데이터 조작

데이터 로드

LOAD

- LOAD DATA ~ OVERWRITE INTO TABLE
    - LOCAL INPATH : 장비에서 파일이 어디 위치하는지를 명세
    - INPATH : fs.default.name 에 설정된 전체 URI(Uniform Resource Identifier)를 로드

```
# local data를 hive table로 로드
LOAD DATA LOCAL INPATH '/usr/local/hadoop/employee.txt'
OVERWRITE INTO TABLE employee;
```

```
# local data를 hive partition으로 로드
LOAD DATA LOCAL INPATH '/usr/local/hadoop/employee.txt'
OVERWRITE INTO TABLE employee
PARTITION (year=2014, month=12);
```

```
# 기본 시스템 경로를 사용해 HDFS 데이터를 hive table로 로드
LOAD DATA INPATH '/user/sample/employee.txt'
OVERWRITE INTO TABLE employee;
```

```
# HDFS 데이터를 전체 URI가 있는 하이브 테이블로 로드
LOCAL DATA INIPATH 'hdfs://localhost:9020/user/sample/employee.txt'
OVERWRITE INTO TABLE employee;
```

INSERT

- 하이브 테이블이나 파티션에서 INSERT를 이용하여 데이터를 추출.
- RDBMS의 INSERT에서 다중 INSERT, 동적 파티션 INSERT OVERWRITE 를 지원함으로써 INSERT문을 개선
- INSERT 종류
  1. INSERT INTO TABLE : 기존 데이터를 덮어쓰지 않고 데이터를 입력(adding data)
  2. INSERT OVERWRITE : 데이터를 입력할 때 기존 데이터를 덮어씀(replacing data)
- 일반 INSERT

```
INSERT INTO TABLE employee
SELECT * FROM employee;
```

- CTE문에서의 INSERT

```
WITH a AS (SELECT name FROM employee_hr)
INSERT OVERWRITE TABLE cte_employee
SELECT * FROM a;
```

- 다중 INSERT

```
FROM employee
INSERT OVERWRITE TABLE cte_employee
SELECT *
INSERT OVERWRITE TABLE employee_2
SELECT *;
```

LOCAL/HDFS FILE에 로드

INSERT

- HIVE 테이블에 로드하는 것과 달리 local/hdfs file에 로드할 경우 OVERWRITE만 사용 가능. 즉, 해당 쿼리 시에 디렉토리안의 파일이 모두 삭제되기 때문에 신중하게 사용해야 함
- 쿼리
  - LOCAL에 로드 : INSERT OVERWRITE LOCAL DIRECTORY <저장할 폴더>
  - HDFS에 로드 : INSERT OVERWRITE DIRECTORY <저장할 폴더>

```
# LOCAL에 로드
INSERT OVERWRITE LOCAL DIRECTORY '/usr/local/hadoop'
SELECT name FROM employee;

# HDFS에 로드
INSERT OVERWRITE DIRECTORY '/sample'
SELECT name FROM employee;
```

```
# 다중 INSERT 문
FROM employee
INSERT OVERWRITE DIRECTORY '/sample'
SELECT *
INSERT OVERWRITE DIRECTORY '/sample2'
SELECT *
```

- 구분자
  - default : FIELDS TERMINATED BY '^A', LINES TERMINATED BY '\n'
  - 원하는 로우 구분자 명세 가능

```
# 명세한 로우 구분자로 로컬 파일에 추가
INSERT OVERWRITE DIRECTORY '/sample'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
SELECT * FROM employee_hr;
```

- 로컬 디렉토리 경로를 하나의 디렉토리에서만 작업할 수 있음. 여러 디렉토리 레벨에서 작업하고 싶다면
  - SET hive.insert.into.multilevel.dirs=true
- INSERT 문을 실행할 때, 리듀서는 많은 조각난 파일을 기본적으로 생성. 조각난 파일을 하나로 병합하기 위해선
  - hdfs dfs -getmerge hdfs://<host_name>:8020/sample

하이브와 HDFS 셸 커맨드

- hive -e '<쿼리문>' 또는 hive -f <hql_filename> 으로 하이브 쿼리문 또는 쿼리 파일 실행 가능. insert와 다르게 파일이 저장될 디렉토리가 아닌 파일을 적어주어야 함.
- 로컬 파일에 추가
  - hive -e '<쿼리문>' >> <저장할 파일>

```
$ hive -e 'select * from employee' >> /usr/local/hadoop/sample
```

