형식의 네임스페이스로 정의된 정규화된 이름으로 생성
    - 시스템 정의함수와 동일한 이름을 가진 UDFs는 생성 불가능
    - Snowflake는 SQL UDF 오버로딩을 지원

    - 생성 및 호출 순서

1. 사용자 정의 함수를 작성할 데이터베이스 선택
2. 사용자 정의 함수를 생성
3. 사용자 정의 함수를 호출

```
CREATE OR REPLACE FUNCTION function_name(param1 data_type, param2 data_type)
RETURNS return_data_type
AS
$body$
  // 함수 내용
$body$
LANGUAGE javascript
```

위 코드에서,

- `function_name` : 사용자 정의 함수의 이름
- `param1`, `param2` : 함수의 매개변수
- `data_type` : 매개변수의 데이터 유형
- `return_data_type` : 함수가 반환하는 데이터 유형

UDF를 호출하려면 다음과 같이 작성 가능

```
SELECT function_name(param1, param2);
```

- **UDFs 와 UDTFs**
  - UDTF(User Defined **Table** Functions) 란 행 기반의 함수
  - 0개 이상의 행을 반환
  - SQL UDTF 와 Java UDTF 지원
  - SQL UDTF 기본 형식 : `TABLE (<output_schema>)`
  - 쿼리에서 UDTF를 호출하면 함수의 결과가 행으로 반환됨
  - UDTF 함수가 반환하는 모든 열은 동일한 데이터 유형이어야 함
  - UDTF 는 다음과 같이 생성됩니다

```
CREATE OR REPLACE FUNCTION function_name(param1 data_type, param2 data_type)
RETURNS TABLE (column1 data_type, column2 data_type, ...)
AS
$body$
  // 함수 내용
$body$
LANGUAGE sql
```

  - 위 코드에서,
    - `function_name` : 사용자 정의 함수의 이름
    - `param1`, `param2` : 함수의 매개변수
    - `data_type` : 매개변수의 데이터 유형
    - `(column1 data_type, column2 data_type, ...)` : 반환되는 열의 이름과 데이터 유형

- UDTF를 호출하려면 다음과 같이 작성할 수 있습니다.

```
SELECT * FROM TABLE(function_name(param1, param2));
```

- UDF 와 UDTF 차이점
  - UDTF 는 행을 반환
  - UDF 는 스칼라 값을 반환
  - UDTF 는 `SELECT` 문에서 사용할 수 있음
  - UDF 는 `SELECT`, `WHERE`, `HAVING` 등에서 사용할 수 있음
  - UDTF 는 `CROSS JOIN` 으로 사용 가능
  - UDF 는 `CROSS JOIN` 으로 사용 불가능

- **Snowflake 의 VARIANT 데이터 유형 (JavaScript UDF 활용) (참고)**
  - Snowflake의 `VARIANT` 데이터 유형은 여러 데이터 유형의 조합을 표현할 수 있는 유연한 데이터 유형
  - 이 데이터 유형은 JSON과 유사한 구조를 가지고 있으며, 데이터 유형이 동적으로 결정되는 경우에 유용
  - `VARIANT`는 다른 데이터 유형과 함께 사용될 수 있으며, 예를 들어 테이블에서 열의 데이터 유형으로 사용
  - `VARIANT` 데이터 유형에서 `NULL`은 `VARIANT` 값이 없음을 나타냅니다. 즉, `VARIANT` 값이 `NULL`이면 값이 없음을 의미
    - `VARIANT` 데이터 유형에서 `NULL`과 일반적인 `NULL` 값 사이에는 차이가 있다. 일반적인 `NULL` 값은 데이터 유형을 갖지 않으나, `VARIANT` 데이터 유형에서 `NULL`은 데이터 유형을 갖음
    - SQL Null 과 VARIANT Null

```
CREATE OR REPLACE FUNCTION variant_nulls(V VARIANT)
      RETURNS VARCHAR
      LANGUAGE JAVASCRIPT
      AS '
      if (V === undefined) {
        return "input was SQL null";
      } else if (V === null) {
        return "input was variant null";
      } else {
        return V;
      }
      ';
select null,
       variant_nulls(cast(null as variant)),
       variant_nulls(PARSE_JSON('null'))
       ;
```

cast(null as variant) → SQL NULL / parse_json('null') → VARIANT NULL