데이터 선택과 범위

SELECT 문

- SELECT 문은 종종 FROM, DISTINCT, WHERE, LIMIT 키워드와 함께 사용됨
    - SELECT * 문은 테이블의 모든 칼럼을 선택했음을 의미. 중복된 로우를 포함해 모든 로우를 리턴.
    - DISTINCT 는 테이블의 유일한 값만 리턴됨.
        - select distinct(name) from employee;
        - select count(distinct name) from employee;
    - LIMIT 키워드는 랜덤하게 리턴된 로우의 개수를 제한
    - SELECT <column_name> FROM <table_name> LIMINT n 과 같은 간단한 SELECT 문장은 맵리듀스 작업을 실행하지 않음.
        - set hive.fetch.task.conversion=more 로 설정하면 하이브의 패치 작업을 변환할 수 있음
- 서브쿼리
    - 서브쿼리는 ( ) 안에 써주어야 함.
    - 서브 쿼리를 WHERE 절에 사용할 경우 서브 쿼리는 WHERE 절의 오른쪽 부분에 위치해야 함.
    - 아래 4가지는 모두 같은 결과를 반환
        - 성별이 남자인 직원의 이름과 성별을 반환
- 중첩 SELECT를 CTE를 사용해 구현할 수 있음.
    - 서브쿼리에 alias를 사용해주어야 함.

```
WITH t1 AS(
SELECT * FROM employee
WHERE sex_age.sex='Male')
SELECT name, sex_age.sex AS sex FROM t1;
```

- FROM 문 뒤에 중첩 SELECT를 구현할 수 있음.
    - 서브쿼리에 alias를 사용해주어야 함.

```
SELECT name, sex_age.sex AS sex
FROM
(SELECT * FROM employee
WHERE sex_age.sex='Male')
t1;
```

- IN, NOT IN 서브쿼리
  - 하나의 칼럼만 지원

```
SELECT name, sex_age.sex as sex
FROM employee a
WHERE a.name IN
(SELECT name FROM employee WHERE sex_age.sex='Male');
```

- EXISTS 서브 쿼리
  - 내부와 외부 표현식을 참조해야 함

```
SELECT name, sex_age.sex as sex
FROM employee a
WHERE EXISTS
(SELECT name FROM employee b WHERE b.sex_age.sex='Male' AND
a.sex_age.sex=b.sex_age.sex);
```

JOIN

- JOIN은 2개 이상의 테이블을 수평으로 합치기 위해 사용됨.
- JOIN은 항상 WHERE 보다 앞서 실행됨. 따라서 가능하면 JOIN 뒤에서 결과 집합을 필터링하는 WHERE 절보다 JOIN 조건과 같은 조건문을 앞에 두는 것이 좋음
- JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN, CROSS JOIN과 같은 RDBMS의 JOIN명령을 지원. 하지만 RDBMS와 달리 비등가 JOIN 대신 등가 JOIN만 지원
  - table_m 은 m개의 row, table_n은 n개의 row를 가짐

JOIN 타입

Aa 이름	로직	리턴된 로우
table_m JOIN table_n	두 테이블에서 일치하는 모든 로우를 리턴	m&n
table_m LEFT (OUTER) JOIN table_n	왼쪽 테이블의 모든 로우와 오른쪽 테이블에서 일치하는 로우를 리턴, 오른쪽 테이블에서 일치하는 로우가 없다면 오른쪽 테이블 칼럼에 null을 리턴	m

Aa 이름	로직	리턴된 로우
table_m RIGHT (OUTER) JOIN table_n	오른쪽 테이블의 모든 로우와 왼쪽 테이블에서 일치하는 로우를 리턴, 왼쪽 테이블에서 일치하는 로우가 없다면 왼쪽 테이블 칼럼에 null을 리턴	n
table_m FULL (OUTER) JOIN table_n	두 테이블의 모든 로우와 두 테이블의 일치하는 로우를 리턴. 왼쪽 테이블 또는 오른쪽 테이블에서 일치하는 로우가 없다면 각 칼럼에 null을 리턴	(m+n)-(m&n) = m|n
table_m CROSS JOIN table_n	카테시안 곱(Cartesian product)을 만들기 위해 두 테이블의 모든 로우를 조합한 후, 조합한 로우를 리턴	mxn

- 여러 테이블 간의 조인을 수행할 때, HDFS의 데이터를 처리하기 위해 맵리듀스 작업이 생성됨. 각 작업은 stage(스테이지)라 불린다.
  - 종종 더 좋은 성능을 내고, 메모리 부족(OOM, Out Of Memory)를 피하기 위해 끝에 큰 테이블을 JOIN 문 오른쪽에 두는 것을 추천. JOIN문의 테이블이 리듀서의 버퍼에 쌓이는데, 여러 JOIN문의 마지막 테이블이 리듀서를 통해 스트림되기 때문
  - /*+STREAMTABLE (<table name>) */ 과 같은 힌트를 명세해 어느 테이블이 스트림될지 명세할 수도 있음

```
# 스트림될 테이블을 명세
SELECT /*+ STREAMTABLE (employee_hr) */
emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name
JOIN employee_id empi ON emph.employee_id=empi.employee_id;
```