- 로컬 파일에 덮어쓰기
  - hive -e '<쿼리문>' > <저장할 파일>

```
$ hive -e 'select * from employee' > sample
```

- HDFS 파일에 추가하기
  - hive -e '<쿼리문>'|hdfs dfs -appendToFile - <저장할 파일>

```
$ hive -e 'select * from employee'|hdfs dfs -appendToFile - /sample/sample2
```

- HDFS 파일에 덮어쓰기
  - hive -e '<쿼리문>'|hdfs dfs -put -f - <저장할 파일>

```
$ hive -e 'select * from employee'|hdfs dfs -put -f - /sample/sample2
```

동적 파티션(다이내믹 파티션)

- 데이터의 크기가 크고, 파티션 칼럼 값을 모를 때 다이내믹 파티션 기능을 사용하면 유용

- load data에서는 사용 불가능. insert 구문에서만(hive에 저장된 데이터에 한에서만) 사용 가능
- 파티션 칼럼명만 설정하고 칼럼값은 설정하지 않으면 다이내믹 파티션 기능을 사용할 수 있음.
- 하이브는 사용자의 실수로 너무 많은 파티션이 생성되는 것에 대비하기 위해 다이내믹 파티션을 비활성화. 따라서 다이나믹 파티션 설정 활성화 이후 가능

다이내믹 파티션 속성

Aa 속성	기본값	내용
hive.exec.dynamic.partition	false	다이내믹 파티션 사용 여부를 설정
hive.exec.dynamic.partition.mode	strict	모든 파티션을 동적으로 설정하고 싶다면 nonstrict으로 설정(원래는 적어도 하나의 정적 파티션 칼럼을 명세해야 함)
hive.exec.max.dynamic.partitions.permode	100	각 매퍼 및 리듀서가 생성할 수 있는 최대 다이내믹 파티션 개수. 예를 들어, 하나의 질의에 매퍼 두 개가 실행 될 경우 각 매퍼는 100개의 파티션을 생성할 수 잇으며, 해당 질의에서는 최대 200개의 파티션이 생성될 수 있음
hive.exec.max.dynamic.partitions	1000	하나의 절에서 만들어질 수 있는 최대 다이내믹 파티션 개수
hive.exec.max.created.files	100000	하나의 질의에서 만들어질 수 있는 최대 다이내믹 파티션 개수

```
# 다이나믹 파티션 설정 활성화
hive> set hive.exec.dynamic.partition.mode=nonstrict; # 모든 파티션을 동적으로 설정
hive> set hive.exec.dynamic.partition=true; # 동적 파티션 기능 활성화
```

```
# 다이나믹 파티션 예1
INSERT OVERWRITE TABLE employee_partitioned
PARTITION (year, month)
SELECT name, array('Toronto') as work_place,
named_struct("sex","Male","age",30) as sex_age,
map("Python",90) as skills_score,
map("R&D",array('Developer')) as depart_title,
year(start_date) as year, month(start_date) as month
FROM employee_hr
WHERE employee_hr.employee_id=102;
```

```
# 다이나믹 파티션 예2

# 파티션 테이블 생성
```

```
CREATE TABLE employee_id_partitioned(
name string)
PARTITIONED BY (employee_id)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;

# 다이나믹 파티션 할당  ->  총 25개의 파티션이 생성됨
INSERT OVERWRITE TABLE employee_id_partitioned
PARTITION (employee_id)
SELECT name, employee_id
FROM employee_hr;
```

데이터 교환

EXPORT

- 테이블 또는 파티션에서 데이터와 메타데이터를 HDFS로 export
  - 데이터 : data라는 하위 디렉토리로 export
  - 메타데이터 : _metadata 파일명으로 export
- export 문을 실행한 후에 다른 하이브 인스턴스에서 exported file을 수동으로 복사할 수도 있고, 다른 HDFS 클러스터에 복사하기 위해 distcp 커맨드를 사용할 수도 있음.
- 쿼리
  - EXPORT TABLE <테이블명> TO '<디렉토리명>
  - 이때, 디렉토리는 빈 디렉토리여야 함.

```
EXPORT TABLE employee TO '/sample2';ㄷ
```

- 파티션 export

```
EXPORT TABLE employee_partitioned
PARTITION (year=2014, month=11)
TO '/sample4';
```

IMPORT

- 테이블 또는 파티션에서 데이터와 메타데이터를 HDFS에서 import
  - 이때, 동일한 이름을 가진 테이블이 존재하면 에러를 출력

```
IMPORT FROM '/sample2';
```

- 새로운 테이블로 데이터를 import

```
IMPORT TABLE employee_imported
FROM '/sample2';
```

- 외부 테이블로 import
  - LOCATION 뒤에 저장할 HDFS 경로 명세해줌

```
IMPORT EXTERNAL TABLE employee_imported_external
FROM '/sample2'
LOCATION '/sample3';
```

- 파티션 import

```
IMPORT TABLE employee_imported_partitioned
FROM '/sample4';
```

데이터 정렬

- 정렬 데이터는 SELECT 문 이후 실행됨
  - FROM → JOIN → WHERE → SELECT → ORDER BY
- default : 오름차순

ORDER BY

- 질의문 결과에 대해 전체 정렬을 수행하며, 이를 위해 마지막 맵리듀스 잡에서 하나의 리듀서만 실행
- 따라서 정렬 대상 데이터가 클 경우 성능이 느려지는 단점이 존재하지만, 전체 정렬이 가능하다는 장점이 존재
- SELECT 뒤에 실행

```
SELECT employee_id FROM employee_id
ORDER BY employee_id DESC;
```

SORT BY

- 전체 정렬을 포기하고 부분 데이터에 대해 정렬함으로써 쿼리 성능을 높임.
- ORDER BY 절과는 다르게 여러 개의 리듀서를 실행하고, 리듀서 별로 출력 결과를 정렬

    - 만약 mapred.reduce.tasks=1 로 설정되어 있다면 전체 정렬이 됨.
- 이 방법은 한 리듀서가 같은 키만 받는 것을 보장하지 않기 때문에 각 리듀서의 출력 결과에 동일한 키가 생성될 수 있음

```
SELECT employee_id FROM employee_id
SORT BY employee_id DESC;
```

DISTRIBUTE BY

- 같은 칼럼값을 갖는 레코드를 같은 리듀서로 보냄. 즉, RDBMS의 group by와 비슷한 기능.
- SORT BY와 함께 사용할 경우, 각 리듀서는 키가 중복되지 않고 정렬을 수행할 수 있음

```
# DISTRIBUTED BY
SELECT name, employee_id, work_place[0] as first_work_place
FROM employee_id
DISTRIBUTE BY first_work_place;

# DISTRIBUTED BY & SORT BY
SELECT name, employee_id, work_place[0] as first_work_place
FROM employee_id
DISTRIBUTE BY first_work_place SORT BY first_work_place;
```

CLUSTER BY

- CLUSTER BY는 DISTRIBUTE BY와 SORT BY를 함께 사용하는 것과 동일한 기능
- 전역 정렬을 실행할 때 사용할 수 있는 모든 리듀서를 작동시키려면 CLUSTER BY를 먼저 실행하고 뒤에 ORDER BY를 사용.

```
SELECT name, employee_id
FROM employee_id
CLUSTER BY employee_id
```

명령어와 함수

- 하이브 함수 분류
    - 수학 함수 : RAND( )와 E( )와 같은 수학적 계산을 주로 수행
    - 컬렉션 함수 : 크기, 키, SIZE(Array<T>)와 같은 복잡한 타입의 값을 찾음
    - 타입 변환 함수 : 특정 타입에서 다른 타입으로 변환하기 위해 CAST와 BINARY함수를 주로 사용
    - 날짜 함수 : YEAR(string date)와 MONTH(string date)와 같은 날짜 관련 계산을 실행

    - 조건 함수 : COALESCE, IF, CASE WHEN과 같은 특정 조건을 확인
    - 문자열 함수 : UPPER(string A)이나 TRIM(string A)같은 문자열 관련 명령어를 실행
    - 집계 함수 : SUM( ), COUNT(*)와 같은 집계를 수행
    - 테이블 생성 함수 : EXPLORE(MAP) 이나 JSON_TUPLE(jsonString, k1,k2, ...) 처럼 하나의 입력 로우를 다중 출력 로우로 변환
    - 사용자 정의 함수 : 하이브를 확장하기 위해 자바 코드로 생성됨
- 하이브의 내장 함수/UDF 목록을 보는 방법

```
# 모든 함수를 출력
SHOW FUNCTIONS;

# 특정 함수에 대해 자세히 살펴봄
DESCRIBE FUNCTION <function_name>;

# 더 자세히 함수를 살펴봄
DESCRIBE FUNCTION EXTENDED <function_name>;
```