INNER JOIN

- 왼쪽과 오른쪽 테이블에서 JOIN 조건에 부합하는 로우를 리턴하는 JOIN 키워드를 사용
- 콤마로 구분된 테이블을 사용하면 INNER JOIN 키워드를 생략할 수 있음
- windows에 저장된 데이터를 linux로 전송

```
scp C:/Users/이혜림/Desktop/Hadoop/하이브/code/employee_hr.txt hduser@127.0.0.1:/usr/local/hadoop
```

- employee_hr 테이블 생성

```
CREATE TABLE IF NOT EXISTS employee_hr(
name string,
employee_id int,
sin_number string,
start_date string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;
```

- 데이터 로드

```
LOAD DATA LOCAL INPATH '/usr/local/hadoop/employee_hr.txt'
OVERWRITE INTO TABLE employee_hr;
```

- 등가 JOIN으로 employee와 employee_hr 테이블을 조인

```
SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name;
```

```
# 남자 직원에 대하여 join

# JOIN 보다 앞서 실행되는 조건문
SELECT emp.name as name, emp.sex_age.sex as sex, emph.sin_number
FROM (select name, sex_age from employee WHERE sex_age.sex='Male') emp
JOIN employee_hr emph ON emp.name=emph.name;

# JOIN 이후에 실행되는 조건문
SELECT emp.name as name, emp.sex_age.sex as sex, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name
WHERE emp.sex_age.sex='Male';
```

- 여러 테이블 간의 JOIN을 사용할 수도 있음

```
SELECT emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name
JOIN employee_id empi ON emp.name=empi.name;
```

- 셀프 조인(self-join)은 한 테이블이 자신과 조인하는 특수한 JOIN.
  - 셀프 조인을 수행할 때, 동일 테이블과 구별하기 위해 여러 alias를 사용

```
SELECT emp.name
FROM employee emp
JOIN employee emp2 ON emp.name=emp2.name;
```

- 암묵적 조인
  - JOIN 키워드가 없는 JOIN 명령

```
SELECT emp.name, emph.sin_number
FROM employee emp, employee_hr emph
WHERE emp.name=emph.name;
```

- JOIN 명령은 조인 조건에 여러 칼럼을 사용하고 추가적인 맵리듀스 작업을 생성

```
SELECT emp.name, empi.employee_id, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name
JOIN employee_id empi ON emph.employee_id=empi.employee_id;
```

OUTER JOIN, CROSS JOIN

- LEFT OUTER JOIN

```
SELECT emp.name, emph.sin_number
FROM employee emp
LEFT OUTER JOIN employee_hr emph ON emp.name=emph.name;
```

- RIGHT OUTER JOIN

```
SELECT emp.name, emph.sin_number
FROM employee emp
RIGHT JOIN employee_hr emph ON emp.name=emph.name;
```

- FULL OUTER JOIN

```
SELECT emp.name, emph.sin_number
FROM employee emp
FULL JOIN employee_hr emph ON emp.name=emph.name;
```

- CROSS JOIN
  - 방법1. CROSS JOIN 명시
  - 방법2. CROSS JOIN 명시 X
    - CROSS JOIN을 사용하지 않고, 조건문 (ON)을 사용하지 않거나, 1=1과 같이 항상 TRUE가 되는 조건을 사용함으로써 CROSS JOIN을 할 수도 있음

```
# 동일한 결과를 출력하는 세가지 방법

# 방법1
SELECT emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph;

# 방법2
SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph;

# 방법3
SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON 1=1;
```

📝 디버깅. 만약 CROSS JOIN에서 다음과 같은 오류가 발생한다면

————————————오류——-
SemanticException Cartesian products are disabled for safety reasons. If you know what you are doing, please sethive.strict.checks.cartesian.product to false and that hive.mapred.mode is not set to 'strict' to proceed. Note that if you may get errors or incorrect results if you make a mistake while using some of the unsafe features.
——

1. set hive.mapred.mode; 로 현재 모드 확인
2. set hive.mapred.moce=nonstrict; 로 변경

- 비등가 JOIN 수행 방법
  - 비등가 작업을 명시적으로 지원하지 않음
  - CROSS JOIN과 WHERE 조건 사용
  - <> 와 != 은 원시타입에 대해서만 동작(null 은 원시타입이 아님)
  - 따라서 not null인 것을 추출하기 위해서는 is not null을 사용해야 함

```
# 오류 발생
SELECT emp.name, emph.sin_number
FROM employee emp
JOIN employee_hr emph ON emp.name <> emph.name;
```

```
# CROSS JOIN과 WHERE 조건 사용
SELECT emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph
WHERE emp.name<>emph.name;
```