하이브의 내장 함수

Aa 함수	내용
concat(칼럼a, 칼럼b)	각 칼럼a의 값 뒤에 각 칼럼b의 값을 붙여서 반환. 예를 들어 칼럼a의 1행이 facebook이고, 칼럼b의 1행이 hive라면 facebookhive를 반환
substr(칼럼a, int start_index)	칼럼a의 start_index에서 마지막 인덱스까지 잘라낸 문자열을 반환. 예를 들어,칼럼a의 어떤 행이 hadoop이고 start_index가 4라면 oop를 반환. 즉, index는 1부터 시작
substr(칼럼a, int start_index, int length)	칼럼a의 start_index에서 설정한 length만큼 잘라낸 문자열을 반환. 예를 들어,칼럼a의 어떤 행이 hadoop이고 start_index가 4, length가 2라면 oo를 반환. 즉, index는 1부터 시작
upper(칼럼)	문자열을 대문자로 변환해서 반환. 예를 들어 칼럼의 값이 hive라면 HIVE를 반환
ucase(칼럼)	upper 함수와 동일
lower(칼럼)	문자열을 소문자로 변환해서 반환. 예를 들어 칼럼의 값이 HIVE라면 hive를 반환
lcase(칼럼)	lower 함수와 동일
trim(칼럼)	문자열 양쪽에 있는 공백을 제거. 예를 들어 칼럼의 값이 ' hive ' 라면 'hive'반환
ltrim(칼럼)	문자열 왼쪽에 있는 공백을 제거. 예를 들어 칼럼의 값이 ' hive ' 라면 'hive '반환
rtirm(칼럼)	문자열 양쪽에 있는 공백을 제거. 예를 들어 칼럼의 값이 ' hive ' 라면 ' hive'반환
regexp_replace(칼럼, string regex, string replacement)	문자열 str의 정규 표현식 regex와 일치하는 모든 문자열을 replacement로 변경해서 반환. 예를 들어 칼럼의 어떤 값이 'hadoop'이고 regex가 'op', replacement가 ''라면, 'had'반환

Aa 함수	내용
from_unixtime(칼럼)	유닉스 시간 문자열(1970-01-01 00:00:00 UTC)를 현재 시스템의 시간대로 변경해서 반환
to_date(칼럼)	타임스태프 문자열에서 시간을 제외한 날짜값만 반환. 예를 들어, "2012-09-01 00:00:00"은 "2012-09-01"을 반환
round(칼럼)	값에 대한 반올림 정수값(BIGINT)을 반환
floor(칼럼)	값보다 작거나 같은 최대 정수값(BIGINT)를 반환
ceil(칼럼)	값보다 크거나 같은 최소 정수값(BIGINT) 반환
rand(). rand(int seed)	랜덤값을 반환. seed 파라미터로 랜덤값의 범위를 설정할 수 있음
year(칼럼)	날짜 혹은 타임스태프 문자열에서 연도만 반환. 예를 들어, "2012-09-11 00:00:00"은 "2012"를 반환
month(칼럼)	날짜 혹은 타임스태프 문자열에서 월만 반환. 예를 들어, "2012-09-11 00:00:00"은 "09"를 반환
day(칼럼)	날짜 혹은 타임스태프 문자열에서 일만 반환. 예를 들어, "2012-09-11 00:00:00"은 "11"를 반환
get_json_object(string json_string, string path)	디렉터리 path에서 문자열 json_string으로부터 json 객체를 추출하고 json 문자열로 반환. 만약 json이 유효하지 않으면 null 반환
size(MAP<K.V>)	맵 타입의 엘리먼트 개수를 반환
size(Array<t>)	배열 타입의 엘리먼트 개수를 반환
cast(칼럼 as<type>)	정규 표현식 expr을 type으로 타입을 변환. 예를 들어, cast('100' as BIGINT)는 '100'을 BIGINT로 변환해서 반환. 변환에 실패할 경우 null 값 반환

SIZE

- MAP, ARRAY, 중첩 MAP/ARRAY 의 크기를 계산하는 데 사용됨
- 크기를 알 수 없으면 -1을 리턴

```
SELECT size(work_place) as array_size,
size(skills_score) as map_size,
size(depart_title) as complex_title,
size(depart_title['Product']) as nest_size
FROM employee;
```