특수 조인 : MAPJOIN

- MAPJOIN 문은 리듀스 작업없이 맵 작업에서 JOIN 작업을 처리
  - shuffle join
    - mapper에서 각 테이블을 읽고 shuffle 단계에서 조인되는 키를 기준으로 파티셔닝 후 셔플을 진행하여 각 리듀서에서 조인
    - 자원을 많이 사용하고 느림
  - map join
    - map 단계에서 join 작업을 처리
    - 작은 테이블을 메모리에 올리고, 큰 테이블의 각 mapper에서 JOIN 실행
    - 리듀스가 필요없기 때문에 JOIN 성능이 개선됨.

- 작은 테이블이 메모리에 들어갈 정도로 작아야 함
- hive.auto.convert.join 설정이 true일 때, 하이브는 맵 조인의 힌트를 확인하는 대신, 가능하면 런타임 시 JOIN을 MAPJOIN으로 자동 변환
- /*+ MAPJOIN (<smaller table>) */ 로 실행 가능
- MAPJOIN이 지원하지 않는 기능
  - UNION ALL, LATERAL VIEW, GROUP BY/JOIN/SORT BY/CLUSTER BY/DISTRIBUTE BY 뒤에서 MAPJOIN에 사용
  - UNION, JOIN, 다른 MAPJOIN 앞에서 MAPJOIN에 사용하기

```
SELECT /*+ MAPJOIN (employee) */
emp.name, emph.sin_number
FROM employee emp
CROSS JOIN employee_hr emph WHERE emp.name=emph.name;
```

- Bucket Map Join
  - 테이블이 bucketing 되어있을 때 사용할 수 있는 MAPJOIN의 특별한 타입
  - 일반 MAPJOIN 처럼 테이블의 전체 로우를 얻는 대신, 버킷 맵은 필요한 버킷 데이터만 맵 조인을 수행.
  - 예를 들어 table1, table2가 있고, 각 버킷 개수가 8, 4개라면, table1의 bucket1, 4와 table2의 bucket1은 같은 map에서 join이 실행됨
  - 버킷 맵을 하기 위해선
    1. 버킷 개수가 배수인지 확인
    2. 설정 추가
       - SET hive.optimize bucketmapjoin=true;
       - SET hive.optimize bucketmpajoin.sortedmerge=true;
       - SET hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
- LEFT SEMI JOIN
  - MAPJOIN 타입 중 하나
  - 오른쪽 테이블은 WHERE 또는 SELECT 절이 아닌 조인 조건에서만 참조해야 함.

```
SELECT a.name
FROM employee a
LEFT SEMI JOIN employee_id b
ON a.name=b.name;
```

```
# 오른쪽 테이블을 SELECT 절에서 참조했기 때문에 오류 발생
# 오른쪽 테이블은 JOIN 절에서만 참조되어야 함.
SELECT a.name, b.employee_id
FROM employee a
LEFT SEMI JOIN employee_id b
ON a.name=b.name;
```

UNION/INTERSECT/MINUS

- 두 테이블을 수직으로 이어붙임
  - UNION ALL : 중복에 상관없이 모두 이어붙임
  - UNION : 합집합
  - INTERSECT : 교집합
  - MINUS : 차집합. 먼저 기술한 table - 나중에 기술한 table
- ALL이 붙으면 결과에 중복이 발생하더라도 그대로 중복을 유지
- 서브 쿼리 및 상위 쿼리 모두에서 사용 가능
- UNION ALL

```
SELECT emp.name
FROM employee emp
UNION ALL
SELECT emph.name
FROM employee_hr emph;
```

- UNION

```
SELECT emp.name
FROM employee emp
UNION
SELECT emph.name
FROM employee_hr emph;
```

```
# UNION ALL을 이용하여 UNION과 같은 결과 반환
SELECT DISTINCT(name)
FROM (SELECT emp.name FROM employee emp
UNION ALL
SELECT emph.name FROM employee_hr emph) empp;
```

- INTERSECT

```
SELECT emp.name
FROM employee emp
INTERSECT
```

```
SELECT emph.name
FROM employee_hr emph;
```

```
# JOIN을 이용하여 INTERSECT와 같은 결과 반환
SELECT emp.name FROM employee emp
JOIN employee_hr emph ON emp.name=emph.name;
```

- MINUS

```
SELECT emp.name
FROM employee emp
MINUS
SELECT emph.name
FROM employee_hr emph;
```

```
SELECT emph.name
FROM employee_hr emph
MINUS
SELECT emp.name
FROM employee emp;
```

```
SELECT emp.name
FROM employee emp
LEFT JOIN employee_hr emph ON emp.name=emph.name
WHERE emph.name IS NULL;
```