ARRAY_CONTAINS, SORT ARRAY

- ARRAY_CONTAINS
  - 배열이 특정 값을 갖고 있는지 확인해 TRUE/FALSE를 리턴
- SORT ARRAY

- SORT_ARRAY는 배열을 알파벳 오름차순으로 정렬

```
SELECT ARRAY_CONTAINS(work_place, 'Toronto') as is_Toronto,
SORT_ARRAY(work_place) as sorted_array
FROM employee;
```

날짜 함수 팁

FROM_UNIXTIME(UNIX_TIMESTAMP())

- FROM_UNIXTIME(UNIX_TIMESTAMP()) 문은 오라클의 SYSDATE와 같은 함수를 실행
- 하이브 서버에서 현재 날짜와 시간을 동적으로 리턴

```
SELECT FROM_UNIXTIME(UNIX_TIMESTAMP()) as current_time;
```

UNIX_TIMESTAMP()

- 두 날짜를 비교하거나, ORDER BY 뒤에 두어 ORDER BY UNIX_TIMESTAMP(string_date, 'dd-MM-yyyy'와 같은 날짜 값의 여러 문자열 타입을 정렬하기 위해 사용
- 계산 시 초 단위로 나오기 때문에 일 단위로 보기 위해선 60x60x24를 나누어 주어야 함

```
SELECT (UNIX_TIMESTAMP('2021-08-05 16:52:00') - UNIX_TIMESTAMP('2020-06-07 16:00:00'))/60/60/24
AS daydiff;
```

TO_DATE

- 날짜에서 시간, 분, 초를 삭제
- WHERE TO_DATE(update_datetime) BETWEEN '2014-11-01' AND '2014-11-31' 와 같은 데이터 범위에서 칼럼 값이 데이터-날짜 타입인지 확인해야 할 때 유용

```
SELECT TO_DATE(FROM_UNIXTIME(UNIX_TIMESTAMP())) as current_tim
```

여러 데이터 타입에 대한 CASE

```
SELECT
CASE WHEN 1 IS NULL THEN 'TRUE' ELSE 0 END
AS case_result;
```

파서와 검색 팁

- LATER VIEW와 EXPLODE를 ARRAY, MAP 타입에 함께 사용함으로써, 칼럼의 ARRAY / MAP을 펼쳐서 볼 수 있음.
  - LATER VIEW
    - 가상의 뷰를 만듦
  - EXPLODE()
    - 칼럼의 MAP / ARRAY 타입을 평평하게 함
    - e.g. ['hi','hello'] → 'hi' 'hello'
    - explode()에 alias를 명세해주어야 함
- 예

sample data

pageid	adid_list
front_page	[1, 2, 3]
contact_page	[3, 4, 5]

sample result

pageid (string)	adid (int)
"front_page"	1
"front_page"	2
"front_page"	3
"contact_page"	3
"contact_page"	4
"contact_page"	5

```
# table에 row를 추가
INSERT INTO TABLE employee
SELECT 'Steven' as name, array(null) as work_place,
named_struct("sex","Male","age",30) as sex_age,
map("Python", 90) as skills_score,
map("R&D", array("Developer")) as depart_title
FROM employee LIMIT 1;
```

```
# EXPLODE가 NULL을 리턴할 때, LATERAL VIEW는 NULL을 가진 로우를 무시
SELECT name, workplace,skills,score
FROM employee
LATERAL VIEW explode(work_place) wp as workplace
LATERAL VIEW explode(skills_score) ss as skills, score;
```

```
# EXPLODE가 NULL을 리턴할 때에도 OUTER LATERAL VIEW는 로우를 유지
SELECT name, workplace,skills,score
FROM employee
LATERAL VIEW OUTER explode(work_place) wp as workplace
LATERAL VIEW explode(skills_score) ss as skills, score;
```

- REVERSE()
  - 문자열의 모든 글자를 역순으로 변경
  - 문자열외에 다른 타입에는 사용할 수 없음

```
SELECT REVERSE('haha');
```

- SPLIT()
  - 특정 구분자를 사용해 문자열을 구분할 수 있음
  - SPLIT(<문자열>,<구분자>)

```
SELECT SPLIT('haha,hoho',',');
```

```
SELECT REVERSE(SPLIT(REVERSE('haha,hoho'),',')[0]);
# hoho가 출력됨
```

- COLLECT_SET(), COLLECT_LIST()
  - 모든 로우의 엘리먼트를 array 형태로 리턴
  - null은 리턴하지 않음
  - COLLECT_SET() : 중복을 삭제
  - COLLECT_LIST() : 중복을 삭제X

```
SELECT collect_set(work_place[0])
as flat_workplace0 FROM employee;
```

```
SELECT collect_set(skills_score)
as flat_workplace0 FROM employee;
```

```
SELECT collect_list(work_place[0])
as flat_workplace0 FROM employee;
```

가상 칼럼

- 하이브의 두 개의 가상 칼럼
  - INPUT__FILE__NAME : 매퍼 작업을 위한 입력 파일의 이름
  - BLOCK__OFFSET__INSIDE__FILE : 현재 전역 파일 위치 또는 현재 블록의 파일 오프셋

file position, file offset
- file position : 파일 내용을 다룰 때, 어떤 operation을 수행하는 위치.
ex) 현재 반짝이는 커서가 어디에 있는지
- file offset : 파일의 시작지점부터 현재 커서의 위치까지 얼마나 떨어져 있는지 정수로 표현한 것

```
SELECT input__file__name, block__offset__inside__file as offside
FROM employee_id_buckets;
```

```
SELECT input__file__name, block__offset__inside__file as offside
FROM employee_id_partitioned;
```

isnull(), isnotnull()

- null값인지, null값이 아닌지 확인하여 true/false를 반환

```
SELECT work_place, isnull(work_place[0]) as is_null,
isnotnull(work_place[0]) as is_not_null
FROM employee;
```

assert_true()

- 조건이 true가 아니면, 예외를 던짐

```
SELECT assert_true(work_place[0] is null)
FROM employee;
```

elt(n, str1, str2, ...)

- n번째 문자열을 리턴
- n은 1부터 시작. 따라서 elt(2, "a", "b", "c") → "b"를 반환

```
SELECT elt(2, "haha","hoho","kiki");
```

current_database();

- 현재 데이터베이스 이름을 반환

```
SELECT current_database();
```

트랜잭션(transaction)

📝 **트랜잭션(Transaction)이란?**
- 트랜잭션은 데이터베이스의 상태를 변화시키기 위해 수행하는 작업의 단위. 이때, 작업의 단위는 여러 질의어 명령문을 기준에 따라 정함.

- 트랜잭션은 ACID의 특징을 가짐
- ACID : Atomicity(원자성), Consistency(일관성), Isolation(격리), Durability(지속성)

- commit : 하나의 트랜잭션이 성공적으로 끝났고, 데이터베이스가 일관성있는 상태에 있을 때, 하나의 트랜잭션이 끝났다는 것을 알려주기 위해 사용하는 연산. 이 연산을 통해 수행했던 트랜잭션이 로그에 저장됨

- rollback : 하나의 트랜잭션 처리가 비정상적으로 종료되어 트랜잭션의 원자성이 깨진 경우, 트랜잭션을 처음부터 다시 시작하거나, 트랜잭션의 부분적으로만 연산된 결과를 다시 취소시킴. commit을 하면 트랜잭션이 처리된 단위대로 rollback하기 용이

- 하이브는 ACID를 제공하는 로우 레벨 트랜잭션을 완벽히 지원

- 하이브의 현재 모든 트랜잭션은 오토커밋이며, ORC(Optimized Row Columnar) 파일 데이터 또는 버킷 테이블의 데이터만 지원
- 하이브에서 트랜잭션을 사용하기 위한 환경설정

```
SET hive.support.concurrency=true;
SET hive.enforce.bucketing=true;
SET hive.exec.dynamic.partition.mode=nonstrict;
SET hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
SET hive.compactor.initiator.on=true;
SET hive.compactor.worker.threads=1;
```

- 현재 열려있거나 취소된 트랜잭션을 보여주는 커맨드

```
SHOW TRANSACTIONS;
```

- INSERT VALUE, UPDATE, DELETE 커맨드로 로우를 운영

```
# INSERT
INSERT INTO TABLE <table_name> [PARTITION (<partition_column>[=val]...)]
VALUES <values_row> [, <values_row>...];

# UPDATE
UPDATE <table_name> SET column = value [, column = value ...]
[WHERE <expression>]

# DELETE
DELETE FROM <table_name> [WHERE <expression>]
```